Mink Documentation

Release 1.6

Konstantin Kudryashov (everzet)

May 13, 2017

Contents

One of the most important parts in the web is a browser. Browser is the window through which web users interact with web applications and other users. Users are always talking with web applications through browsers.

So, in order to test that our web application behaves correctly, we need a way to simulate this interaction between the browser and the web application in our tests. We need a **Mink**.

Mink is an open source browser controller/emulator for web applications, written in PHP 5.3.

Read *Mink at a Glance* to learn more about Mink and why you need it.

CHAPTER 1

Installation

Mink is a php 5.3 library that you'll use inside your test suites or project. Before you begin, ensure that you have at least PHP 5.3.1 installed.

The recommended way to install Mink with all its dependencies is through Composer:

```
$ composer require behat/mink
```

Note: For local installations of composer you must call it like this: `$ php composer.phar require behat/mink .` In this case you must use the different call `php composer.phar` everywhere instead of the simple command `composer`.

Everything will be installed inside `vendor` folder. Finally, include Composer autoloading script to your project:

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

Note: By default, Mink will be installed with no drivers. In order to be able to use additional drivers, you should install them (through composer). Require the appropriate dependencies:

- GoutteDriver - `behat/mink-goutte-driver`
- Selenium2Driver - `behat/mink-selenium2-driver`
- BrowserKitDriver - `behat/mink-browserkit-driver`
- ZombieDriver - `behat/mink-zombie-driver`
- SeleniumDriver - `behat/mink-selenium-driver`
- SahiDriver - `behat/mink-sahi-driver`
- WUnitDriver - `behat/mink-wunit-driver`

If you're newcomer or just don't know what to choose, you should probably start with the GoutteDriver and the Selenium2Driver (you will be able to tuneup it later):

CHAPTER 2

Guides

Learn Mink with the topical guides:

Mink at a Glance

There's huge number of browser emulators out there, like Goutte, Selenium, Sahi and others. They all do the same job, but do it very differently. They behave differently and have very different API's. But, what's more important, there is actually 2 completely different types of browser emulators out there:

- Headless browser emulators
- Browser controllers

First type browser emulators are simple pure HTTP specification implementations, like Goutte. Those browser emulators send a real HTTP requests against an application and parse the response content. They are very simple to run and configure, because this type of emulators can be written in any available programming language and can be run through console on servers without GUI. Headless emulators have both advantages and disadvantages. Advantages are simplicity, speed and ability to run it without the need of a real browser. But this type of browsers has one big disadvantage, they have no JS/AJAX support. So, you can't test your rich GUI web applications with headless browsers.

Second browser emulators type are browser controllers. Those emulators aim to control the real browser. That's right, a program to control another program. Browser controllers simulate user interactions on browser and are able to retrieve actual information from current browser page. Selenium and Sahi are the two most famous browser controllers. The main advantage of browser controllers usage is the support for JS/AJAX interactions on page. The disadvantage is that such browser emulators require the installed browser, extra configuration and are usually much slower than headless counterparts.

So, the easy answer is to choose the best emulator for your project and use its API for testing. But as we've already seen, both browser emulator types have both advantages and disadvantages. If you choose headless browser emulator, you will not be able to test your JS/AJAX pages. And if you choose browser controller, your overall test suite will become very slow at some point. So, in real world we should use both! And that's why you need a **Mink**.

Mink removes API differences between different browser emulators providing different drivers (read in *Drivers* chapter) for every browser emulator and providing you with the easy way to control the browser (*Controlling the Browser*),

traverse pages (*Traversing Pages*), manipulate page elements (*Manipulating Pages*) or interact with them (*Interacting with Pages*).

Controlling the Browser

In Mink, the entry point to the browser is called the session. Think about it as being your browser window (some drivers even let you switch tabs!).

First, start your session (it's like opening your browser tab). Nothing can be done with it before starting it.

```
// Choose a Mink driver. More about it in later chapters.
$driver = new \Behat\Mink\Driver\GoutteDriver();

$session = new \Behat\Mink\Session($driver);

// start the session
$session->start();
```

Note: The first argument to the session constructor is a driver object. Drivers are the way the Mink abstraction layer works. You will discover more about the available drivers in a *later chapter*.

