CHAPTER 15

Unit and Functional Testing

Automated tests are one of the greatest advances in programming since object orientation. Particularly conducive to developing web applications, they can guarantee the quality of an application even if releases are numerous. Symfony provides a variety of tools for facilitating automated testing, and these are introduced in this chapter.

Automated Tests

Any developer with experience developing web applications is well aware of the time it takes to do testing well. Writing test cases, running them, and analyzing the results is a tedious job. In addition, the requirements of web applications tend to change constantly, which leads to an ongoing stream of releases and a continuing need for code refactoring. In this context, new errors are likely to regularly crop up.

That's why automated tests are a suggested, if not required, part of a successful development environment. A set of test cases can guarantee that an application actually does what it is supposed to do. Even if the internals are often reworked, the automated tests prevent accidental regressions. Additionally, they compel developers to write tests in a standardized, rigid format capable of being understood by a testing framework.

Automated tests can sometimes replace developer documentation since they can clearly illustrate what an application is supposed to do. A good test suite shows what output should be expected for a set of test inputs, and that is a good way to explain the purpose of a method.

The symfony framework applies this principle to itself. The internals of the framework are validated by automated tests. These unit and functional tests are not bundled with the standard symfony distribution, but you can check them out from the SVN repository or browse them online at `http://www.symfony-project.com/trac/browser/trunk/test`.

Unit and Functional Tests

Unit tests confirm that a unitary code component provides the correct output for a given input. They validate how functions and methods work in every particular case. Unit tests deal with one case at a time, so for instance a single method may need several unit tests if it works differently in certain situations.

Functional tests validate not a simple input-to-output conversion, but a complete feature. For instance, a cache system can only be validated by a functional test, because it involves more than one step: The first time a page is requested, it is rendered; the second time, it is

taken from the cache. So functional tests validate a process and require a scenario. In symfony, you should write functional tests for all your actions.

For the most complex interactions, these two types may fall short. Ajax interactions, for instance, require a web browser to execute JavaScript, so automatically testing them requires a special third-party tool. Furthermore, visual effects can only be validated by a human.

If you have an extensive approach to automated testing, you will probably need to use a combination of all these methods. As a guideline, remember to keep tests simple and readable.

■ **Note** Automated tests work by comparing a result with an expected output. In other words, they evaluate *assertions* (expressions like `$a == 2`). The value of an assertion is either `true` or `false`, and it determines whether a test passes or fails. The word "assertion" is commonly used when dealing with automated testing techniques.

Test-Driven Development

In the test-driven development (TDD) methodology, the tests are written before the code. Writing tests first helps you to focus on the tasks a function should accomplish before actually developing it. It's a good practice that other methodologies, like Extreme Programming (XP), recommend as well. Plus it takes into account the undeniable fact that if you don't write unit tests first, you never write them.

For instance, imagine that you must develop a text-stripping function. The function removes white spaces at the beginning and at the end of the string, replaces nonalphabetical characters by underscores, and transforms all uppercase characters to lowercase ones. In test-driven development, you would first think about all the possible cases and provide an example "nput and expected output for each, as shown in Table 15-1.

Table 15-1. *A List of Test Cases for a Text-Stripping Function*

Input	Expected Output
`"  foo  "`	`"foo"`
`"foo bar"`	`"foo_bar"`
`"-)foo:..=bar?"`	`"__foo___bar_"`
`"FooBar"`	`"foobar"`
`"Don't foo-bar me!"`	`"don_t_foo_bar_me_"`

You would write the unit tests, run them, and see that they fail. You would then add the necessary code to handle the first test case, run the tests again, see that the first one passes, and go on like that. Eventually, when all the test cases pass, the function is correct.

An application built with a test-driven methodology ends up with roughly as much test code as actual code. As you don't want to spend time debugging your tests cases, keep them simple.

■ **Note** Refactoring a method can create new bugs that didn't use to appear before. That's why it is also a good practice to run all automated tests before deploying a new release of an application in production—this is called *regression testing*.

The Lime Testing Framework

There are many unit test frameworks in the PHP world, with the most well known being PhpUnit and SimpleTest. Symfony has its own, called *lime*. It is based on the `Test::More` Perl library, and is *TAP compliant*, which means that the result of tests is displayed as specified in the Test Anything Protocol, designed for better readability of test output.

Lime provides support for unit testing. It is more lightweight than other PHP testing frameworks and has several advantages:

- It launches test files in a sandbox to avoid strange side effects between each test run. Not all testing frameworks guarantee a clean environment for each test.
- Lime tests are very readable, and so is the test output. On compatible systems, lime uses color output in a smart way to distinguish important information.
- Symfony itself uses lime tests for regression testing, so many examples of unit and functional tests can be found in the symfony source code.
- The lime core is validated by unit tests.
- It is written in PHP, and it is fast and well coded. It is contained in a single file, `lime.php`, without any dependence.

The various tests described next use the lime syntax. They work out of the box with any symfony installation.

■ **Note** Unit and functional tests are not supposed to be launched in production. They are developer tools, and as such, they should be run in the developer's computer, not in the host server.

Unit Tests

Symfony unit tests are simple PHP files ending in `Test.php` and located in the `test/unit/` directory of your application. They follow a simple and readable syntax.

What Do Unit Tests Look Like?

