How to use Selenium, successfully

The Selenium Guidebook
Python Edition

by Dave Haeffner

Preface

This book is not a full and comprehensive treatise that outlines every possible permutation of Selenium (the open-source software test automation tool for web applications). There are other books that already do this. My goal, instead, is to teach you the necessary pieces to use Selenium successfully for your circumstance.

What you have before you is a distilled and actionable guide culled from my consulting practice and full time positions held doing Quality Assurance over the past seven years.

My goal in writing this is to provide you with the materials I wish existed when I was starting out with automated acceptance testing. I hope it serves you well.

What This Book Will Cover

This book focuses on the latest stable version of Selenium 2 (a.k.a. Selenium WebDriver) and its use to test desktop browsers. Its predecessor, Selenium RC, will not be covered. Mobile testing with Appium will also not be covered. But you can see a series of getting started with Appium posts I've written here.

Record and Playback tools like Selenium IDE and Selenium Builder are great for getting started, but abysmal for growing past that point. So they will not be covered. Instead, an approach on writing well factored tests, in code, will be the primary focus of this book.

Who This Book Is For

This book is for anyone who wants to take automated acceptance testing seriously and isn't afraid to get their hands a little dirty.

That is to say, this book is for anyone who wants to use computers for what they're good at, and free you up (and potentially the people on your team) to do what they are inherently good at (which does not include repetitive, mundane testing tasks). And don't worry if you're new to programming. I'll cover the essentials so you'll have a good place to start from.

About The Examples In This Book

The examples in this book are written in Python, but the strategies and patterns used are applicable regardless of your technical stack.

The tests in this book are written to exercise functionality from an open-source project I created and maintain called the-internet -- available here on GitHub and viewable here on Heroku.

The test examples are written to run against pytest with pip managing the third-party

dependencies.

All of the code examples from the book are available in an accompanying zip file. It contains folders for each chapter where code was written or altered. Chapters with multiple parts will have multiple sub-folders (e.g., code examples referenced in Part 2 of Chapter 9 can be found in `09/02/` in the zip file).

How To Read This Book

Chapters 1 through 5 focus on the things you need to consider when it comes to test strategy, programming language selection, and good test design. Chapter 6 is where we first start to code. From there, the examples build upon each other through chapter 16.

Chapter 17 paints a picture of the Selenium landscape so you're better able to find information on your own.

Feedback

If you find an error in the book (e.g., grammar issue, code issue, etc.) or have questions/feedback -- please feel free to e-mail me at dhaeffner@gmail.com.

If you submit something and I end up using it in a future version of the book I'll give you a shout-out in the Acknowledgements.

Table of Contents

Chapter 1
Selenium In A Nutshell

Selenium is a software robot sent from the future to help us test web applications. But keep in mind that it's not one of those fancy shape-shifting robots than can run really fast. It's more like one of those really strong robots that's not very fast and is best suited for accomplishing a certain objective.

That is to say -- Selenium is really good at a specific set of things. If you know what those are and stick to them then you can easily write reliable, scalable, and maintainable tests that you and your team can trust.

What Selenium Is and Is Not Good At

Selenium is built to automate browsers, specifically human interaction with them. Things like navigating to pages, clicking on elements, typing text into input fields, etc.

It's less ideal for checking lower-level functionality like HTTP status codes or HTTP headers. While you can use Selenium this way, it requires additional setup of a third-party tool (e.g., a proxy server like BrowserMob Proxy), and it is a slippery slope since there are numerous edge cases to consider at this level.

Selenium Highlights

Selenium works on every major browser, with a number of major programming languages, and on every major operating system. Each language binding and browser are actively being developed to stay current. Yes, even Internet Explorer (thanks to [Jim Evans](#)!).

Selenium can be run on your local computer, on a server (with Selenium Remote), on your own set of servers (with Selenium Grid), or on a third-party cloud provider (like [Sauce Labs](#)). As your test suite grows, your test runs will take longer to complete. To speed them up you will want to run them in parallel, which is where the benefit of having your own servers or using a cloud provider comes in -- that, and the ability to have numerous browser and operating system combinations to run your tests on.

One of the guiding philosophies of Selenium is to be able to write your tests once and run them across multiple browsers. While this is a rosy proposition, it's not entirely accurate. There are plenty of gotchas to watch out for when you get into it. But don't worry, We'll step through these in detail throughout the book.

Chapter 2
Defining A Test Strategy

A great way to increase your chances of success with automated web testing is to first map out a testing strategy. The best way to do it is to answer these four questions:

1. How does your business make money?
2. What features in your application are being used?
3. What browsers are your users using?
4. What things have broken in the application before?

NOTE: For organizations that don't deal directly in dollars and cents (e.g., non-profits, federal agencies, etc.) you should focus on how the application generates value for the end user and the organization.

After answering these questions you will have an understanding of the functionality and browsers that matter for the application you're testing. This will help you focus your efforts on the things that matter most.

This strategy works best for applications with existing functionality and does not speak directly to testing new functionality that's being actively developed. That's not to say that the two couldn't co-exist. It largely depends on your available resources and pace of development. But in order to reach high quality at speed, you first have to go slow.

What To Do With The Answers

After answering these you should end up with a prioritized punch list (a.k.a., backlog) of test automation tasks that you can work off of and track progress against.

Question 1: Money/Value

Every company's application makes money (or generates value) through core functionality that is defined by a series of increasingly-valuable user interactions -- a.k.a. a "funnel". Your answers to this question will help you determine what your funnel is.

These items will be your highest priority for automation. Start an automation backlog to keep track of them.

Question 2: Usage Data

Odds are your application offers a robust set of functionality well beyond your funnel. Your answers to this question will help highlight what it is. And if you're basing these answers on usage data (e.g., Google Analytics), then it will be broken down from highly used to lightly used.

Tack these items onto your automation backlog (below the items from question #1) based on their frequency of use.

Question 3: Browsers

Now that you know what functionality is business critical and widely adopted by your users, you need to determine what browsers to focus your automated web testing efforts on. Your usage data will tell you this as well. It will help you determine which browsers you can reasonably avoid testing in (e.g., based on non-existent or low usage numbers).

Note the top 2 (or 3 depending on your numbers), but focus on the top 1 for now. This is the browser you will start using for automation.

Question 4: Risky Bits

To round out the strategy it is also best to think about what things have broken in the application before. To answer this question it's best to check your defect/bug tracker (if you have one) and to ask your team. What you end up with may read like a laundry list of browser specific issues or functionality that has been flaky or forgotten about in the past. But it's all useful information.

Be sure to check this list against your automation backlog. If somethings not there, add it to the bottom of the backlog. If it is there, make a note in the backlog item that it has been an issue in the past.

If the issue has happened numerous times and has the potential to occur again, move the item up in the backlog. And if issues keep cropping up that are related to a specific browser, compare this browser to your short list of browsers from question #3. If it's a browser that's not in your list but it's still important (e.g., a small pocket of influential users), track it on the backlog, but put it at the bottom.

Now You Are Ready

Having answered these questions, you should now have a prioritized backlog of critical business functionality, a short list of browsers to focus on, and an understanding of the risky parts of your application to watch out for. With it, you're on the right track -- focusing on things that matter for your business and its users.

Chapter 3
Picking A Language

In order to work well with Selenium you need to choose a programming language to write your automated acceptance tests in. Conventional wisdom will tell you to choose the same language that the application is written in. That way if you get stuck you can ask the developers on your team for help. But if you're not proficient in this language (or new to programming), then your progress will be slow and you'll likely end up asking for more developer help than they have time for -- hindering your automation efforts and setting you up for failure.

A great way to determine which language to go with is to answer one simple question: Who will own the automated tests?