Caution: Although Mink does its best to remove differences between the different drivers, each driver has unique features and shortcomings. See the *Driver Feature Support* to see which features are supported by each driver.

Basic Browser Interaction

Now that your session is started, you'll want to open a page with it. Just after starting, the session is not on any page (in a real browser, you would be on the `about:blank` page), and calling any other action is likely to fail.

```
$session->visit('http://my_project.dev/some_page.php');
```

Note: Mink is primarily designed to be used for testing websites. To allow you to browse and test error pages, the `Session::visit` method does not consider error status codes as invalid. It will *not* throw an exception in this case. You will need to check the status code (or certain text on the page) to know if the response was successful or not.

Interacting with the Page

The session gives you access to the page through the `Session::getPage` method. This allows you to *traverse the page*, *manipulate page elements* and *interact* with them. The next chapters cover the page API in depth. Most of what you'll do with Mink will use this object, but you can continue reading to learn more about the Session.

Using the Browser History

The session gives you access to the browser history:

```php
// get the current page URL:
echo $session->getCurrentUrl();

// use history controls:
$session->reload();
$session->back();
$session->forward();
```

Cookie Management

The session can manipulate cookies available in the browser.

```php
// set cookie:
$session->setCookie('cookie name', 'value');

// get cookie:
echo $session->getCookie('cookie name');

// delete cookie:
$session->setCookie('cookie name', null);
```

Note: With drivers that use JavaScript to control the browser - like Sahi - you may be restricted to accessing/setting all, but HttpOnly cookies .

Status Code Retrieval

The session lets you retrieve the HTTP status code of the response:

```php
echo $session->getStatusCode();
```

Headers Management

The session lets you manipulate request headers and access response headers:

```php
// setting browser language:
$session->setRequestHeader('Accept-Language', 'fr');

// retrieving response headers:
print_r($session->getResponseHeaders());
```

Note: Headers handling is only supported in headless drivers (e.g. Goutte). Browser controllers (e.g. Selenium2) cannot access that information.

HTTP Authentication

The session has a special method to perform HTTP Basic authentication:

```
$session->setBasicAuth($user, $password);
```

The method can also be used to reset a previous authentication:

```
$session->setBasicAuth(false);
```

Note: Automatic HTTP authentication is only supported in headless drivers. Because HTTP authentication in browser requires manual user action, that can't be done remotely for browser controllers.

Javascript Evaluation

The session allows you to execute or evaluate Javascript.

```
// Execute JS
$session->executeScript('document.body.firstChild.innerHTML = "";');

// evaluate JS expression:
echo $session->evaluateScript(
    "return 'something from browser';"
);
```

Note: The difference between these methods is that `Session::evaluateScript` returns the result of the expression. When you don't need to get a return value, using `Session::executeScript` is better.

You can also wait until a given JS expression returns a truthy value or the timeout is reached:

```
// wait for n milliseconds or
// till JS expression becomes truthy:
$session->wait(
    5000,
    "$('.suggestions-results').children().length"
);
```

Note: The `Session::wait` method returns `true` when the evaluation becomes truthy. It will return `false` when the timeout is reached.

Resetting the Session

The primary aim for Mink is to provide a single consistent web browsing API for acceptance tests. But a very important part in testing is isolation.

Mink provides two very useful methods to isolate tests, which can be used in your test's `teardown` methods:

```
// soft-reset:
$session->reset();

// hard-reset:
$session->stop();
```

```
// or if you want to start again at the same time
$session->restart();
```

Stopping the session is the best way to reset the session to its initial state. It will close the browser entirely. To use the session again, you need to start the session before any other action. The `Session::restart` shortcut allows you to do these 2 steps in a single call.

The drawback of closing the browser and starting it again is that it takes time. In many cases, a lower level of isolation is enough in favor of a faster resetting. The `Session::reset` method covers this use case. It will try to clear the cookies and reset the request headers and the browser history to the limit of the driver possibilities.

Taking all this into account, it is recommended to use `Session::reset()` by default and to call `Session::stop()` when you need really full isolation.

Traversing Pages

Most usages of Mink will involve working with the page opened in your browser. This is done thanks to the powerful Element API. This API allows to traverse the page (similar to the DOM in Javascript), *manipulate page elements* and to *interact with them*, which will be covered in the next chapters.