Listing 15-1 shows a typical set of unit tests for the `strtolower()` function. It starts by an instantiation of the `lime_test` object (you don't need to worry about the parameters for now). Each unit test is a call to a method of the `lime_test` instance. The last parameter of these methods is always an optional string that serves as the output.

Listing 15-1. *Example Unit Test File, in `test/unit/strtolowerTest.php`*

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
require_once(dirname(__FILE__).'/../../lib/strtolower.php');

$t = new lime_test(7, new lime_output_color());

// strtolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string',
    'strtolower() returns a string');
$t->is(strtolower('FOO'), 'foo',
    'strtolower() transforms the input to lowercase');
$t->is(strtolower('foo'), 'foo',
    'strtolower() leaves lowercase characters unchanged');
$t->is(strtolower('12#?@~'), '12#?@~',
    'strtolower() leaves non alphabetical characters unchanged');
$t->is(strtolower('FOO BAR'), 'foo bar',
    'strtolower() leaves blanks alone');
$t->is(strtolower('FoO bAr'), 'foo bar',
    'strtolower() deals with mixed case input');
$t->is(strtolower(''), 'foo',
    'strtolower() transforms empty strings into foo');
```

Launch the test set from the command line with the `test-unit` task. The command-line output is very explicit, and it helps you localize which tests failed and which passed. See the output of the example test in Listing 15-2.

Listing 15-2. *Launching a Single Unit Test from the Command Line*

```
> symfony test-unit strtolower
```

```
1..7
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
ok 3 - strtolower() leaves lowercase characters unchanged
ok 4 - strtolower() leaves non alphabetical characters unchanged
ok 5 - strtolower() leaves blanks alone
```

```
ok 6 - strtolower() deals with mixed case input
not ok 7 - strtolower() transforms empty strings into foo
#     Failed test (.\batch\test.php at line 21)
#            got: ''
#       expected: 'foo'
# Looks like you failed 1 tests of 7.
```

■ **Tip** The `include` statement at the beginning of Listing 15-1 is optional, but it makes the test file an independent PHP script that you can execute without the symfony command line, by calling `php test/unit/strtolowerTest.php`.

Unit Testing Methods

The `lime_test` object comes with a large number of testing methods, as listed in Table 15-2.

Table 15-2. *Methods of the `lime_test` Object for Unit Testing*

Method	Description
`diag($msg)`	Outputs a comment but runs no test
`ok($test, $msg)`	Tests a condition and passes if it is true
`is($value1, $value2, $msg)`	Compares two values and passes if they are equal (`==`)
`isnt($value1, $value2, $msg)`	Compares two values and passes if they are not equal
`like($string, $regexp, $msg)`	Tests a string against a regular expression
`unlike($string, $regexp, $msg)`	Checks that a string doesn't match a regular expression
`cmp_ok($value1, $operator, $value2, $msg)`	Compares two arguments with an operator
`isa_ok($variable, $type, $msg)`	Checks the type of an argument
`isa_ok($object, $class, $msg)`	Checks the class of an object
`can_ok($object, $method, $msg)`	Checks the availability of a method for an object or a class
`is_deeply($array1, $array2, $msg)`	Checks that two arrays have the same values
`include_ok($file, $msg)`	Validates that a file exists and that it is properly included
`fail()`	Always fails—useful for testing exceptions
`pass()`	Always passes—useful for testing exceptions
`skip($msg, $nb_tests)`	Counts as `$nb_tests` tests—useful for conditional tests
`todo()`	Counts as a test—useful for tests yet to be written

The syntax is quite straightforward; notice that most methods take a message as their last parameter. This message is displayed in the output when the test passes. Actually, the best way to learn these methods is to test them, so have a look at Listing 15-3, which uses them all.

Listing 15-3. *Testing Methods of the* `lime_test` *Object, in* `test/unit/exampleTest.php`

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');

// Stub objects and functions for test purposes
class myObject
{
  public function myMethod()
  {
  }
}

function throw_an_exception()
{
  throw new Exception('exception thrown');
}

// Initialize the test object
$t = new lime_test(16, new lime_output_color());

$t->diag('hello world');
$t->ok(1 == '1', 'the equal operator ignores type');
$t->is(1, '1', 'a string is converted to a number for comparison');
$t->isnt(0, 1, 'zero and one are not equal');
$t->like('test01', '/test\d+/', 'test01 follows the test numbering pattern');
$t->unlike('tests01', '/test\d+/', 'tests01 does not follow the pattern');
$t->cmp_ok(1, '<', 2, 'one is inferior to two');
$t->cmp_ok(1, '!==', true, 'one and true are not identical');
$t->isa_ok('foobar', 'string', '\'foobar\' is a string');
$t->isa_ok(new myObject(), 'myObject', 'new creates object of the right class');
$t->can_ok(new myObject(), 'myMethod', 'objects of class myObject do have a ➥
myMethod method');
$array1 = array(1, 2, array(1 => 'foo', 'a' => '4'));
$t->is_deeply($array1, array(1, 2, array(1 => 'foo', 'a' => '4')),
    'the first and the second array are the same');
$t->include_ok('./fooBar.php', 'the fooBar.php file was properly included');

try
{
  throw_an_exception();
  $t->fail('no code should be executed after throwing an exception');
}
```

```
catch (Exception $e)
{
  $t->pass('exception catched successfully');
}

if (!isset($foobar))
{
  $t->skip('skipping one test to keep the test count exact in the condition', 1);
}
else
{
  $t->ok($foobar, 'foobar');
}

$t->todo('one test left to do');
```

You will find a lot of other examples of the usage of these methods in the symfony unit tests.