The answer to this, and the discussion that unfolds from it, will help you more effectively choose a programming language.

What To Do With The Answer

If you're a tester (or a team of testers) and you will be building and maintaining the test automation, then it's best to consider what languages you (and your team) already have experience with, or are interested in working with. Since your Selenium tests don't need to be written in the same language as the application you are testing, have a discussion with your team to see what interests them and go from there.

If you're a developer who is working on the application and just looking to add automated acceptance testing to the mix, then it makes sense to continue with the same language.

Some Additional Food For Thought

As your suite of tests grows you will find the need to add functionality to make things easier to use, run faster, offer better reporting, etc. When this happens you will effectively be doing two jobs; no longer just writing automated tests but also building and maintaining a framework (a.k.a. a test harness).

As you're considering which language to go with consider what open source frameworks already exist for the languages you are considering. Going with one can save you a lot of time and give you a host of functionality out of the box that you would otherwise have to create yourself -- and they're FREE.

You can find a list of open source Selenium WebDriver frameworks and their respective languages [here](#).

Outro

Choosing a programming language for automated testing is not a decision that should be taken lightly. If you're just starting out (or looking to port your tests) then considering and discussing these things will help position you for long term success.

With all that being said, the rest of this book will show you how to build your own test harness (in addition to writing well factored tests) in Python with pytest.

Chapter 4
A Programming Primer

This section will prime you with just enough programming concepts (and how they pertain to Selenium) so you have some working knowledge and a vocabulary. This will help you more effectively comprehend what you will see throughout this book (and in your work afterwards).

Don't get too hung up on the details though. If something doesn't make sense it should once we dig into the full examples in the following chapters. Consider this more of a glossary than a code school.

Installation

There's a great guide on "Properly Installing Python" which you can find [here](). It covers Mac OSX, Windows, and Ubuntu.

NOTE: For doing proper software development in Python you'd want to consider something like [Virtual Environments]() to effectively manage third-party dependencies. But for the needs of the examples in this book, it's not necessary.

Installing Third-Party Libraries

There are over 84,000 third-party libraries (a.k.a. "packages") available for Python through PyPI (the Python Package Index). To install packages from it you use a program called `pip`.

To install them you use `pip install package-name` from the command-line.

Here is a list of the packages that will be used throughout the book.

- `pytest`
- `pytest-randomly`
- `pytest-xdist`
- `selenium`

Interactive Prompt

One of the immediate advantages to using a scripting language like Python is that you get access to an interactive prompt. Just type `python` from the command-line. It will load a prompt that looks like this:

```
Python 2.7.10 (default, Jul 14 2015, 19:46:27)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

In this prompt you can type out Python code. It will evaluate it and return the result. As you step through this chapter it will serve as a great place to practice the commands and concepts you will see.

When you're done, just type `quit()`.

Choosing A Text Editor

In order to write Python code, you will need to use a text editor. Some popular ones are [Vim](#), [Emacs](#), [Sublime Text](#).

There's also the option of going for an IDE (Integrated Development Environment) like [PyCharm](#).

It's important to pick an editor that works for you and makes you productive. So if you're new to programming and text editors then it's probably best to go with something more intuitive like Sublime Text or PyCharm.

Programming Concepts In A Nutshell

Programming can be a deep and intimidating rabbit hole if you're new to it. But don't worry. When it comes to automated browser testing there is only a small subset of programming that we really need to know in order to get started. Granted, the more you know, the better off you'll be. But you don't need to know a whole lot in order to be effective right now.

Of all the programming concepts out there, here are the ones you'll need to pay attention right now:

- Object Structures (Variables, Methods, and Classes)
- Scope
- Types of Objects (Strings, Integers, Data Structures, Booleans, etc.)
- Actions (Assertions and Conditionals)
- Decorators
- Inheritance

Let's step through each and how they pertain to testing with Selenium.

Object Structures

Variables

Variables are places to store and retrieve values of various types (e.g., Strings, Integers, etc. -- more on these later). Variables are created and then referenced by their name.

A variable name:

- can be one or more words in length
- use an underbar (_) to separate the words (e.g., `example_variable`)
- start with a lowercase letter
- are often entirely lowercase
- multiple word variables can be combined into a single word for better readability

You can store things in them by using an equals sign (`=`) after their name. In Python, a variable takes on the type of the value you store in it (more on object types later).

```
>>> example_variable = "42"
>>> print(type(example_variable))
# outputs: <type 'str'>
# 'str' is short for "string"

>>> example_variable = 42
>>> print(type(example_variable))
# outputs: <type 'int'>
# 'int' is short for "integer"
```

In the above example `print` is used to output a message. This is a common command that is useful for generating output to the terminal.

In Selenium, a common example of a variable is when we need to store an element (or the value from an element) in the beginning of a test to be referenced later. For example, getting a page's title.

```
page_title = driver.title
```

`driver` is the variable we will use to interact with Selenium.

Methods

One way to group common actions (a.k.a. behavior) for easy reuse is to place them into methods. We define a method with the opening keyword `def` , a name (in the same fashion as a variable), and end the method with white-space. Referencing a method is done the same way as a variable -- by its name. And the lines of code within the method need to be indented.

```
def example_method():
  # your code
  # goes here


example_method()
```

Additionally, we can specify arguments we want to pass into the method when calling it.

```
>>> def say(message):
...     print(message)
...
>>> say("Hello World!")
# outputs: Hello World!
```

When setting an argument, we can also set a default value to use if no argument is provided.

```
>>> def say(message="Hello World!")
...     print(message)
...
>>> say()
# outputs: # Hello World!
>>> say("something else")
# outputs: something else
```

We'll see something like this used in Selenium when we are telling Selenium how to wait with explicit waits (more on that in Chapter 10).

Classes

Classes are a useful way to represent concepts that will be reused numerous times in multiple places. They can contain variables and methods and are defined with the word `class` followed by the name you wish to give it.

Class names:

- start with a capital letter
- should be PascalCase for multiple words (e.g., `class ExampleClass`)
- should be descriptive (e.g., a noun, whereas methods should be a verb)
- end with whitespace (just like methods)

You first have to define a class, and then create an instance of it (a.k.a. instantiation) in order to use it. Once you have an instance you can access the methods within it to trigger an action. Methods in classes need to use the keyword `self` as an argument in order to be properly referenced.

```
>>> class Message():
...     def say(self, message = "Hello World!"):
...         print(message)
...
>>> message_instance = Message()
>>> message_instance.say("This is an instance of a class")
# outputs: This is an instance of a class
```

An example of this in Selenium is the representation of a web page -- also known as a 'Page Object'. In it you will store the page's elements and behavior we want to interact with.

```
class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}

    def with_(self, username, password):
# ...
```

The variables that start with underscores (e.g., `_`) are considered internal (or "local") variables, the values in curly brackets (`{}`) are called dictionaries. More on all of that soon.

Scope

Now that we have our different object structures it's worth briefly covering how they can and cannot access one another. The easiest way to do this is by discussing the different types of variables you are likely to run into.

Local Variables

Local variables enable you to store and retrieve values from within a confined area (this is no different than what was described in the variables section earlier). A variable that was declared within a method is a great example of this. It is useful within the method it was declared, but inaccessible outside of it.

Instance Variables

Instance variables enable you to store and retrieve values more broadly (e.g., both inside and outside of methods). They are named the same way as regular variables, except that they start with `self.` . This is only applicable for variables within a class.

A common example you will see throughout this book is the usage of `self.driver` . This is an instance of Selenium stored in an instance variable. This object is what enables us to control the browser and by storing it as an instance variable we'll be able to use it where necessary.

NOTE: There are also class variables, which are similar to instance variables in terms of their scope. They do not require `.self` as part of their declaration. We'll see these when we get into Page Objects in Chapter 8.

Private Objects

Python is a dynamic language with few constraints (which is very much the opposite of defensive compiled languages like Java). The idea of limited object access (either implied by an object's scope or explicitly by a `private` keyword like in other programming languages) isn't an enforcible concept in Python. But there is a saying in Python -- "We are all responsible users".

It means that rather than relying on hard constraints we should rely on [a set of conventions](#). These conventions are meant to indicate which elements should and should not be accessed directly.

The convention for this is to start variable and method names which are meant to be private (e.g., only meant to be used internally where the object was declared) with an underscore. (e.g., `_example_variable` or `_example_method():`).

Environment Variables

Environment variables are a way to pass information into our program from outside of it. They are also a way to make a value globally accessible (e.g., across an entire program, or set of programs). They can be set and retrieved from within your code by:

- importing the built-in `os` Python library (e.g., `import os`)
- using the `os.environ` lookup syntax
- specify the environment variable name with it

Environment variables are often used to store configuration values that could change. A great example of this is the access key for a third-party service provider that we'll use in our tests.