DocumentElement and NodeElement

The Element API consists of 2 main classes. The `DocumentElement` instance represents the page being displayed in the browser, while the `NodeElement` class is used to represent any element inside the page. Both classes share a common set of methods to traverse the page (defined in `TraversableElement`).

The `DocumentElement` instance is accessible through the `Session::getPage` method:

```
$page = $session->getPage();

// You can now manipulate the page.
```

Note: The `DocumentElement` instance represents the `<html>` node in the DOM. It is equivalent to `document.documentElement` in the Javascript DOM API.

Traversal Methods

Elements have 2 main traversal methods: `ElementInterface::findAll` returns an array of `NodeElement` instances matching the provided *selector* inside the current element while `ElementInterface::find` returns the first match or `null` when there is none.

The `TraversableElement` class also provides a bunch of shortcut methods on top of `find()` to make it easier to achieve many common use cases:

`ElementInterface::has` Checks whether a child element matches the given selector but without returning it.

`TraversableElement::findById` Looks for a child element with the given id.

`TraversableElement::findLink` Looks for a link with the given text, title, id or `alt` attribute (for images used inside links).

`TraversableElement::findButton` Looks for a button with the given text, title, id, `name` attribute or `alt` attribute (for images used inside links).

`TraversableElement::findField` Looks for a field (`input`, `textarea` or `select`) with the given label, placeholder, id or `name` attribute.

Note: These shortcuts are returning a single element. If you need to find all matches, you will need to use `findAll` with the *named selector*.

Nested Traversing

Every `find*()` method will return a `Behat\Mink\Element\NodeElement` instance and `findAll()` will return an array of such instances. The fun part is that you can make same old traversing on such elements as well:

```
$registerForm = $page->find('css', 'form.register');

if (null === $registerForm) {
    throw new \Exception('The element is not found');
}

// find some field INSIDE form with class="register"
$field = $registerForm->findField('Email');
```

Selectors

The `ElementInterface::find` and `ElementInterface::findAll` methods support several kinds of selectors to find elements.

CSS Selector

The `css` selector type lets you use CSS expressions to search for elements on the page:

```
$title = $page->find('css', 'h1');

$buttonIcon = $page->find('css', '.btn > .icon');
```

XPath Selector

The `xpath` selector type lets you use XPath queries to search for elements on the page:

```
$anchorsWithoutUrl = $page->findAll('xpath', '//a[not(@href)]');
```

Caution: This selector searches for an element inside the current node (which is `<html>` for the page object). This means that trying to pass it the XPath of and element retrieved with `ElementInterface::getXpath` will not work (this query includes the query for the root node). To check whether an element object still exists on the browser page, use `ElementInterface::isValid` instead.

Named Selectors

Named selectors provide a set of reusable queries for common needs. For conditions based on the content of elements, the named selector will try to find an exact match first. It will then fallback to partial matching in case there is no

result for the exact match. The `named_exact` selector type can be used to force using only exact matching. The `named_partial` selector type can be used to apply partial matching without preferring exact matches.

For the named selector type, the second argument of the `find()` method is an array with 2 elements: the name of the query to use and the value to search with this query:

```
$topLink = $page->find('named', array('link', $escapedValue));
```

The following queries are supported by the named selector:

`id` Searches for an element by its id.

`id_or_name` Searches for an element by its id or name.

`link` Searches for a link by its id, title, img alt, rel or text.

`button` Searches for a button by its name, id, text, img alt or title.

`link_or_button` Searches for both links and buttons.

`content` Searches for a specific page content (text).

`field` Searches for a form field by its id, name, label or placeholder.

`select` Searches for a select field by its id, name or label.

`checkbox` Searches for a checkbox by its id, name, or label.

`radio` Searches for a radio button by its id, name, or label.

`file` Searches for a file input by its id, name, or label.

`optgroup` Searches for an optgroup by its label.

`option` Searches for an option by its content or value.

`fieldset` Searches for a fieldset by its id or legend.

`table` Searches for a table by its id or caption.

Custom Selector

Mink lets you register your own selector types through implementing the `Behat\Mink\Selector\SelectorInterface`. It should then be registered in the `SelectorsHandler` which is the registry of available selectors.