■ **Tip** You may wonder why you would use `is()` as opposed to `ok()` here. The error message output by `is()` is much more explicit; it shows both members of the test, while `ok()` just says that the condition failed.

Testing Parameters

The initialization of the `lime_test` object takes as its first parameter the number of tests that should be executed. If the number of tests finally executed differs from this number, the lime output warns you about it. For instance, the test set of Listing 15-3 outputs as Listing 15-4. The initialization stipulated that 16 tests were to run, but only 15 actually took place, so the output indicates this.

Listing 15-4. *The Count of Test Run Helps You to Plan Tests*

```
> symfony test-unit example
```

```
1..16
# hello world
ok 1 - the equal operator ignores type
ok 2 - a string is converted to a number for comparison
ok 3 - zero and one are not equal
ok 4 - test01 follows the test numbering pattern
ok 5 - tests01 does not follow the pattern
ok 6 - one is inferior to two
ok 7 - one and true are not identical
ok 8 - 'foobar' is a string
ok 9 - new creates object of the right class
```

```
ok 10 - objects of class myObject do have a myMethod method
ok 11 - the first and the second array are the same
not ok 12 - the fooBar.php file was properly included
#     Failed test (.\test\unit\testTest.php at line 27)
#       Tried to include './fooBar.php'
ok 13 - exception catched successfully
ok 14 # SKIP skipping one test to keep the test count exact in the condition
ok 15 # TODO one test left to do
# Looks like you planned 16 tests but only ran 15.
# Looks like you failed 1 tests of 16.
```

The `diag()` method doesn't count as a test. Use it to show comments, so that your test output stays organized and legible. On the other hand, the `todo()` and `skip()` methods count as actual tests. A `pass()`/`fail()` combination inside a `try`/`catch` block counts as a single test.

A well-planned test strategy must contain an expected number of tests. You will find it very useful to validate your own test files—especially in complex cases where tests are run inside conditions or exceptions. And if the test fails at some point, you will see it quickly because the final number of run tests won't match the number given during initialization.

The second parameter of the constructor is an output object extending the `lime_output` class. Most of the time, as tests are meant to be run through a CLI, the output is a `lime_output_color` object, taking advantage of bash coloring when available.

The test-unit Task

The `test-unit` task, which launches unit tests from the command line, expects either a list of test names or a file pattern. See Listing 15-5 for details.

Listing 15-5. *Launching Unit Tests*

```
// Test directory structure
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php

> symfony test-unit myFunction                    ## Run myFunctionTest.php
> symfony test-unit myFunction mySecondFunction   ## Run both tests
> symfony test-unit 'foo/*'                       ## Run barTest.php
> symfony test-unit '*'                           ## Run all tests (recursive)
```

Stubs, Fixtures, and Autoloading

In a unit test, the autoloading feature is not active by default. Each class that you use in a test must be either defined in the test file or required as an external dependency. That's why many test files start with a group of `include` lines, as Listing 15-6 demonstrates.

Listing 15-6. *Including Classes in Unit Tests*

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
include(dirname(__FILE__).'/../../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');

$t = new lime_test(7, new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
$t->is(sfToolkit::isPathAbsolute('/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('C:\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('d:/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('..\\test'), false,
    'isPathAbsolute() returns false if path is relative');
```

In unit tests, you need to instantiate not only the object you're testing, but also the object it depends upon. Since unit tests must remain unitary, depending on other classes may make more than one test fail if one class is broken. In addition, setting up real objects can be expensive, both in terms of lines of code and execution time. Keep in mind that speed is crucial in unit testing because developers quickly tire of a slow process.

Whenever you start including many scripts for a unit test, you may need a simple autoloading system. For this purpose, the `sfCore` class (which must be manually included) provides an `initSimpleAutoload()` method, which expects an absolute path as parameter. All the classes located under this path will be autoloaded. For instance, if you want to have all the classes located under `$sf_symfony_lib_dir/util/` autoloaded, start your unit test script as follows:

```
require_once($sf_symfony_lib_dir.'/util/sfCore.class.php');
sfCore::initSimpleAutoload($sf_symfony_lib_dir.'/util');
```

■ **Tip** The generated Propel objects rely on a long cascade of classes, so as soon as you want to test a Propel object, autoloading is necessary. Note that for Propel to work, you also need to include the files under the `vendor/propel/` directory (so the call to `sfCore` becomes `sfCore::initSimpleAutoload(array(SF_ROOT_DIR.'/lib/model', $sf_symfony_lib_dir.'/vendor/propel'));`) and to add the Propel core to the include path (by calling `set_include_path($sf_symfony_lib_dir.'/vendor'.PATH_SEPARATOR.SF_ROOT_DIR.PATH_SEPARATOR.get_include_path()`.

Another good workaround for the autoloading issues is the use of *stubs*. A stub is an alternative implementation of a class where the real methods are replaced with simple canned data. It mimics the behavior of the real class, but without its cost. A good example of stubs is a database connection or a web service interface. In Listing 15-7, the unit tests for a mapping API rely on a `WebService` class. Instead of calling the real `fetch()` method of the actual web service class, the test uses a stub that returns test data.

Listing 15-7. *Using Stubs in Unit Tests*