```python
import os
_credentials = '%s:%s' % (os.environ["SAUCE_USERNAME"],
                          os.environ["SAUCE_ACCESS_KEY"])
```

Types of Objects

Strings

Strings are alpha-numeric characters packed together (e.g., letters, numbers, and most special characters) surrounded by either single (') or double (") quotes.

You'll run into Strings when working with copy on a page (e.g., pulling a page's URL, title, or h1 tag to determine if your test is in the right place before proceeding).

Numbers

The two common types of numbers you will run into with testing are Integers (whole numbers) and Float (decimals). Or `'int'` and `'float'` with regards to how the Python language refers to them.

If you have a test that needs to pull some values from a page and add/subtract/multiply/divide them, then this will come in handy. Although you may need to convert the values from a String to an Integer first. But don't sweat it, this is a trivial thing to do in Python.

```python
int("42")
```

The conversion from a `'str'` to an `'int'` is done with `int()` method. If you're working with decimals, you can use the `decimal` library built into Python to convert it.

```python
>>> import decimal
>>> decimal.Decimal("42.00")
```

Data Structures

Data Structures enable you to gather a set of data for later use. In Python there are numerous data structures. The one we'll want to pay attention to is Dictionaries.

Dictionaries are an unordered set of data stored in key/value pairs. The keys are unique and are used to look up the data in the dictionary.

```python
>>> a_dictionary = {"this": "that", "the": "other"}
>>> print(a_dictionary["this"])
# outputs: that
>>> print(a_dictionary["the"])
# outputs: other
```

You'll end up working with Dictionaries in your Page Objects to store and retrieve your page's locators.

```python
class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}
    # ...
```

Booleans

Booleans are binary values that are returned when asking a question of your code. They are what enable us to complete assertions.

There are numerous ways to ask questions. Some involve various [comparison operators](#) (e.g., `==`, `!=`, `<`, `>`, `<=`, `>=`). The response is either `True` or `False`.

```
>>> 2 + 2 == 4
True
```

Selenium also has commands that return a boolean result when we ask questions of the page we're testing.

```
element.is_displayed()
# outputs: returns `True` if the element is on the page and visible
```

Actions

A benefit of booleans is that we can use them to perform an assertion.

Assertions

Assertions are made against booleans and result in either a passing or failing test. In order to leverage assertions we will need to use a testing framework (e.g., [pytest](#), [unittest](#), or [nose](#)). For the examples in this book we will be using pytest (version 2.9.2).

pytest has a built-in assertion method which accepts an argument for something that returns a boolean.

```
driver.get("http://the-internet.herokuapp.com")
assert(driver.title == "The Internet")

# or

driver.get("http://the-internet.herokuapp.com")
title_present = driver.title == "The Internet"
assert(title_present)
```

Both approaches will work, resulting in a passing assertion. If this is the only assertion in your test then this will result in a passing test. More on this and other good test writing practices in Chapter 5.

Conditionals

Conditionals work with booleans as well. They enable you execute different code paths based on their values.

The most common conditionals in Python are `if`, `elif` (short for else/if), and `else` statements.

```python
number = 10
if number > 10:
  print("The number is greater than 10")
elif number < 10:
  print("The number is less than 10")
elif number == 10:
  print("The number is 10")
else:
  print("I don't know what the number is.")
end

# outputs: The number is 10
```

You'll end up using conditionals in your test setup code to determine which browser to load based on a configuration value. Or whether or not to run your tests locally or somewhere else.

```python
if config.host == "localhost":
        if config.browser == "firefox":
            driver_ = webdriver.Firefox()
        elif config.browser == "chrome":
            _chromedriver = os.path.join(os.getcwd() + 'vendor', 'chromedriver')
            driver_ = webdriver.Chrome(_chromedriver)
```

More on that in chapters 12 and 13.

Decorators

Decorators are a form of metadata. They are used by various libraries to enable additional functionality in your tests.

The most common way we're going to use this is for our test setup and teardown. In pytest we do this with a fixture (which is a function that gets called before our test with the option to execute it after the test as well).

```python
@pytest.fixture
    def driver(self, request):
        driver_ = webdriver.Firefox()

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return driver_
```

The decorator with the `@` symbol, has a name to go along with it (e.g., `@pytest.fixture`), and is placed above the object that it effects (e.g., the `driver` method). We'll see decorators used for a few different things throughout the book.

Inheritance

Classes have the ability to connect to one-another through parent/child inheritance. By having a single parent class we can store common actions in methods that can be readily available to all child classes.

Inheritance is done when declaring a child class by:

- providing the class name
- importing the parent class (when in another file)
- specifying the parent class name as a declaration argument

```python
>>> class Parent():
...     hair_color = "brown"
...
>>> class Child(Parent):
...     pass
...
>>> c = Child()
>>> c.hair_color