The recommended way to register a custom selector is to do it when building your `Session`:

```
$selector = new \App\MySelector();

$handler = new \Behat\Mink\Selector\SelectorsHandler();
$handler->registerSelector('mine', $selector);

$driver = // ...

$session = new \Behat\Mink\Session($driver, $handler);
```

Manipulating Pages

Once you *get a page element*, you will want to manipulate it. You can also interact with the page, which is covered in the *next chapter*.

Getting the tag name

The `NodeElement::getTagName` method allows you to get the tag name of the element. This tag name is always returned lowercased.

```
$el = $page->find('css', '.something');

// get tag name:
echo $el->getTagName(); // displays 'a'
```

Accessing HTML attributes

The `NodeElement` class gives you access to HTML attributes of the element.

`NodeElement::hasAttribute` Checks whether the element has a given attribute.

`NodeElement::getAttribute` Gets the value of an attribute.

`NodeElement::hasClass` Checks whether the element has the given class (convenience wrapper around `getAttribute('class')`).

```
$el = $page->find('css', '.something');

if ($el->hasAttribute('href')) {
    echo $el->getAttribute('href');
} else {
    echo 'This anchor is not a link. It does not have an href.';
}
```

Element Content and Text

The `Element` class provides access to the content of elements.

`Element::getHtml` Gets the inner HTML of the element, i.e. all children of the element.

`Element::getOuterHtml` Gets the outer HTML of the element, i.e. including the element itself.

`Element::getText` Gets the text of the element.

Note: `getText()` will strip tags and unprinted characters out of the response, including newlines. So it'll basically return the text, that user sees on the page.

Checking Element Visibility

The `NodeElement::isVisible` methods allows to checks whether the element is visible.

Accessing Form State

The `NodeElement` class allows to access the state of form elements:

`NodeElement::getValue` Gets the value of the element. See *Interacting with Forms*.

`NodeElement::isChecked` Checks whether the checkbox or radio button is checked.

NodeElement::isSelected Checks whether the `<option>` element is selected.

Shortcut methods

The `TraversableElement` class provides a few shortcut methods allowing to find a child element in the page and checks the state of it immediately:

TraversableElement::hasCheckedField Looks for a checkbox (see findField) and checks whether it is checked.

TraversableElement::hasUncheckedField Looks for a checkbox (see findField) and checks whether it is not checked.

Interacting with Pages

Most usages of Mink will involve working with the page opened in your browser. The Mink Element API lets you interact with elements of the page.

Interacting with Links and Buttons

The `NodeElement::click` and `NodeElement::press` methods let you click the links and press the buttons on the page.

Note: These methods are actually equivalent internally (pressing a button involves clicking on it). Having both methods allows to keep the code more readable.

Interacting with Forms

The `NodeElement` class has a set of methods allowing to interact with forms:

NodeElement::getValue gets the value of a form field. The value depends on the type of field:

- the value of the selected option for single select boxes (or `null` when none are selected);
- an array of selected option values for multiple select boxes;
- the value of the checkbox field when checked, or `null` when not checked;
- the value of the selected radio button in the radio group for radio buttons;
- the value of the field for textual fields and textareas;
- an undefined value for file fields (because of browser limitations).

NodeElement::setValue sets the value of a form field

- for a file field, it should be the absolute path to the file;
- for a checkbox, it should be a boolean indicating whether it is checked;
- for other fields, it should match the behavior of `getValue`.

NodeElement::isChecked reports whether a radio button or a checkbox is checked.

NodeElement::isSelected reports whether an `<option>` element is selected.

NodeElement::check checks a checkbox field.

NodeElement::uncheck unchecks a checkbox field.

NodeElement::selectOption select an option in a select box or in a radio group.

NodeElement::attachFile attaches a file in a file input.

NodeElement::submit submits the form.

Interacting with the Mouse

The `NodeElement` class offers a set of methods allowing to interact with the mouse:

NodeElement::click performs a click on the element.

NodeElement::doubleClick performs a double click on the element.

NodeElement::rightClick performs a right click on the element.

NodeElement::mouseOver moves the mouse over the element.

Interacting with the Keyboard

Mink lets you interact with the keyboard thanks to the `NodeElement::keyDown`, `NodeElement::keyPress` and `NodeElement::keyUp` methods.

