The Book Of Geb

Luke Daley, Marcin Erdmann, Erik Pragt
version 1.1.1
Very groovy browser automation… web testing, screen scraping and more
Table of Contents

1. Introduction

Geb is a developer focused tool for automating the interaction between web browsers and web content. It uses the dynamic language features of Groovy to provide a powerful content definition DSL (for modelling content for reuse) and key concepts from jQuery to provide a powerful content inspection and traversal API (for finding and interacting with content).

Geb was born out of a desire to make browser automation (originally for web testing) easier and more productive. It aims to be a **developer tool** in that it allows and encourages the using of programming and language constructs instead of creating a restricted environment. It uses Groovy's dynamism to remove the noise and boiler plate code in order to focus on what's important — the content and interaction.

1.1. The browser automation technology

Geb builds on the WebDriver browser automation library, which means that Geb can work with any browser that WebDriver can. While Geb provides an extra layer of convenience and productivity, it is always possible to "drop down" to the WebDriver level to do something directly should you need to.

For more information see the manual section on using a driver implementation.

1.2. The Page Object pattern

The Page Object Pattern gives us a common sense way to model content in a reusable and maintainable way. From the WebDriver wiki page on the Page Object Pattern:

> Within your web app's UI there are areas that your tests interact with. A Page Object simply models these as objects within the test code. This reduces the amount of duplicated code and means that if the UI changes, the fix need only be applied in one place.

Furthermore (from the same document):

> PageObjects can be thought of as facing in two directions simultaneously. Facing towards the developer of a test, they represent the services offered by a particular page. Facing away from the developer, they should be the only thing that has a deep knowledge of the structure of the HTML of a page (or part of a page) It's simplest to think of the methods on a Page Object as offering the "services" that a page offers rather than exposing the details and mechanics of the page. As an example, think of the inbox of any web-based email system. Amongst the services that it offers are typically the ability to compose a new email, to choose to read a single email, and to list the subject lines of the emails in the inbox. How these are implemented shouldn't matter to the test.

The Page Object Pattern is an important technique, and Geb provides first class support via its page and module constructs.

1.3. The jQuery-ish navigator API

The jQuery JavaScript library provides an excellent API for (among other things) selecting or targeting content on a page and traversing through and around content. Geb takes a lot of inspiration from this.

In Geb, content is selected through the `$` function, which returns a `Navigator` object. A `Navigator` object is in someways analogous to the `jQuery` data type in jQuery in that it represents one or more targeted elements on the page.

Let's see some examples:

```
$("div") (1)

$("div", 0) (2)

$("div", title: "section") (3)

$("div", 0, title: "section") (4)

$("div.main") (5)

$("div.main", 0) (6)
```

1 Match all `"div"` elements on the page.
2 Match the first `"div"` element on the page.
3 Match all `"div"` elements with a title attribute value of `"section"`.
4 Match the first "div" element with a title attribute value of `"section"`.
5 Match all `"div"` elements who have the class `"main"`.
6 Match the first `"div"` element with the class `"main"`.

These methods return `Navigator` objects that can be used to further refine the content.

```
$("p", 0).parent() (1)

$("p").find("table", cellspacing: '0') (2)
```

1 The parent of the first paragraph.
2 All tables with a cellspacing attribute value of 0 that are nested in a paragraph.

This is just the beginning of what is possible with the Navigator API. See the chapter on the navigator for more details.

1.4. Full examples

Let's have a look at a simple case of going to Geb's home page and opening the highlights page for jQuery-like API.

1.4.1. Inline scripting

Here's an example of using Geb in an inline (i.e. no page objects or predefined content) scripting style…

```
import geb.Browser

Browser.drive {
    go "http://gebish.org"

    assert title == "Geb - Very Groovy Browser Automation" (1)

    $("#sidebar .sidemenu a", text: "jQuery-like API").click()(2)

    assert $("#main h1")*.text() == ["Navigating Content", "Form Control Shortcuts"] (3)
    assert $("#sidebar .sidemenu a", text: "jQuery-like API").parent().hasClass("selected")(4)
}
```

1 Check that we are at Geb's homepage.
2 Click on the link for information on jQuery-like API.
3 Check that section titles are as expected for the page about jQuery-like API.
4 Ensure that the previously clicked link is shown as the selected one.

1.4.2. Scripting with Page Objects

This time let us define our content up front using the Page Object pattern…

```
import geb.Module
import geb.Page

class SelectableLinkModule extends Module { (1)
    boolean isSelected() { (2)
        parent().hasClass("selected")
    }
}

class HighlightsModule extends Module {
    static content = { (3)
        highlightsLink { text -> $("a", text: text).module(SelectableLinkModule) }
        jQueryLikeApi { highlightsLink("jQuery-like API") } (4)
    }
}

class GebHomePage extends Page {
    static url = "http://gebish.org" (5)

    static at = { title == "Geb - Very Groovy Browser Automation" } (6)

    static content = {
        highlights { $("#sidebar .sidemenu").module(HighlightsModule) } (7)
        sectionTitles { $("#main h1")*.text() } (8)
    }
}
```

1 Modules are reusable fragments that can be used across pages. Here we are using a module to model a link that can be shown as selected.
2 Modules can contain methods that allow to hide document structure details.
3 Content DSL.
4 Content definitions can use other, more general content definitions to define more specific content elements.
5 Pages can define their location, either absolute or relative to a base.
6 "at checkers" allow verifying that the browser is at the expected page.
7 Include the previously defined module using a base.
8 Content definitions can not only return document elements but also other types. Here we return a list of section titles.

Now our script again, using the above defined content…

```
import geb.Browser

Browser.drive {
```

```
    to GebHomePage (1)

    highlights.jQueryLikeApi.click()

    assert sectionTitles == ["Navigating Content", "Form Control Shortcuts"]
    assert highlights.jQueryLikeApi.selected
}
```

1 Go to the url defined by `GebHomePage` and also verify it's "at checker".

1.4.3. Testing

Geb itself does not include any kind of testing or execution framework. Rather, it works with existing popular tools like Spock, JUnit, TestNg and Cucumber-JVM. While Geb works well with all of these test tools, we encourage the use of Spock as it's a great match for Geb with its focus and style.

Here is our Geb homepage case again, this time using Geb's Spock integration…

```
import geb.spock.GebSpec

class GebHomepageSpec extends GebSpec {

    def setup() {
        browser.driver.javascriptEnabled = false
    }

    def cleanup() {
        CachingDriverFactory.clearCache()
    }

    def "can access The Book of Geb via homepage"() {
        when:
        to GebHomePage

        and:
        highlights.jQueryLikeApi.click()

        then:
        sectionTitles == ["Navigating Content", "Form Control Shortcuts"]
        highlights.jQueryLikeApi.selected
    }
}
```

For more information on using Geb for web and functional testing, see the testing chapter.

1.5. Installation & usage

Geb itself is a available as a single geb-core jar from the central Maven repository. To get up and running you simply need this jar, a WebDriver driver implementation and the `selenium-support` jar.

Via `@Grab`…

```
@Grapes([
    @Grab("org.gebish:geb-core:1.1.1"),
    @Grab("org.seleniumhq.selenium:selenium-firefox-driver:2.52.0"),
    @Grab("org.seleniumhq.selenium:selenium-support:2.52.0")
])
import geb.Browser
```

Via Maven…

```
<dependency>
  <groupId>org.gebish</groupId>
  <artifactId>geb-core</artifactId>
  <version>1.1.1</version>
</dependency>
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-firefox-driver</artifactId>
  <version>2.52.0</version>
</dependency>
<dependency>
```

```
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-support</artifactId>
  <version>2.52.0</version>
</dependency>
```

Via Gradle…

```
compile "org.gebish:geb-core:1.1.1"
compile "org.seleniumhq.selenium:selenium-firefox-driver:2.52.0"
compile "org.seleniumhq.selenium:selenium-support:2.52.0"
```

Alternatively, if using an integration such as `geb-spock` or `geb-junit4` you can depend on that instead of `geb-core`. You can check out the listing of all artifacts in `org.gebish` group to see what's available.

Be sure to check the chapter on build integrations for information on using Geb with particular environments.

Snapshot repository

If you fancy living on the bleeding edge then you can try out Geb's snapshot artifacts located in the Maven repository at https://oss.sonatype.org/content/repositories/snapshots.

2. The `Browser`

The entry point to Geb is the `Browser` object. A browser object marries a `WebDriver` instance (which drives the actual web browser being automated) with the concept of a "current page".

Browser objects are created with a `Configuration` that specifies which driver implementation to use, the base URL to resolve relative links against and other bits of config. The configuration mechanism allows you to externalise how Geb should operate, which means you can use the same suite of Geb code or tests with different browsers or site instances. The chapter on configuration contains more details on how to manage the configuration parameters and what they are.

The default constructor of `Browser` simply loads its settings from the config mechanism.

```
import geb.Browser

def browser = new Browser()
```

However, if you prefer to specify the driver implementation (or any other settable property on the `Browser`) you can by using Groovy's map constructor syntax.

```
import geb.Browser
import org.openqa.selenium.htmlunit.HtmlUnitDriver

def browser = new Browser(driver: new HtmlUnitDriver())
```

Which is the same as…

```
def browser = new Browser()
browser.driver = new HtmlUnitDriver()
```

Any property set this way will **override** any settings coming from the config mechanism.

The behaviour is undefined if a browser's driver is changed after its first use, so you should avoid setting the driver this way and prefer the configuration mechanism.

For drastically custom configuration requirements, you can create your own `Configuration` object and construct the browser with it, likely using the configuration loader.

```
import geb.Browser
import geb.ConfigurationLoader

def loader = new ConfigurationLoader("a-custom-environment")
def browser = new Browser(loader.conf)
```

Wherever possible, you should strive to use the no-arg constructor and manage Geb through the inbuilt configuration mechanism as it offers a great deal of flexibility and separates your configuration from your code.

Geb integrations typically remove the need to construct a browser object and do this for you, leaving you to just manage the configuration.

2.1. The `drive()` method

The Browser class features a static method, `drive()`, that makes Geb scripting a little more convenient.

```
Browser.drive {
    go "signup"
    assert $("h1").text() == "Signup Page"
}
```

Which is equivalent to:

```
def browser = new Browser()
browser.go "signup"
assert browser.$("h1").text() == "Signup Page"
```

The `drive()` method takes all of the arguments that the `Browser` constructor takes (i.e. none, a `Configuration` and/or property overrides) or an existing browser instance, and a closure. The closure is evaluated against created browser instance (i.e. the browser is made the *delegate* of the closure). The net result is that all top level method calls and property accesses are implied to be against the browser.

The `drive()` method always returns the browser object that was used, so if you need to quit the browser after the drive session you can do something like…

```
Browser.drive {
    //…
}.quit()
```

For more on when/why you need to manually quit the browser, see the section on the driver.

2.2. Making requests

2.2.1. The base URL

Browser instances maintain a `baseUrl` property that is used to resolve all relative URLs. This value can come from configuration or can be explicitly set on the browser.

Care must be taken with slashes when specifying both the base URL and the relative URL as trailing and leading slashes have significant meaning. The following table illustrates the resolution of different types of URLs.

Base	Navigating To	Result
http://myapp.com/	abc	http://myapp.com/abc
http://myapp.com	abc	http://myapp.comabc
http://myapp.com	/abc	http://myapp.com/abc
http://myapp.com/abc/	def	http://myapp.com/abc/def
http://myapp.com/abc	def	http://myapp.com/def

Base	Navigating To	Result
http://myapp.com/abc/	/def	http://myapp.com/def
http://myapp.com/abc/def/	jkl	http://myapp.com/abc/def/jkl
http://myapp.com/abc/def	jkl	http://myapp.com/abc/jkl
http://myapp.com/abc/def	/jkl	http://myapp.com/jkl

It is usually most desirable to define your base urls with trailing slashes and not to use leading slashes on relative URLs.

2.2.2. Using pages

Page objects (discussed further shortly) can define a url that will be used when explicitly navigating to that page. This is done with the `to()` and `via()` methods.

```
class SignupPage extends Page {
    static url = "signup"
}

Browser.drive {
    to SignupPage
    assert $("h1").text() == "Signup Page"
    assert page instanceof SignupPage
}
```

The `to()` and `via()` method makes a request to the resolved URL and sets the browser's page instance to an instance of the given class. Most Geb scripts and tests start with a `to()` or `via()` call.

See the section on advanced page navigation for more information on how to use more complicated URL resolution for pages.

2.2.3. Direct

You can also make a new request to a URL without setting or changing the page using the `go()` methods.

```
import geb.Page
import geb.spock.GebSpec

class GoogleHomePage extends Page {
    static url = "http://www.google.com"
}

class GoogleSpec extends GebSpec {
    def "go method does NOT set the page"() {
        given:
        Page oldPage = page

        when:
        go "http://www.google.com"

        then:
        oldPage == page
        currentUrl.contains "google"
    }

    def "to method does set the page and change the current url"() {
        given:
        Page oldPage = page

        when:
        to GoogleHomePage

        then:
```

```
        oldPage != page
        currentUrl.contains "google"
    }
}
```

The following examples use a baseUrl of “`http://localhost/`”.

```
Browser.drive {
    go() (1)

    go "signup" (2)

    go "signup", param1: "value1", param2: "value2" (3)
}
```

1 Go to the base URL.

2 Go to a URL relative to base URL.

3 Go to a URL with request params i.e `http://localhost/signup?param1=value1&param2=value2`

2.3. The Page

Browser instances hold a reference to a *page*. This page instance is retrievable via the page property. Initially, all browser instances have a page of type Page which provides the basic navigation functions and is the superclass for all page objects.

However, the page property is rarely accessed directly. The browser object will *forward* any method calls or property read/writes that it can’t handle to the current page instance.

```
Browser.drive {
    go "signup"

    assert $("h1").text() == "Signup Page" (1)
    assert page.$("h1").text() == "Signup Page" (1)
}
```

1 These two calls are equivalent.

The *page* is providing the `$()` function, not the browser. This forwarding facilitates very concise code, void of unnecessary noise.

For more information on the `$()` function which is used to interact with page content, see the section on the Navigator API.

When using the Page Object pattern, you create subclasses of Page that define content via a powerful DSL that allows you to refer to content by meaningful names instead of tag names or CSS expressions.

```
class SignupPage extends Page {
    static url = "signup"

    static content = {
        heading { $("h1").text() }
    }
}

Browser.drive {
    to SignupPage
    assert heading == "Signup Page"
}
```

Page objects are discussed in depth in the Pages chapter, which also explores the Content DSL.

Changing the page

We have already seen that that `to()` methods change the browser’s page instance. It is also possible to change the page instance without initiating a new request with the `page()` methods.

The `<T extends Page> T page(Class<T> pageType)` method allows you to change the page to a new instance of **the given class**. The class must be Page or a subclass thereof. This method **does not** verify that the given page actually matches the content (at checking is discussed shortly).

The `<T extends Page> T page(T pageInstance)` method allows you to change the page to **the given instance**. Similarly to the method taking a page class it **does not** verify that the given page actually matches the content.

The `Page page(Class<? extends Page>[] potentialPageTypes)` method allows you to specify a number of *potential* page types. Each of the potential pages is instantiated and checked to see if it matches the content the browser is actually currently at by running each page's at checker. All of the page classes passed in must have an "at" checker defined otherwise an `UndefinedAtCheckerException` will be thrown.

The `Page page(Page[] potentialPageInstances)` method allows you to specify a number of *potential* page instances. Each of the potential page instances is initialized and checked to see if it matches the content the browser is actually currently at by running each pages at checker. All of the page instances passed in must have an "at" checker defined otherwise an `UndefinedAtCheckerException` will be thrown.

These two methods taking arrays as arguments are not typically used explicitly but are used by the `to()` method and content definitions that specify the page that the content navigates to when clicked (see the section on the `to` attribute of the Content DSL for more information about this). However, should you need to manually change the page type, they are there.

2.4. At checking

Pages define an "at checker" that the browser uses for checking if it is pointing at a given page.

```
class SignupPage extends Page {
    static url = "signup"

    static at = {
        $("h1").text() == "Signup Page"
    }
}

Browser.drive {
    to SignupPage
}
```

> Not using explicit `return` statements in "at checkers" is preferred. Geb transforms all "at checkers" so that each statement in them is asserted (just like for `then:` blocks in Spock specifications). Thanks to that you can immediately see evaluated values of your "at checker" if it fails. See the "at checker" section for more details.

The `to()` method that takes a single page type or instance **verifies** that the the browser ends up at the given page. If the request may initiate a redirect and take the browser to a different page you should use `via()` method:

```
Browser.drive {
    via SecurePage
    at AccessDeniedPage
}
```

Browser objects have `<T extends Page> T at(Class<T> pageType)` and `<T extends Page> T at(T page)` methods that test whether or not the browser is currently at the page modeled by the given page class or instance.

The `at AccessDeniedPage` method call return a page instance if the "at checker" is fulfilled. If on the other hand it is not then an `AssertionError` will be thrown even if there are no explicit assertions in the "at checker" (the default, see the section on Implicit assertions for details) or return null if implicit assertions are disabled.

It's always a good idea to either use the `to()` method which implicitly verifies the "at checker" or the `via()` method followed by an `at()` check whenever the page changes in order to *fail fast*. Otherwise, subsequent steps may fail in harder to diagnose ways due to the content not matching what is expected and content look-ups returning strange results.

If you pass a page class or instance that doesn't define an "at checker" to `at()` you will get an `UndefinedAtCheckerException` - "at checkers" are mandatory when doing explicit "at checks". This is not the case when implicit "at checks" are being performed, like when using `to()`. This behaviour is intended to make you aware that you probably want to define an "at checker" when explicitly verifying if you're at a given page but not forcing you to do so when using implicit "at checking".

The `at()` method will also update the browser's page instance if the "at checker" is successful. This means that you don't have to manually set browser's page to the new one after "at checking".

Pages can also define content that declares what the browser's page type should change to when that content is clicked. After clicking on such content the declared page is automatically "at checked" if it defines an "at checker" (see the content DSL reference for the `to`

parameter).

```
class LoginPage extends Page {
    static url = "/login"

    static content = {
        loginButton(to: AdminPage) { $("input", type: "submit") }
        username { $("input", name: "username") }
        password { $("input", name: "password") }
    }
}

class AdminPage extends Page {
    static at = {
        $("h1").text() == "Admin Page"
    }
}

Browser.drive {
    to LoginPage

    username.value("admin")
    password.value("p4sw0rd")
    loginButton.click()

    assert page instanceof AdminPage
}
```

2.5. Page change listening

It is possible to be notified when a browser's page *instance* changes (note that this is not necessarily when the browser makes a request to a new URL) using the `PageChangeListener` interface.

As soon as a listener is registered, its `pageWillChange()` method will be called with `newPage` as the current page and `oldPage` as `null`. Subsequently, each time the page changes `oldPage` will be the page that the browser currently has, and `newPage` will be the page that will soon be the browser's page.

```
class RecordingPageChangeListener implements PageChangeListener {
    Page oldPage
    Page newPage

    @Override
    void pageWillChange(Browser browser, Page oldPage, Page newPage) {
        this.oldPage = oldPage
        this.newPage = newPage
    }
}

class FirstPage extends Page {
}

class SecondPage extends Page {
}

def listener = new RecordingPageChangeListener()
def browser = new Browser()

browser.page(FirstPage)
browser.registerPageChangeListener(listener)

assert listener.oldPage == null
assert listener.newPage instanceof FirstPage

browser.page(SecondPage)

assert listener.oldPage instanceof FirstPage
assert listener.newPage instanceof SecondPage
```

You can remove a listener at any time…

```
browser.removePageChangeListener(listener)
```

The `removePageChangeListener(PageChangeListener listener)` returns `true` if `listener` was registered and has now been removed, otherwise it returns `false`.

Listeners cannot be registered twice. If an attempt is made to register a listener that is already registered (i.e. there is another listener that is *equal* to the listener being registered, based on their `equals()` implementation) then a `PageChangeListenerAlreadyRegisteredException` will be raised.

2.6. Working with multiple tabs and windows

When you're working with an application that opens new windows or tabs, for example when clicking on a link with a target attribute set, you can use `withWindow()` and `withNewWindow()` methods to execute code in the context of other windows.

If you really need to know the name of the current window or all the names of open windows use `Browser.getCurrentWindow()` and `Browser.getAvailableWindows()` methods but `withWindow()` and `withNewWindow()` are the preferred methods when it comes to dealing with multiple windows.

2.6.1. Already opened windows

If you know the name of the window in context of which you want to execute the code you can use `withWindow(String windowName, Closure block)`.

Given that this HTML is rendered for the `baseUrl`:

```
<a href="http://www.gebish.org" target="myWindow">Geb</a>
```

This code passes:

```
Browser.drive {
    go()
    $("a").click()
    withWindow("myWindow") {
        assert title == "Geb - Very Groovy Browser Automation"
    }
}
```

If you don't know the name of the window but you know something about the content of the window you can use the `withWindow(Closure specification, Closure block)` method. The first closure passed should return true for the window, or windows, you want to use as context for execution of the second closure.

If there is no window for which the window specification closure returns `true` then `NoSuchWindowException` is thrown.

So given:

```
<a href="http://www.gebish.org" target="_blank">Geb</a>
```

This code passes:

```
Browser.drive {
    go()
    $("a").click()
    withWindow({ title == "Geb - Very Groovy Browser Automation" }) {
        assert $("#slogan").text().startsWith("very groovy browser automation…")
    }
}
```

If code of the closure passed as the last argument changes browser's current page instance (e.g. by using `page(Page)` or `at(Page)`) then it will be reverted to its original value after returning from `withWindow()`.

`withWindow()` options

There are some additional options that can be passed to a `withWindow()` call which make working with already opened windows even simpler. The general syntax is:

```
withWindow(«window specification», «option name»: «option value», ...) {
    «action executed within the context of the window»
}
```

close

Default value: `false`

If you pass any *truthy* value as the `close` option then all matching windows will be closed after the execution of the closure passed as the last argument to a `withWindow()` call.

page

Default value: `null`

If you pass a class or an instance of a class that extends `Page` as `page` option, then browser's page will be set to that value before executing the closure passed as the last argument and will be reverted to its original value afterwards. If the page class defines an "at checker" then it will be verified when the page is set on the browser.

2.6.2. Newly opened windows

If you wish to execute code in a window that is newly opened by some of your actions, use the `withNewWindow(Closure windowOpeningBlock, Closure block)` method. Given that this HTML is rendered for the `baseUrl`:

```
<a href="http://www.gebish.org" target="_blank">Geb</a>
```

The following will pass:

```
Browser.drive {
    go()
    withNewWindow({ $('a').click() }) {
        assert title == 'Geb - Very Groovy Browser Automation'
    }
}
```

If the first parameter opens none or more than one window, then `NoNewWindowException` is thrown.