```
require_once(dirname(__FILE__).'/../../lib/WebService.class.php');
require_once(dirname(__FILE__).'/../../lib/MapAPI.class.php');

class testWebService extends WebService
{
  public static function fetch()
  {
    return file_get_contents(dirname(__FILE__).'/fixtures/data/ ➥
fake_web_service.xml');
  }
}

$myMap = new MapAPI();

$t = new lime_test(1, new lime_output_color());

$t->is($myMap->getMapSize(testWebService::fetch(), 100);
```

The test data can be more complex than a string or a call to a method. Complex test data is often referred to as *fixtures*. For coding clarity, it is often better to keep fixtures in separate files, especially if they are used by more than one unit test file. Also, don't forget that symfony can easily transform a YAML file into an array with the `sfYAML::load()` method. This means that instead of writing long PHP arrays, you can write your test data in a YAML file, as in Listing 15-8.

Listing 15-8. *Using Fixture Files in Unit Tests*

```
// In fixtures.yml:
-
  input:   '/test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   '\\test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'C:\\test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'd:/test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative
-
  input:   '../test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative
-
  input:   '..\\test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative

// In testTest.php
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
include(dirname(__FILE__).'/../../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');
require_once($sf_symfony_lib_dir.'/util/sfYaml.class.php');

$testCases = sfYaml::load(dirname(__FILE__).'/fixtures.yml');

$t = new lime_test(count($testCases), new lime_output_color());
```

```
// isPathAbsolute()
$t->diag('isPathAbsolute()');
foreach ($testCases as $case)
{
  $t->is(sfToolkit::isPathAbsolute($case['input']), $case['output'], ➥
 $case['comment']);
}
```

Functional Tests

Functional tests validate parts of your applications. They simulate a browsing session, make requests, and check elements in the response, just like you would do manually to validate that an action does what it's supposed to do. In functional tests, you run a scenario corresponding to a use case.

What Do Functional Tests Look Like?

You could run your functional tests with a text browser and a lot of regular expression assertions, but that would be a great waste of time. Symfony provides a special object, called `sfBrowser`, which acts like a browser connected to a symfony application without actually needing a server—and without the slowdown of the HTTP transport. It gives access to the core objects of each request (the request, session, context, and response objects). Symfony also provides an extension of this class called `sfTestBrowser`, designed especially for functional tests, which has all the abilities of the `sfBrowser` object plus some smart assert methods.

A functional test traditionally starts with an initialization of a test browser object. This object makes a request to an action and verifies that some elements are present in the response.

For example, every time you generate a module skeleton with the `init-module` or the `propel-init-crud` tasks, symfony creates a simple functional test for this module. The test makes a request to the default action of the module and checks the response status code, the module and action calculated by the routing system, and the presence of a certain sentence in the response content. For a `foobar` module, the generated `foobarActionsTest.php` file looks like Listing 15-9.

Listing 15-9. *Default Functional Test for a New Module, in `tests/functional/frontend/foobarActionsTest.php`*

```
<?php

include(dirname(__FILE__).'/../../bootstrap/functional.php');

// Create a new test browser
$browser = new sfTestBrowser();
$browser->initialize();
```

```
$browser->
  get('/foobar/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'foobar')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '!/This is a temporary page/')
;
```

Tip The browser methods return an `sfTestBrowser` object, so you can chain the method calls for more readability of your test files. This is called a *fluid interface* to the object, because nothing stops the flow of method calls.

A functional test can contain several requests and more complex assertions; you will soon discover all the possibilities in the upcoming sections.

To launch a functional test, use the `test-functional` task with the symfony command line, as shown in Listing 15-10. This task expects an application name and a test name (omit the `Test.php` suffix).

Listing 15-10. *Launching a Single Functional Test from the Command Line*

```
> symfony test-functional frontend foobarActions
```

```
# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
not ok 4 - response selector body does not match regex /This is a temporary page/
# Looks like you failed 1 tests of 4.
1..4
```

The generated functional tests for a new module don't pass by default. This is because in a newly created module, the `index` action forwards to a congratulations page (included in the symfony `default` module), which contains the sentence "This is a temporary page." As long as you don't modify the `index` action, the tests for this module will fail, and this guarantees that you cannot pass all tests with an unfinished module.

Note In functional tests, the autoloading is activated, so you don't have to include the files by hand.

Browsing with the sfTestBrowser Object

The test browser is capable of *making GET and POST requests*. In both cases, use a real URI as parameter. Listing 15-11 shows how to write calls to the `sfTestBrowser` object to simulate requests.

Listing 15-11. *Simulating Requests with the `sfTestBrowser` Object*

```
include(dirname(__FILE__).'/../../bootstrap/functional.php');

// Create a new test browser
$b = new sfTestBrowser();
$b->initialize();

$b->get('/foobar/show/id/1');                    // GET request
$b->post('/foobar/show', array('id' => 1));      // POST request

// The get() and post() methods are shortcuts to the call() method
$b->call('/foobar/show/id/1', 'get');
$b->call('/foobar/show', 'post', array('id' => 1));

// The call() method can simulate requests with any method
$b->call('/foobar/show/id/1', 'head');
$b->call('/foobar/add/id/1', 'put');
$b->call('/foobar/delete/id/1', 'delete');
```

A typical browsing session contains not only requests to specific actions, but also *clicks on links and on browser buttons*. As shown in Listing 15-12, the `sfTestBrowser` object is also capable of simulating those.

Listing 15-12. *Simulating Navigation with the `sfTestBrowser` Object*