# outputs: brown
```

You'll see this in your tests when writing all of the common Selenium actions you intend to use into methods in a parent class. Inheriting this class will allow you to call these methods in your child classes (more on this in Chapter 9).

Additional Resources

Here are some additional resources that can help you continue your Python learning journey.

- [A list of getting started resources from the Python project](#)
- Interactive online resources from [learnpython.org](#), [codecademy](#), and [Code School](#)
- [Learn Python the Hard Way](#)

Chapter 5
Anatomy Of A Good Acceptance Test

In order to write automated web tests that are easy to maintain, perform well, and are ultimately resilient there are some simple guidelines to follow:

- Write atomic and autonomous tests
- Group like tests together in small batches
- Be descriptive
- Use a Test Runner
- Store tests in a Version Control System

Atomic & Autonomous Tests

Each test needs to be concise (e.g., testing a single feature rather than multiple features) and be capable of being run independently (e.g., sets up its own data rather than relying on a previous test to do it). Doing this may require a mental shift, discipline, and more up front effort. But it will make a dramatic impact on the quality, effectiveness, and maintainability of your tests. Especially when you get into parallel test execution.

Grouping Tests

As your test suite grows you will have numerous test files. Each one containing a grouping of tests that exercise similar functionality. These test files should be grouped together in a simple directory structure as the groupings become obvious. If you want to create a test run of disparate tests, this is something that is easy to handle when using a Test Runner (covered briefly below, and in-depth in Chapter 15).

Being Descriptive

A test file should have a high level name that denotes what the tests within it are exercising. Each test should have an informative name (even if it is a bit verbose). Also, each test (or grouping of tests) should include some helpful metadata (e.g., Categories) which can provide additional information about the test as well as enable flexible test execution (more on that in Chapter 15). This way all or part of your test suite can be run, and the results will be informative thanks to helpful naming.

This enables developers to run a subset of tests to exercise functionality they just modified (as part of their pre-check-in testing) while also enabling you to intelligently wire your test suite up to a Continuous Integration (CI) server for fast and dynamic feedback (more on CI servers in Chapter 16).

Test Runners

At the heart of every test harness is some kind of a test runner that does a lot of the heavy lifting (e.g., test execution, centralized configuration, test output, etc.). Rather than reinvent the wheel you can use one of the many test runners that already exist today. With it you can bolt on third party libraries to extend its functionality if there's something missing.

Version Control

In order to effectively collaborate with other testers and developers on your team, your test code must live in a version control system of some sort. Look to see what your development team uses and add your code to it. Otherwise, set up one of the following:

- [Git](#)
- [Mercurial](#)
- [Subversion](#)

Keep in mind that your test code can live in a separate repository from the code of the application you're testing. Combining them may be advantageous but if all you're doing is writing and running tests against web endpoints (which is a majority of what your Selenium tests will be doing) then leaving your test code in a separate repository is a fine way to go.

Chapter 6
Writing Your First Test

Fundamentally, Selenium works with two pieces of information -- the element on a page you want to use and what you want to do with it. This one-two punch will be repeated over and over until you achieve the outcome you want in your application -- at which point you will perform an assertion to confirm that the result is what you intended.

Let's take logging in to a website as an example. With Selenium you would:

1. Visit the login page of a site
2. Find the login form's username field and input the username
3. Find the login form's password field and input the password
4. Find the submit button and click it

Selenium is able to find and interact with elements on a page by way of various locator strategies. The list includes (sorted alphabetically):

- Class
- CSS Selector
- ID
- Link Text
- Name
- Partial Link Text
- Tag Name
- XPath

While each serves a purpose, you only need to know a few to start writing effective tests.

How To Find Locators

The simplest way to find locators is to inspect the elements on a page. The best way to do this is from within your web browser. Fortunately, popular browsers come pre-loaded with development tools that make this simple to accomplish.

When viewing the page, right-click on the element you want to interact with and click Inspect Element. This will bring up a small window with all of the markup for the page but zoomed into your highlighted selection. From here you can see if there are unique or descriptive attributes you can work with.

How To Find Quality Elements

You want to find an element that is unique, descriptive, and unlikely to change.

Ripe candidates for this are `id` and `class` attributes. Whereas text (e.g., the text of a link) is less ideal since it is more apt to change. This may not hold true for when you make assertions, but it's a good goal to strive for.

If the elements you are attempting to work with don't have unique `id` or `class` attributes directly on them, look at the element that houses them (a.k.a. the parent element). Oftentimes the parent element has a unique element that you can use to start with and walk down to the child element you want to use.

When you can't find any unique elements have a conversation with your development team letting them know what you are trying to accomplish. It's typically a trivial thing for them to add helpful semantic markup to a page to make it more testable. This is especially true when they know the use case you're trying to automate. The alternative can be a lengthy and painful process which will probably yield working test code but it will be brittle and hard to maintain.

Once you've identified the target elements for your test, you need to craft a locator using one Selenium's strategies.

An Example

Part 1: Find The Elements And Write The Test

Here's the markup for a standard login form (pulled from [the login example on the-internet](#)).

```
<form name="login" id="login" action="/authenticate" method="post">
  <div class="row">
   <div class="large-6 small-12 columns">
      <label for="username">Username</label>
      <input type="text" name="username" id="username">
   </div>
  </div>
  <div class="row">
    <div class="large-6 small-12 columns">
      <label for="password">Password</label>
      <input type="password" name="password" id="password">
    </div>
  </div>
    <button class="radius" type="submit"><i class="icon-2x icon-signin"> Login
</i></button>
</form>
```

Note the unique elements on the form. The username input field has a unique `id`, as does the password input field. The submit button doesn't, but it's the only button on the page so we can easily find it and click it.

Let's put these elements to use in our first test. First we'll need to create a new folder called `tests` in the root of our project directory. In it we'll create a new test file called `login_test.py` and a requisite Python file for the directory called `__init__.py`. We'll also need to create a `vendor` directory for third-party files and download `geckodriver` into it (you can grab the latest release from here). This is a required file (known as a browser driver) in order to make Selenium work with Firefox 48 and newer. See Chapter 12 for more detail about browser drivers.

When we're done our directory structure should look like this.

```
tests
    __init__.py
    login_test.py
vendor
    geckodriver
```

And here is the code we will add to the test file for our Selenium commands, locators, etc.

```
# filename: tests/login_test.py
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By


class TestLogin():

    @pytest.fixture
    def driver(self, request):
        _geckodriver = os.path.join(os.getcwd(), 'vendor', 'geckodriver')
        driver_ = webdriver.Firefox(executable_path=_geckodriver)

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return driver_


    def test_valid_credentials(self, driver):
        driver.get("http://the-internet.herokuapp.com/login")
        driver.find_element(By.ID, "username").send_keys("tomsmith")
        driver.find_element(By.ID, "password").send_keys("SuperSecretPassword!")
        driver.find_element(By.CSS_SELECTOR, "button").click()
```

After importing the requisite classes for pytest and Selenium we declare a test class (e.g., `class TestLogin()`). We then declare a method within the class called `driver`. At the top of the method we add a decorator to denote that this is a fixture (e.g., `pytest.fixture`). By default fixture methods in pytest are called around each test method. So we'll use this to setup and teardown our instance of Selenium.

To create an instance of Selenium we call `webdriver.Firefox()` and store it in a variable. Since the name of the method is `driver`, we refer to this variable as `driver_`. This variable gets returned at the end of the method which will get used in our test (more on that soon). In order for this to work we need to provide the path to the `geckodriver` file, which we do by finding it and storing it in a local variable called `_geckodriver` at the top of the method and passing it in as part of our Selenium instantiation (e.g., `webdriver.Firefox(executable_path=_geckodriver)`).

NOTE: If you are testing on an older version of Firefox (e.g., 47 or earlier) then you don't need to download `geckodriver`. You will be able to use the legacy FirefoxDriver implementation. To do that you just need to disable Marionette (the new Firefox WebDriver implementation that geckodriver connects to) which would look like this: `driver_ = webdriver.Firefox(capabilities={'marionette': False})`

The `driver` method has two parameters, `self` and `request`. `self` is a required parameter for class methods, `request` is a parameter made available to fixtures. It enables access to loads of things during a test run. For now, the relevant piece is the ability to call `request.addfinalizer`. Actions passed to `addfinalizer` get executed after a test method completes. So we're calling `driver_.quit()` and passing it into the `addfinalizer` method.

Our test method starts with the word `test_` (that's how pytest knows it's a test) and it has two parameters (`self` and `driver`). `driver` is our access to the fixture method created at the top of the class. Since it returns a browser instance we can reference this variable directory to use Selenium commands. In this test we're visiting the login page by its URL (with `driver.get()`), finding the input fields by their ID (with `driver.find_element(By.ID, "username")`), inputting text into them (with `.send_keys()`), and submitting the form by clicking the submit button (e.g., `By.CSSSELECTOR("button")).click()`).

If we save this and run it (by running `py.test` from the command-line), it will run and pass. But there's one thing missing -- an assertion. In order to find an element to write an assertion against we need to see what the markup of the page is after submitting the login form.

Part 2: Figure Out What To Assert

Here is the markup that renders on the page after logging in.