Manipulating the Focus

The `NodeElement` class lets you give and remove focus on the element thanks to the `NodeElement::focus` and `NodeElement::blur` methods.

Drag'n'Drop

Mink supports drag'n'drop of one element onto another:

```
$dragged = $page->find(...);
$target = $page->find(...);

$dragged->dragTo($target);
```

Shortcut Methods

The `TraversableElement` class provides a few shortcut methods allowing to find a child element on the page and perform an action on it immediately:

TraversableElement::clickLink Looks for a link (see findLink) and clicks on it.

TraversableElement::pressButton Looks for a button (see findButton) and presses on it.

TraversableElement::fillField Looks for a field (see findField) and sets a value in it.

TraversableElement::checkField Looks for a checkbox (see findField) and checks it.

TraversableElement::uncheckField Looks for a checkbox (see findField) and unchecks it.

TraversableElement::selectFieldOption Looks for a select or radio group (see findField) and selects a choice in it.

TraversableElement::attachFileToField Looks for a file field (see findField) and attach a file to it.

Note: All these shortcut methods are throwing an `ElementNotFoundException` in case the child element cannot be found.

Drivers

How does Mink provide a consistent API for very different browser library types, often written in different languages? Through drivers! A Mink driver is a simple class, that implements `Behat\Mink\Driver\DriverInterface`. This interface describes bridge methods between Mink and real browser emulators. Mink always talks with browser emulators through its driver. It doesn't know anything about how to start/stop or traverse page in that particular browser emulator. It only knows what driver method it should call in order to do this.

Mink comes with six drivers out of the box:

GoutteDriver

GoutteDriver provides a bridge for the Goutte headless browser. Goutte is a classical pure-php headless browser, written by the creator of the Symfony framework Fabien Potencier.

Note: The GoutteDriver extends the *BrowserKitDriver* to fix a small edge case in the Goutte implementation of BrowserKit. It is also able to instantiate the Goutte client automatically.

Installation

GoutteDriver is a pure PHP library available through Composer:

```
$ composer require behat/mink-goutte-driver
```

Note: GoutteDriver is compatible with both Goutte 1.x which relies on Guzzle 3 and Goutte 2.x which relies on Guzzle 4+ for the underlying HTTP implementation.

Composer will probably select Goutte 2.x by default.

Usage

In order to talk with Goutte, you should instantiate a `Behat\Mink\Driver\GoutteDriver`:

```
$driver = new \Behat\Mink\Driver\GoutteDriver();
```

Also, if you want to configure Goutte more precisely, you could do the full setup by hand:

```php
$client = new \Goutte\Client();
// Do more configuration for the Goutte client

$driver = new \Behat\Mink\Driver\GoutteDriver($client);
```

BrowserKitDriver

BrowserKitDriver provides a bridge for the Symfony BrowserKit component. BrowserKit is a browser emulator provided by the Symfony project.

Installation

BrowserKitDriver is a pure PHP library available through Composer:

```
$ composer require behat/mink-browserkit-driver
```

Note: The BrowserKit component only provides an abstract implementation. The actual implementation are provided by other projects, like Goutte or the Symfony HttpKernel component.

If you are using Goutte, you should use the special *GoutteDriver* which ensures full compatibility for Goutte due to an edge case in Goutte.

Usage

In order to talk with BrowserKit, you should instantiate a `Behat\Mink\Driver\BrowserKitDriver`:

```php
$browserkitClient = // ...

$driver = new \Behat\Mink\Driver\BrowserKitDriver($browserkitClient);
```

Selenium2Driver

Selenium2Driver provides a bridge for the Selenium2 (webdriver) tool. If you just love Selenium2, you can now use it right out of the box too.

Installation

Selenium2Driver is available through Composer:

```
$ composer require behat/mink-selenium2-driver
```

In order to talk with selenium server, you should install and configure it first:

1. Download the Selenium Server from the project website.
2. Run the server with the following command (update the version number to the one you downloaded):

   ```
   $ java -jar selenium-server-standalone-2.44.0.jar
   ```

Tip: The Selenium2Driver actually relies on the WebDriver protocol defined by Selenium2. This means that it is possible to use it with other implementations of the protocol. Note however that other implementations may have some bugs.

The testsuite of the driver is run against the Phantom.js implementation but it still triggers some failures because of bugs in their implementation.