```
$b->get('/');                  // Request to the home page
$b->get('/foobar/show/id/1');
$b->back();                    // Back to one page in history
$b->forward();                 // Forward one page in history
$b->reload();                  // Reload current page
$b->click('go');               // Look for a 'go' link or button and click it
```

The test browser handles a stack of calls, so the `back()` and `forward()` methods work as they do on a real browser.

■ **Tip** The test browser has its own mechanisms to manage sessions (`sfTestStorage`) and cookies.

Among the interactions that most need to be tested, those associated with *forms* probably rank first. To simulate form input and submission, you have three choices. You can either make a POST request with the parameters you wish to send, call `click()` with the form parameters as an array, or fill in the fields one by one and click the submit button. They all result in the same POST request anyhow. Listing 15-13 shows an example.

Listing 15-13. *Simulating Form Input with the `sfTestBrowser` Object*

```
// Example template in modules/foobar/templates/editSuccess.php
<?php echo form_tag('foobar/update') ?>
  <?php echo input_hidden_tag('id', $sf_params->get('id')) ?>
  <?php echo input_tag('name', 'foo') ?>
  <?php echo submit_tag('go') ?>
  <?php echo textarea('text1', 'foo') ?>
  <?php echo textarea('text2', 'bar') ?>
</form>

// Example functional test for this form
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');

// Option 1: POST request
$b->post('/foobar/update', array('id' => 1, 'name' => 'dummy', 'commit' => 'go'));

// Option 2: Click the submit button with parameters
$b->click('go', array('name' => 'dummy'));

// Option 3: Enter the form values field by field name then click the submit button
$b->setField('name', 'dummy')->
    click('go');
```

■ **Note** With the second and third options, the default form values are automatically included in the form submission, and the form target doesn't need to be specified.

When an action finishes by a `redirect()`, the test browser doesn't automatically follow the redirection; you must follow it manually with `followRedirect()`, as demonstrated in Listing 15-14.

Listing 15-14. *The Test Browser Doesn't Automatically Follow Redirects*

```
// Example action in modules/foobar/actions/actions.class.php
public function executeUpdate()
{
  ...
  $this->redirect('foobar/show?id='.$this->getRequestParameter('id'));
}
```

```
// Example functional test for this action
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    click('go', array('name' => 'dummy'))->
    isRedirected()->   // Check that request is redirected
    followRedirect();    // Manually follow the redirection
```

There is one last method you should know about that is useful for browsing: `restart()` reinitializes the browsing history, session, and cookies—as if you restarted your browser.

Once it has made a first request, the `sfTestBrowser` object can *give access to the request, context, and response* objects. It means that you can check a lot of things, ranging from the text content to the response headers, the request parameters, and configuration:

```
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();
```

THE SFBROWSER OBJECT

All the browsing methods described in Listings 15-10 to 15-13 are also available out of the testing scope, throughout the `sfBrowser` object. You can call it as follows:

```
// Create a new browser
$b = new sfBrowser();
$b->initialize();
$b->get('/foobar/show/id/1')->
    setField('name', 'dummy')->
    click('go');
$content = $b()->getResponse()->getContent();
...
```

The `sfBrowser` object is a very useful tool for batch scripts, for instance, if you want to browse a list of pages to generate a cached version for each (refer to Chapter 18 for a detailed example).

Using Assertions

Due to the `sfTestBrowser` object having access to the response and other components of the request, you can do tests on these components. You could create a new `lime_test` object for that purpose, but fortunately `sfTestBrowser` proposes a `test()` method that returns a `lime_test` object where you can call the unit assertion methods described previously. Check Listing 15-15 to see how to do assertions via `sfTestBrowser`.

Listing 15-15. *The Test Browser Provides Testing Abilities with the `test()` Method*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();

// Get access to the lime_test methods via the test() method
$b->test()->is($request->getParameter('id'), 1);
$b->test()->is($response->getStatuscode(), 200);
$b->test()->is($response->getHttpHeader('content-type'), 'text/html; ➥
charset=utf-8');
$b->test()->like($response->getContent(), '/edit/');
```

■ **Note** The `getResponse()`, `getContext()`, `getRequest()`, and `test()` methods don't return an `sfTestBrowser` object, therefore you can't chain other `sfTestBrowser` method calls after them.

You can check incoming and outgoing cookies easily via the request and response objects, as shown in Listing 15-16.

Listing 15-16. *Testing Cookies with `sfTestBrowser`*

```
$b->test()->is($request->getCookie('foo'), 'bar');     // Incoming cookie
$cookies = $response->getCookies();
$b->test()->is($cookies['foo'], 'foo=bar');            // Outgoing cookie
```

Using the `test()` method to test the request elements ends up in long lines. Fortunately, `sfTestbrowser` contains a bunch of *proxy methods* that help you keep your functional tests readable and short—in addition to returning an `sfTestBrowser` object themselves. For instance, you can rewrite Listing 15-15 in a faster way, as shown in Listing 15-17.

Listing 15-17. *Testing Directly with `sfTestBrowser`*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    isRequestParameter('id', 1)->
    isStatutsCode()->
    isResponseHeader('content-type', 'text/html; charset=utf-8')->
    responseContains('edit');
```

The status 200 is the default value of the parameter expected by `isStatusCode()`, so you can call it without any argument to test a successful response.

One more advantage of proxy methods is that you don't need to specify an output text as you would with a `lime_test` method. The messages are generated automatically by the proxy methods, and the test output is clear and readable.