If code of the closure passed as the last argument changes browser's current page instance (e.g. by using `page(Page)` or `at(Page)`) then it will be reverted to its original value after returning from `withNewWindow()`.

withNewWindow() options

There are several options that can be passed to a `withNewWindow()` call which make working with newly opened windows even simpler. The general syntax is:

```
withNewWindow({ «window opening action» }, «option name»: «option value», ...) {
    «action executed within the context of the window»
}
```

close

Default value: `true`

If you pass any *truthy* value as `close` option then the newly opened window will be closed after the execution of the closure passed as the last argument to the `withNewWindow()` call.

page

Default value: `null`

If you pass a class or an instance of a class that extends `Page` as `page` option then browser's page will be set to that value before executing the closure passed as the last argument and will be reverted to its original value afterwards.

`wait`

Default value: `null`

You can specify `wait` option if the action defined in the window opening closure passed as the first argument is asynchronous and you need to wait for the new window to be opened. The possible values for the `wait` option are consistent with the ones for `wait` option of content definitions.

Given that the following HTML is rendered for the `baseUrl`:

```
<a href="http://google.com" target="_blank" id="new-window-link">Google</a>
```

together with the following javascript:

```
function openNewWindow() {
    setTimeout(function() {
        document.getElementById('new-window-link').click();
    }, 200);
}
```

then the following will pass:

```
Browser.drive {
    go()
    withNewWindow({ js.openNewWindow() }, wait: true) {
        assert title == 'Google'
    }
}
```

2.7. Quitting the browser

The browser object has `quit()` and `close()` methods (that simply delegate to the underlying driver). See the section on driver management for more information on when and why you need to quit the browser.

3. The WebDriver implementation

A `Browser` instance interacts with an actual browser via an instance of `WebDriver`. The browser's driver can always be retrieved via the `getDriver()` method.

> One of the key design principles that WebDriver embraces is that tests/scripts should be written to the `WebDriver` API making them agnostic to the actual browser being driven, and therefore portable. Geb always supports this goal. However, the reality is that there are still quirks and behavioural differences between driver implementations. Each release of WebDriver has historically worked to minimise these issues so expect the situation to improve over time as WebDriver matures.

3.1. Explicit driver management

One option for specifying the driver implementation is to construct the driver instance and pass it to the `Browser` to be used when it is constructed.

However, where possible prefer implicit driver management which is discussed later in this chapter.

Explicit lifecycle

When the driver is constructed by the user, the user is responsible for quitting the driver at the appropriate time. This can be done via the methods on the `WebDriver` instance (obtainable via `getDriver()`) or by calling the delegating methods on the browser object.

3.2. Implicit driver management

If a driver is not given when a `Browser` object is constructed, one will be created and managed implicitly by Geb by the the configuration mechanism.

Implicit lifecycle

By default, Geb internally caches and reuses the first driver created, meaning that all subsequent browser instances created without an explicit driver will reuse the cached driver. This avoids the overhead of creating a new driver each time, which can be significant when working with a real browser.

This means that you may need to call the `clearCookies()` or `clearCookies(String… additionalUrls)` method on the browser in order not to get strange results due to cookies from previous executions.

> Some of the integrations (e.g. Spock, JUnit) automatically clear the browser cookies for the current domain at appropriate times such as after each test. Consult the section on testing for specifics.

The shared driver will be closed and quited when the JVM shuts down.

A new driver can be forced at anytime by calling either of `CachingDriverFactory.clearCache()` or `CachingDriverFactory.clearCacheAndQuitDriver()` both of which are `static`. After calling any of these methods, the next request for a default driver will result in a new driver instance being created.

This caching behavior is configurable.

3.3. Driver quirks

This section details various quirks or issues that have been encountered with different driver implementations.

`HTMLUnitDriver`

Dealing with pages that use HTML refreshes

The default behaviour of the HTMLUnit driver is to immediately refresh the page as soon as it encounters a `<meta http-equiv="refresh" content="5">` regardless of the specified time. The solution is to use a refresh handler that handles the refresh asynchronously.

```
import com.gargoylesoftware.htmlunit.ThreadedRefreshHandler

Browser.drive {
    driver.webClient.refreshHandler = new ThreadedRefreshHandler()
    (1)
}
```

1 From here on refresh meta tag value will be respected.

See this mailing list thread for details.

Configuring logging

HTMLUnit can be very noisy, and it's not clear how to make it not so noisy.

See this issue for some tips on how to tune its logging.

4. Interacting with content

Geb provides a concise and Groovy interface to the content and controls in your browser. This is implemented through the `Navigator` API which is a jQuery inspired mechanism for finding, filtering and interacting with DOM elements.

4.1. The `$()` function

The `$()` function is the access point to the browser's page content. It returns a `Navigator` object that is roughly analogous to a jQuery object. It is analogous in that it represents one or more elements on the page and can be used to refine the matched content or query the matched content. When a `$()` function is called that does not match any content, an "empty" navigator object is returned that represents

no content. Operations on “empty” navigators return `null` or another “empty” navigator or other values that make sense (e.g. the `size()` method returns `0`).

The signature of the `$()` function is as follows…

```
$(«css selector», «index or range», «attribute / text matchers»)
```

The following is a concrete example…

```
$("h1", 2, class: "heading")
```

This would find the 3rd (elements are 0 indexed) `h1` element whose `class` *attribute* is exactly “`heading`”.

All arguments are optional, meaning the following calls are all valid:

```
$("div p", 0)
$("div p", title: "something")
$(0)
$(title: "something")
```

There is an alias for the dollar function named “find” if a method named “$” is not to your taste.

4.1.1. CSS Selectors

You can use any CSS selector that the underlying `WebDriver` supports…

```
$('div.some-class p:first-child[title^="someth"]')
```

HTMLUnit driver only partially supports CSS 3. A future version of the HTMLUnit driver may gain better CSS 3 support.

4.1.2. Using WebDriver’s `By` class selectors

For all signatures of `$()` function that accept a css selector there is an equivalent signature where an instance of WebDriver’s `By` class can be used instead of a `String`.

Using CSS selectors is the idiomatic way of using Geb and should be preferred to using `By` selectors. It is always possible to select the same elements using a css selector as when using a certain `By` selector apart from certain XPath selectors which is why this convenience mechanism is provided.

Following are some examples of using `By` selectors…

```
$(By.id("some-id"))
$(By.className("some-class"))
$(By.xpath('//p[@class="xpath"]'))
```

4.1.3. Indexes and ranges

When matching, a single positive integer or integer range can be given to restrict by index.

Consider the following html…

```
<p>a</p>
<p>b</p>
<p>c</p>
```

We can use indexes to match content like so…

```
assert $("p", 0).text() == "a"
assert $("p", 2).text() == "c"
assert $("p", 0..1)*.text() == ["a", "b"]
assert $("p", 1..2)*.text() == ["b", "c"]
```

See below for an explanation of the `text()` method and the use of the spread operator.

4.1.4. Attribute and text matching

Matches can be made on attributes and node text values via Groovy's named parameter syntax. The value `text` is treated specially as a match against the node's text. All other values are matched against their corresponding attribute values.

Consider the following html…

```
<p attr1="a" attr2="b">p1</p>
<p attr1="a" attr2="c">p2</p>
```

We can use attribute matchers like so…

```
assert $("p", attr1: "a").size() == 2
assert $("p", attr2: "c").size() == 1
```

Attribute values are `and`ed together…

```
assert $("p", attr1: "a", attr2: "b").size() == 1
```

We can use text matchers like so…

```
assert $("p", text: "p1").size() == 1
```

You can mix attribute and text matchers…

```
assert $("p", text: "p1", attr1: "a").size() == 1
```

Using patterns

To match the entire value of an attribute or the text you use a `String` value. It is also possible to use a `Pattern` to do regexp matching…

```
assert $("p", text: ~/p./).size() == 2
```

Geb also ships with a bunch of shortcut pattern methods…

```
assert $("p", text: startsWith("p")).size() == 2
assert $("p", text: endsWith("2")).size() == 1
```

The following is the complete listing:

Case Sensitive	Case Insensitive	Description
`startsWith`	`iStartsWith`	Matches values that start with the given value
`contains`	`iContains`	Matches values that contain the given value anywhere
`endsWith`	`iEndsWith`	Matches values that end with the given value
`containsWord`	`iContainsWord`	Matches values that contain the given value surrounded by either whitespace or the beginning or end of the value
`notStartsWith`	`iNotStartsWith`	Matches values that DO NOT start with the given value
`notContains`	`iNotContains`	Matches values that DO NOT contain the given value anywhere

Case Sensitive	Case Insensitive	Description
`notEndsWith`	`iNotEndsWith`	Matches values that DO NOT end with the given value
`notContainsWord`	`iNotContainsWord`	Matches values that DO NOT contain the given value surrounded by either whitespace or the beginning or end of the value

All of these methods themselves can take a `String` or a `Pattern`…

```
assert $("p", text: contains(~/\d/)).size() == 2
```

You might be wondering how this magic works, i.e. where these methods come from and where they can be used. They are methods that are available on `geb.Page` and other *places* where you can use the $ function. They simply just return patterns.

4.1.5. Navigators are iterable

The navigator objects implement the Java `Iterable` interface, which allows you to do lots of Groovy stuff like use the `max()` function…

Consider the following html…

```
<p>1</p>
<p>2</p>
```

You can use the `max()` function on `Navigator` instances…

```
assert $("p").max { it.text() }.text() == "2"
```

This also means that navigator objects work with the Groovy spread operator…

```
assert $("p")*.text().max() == "2"
```

When treating a navigator as `Iterable`, the iterated over content is always the exact matched elements (as opposed to including children).

4.1.6. `equals()` and `hashCode()`

It's possible to check `Navigator` instances for equality. The rules are simple - two empty navigators are always equal and two non empty navigators are only equal if they contain the exact same elements in the same order.

Consider the following HTML…

```
<p class="a"></p>
<p class="b"></p>
```

Here are some examples of equal Navigator instances…

```
assert $("div") == $(".foo") (1)
assert $(".a") == $(".a") (2)
assert $(".a") == $("p").not(".b") (3)
assert $("p") == $("p") (4)
assert $("p") == $(".a").add(".b") (5)
```

1 Two empty navigators
2 Two single element navigators containing the same element
3 Two single element navigators containing the same element created using different methods
4 Two multi element navigators containing the same elements
5 Two multi element navigators containing the same elements created using different methods

And some that are not equal…

```
assert $("div") != $("p") (1)
assert $(".a") != $(".b") (2)
assert $(".a").add(".b") != $(".b").add(".a") (3)
```

1 Empty and not empty navigators
2 Single element navigators containing different elements
3 Multi element navigators containing the same elements but in a different order

4.2. Finding & filtering

Navigator objects have `find()` and `$()` methods for finding descendants, and `filter()` and `not()` methods for reducing the matched content.

Consider the following HTML…

```
<div class="a">
    <p class="b">geb</p>
</div>
<div class="b">
    <input type="text"/>
</div>
```

We can select `p.b` by…

```
$("div").find(".b")
$("div").$(".b")
```

We can select `div.b` by…

```
$("div").filter(".b")
```

or…

```
$(".b").not("p")
```

We can select the `div` containing the `p` with…

```
$("div").has("p")
```

Or select the `div` containing the `input` with a type attribute of "text" like so…

```
$("div").has("input", type: "text")
```

We can select the `div` that does not contain the `p` with…

```
$("div").hasNot("p")
```

Or select the `div` that does not contain the `input` with a type attribute of "text" like so…

```
$("div").hasNot("input", type: "text")
```

Or select the two `div` that do not contain `input` with a type attribute of "submit" like so…

```
$("div").hasNot("input", type: "submit")
```

The `find()` and `$()` methods support the **exact same argument types as the `$()` function**.

The `filter()`, `not()`, `has()` and `hasNot()` methods have the same signatures - they accept: a selector string, a predicates map or both.

These methods return a new navigator object that represents the new content.

4.3. Composition

It is also possible to compose navigator objects from other navigator objects, for situations where you can't express a content set in one query. To do this, simply call the `$()` function and pass in the navigators to compose.

Consider the following markup…

```
<p class="a">1</p>
<p class="b">2</p>
<p class="c">3</p>
```

You can then create a new navigator object that represents both the `a` and `b` paragraphs the following way:

```
assert $($("p.a"), $("p.b"))*.text() == ["1", "2"]
```

An alternative way is to use the `add()` method of `Navigator` that takes either a `String` or a Webdriver's `By` selector:

```
assert $("p.a").add("p.b").add(By.className("c"))*.text() == ["1", "2", "3"]
```

Finally, you can compose navigator objects from content. So given a page content definition:

```
static content = {
    pElement { pClass -> $('p', class: pClass) }
}
```

You can compose content elements into a navigator in the following way:

```
assert $(pElement("a"), pElement("b"))*.text() == ["1", "2"]
```

4.4. Traversing

Navigators also have methods for selecting content *around* the matched content.

Consider the following HTML…

```
<div class="a">
    <div class="b">
        <p class="c"></p>
        <p class="d"></p>
        <p class="e"></p>
    </div>
    <div class="f"></div>
</div>
```

You can select content *around* `p.d` by…

```
assert $("p.d").previous() == $("p.c")
assert $("p.e").prevAll() == $("p.c").add("p.d")
assert $("p.d").next() == $("p.e")
assert $("p.c").nextAll() == $("p.d").add("p.e")
assert $("p.d").parent() == $("div.b")
assert $("p.c").siblings() == $("p.d").add("p.e")
assert $("div.a").children() == $("div.b").add("div.f")
```

Consider the following HTML…

```
<div class="a">
    <p class="a"></p>
    <p class="b"></p>
    <p class="c"></p>
</div>
```

The following code will select `p.b` & `p.c`…

```
assert $("p").next() == $("p.b").add("p.c")
```

The `previous()`, `prevAll()`, `next()`, `nextAll()`, `parent()`, `parents()`, `closest()`, `siblings()` and `children()` methods can also take CSS selectors and attribute matchers.

Using the same html, the following examples will select `p.c`…

```
assert $("p").next(".c") == $("p.c").add("p.c")
assert $("p").next(class: "c") == $("p.c").add("p.c")
assert $("p").next("p", class: "c") == $("p.c").add("p.c")
```

Likewise, consider the following HTML…

```
<div class="a">
    <div class="b">
        <p></p>
    </div>
</div>
```

The following examples will select `div.b`…

```
assert $("p").parent(".b") == $("div.b")
assert $("p").parent(class: "b") == $("div.b")
assert $("p").parent("div", class: "b") == $("div.b")
```

The `closest()` method is a special case in that it will select the first ancestor of the current elements that matches a selector. There is no no-argument version of the `closest()` method. For example, these will select `div.a`…

```
assert $("p").closest(".a") == $("div.a")
assert $("p").closest(class: "a") == $("div.a")
assert $("p").closest("div", class: "a") == $("div.a")
```

These methods do not take indexes as they automatically select the first matching content. To select multiple elements you can use `prevAll()`, `nextAll()` and `parents()` all of which have no-argument versions and versions that filter by a selector.

The `nextUntil()`, `prevUntil()` and `parentsUntil()` methods return all nodes along the relevant axis *until* the first one that matches a selector or attributes. Consider the following markup:

```
<div class="a"></div>
<div class="b"></div>
<div class="c"></div>
<div class="d"></div>
```

The following examples will select `div.b` and `div.c`:

```
assert $(".a").nextUntil(".d") == $("div.b").add("div.c")
assert $(".a").nextUntil(class: "d") == $("div.b").add("div.c")
assert $(".a").nextUntil("div", class: "d") == $("div.b").add("div.c")
```

4.5. Clicking

Navigator objects implement the `click()` method, which will instruct the browser to click the sole element the navigator has matched. If that method is called on a multi element `Navigator` then a `SingleElementNavigatorOnlyMethodException` is thrown.

There are also `click(Class)` , `click(Page)` and `click(List)` methods that are analogous to the browser object's `page(Class<? extends Page>), page(Page), and page(Class<? extends Page>[]), page(Page[])` methods respectively. This allow page changes to be specified at the same time as click actions.

For example…

```
$("a.login").click(LoginPage)
```

Would click the `a.login` element, then effectively call `browser.page(LoginPage)` and verify that the browser is at the expected page.

All of the page classes passed in when using the list variant have to have an "at" checker defined, otherwise an `UndefinedAtCheckerException` will be thrown.

4.6. Determining visibility

`Navigator` objects have a `displayed` property that indicates whether the element is visible to the user or not. The `displayed` property of a navigator object that doesn't match anything is always `false`.

4.7. Focus

It is possible to verify if a given `Navigator` object holds the currently focused element using its `focused` property. `Navigator` objects that don't match any elements will return `false` as the value of `focused` property and ones that match multiple elements will throw a `SingleElementNavigatorOnlyMethodException`.

4.8. Size and location

You can obtain the size and location of content on the page. All units are in pixels. The size is available via the `height` and `width` properties, while the location is available as the `x` and `y` properties which represent the distance from the top left of the page (or parent frame) to the top left point of the content.

All of these properties operate on the **sole** matched element only and a `SingleElementNavigatorOnlyMethodException` is thrown if they are accessed on a multi element `Navigator`.

Consider the following html…

```
<div style="height: 20px; width: 40px; position: absolute; left: 20px; top: 10px"></div>
<div style="height: 40px; width: 100px; position: absolute; left: 30px; top: 150px"></div>
```

The following conditions are satisfied for it…

```
assert $("div", 0).height == 20
assert $("div", 0).width == 40
assert $("div", 0).x == 20
assert $("div", 0).y == 10
```

To obtain any of the properties for all matched elements, you can use the Groovy spread operator.

```
assert $("div")*.height == [20, 40]
assert $("div")*.width == [40, 100]
assert $("div")*.x == [20, 30]
assert $("div")*.y == [10, 150]
```

4.9. Accessing tag name, attributes, text and classes

The `tag()`, `text()` and `classes()` methods as well as accessing attributes via the `@` notation or `attr()` method return the requested content on the **sole** matched element. If these methods are called on a multi element `Navigator` then a `SingleElementNavigatorOnlyMethodException` is thrown. The `classes()` method returns a `java.util.List` of class names sorted alphabetically.

Consider the following HTML…

```
<p title="a" class="a para">a</p>
<p title="b" class="b para">b</p>
<p title="c" class="c para">c</p>
```

The following assertions are valid…

```
assert $(".a").text() == "a"
assert $(".a").tag() == "p"
assert $(".a").@title == "a"
assert $(".a").classes() == ["a", "para"]
```

To obtain information about all matched content, you use the Groovy *spread operator*…

```
assert $("p")*.text() == ["a", "b", "c"]
assert $("p")*.tag() == ["p", "p", "p"]
assert $("p")*.@title == ["a", "b", "c"]
assert $("p")*.classes() == [["a", "para"], ["b", "para"], ["c", "para"]]
```

4.10. CSS properties

Css properties of a single element navigator can be accessed using the `css()` method. If that method is called on a multi element `Navigator` then a `SingleElementNavigatorOnlyMethodException` is thrown.

Consider the following HTML…

```
<div style="float: left">text</div>
```

You can obtain the value of `float` css property in the following way…

```
assert $("div").css("float") == "left"
```

There are some limitations when it comes to retrieving css properties of `Navigator` objects. Color values should be returned as rgba strings, so, for example if the `background-color` property is set as `green` in the HTML source, the returned value will be `rgba(0, 255, 0, 1)`. Note that shorthand CSS properties (e.g. `background`, `font`, `border`, `border-top`, `margin`, `margin-top`, `padding`, `padding-top`, `list-style`, `outline`, `pause`, `cue`) are not returned, in accordance with the DOM CSS2 specification - you should directly access the longhand properties (e.g. `background-color`) to access the desired values.

4.11. Sending keystrokes

Given the following html…

```
<input type="text"/>
```

You can send keystrokes to the input (and any other content) via the leftShift operator, which is a shortcut for the `sendKeys()` method of WebDriver.

```
$("input") << "foo"
assert $("input").value() == "foo"
```

How content responds to the keystrokes depends on what the content is.

Non characters (e.g. delete key, key chords, etc.)

It is possible to send non-textual characters to content by using the WebDriver `Keys` enumeration…

```
import org.openqa.selenium.Keys

$("input") << Keys.chord(Keys.CONTROL, "c")
```

Here we are sending a “control-c” to an input.

See the documentation for `Keys` enumeration for more information on the possible keys.

4.12. Accessing input values

The value of `input`, `select` and `textarea` elements can be retrieved and set with the `value` method. Calling `value()` with no arguments will return the String value of *the sole* element in the Navigator. If that method is called on a multi element `Navigator` then a `SingleElementNavigatorOnlyMethodException` is thrown. Calling `value(value)` will set the current value of *all* elements in the Navigator. The argument can be of any type and will be coerced to a `String` if necessary. The exceptions are that when setting a `checkbox` value the method expects a `boolean` (or an existing checkbox value or label) and when setting a multiple `select` the method expects an array or `Collection` of values.

4.13. Form control shortcuts

Interacting with form controls (`input`, `select` etc.) is such a common task in web functional testing that Geb provides convenient shortcuts for common functions.

Geb supports the following shortcuts for dealing with form controls.

Consider the following HTML…

```
<form>
    <input type="text" name="geb" value="testing" />
</form>
```

The value can be read and written via property notation…

```
assert $("form").geb == "testing"
$("form").geb = "goodness"
assert $("form").geb == "goodness"
```

These are literally shortcuts for…

```
assert $("form").find("input", name: "geb").value() == "testing"
$("form").find("input", name: "geb").value("goodness")
assert $("form").find("input", name: "geb").value() == "goodness"
```

There is also a shortcut for obtaining a navigator based on a control name…

```
assert $("form").geb() == $("form").find("input", name: "geb")
```

In the above and below examples with form controls we are using code like `$("form").someInput` where we could be using just `someInput` as long as there is only one control with the *name* `someInput` on the page. In the examples we are using `$("form").someInput` to hopefully be clearer.

If your content definition (either a page or a module) describes content which is an `input`, `select` or `textarea`, you can access and set its value the same way as described above for forms. Given a page and module definitions for the above mentioned HTML:

```
class ShortcutModule extends Module {
    static content = {
        geb { $('form').geb() }
    }
}

class ShortcutPage extends Page {
    static content = {
        geb { $('form').geb() }
        shortcutModule { module ShortcutModule }
    }
}
```

The following will pass:

```
page ShortcutPage
assert geb == "testing"
geb = "goodness"
assert geb == "goodness"
```

As well as:

```
page ShortcutPage
assert shortcutModule.geb == "testing"
shortcutModule.geb = "goodness"
assert shortcutModule.geb == "goodness"
```

4.14. Setting form contr ol values

The following examples describe usage of form controls only using code like `$("form").someInput`. Given a content definition `myContent { $("form").someInput }`, you can substitute `$("form").someInput` in the examples with `myContent`.

Trying to set a value on an element which is not one of `input`, `select` or `textarea` will cause an `UnableToSetElementException` to be thrown.