Usage

That's it, now you can use Selenium2Driver:

```
$driver = new \Behat\Mink\Driver\Selenium2Driver('firefox');
```

ZombieDriver

ZombieDriver provides a bridge for the Zombie.js browser emulator. Zombie.js is a headless browser emulator, written in node.js. It supports all JS interactions that *Selenium* and *Sahi* do and works almost as fast as Goutte does. It is the best of both worlds actually, but still limited to only one browser type (Webkit). Also it is still slower than Goutte and requires node.js and npm to be installed on the system.

Installation

ZombieDriver is available through Composer:

```
$ composer require behat/mink-zombie-driver
```

In order to talk with a zombie.js server, you need to install and configure zombie.js first:

1. Install node.js by following instructions from the official site: http://nodejs.org/.
2. Install npm (node package manager) by following the instructions from http://npmjs.org/.
3. Install zombie.js with npm:

   ```
   $ npm install -g zombie
   ```

After installing npm and zombie.js, you'll need to add npm libs to your `NODE_PATH`. The easiest way to do this is to add:

```
export NODE_PATH="/PATH/TO/NPM/node_modules"
```

into your `.bashrc`.

Usage

After that, you'll be able to just use ZombieDriver without manual server setup. The driver will do all that for you automatically:

```
$driver = new \Behat\Mink\Driver\ZombieDriver(
    new \Behat\Mink\Driver\NodeJS\Server\ZombieServer()
);
```

If you want more control during driver initialization, like for example if you want to configure the driver to init the server on a specific port, use the more verbose version:

```
$driver = new \Behat\Mink\Driver\ZombieDriver(
    new \Behat\Mink\Driver\Zombie\Server($host, $port, $nodeBin, $script)
);
```

Note: `$host` simply defines the host on which zombie.js will be started. It's `127.0.0.1` by default.

`$port` defines a zombie.js port. Default one is `8124`.

`$nodeBin` defines full path to node.js binary. Default one is just `node`.

`$script` defines a node.js script to start zombie.js server. If you pass a `null` the default script will be used. Use this option carefully!

SahiDriver

SahiDriver provides a bridge for the Sahi browser controller. Sahi is a new JS browser controller, that fast replaced old Selenium testing suite. It's both easier to setup and to use than classical Selenium. It has a GUI installer for each popular operating system out there and is able to control every systems browser through a special bundled proxy server.

Installation

SahiDriver is available through Composer:

```
$ composer require behat/mink-sahi-driver
```

In order to talk with a real browser through Sahi, you should install and configure Sahi first:

1. Download and run the Sahi jar from the Sahi project website and run it. It will run the installer, which will guide you through the installation process.
2. Run Sahi proxy before your test suites (you can start this proxy during system startup):

```
cd $YOUR_PATH_TO_SAHI/bin
./sahi.sh
```

Usage

After installing Sahi and running the Sahi proxy server, you will be able to control it with `Behat\Mink\Driver\SahiDriver`:

```
$driver = new \Behat\Mink\Driver\SahiDriver('firefox');
```

Note: Notice, that first argument of `SahiDriver` is always a browser name, supported by Sahi.

If you want more control during the driver initialization, like for example if you want to configure the driver to talk with a proxy on another machine, use the more verbose version with a second client argument:

```
$driver = new \Behat\Mink\Driver\SahiDriver(
    'firefox',
    new \Behat\SahiClient\Client(
        new \Behat\SahiClient\Connection($sid, $host, $port)
    )
);
```

Note: `$sid` is a Sahi session ID. It's a unique string, used by the driver and the Sahi proxy in order to be able to talk with each other. You should fill this with `null` if you want Sahi to start your browser automatically or with some unique string if you want to control an already started browser.

`$host` simply defines the host on which Sahi is started. It is `localhost` by default.

`$port` defines a Sahi proxy port. The default one is `9999`.

SeleniumDriver

SeleniumDriver provides a bridge for the famous Selenium tool. If you need legacy Selenium, you can use it right out of the box in your Mink test suites.

Caution: The SeleniumRC protocol used by this driver is deprecated and does not support all Mink features. For this reason, the SeleniumDriver is deprecated in favor of the *Selenium2Driver*, which is based on the new protocol and is more powerful.