```html
<div class="row">
  <div id="flash-messages" class="large-12 columns">
    <div data-alert="" id="flash" class="flash success">
      You logged into a secure area!
      <a href="#" class="close">x</a>
    </div>
  </div>
</div>

<div id="content" class="large-12 columns">
  <div class="example">
    <h2><i class="icon-lock"></i> Secure Area</h2>
    <h4 class="subheader">Welcome to the Secure Area. When you are done click logout
below.</h4>
    <a class="button secondary radius" href="/logout"><i class="icon-2x icon-signout">
Logout</i></a>
  </div>
</div>
```

There are a couple of elements we can use for our assertion in this markup. There's the flash message class (most appealing), the logout button (appealing), or the copy from either the `h2` or the flash message (least appealing).

Since the flash message class name is descriptive, denotes a successful login, and is less likely to

change than the copy, let's go with that.

```
class="flash success"
```

When we try to access an element like this (e.g., with a multi-worded class) we will need to use a CSS selector or an XPath.

NOTE: Both CSS selectors and XPath can work well, but the examples throughout this book will focus on how to use CSS selectors.

A Quick Primer on CSS Selectors

In web design CSS (Cascading Style Sheets) are used to apply styles to the markup (HTML) on a page. CSS is able to do this by declaring which bits of the markup it wants to alter through the use of selectors. Selenium operates in a similar manner but instead of changing the style of elements, it interacts with them by clicking, getting values, typing, sending keys, etc.

CSS selectors are a pretty straightforward and handy way to write locators, especially for hard to reach elements.

For right now, here's what you need to know. In CSS, class names start with a dot (`.`). For classes with multiple words, put a dot in front of each word, and remove the space between them (e.g., `.flash.success` for `class='flash success'`).

For a good resource on CSS Selectors I encourage you to check out [Sauce Labs' write up on them](#).

Part 3: Write The Assertion And Verify It

Now that we have our locator, let's add an assertion that uses it.

```python
# filename: tests/login_test.py
#   ...
    def test_valid_credentials(self, driver):
        driver.get("http://the-internet.herokuapp.com/login")
        driver.find_element(By.ID, "username").send_keys("tomsmith")
        driver.find_element(By.ID, "password").send_keys("SuperSecretPassword!")
        driver.find_element(By.CSS_SELECTOR, "button").click()
        assert driver.find_element(By.CSS_SELECTOR, ".flash.success").is_displayed()
```

With `assert` we are checking for a `true` Boolean response. If one is not received the test will fail. With Selenium we are seeing if the success message element is displayed on the page (with `.is_displayed`). This Selenium command returns a Boolean. So if the element is rendered on the page and is visible (e.g., not hidden or covered up by an overlay), `true` will be returned, and our test will pass.

But we're not done yet. Unfortunately if we run this it would fail due to [a documented bug in](#) `geckodriver`. Clicking the submit button triggers a new page to load and under normal circumstances Selenium would automatically wait for this new page to load before looking for the `.flash.success` element. As of right now this automatic wait is not observed which will cause Selenium to inadvertently look for an element that hasn't appeared yet. To account for this, we'll need to add a brief delay into our test.

```python
# filename: tests/login_test.py
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By
import time
# ...
    def test_valid_credentials(self, driver):
        # ...
        time.sleep(1)
        assert driver.find_element(By.CSS_SELECTOR, ".flash.success").is_displayed()
```

NOTE: Adding a hard-coded sleep (even for just `1` second) is NOT a recommended practice. There are MUCH better ways to make your test wait. They'll be covered in [Chapter 10](#) where we'll also revisit this code and improve upon it to remove this hard-coded sleep.

When we save this and run it (e.g., `py.test` from the command-line) it will run and pass just like before, but now there is an assertion which will catch a failure if something is amiss.

Just To Make Sure

Just to make certain that this test is doing what we think it should, let's change the locator in the assertion to attempt to force a failure and run it again. A simple fudging of the locator will suffice.

```python
assert driver.find_element(By.CSS_SELECTOR, ".flash.successasdf").is_displayed()
```

If it fails then we can feel reasonably confident that the test is doing what we expect and we can change the assertion back to normal before committing our code.

This trick will save you more trouble that you know. Practice it often.

Chapter 7
Verifying Your Locators

If you're fortunate enough to be working with unique IDs and Classes, then you're usually all set. But when you have to handle more complex actions like traversing a page, or you need to run down odd test behavior, it can be a real challenge to verify that you have the right locators to accomplish what you want.

Instead of the painful and tedious process of trying out various locators in your tests until you get what you're looking for, try verifying them in the browser instead.

A Solution

Built into every major browser is the ability to verify locators from the JavaScript Console.

Simply open the developer tools in your browser and navigate to the JavaScript Console (e.g., right-click on an element, select `Inspect Element`, and click into the `Console` tab). From here it's a simple matter of specifying the locator you want to look up by the `$$('')` command (e.g., `$$('#username')`) and hovering your mouse over what is returned in the console. The element that was found will be highlighted in the viewport pane.

An Example

Let's try to identify the locators necessary to traverse a few levels into a large set of nested divs.

```
# a snippet from http://the-internet.herokuapp.com/large

<div id='siblings'>
  <div id='sibling-1.1'>1.1
  <div id='sibling-1.2'>1.2</div>
  <div id='sibling-1.3'>1.3</div>
    <div id='sibling-2.1'>2.1
    <div id='sibling-2.2'>2.2</div>
    <div id='sibling-2.2'>2.3</div>
      <div id='sibling-3.1'>3.1
      <div id='sibling-3.2'>3.2</div>
      <div id='sibling-3.2'>3.3</div>
        <div id='sibling-3.1'>4.1
        <div id='sibling-3.2'>4.2</div>
        <div id='sibling-3.2'>4.3</div>
        <!-- ... -->
```

If we perform a `FindElement` action using the following locator, it works.

```
driver.find_element_by_css_selector("div#siblings > div:nth-of-type(1) >
div:nth-of-type(1)");
```

But if we try to go one level deeper with the same strategy, it won't work.