4.14.1. Select

Select values are set by assigning the value or text of the required option. Assigned values are automatically coerced to String. For example…

```
<form>
    <select name="artist">
        <option value="1">Ima Robot</option>
        <option value="2">Edward Sharpe and the Magnetic Zeros</option>
        <option value="3">Alexander</option>
    </select>
</form>
```

We can select options with…

```
$("form").artist = "1" (1)
$("form").artist = 2 (2)
$("form").artist = "Alexander" (3)
```

1 First option selected by its value attribute.

2 Second option selected by its value attribute with argument coercion.

3 Third option selected by its text.

If you attempt to set a select to a value that does not match the value or text of any options, an `IllegalArgumentException` will be thrown.

4.14.2. Multiple select

If the select has the `multiple` attribute it is set with a array or `Collection` of values. Any options not in the values are un-selected. For example…

```
<form>
    <select name="genres" multiple>
        <option value="1">Alt folk</option>
        <option value="2">Chiptunes</option>
        <option value="3">Electroclash</option>
        <option value="4">G-Funk</option>
        <option value="5">Hair metal</option>
    </select>
</form>
```

We can select options with…

```
$("form").genres = ["2", "3"] (1)
$("form").genres = [1, 4, 5] (2)
$("form").genres = ["Alt folk", "Hair metal"] (3)
$("form").genres = [] (4)
```

1 Second and third options selected by their value attributes.

2 First, fourth and fifth options selected by their value attributes with argument coercion.

3 First and last options selected by their text.

4 All options un-selected.

If the collection being assigned contains a value that does not match the value or text of any options, an `IllegalArgumentException` will be thrown.

4.14.3. Checkbox

Checkboxes are generally checked/unchecked by setting their value to `true` or `false`.

Consider the following html…

```
<form>
    <input type="checkbox" name="pet" value="dog" />
</form>
```

You can set a single checkbox in the following manner…

```
$("form").pet = true
```

Calling `value()` on a checked checkbox will return the value of its `value` attribute, i.e:

```
<form>
    <input type="checkbox" name="pet" value="dog" checked/>
</form>
```

```
assert $("input", name: "pet").value() == "dog"
assert $("form").pet == "dog"
```

Calling `value()` on an unchecked checkbox will return `false`, i.e:

```
<form>
    <input type="checkbox" name="pet" value="dog"/>
</form>
```

```
assert $("input", name: 'pet').value() == false
assert $("form").pet == false
```

In general you should use Groovy Truth when checking if a checkbox is checked:

```
<form>
    <input type="checkbox" name="checkedByDefault" value="checkedByDefaulValue" checked/>
    <input type="checkbox" name="uncheckedByDefault" value="uncheckedByDefaulValue"/>
</form>
```

```
assert $("form").checkedByDefault
assert !$("form").uncheckedByDefault
```

4.14.4. Multiple checkboxes

You can also check a checkbox by explicitly setting its `value` or using its label. This is useful when you have a number of checkboxes with the same name, i.e.:

```
<form>
    <label for="dog-checkbox">Canis familiaris</label>
    <input type="checkbox" name="pet" value="dog" id="dog-checkbox"/>
    <label for="cat-checkbox">Felis catus</label>
    <input type="checkbox" name="pet" value="cat" id="cat-checkbox" />
    <label for="lizard-checkbox">Lacerta</label>
    <input type="checkbox" name="pet" value="lizard" id="lizard-checkbox" />
</form>
```

You can select dog as your pet type as follows:

```
$("form").pet = "dog"
$("form").pet = "Canis familiaris"
```

If you wish to select multiple checkboxes instead of only one then you can use a collection as the value:

```
$("form").pet = ["dog", "lizard"]
$("form").pet = ["Canis familiaris", "Lacerta"]
```

When checking if a checkbox is checked and there are multiple checkboxes with the same name make sure that you use a navigator that holds only one of them before calling `value()` on it:

```
<form>
    <input type="checkbox" name="pet" value="dog" checked/>
    <input type="checkbox" name="pet" value="cat" />
</form>
```

```
assert $("input", name: "pet", value: "dog").value()
assert !$("input", name: "pet", value: "cat").value()
```

4.14.5. Radio

Radio values are set by assigning the value of the radio button that is to be selected or the label text associated with a radio button.

For example, with the following radio buttons…

```
<form>
    <label for="site-current">Search this site</label>
    <input type="radio" id="site-current" name="site" value="current">

    <label>Search Google
        <input type="radio" name="site" value="google">
    </label>
</form>
```

We can select the radios by value…

```
$("form").site = "current"
assert $("form").site == "current"
$("form").site = "google"
assert $("form").site == "google"
```

Or by label text…

```
$("form").site = "Search this site"
assert $("form").site == "current"
$("form").site = "Search Google"
assert $("form").site == "google"
```

4.14.6. Text inputs and textareas

In the case of a text `input` or a `textarea`, the assigned value becomes the element's *value* attribute.

```
<form>
    <input type="text" name="language"/>
    <input type="text" name="description"/>
</form>
```

```
$("form").language = "gro"
$("form").description = "Optionally statically typed dynamic lang"
assert $("form").language == "gro"
assert $("form").description == "Optionally statically typed dynamic lang"
```

It is also possible to append text by using the send keys shorthand…

```
$("form").language() << "ovy"
$("form").description() << "uage"
assert $("form").language == "groovy"
assert $("form").description == "Optionally statically typed dynamic language"
```

Which an also be used for non-character keys…

```
import org.openqa.selenium.Keys

$("form").language() << Keys.BACK_SPACE
assert $("form").language == "groov"
```

WebDriver has some issues with textareas and surrounding whitespace. Namely, some drivers implicitly trim whitespace from the beginning and end of the value. You can track this issue here.

4.14.7. File upload

It's currently not possible with WebDriver to simulate the process of a user clicking on a file upload control and choosing a file to upload via the normal file chooser. However, you can directly set the value of the upload control to the *absolute path* of a file on the system where the driver is running and on form submission that file will be uploaded. So if your html looks like…

```
<input type="file" name="csvFile"/>
```

And the `uploadedFile` variable holds a `File` instance pointing at the file you want to upload then this is how you can set the value of the upload control…

```
$("form").csvFile = uploadedFile.absolutePath
```

4.15. Complex interactions

WebDriver supports interactions that are more complex than simply clicking or typing into items, such as dragging. You can use this API directly when using Geb or use the more Geb friendly `interact` DSL.

4.15.1. Using the WebDriver `Actions` API directly

A Geb navigator object is built on top of a collection of WebDriver WebElement objects. It is possible to access the contained `WebElement` instances via the following methods on navigator objects:

```
WebElement singleElement()
WebElement firstElement()
WebElement lastElement()
Collection<WebElement> allElements()
```

By using the methods of the WebDriver `Actions` class with `WebElement` instances, complex user gestures can be emulated. First you will need to create an `Actions` instance after obtaining the WebDriver driver:

```
def actions = new Actions(driver)
```

Next, use methods of `Actions` to compose a series of UI actions, then call `build()` to create a concrete `Action`:

```
WebElement someItem = $("li.clicky").firstElement()
def shiftClick = actions.keyDown(Keys.SHIFT).click(someItem).keyUp(Keys.SHIFT).build()
```

Finally, call `perform()` to actually trigger the desired mouse or keyboard behavior:

```
shiftClick.perform()
```

4.15.2. Using `interact()`

To avoid having to manage building and performing an `Action` via the lifecycle of an `Actions` instance as well as having to obtain `WebElement` instances from `Navigator` when emulating user gestures, Geb adds the `interact()` method. When using that method, an `Actions` instance is implicitly created, built into an `Action`, and performed. The delegate of the closure passed to `interact()` is an instance of `InteractDelegate` which declares the same methods as `Actions` but takes `Navigator` as arguments for methods of `Actions` which take `WebElement`.

This `interact()` call performs the same work as the calls in the Using the WebDriver `Actions` API directly section:

```
interact {
    keyDown Keys.SHIFT
    click $("li.clicky")
    keyUp Keys.SHIFT
}
```

For the full list of available methods that can be used for emulating user gestures see the documentation for the `InteractDelegate` class.

4.15.3. Interact examples

Calls to `interact()` can be used to perform behaviors that are more complicated than clicking buttons and anchors or typing in input fields.

Drag and drop

`clickAndHold()`, `moveByOffset()`, and then `release()` will drag and drop an element on the page.

```
interact {
    clickAndHold($('#draggable'))
    moveByOffset(150, 200)
    release()
}
```

Drag-and-dropping can also be accomplished using the `dragAndDropBy()` convenience method from the Actions API:

```
interact {
    dragAndDropBy($("#draggable"), 150, 200)
}
```

In this particular example, the element will be clicked then dragged 150 pixels to the right and 200 pixels downward before being released.

Moving to arbitrary locations with the mouse is currently not supported by the HTMLUnit driver, but moving directly to elements is.

Control-clicking

Control-clicking several elements, such as items in a list, is performed the same way as shift-clicking.

```
import org.openqa.selenium.Keys

interact {
    keyDown(Keys.CONTROL)
    click($("ul.multiselect li", text: "Order 1"))
    click($("ul.multiselect li", text: "Order 2"))
    click($("ul.multiselect li", text: "Order 3"))
    keyUp(Keys.CONTROL)
}
```

5. Pages

Before reading this chapter, please make sure you have read the section on the `Browser.drive()` method

5.1. The Page Object pattern

```
Browser.drive {
    go "search"
    $("input[name='q']").value "Chuck Norris"
    $("input[value='Search']").click()
    assert $("li", 0).text().contains("Chuck")
}
```

This is valid Geb code, and it works well for a one off script but there are two big issues with this approach. Imagine that you have *many* tests that involve searching and checking results. The implementation of how to search and how to find the results is going to have to be duplicated in *every* test, maybe *many times* per test. As soon as something as trivial as the name of the search field changes you have to update a lot of code. The Page Object Pattern allows us to apply the same principles of modularity, reuse and encapsulation that we use in other aspects of programming to avoid such issues in browser automation code.

Here is the same script, utilising page objects…

```
import geb.Page

class SearchPage extends Page {
    static url = "search"
    static at = { title == "Search engine" }
    static content = {
        searchField { $("input[name=q]") }
        searchButton(to: ResultsPage) { $("input[value='Search']") }
    }

    void search(String searchTerm) {
        searchField.value searchTerm
        searchButton.click()
    }
}

class ResultsPage extends Page {
    static at = { title == "Results" }
    static content = {
        results { $("li") }
        result { index -> results[index] }
    }
}

Browser.drive {
    to SearchPage
    search "Chuck Norris"
    assert result(0).text().contains("Chuck")
}
```

You have now encapsulated, in a reusable fashion, information about each page and how to interact with it. As anyone who has tried to knows, maintaining a large suite of functional web tests for a changing application can become an expensive and frustrating process.

Geb's support for the Page Object pattern addresses this problem.

5.2. The Page superclass

All page objects **must** inherit from Page.

5.3. The content DSL

Geb features a DSL for defining page content in a *templated* fashion, which allows very concise yet flexible page definitions. Pages define a `static` closure property called `content` that describes the page content.

Consider the following HTML…

```
<div id="a">a</div>
```

We could define this content as so…

```
class PageWithDiv extends Page {
    static content = {
        theDiv { $('div', id: 'a') }
    }
}
```

The structure to the content DSL is…

```
«name» { «definition» }
```

Where «definition» is Groovy code that is evaluated against the instance of the page.

Here is how it could be used…

```
Browser.drive {
    to PageWithDiv
    assert theDiv.text() == "a"
}
```

So how is this working? First, remember that the `Browser` instance delegates any method calls or property accesses that it doesn't know about to the current page instance. So the above code is the same as…

```
Browser.drive {
    to PageWithDiv
    assert page.theDiv.text() == "a"
}
```

Secondly, defined content becomes available as properties and methods on instance of the page…

```
Browser.drive {
    to PageWithDiv

    // Following two lines are equivalent
    assert theDiv.text() == "a"
    assert theDiv().text() == "a"
}
```

The Content DSL actually defines content *templates*. This is best illustrated by example…

```
class TemplatedPageWithDiv extends Page {
    static content = {
        theDiv { id -> $('div', id: id) }
    }
}

Browser.drive {
    to TemplatedPageWithDiv
    assert theDiv("a").text() == "a"
}
```

There are no restrictions on what arguments can be passed to content templates.

A content template can return *anything*. Typically they will return a `Navigator` object through the use of the `$()` function, but it can be anything.

```
class PageWithStringContent extends Page {
    static content = {
        theDivText { $('div#a').text() }
    }
}

Browser.drive {
    to PageWithStringContent
    assert theDivText == "a"
}
```

It's important to realise that «`definition`» code is evaluated against the page instance. This allows code like the following…

```
class PageWithContentReuse extends Page {
    static content = {
        theDiv { $("div#a") }
        theDivText { theDiv.text() }
    }
}
```

And this is not restricted to other content…

```
class PageWithContentUsingAField extends Page {
    def divId = "a"

    static content = {
        theDiv { $('div', id: divId) }
    }
}
```

Or…

```
class PageWithContentUsingAMethod extends Page {
    static content = {
        theDiv { $('div', id: divId()) }
    }

    def divId() {
        "a"
    }
}
```

5.3.1. Template options

Template definitions can take different options. The syntax is…

```
«name»(«options map») { «definition» }
```

For example…

```
theDiv(cache: false, required: false) { $("div", id: "a") }
```

The following are the available options.

`required`

Default value: `true`

The `required` option controls whether or not the content returned by the definition has to exist or not. This is only relevant when the definition returns a `Navigator` object (via the `$()` function) or `null`, it is ignored if the definition returns anything else.

If the `required` option is set to `true` and the returned content does not exist, a `RequiredPageContentNotPresent` exception will be thrown.

Given a completely empty html document the following will pass…

```
class PageWithTemplatesUsingRequiredOption extends Page {
    static content = {
        requiredDiv { $("div", id: "b") }
        notRequiredDiv(required: false) { $("div", id: "b") }
    }
}
```

```
to PageWithTemplatesUsingRequiredOption

assert !notRequiredDiv

def thrown = false
try {
    page.requiredDiv
} catch (RequiredPageContentNotPresent e) {
    thrown = true
}
assert thrown
```

cache

Default value: `false`

The `cache` option controls whether or not the definition is evaluated each time the content is requested (the content is cached for each unique set of parameters).

```
class PageWithTemplateUsingCacheOption extends Page {
    def value = 1
    static content = {
        notCachedValue { value }
        cachedValue(cache: true) { value }
    }
}
```

```
to PageWithTemplateUsingCacheOption
assert notCachedValue == 1
assert cachedValue == 1

value = 2

assert notCachedValue == 2
assert cachedValue == 1
```

Caching is a performance optimisation and is disabled by default. You may want to enable if you notice that the a particular content definition is taking a long time to resolve.

to

Default value: `null`

The `to` option allows the definition of which page the browser will be sent to if the content is clicked.

```
class PageWithTemplateUsingToOption extends Page {
    static content = {
        helpLink(to: HelpPage) { $("a", text: "Help") }
    }
}

class HelpPage extends Page { }
```

```
to PageWithTemplateUsingToOption
helpLink.click()
assert page instanceof HelpPage
```

The `to` value will be implicitly used as an argument to the content's `click()` method, effectively setting the new page type and verifying its at checker. See the section on clicking content for how this changes the browser's page object.

This option also supports all types that can be passed to any of the `Browser.page()` method variants:

- a page instance

- a list of page classes
- a list of page instances

When using lists variants (here shown with page classes)…

```
static content = {
    loginButton(to: [LoginSuccessfulPage, LoginFailedPage]) { $("input.loginButton") }
}
```

Then, on click, the browser's page is set to the first page in the list whose at checker passes. This is equivalent to the `page(Class<? extends Page>[])` and `page(Page[])` browser methods which are explained in the section on changing pages.

All of the page classes and classes of the page instances passed in when using any variant of the `to` option have to have an "at" checker defined otherwise an `UndefinedAtCheckerException` will be thrown.

`wait`

Default value: `false`

Allowed values:

- **`true`** - wait for the content using the *default wait* configuration
- **a string** - wait for the content using the *wait preset* with this name from the configuration
- **a number** - wait for the content for this many seconds, using the *default retry interval* from the configuration
- **a 2 element list of numbers** - wait for the content using element 0 as the timeout seconds value, and element 1 as the retry interval seconds value

Any other value will be interpreted as `false`.

The `wait` option allows Geb to wait an amount of time for content to appear on the page, instead of throwing a `RequiredPageContentNotPresent` exception if the content is not present when requested.

```
class DynamicPageWithWaiting extends Page {
    static content = {
        dynamicallyAdded(wait: true) { $("p.dynamic") }
    }
}
```

```
to DynamicPageWithWaiting
assert dynamicallyAdded.text() == "I'm here now"
```

This is equivalent to:

```
class DynamicPageWithoutWaiting extends Page {
    static content = {
        dynamicallyAdded { $("p.dynamic") }
    }
}
```

```
to DynamicPageWithoutWaiting
assert waitFor { dynamicallyAdded }.text() == "I'm here now"
```

See the section on waiting for the semantics of the `waitFor()` method, that is used here internally. Like `waitFor()` a `WaitTimeoutException` will be thrown if the wait timeout expires.

It is also possible to use `wait` when defining non-element content, such as a string or number. Geb will wait until the content definition returns a value that conforms to the Groovy Truth.

```
class DynamicPageWithNonNavigatorWaitingContent extends Page {
    static content = {
        status { $("p.status") }
        success(wait: true) { status.text().contains("Success") }
    }
}
```

```
to DynamicPageWithNonNavigatorWaitingContent
assert success
```

In this case, we are inherently waiting for the `status` content to be on the page and for it to contain the string “Success”. If the `status` element is not present when we request `success`, the `RequiredPageContentNotPresent` exception that would be thrown is swallowed and Geb will try again after the retry interval has expired.

You can modify the behaviour of content with `wait` option set to true if you use it together with `required` option set to false. Given a content definition:

```
static content = {
    dynamicallyAdded(wait: true, required: false) { $("p.dynamic") }
}
```

Then if wait timeout expires when retrieving `dynamicallyAdded`, there will be no `WaitTimeoutException` thrown, and the last closure evaluation value will be returned. If there is an exception thrown during closure evaluation, it will be wrapped in an `UnknownWaitForEvaluationResult` instance and returned.

Waiting content blocks are subject to “implicit assertions”. See the section on implicit assertions for more information.

toWait

Default value: `false`

Allowed values are the same as for the `wait` option.

Can be used together with the `to` option to specify that the page changing action performed when the content is clicked is asynchronous. This essentially means that verification of page transition (“at checking”) should be wrapped in a `waitFor()` call.

```
class PageWithTemplateUsingToWaitOption extends Page {
    static content = {
        asyncPageLoadButton(to: AsyncPage, toWait: true) { $("button#load-content") } (1)
    }
}

class AsyncPage extends Page {
    static at = { $("#async-content") }
}
```

1 Page change is asynchronous, e.g. an ajax call is involved.

```
to PageWithTemplateUsingToWaitOption
asyncPageLoadButton.click()
assert page instanceof AsyncPage
```

See the section on waiting for the semantics of the `waitFor()` method, that is used here internally. Like `waitFor()` a `WaitTimeoutException` will be thrown if the wait timeout expires.

page

Default value: `null`

The `page` option allows the definition of a page the browser will be set to if the content describes a frame and is used in a `withFrame()` call.

Given the following HTML…

```
<html>
    <body>
        <frame id="frame-id" src="frame.html"></frame>
    <body>
</html>
```

…and the contents of frame.html…

```
<html>
    <body>
        <span>frame text</span>
```

```
    </body>
</html>
```

…the following will pass…

```
class PageWithFrame extends Page {
    static content = {
        myFrame(page: FrameDescribingPage) { $('#frame-id') }
    }
}

class FrameDescribingPage extends Page {
    static content = {
        frameContentsText { $('span').text() }
    }
}
```

```
to PageWithFrame
withFrame(myFrame) {
    assert frameContentsText == 'frame text'
}
```

5.3.2. Aliasing

If you wish to have the same content definitions available under different names you can create a content definition that specifies `aliases` parameter:

```
class AliasingPage extends Page {
    static content = {
        someDiv { $("div#aliased") }
        aliasedDiv(aliases: "someDiv")
    }
}
```

```
to AliasingPage
assert someDiv.@id == aliasedDiv.@id
```

Remember that the aliased content has to be defined before the aliasing content, otherwise you will get a `InvalidPageContent` exception.

5.4. “At” verification

Each page can define a way to check whether the underlying browser is at the page that the page class actually represents. This is done via a `static at` closure…

```
class PageWithAtChecker extends Page {
    static at = { $("h1").text() == "Example" }
}
```

This closure can either return a `false` value or throw an `AssertionError` (via the `assert` method). The `verifyAt()` method call will either:

- return `true` if the “at” checker passes
- if implicit assertions are enabled and the “at” checker fails it will throw an `AssertionError`
- return `false` if implicit assertions are not enabled and the the “at” checker fails

Considering the example above you could use it like this…

```
class PageLeadingToPageWithAtChecker extends Page {
    static content = {
        link { $("a#to-page-with-at-checker") }
    }
}
```

```
to PageLeadingToPageWithAtChecker
link.click()
page PageWithAtChecker
verifyAt()
```

The `verifyAt()` method is used by the `Browser.at()` method that takes a page class or instance. It behaves the same as `verifyAt()` if the "at" checker fails and returns a page instance if the checker succeeds (which is useful if you wish to write strongly typed code with Geb)…

```
to PageLeadingToPageWithAtChecker
link.click()
at PageWithAtChecker
```

> There is no need to perform "at" verification after navigating to a page using `to()` method or clicking on content with `to` option specified as it is performed implicitly in these cases.

At checkers are subject to "implicit assertions". See the section on implicit assertions for more information.

If you don't wish to get an exception when "at" checking fails there are methods that return `false` in that case: `Page#verifyAtSafely()` and `Browser#isAt(Class<? extends Page>)`.

As mentioned previously, when a content template defines a `to` option of more than one page the page's `verifyAt()` method is used to determine which one of the pages to use. In this situation, any `AssertionError` thrown by "at" checkers are suppressed.

The "at" checker is evaluated against the page instance, and can access defined content or any other variables or methods…

```
class PageWithAtCheckerUsingContent extends Page {
    static at = { heading == "Example" }

    static content = {
        heading { $("h1").text() }
    }
}
```

If a page does not have an "at" checker, the `verifyAt()` and `at()` methods will throw an `UndefinedAtCheckerException`. The same will happen if any of the pages in the list used as `to` content template option doesn't define an "at" checker.

It can sometimes prove useful to wrap "at" verification in `waitFor()` calls by default - some drivers are known to return control after URL change before the page is fully loaded in some circumstances or before one might consider it to be loaded. This behaviour can be required using `atCheckWaiting` configuration entry.

Unexpected pages

A list of unexpected pages can be provided via `unexpectedPages` configuration entry.