```
# get /foobar/edit/id/1
ok 1 - request parameter "id" is "1"
ok 2 - status code is "200"
ok 3 - response header "content-type" is "text/html"
ok 4 - response contains "edit"
1..4
```

In practice, the proxy methods of Listing 15-17 cover most of the usual tests, so you will seldom use the `test()` method on an `sfTestBrowser` object.

Listing 15-14 showed that `sfTestBrowser` doesn't automatically follow redirections. This has one advantage: You can test a redirection. For instance, Listing 15-18 shows how to test the response of Listing 15-14.

Listing 15-18. *Testing Redirections with* `sfTestBrowser`

```
$b = new sfTestBrowser();
$b->initialize();
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'update')->

    isRedirected()->      // Check that the response is a redirect
    followRedirect()->    // Manually follow the redirection

    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'show');
```

Using CSS Selectors

Many of the functional tests validate that a page is correct by checking for the presence of text in the content. With the help of regular expressions in the `responseContains()` method, you can check displayed text, a tag's attributes, or values. But as soon as you want to check something deeply buried in the response DOM, regular expressions are not ideal.

That's why the `sfTestBrowser` object supports a `getResponseDom()` method. It returns a libXML2 DOM object, much easier to parse and test than a flat text. Refer to Listing 15-19 for an example of using this method.

Listing 15-19. *The Test Browser Gives Access to the Response Content As a DOM Object*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$dom = $b->getResponseDom();
$b->test()->is($dom->getElementsByTagName('input')->item(1)->getAttribute('type'),➥
 'text');
```

But parsing an HTML document with the PHP DOM methods is still not fast and easy enough. If you are familiar with the CSS selectors, you know that they are an ever more powerful way to retrieve elements from an HTML document. Symfony provides a tool class called `sfDomCssSelector` that expects a DOM document as construction parameter. It has a `getTexts()` method that returns an array of strings according to a CSS selector, and a `getElements()` method that returns an array of DOM elements. See an example in Listing 15-20.

Listing 15-20. *The Test Browser Gives Access to the Response Content As an `sfDomCssSelector` Object*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$c = new sfDomCssSelector($b->getResponseDom())
$b->test()->is($c->getTexts('form input[type="hidden"][value="1"]'), array(''));
$b->test()->is($c->getTexts('form textarea[name="text1"]'), array('foo'));
$b->test()->is($c->getTexts('form input[type="submit"]'), array(''));
```

In its constant pursuit for brevity and clarity, symfony provides a shortcut for this: the `checkResponseElement()` proxy method. This method makes Listing 15-20 look like Listing 15-21.

Listing 15-21. *The Test Browser Gives Access to the Elements of the Response by CSS Selectors*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    checkResponseElement('form input[type="hidden"][value="1"]', true->
    checkResponseElement('form textarea[name="text1"]', 'foo')->
    checkResponseElement('form input[type="submit"]', 1);
```

The behavior of the `checkResponseElement()` method depends on the type of the second argument that it receives:

- If it is a Boolean, it checks that an element matching the CSS selector exists.
- If it is an integer, it checks that the CSS selector returns this number of results.
- If it is a regular expression, it checks that the first element found by the CSS selector matches it.

- If it is a regular expression preceded by `!`, it checks that the first element doesn't match the pattern.
- For other cases, it compares the first element found by the CSS selector with the second argument as a string.

The method accepts a third optional parameter, in the shape of an associative array. It allows you to have the test performed not on the first element returned by the selector (if it returns several), but on another element at a certain position, as shown in Listing 15-22.

Listing 15-22. *Using the Position Option to Match an Element at a Certain Position*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form textarea', 'foo')->
    checkResponseElement('form textarea', 'bar', array('position' => 1));
```

The options array can also be used to perform two tests at the same time. You can test that there is an element matching a selector and how many there are, as demonstrated in Listing 15-23.

Listing 15-23. *Using the Count Option to Count the Number of Matches*

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form input', true, array('count' => 3));
```

The selector tool is very powerful. It accepts most of the CSS 2.1 selectors, and you can use it for complex queries such as those of Listing 15-24.

Listing 15-24. *Example of Complex CSS Selectors Accepted by `checkResponseElement()`*

```
$b->checkResponseElement('ul#list li a[href]', 'click me');
$b->checkResponseElement('ul > li', 'click me');
$b->checkResponseElement('ul + li', 'click me');
$b->checkResponseElement('h1, h2', 'click me');
$b->checkResponseElement('a[class$="foo"][href*="bar.html"]', 'my link');
```

Working in the Test Environment

The `sfTestBrowser` object uses a special front controller, set to the `test` environment. The default configuration for this environment appears in Listing 15-25.

Listing 15-25. *Default Test Environment Configuration, in `myapp/config/settings.php`*

```
test:
  .settings:
    # E_ALL | E_STRICT & ~E_NOTICE = 2047
```

```
  error_reporting:        2047
  cache:                  off
  web_debug:              off
  no_script_name:         off
  etag:                   off
```

The cache and the web debug toolbar are set to `off` in this environment. However, the code execution still leaves traces in a log file, distinct from the `dev` and `prod` log files, so that you can check it independently (`myproject/log/myapp_test.log`). In this environment, the exceptions don't stop the execution of the scripts—so that you can run an entire set of tests even if one fails. You can have specific database connection settings, for instance, to use another database with test data in it.