```
driver.find_element_by_css_selector("div#siblings > div:nth-of-type(1) >
div:nth-of-type(1) > div:nth-of-type(1)");
```

Fortunately with our in-browser approach to verifying our locators, we can quickly discern where the issue is. Here's what it shows us for the locators that "worked".
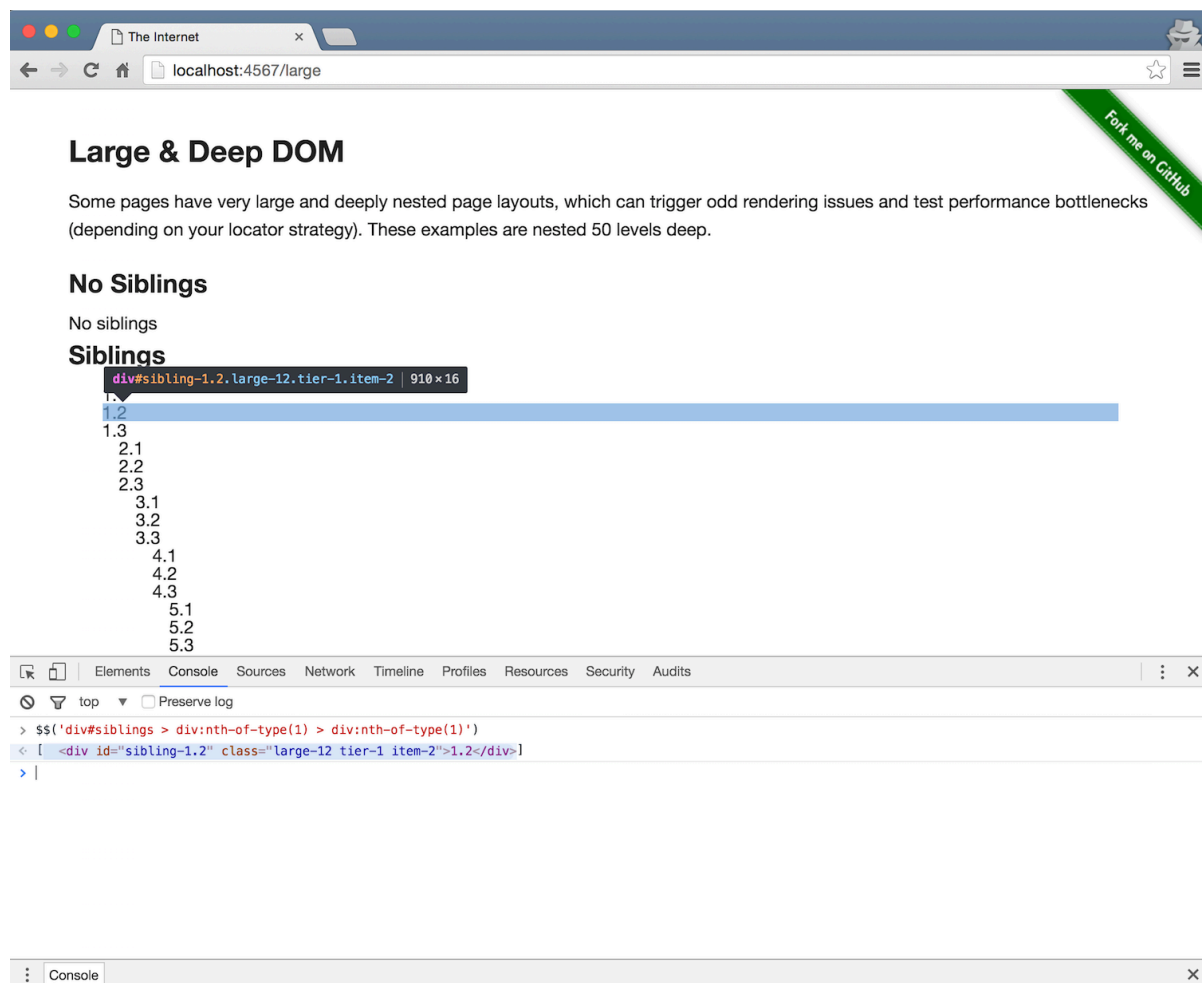
It looks like our locators are scoping to the wrong part of the first level (1.2). But we need to reference the third part of each level (e.g., 1.3, 2.3, 3.3) in order to traverse deeper since the nested divs live under the third part of each level.

So if we try this locator instead, it should work.

```
driver.find_element_by_css_selector("div#siblings > div:nth-of-type(1) >
div:nth-of-type(3) > div:nth-of-type(3)");
```

And we can confirm that it works before changing any test code by looking in the JavaScript
Console.

This should help save you time and frustration when running down tricky locators in your tests. It
definitely has for me.

Chapter 8
Writing Re-usable Test Code

One of the biggest challenges with Selenium tests is that they can be brittle and challenging to maintain over time. This is largely due to the fact that things in the application you're testing change -- causing your tests to break.

But the reality of a software project is that change is a constant. So we need to account for this reality somehow in our test code in order to be successful.

Enter Page Objects.

A Page Objects Primer

Rather than write your test code directly against your app, you can model the behavior of your application into simple objects and write your tests against them instead. That way when your app changes and your tests break, you only have to update your test code in one place to fix it.

With this approach we not only get the benefit of controlled chaos, we also get reusable functionality across our suite of tests and more readable tests.

An Example

Part 1: Create A Page Object And Update Test

Let's take our login example from earlier, create a page object for it, and update our test accordingly.

First we'll need to create a new folder called `pages` in the root of our project (just like we did for `tests`). Then let's add a file to the `pages` directory called `login_page.py` and a requisite `__init__.py` file. When we're done our directory structure should look like this.

```
pages
    __init__.py
    login_page.py
tests
    __init__.py
    login_test.py
vendor
    geckodriver
```

And here's the code that goes with it.

```python
# filename: pages/login_page.py
from selenium.webdriver.common.by import By
import time


class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}
    _submit_button = {"by": By.CSS_SELECTOR, "value": "button"}
    _success_message = {"by": By.CSS_SELECTOR, "value": ".flash.success"}
    _failure_message = {"by": By.CSS_SELECTOR, "value": ".flash.error"}

    def __init__(self, driver):
        self.driver = driver
        self.driver.get("http://the-internet.herokuapp.com/login")

    def with_(self, username, password):
        self.driver.find_element(self._username_input["by"],
                                 self._username_input["value"]).send_keys(username)
        self.driver.find_element(self._password_input["by"],
                                 self._password_input["value"]).send_keys(password)
        self.driver.find_element(self._submit_button["by"],
                                 self._submit_button["value"]).click()
        time.sleep(1)

    def success_message_present(self):
        return self.driver.find_element(self._success_message["by"], self.
_success_message["value"]).is_displayed()
```

At the top of the file we include the Selenium class to handle our locators. We then declare the class (e.g., `class Login():`), specify our local variables for the page's locators, and specify three methods.

The first method (e.g., `def __init__(self, driver):`) is the constructor. It will run whenever a new instance of this class is created. In order for this class to work we need access to the Selenium driver object, so we accept it as a parameter here and store it in the `self.driver` class variable (so other methods in the class can access it). Then the login page is visited (with `driver.get()`).

The second method (e.g., `public void with_(self, username, password)`) is the core functionality of the login page. It's responsible for filling in the login form and submitting it. By accepting parameters for the username and password we're able to make the functionality here reusable for additional tests. Its name has a trailing underscore because the word `with` is protected keyword in Python. The general guidance for this situation is to either find a synonym

for the method name or to keep the name and end it with an underscore. You can find out more about the Python style guidelines [here](#).

The last method (e.g., `def success_message_present(self):`) is the display check from earlier that was used in our assertion. It will return a Boolean result just like before.

Now let's update our test to use this page object.

```python
# filename: tests/login_test.py
import pytest
from selenium import webdriver
from pages import login_page


class TestLogin():

    @pytest.fixture
    def login(self, request):
        driver_ = webdriver.Firefox()

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return login_page.LoginPage(driver_)

    def test_valid_credentials(self, login):
        login.with_("tomsmith", "SuperSecretPassword!")
        assert login.success_message_present()
```

Before we can use the page object we first need to import it (e.g., `from pages import login_page`).

Then it's a simple matter of updating our setup fixture to return an instance of the login page and updating the test method with the new actions.

Now the test is more concise and readable. And when you save everything and run it, it will run and pass just like before.

Part 2: Write Another Test

Creating a page object may feel like more work than what we started with initially. But it's well worth the effort since we're in a much sturdier position and able to easily write follow-on tests (since the specifics of the page are abstracted away for simple reuse).

Let's add another test for a failed login to demonstrate.

First, let's take a look at the markup that gets rendered when we provide invalid credentials:

```html
<div id="flash-messages" class="large-12 columns">
  <div data-alert="" id="flash" class="flash error">
    Your username is invalid!
    <a href="#" class="close">x</a>
  </div>
</div>
```

Here is the element we'll want to use in our assertion.

```
class="flash error"
```

Let's add a locator for this element to our page object along with a new method to perform a display check against it.

```python
# filename: pages/login_page.py
# ...
class LoginPage():
    # ...
    _failure_message = {"by": By.CSS_SELECTOR, "value": ".flash.error"}
    # ...
    def failure_message_present(self):
        return self.driver.find_element(self._failure_message["by"],
                                        self._failure_message["value"]).is_displayed()
    # ...
```

Now we're ready to add a test for failed login to our `tests/login_test.py` file.

```python
# filename: tests/login_test.py
# ...
    def test_invalid_credentials(self, login):
        login.with_("tomsmith", "bad password")
        assert login.failure_message_present()
```

If we save these changes and run our tests we will see two browser windows open (one after the other) testing for successful and failure login scenarios.

Why Asserting False Won't Work (yet)

You may be wondering why we didn't just check to see if the success message wasn't present in our assertion.