> Note that this feature does not operate on HTTP response codes as these are not exposed by WebDriver thus Geb does not have access to them. To use this feature your application has to render custom error pages that can be modeled as page classes and detected by an "at" checker.

If configured, the classes from the list specified as `unexpectedPages` configuration entry will be checked for first when "at" checking is performed for any page, and an `UnexpectedPageException` with an appropriate message will be raised if any of them is encountered.

Given that your application renders a custom error page when a page is not found with a text like "Sorry but we could not find that page", you can model that page with the following class:

```
class PageNotFoundPage extends Page {
    static at = { $("#error-message").text() == "Sorry but we could not find that page" }
}
```

Then register that page in configuration:

```
unexpectedPages = [PageNotFoundPage]
```

When checking if the browser is at a page while the "at" checker for `PageNotFoundPage` is fulfilled, an `UnexpectedPageException` will be raised.

```
try {
    at ExpectedPage
    assert false //should not get here
```

```
} catch (UnexpectedPageException e) {
    assert e.message.startsWith("An unexpected page ${PageNotFoundPage.name} was encountered")
}
```

Unexpected pages will be checked for whenever “at” checking is performed, even implicitly like when using `to` content template option or passing one or many page classes to `Navigator.click()` method.

It is possible to explicitly check if the browser is at an unexpected page. Following will pass without throwing an `UnexpectedPageException` if “at” checking for `PageNotFoundPage` succeeds:

```
at PageNotFoundPage
```

The global `atCheckWaiting` configuration does not apply when checking for unexpected pages. That is, even if configuration calls for implicitly wrapping “at checks” in a `waitFor()` call it is not done when verifying “at checkers” of unexpected pages. This is due to the fact that “at checkers” for unexpected pages are not fulfilled for most of the time and they are checked for every `at()` and `to()` call thus wrapping them in an implicit `waitFor()` call would make these methods extremely slow. The page level `atCheckWaiting` configuration on the other hand applies to unexpected pages so you can use it if you actually need to wait around “at checkers” of such pages.

5.5. Page URLs

Pages can define URLs via the `static url` property.

```
class PageWithUrl extends Page {
    static url = "example"
}
```

The url is used when using the browser `to()` method.

```
to PageWithUrl
assert currentUrl.endsWith("example")
```

See the section on the base url for notes about URLs and slashes.

5.5.1. URL fragments

Pages can also define URL fragment identifiers (the part after a # character at the end of an url) via the `static fragment` property.

The value assigned can be either a `String` which will be used as is or a `Map` which will be translated into an `application/x-www-form-urlencoded String`. The latter is particularly useful when dealing with single page applications that store state in the fragment identifier by form encoding it.

There is no need to escape any of the strings used for url fragments as all the necessary escaping is performed by Geb.

Consider the following page which defines an url fragment in such single page application scenario:

```
class PageWithFragment extends Page {
    static fragment = [firstKey: "firstValue", secondKey: "secondValue"]
}
```

The fragment is then used when using the browser `to()` method.

```
to PageWithFragment
assert currentUrl.endsWith("#firstKey=firstValue&secondKey=secondValue")
```

You can also use fragments which are dynamic - you can learn how in *URL fragments* subsection of *Advanced page navigation* chapter.

5.5.2. Page level `atCheckWaiting` configuration

At checkers for a specific page can be configured to be implicitly wrapped with `waitFor()` calls. This can be set with the `static atCheckWaiting` property.

```
class PageWithAtCheckWaiting extends Page {
    static atCheckWaiting = true
}
```

The possible values for the `atCheckWaiting` option are the same as for the `wait` content template option.

The `atCheckWaiting` value configured at page level takes priority over the global value specified in the configuration.

5.6. Advanced page navigation

Page classes can customise how they generate URLs when used in conjunction with the browser `to()` method.

Consider the following example…

```
class PageObjectsPage extends Page {
    static url = "pages"
}
```

```
Browser.drive(baseUrl: "http://www.gebish.org/") {
    to PageObjectsPage
    assert currentUrl == "http://www.gebish.org/pages"
}
```

The `to()` method can also take arguments…

```
class ManualsPage extends Page {
    static url = "manual"
}
```

```
Browser.drive(baseUrl: "http://www.gebish.org/") {
    to ManualsPage, "0.9.3", "index.html"
    assert currentUrl == "http://www.gebish.org/manual/0.9.3/index.html"
}
```

Any arguments passed to the `to()` method after the page class are converted to a URL path by calling `toString()` on each argument and joining them with "`/`".

However, this is extensible. You can specify how a set of arguments is converted to a URL path to be added to the page URL. This is done by overriding the `convertToPath()` method. The `Page` implementation of this method looks like this…

```
String convertToPath(Object[] args) {
    args ? '/' + args*.toString().join('/') : ""
}
```

You can either overwrite this catchall method to control path conversion for all invocations or provide an overloaded version for a specific type signature. Consider the following…

```
class Manual {
    String version
}

class ManualsPage extends Page {
    static url = "manual"
    String convertToPath(Manual manual) {
        "/${manual.version}/index.html"
    }
}
```

```
def someManualVersion = new Manual(version: "0.9.3")

Browser.drive(baseUrl: "http://www.gebish.org/") {
    to ManualsPage, someManualVersion
    assert currentUrl == "http://www.gebish.org/manual/0.9.3/index.html"
}
```

5.6.1. Named params

Any type of argument can be used with the `to()` method, **except** named parameters (i.e. a `Map`). Named parameters are **always** interpreted as query parameters and they are never sent Using the classes from the above example…

```
def someManualVersion = new Manual(version: "0.9.3")

Browser.drive(baseUrl: "http://www.gebish.org/") {
    to ManualsPage, someManualVersion, flag: true
    assert currentUrl == "http://www.gebish.org/manual/0.9.3/index.html?flag=true"
}
```

5.6.2. URL fragments

An instance of `UrlFragment` can be passed as an argument which follows a page class or instance when calling `to()` in order to dynamically control the fragment identifier part of the url. The `UrlFragment` class comes with two static factory methods: one for creating a fragment from an explicit `String` and one for creating a fragment from a `Map` which is then form encoded.

The following shows a usage example utilising the classes from the examples above…

```
Browser.drive(baseUrl: "http://www.gebish.org/") {
    to ManualsPage, UrlFragment.of("advanced-page-navigation"), "0.9.3", "index.html"
    assert currentUrl == "http://www.gebish.org/manual/0.9.3/index.html#advanced-page-navigation"
}
```

If you are using parameterized pages and you wish the fragment to be determined dynamically, e.g. based on page properties then you can override the `getPageFragment()` method:

```
class ParameterizedManualsPage extends Page {
    String version
    String section

    @Override
    String convertToPath(Object[] args) {
        "manual/$version/index.html"
    }

    @Override
    UrlFragment getPageFragment() {
        UrlFragment.of(section)
    }
}

Browser.drive(baseUrl: "http://www.gebish.org/") {
    to new ParameterizedManualsPage(version: "0.9.3", section: "advanced-page-navigation")
    assert currentUrl == "http://www.gebish.org/manual/0.9.3/index.html#advanced-page-navigation"
}
```

5.7. Parametrized pages

`Browser` methods like `to()`, `via()`, `at()` and `page()` accept not only page classes but page instances as well. This is useful for example when parameterizing pages to use property values in “at” checkers:

```
class BooksPage extends Page {
    static content = {
        book { bookTitle -> $("a", text: bookTitle) }
    }
}

class BookPage extends Page {
    String forBook

    static at = { forBook == bookTitle }

    static content = {
        bookTitle { $("h1").text() }
    }
}
```

```
Browser.drive {
    to BooksPage
    book("The Book of Geb").click()

    at(new BookPage(forBook: "The Book of Geb"))
}
```

Manually instantiated pages have to be initialized before they can be used. Initialization is performed as part of the `Browser` methods mentioned above. Failing to pass the page instance to one of these methods and calling any method on an uninitialized page instance might result in a `PageInstanceNotInitializedException`.

5.8. Inheritance

Pages can be arranged in an inheritance hierarchy. The content definitions are merged…

```
class BasePage extends Page {
    static content = {
        heading { $("h1") }
    }
}

class SpecializedPage extends BasePage {
    static content = {
        footer { $("div.footer") }
    }
}
```

```
Browser.drive {
    to SpecializedPage
    assert heading.text() == "Specialized page"
    assert footer.text() == "This is the footer"
}
```

If a subclass defines a content template with the same name as a content template defined in a superclass, the subclass version replaces the version from the superclass.

5.9. Lifecycle hooks

Page classes can optionally implement methods that are called when the page is set as the browser's current page and when it is swapped out for another page. This can be used to transfer state between pages.

5.9.1. onLoad(Page previousPage)

The `onLoad()` method is called with previous page object instance when the page becomes the new page object for a browser.

```
class FirstPage extends Page {
}

class SecondPage extends Page {
    String previousPageName

    void onLoad(Page previousPage) {
        previousPageName = previousPage.class.simpleName
    }
}
```

```
Browser.drive {
    to FirstPage
    to SecondPage
    assert page.previousPageName == "FirstPage"
}
```

5.9.2. onUnload(Page newPage)

The `onUnload()` method is called with next page object instance when the page is being replaced as the page object for the browser.

```
class FirstPage extends Page {
    String newPageName

    void onUnload(Page newPage) {
        newPageName = newPage.class.simpleName
    }
}

class SecondPage extends Page {
}
```

```
Browser.drive {
    def firstPage = to FirstPage
    to SecondPage
    assert firstPage.newPageName == "SecondPage"
}
```

5.10. Dealing with frames

Frames might seem a thing of the past, but if you're accessing or testing some legacy application with Geb, you might still need to deal with them. Thankfully, Geb makes working with them groovier thanks to the `withFrame()` method which is available on `Browser`, `Page` and `Module` instances.

5.10.1. Executing code in the context of a frame

There are multiple flavours of the `withFrame()` method, but for all of them the last closure parameter is executed in the context of a frame specified by the first parameter, and after the execution the page is restored to what it was before the call:

- `withFrame(String, Closure)` - `String` parameter contains the name or id of a frame element
- `withFrame(int, Closure)` - `int` parameter contains the index of the frame element, that is, if a page has three frames, the first frame would be at index `0`, the second at index `1` and the third at index `2`
- `withFrame(Navigator, Closure)` - `Navigator` parameter should contain a frame element
- `withFrame(SimplePageContent, Closure)` - `SimplePageContent`, which is a type returned by content templates, should contain a frame element

Given the following HTML…

```
<html>
    <body>
        <iframe name="header" src="frame.html"></iframe>
        <iframe id="footer" src="frame.html"></iframe>
        <iframe id="inline" src="frame.html"></iframe>
        <span>main</span>
    <body>
</html>
```

…the code for frame.html…

```
<html>
    <body>
        <span>frame text</span>
    </body>
</html>
```

…and a page class…

```
class PageWithFrames extends Page {
    static content = {
        footerFrame { $('#footer') }
    }
}
```

…then this code will pass…

```
to PageWithFrames

withFrame('header') { assert $('span').text() == 'frame text' }
withFrame('footer') { assert $('span').text() == 'frame text' }
withFrame(0) { assert $('span').text() == 'frame text' }
withFrame($('#footer')) { assert $('span').text() == 'frame text' }
withFrame(footerFrame) { assert $('span').text() == 'frame text' }

assert $('span').text() == 'main'
```

If a frame cannot be found for a given first argument of the `withFrame()` call, then `NoSuchFrameException` is thrown.

5.10.2. Switching pages and frames at once

All of the aforementioned `withFrame()` variants also accept an optional second argument (a page class or a page instance) which allows to switch page for the execution of the closure passed as the last parameter. If the page used specifies an "at" checker it will be verified after switching the context to the frame.

Given the html and page class from the previous example the following is an example usage with a page class:

```
class PageDescribingFrame extends Page {
    static content = {
        text { $("span").text() }
    }
}
```

```
to PageWithFrames

withFrame('header', PageDescribingFrame) {
    assert page instanceof PageDescribingFrame
    assert text == "frame text"
}

assert page instanceof PageWithFrames
```

And this is how an example usage with a page instance looks like:

```
class ParameterizedPageDescribingFrame extends Page {
    String expectedFrameText

    static at = { text == expectedFrameText }

    static content = {
        text { $("span").text() }
    }
}
```

```
to PageWithFrames

withFrame('header', new ParameterizedPageDescribingFrame(expectedFrameText: "frame text")) {
    assert page instanceof ParameterizedPageDescribingFrame
}

assert page instanceof PageWithFrames
```

It is also possible to specify a page to switch to for a page content that describes a frame.

6. Modules

Modules are re-usable definitions of content that can be used across multiple pages. They are useful for modelling things like UI widgets that are used across multiple pages, or even for defining more complex UI elements in a page.

They are defined in a manner similar to pages, but extend `Module`…

```
class FormModule extends Module {
    static content = {
        button { $("input", type: "button") }
    }
}
```

Pages can "include" modules using the following syntax…

```
class ModulePage extends Page {
    static content = {
        form { module FormModule }
    }
}
```

The `module` method returns an instance of a module class which can then be used in the following way…

```
Browser.drive {
    to ModulePage
```

```
    form.button.click()
}
```

Modules can also be parameterised…

```
class ParameterizedModule extends Module {
    String formId
    static content = {
        button {
            $("form", id: formId).find("input", type: "button")
        }
    }
}
```

Where the parameters are passed to constructor of the module…

```
class ParameterizedModulePage extends Page {
    static content = {
        form { id -> module(new ParameterizedModule(formId: id)) }
    }
}
```

```
Browser.drive {
    to ParameterizedModulePage
    form("personal-data").button.click()
}
```

Modules can also include other modules…

```
class OuterModule extends Module {
    static content = {
        form { module FormModule }
    }
}

class OuterModulePage extends Page {
    static content = {
        outerModule { module OuterModule }
    }
}
```

```
Browser.drive {
    to OuterModulePage
    outerModule.form.button.click()
}
```

6.1. Base and context

Modules can be localised to a specific section of the page that they are used in, or they can specify an absolute context as part of their definition. There are two ways that a modules base/context can be defined.

Module can be based on a `Navigator` instance…

```
class FormModule extends Module {
    static content = {
        button { $("input", type: "button") }
    }
}

class PageDefiningModuleWithBase extends Page {
    static content = {
        form { $("form").module(FormModule) }
    }
}
```

```
Browser.drive {
    to PageDefiningModuleWithBase
    form.button.click()
}
```

It can also be done outside of a content definition…

```
Browser.drive {
    go "/"
    $("form").module(FormModule).button.click()
}
```

We can define a `Navigator` context when including the module using the above syntax. This now means that calls to *all* `Navigator` (e.g. `$()`) method calls that occur within the module are against the given context (in this case, the `form` element).

However, module classes can also define their own base…

```
class FormModuleWithBase extends FormModule {
    static base = { $("form") }
}

class PageUsingModuleWithBase extends Page {
    static content = {
        form { module FormModuleWithBase }
    }
}
```

```
Browser.drive {
    to PageUsingModuleWithBase
    form.button.click()
}
```

Basing a module on a `Navigator` and defining a base in a module can be combined. Consider the following HTML…

```
<html>
    <div class="a">
        <form>
            <input name="thing" value="a"/>
        </form>
    </div>
    <div class="b">
        <form>
            <input name="thing" value="b"/>
        </form>
    </div>
</html>
```

And the following content definitions…

```
class ThingModule extends Module {
    static base = { $("form") }
    static content = {
        thingValue { thing().value() }
    }
}

class ThingsPage extends Page {
    static content = {
        formA { $("div.a").module(ThingModule) }
        formB { $("div.b").module(ThingModule)  }
    }
}
```

Then they can be used in the following way…

```
Browser.drive {
    to ThingsPage
    assert formA.thingValue == "a"
    assert formB.thingValue == "b"
}
```

If the module declares a base, it is always calculated *relative* to the `Navigator` used in the initialization statement. If the initialization statement does not use a `Navigator`, the module's base is calculated relative to the document root.

6.2. `Module` is-a `Navigator`

Modules always have a base navigator associated with them (if you don't specify a base for a module at all then it will be assigned the root element of the document as the base) so it is natural to think of them as navigators. Keeping in mind that `Module` implements `Navigator` and considering the following HTML…

```
<html>
    <form method="post" action="login">
        <input name="login" type="text"></input>
        <input name="password" type="password"></input>
        <input type="submit" value="Login"></input>
    </from>
</html>
```

As well as these content definitions…

```
class LoginFormModule extends Module {
    static base = { $("form") }
}

class LoginPage extends Page {
    static content = {
        form { module LoginFormModule }
    }
}
```

The following will pass…

```
Browser.drive {
    to LoginPage
    assert form.@method == "post"
    assert form.displayed
}
```

It's also possible to use `Navigator` methods inside of a module implementation…

```
class LoginFormModule extends Module {
    String getAction() {
        getAttribute("action")
    }
}
```

```
Browser.drive {
    to LoginPage
    assert form.action == "login"
}
```

6.3. Reusing modules acr oss pages

As previously stated, modules can be used to model page fragments that are reused across multiple pages. For example, many different types of pages in your application may show information about the user's shopping cart. You could handle this with modules…

```
class CartInfoModule extends Module {
    static content = {
        section { $("div.cart-info") }
        itemCount { section.find("span.item-count").text().toInteger() }
        totalCost { section.find("span.total-cost").text().toBigDecimal() }
    }
}

class HomePage extends Page {
    static content = {
        cartInfo { module CartInfoModule }
    }
}

class OtherPage extends Page {
    static content = {
        cartInfo { module CartInfoModule }
    }
}
```

Modules work well for this.

6.4. Using modules for r epeating content

Other than content that is repeated on different pages (like the shopping cart mentioned above), pages also can have content that is repeated on the page itself. On a checkout page, the contents of the shopping cart could be summarized with the product name, the quantity and price for each product contained. For this kind of page, a list of modules can be collected using the `moduleList()` methods of `Navigator`.

Consider the following HTML for our cart contents:

```
<html>
    <table>
        <tr>
            <th>Product</th><th>Quantity</th><th>Price</th>
        </tr>
        <tr>
            <td>The Book Of Geb</td><td>1</td><td>5.99</td>
        </tr>
        <tr>
            <td>Geb Single-User License</td><td>1</td><td>99.99</td>
        </tr>
        <tr>
            <td>Geb Multi-User License</td><td>1</td><td>199.99</td>
        </tr>
    </table>
</html>
```

We can model one line of the table like this:

```
class CartRow extends Module {
    static content = {
        cell { $("td", it) }
        productName { cell(0).text() }
        quantity { cell(1).text().toInteger() }
        price { cell(2).text().toBigDecimal() }
    }
}
```

And define a list of CartRows in our Page:

```
class CheckoutPage extends Page {
    static content = {
        cartItems {
            $("table tr").tail().moduleList(CartRow) // tailing to skip the header row
        }
    }
}
```

Because the return value of `cartItems` is a list of CartRow instances, we can use any of the usual collection methods:

```
assert cartItems.every { it.price > 0.0 }
```

We can also access the cart items using subscript operator together with an index or a range of indexes:

```
assert cartItems[0].productName == "The Book Of Geb"
assert cartItems[1..2]*.productName == ["Geb Single-User License", "Geb Multi-User License"]
```

Keep in mind that you can use parametrized module instances to create lists of modules for repeating content:

```
class ParameterizedCartRow extends Module {
    def nameIndex
    def quantityIndex
    def priceIndex

    static content = {
        cell { $("td", it) }
        productName { cell(nameIndex).text() }
        quantity { cell(quantityIndex).text().toInteger() }
        price { cell(priceIndex).text().toBigDecimal() }
    }
}

class CheckoutPageWithParametrizedCart extends Page {
    static content = {
        cartItems {
            $("table tr").tail().moduleList {
```

```
                new ParameterizedCartRow(nameIndex: 0, quantityIndex: 1, priceIndex: 2)
            }
        }
    }
}
```

You might be wondering why the `moduleList()` method flavour that allows the use parameterized module instances takes a closure instead of a module instance, which is what the `module()` method for creating parameterized modules does. If it took a single instance then it could only initialize it multiple times and return a list which would contain the same instance but many times! To be able to return a list of different instances it's needs a closure which acts as a factory of module instances.

6.5. The content DSL

The Content DSL used for modules is *exactly* the same as the one used for pages, so all of the same options and techniques can be used.

6.6. Inheritance

Modules can use inheritance in the same way that pages can. That is, their content definitions are merged with any content redefined in the subclass taking precedence of the superclass.

6.7. Form control modules

If you are using Geb in a strongly typed manner you might consider using the provided modules modelling form controls instead of manipulating them directly using the `Navigator` API. This will result in longer content definitions but using them will be easier because you won't have to remember what is the meaning of `value()` calls for different types of controls. This is for example the case when manipulating checkboxes for which checking and unchecking is achieved by passing booleans to `value()` when interacting with them via `Navigator` API.

All of these modules (apart from `RadioButtons`) are expecting to be based on single element navigators and will throw `InvalidModuleBaseException` with an appropriate message if that is not the case. They also verify that the base contains an element of the expected type providing better error reporting in case of mistakenly selecting an incorrect element as the base.

6.7.1. `FormElement`

`FormElement` is a base class for all modules modelling form controls (apart from `RadioButtons`) and provides shortcut property methods for checking if a control is disabled or read only. You will usually call these methods on the module classes for specific control types and rarely use this module directly.

Given the html…

```
<html>
    <body>
        <input disabled="disabled" name="disabled"/>
        <input readonly="readonly" name="readonly"/>
    </body>
</html>
```

Following is an example of using the shortcut property methods provided…

```
assert $(name: "disabled").module(FormElement).disabled
assert !$(name: "disabled").module(FormElement).enabled
assert $(name: "readonly").module(FormElement).readOnly
assert !$(name: "readonly").module(FormElement).editable
```

6.7.2. `Checkbox`

The `Checkbox` module provides utility methods for checking and unchecking checkboxes as well as property methods for retrieving their state.

Given the html…

```
<html>
    <body>
        <input type="checkbox" name="flag"/>
    </body>
</html>
```

It can be used this way…

```
def checkbox = $(name: "flag").module(Checkbox)

assert !checkbox.checked
assert checkbox.unchecked

checkbox.check()

assert checkbox.checked

checkbox.uncheck()

assert checkbox.unchecked
```

6.7.3. Select

The Select module provides property methods for selecting options as well as retrieving selected option's value and text of a single choice select element.

Given the html…

```
<html>
    <body>
        <select name="artist">
            <option value="1">Ima Robot</option>
            <option value="2">Edward Sharpe and the Magnetic Zeros</option>
            <option value="3">Alexander</option>
        </select>
    </body>
</html>
```

It can be used this way…

```
def select = $(name: "artist").module(Select)
select.selected = "2"

assert select.selected == "2"
assert select.selectedText == "Edward Sharpe and the Magnetic Zeros"

select.selected = "Alexander"

assert select.selected == "3"
assert select.selectedText == "Alexander"
```

6.7.4. MultipleSelect

The MultipleSelect module provides property methods for selecting options as well as retrieving selected option's value and text of a multiple choice select element. These methods take and return lists of strings.