Before using the `sfTestBrowser` object, you have to initialize it. If you need to, you can specify a hostname for the application and an IP address for the client—that is, if your application makes controls over these two parameters. Listing 15-26 demonstrates how to do this.

Listing 15-26. *Setting Up the Test Browser with Hostname and IP*

```
$b = new sfTestBrowser();
$b->initialize('myapp.example.com', '123.456.789.123');
```

The test-functional Task

The `test-functional` task can run one or more functional tests, depending on the number of arguments received. The rules look much like the ones of the `test-unit` task, except that the functional test task always expects an application as first argument, as shown in Listing 15-27.

Listing 15-27. *Functional Test Task Syntax*

```
// Test directory structure
test/
  functional/
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php

## Run all functional tests for one application, recursively
> symfony test-functional frontend

## Run one given functional test
> symfony test-functional frontend myScenario

## Run several tests based on a pattern
> symfony test-functional frontend my*
```

Test Naming Practices

This section lists a few good practices to keep your tests organized and easy to maintain. The tips concern file organization, unit tests, and functional tests.

As for the *file structure*, you should name the unit test files using the class they are supposed to test, and name the functional test files using the module or the scenario they are supposed to test. See Listing 15-28 for an example. Your `test/` directory will soon contain a lot of files, and finding a test might prove difficult in the long run if you don't follow these guidelines.

Listing 15-28. *Example File Naming Practice*

```
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php
  functional/
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php
```

For *unit tests*, a good practice is to group the tests by function or method, and start each test group with a `diag()` call. The messages of each unit test should contain the name of the function or method tested, followed by a verb and a property, so that the test output looks like a sentence describing a property of the object. Listing 15-29 shows an example.

Listing 15-29. *Example Unit Test Naming Practice*

```
// srttolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string', 'strtolower() returns a string');
$t->is(strtolower('FOO'), 'foo', 'strtolower() transforms the input to lowercase');
```

```
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
```

Functional tests should be grouped by page and start by a request. Listing 15-30 illustrates this practice.

Listing 15-30. *Example Functional Test Naming Practice*

```
$browser->
  get('/foobar/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'foobar')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '/foobar/')
;
```

```
# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
ok 4 - response selector body matches regex /foobar/
```

If you follow this convention, the output of your test will be clean enough to use as a developer documentation of your project—enough so in some cases to make actual documentation useless.

Special Testing Needs

The unit and functional test tools provided by symfony should suffice in most cases. A few additional techniques are listed here to resolve common problems in automated testing: launching tests in an isolated environment, accessing a database within tests, testing the cache, and testing interactions on the client side.

Executing Tests in a Test Harness

The `test-unit` and `test-functional` tasks can launch a single test or a set of tests. But if you call these tasks without any parameter, they launch all the unit and functional tests written in the `test/` directory. A particular mechanism is involved to isolate each test file in an independent sandbox, to avoid contamination risks between tests. Furthermore, as it wouldn't make sense to keep the same output as with single test files in that case (the output would be thousands of lines long), the tests results are compacted into a synthetic view. That's why the execution of a large number of test files uses a *test harness*, that is, an automated test framework with special abilities. A test harness relies on a component of the lime framework called `lime_harness`. It shows a test status file by file, and an overview at the end of the number of tests passed over the total, as you see in Listing 15-31.

Listing 15-31. *Launching All Tests in a Test Harness*

```
> symfony test-unit
```

```
unit/myFunctionTest.php.............ok
unit/mySecondFunctionTest.php.........ok
unit/foo/barTest.php..................not ok

Failed Test                    Stat  Total   Fail  List of Failed
---------------------------------------------------------------
unit/foo/barTest.php              0      2      2  62 63
Failed 1/3 test scripts, 66.66% okay. 2/53 subtests failed, 96.22% okay.
```

The tests are executed the same way as when you call them one by one, only the output is made shorter to be really useful. In particular, the final chart focuses on the failed tests and helps you locate them.

You can launch all the tests with one call using the `test-all` task, which also uses a test harness, as shown in Listing 15-32. This is something that you should do before every transfer to production, to ensure that no regression has appeared since the latest release.

Listing 15-32. *Launching All the Tests of a Project*

```
> symfony test-all
```

Accessing a Database

Unit tests often need to access a database. To initialize a database connection, use the `getConnection()` method of the `Propel` class, as in Listing 15-33.

Listing 15-33. *Initializing a Database in a Test*

```
$con = Propel::getConnection();
```

You should populate the database with fixtures before starting the tests. This can be done via the `sfPropelData` object. This object can load data from a file, just like the `propel-load-data` task, or from an array, as shown in Listing 15-34.

Listing 15-34. *Populating a Database from a Test File*

```
$data = new sfPropelData();

// Loading data from file
$data->loadData(sfConfig::get('sf_data_dir').'/fixtures/test_data.yml');
```

```
// Loading data from array
$fixtures = array(
  'Article' => array(
    'article_1' => array(
      'title'      => 'foo title',
      'body'       => 'bar body',
      'created_at' => time(),
    ),
    'article_2'    => array(
      'title'      => 'foo foo title',
      'body'       => 'bar bar body',
      'created_at' => time(),
    ),
  ),
);
$data->loadDataFromArray($fixtures);
```

Then, use the Propel objects as you would in a normal application, according to your testing needs. Remember to include their files in unit tests (you can use the `sfCore::sfSimpleAutoloading()` method to automate it, as explained in a tip in the "Stubs, Fixtures, and Autoloading" section previously in this chapter). Propel objects are autoloaded in functional tests.