```
assert login.success_message_present() == False
```

There are two problems with this approach. First, our test will fail. This is because Selenium errors when it looks for an element that's not present on the page -- which looks like this:

```
NoSuchElementException: Message: Unable to locate element: {"method":"css selector",
"selector":".flash.success"}
```

But don't worry, we'll address this in the next chapter.

Second, the absence of a success message doesn't necessarily indicate a failed login. The assertion we ended up with originally is more effective.

Part 3: Confirm We're In The Right Place

Before we can call our page object finished, there's one more addition we should make. We'll want to add an assertion to make sure that Selenium is in the right place before proceeding. This will help add some resiliency to our test.

As a rule, you want to keep assertions in your tests and out of your page objects. But this is an exception to the rule.

```python
# filename: pages/login_page.py
class LoginPage():
    _login_form = {"by": By.ID, "value": "login"}
    # ...

    def __init__(self, driver):
        self.driver = driver
        self.driver.get("http://the-internet.herokuapp.com/login")
        assert self.driver.find_element(
            self._login_form["by"], self._login_form["value"]).is_displayed()
# ...
```

After adding a new locator to the page object for the login form we add an assertion to the constructor, just after Selenium visits the login page. This will check that the login form is displayed. If it is the tests using this page object will proceed. If not the test will fail and provide an output message stating that the login form wasn't present.

Now when we save everything and run our tests they will run just like before. But now we can feel confident that the tests will only proceed if login page is in a ready state.

Outro

With Page Objects you'll be able to easily maintain and extend your tests. But how you write your Page Objects may vary depending on your preference and experience. The example demonstrated above is a simple approach. It's worth taking a look at the Selenium project wiki page for Page Objects [here](#) (even if its examples are only written in Java).

Chapter 9
Writing Really Re-usable Test Code

In the previous chapter we stepped through creating a simple page object to capture the behavior of the page we were interacting with. While this was a good start, it leaves some room for improvement.

As our test suite grows and we add more page objects we will start to see common behavior that we will want to use over and over again throughout our suite. If we leave this unchecked we will end up with duplicative code which will slowly make our page objects harder to maintain.

Right now we are using Selenium actions directly in our page object. While on the face of it this may seem fine, it has some long term impacts, like:

- slower page object creation due to the lack of a simple Domain Specific Language (DSL)
- test maintenance issues when the Selenium API changes (e.g., major changes between Selenium RC and Selenium WebDriver)
- the inability to swap out the driver for your tests (e.g., mobile, REST, etc.)

With a Base Page Object (a.k.a. a facade layer) we can easily side step these concerns by abstracting all of our common actions into a central class and leveraging it in our page objects.

An Example

Let's step through an example with our login page object.

Part 1: Create The Base Page Object

First let's create the base page object by adding a file called `base_page.py` to the `pages` directory.

```
pages
    __init__.py
    base_page.py
    login_page.py
tests
    __init__.py
    login_test.py
vendor
    geckodriver
```

Next let's populate the file.

```python
#  filename: pages/base_page.py
class BasePage():
    def __init__(self, driver):
        self.driver = driver

    def _visit(self, url):
        self.driver.get(url)

    def _find(self, locator):
        return self.driver.find_element(locator["by"], locator["value"])

    def _click(self, locator):
        self._find(locator).click()

    def _type(self, locator, input_text):
        self._find(locator).send_keys(input_text)

    def _is_displayed(self, locator):
        return self._find(locator).is_displayed()
```

After declaring the class (e.g., `class BasePage():`) we receive and store an instance of Selenium just like in our Login page object. But what's different here is the methods that come after the constructor (e.g., `_visit` , `_find` , `_click` , `_type` , and `_is_displayed`). Each one stores a specific behavior we've used in our tests. Some of the names are the same as you've seen in Selenium, others renamed (for improved readability).

Now that we have all of our Selenium actions in one place, let's update our login page object to leverage this facade.

```python
#  filename: pages/login_page.py
from selenium.webdriver.common.by import By
from base_page import BasePage


class LoginPage(BasePage):
    _login_form = {"by": By.ID, "value": "login"}
    _username_input = {"by": By.ID, "value": "username"}
    _password_input = {"by": By.ID, "value": "password"}
    _submit_button = {"by": By.CSS_SELECTOR, "value": "button"}
    _success_message = {"by": By.CSS_SELECTOR, "value": ".flash.success"}
    _failure_message = {"by": By.CSS_SELECTOR, "value": ".flash.error"}

    def __init__(self, driver):
        self.driver = driver
        self._visit("http://the-internet.herokuapp.com/login")
        assert self._is_displayed(self._login_form)

    def with_(self, username, password):
        self._type(self._username_input, username)
        self._type(self._password_input, password)
        self._click(self._submit_button)

    def success_message_present(self):
        return self._is_displayed(self._success_message)

    def failure_message_present(self):
        return self._is_displayed(self._failure_message)
```

Two fundamental things have changed in our Login page object.

We've established inheritance between `BasePage` and `LoginPage` (with `class LoginPage(BasePage):`. This means that `LoginPage` is now a child of `BasePage`. This enables us to swap out all of the Selenium actions to use the methods we just specified in the `BasePage`.

If we save everything and run our tests they will run and pass just like before. But now our page objects are more readable, simpler to write, and easier to maintain and extend.

Part 2: Add Some Error Handling

Remember in the previous chapter when we ran into an error with Selenium when we looked for an element that wasn't on the page? Let's address that now.

To recap -- here's the error message we saw:

```
NoSuchElementException: Message: Unable to locate element: {"method":"css selector",
"selector":".flash.success"}
```

The important thing to note is the name of the exception Selenium offered up --
`NoSuchElementException`. Let's modify the `is_displayed` method in our base page object to
handle it.

```python
#   filename: pages/base_page.py
from selenium.common.exceptions import NoSuchElementException
#   ...
    def _is_displayed(self, locator):
        try:
            self._find(locator).is_displayed()
        except NoSuchElementException:
            return False
        return True
```

By wrapping our Selenium action (e.g., `self._find(locator).is_displayed()`) in a `try` / `except`
we're able to catch the exception and return `False` instead. This will enable us to see if an
element is on the page. If it's not, we'll receive a `False` Boolean rather than an exception. And if
it is on the page, it will return `True` just like before.

With this new handling in place, let's revisit our `test_invalid_credentials)` test and alter it so it
checks to see if the success message is not present (which would normally trigger a
`NoSuchElementException` exception) to make sure things work as we expect.

```python
#   filename: tests/login_test.py
#   ...
    def test_invalid_credentials(self, login):
        login.with_("tomsmith", "bad password")
        assert login.success_message_present() == False
```

When we save our changes and run this test it will run and pass without throwing an exception.

Chapter 10
Writing Resilient Test Code

Ideally you should be able to write your tests once and run them across all supported browsers. While this is a rosy proposition, there is some work to make this a reliable success. And sometimes there may be a hack or two involved. But the lengths you must go really depends on the browsers you care about and the functionality you're dealing with in your application.

By using high quality locators we're already in good shape, but there are still some issues to deal with. Most notably... timing. This is especially true when working with dynamic, JavaScript heavy pages (which is more the rule than the exception in a majority of web applications you'll deal with).

But there is a simple approach that makes up the bedrock of reliable and resilient Selenium tests -- and that's how you wait and interact with elements. The best way to accomplish this is through the use of explicit waits.

An Explicit Waits Primer

Explicit waits are applied to individual test actions. Each time you want to use one you specify an amount of time (in seconds) and the Selenium action you want to accomplish.

Selenium will repeatedly try this action until either it can be accomplished, or until the amount of time specified has been reached. If the latter occurs, a timeout exception will be thrown.

An Example

Let's step through an example that demonstrates this against [a dynamic page on the-internet](). The functionality is pretty simple -- there is a button. When you click it a loading bar appears for 5 seconds. After that it disappears and is replaced with the text `Hello World!`.

Part 1: Create A New Page Object And Update The Base Page Object

Here's the markup from the page.