Given the html…

```
<html>
    <body>
        <select name="genres" multiple>
            <option value="1">Alt folk</option>
            <option value="2">Chiptunes</option>
            <option value="3">Electroclash</option>
            <option value="4">G-Funk</option>
            <option value="5">Hair metal</option>
        </select>
    </body>
</html>
```

It can be used this way…

```
def multipleSelect = $(name: "genres").module(MultipleSelect)
multipleSelect.selected = ["2", "3"]

assert multipleSelect.selected == ["2", "3"]
assert multipleSelect.selectedText == ["Chiptunes", "Electroclash"]

multipleSelect.selected = ["G-Funk", "Hair metal"]

assert multipleSelect.selected == ["4", "5"]
assert multipleSelect.selectedText == ["G-Funk", "Hair metal"]
```

6.7.5. TextInput

The TextInput module provides property methods for setting and retrieving text of an input text element.

Given the html…

```
<html>
    <body>
        <input type="text" name="language"/>
    </body>
</html>
```

It can be used this way…

```
def input = $(name: "language").module(TextInput)
input.text = "Groovy"

assert input.text == "Groovy"
```

6.7.6. Textarea

The Textarea module provides property methods for setting and retrieving text of a textarea element.

Given the html…

```
<html>
    <body>
        <textarea name="language"/>
    </body>
</html>
```

It can be used this way…

```
def textarea = $(name: "language").module(Textarea)
textarea.text = "Groovy"

assert textarea.text == "Groovy"
```

6.7.7. FileInput

The FileInput module provides property methods for setting and getting the file location for a file input element. These methods take and return `File` instances.

Given the html…

```
<html>
    <body>
        <input type="file" name="csv"/>
    </body>
</html>
```

It can be used this way…

```
def csvFile = new File("data.csv")
def input = $(name: "csv").module(FileInput)

input.file = csvFile
```

```
assert input.file.absolutePath == csvFile.absolutePath
```

6.7.8. RadioButtons

The RadioButtons module provides property methods for checking radio buttons as well as retrieving selected button's value and text of the label associated with it.

Given the html…

```
<html>
    <body>
        <label for="site-current">Search this site</label>
        <input type="radio" id="site-current" name="site" value="current">

        <label for="site-google">Search Google
            <input type="radio" id="site-google" name="site" value="google">
        </label>
    </body>
</html>
```

It can be used this way…

```
def radios = $(name: "site").module(RadioButtons)
radios.checked = "current"

assert radios.checked == "current"
assert radios.checkedLabel == "Search this site"

radios.checked = "Search Google"

assert radios.checked == "google"
assert radios.checkedLabel == "Search Google"
```

6.8. Unwrapping modules returned from the content DSL

For the sake of better error reporting, current implementation wraps any module declared within `content` block into `geb.content.TemplateDerivedPageContent` instance.

Given a page defined as follows:

```
class ModuleUnwrappingPage extends Page {
    static content = {
        theModule { module(UnwrappedModule) }
    }
}
```

And a custom module:

```
class UnwrappedModule extends Module {
    static content = {
        theContent { $(".the-content") }
    }
}
```

A module assignment to a variable of its declared type will fail with `GroovyCastException`:

```
Browser.drive {
    to ModuleUnwrappingPage
    UnwrappedModule foo = theModule   (1)
}
```

1 `GroovyCastException` is thrown

An invocation of a method which takes a module as argument with its declared type will fail with `MissingMethodException`:

```
String getContentText(UnwrappedModule module) {
    module.theContent.text()
}
```

```
Browser.drive {
    to ModuleUnwrappingPage
    getContentText(theModule)   (1)
}
```

1 `MissingMethodException` is thrown

As you may like or need to use strong typing for modules there is a way to do that. Module can be cast to its declared type with the Groovy `as` operator:

```
Browser.drive {
    to ModuleUnwrappingPage
    UnwrappedModule unwrapped = theModule as UnwrappedModule
    getContentText(theModule as UnwrappedModule)
}
```

Bear in mind that casting of a module to its declared type means the module gets unwrapped. By doing so the convenient error messages for such module are gone.

What's the trade-off? Calling `toString()` on any content element, including module, gives a meaningful path like:

```
modules.ModuleUnwrappingPage -> theModule: modules.UnwrappedModule -> theContent: geb.navigator.NonEmptyNavigator
```

Such paths can be seen in error messages and this is exactly what you are going to give away for unwrapped modules.

7. Configuration

Geb provides a configuration mechanism that allows you to control various aspects of Geb in a flexible way. At the heart of this is the `Configuration` object, which the `Browser` and other objects query at runtime.

There are three general mechanisms for influencing configuration; *system properties*, *config script* and the *build adapter*.

7.1. Mechanisms

7.1.1. The config script

Geb attempts to load a `ConfigSlurper` script named `GebConfig.groovy` from the *default package* (in other words, in the root of a directory that is on the classpath). If it is not found, Geb will try to load a `ConfigSlurper` class named `GebConfig` from the *default package* - this is useful if you run tests that use Geb from an IDE because you won't have to specify `GebConfig.groovy` as a resource, Geb will simply fall back to the compiled version of the script. If both script and class are not found Geb will continue using all defaults.

First, the script is looked for with the **executing thread's context class loader** and if it is not found, then it is looked for with the class loader that loaded Geb. This covers 99% of scenarios out of the box perfectly well without any intervention. If however you do need to configure the context class loader to load the config script, you **must** make sure that it is either the same as the class loader that loaded Geb or a child of it. If the script is not found by both of those class loaders the procedure will be repeated but this time the class will be searched for - first using **executing thread's context class loader** and then using the class loader that loaded Geb.

If you are using a build tool such as Gradle or Maven that has the concept of test "resources", then that directory is a suitable place. You can also put your script together with your compilation source and then the compiled version of the script will be used.

Environment sensitivity

The Groovy `ConfigSlurper` mechanism has built in support for environment sensitive configuration, and Geb leverages this by using the **`geb.env`** system property to determine the environment to use. An effective use of this mechanism is to configure different drivers based on the designated Geb "environment" (concrete details on how to do this further down).

How you set the environment system property is going to be dependent on the build system you are using. For example, when using Gradle you could control the Geb environment by specifying it in the configuration of the test task running your tests…

```
test {
    systemProperty 'geb.env', 'windows'
```

```
}
```

Other build environments will allow you to do this in different ways.

7.1.2. System properties

Some config options can be specified by system properties. In general, config options specified by system properties will *override* values set in the config script. See the config options below for which options are controllable via system properties.

7.1.3. Build adapter

The build adapter mechanism exists to allow Geb to integrate with development/build environments that logically dictate config options.

This mechanism works by loading the name of the class (fully qualified) by the system property `geb.build.adapter` that must implement the `BuildAdapter` interface. Currently, the build adapter can only influence the base URL to use, and the location of the reports directory.

If the `geb.build.adapter` system property is not explicitly set, it defaults to `SystemPropertiesBuildAdapter. As you can probably deduce, this default implementation uses system properties to specify values, so is usable in most circumstances. See the linked API doc for the details of the specific system properties it looks for.

> While the default build adapter uses system properties, it should not be considered to be the same as system property configuration due to values in the config script taking precedence over the build adapter which is not true for system properties.

7.2. Config options

7.2.1. Driver implementation

The driver to use is specified by the config key `driver`, or the system property `geb.driver`.

Factory closure

In the config script it can be a closure that when invoked with no arguments returns an instance of `WebDriver`…

```
import org.openqa.selenium.firefox.FirefoxDriver

driver = { new FirefoxDriver() }
```

This is the preferred mechanism, as it allows the most control over the drivers creation and configuration.

You can use the `ConfigSlurper` mechanism's environment sensitivity to configure different drivers per environment…

```
import org.openqa.selenium.htmlunit.HtmlUnitDriver

import org.openqa.selenium.remote.DesiredCapabilities
import org.openqa.selenium.remote.RemoteWebDriver

// default is to use htmlunit
driver = { new HtmlUnitDriver() }

environments {
    // when system property 'geb.env' is set to 'remote' use a remote Firefox driver
    remote {
        driver = {
            def remoteWebDriverServerUrl = new URL("http://example.com/webdriverserver")
            new RemoteWebDriver(remoteWebDriverServerUrl, DesiredCapabilities.firefox())
        }
    }
}
```

> WebDriver has the ability to drive browsers on a remote host, which is what we are using above. For more information consult the WebDriver documentation on `RemoteWebDriver` and `RemoteWebDriverServer`.

Driver class name

The name of the driver class to use (it will be constructed with no arguments) can be specified as a string with the key `driver` in the config script or via the `geb.driver` system property (the class must implement the `WebDriver` interface).

```
driver = "org.openqa.selenium.firefox.FirefoxDriver"
```

Or it can be one of the following short names: `ie`, `htmlunit`, `firefox` or `chrome`. These will be implicitly expanded to their fully qualified class names…

```
driver = "firefox"
```

The following table gives the possible short names that can be used:

Short Name	Driver
`htmlunit`	org.openqa.selenium.htmlunit.HtmlUnitDriver
`firefox`	org.openqa.selenium.firefox.FirefoxDriver
`ie`	org.openqa.selenium.ie.InternetExplorerDriver
`chrome`	org.openqa.selenium.chrome.ChromeDriver
`edge`	org.openqa.selenium.edge.EdgeDriver

If no explicit driver is specified then Geb will look for the following drivers on the classpath in the order they are listed in the above table. If none of these classes can be found, a `UnableToLoadAnyDriversException` will be thrown.

7.2.2. Navigator factory

It is possible to specify your own implementation of `NavigatorFactory` via configuration. This is useful if you want to extend the `Navigator` class to provide your own behaviour extensions.

Rather than inject your own `NavigatorFactory`, it is simpler to inject a custom `InnerNavigatorFactory` which is a much simpler interface. To do this, you can specify a closure for the config key `innerNavigatorFactory`…

```
import geb.Browser
import org.openqa.selenium.WebElement

innerNavigatorFactory = { Browser browser, List<WebElement> elements ->
    elements ? new MyCustomNavigator(browser, elements) : new MyCustomEmptyNavigator()
}
```

This is a rather advanced use case. If you need to do this, check out the source code or get in touch via the mailing list if you need help.

7.2.3. Driver caching

Geb's ability to cache a driver and re-use it for the lifetime of the JVM (i.e. the implicit driver lifecycle) can be disabled by setting the `cacheDriver` config option to `false`. However, if you do this you become responsible for quitting every driver that is created at the appropriate time.

The default caching behavior is to cache the driver globally across the JVM. If you are using Geb in multiple threads this may not be what you want, as neither Geb `Browser` objects nor WebDriver at the core is thread safe. To remedy this, you can instruct Geb to cache the driver instance per thread by setting the config option `cacheDriverPerThread` to `true`.

Also, by default Geb will register a shutdown hook to quit any cached browsers when the JVM exits. You can disable this by setting te config property `quitCachedDriverOnShutdown` to `false`.

7.2.4. Base URL

The base URL to be used can be specified by setting the `baseUrl` config property (with a `String`) value or via the build adapter (the default implementation of which looks at the `geb.build.baseUrl` system property). Any value set in the config script will take precedence over the value provided by the build adapter.

7.2.5. Waiting

The `waitFor()` methods available on browser, page and module objects can be affected by configuration (this is also true for implicitly waiting content). It is possible to specify default values for the timeout and retry interval, and to define presets of these values to be referred to by name.

Defaults

Defaults can be specified via:

```
waiting {
    timeout = 10
    retryInterval = 0.5
}
```

Both values are optional and in seconds. If unspecified, the values of `5` for `timeout` and `0.1` for `retryInterval`.

Presets

Presets can be specified via:

```
waiting {
    presets {
        slow {
            timeout = 20
            retryInterval = 1
        }
        quick {
            timeout = 1
        }
    }
}
```

Here we have defined two presets, `slow` and `quick`. Notice that the `quick` preset does not specify a `retryInterval` value; defaults will be substituted in for any missing values (i.e. giving the `quick` preset the default `retryInterval` value of `0.1`).

Failure causes

When waiting fails because the condition throws an exception be it an assertion failure or any other exception then that exception is set as the cause of `WaitTimeoutException` thrown by Geb. This usually provides fairly good diagnostics of what went wrong. Unfortunately some runtimes, namely Maven Surefire Plugin, don't print full exception stacktraces and exclude the cause from them. To make diagnostics easier in such situations it's possible to configure Geb to include string representation of the cause as part of `WaitTimeoutException` message:

```
waiting {
    includeCauseInMessage = true
}
```

7.2.6. Waiting in "at" checkers

At checkers can be configured to be implicitly wrapped with `waitFor()` calls. This can be set with:

```
atCheckWaiting = true
```

The possible values for the `atCheckWaiting` property are consistent with the ones for `wait` option of content template definitions.

This global setting can also be overridden on a per page class basis.

7.2.7. Waiting for base navigator

Sometimes Firefox driver times out when trying to find the root HTML element of the page. This manifests itself in an error similar to:

```
org.openqa.selenium.NoSuchElementException: Unable to locate element: {"method":"tag name","selector":"html"}
Command duration or timeout: 576 milliseconds
For documentation on this error, please visit: http://seleniumhq.org/exceptions/no_such_element.html
```

You can prevent this error from happening by configuring a wait timeout to use when the driver is locating the root HTML element, using:

```
baseNavigatorWaiting = true
```

The possible values for the `baseNavigatorWaiting` option are consistent with the ones for `wait` option of content template definitions.

7.2.8. Unexpected pages

The `unexpectedPages` configuration property allows to specify a list of unexpected `Page` classes that will be checked for when "at" checks are performed. Given that `PageNotFoundPage` and `InternalServerErrorPage` have been defined you can use the following to configure them as unexpected pages:

```
unexpectedPages = [PageNotFoundPage, InternalServerErrorPage]
```

See this section for more information on unexpected pages.

7.2.9. Reporter

The *reporter* is the object responsible for snapshotting the state of the browser (see the Reporting chapter for details). All reporters are implementations of the `Reporter` interface. If no reporter is explicitly defined, a composite reporter will be created from a `ScreenshotReporter` (takes a PNG screenshot) and `PageSourceReporter` (dumps the current DOM state as HTML). This is a sensible default, but should you wish to use a custom reporter you can assign it to the `reporter` config key.

```
reporter = new CustomReporter()
```

7.2.10. Reports directory

The reports dir configuration is used by to control where the browser should write reports (see the Reporting chapter for details).

In the config script, you can set the path to the directory to use for reports via the `reportsDir` key…

```
reportsDir = "target/geb-reports"
```

The value is interpreted as a path, and if not absolute will be relative to the JVM's working directory.

The reports dir can also be specified by the build adapter (the default implementation of which looks at the `geb.build.reportsDir` system property). Any value set in the config script will take precedence over the value provided by the build adapter.

It is also possible to set the `reportsDir` config item to a file.

```
reportsDir = new File("target/geb-reports")
```

By default this value is **not set**. The browser's `report()` method requires a value for this config item so if you are using the reporting features you **must** set a reports dir.

7.2.11. Report test failures only

By default Geb will take a report at the end of each test method, regardless of whether it ended successfully or not. The `reportOnTestFailureOnly` setting allows you to specify that a report should be taken only if a failure occurs. This might be useful as a way to speed up large test suites.

```
reportOnTestFailureOnly = true
```

7.2.12. Reporting listener

It is possible to specify a listener that will be notified when reports are taken. See the section on listening to reporting for details.

7.2.13. Auto clearing cookies

Certain integrations will automatically clear the driver's cookies for the current domain, which is usually necessary when using an implicit driver. This configuration flag, which is `true` by default, can be disabled by setting the `autoClearCookies` value in the config to `false`.

```
autoClearCookies = false
```

7.3. Runtime overrides

The Configuration object also has setters for all of the config properties it exposes, allowing you to override config properties at runtime in particular circumstances if you need to.

For example, you may have one Spock spec that requires the `autoClearCookies` property to be disabled. You could disable it for just this spec by doing something like…

```
import geb.spock.GebReportingSpec

class FunctionalSpec extends GebReportingSpec {

    def setup() {
        browser.config.autoClearCookies = false
    }
}
```

8. Implicit assertions

As of Geb 0.7.0, certain parts of Geb utilise "**implicit assertions**". This sole goal of this feature is to provide more informative error messages. Put simply, it means that for a given block of code, all *expressions* are automatically turned into assertions. So the following code:

```
1 == 1
```

Becomes…

```
assert 1 == 1
```

If you've used the Spock Framework you will be well familiar with the concept of implicit assertions from Spock's `then:` blocks.

In Geb, waiting expressions and at expressions automatically use implicit assertions. Take the following page object…

```
class ImplicitAssertionsExamplePage extends Page {

    static at = { title == "Implicit Assertions!" }


    def waitForHeading() {
        waitFor { $("h1") }
    }
}
```

This automatically becomes…

```
class ImplicitAssertionsExamplePage extends Page {

    static at = { assert title == "Implicit Assertions!" }

    def waitForHeading() {
        waitFor { assert $("h1") }
    }
}
```

Because of this, Geb is able to provide much better error messages when the expression fails due to Groovy's power assertions.

A special form of `assert` is used by Geb that returns the value of the expression, whereas a regular `assert` returns `null`.

This means that given…

```
static content = {
    headingText(wait: true) { $("h1").text() }
}
```

Accessing `headingText` here will *wait* for there to be a `h1` and for it to have some text (because an empty string is `false` in Groovy), which will then be returned. This means that even when implicit assertions are used, the value is still returned and it is usable.

8.1. At verification

Let's take the "at checker" case.

If you're unfamiliar with Geb's "at checking", please read this section.

Consider the following small Geb script…

```
to ImplicitAssertionsExamplePage
```

At checking works by verifying that the page's "at check" returns a *trueish* value. If it does, the `at()` method returns `true`. If not, the `at()` method will return `false`. However, due to implicit assertions, the "at check" will never return `false`. Instead, the at checker will throw an `AssertionError`. Because the page's "at check" is turned into an assertion, you'll see the following in the stacktrace:

```
Assertion failed:

title == "Implicit Assertions!"
|     |
|     false
Something else
```

As you can see, this is much more informative than the `at()` method simply returning `false`.

8.2. Waiting

Another place where implicit assertions are utilised is for *waiting*.

If you're unfamiliar with Geb's "waiting" support, please read this section.

Consider the following Geb script:

```
waitFor { title == "Page Title" }
```

The `waitFor` method verifies that the given clause returns a *trueish* value within a certain timeframe. Because of implicit assertions, when this fails you'll see the following in the stacktrace:

```
Assertion failed:

title == "Page Title"
|     |
|     false
Something else
```

The failed assertion is carried as the cause of the `geb.waiting.WaitTimeoutException` and gives you an informative message as to why the waiting failed.

Waiting content

The same implicit assertion semantics apply to content definitions that are waiting.

If you're unfamiliar with Geb's "waiting content" support, please read this section.

Any content definitions that declare a `wait` parameter have implicit assertions added to each expression just like `waitFor()` method calls.

8.3. How it works

The "implicit assertions" feature is implemented as a Groovy compile time transformation, which literally turns all expressions in a candidate block of code into assertions.

This transform is packaged as a separate JAR named `geb-implicit-assertions`. This JAR needs to be on the compilation classpath of your Geb test/pages/modules (and any other code that you want to use implicit assertions) in order for this feature to work.

If you are obtaining Geb via a dependency management system, this is typically not something you need to be concerned about as it will happen automatically. Geb is distributed via the Maven Central repository in Apache Maven format (i.e. via POM files). The main Geb module, `geb-core` depends on the `geb-implicit-assertions` module as a `compile` dependency.

If your dependency management system *inherits* transitive compile dependencies (i.e. also makes compile dependencies of first class compile dependencies first class compile dependencies) then you will automatically have the `geb-implicit-assertions` module as a compile dependency and everything will work fine (Maven, Gradle and most configurations of Ivy do this). If your dependency management system does not do this, or if you are manually managing the `geb-core` dependency, be sure to include the `geb-implicit-assertions` dependency as a compile dependency.

9. Javascript, AJAX and dynamic pages

This section discusses how to deal with some of the challenges in testing and/or automating modern web applications.

9.1. The "js" object

The browser instance exposes a "`js`" object that provides support for working with JavaScript over and above what WebDriver provides. It's important to understand how WebDriver does handle JavaScript, which is through a driver's implementation of `JavascriptExecutor.executeScript()` method.

> Before reading further, it's **strongly** recommended to read the description of `JavascriptExecutor.executeScript()` in order to understand how type conversion works between the two worlds.

You can execute JavaScript like you would with straight WebDriver using the driver instance via the browser…

```
assert browser.driver.executeScript("return arguments[0];", 1) == 1
```

This is a bit long winded, and as you would expect Geb uses the dynamism of Groovy to make life easier.

> The `JavascriptExecutor` interface does not define any contract in regards to the driver's responsibility when there is some issue executing JavaScript. All drivers however throw *some kind* of exception when this happens.

9.1.1. Accessing variables

Any *global* JavaScript variables inside the browser can be read as *properties* of the `js` object.

Given the following page…

```
<html>
    <head>
        <script type="text/javascript">
            var aVariable = 1;
        </script>
    </head>
</html>
```

We could access the JavaScript variable "`aVariable`" with…

```
Browser.drive {
    go "/"
```

```
    assert js.aVariable == 1
}
```

Or if we wanted to map it to page content…

```
Browser.drive {
    to JsVariablePage
    assert aVar == 1
}
```

We can even access *nested* variables…

```
assert js."document.title" == "Book of Geb"
```

9.1.2. Calling methods

Any *global* JavaScript functions can be called as methods on the `js` object.

Given the following page…

```
<html>
    <head>
        <script type="text/javascript">
            function addThem(a,b) {
                return a + b;
            }
        </script>
    </head>
</html>
```

We can call the `addThem()` function with…

```
Browser.drive {
    go "/"
    assert js.addThem(1, 2) == 3
}
```

This also works from pages and modules.

To call *nested* methods, we use the same syntax as for properties…

```
Browser.drive {
    go "/"
    js."document.write"("<html>Hello World!</html>")
    assert $().text() == "Hello World!"
}
```

9.1.3. Executing arbitrary code

The `js` object also has an `exec()` method that can be used to run snippets of JavaScript. It is identical to the `JavascriptExecutor.executeScript()` method, except that it takes its arguments in the other order…

```
assert js.exec(1, 2, "return arguments[0] + arguments[1];") == 3
```

You might be wondering why the order has been changed (i.e. the arguments go *before* the script). It makes writing multiline JavaScript more convenient…

```
js.exec 1, 2, """
    someJsMethod(1, 2);
    // lots of javascript
    return true;
"""
```

9.2. Waiting

Geb provides some convenient methods for *waiting* for a certain condition to be true. This is useful for testing pages using AJAX, timers or effects.