Testing the Cache

When you enable caching for an application, the functional tests should verify that the cached actions do work as expected.

The first thing to do is enable cache for the test environment (in the `settings.yml` file). Then, if you want to test whether a page comes from the cache or whether it is generated, you should use the `isCached()` test method provided by the `sfTestBrowser` object. Listing 15-35 demonstrates this method.

Listing 15-35. *Testing the Cache with the `isCached()` Method*

```
<?php

include(dirname(__FILE__).'/../../bootstrap/functional.php');

// Create a new test browser
$b = new sfTestBrowser();
$b->initialize();

$b->get('/mymodule');
$b->isCached(true);       // Checks that the response comes from the cache
$b->isCached(true, true); // Checks that the cached response comes with layout
$b->isCached(false);      // Checks that the response doesn't come from the cache
```

■ **Note** You don't need to clear the cache at the beginning of a functional test; the bootstrap script does it for you.

Testing Interactions on the Client

The main drawback of the techniques described previously is that they cannot simulate JavaScript. For very complex interactions, like with Ajax interactions for instance, you need to be able to reproduce exactly the mouse and keyboard input that a user would do and execute scripts on the client side. Usually, these tests are reproduced by hand, but they are very time consuming and prone to error.

The solution is called *Selenium* (`http://www.openqa.org/selenium/`), which is a test framework written entirely in JavaScript. It executes a set of actions on a page just like a regular user would, using the current browser window. The advantage over the `sfBrowser` object is that Selenium is capable of executing JavaScript in a page, so you can test even Ajax interactions with it.

Selenium is not bundled with symfony by default. To install it, you need to create a new `selenium/` directory in your `web/` directory, and in it unpack the content of the Selenium archive (`http://www.openqa.org/selenium-core/download.action`). This is because Selenium relies on JavaScript, and the security settings standard in most browsers wouldn't allow it to run unless it is available on the same host and port as your application.

■ **Caution** Be careful not to transfer the `selenium/` directory to your production server, since it would be accessible by anyone having access to your web document root via the browser.

Selenium tests are written in HTML and stored in the `web/selenium/tests/` directory. For instance, Listing 15-36 shows a functional test where the home page is loaded, the link click me is clicked, and the text "Hello, World" is looked for in the response. Remember that in order to access the application in the `test` environment, you have to specify the `myapp_test.php` front controller.

Listing 15-36. *A Sample Selenium Test, in* `web/selenium/test/testIndex.html`

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
  <title>Index tests</title>
</head>
<body>
<table cellspacing="0">
```

```
<tbody>
  <tr><td colspan="3">First step</td></tr>
  <tr><td>open</td>              <td>/myapp_test.php/</td> <td> </td></tr>
  <tr><td>clickAndWait</td>      <td>link=click me</td>    <td> </td></tr>
  <tr><td>assertTextPresent</td> <td>Hello, World!</td>    <td> </td></tr>
</tbody>
</table>
</body>
</html>
```

A test case is represented by an HTML document containing a table with three columns: command, target, and value. Not all commands take a value, however. In this case, either leave the column blank or use ` ` to make the table look better. Refer to the Selenium website for a complete list of commands.

You also need to add this test to the global test suite by inserting a new line in the table of the `TestSuite.html` file, located in the same directory. Listing 15-37 shows how.

Listing 15-37. *Adding a Test File to the Test Suite, in* `web/selenium/test/TestSuite.html`

```
...
<tr><td><a href='./testIndex.html'>My First Test</a></td></tr>
...
```

To run the test, simply browse to

```
http://myapp.example.com/selenium/index.html
```

Select Main Test Suite, click the button to run all tests, and watch your browser as it reproduces the steps that you have told it to do.

■ **Note** As Selenium tests run in a real browser, they also allow you to test browser inconsistencies. Build your test with one browser, and test them on all the others on which your site is supposed to work with a single request.

The fact that Selenium tests are written in HTML could make the writing of Selenium tests a hassle. But thanks to the Firefox Selenium extension (`http://seleniumrecorder.mozdev.org/`), all it takes to create a test is to execute the test once in a recorded session. While navigating in a recording session, you can add assert-type tests by right-clicking in the browser window and selecting the appropriate check under Append Selenium Command in the pop-up menu.

You can save the test to an HTML file to build a test suite for your application. The Firefox extension even allows you to run the Selenium tests that you have recorded with it.

■ **Note** Don't forget to reinitialize the test data before launching the Selenium test.

Summary

Automated tests include *unit tests* to validate methods or functions and *functional tests* to validate features. Symfony relies on the *lime* testing framework for unit tests and provides an `sfTestBrowser` class especially for functional tests. They both provide many assertion methods, from basic to the most advanced, like *CSS selectors*. Use the *symfony command line* to launch tests, either one by one (with the `test-unit` and `test-functional` tasks) or in a *test harness* (with the `test-all` task). When dealing with data, automated tests use *fixtures* and *stubs*, and this is easily achieved within symfony unit tests.

If you make sure to write enough unit tests to cover a large part of your applications (maybe using the TDD methodology), you will feel safer when *refactoring* internals or adding new features, and you may even gain some time on the *documentation* task.