Installation

SeleniumDriver is available through Composer:

```
$ composer require behat/mink-selenium-driver
```

In order to talk with the selenium server, you should install and configure it first:

1. Download the Selenium Server from the project website.
2. Run the server with the following command (update the version number to the one you downloaded):

```
$ java -jar selenium-server-standalone-2.44.0.jar
```

Usage

That's it, now you can use SeleniumDriver:

```
$client = new \Selenium\Client($host, $port);
$driver = new \Behat\Mink\Driver\SeleniumDriver(
    'firefox', 'base_url', $client
);
```

Driver Feature Support

Although Mink does its best on removing browser differences between different browser emulators, it can't do much in some cases. For example, BrowserKitDriver cannot evaluate JavaScript and Selenium2Driver cannot get the response status code. In such cases, the driver will always throw a meaningful `Behat\Mink\Exception\UnsupportedDriverActionException`.

Feature	BrowserKit/Goutte	Selenium2	Zombie	Selenium	Sahi
Page traversing	Yes	Yes	Yes	Yes	Yes
Form manipulation	Yes	Yes	Yes	Yes	Yes
HTTP Basic auth	Yes	No	Yes	No	No
Windows management	No	Yes	No	Yes	Yes
iFrames management	No	Yes	No	Yes	No
Request headers access	Yes	No	Yes	No	No
Response headers	Yes	No	Yes	No	No
Cookie manipulation	Yes	Yes	Yes	Yes	Yes
Status code access	Yes	No	Yes	No	No
Mouse manipulation	No	Yes	Yes	Yes	Yes
Drag'n Drop	No	Yes	No	Yes	Yes
Keyboard actions	No	Yes	Yes	Yes	Yes
Element visibility	No	Yes	No	Yes	Yes
JS evaluation	No	Yes	Yes	Yes	Yes
Window resizing	No	Yes	No	No	No
Window maximizing	No	Yes	No	Yes	No

Managing Sessions

Although the *session object* is already usable enough, it's not as easy to write multisession (multidriver/multibrowser) code. Yep, you've heard right, with Mink you can manipulate multiple browser emulators simultaneously with a single consistent API:

```
// init sessions
$session1 = new \Behat\Mink\Session($driver1);
$session2 = new \Behat\Mink\Session($driver2);

// start sessions
$session1->start();
$session2->start();

$session1->visit('http://my_project.dev/chat.php');
$session2->visit('http://my_project.dev/chat.php');
```

Caution: The state of a session is actually managed by the driver. This means that each session must use a different driver instance.

Isn't it cool? But Mink makes it even cooler:

```
$mink = new \Behat\Mink\Mink();
$mink->registerSession('goutte', $goutteSession);
$mink->registerSession('sahi', $sahiSession);
$mink->setDefaultSessionName('goutte');
```

With such configuration, you can talk with your sessions by name through one single container object:

```
$mink->getSession('goutte')->visit('http://my_project.dev/chat.php');
$mink->getSession('sahi')->visit('http://my_project.dev/chat.php');
```

Note: Mink will even lazy-start your sessions when needed (on first `getSession()` call). So, the browser will not be started until you really need it!

Or you could even omit the session name in default cases:

```
$mink->getSession()->visit('http://my_project.dev/chat.php');
```

This call is possible thanks to `$mink->setDefaultSessionName('goutte')` setting previously. We've set the default session, that would be returned on `getSession()` call without arguments.

Tip: The `Behat\Mink\Mink` class also provides an easy way to reset or restart your started sessions (and only started ones):

```
// reset started sessions
$mink->resetSessions();

// restart started sessions
$mink->restartSessions();
```

Contributing

Reporting issues

If your issue affects only a single driver, it should be reported to the driver itself. Any other issues in the Mink abstraction layer and feature requests should be reported to the main Mink repository.

Contributing features

Contributions should be sent using GitHub pull requests.

Any contribution should be covered by tests to be accepted. Changes impacting drivers should include tests in the common driver testsuite to ensure consistency between implementations.

New features should always be contributed to the main Mink repository first. Features submitted only to a single driver without being part of the Mink abstraction won't be accepted.

CHAPTER 3

Testing Tools Integration

Mink has integrations for several testing tools:

- Behat through the Behat MinkExtension
- PHPUnit through the phpunit-mink package