The `waitFor` methods are provided by the `WaitingSupport` mixin which delegates to the `Wait` class (see the documentation of the `waitFor()` method of this class for the precise semantics of *waiting*). These methods take various parameters that determine how long to wait for the given closure to return a true object according to the Groovy Truth, and how long to wait in between invoking the closure again.

```
waitFor {} (1)
waitFor(10) {} (2)
waitFor(10, 0.5) {} (3)
waitFor("quick") {} (4)
```

1 Use default configuration.

2 Wait for up to 10 seconds, using the default retry interval.

3 Wait for up to 10 seconds, waiting half a second in between retries. See the section on wait configuration for how to change the default values and define presets.

4 Use the preset “quick” as the wait settings

It is also possible to declare that content should be implicitly waited on, see the `wait` option for content definition.

9.2.1. Examples

Here is an example showing one way of using `waitFor()` to deal with the situation where clicking a button invokes an AJAX request that creates a new `div` on its completion.

```
class DynamicPage extends Page {
    static content = {
        theButton { $("input", value: "Make Request") }
        theResultDiv { $("div#result") }
    }

    def makeRequest() {
        theButton.click()
        waitFor { theResultDiv.present }
    }
}
```

```
Browser.drive {
    to DynamicPage
    makeRequest()
    assert theResultDiv.text() == "The Result"
}
```

Recall that the `return` keyword is optional in Groovy, so in the example above the `$("div#result").present` statement acts as the return value for the closure and is used as the basis on whether the closure *passed* or not. This means that you must ensure that the last statement inside the closure returns a value that is `true` according to the Groovy Truth (if you’re unfamiliar with the Groovy Truth **do** read that page).

Because the browser delegates method calls to the page object, the above could have been written as…

```
Browser.drive {
    go "/"
    $("input", value: "Make Request").click()
    waitFor { $("div#result") }
    assert $("div#result").text() == "The Result"
}
```

Not using explicit `return` statements in closure expressions passed to `waitFor()` is actually preferred. See the section on implicit assertions for more information.

The closures given to the `waitFor()` method(s) do not need to be single statement.

```
waitFor {
    def result = $("div#result")
    result.text() == "The Result"
}
```

That will work fine.

If you wish to *test* multiple conditions as separate statement inside a `waitFor` closure, you can just put them in separate lines.

```
waitFor {
    def result = $("div")
    result.@id == "result"
    result.text() == "The Result"
}
```

9.2.2. Custom message

If you wish to add a custom message to `WaitTimeoutException` that is being thrown when `waitFor` call times out you can do so by providing a message parameter to the `waitFor` call:

```
waitFor(message: "My custom message") { $("div#result") }
```

9.3. Alert and confirm dialogs

WebDriver currently does not handle the alert() and confirm() dialog windows. However, we can fake it through some JavaScript magic as discussed on the WebDriver issue for this. Geb implements a workaround based on this solution for you. Note that this feature relies on making changes to the browser's `window` DOM object so may not work on all browsers on all platforms. At the time when WebDriver adds support for this functionality the underlying implementation of the following methods will change to use that which will presumably be more robust. Geb adds this functionality through the `AlertAndConfirmSupport` class that is mixed into `Page` and `Module`.

The Geb methods **prevent** the browser from actually displaying the dialog, which is a good thing. This prevents the browser blocking while the dialog is displayed and causing your test to hang indefinitely.

> Unexpected `alert()` and `confirm()` calls can have strange results. This is due to the nature of how Geb handles this internally. If you are seeing strange results, you may want to run your tests/scripts against a real browser and watch what happens to make sure there aren't any alert or confirm windows being opened that you aren't expecting. To do this, you need to disable Geb's handling by changing your code to not use the methods below.

9.3.1. `alert()`

There are three methods that deal with `alert()` dialogs:

```
def withAlert(Closure actions)
def withAlert(Map params, Closure actions)
void withNoAlert(Closure actions)
```

The first method, `withAlert()`, is used to verify actions that will produce an alert dialog. This method returns the alert message.

Given the following HTML…

```
<input type="button" name="showAlert" onclick="alert('Bang!');" />
```

The `withAlert()` method is used like so…

```
assert withAlert { $("input", name: "showAlert").click() } == "Bang!"
```

If an alert dialog is not raised by the given "actions" closure, an `AssertionError` will be thrown.

The `withAlert()` method also accepts a wait option. It is useful if the code in your "actions" closure is raising a dialog in an asynchronous manner and can be used like that:

```
assert withAlert(wait: true) { $("input", name: "showAlert").click() } == "Bang!"
```

The possible values for the `wait` option are consistent with the ones for `wait` option of content definitions.

The second method, `withNoAlert()`, is used to verify actions that will not produce an `alert()` dialog. If an alert dialog is raised by the given "actions" closure, an `AssertionError` will be thrown.

Given the following HTML…

```
<input type="button" name="contShowAlert" />
```

The `withNoAlert()` method is used like so…

```
withNoAlert { $("input", name: "dontShowAlert").click() }
```

> It's a good idea to use `withNoAlert()` when doing something that *might* raise an alert. If you don't, the browser is going to raise a real alert dialog and sit there waiting for someone to click it which means your test is going to hang. Using `withNoAlert()` prevents this.

A side effect of the way that this is implemented is that we aren't able to definitively handle actions that cause the browser's actual page to change (e.g. clicking a link in the closure given to `withAlert()`/`withNoAlert()`). We can detect that the browser page did change, but we can't know if `alert()` did or did not get called before the page change. If a page change was detected the `withAlert()` method will return a literal `true` (whereas it would normally return the alert message), while the `withNoAlert()` will succeed.

9.3.2. `confirm()`

There are five methods that deal with `confirm()` dialogs:

```
def withConfirm(boolean ok, Closure actions)
def withConfirm(Closure actions)
def withConfirm(Map params, Closure actions)
def withConfirm(Map params, boolean ok, Closure actions)
void withNoConfirm(Closure actions)
```

The first method, `withConfirm()` (and its '`ok`' defaulted relative), is used to verify actions that will produce a confirm dialog. This method returns the confirmation message. The `ok` parameter controls whether the the "OK" or "Cancel" button should be clicked.

Given the following HTML…

```
<input type="button" name="showConfirm" onclick="confirm('Do you like Geb?');" />
```

The `withConfirm()` method is used like so…

```
assert withConfirm(true) { $("input", name: "showConfirm").click() } == "Do you like Geb?"
```

If a confirmation dialog is not raised by the given "actions" closure, an `AssertionError` will be thrown.

The `withConfirm()` method also accepts a wait option just like the `withAlert()` method. See the description of `withAlert()` to learn about the possible values and usage.

The other method, `withNoConfirm()`, is used to verify actions that will not produce a confirm dialog. If a confirmation dialog is raised by the given "actions" closure, an `AssertionError` will be thrown.

Given the following HTML…

```
<input type="button" name="dontShowConfirm" />
```

The `withNoConfirm()` method is used like so…

```
withNoConfirm { $("input", name: "dontShowConfirm").click() }
```

> It's a good idea to use `withNoConfirm()` when doing something that *might* raise a a confirmation. If you don't, the browser is going to raise a real confirmation dialog and sit there waiting for someone to click it, which means your test is going to hang. Using `withNoConfirm()` prevents this.

A side effect of the way that this is implemented is that we aren't able to definitively handle actions that cause the browser's actual page to change (e.g. clicking a link in the closure given to `withConfirm()`/`withNoConfirm()`). We can detect that the browser page did change, but we can't know if `confirm()` did or did not get called before the page change. If a page change was detected, the `withConfirm()` method will return a literal `true` (whereas it would normally return the alert message), while the `withNoConfirm()` will succeed.

9.3.3. `prompt()`

Geb does not provide any support for `prompt()` due to its infrequent and generally discouraged use.

9.4. jQuery integration

Geb has special support for jQuery. Navigator objects have a special adapter that makes calling jQuery methods against the underlying DOM elements simple. This is best explained by example.

> The jQuery integration only works when the pages you are working with include jQuery, Geb does not install it in the page for you. The minimum supported version of jQuery is 1.4.

Consider the following page:

```
<html>
    <head>
        <script type="text/javascript" src="/js/jquery-2.1.4.min.js"></script>
        <script type="text/javascript">
            $(function() {
                $("#a").mouseover(function() {
                    $("#b").show();
                });
            });
        </script>
    </head>
    <body>
        <div id="a"></div>
        <div id="b" style="display:none;"><a href="http://www.gebish.org">Geb!</a></div>
    </body>
</html>
```

We want to click the Geb link, but can't because it's hidden (WebDriver does not let you interact with hidden elements). The div containing the link (div "a") is only displayed when the mouse moves over div "a".

The jQuery library provides convenient methods for triggering browser events. We can use this to simulate the mouse being moved over the div "a".

In straight jQuery JavaScript we would do…

```
jQuery("div#a").mouseover();
```

Which we could invoke via Geb easy enough…

```
js.exec 'jQuery("div#a").mouseover()'
```

That will work, but can be inconvenient as it duplicates content definitions in our Geb pages. Geb's jQuery integration allows you to use your defined content in Geb with jQuery. Here is how we could call the `mouseover` jQuery function on an element from Geb…

```
$("div#a").jquery.mouseover()
```

To be clear, that is Groovy (not JavaScript code). It can be used with pages…

```
class JQueryPage extends Page {
    static content = {
        divA { $("#a") }
        divB { $("#b") }
    }
}
```

```
to JQueryPage
divA.jquery.mouseover()
assert divB.displayed
```

The `jquery` property of a navigator is conceptually equivalent to a jQuery object for *all* of the navigator's matched page elements.

The methods can also take arguments…

```
$("#a").jquery.trigger('mouseover')
```

The same set of restricted types as allowed by WebDriver's `JavascriptExecutor.executeScript()` method are permitted here.

The return value of methods called on the `jquery` property depends on what the corresponding jQuery method returns. A jQuery object will be converted to a Navigator representing the same set of elements, other values such as objects, strings and numbers are returned as per WebDriver's `JavascriptExecutor.executeScript()` method.

Why?

This functionality was developed to make triggering mouse related events easier. Some applications are very sensitive to mouse events, and triggering these events in an automated environment is a challenge. jQuery provides a good API for faking these events, which makes for a good solution. An alternative is using `the interact() method`.

10. Direct downloading

Geb features an API that can be used to make direct HTTP requests from the application that is executing the Geb scripts or tests. This facilitates fine grained requests and downloading content such as PDF, CSVs, images etc. into your scripts or tests to then do something with.

The direct download API works by using `java.net.HttpURLConnection` to directly connect to a URL from the application executing Geb, bypassing WebDriver.

The Direct Download API is provided by the `DownloadSupport` class, which is mixed in to pages and modules (which means you can just call these instance methods directly from anywhere where you would want to, e.g. drive blocks, in tests/specs, methods on page objects, methods on modules). Consult the `DownloadSupport` API reference for the various `download*()` methods that are available.

10.1. Downloading example

For example, let's say you are using Geb to exercise a web application that generates PDF documents. The WebDriver API can only deal with HTML documents. You want to hit the PDF download link and also do some tests on the downloaded PDF. The direct download API is there to fill this need.

```
class LoginPage extends Page {
    static content = {
        loginButton(to: PageWithPdfLink) { $("input", name: "login") }
    }

    void login(String user, String pass) {
        username = user
        password = pass
        loginButton.click()
    }
}

class PageWithPdfLink extends Page {
    static content = {
        pdfLink { $("a#pdf-download-link") }
    }
}
```

```
Browser.drive {
    to LoginPage
    login("me", "secret")

    def pdfBytes = downloadBytes(pdfLink.@href)
}
```

Simple enough, but consider what is happening behind the scenes. Our application required us to log in, which implies some kind of session state. Geb is using `HttpURLConnection` to get the content and before doing so the cookies from the real browser are being transferred onto the connection allowing it to reuse the same session. The PDF download link href may also be relative, and Geb handles this by resolving the link passed to the download function against the browser's current page URL.

10.2. Fine grained request

The Direct Download API can also be used for making fine grained requests which can be useful for testing edge cases or abnormal behavior.

All of the `download*()` methods take an optional closure that can configure the `HttpURLConnection` that will be used to make the request (after the `Cookie` header has been set).

For example, we could test what happens when we send `application/json` in the `Accept` header.

```
Browser.drive {
    go "/"
    def jsonBytes = downloadBytes { HttpURLConnection connection ->
        connection.setRequestProperty("Accept", "application/json")
    }
}
```

Before doing something like the above, it's worth considering whether doing such testing via Geb (a browser automation tool) is the right thing to do. You may find that it's more appropriate to directly use `HttpURLConnection` or a http client library without Geb. That said, there are scenarios where such fine grained request control can be useful.

10.3. Dealing with untrusted certificates

When facing web applications using untrusted (e.g. self-signed) SSL certificates, you will likely get exceptions when trying to use Geb's download API. By overriding the behavior of the request you can get around this kind of problem. Using the following code will allow running requests against a server which uses a certificate from the given keystore:

```
import geb.download.helper.SelfSignedCertificateHelper

def text = downloadText { HttpURLConnection connection ->
    if (connection instanceof HttpsURLConnection) {
        def keystore = getClass().getResource('/keystore.jks')
        def helper = new SelfSignedCertificateHelper(keystore, 'password')
        helper.acceptCertificatesFor(connection as HttpsURLConnection)
    }
}
```

10.4. Default configuration

In the configuration, the default behaviour of the `HttpURLConnection` object can be specified by providing a closure as the `defaultDownloadConfig` property.

The below example configures all requests executed using direct downloading support to carry a `User-Agent` header.

```
defaultDownloadConfig = { HttpURLConnection connection ->
    connection.setRequestProperty("User-Agent", "Geb")
}
```

This config closure will be run first, so anything set here can be overridden using the fine grained request configuration shown above.

10.5. Errors

Any I/O type errors that occur during a download operation (e.g. HTTP 500 responses) will result in a `DownloadException` being thrown that wraps the original exception and provides access to the `HttpURLConnection` used to make the request.

11. Scripts and binding

Geb supports being used in scripting environments via both the `Browser.drive()` method, and by using the `geb.binding.BindingUpdater` class that populates and updates a `groovy.lang.Binding` that can be used with scripts. This is also the same mechanism that can be used with Cucumber-JVM.

11.1. Setup

To use the binding support, you simply create a `BindingUpdater` object with a `Binding` and `Browser`…

```
import geb.Browser
import geb.binding.BindingUpdater

def binding = new Binding()
```

```
def browser = new Browser()
def updater = new BindingUpdater(binding, browser)

updater.initialize() (1)

def script = getClass().getResource(resourcePath).text
def result = new GroovyShell(binding).evaluate(script) (2)

updater.remove() (3)
```

1 Populate and start updating the browser.
2 Run a script from a resource loaded from the classpath.
3 Remove Geb bits from the binding and stop updating it.

11.2. The binding environment

11.2.1. Browser methods and properties

The `BindingUpdater` installs shortcuts into the binding for most of the `Browser` public methods.

The following is an example script that will work if `BindingUpdater` is initialized on its binding…

```
go "some/page"
at(SomePage)
waitFor { $("p#status").text() == "ready" }
js.someJavaScriptFunction()
downloadText($("a.textFile").@href)
```

In a managed binding, all of the methods/properties that you can usually call in the `Browser.drive()` method are available. This includes the `$()` method.

The following methods are available:

- `$()`
- `go()`
- `to()`
- `via()`
- `at()`
- `waitFor()`
- `withAlert()`
- `withNoAlert()`
- `withConfirm()`
- `withNoConfirm()`
- `download()`
- `downloadStream()`
- `downloadText()`
- `downloadBytes()`
- `downloadContent()`
- `report()`
- `reportGroup()`
- `cleanReportGroupDir()`

The JavaScript interface property `js` is also available. The `Browser` object itself is available as the `browser` property.

11.2.2. The current page

The binding updater also updates the `page` property of the binding to be the browser's current page…

```
import geb.Page

class InitialPage extends Page {
    static content = {
        button(to: NextPage) { $("input.do-stuff") }
    }
}

class NextPage extends Page {
}

to InitialPage
assert page instanceof InitialPage
page.button.click()
assert page instanceof NextPage
```

12. Reporting

Geb includes a simple reporting mechanism which can be used to snapshot the state of the browser at any point in time. Reporters are implementations of the `Reporter` interface. Geb ships with two implementations: `PageSourceReporter` and `ScreenshotReporter`. There are three bits of configuration that pertain to reporting: the reporter implementation, the reports directory and whether to only report test failures or not.

If no reporter is explicitly defined, a composite reporter will be created from a `ScreenshotReporter` (takes a PNG screenshot) and `PageSourceReporter` (dumps the current DOM state as HTML).

You take a report by calling the `report(String label)` method on the browser object.

```
Browser.drive {
    go "http://google.com"
    report "google home page"
}
```

The `report()` method will thrown an exception if it is called and there is no configured `reportsDir`. If you are going to use reporting you **must** specify a `reportsDir` via config.

Assuming that we configured a `reportsDir` of "`reports/geb`", after running this script we will find two files in this directory:

- `google home page.html` - A HTML dump of the page source
- `google home page.png` - A screenshot of the browser as a PNG file (if the driver implementation supports this)

To avoid issues with reserved characters in filenames, Geb replaces any character in the report name that is not an alphanumeric, a space or a hyphen with an underscore.

12.1. The report group

The configuration mechanism allows you to specify the base `reportsDir` which is where reports are written to by default. It is also possible to change the report group to a relative path inside this directory.

```
Browser.drive {
    reportGroup "google"
    go "http://google.com"
    report "home page"

    reportGroup "github"
    go "http://github.com"
```

```
    report "home page"
}
```

We have now created the following files inside the `reportsDir`…

- `google/home page.html`
- `google/home page.png`
- `wikipedia/home page.html`
- `wikipedia/home page.png`

The browser will create the directory for the report group as needed. By default, the report group is not set which means that reports are written to the base of the `reportsDir`. To go back to this after setting a report group, simply call `reportGroup(null)`.

It is common for test integrations to manage the report group for you, setting it to the name of the test class.

12.2. Listening to reporting

It is possible to register a listener on the reporter that gets notified when a report is taken. This was added to make it possible to write something to stdout when a report is taken, which is how the Jenkins JUnit Attachments Plugin makes it possible to associate arbitrary files to test execution. Reporting listeners are of type `ReportingListener` can be specified as part of the config…

```
import geb.report.*

reportingListener = new ReportingListener() {
    void onReport(Reporter reporter, ReportState reportState, List<File> reportFiles) {
        reportFiles.each {
            println "[[ATTACHMENT|$it.absolutePath]]"
        }
    }
}
```

12.3. Cleaning

Geb does not automatically clean the reports dir for you. It does however provide a method that you can call to do this.

```
Browser.drive {
    cleanReportGroupDir()
    go "http://google.com"
    report "home page"
}
```

The `cleanReportGroupDir()` method will remove whatever the reports group dir is set to at the time. If it cannot do this it will throw an exception.

The Spock, JUnit and TestNG test integrations **do** automatically clean the reports dir for you, see the section in the testing chapter on these integrations.

13. Testing

Geb provides first class support for functional web testing via integration with popular testing frameworks such as Spock, JUnit, TestNg and Cucumber-JVM.

13.1. Spock, JUnit & TestNG

The Spock, JUnit and TestNG integrations work fundamentally the same way. They provide subclasses that setup a `Browser` instance that all method calls and property accesses/references resolve against via Groovy's `methodMissing` and `propertyMissing` mechanism.

Recall that the browser instance also forwards any method calls or property accesses/references that it can't handle to its current page

object, which helps to remove a lot of noise from the test.

Consider the following Spock spec…

```
import geb.spock.GebSpec

class FunctionalSpec extends GebSpec {
    def "go to login"() {
        when:
        go "/login"

        then:
        title == "Login Screen"
    }
}
```

Which is equivalent to…

```
import geb.spock.GebSpec

class FunctionalSpec extends GebSpec {
    def "verbosely go to login"() {
        when:
        browser.go "/login"

        then:
        browser.page.title == "Login Screen"
    }
}
```

13.1.1. Configuration

The browser instance is created by the testing integrations. The configuration mechanism allows you to control aspects such as the driver implementation and base URL.

13.1.2. Reporting

The Spock, JUnit and TestNG integrations also ship a superclass (the name of the class for each integration module is provided below) that automatically takes reports at the end of test methods with the label “end”. They also set the report group to the name of the test class (substituting “.” with “/”).

The `report(String label)` browser method is replaced with a specialised version. This method works the same as the browser method, but adds counters and the current test method name as prefixes to the given label.

```
package my.tests

import geb.spock.GebReportingSpec

class ReportingFunctionalSpec extends GebReportingSpec {
    def "login"() {
        when:
        go "/login"
        username = "me"
        report "login screen" (1)
        login().click()

        then:
        title == "Logged in!"
    }
}
```

1 Take a report of the login screen.

Assuming a configured `reportsDir` of `reports/geb` and the default reporters (i.e. `ScreenshotReporter` and `PageSourceReporter`), we would find the following files:

- `reports/geb/my/tests/ReportingFunctionalSpec/001-001-login-login screen.html`
- `reports/geb/my/tests/ReportingFunctionalSpec/001-001-login-login screen.png`

- `reports/geb/my/tests/ReportingFunctionalSpec/001-002-login-end.html`
- `reports/geb/my/tests/ReportingFunctionalSpec/001-002-login-end.png`

The report file name format is:

`«test method number»-«report number in test method»-«test method name»-«label».«extension»`

Reporting is an extremely useful feature and can help you diagnose test failures much easier. Wherever possible, favour the use of the auto-reporting base classes.

13.1.3. Cookie management

The Spock, JUnit and TestNG integrations will automatically clear the browser's cookies for the current domain at the end of each test method. For JUnit 3 this happens in the `tearDown()` method in `geb.junit3.GebTest`, for JUnit 4 it happens in an `@After` method in `geb.junit4.GebTest` and for TestNG it happens in an `@AfterMethod` method in `geb.testng.GebTestTrait`.

The `geb.spock.GebSpec` class will clear the cookies in the `cleanup()` method unless the spec is `@Stepwise`, in which case they are cleared in `cleanupSpec()` (meaning that all feature methods in a stepwise spec share the same browser state).

This auto-clearing of cookies can be disabled via configuration.

If you need to clear cookies in multiple domains you will need to manually track the urls and call `clearCookies(String… additionalUrls)`.

13.1.4. JAR and class names

The following table illustrates the specific JARs and class names for various test frameworks that Geb integrates with.

Framework	JAR	Base Class / Trait	Reporting Base Class / Trait
Spock	geb-spock	`geb.spock.GebSpec`	`geb.spock.GebReportingSpec`
JUnit 4	geb-junit4	`geb.junit4.GebTest`	`geb.junit4.GebReportingTest`
JUnit 3	geb-junit3	`geb.junit3.GebTest`	`geb.junit3.GebReportingTest`
TestNG	geb-testng	`geb.testng.GebTestTrait`	`geb.testng.GebReportingTestTrait`

13.1.5. Example projects

The following projects can be used as starting references:

- geb-example-gradle

13.2. Cucumber (Cucumber-JVM)

It is possible to both:

- Write your own Cucumber-JVM steps that manipulate Geb
- Use a library of pre-built steps that drives Geb to do many common tasks

13.2.1. Writing your own steps

Use Geb's binding management features to bind a browser in before / after hooks, often in a file named `env.groovy`:

```
def bindingUpdater
Before() { scenario ->
    bindingUpdater = new BindingUpdater(binding, new Browser())
    bindingUpdater.initialize()
```

```
}

After() { scenario ->
    bindingUpdater.remove()
}
```

Then normal Geb commands and objects are available in your Cucumber steps:

```
import static cucumber.api.groovy.EN.*

Given(~/I am on the DuckDuckGo search page/) { ->
    to DuckDuckGoHomePage
    waitFor { at(DuckDuckGoHomePage) }
}

When(~/I search for "(.*)"/) { String query ->
    page.search.value(query)
    page.searchButton.click()
}

Then(~/I can see some results/) { ->
    assert at(DuckDuckGoResultsPage)
}

Then(~/the first link should be "(.*)"/) { String text ->
    waitFor { page.results }
    assert page.resultLink(0).text()?.contains(text)
}
```

13.2.2. Using pre-built steps

The geb-cucumber project has a set of pre-built cucumber steps that drive Geb. So for example a feature with steps similar to the above would look like:

```
When I go to the duck duck go home page
And I enter "cucumber-jvm github" into the search field
And I click the search button
Then the results table 1st row link matches /cucumber\/cucumber-jvm · GitHub.*/
```

See geb-cucumber for more examples.

geb-cucumber also does Geb binding automatically, so if it is picked up you don't need to do it yourself as above.

13.2.3. Example project

The following project has examples of both writing your own steps and using geb-cucumber:

- geb-example-cucumber-jvm

14. Cloud browser testing

When you want to perform web testing on multiple browsers and operating systems, it can be quite complicated to maintain machines for each of the target environments. There are a few companies that provide "remote web browsers as a service", making it easy to do this sort of matrix testing without having to maintain the multiple browser installations yourself. Geb provides easy integration with two such services, SauceLabs and BrowserStack. This integration includes two parts: assistance with creating a driver in `GebConfig.groovy` and a Gradle plugin.

14.1. Creating a driver

For both SauceLabs and BrowserStack, a special driver factory is provided that, given a browser specification as well as an username and access key, creates an instance of `RemoteWebDriver` configured to use a browser in the cloud. Examples of typical usage in `GebConfig.groovy` are included below. They will configure Geb to run in SauceLabs/BrowserStack if the appropriate system property is set, and if not it will use whatever driver that is configured. This is useful if you want to run the code in a local browser for development. In theory you could use any system property to pass the browser specification but `geb.saucelabs.browser`/`geb.browserstack.browser` are also used by the Geb Gradle plugins, so it's a good idea to stick with those property names.

The first parameter passed to the `create()` method is a "browser specification" and it should be a list of required browser capabilities in Java properties file format:

```
browserName=«browser name as per values of fields in org.openqa.selenium.remote.BrowserType»
platform=«platform as per enum item names in org.openqa.selenium.Platform»
version=«version»
```

Assuming you're using the following snippet in your `GebConfig.groovy` to execute your code via SauceLabs with Firefox 19 on Linux, you would set the `geb.saucelabs.browser` system property to:

```
browserName=firefox
platform=LINUX
version=19
```

and to execute it with IE 9 on Vista to:

```
browserName=internet explorer
platform=VISTA
version=9
```

Some browsers like Chrome automatically update to the latest version; for these browsers you don't need to specify the version as there's only one, and you would use something like:

```
browserName=chrome
platform=MAC
```

as the "browser specification". For a full list of available browsers, versions and operating systems refer to your cloud provider's documentation:

- SauceLabs platform list
- BrowserStack Browsers and Platforms list

Please note that Geb Gradle plugins can set the `geb.saucelabs.browser`/`geb.browserstack.browser` system properties for you using the aforementioned format.

Following the browser specification are the username and access key used to identify your account with the cloud provider. The example uses two environment variables to access this information. This is usually the easiest way of passing something secret to your build in open CI services like drone.io or Travis CI if your code is public, but you can use other mechanisms if desired.

You can optionally pass additional configuration settings by providing a Map to the `create()` method as the last parameter. The configuration options available are described in your cloud provider's documentation:

- SauceLabs additional config
- BrowserStack Capabilities

Finally, there is also an overloaded version of `create()` method available that doesn't take a string specification and allows you to simply specify all the required capabilities using a map. This method might be useful if you just want to use the factory, but don't need the build level parametrization.

14.1.1. `SauceLabsDriverFactory`

The following is an example of utilizing `SauceLabsDriverFactory` in `GebConfig.groovy` to configure a driver that will use a browser provided in the SauceLabs cloud.

```
def sauceLabsBrowser = System.getProperty("geb.saucelabs.browser")
if (sauceLabsBrowser) {
    driver = {
       def username = System.getenv("GEB_SAUCE_LABS_USER")
       assert username
       def accessKey = System.getenv("GEB_SAUCE_LABS_ACCESS_PASSWORD")
       assert accessKey
       new SauceLabsDriverFactory().create(sauceLabsBrowser, username, accessKey)
    }
}
```

14.1.2. `BrowserStackDriverFactory`

The following is an example of utilizing `BrowserStackDriverFactory` in `GebConfig.groovy` to configure a driver that will use a browser provided in the BrowserStack cloud.

```
def browserStackBrowser = System.getProperty("geb.browserstack.browser")
if (browserStackBrowser) {
    driver = {
       def username = System.getenv("GEB_BROWSERSTACK_USERNAME")
       assert username
       def accessKey = System.getenv("GEB_BROWSERSTACK_AUTHKEY")
       assert accessKey
       new BrowserStackDriverFactory().create(browserStackBrowser, username, accessKey)
    }
}
```

If using `localIdentifier` support:

```
def browserStackBrowser = System.getProperty("geb.browserstack.browser")
if (browserStackBrowser) {
    driver = {
       def username = System.getenv("GEB_BROWSERSTACK_USERNAME")
       assert username
       def accessKey = System.getenv("GEB_BROWSERSTACK_AUTHKEY")
       assert accessKey
       def localId = System.getenv("GEB_BROWSERSTACK_LOCALID")
       assert localId
       new BrowserStackDriverFactory().create(browserStackBrowser, username, accessKey, localId)
    }
}
```

14.2. Gradle plugins

For both SauceLabs and BrowserStack, Geb provides a Gradle plugin which simplifies declaring the account and browsers that are desired, as well as configuring a tunnel to allow the cloud provider to access local applications. These plugins allow easily creating multiple `Test` tasks that will have the appropriate `geb.PROVIDER.browser` property set (where *PROVIDER* is either `saucelabs` or `browserstack`). The value of that property can be then passed in configuration file to `SauceLabsDriverFactory`/`BrowserStackDriverFactory` as the ”browser specification“. Examples of typical usage are included below.

14.2.1. geb-saucelabs plugin

Following is an example of using the geb-saucelabs Gradle plugin.

```
import geb.gradle.saucelabs.SauceAccount

apply plugin: "geb-saucelabs" (1)

buildscript { (2)
    repositories {
       mavenCentral()
    }
    dependencies {
       classpath 'org.gebish:geb-gradle:1.1.1'
    }
}

repositories { (3)
    maven { url "http://repository-saucelabs.forge.cloudbees.com/release" }
}

dependencies { (4)
    sauceConnect "com.saucelabs:ci-sauce:1.81"
}

sauceLabs {
    browsers { (5)
       firefox_linux_19
       chrome_mac
       delegate."internet explorer_vista_9"
       nexus4 { (6)
         capabilities(
            browserName: "android",
            platform: "Linux",
            version: "4.4",
            deviceName: "LG Nexus 4"
         )
       }
    }
```

```
    task { (7)
       testClassesDir = test.testClassesDir
       testSrcDirs = test.testSrcDirs
       classpath = test.classpath
    }
    account { (8)
       username = System.getenv(SauceAccount.USER_ENV_VAR)
       accessKey = System.getenv(SauceAccount.ACCESS_KEY_ENV_VAR)
    }
    connect { (9)
       port = 4444 (10)
       additionalOptions = ['--proxy', 'proxy.example.com:8080'] (11)
    }
}
```

1 Apply the plugin to the build.

2 Specify how to resolve the plugin.

3 Declare a repository for resolving SauceConnect.

4 Declare version of SauceConnect to be used as part of the `sauceConnect` configuration. This will be used by tasks that open a SauceConnect tunnel before running the generated test tasks which means that the browsers in the cloud will have localhost pointing at the machine running the build.

5 Declare that tests should run in 3 different browsers using the shorthand syntax; this will generate the following `Test` tasks: `firefoxLinux19Test`, `chromeMacTest` and `internet explorerVista9Test`.

6 Explicitly specify the required browser capabilities if the shorthand syntax doesn't allow you to express all needed capabilities; the example will generate a `Test` task named `nexus4Test`.

7 Configure all of the generated test tasks; for each of them the closure is run with delegate set to a test task being configured.

8 Pass credentials for SauceConnect.

9 Additionally configure SauceConnect if desired.

10 Override the port used by SauceConnect, defaults to 4445.

11 Pass additional command line options to SauceConnect.

You can use `allSauceLabsTests` task that will depend on all of the generated test tasks to run all of them during a build.

14.2.2. geb-browserstack plugin

Following is an example of using the geb-browserstack Gradle plugin.

```
import geb.gradle.browserstack.BrowserStackAccount

apply plugin: "geb-browserstack" (1)

buildscript { (2)
    repositories {
       mavenCentral()
    }
    dependencies {
       classpath 'org.gebish:geb-gradle:1.1.1'
    }
}

browserStack {
    application 'http://localhost:8080' (3)
    forceLocal = true (4)
    browsers { (5)
       firefox_mac_19
       chrome_mac
       delegate."internet explorer_windows_9"
       nexus4 { (6)
         capabilities browserName: "android", platform: "ANDROID", device: "Google Nexus 4"
       }
    }
    task { (7)
       testClassesDir = test.testClassesDir
       testSrcDirs = test.testSrcDirs
       classpath = test.classpath
    }
    account { (8)
       username = System.getenv(BrowserStackAccount.USER_ENV_VAR)
```

```
        accessKey = System.getenv(BrowserStackAccount.ACCESS_KEY_ENV_VAR)
    }
}
```

1 Apply the plugin to the build.

2 Specify how to resolve the plugin.

3 Specify which urls the BrowserStack Tunnel should be able to access. Multiple applications can be specified. If no applications are specified, the tunnel will not be restricted to particular URLs.

4 Configure BrowserStack tunnel to route all traffic via the local machine. This configuration property controls the `-forcelocal` flag and the default value for it is `false`.

5 Declare that tests should run in 3 different browsers using the shorthand syntax; this will generate the following `Test` tasks: `firefoxLinux19Test`, `chromeMacTest` and `internet explorerVista9Test`.

6 Explicitly specify the required browser capabilities if the shorthand syntax doesn't allow you to express all needed capabilities; the example will generate a `Test` task named `nexus4Test`.

7 Configure all of the generated test tasks; for each of them the closure is run with delegate set to a test task being configured.

8 Pass credentials for BrowserStack.

It's also possible to specify location and credentials for the proxy to be used with the BrowserStack Tunnel:

```
browserStack {
    account {
        proxyHost = '127.0.0.1'
        proxyPort = '8080'
        proxyUser = 'user'
        proxyPass = 'secret'
    }
}
```

You can use `allBrowserStackTests` task that will depend on all of the generated test tasks to run all of them during a build.

15. Build system & framework integrations

This kind of integration for Geb is typically focused on managing the base URL and reports dir, as build systems tend to be able to provide this configuration (via the build adapter mechanism).

15.1. Gradle

Using Geb with Gradle simply involves pulling in the appropriate dependencies, and configuring the base URL and reports dir in the build script if they are necessary.

Below is a valid Gradle `build.gradle` file for working with Geb for testing.

```
apply plugin: "groovy"

repositories {
    mavenCentral()
}

dependencies {
    groovy "org.codehaus.groovy:groovy-all:2.4.7"

    def gebVersion = "1.1.1"
    def seleniumVersion = "2.52.0"

    // If using Spock, need to depend on geb-spock
    testCompile "org.gebish:geb-spock:1.1.1"
    testCompile "org.spockframework:spock-core:1.0-groovy-2.4"

    // If using JUnit, need to depend on geb-junit (3 or 4)
    testCompile "org.gebish:geb-junit4:1.1.1"
    testCompile "junit:junit-dep:4.8.2"

    // Need a driver implementation
    testCompile "org.seleniumhq.selenium:selenium-firefox-driver:2.52.0"
    testRuntime "org.seleniumhq.selenium:selenium-support:2.52.0"
}
```

```
test {
    systemProperties "geb.build.reportsDir": "$reportsDir/geb"
}
```

There is a Gradle example project available.

- geb-example-gradle

15.2. Maven

Using Geb with Maven simply involves pulling in the appropriate dependencies, and configuring the base URL and reports dir in the build script if they are necessary.

Below is a valid `pom.xml` file for working with Geb for testing (with Spock).

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.gebish.example</groupId>
  <artifactId>geb-maven-example</artifactId>
  <packaging>jar</packaging>
  <version>1</version>
  <name>Geb Maven Example</name>
  <url>http://www.gebish.org</url>
  <dependencies>
    <dependency>
      <groupId>org.codehaus.groovy</groupId>
      <artifactId>groovy-all</artifactId>
      <version>2.4.7</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.spockframework</groupId>
      <artifactId>spock-core</artifactId>
      <version>1.0-groovy-2.4</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.gebish</groupId>
      <artifactId>geb-spock</artifactId>
      <version>1.1.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-firefox-driver</artifactId>
      <version>2.52.0</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.seleniumhq.selenium</groupId>
      <artifactId>selenium-support</artifactId>
      <version>2.52.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.18.1</version>
        <configuration>
          <includes>
            <include>*Spec.*</include>
          </includes>
          <systemPropertyVariables>
            <geb.build.baseUrl>http://google.com/ncr</geb.build.baseUrl>
            <geb.build.reportsDir>target/test-reports/geb</geb.build.reportsDir>
```

```
          </systemPropertyVariables>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.gmaven</groupId>
        <artifactId>gmaven-plugin</artifactId>
        <version>1.3</version>
        <configuration>
          <providerSelection>1.7</providerSelection>
        </configuration>
        <executions>
          <execution>
            <goals>
              <goal>testCompile</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

There is a Maven example project available.

- geb-example-maven

16. IDE support

Geb does not require any special plugins or configuration for use inside an IDE. However, there are some considerations that will be addressed in this chapter.

16.1. Execution

Geb *scripts* can be executed in an IDE if that IDE supports executing Groovy scripts. All IDEs that support Groovy typically support this. There are typically only two concerns in the configuration of this: getting the Geb classes on the classpath, and the `GebConfig.groovy` file.

Geb *tests* can be executed in an IDE if that IDE supports Groovy scripts and the testing framework that you are using with Geb. If you are using JUnit or Spock (which is based on JUnit) this is trivial, as all modern Java IDEs support JUnit. As far as the IDE is concerned, the Geb test is simply a JUnit test and no special support is required. As with executing scripts, the IDE must put the Geb classes on the classpath for test execution and the `GebConfig.groovy` file must be accessible (typically putting this file at the root of the test source tree is sufficient).

In both cases, the simplest way to create such an IDE configuration is to use a build tool (such as Gradle or Maven) that supports IDE integration. This will take care of the classpath setup and other concerns.

16.2. Authoring assistance (autocomplete and navigation)

This section discusses what kind of authoring assistance can be provided by IDEs and usage patterns that enable better authoring support.

16.2.1. Dynamism and conciseness vs tooling support

Geb heavily embraces the dynamic typing offered by Groovy, to achieve conciseness for the purpose of readability. This immediately reduces the amount of authoring assistance that an IDE can provide when authoring Geb code. This is an intentional compromise. The primary cost in functional/acceptance testing is in the *maintenance* of the test suite over time. Geb optimizes for this in several ways, one of which being the focus on intention revealing code (which is achieved through conciseness).

That said, if authoring support is a concern for you, read on to learn for details on ways to forsake conciseness in order to improve authoring support.

16.2.2. Strong typing

In order to gain improved authoring support, you must include types in your tests and page objects. Additionally, you must explicitly access the browser and page objects instead of relying on dynamic dispatch.

Here’s an example of idiomatic (untyped) Geb code.

```
to HomePage
loginPageLink.click()

at LoginPage
username = "user1"
password = "password1"
loginButton.click()

at SecurePage
```

The same code written with types would look like:

```
HomePage homePage = browser.to HomePage
homePage.loginPageLink.click()

LoginPage loginPage = browser.at LoginPage
SecurePage securePage = loginPage.login("user1", "password1")
```

Where the page objects are:

```
class HomePage extends Page {
    Navigator getLoginPageLink() {
        $("#loginLink")
    }
}

class LoginPage extends Page {

    static at = { title == "Login Page" }

    Navigator getLoginButton() {
        $("input", type: "submit")
    }

    SecurePage login(String username, String password) {
        $(name: "username").value username
        $(name: "password").value password
        loginButton.click()
        browser.at SecurePage
    }
}

class SecurePage extends Page {
    static at = { title == "Secure Page" }
}
```

In summary:

1. Use the `browser` object explicitly (made available by the testing adapters)
2. Use the page instance returned by the `to()` and `at()` methods instead of calling through the browser
3. Use methods on the `Page` classes instead of the `content {}` block and dynamic properties
4. If you need to use content definition options like `required:` and `wait:` then you can still reference content elements defined using the DSL in methods on `Page` and `Module` classes as usual, e.g.:

```
static content = {
    async(wait: true) { $("#async") }
}

String asyncText() {
    async.text() (1)
}
```

1 Wait here for the async definition to return a non-empty Navigator…

Using this “typed” style is not an all or nothing proposition. The typing options exist on a spectrum and can be used selectively where/when the cost of the extra “noise” is worth it to achieve better IDE support. For example, a mix of using the `content {}` DSL and methods can easily be used. The key enabler is to capture the result of the `to()` and `at()` methods in order to access the page object instance.

IntelliJ IDEA support

IntelliJ IDEA (since version 12) has special support for authoring Geb code. This is built in to the Groovy support; no additional installations are required.

The support provides:

- Understanding of implicit browser methods (e.g. `to()`, `at()`) in test classes (e.g. `extends GebSpec`)
- Understanding of content defined via the Content DSL (within `Page` and `Module` classes only)
- Completion in `at {}` and `content {}` blocks

This effectively enables more authoring support with less explicit type information. The Geb development team would like to thank the good folks at JetBrains for adding this explicit support for Geb to IDEA.

17. About the project

The Geb home page can be found at http://www.gebish.org.

17.1. API reference

The API reference can be found here.

17.2. Support & development

Support for Geb is offered on the geb-user@googlegroups.com mailing list, which can be subscribed to here.

Ideas and new features for Geb can be discussed on the geb-dev@googlegroups.com mailing list, which can be subscribed to here.

17.3. Credits

17.3.1. Committers

- Luke Daley
- Marcin Erdmann
- Chris Prior

17.3.2. Contributors

- Robert Fletcher
- Peter Niederwieser
- Alexander Zolotov - TestNG Integration
- Christoph Neuroth - Various fixes and patches
- Antony Jones - Various fixes and patches, doc improvements
- Jan-Hendrik Peters - Doc improvements
- Tomás Lin - Doc improvements
- Jason Cahoon - Bug fix around text matchers
- Tomasz Kalkosiński - Doc improvements
- Rich Douglas Evans - Doc improvements
- Ian Durkan - Doc improvements

- Colin Harrington - Doc improvements
- Bob Herrmann - Doc improvements
- George T Walters II - Page option support for `withWindow()`
- Craig Atkinson - Doc improvements
- Andy Duncan - Fail fast when unexpected pages are encountered
- John Engelman - Grails integration improvements
- Michael Legart - Grails integration improvements
- Graeme Rocher - Grails integration improvements
- Craig Atkinson - Bug fix around unexpected pages, fix for [#422]
- Ken Geis - Doc improvements
- Kelly Robinson - Additional configuration parameters for SauceLabs
- Todd Gerspacher - Doc improvements, Cleaned up settings.gradle
- David M. Carr - BrowserStack integration
- Tom Dunstan - Cucumber integration and related documentation
- Brian Kotek - Doc improvements
- David W Millar - Doc improvements
- Ai-Lin Liou - Doc improvements
- Varun Menon - Selenium By selector support and related documentation, Support navigating to page instances in addition to classes
- Anders D. Johnson - Doc improvements
- Hiroyuki Ohnaka - Doc improvements
- Erik Pragt - Initial migration of the manual to AsciiDoctor
- Vijay Bolleypally - Various fixes
- Pierre Hilt - `hasNot()` filtering
- Yotaro Takahashi - Doc improvements
- Jochen Berger - Better error reporting when trying to set a nonexistent select option
- Matan Katz - Support for setting the `forcelocal` flag for BrowserStack tunnel
- Victor Parmar - Add configuration option for including cause string representation in message of `WaitTimeoutException`
- Berardino la Torre - Fix incorrect delegation of waitFor() methods
- Markus Schlichting - Doc fixes
- Andrey Hitrin - Improvement to character replacing in report file names
- Leo Fedorov - Support for `reportOnTestFailureOnly` config option for Spock integration
- Chris Byrneham - Ability to specify proxy location and credentials to use with BrowserStack tunnel
- Aseem Bansal - Doc improvements
- Tomasz Przybysz - Unwrapping module to its declared type
- Brian Stewart - Doc improvements

- gihtub-profile:JacobAae[Jacob Aae Mikkelsen] - Various fixes and improvements
- Patrick Radtke - Doc improvements
- Leonard Brünings - Fix for unexpected pages only being checked after at check fails for the expected page

17.4. History

This page lists the high level changes between versions of Geb.

1.1.1

Fixes

- Do not double encode query parameters when building urls for arguments passed to `go()`, `to()` and `via()`. [#469]

1.1

Fixes

- Delegate to `browser` instead of the module from blocks passed to `withFrame()` in module classes. [#461]
- Fix implicit assertions in “at checkers” to not return `null` if the last call is to a void method. [#462]

Improvements

- Support for selecting Edge as the browser using name in configuration. [#425]
- Support for using url fragment identifiers when navigating to pages. [#463]
- Unexpected pages are only checked after at check fails for the expected page. [#450]
- Support equality checking between all core implementations of `Navigator`, based on comparing collections of web elements wrapped by them. [#459]
- Support using label text to select checkboxes and using collections as value to select multiple checkboxes when dealing a number of checkboxes with the same name. [#460]

Deprecations

- Grails 2.x plugin has been discontinued. [#456]

1.0

Fixes

- Fix the direct field operator shortcut (`@`) for accessing element attributes to work on classes extending `Module`. [#438]
- Fix reporting on failure only in `GebReportingSpec` to work with Spock 1.1. [#445]

Improvements

- Add ability to unwrap modules returned from content dsl to their original type. [#434]
- Add support for using attribute css selectors with navigator filtering methods like `filter()`, `not()`, `closest()`, etc. [#437]

Breaking changes

- `geb.testng.GebTest` and `geb.testng.GebReportingTest` which were deprecated in 0.13.0 have been removed.
- `isDisabled()`, `isEnabled()`, `isReadOnly()` and `isEditable()` methods of `Navigator` which were deprecated in 0.12.0 have been removed.

- Loosely typed `module()` and `moduleList()` methods of the content DSL which were deprecated in 0.12.0 have been removed.

0.13.1

Fixes

- Fix a MissingMethodException thrown from `Navigator.value()` when using Groovy versions < 2.4.0. [#422]
- Don't unnecessarily synchronize methods of `CachingDirverFactory.ThreadLocalCache`. [#421]
- Ensure `ConfigSlurper.parse(Script, URL)` is called in a thread safe way from `ConfigurationLoader`. [#423]

0.13.0

New features

- `reportOnTestFailureOnly` config option is now also effective in Spock and JUnit4 integrations. [#92]
- `isFocused()` method has been added to the `Navigator` class. [#208]
- `InvalidPageContent` exception is thrown when a content is defined with a name that will result in that content being not accessible. [#109] [#122]
- Ability to specify proxy location and credentials to use with BrowserStack tunnel. [#419]

Fixes

- Fix a bug that caused reports for all but the last executed test class in TestNG integration to be wiped out. [#407]
- Fix a bug preventing using module as a base of another module. [#411]
- Restore `browser` property of `Module`. [#416]
- Handle setting values of form elements that cause page change or reload when their value changes. [#155]

Improvements

- Non-ASCII word characters are not longer replaced in report file names. [#399]
- Change TestNG support to be based on traits. [#412]
- Add `Navigator.moduleList()` methods as an alternative to the deprecated `moduleList()` methods available in the content DSL. [#402]
- Add support for using Geb with Selendroid and other Selenium based frameworks for testing non-web applications. [#320]
- Improve documentation for `Browser.clearCookies()` around what exactly is cleared, add a helper method for removing cookies across multiple domains. [#159]
- Don't depend on UndefinedAtCheckerException for flow control. [#368]
- Document that `Navigator.text()` returns the text of the element only if it's visible. [#403]
- Make implementation of `interact()` less dynamic. [#190]
- Improve documentation for `interact()`. [#207]
- Don't unnecessarily request tag name and type attribute multiple times when setting text input values. [#417]
- Improve usefulness of string representation of content elements. [#274]

Deprecations

- `geb.testng.GebTest` and `geb.testng.GebReportingTest` have been deprecated in favour of `geb.testng.GebTestTrait` and `geb.testng.GebReportingTestTrait` respectively.

Breaking changes

- Geb is now built with Groovy 2.4.5 and Spock 1.0-groovy-2.4.
- The following `Navigator` methods now throw `SingleElementNavigatorOnlyMethodException` when called on a multi element `Navigator`: `hasClass(java.lang.String)`, `is(java.lang.String)`, `isDisplayed()`, `isDisabled()`, `isEnabled()`, `isReadOnly()`, `isEditable()`, `tag()`, `text()`, `getAttribute(java.lang.String)`, `attr(java.lang.String)`, `classes()`, `value()`, `click()`, `getHeight()`, `getWidth()`, `getX()`, `getY()`, `css(java.lang.String)`, `isFocused()`. [#284]

0.12.2

Fixes

- Fix incorrect delegation in variant of `waitFor()` that takes timeout and interval. [#395]
- Fix NPE on implicitly asserted statements that contain a safely navigated method call on null target. [#398]

0.12.1

Fixes

- Change implementation of `waitFor()` method delegation so that IntelliJ does not complain that page and module classes supposedly need to implement it. [#391]
- Properly handle class attribute when it's passed to `$()` together with a css selector. [#390]

0.12.0

New features

- Support for finding elements using Webdriver's `By` selectors. [#348]
- Support for navigating to page instances in addition to classes. [#310]
- Support for using page instances as `page` option value of window switching methods. [#352]
- Support for using page instances together with frame switching methods. [#354]
- Support for using page instances with `Navigator.click()` methods. [#355]
- Support for using page instances and lists of page instances as `page` option value of content templates. [#356]
- New `Navigator.module(Class<? extends Module>)` and `Navigable.module(Class<? extends Module>)`. [#312]
- New `Navigable.module(Module)` and `Navigable.module(Module)`. [#311]
- Support for using `interact {}` blocks in modules. [#364]
- Support page level `atCheckWaiting` configuration. [#287]
- `Navigator` elements can now also be filtered using `hasNot()` method. [#370]
- Custom implementation of `equals()` and `hashCode()` methods have been added to classes implementing `Navigator`. [#374]
- Support setting `forcelocal` flag for BrowserStack tunnel. [#385]
- Add configuration option for including cause string representation in message of `WaitTimeoutException`. [#386]

Improvements

- Using unrecognized content template parameters result in an `InvalidPageContent` exception to make catching typos in them easier. [#377]
- Improve error reporting when no at checkers pass if using multiple candidates for page switching. [#346]

- Don't unnecessarily lookup root element for every baseless element lookup using `$()` in context of `Navigable`. [#306]
- Attribute based searches are compiled to CSS selectors where possible. [#280]
- Attribute based searches using an id, class or name are performed using an appropriate `By` selector where possible. [#333]

Fixes

- Improved message thrown from Navigator.isDisabled() and Navigator.isReadOnly() when navigator does not contain a form element. [#345]
- Browser.verifyAtIfPresent() should fail for at checkers returning false when implicit assertions are disabled. [#357]
- Provide better error reporting when unexpected pages configuration is not a collection that contains classes which extend `Page`. [#270]
- Don't fail when creating a report and driver's screenshot taking method returns null. [#292]
- Classes that can define content should not throw custom exceptions from `propertyMissing()`. [#367]
- "At checkers" of pages passed to `withFrame()` methods are now verified. [#358]

Breaking changes

- `Page.toString()` now returns full page class name instead of its simple name.
- `MissingPropertyException` is thrown instead of `UnresolvablePropertyException` when content with a given name is not found on page or module.
- Geb is now built with Groovy 2.3.10 and Spock 1.0-groovy-2.3.

Deprecations

- `module(Class<? extends Module>, Navigator base)` available in content DSL has been deprecated in favour of `Navigator.module(Class<? extends Module>)` and will be removed in a future version of Geb.
- `module(Class<? extends Module>, Map args)` available in content DSL has been deprecated in favour of `Navigable.module(Module)` and will be removed in a future version of Geb.
- `module(Class<? extends Module>, Navigator base, Map args)` available in content DSL has been deprecated in favour of `Navigator.module(Module)` and will be removed in a future version of Geb.
- all variants of `moduleList()` method available in content DSL have been deprecated in favour of using `Navigator.module()` methods together with a `collect()` call and will be removed in a future version of Geb, see chapter on using modules for repeating content for examples [#362]
- `isDisabled()`, `isEnabled()`, `isReadOnly()` and `isEditable()` methods of `Navigator` have been deprecated and will be removed in a future version of Geb. These methods are now available on the new `FormElement` module class.

Project related changes

- User mailing list has moved to Google Groups.
- The Book of Geb has been migrated to Asciidoctor and the examples have been made executable. [#350]

0.10.0

New features

- New `css()` method on `Navigator` that allows to access CSS properties of elements. [#141]
- Added attribute based methods to relative content navigators such as next(), children() etc. [#299]
- Added signature that accepts `localIdentifier` to `BrowserStackDriverFactory.create`. [#332]

- Added `toWait` content definition option which allows specifying that page transition happens asynchronously. [#134]
- Added support for explicitly specifying browser capabilities when using cloud browsers Gradle plugins. [#340]
- Added an overloaded `create()` method on cloud driver factories that allow specifying browser capabilities in a map and don't require a string capabilities specification. [#281]

Fixes

- Allow access to module properties from its content block. [#245]
- Support setting of elements for WebDriver implementations that return uppercase tag name. [#318]
- Use native binaries for running BrowserStack tunnel. [#326]
- Update BrowserStack support to use command-line arguments introduced in tunnel version 3.1. [#332]
- Fix PermGen memory leak when using groovy script backed configuration. [#335]
- Don't fail in `Browser.isAt()` if at check waiting is enabled and it times out. [#337]
- The value passed to `aliases` content option in documentation examples should be a String [#338]
- Added `$()` method on Navigator with all signatures of `find()`. [#321]
- `geb-saucelabs` plugin now uses a native version of SauceConnect. [#341]
- Don't modify the predicate map passed to "Navigator.find(Map<String. [#339]
- Functional tests implemented using JUnit and Geb run twice in Grails 2.3+. [#314]
- Mention in the manual where snapshot artifacts can be downloaded from. [#305]
- Document that `withNewWindow()` and `withWindow()` switch page back to the original one. [#279]
- Fix selectors in documentation for manipulating checkboxes. [#268]

Project related changes

- Updated cucumber integration example to use `cucumber-jvm` instead of the now defunct `cuke4duke`. [#324]
- Setup CI for all of the example projects. [#188]
- Incorporate the example projects into the main build. [#189]
- Add a test crawling the site in search for broken links. [#327]
- Document the release process. [#325]

Breaking changes

- Use Groovy 2.3.6 to build Geb. [#330]
- Format of browser specification passed to `BrowserStackBrowserFactory.create()` and `SauceLabsBrowserFactory.create()` has changed to be a string in Java properties file format defining the required browser capabilities.
- `sauceConnect` configuration used with `geb-saucelabs` plugin should now point at a version of 'ci-sauce' artifact from 'com.saucelabs' group.

0.9.3

New features

- Added `baseNavigatorWaiting` setting to prevent intermittent Firefox driver errors when creating base navigator. [#269]
- Page content classes including `Module` now implement `Navigator` interface [#181]

- Added some tests that guard performance by verifying which WebDriver commands are executed [#302]
- Added BrowserStack integration [#307]
- Added a shortcut to `Browser` for getting current url [#294]
- Verify pages at checker when passed as an option to open a new window via `withWindow()` and `withNewWindow()` [#278]

Fixes

- Ignore `atCheckWaiting` setting when checking for unexpected pages. [#267]
- Added missing range variants of find/$ methods. [#283]
- Migrated `UnableToLoadException` to java. [#263]
- Exception thrown when trying to set value on an invalid element (non form control). [#286]
- Support for jQuery methods like offset() and position() which return a native Javascript object. [#271]
- Finding elements when passing ids with spaces in the predicates map to the $() method. [#308]

Breaking changes

- Removed easyb support. [#277]
- `MissingMethodException` is now thrown when using shortcut for obtaining a navigator based on a control name and the returned navigator is empty. [#239]
- When using SauceLabs integration, the `allSauceTests` task was renamed to`allSauceLabsTests`
- When using SauceLabs integration, the `geb.sauce.browser` system property was renamed to `geb.saucelabs.browser`
- `Module` now implements `Navigator` instead of `Navigable` so `Navigator's methods can be called on it without having to first call `$()` to get the module's base `Navigator`.

Project related changes

- Documentation site has been migrated to Ratpack. [#261]
- Cross browser tests are now also executed using Safari driver [#276]
- Artifact snapshots are uploaded and gebish.org is updated after every successful build in CI [#295]
- Migrated continuous integration build to Snap CI
- Added a Travis CI build that runs tests on submitted pull requests [#309]

0.9.2

New features

- `page` and `close` options can be passed to `withWindow()` calls, see this manual section for more information.
- Unexpected pages can be specified to fail fast when performing "at" checks. This feature was contributed at a Hackergarten thanks to Andy Duncan. See this manual section for details. [#70]
- Support for running Geb using SauceLabs provided browsers, see this manual section for details.
- New `isEnabled()` and `isEditable()` methods on `Navigator`.
- Support for ephemeral port allocation with Grails integration
- Compatibility with Grails 2.3

Fixes

- Default value of `close` option for `withNewWindow()` is set to `true` as specified in the documentation. [#258]

Breaking changes

- `isDisabled()` now throws `UnsupportedOperationException` if called on an `EmptyNavigator` or on a `Navigator` that contains anything else than a button, input, option, select or textarea.
- `isReadOnly()` now throws `UnsupportedOperationException` if called on an `EmptyNavigator` or on a `Navigator` that contains anything else than an input or a textarea.

0.9.1

Breaking changes

- Explicitly calling `at()` with a page object will throw `UndefinedAtCheckerException` instead of silently passing if the page object does not define an at checker.
- Passing a page with no at checker to `click(List<Class<? extends Page>>)` or as one of the pages in `to` template option will throw `UndefinedAtCheckerException`.

New features

- Support for dealing with self-signed certificates in Download API using `SelfSignedCertificateHelper`. [#150]
- Connections created when using Download API can be configured globally using `defaultDownloadConfig` configuration option.
- New `atCheckWaiting` configuration option allowing to implictly wrap “at” checkers in `waitFor` calls. [#253]

Fixes

- `containsWord()` and `iContainsWord()` now return expected results when matching against text that contains spaces [#254]
- `has(Map<String, Object> predicates, String selector)` and `has(Map<String, Object> predicates)` were added to Navigator for consistency with `find()` and `filter()` [#256]
- Also catch WaitTimeoutException when page verification has failed following a `click()` call [#255]
- `not(Map<String, Object> predicates, String selector)` and `not(Map<String, Object> predicates)` were added to Navigator for consistency with `find()` and `filter()` [#257]
- Make sure that `NullPointerException` is not thrown for incorrect driver implementations of getting current url without previously driving the browser to a url [#291]

0.9.0

New features

- New `via()` method that behaves the same way as `to()` behaved previously - it sets the page on the browser and does not verify the at checker of that page[#249].
- It is now possible to provide your own `Navigator` implementations by specifying a custom `NavigatorFactory`, see this manual section for more information [#96].
- New variants of `withFrame()` method that allow to switch into frame context and change the page in one go and also automatically change it back to the original page after the call, see [switching pages and frames at once][switch-frame-and-page] in the manual [#213].
- `wait`, `page` and `close` options can be passed to `withNewWindow()` calls, see this manual section for more information [#167].
- Improved message of UnresolvablePropertyException to include a hint about forgetting to import the class [#240].

- Improved signature of `Browser.at()` and `Browser.to()` to return the exact type of the page that was asserted to be at / was navigated to.
- `ReportingListener` objects can be registered to observe reporting (see: this manual section)

Fixes

- Fixed an issue where waitFor would throw a WaitTimeoutException even if the last evaluation before timeout returned a truthy value [#215].
- Fixed taking screenshots for reporting when the browser is not on a HTML page (e.g. XML file) [#126].
- Return the last evaluation value for a `(wait: true, required: false)` content instead of always returning null [#216].
- `isAt()` behaves the same as `at()` in regards to updating the browser's page instance to the given page type if its at checker is successful [#227].
- Handling of `select` elements has been reworked to be far more efficient [#229].
- Modules support accessing base attributes' values using @attributeName notation [#237].
- Use of text matchers in module base definitions is supported [#241].
- Reading of textareas have been updated so that the current value of the text field is returned, instead of the initial [#174].

Breaking changes

- `to(Class<? extends Page>)` method now changes the page on the browser and verifies the at checker of that page in one method call [#1], [#249]; use `via()` if you need the old behaviour
- `getAttribute(String)` on `Navigator` now returns `null` for boolean attributes that are not present.
- `at()` and `to()` methods on `Browser` now return a page instance if they succeed and `via()` method always returns a page instance [#217].
- `withFrame()` calls that don't take a page argument now change the browser page to what it was before the call, after the call [#222].
- due to performance improvements duplicate elements are not removed when creating new `Navigator`s anymore; the new `unique()` method on `Navigator` can be used to remove duplicates if needed [#223].
- `at(Page)` and `isAt(Page)` methods on `Browser` have been removed as they were inconsistent with the rest of the API [#242].
- Navigator's `click(Class<? extends Page>)` and `to:` content option now verify page after switching to the new one to stay consistent with the new behaviour of `to(Class<? extends Page>)` [#250].
- Reading an attribute that is not set on a navigator now returns an empty string across all drivers [#251].

0.7.2

Fixes

- Further fixes for Java 7 [#211].

0.7.1

New features

- Geb is now built with Groovy 1.8.6. This was forced to resolve [#194].

Fixes

- `startsWith()`, `contains()` etc. now work for selecting via element text now works for multiline (i.e. `<br/>`) text [#202]
- Geb now works with Java 7 [#194].

0.7.0

New features

- Added support for indexes and ranges in `moduleList()` method
- Form control shortcuts now also work on page and module content
- Custom timeout message for `waitFor()` calls
- Navigators can be composed also from content
- Closure expressions passed to `waitFor()` calls are now transformed so that every statement in them is asserted - this provides better reporting on `waitFor()` timeouts.
- `at` closure properties of Page classes are now transformed so that every statement in them is asserted - this provides better reporting on failed at checks
- new `isAt()` method on Browser that behaves like `at()` used to behave before, i.e. does not throw AssertionError but returns `false` if at checking fails
- `withAlert()` and `withConfirm()` now accept a `wait` option and the possible values are the same as for waiting content

Breaking changes

- `click()` now instructs the browser to click **only on the first** element the navigator has matched
- All `click()` method variants return the reciever
- Content definitions with `required: false, wait: true` return `null` and do not throw `WaitTimeoutException` if the timeout expires
- Assignment statements are not allowed anymore in closure expressions passed to `waitFor()` calls
- `at()` now throws AssertionException if at checking fails instead of returning false

0.6.3

New features

- Compatibility with Spock 0.6

0.6.2

New features

- New `interact()` function for mouse and keyboard actions which delegates to the WebDriver Actions class
- New `moduleList()` function for repeating content
- New `withFrame()` method for working with frames
- New `withWindow()` and `withNewWindow()` methods for working with multiple windows
- Added `getCurrentWindow()` and `getAvailableWindows()` methods to browser that delegate to the underlying driver instance
- Content aliasing is now possible using `aliases` parameter in content DSL
- If config script is not found a config class will be used if there is any - this is usefull if you run test using Geb from IDE
- Drivers are now cached across the whole JVM, which avoids the browser startup cost in some situations
- Added config option to disable quitting of cached browsers on JVM shutdown

Breaking changes

- The `Page.convertToPath()` function is now responsible for adding a prefix slash if required (i.e. it's not added implicitly in `Page.getPageUrl()`) [GEB-139].
- Unchecked checkboxes now report their value as `false` instead of null

0.6.1

New features

- Compatibility with at least Selenium 2.9.0 (version 0.6.0 of Geb did not work with Selenium 2.5.0 and up)
- Attempting to set a select to a value that it does not contain now throws an exception
- The waiting algorithm is now time based instead of number of retries based, which is better for blocks that are not near instant
- Better support for working with already instantiated pages

Breaking changes

- Using `<select>` elements with Geb now requires an explicit dependency on an extra WebDriver jar (see the section on installation for more info)
- The `Navigator` `classes()` method now returns a `List` (instead of `Set`) and guarantees that it will be sorted alphabetically

0.6

New features

- selenium-common is now a 'provided' scoped dependency of Geb
- Radio buttons can be selected with their label text as well as their value attribute.
- Select options can be selected with their text as well as their value attribute.
- `Navigator.getAttribute` returns `null` rather than the empty string when an attribute is not found.
- The `jquery` property on `Navigator` now returns whatever the jQuery method called on it returns.
- All waitFor clauses now treat exceptions raised in the condition as an evaluation failure, instead of propagating the exception
- Content can be defined with `wait: true` to make Geb implicitly wait for it when it is requested
- Screenshots are now taken when reporting for all drivers that implement the `TakesScreenshot` interface (which is nearly all)
- Added `BindingUpdater` class that can manage a groovy script binding for use with Geb
- Added `quit()` and `close()` methods to browser that delegate to the underlying driver instance
- `geb.Browser.drive()` methods now return the used `Browser` instance
- The underlying WebElements of a Navigator are now retrievable
- Added $() methods that take one or more Navigator or WebElement objects and returns a new Navigator composed of these objects
- Added Direct Download API which can be used for directly downloading content (PDFs, CSVs etc.) into your Geb program (not via the browser)
- Introduced new configuration mechanism for more flexible and environment sensitive configuration of Geb (e.g. driver implementation, base url)
- Default wait timeout and retry interval is now configurable, and can now also use user configuration presets (e.g. quick, slow)
- Added a "build adapter" mechanism, making it easier for build systems to take control of relevant configuration
- The JUnit 3 integration now includes the test method name in the automatically generated reports

- The reporting support has been rewritten, making it much friendlier to use outside of testing
- Added the TestNG support (contributed by Alexander Zolotov)
- Added the `height`, `width`, `x` and `y` properties to navigator objects and modules

Breaking changes

- Raised minimum Groovy version to 1.7
- All failed waitFor clauses now throw a `geb.waiting.WaitTimeoutException` instead of `AssertionError`
- Upgraded minimum version requirement of WebDriver to 2.0rc1
- The `onLoad()` and `onUnload()` page methods both have changed their return types from `def` to `void`
- The Grails specific testing subclasses have been REMOVED. Use the direct equivalent instead (e.g `geb.spock.GebReportingSpec` instead of `grails.plugin.geb.GebSpec`)
- The Grails plugin no longer depends on the test integration modules, you need to depend on the one you want manually
- The `getBaseUrl()` method from testing subclasses has been removed, use the configuration mechanism
- Inputs with no value now report their value as an empty string instead of `null`
- Select elements that are not multiple select enabled no longer report their value as a 1 element list, but now as the value of the selected element (if no selection, `null` is returned)

0.5.1

- Fixed problem with incorrectly compiled specs and the geb grails module

0.5

New features

- Navigator objects now implement the Groovy truth (empty == false, non empty == true)
- Introduced “js” short notation
- Added “easyb” support (`geb-easyb`) and Grails support
- Page change listening support through `geb.PageChangeListener`
- `waitFor()` methods added, making dealing with dynamic pages easier
- Support for `alert()` and `confirm()` dialogs
- Added jQuery integration
- Reporting integration classes (e.g. GebReportingSpec) now save a screenshot if using the FirefoxDriver
- Added `displayed` property to navigator objects for determining visibility
- Added `find` as an alias for `$` (e.g. `find("div.section")`)
- Browser objects now implement the `page(List<Class>)` method that sets the page to the first type whose at-checker matches the page
- The click() methods that take one or more page classes are now available on `Navigator` objects
- Added page lifecycle methods `onLoad()`/`onUnload()`

Breaking changes

- Exceptions raised in `drive()` blocks are no longer wrapped with `DriveException`

- the `at(Class pageClass)` method no longer requires the existing page instance to be of that class (page will be updated if the given type matches)

0.4

Initial Public Release

Version 1.1.1
Last updated 2015-06-24 15:54:54 BST