```
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 1: Element on page that is hidden</h4>


  <br>


  <div id="start">
    <button>Start</button>
  </div>


  <div id="finish" style="display:none">
    <h4>Hello World!</h4>
  </div>


</div>
```

At a glance it's simple enough to tell that there are unique `id` attributes that we can use to find and click on the start button and verify the finish text.

When writing automation for new functionality like this, you may find it easier to write the test first (to get it working how you'd like) and then create a page object for it (pulling out the behavior and locators from your test). There's no right or wrong answer here. Do what feels intuitive to you. But for this example, we'll create the page object first, and then write the test.

Let's create a new page object file called `dynamic_loading_page.py` in the `pages` directory.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    login_test.py
vendor
    geckodriver
```

In this file we'll establish inheritance to the base page object and specify the locators and behavior we'll want to use.

```python
# filename: pages/dynamic_loading_page.py
from selenium.webdriver.common.by import By
from base_page import BasePage


class DynamicLoadingPage(BasePage):
    _start_button = {"by": By.CSS_SELECTOR, "value": "#start button"}
    _finish_text = {"by": By.ID, "value": "finish"}

    def ___init___(self, driver):
        self.driver = driver

    def load_example(self, example_number):
        self._visit("/dynamic_loading/" + example_number)
        self._click(self._start_button)

    def finish_text_present(self):
        return self._wait_for_is_displayed(self._finish_text, 10)
```

Since there are two examples to choose from on the-internet we created the method `load_example` which accepts a number as an argument so we can specify which of the examples we want to visit and start.

And similar to our Login page object, we have a display check for the finish text (e.g., `finish_text_present()`). This check is slightly different though. Aside from the different name, it has a second argument (an integer value of `10`). This second argument is how we'll specify how long we'd like Selenium to wait for an element to be displayed before giving up.

Let's update our base page object to enable explicit waits by adding an additional `wait_for_is_displayed` method.

```python
# filename: pages/base_page.py
from selenium.common.exceptions import NoSuchElementException, TimeoutException
from selenium.webdriver.support import expected_conditions
from selenium.webdriver.support.wait import WebDriverWait
# ...
    def _wait_for_is_displayed(self, locator, timeout):
        try:
            wait = WebDriverWait(self.driver, timeout)
            wait.until(
                expected_conditions.visibility_of_element_located(
                    (locator['by'], locator['value'])))
        except TimeoutException:
            return False
        return True
```

Selenium comes with a wait function which we wrap in a new method.

The `wait_for_is_displayed` method takes a locator and a timeout. Inside the method we create an instance of `WebDriverWait`, pass in the timeout (which is assumed to be in seconds), and then call `wait.until`. With `wait.until` we specify the condition we want to wait for and the locator it is in reference to. In this case the expected condition we want to wait for is `visibility_of_element_located` which is ultimately doing the same thing as our previous display lookup. It's just using a different API call that will work with the explicit waits function. You can see a full list of Selenium's `ExpectedConditions` here.

If all goes well then we `return True` for the method. If the condition is not met by Selenium in the amount of time provided it will throw a timeout exception. So we catch it (with a `try`/`except` block) and `return False` instead.

More On Explicit Waits

The major benefit of explicit waits is that if the behavior on the page takes longer than we expect (e.g., due to slow load times, or a feature change), we can simply adjust this one wait time to fix the test -- rather than increase a blanket wait time (which impacts every test). And since the wait is dynamic (e.g., constantly polling), it won't take the full amount of time to complete (like a hard-coded sleep would).

If you're thinking about mixing explicit waits with an implicit wait -- don't. If you use both together you're going to run into issues later on due to inconsistent implementations of the implicit wait functionality across local and remote browser drivers. Long story short, you'll end up with tests that could fail randomly and when they do they will be hard to debug. You can read more about the specifics here.

Part 2: Write A Test To Use The New Page Object

Now that we have our new page object and an updated base page, it's time to write our test to use it.

Let's create a new file called `dynamic_loading_test.py` in the `tests` directory.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    dynamic_loading_test.py
    login_test.py
vendor
    geckodriver
```

The contents of this test file are similar to `login_test` with regards to its setup and structure.

```python
# filename: tests/dynamic_loading_test.py
import pytest
from selenium import webdriver
from pages import dynamic_loading_page


class TestDynamicLoading():

    @pytest.fixture
    def dynamic_loading(self, request):
        driver_ = webdriver.Firefox()

        def quit():
            driver_.quit()

        request.addfinalizer(quit)
        return dynamic_loading_page.DynamicLoadingPage(driver_)

    def test_hidden_element(self, dynamic_loading):
        dynamic_loading.load_example("1")
        assert dynamic_loading.finish_text_present()
```

In our test (e.g., `def test_hidden_element:`) we are visiting the first dynamic loading example and clicking the start button (which is accomplished in `dynamic_loading.load_example("1")`). We're then asserting that the finish text gets rendered.

When we save this and run it it will:

- Launch a browser
- Visit the page
- Click the start button

- Wait for the loading bar to complete
- Find the finish text
- Assert that it is displayed.
- Close the browser

Part 3: Update Page Object And Add A New Test

Let's step through one more example to see if our explicit wait holds up.

[The second dynamic loading example](#) is laid out similarly to the last one. The only difference is that it renders the final text after the progress bar completes (whereas the previous example had the element on the page but it was hidden).

Here's the markup for it.

```html
<div class="example">
  <h3>Dynamically Loaded Page Elements</h3>
  <h4>Example 2: Element rendered after the fact</h4>


  <br>


  <div id="start">
    <button>Start</button>
  </div>


  <br>
</div>
```

In order to find the selector for the finish text element we need to inspect the page after the loading bar sequence finishes. Here's what it looks like.

```html
<div id="finish" style=""><h4>Hello World!</h4></div>
```

Let's add a second test to `dynamic_loading_test.py` called `test_rendered_element())` that will load this second example and perform the same check as we did for the previous test.

```python
# filename: tests/dynamic_loading_test.py
# ...
    def test_rendered_element(self, dynamic_loading):
        dynamic_loading.load_example("2")
        assert dynamic_loading.finish_text_present()
```

When we run both tests we will see that the same approach will work in both cases of how the page is constructed.

Revisiting Login

Now that we have an explicit wait helper method available to us, let's revisit the login page object and refactor it to remove the hard-coded sleep.

```python
# filename: pages/login_page.py
from selenium.webdriver.common.by import By
from base_page import BasePage
# ...
    def with_(self, username, password):
        self._type(self._username_input, username)
        self._type(self._password_input, password)
        self._click(self._submit_button)

    def success_message_present(self):
        self._wait_for_is_displayed(self._success_message, 1)
        return self._is_displayed(self._success_message)

    def failure_message_present(self):
        self._wait_for_is_displayed(self._failure_message, 1)
        return self._is_displayed(self._failure_message)
```

By using calls to `_wait_for_is_displayed` from `success_message_present` and `failure_message_present` we are able to remove the `time.sleep(1)` call in the `with_` method in favor of an explicit wait. This will cause Selenium to wait up to `1` second for the correct element to be displayed on the page rather than waiting regardless of the state of the page.

Browser Timing

Using explicit waits gets you pretty far. But there are a few things you'll want to think about when it comes to writing your tests to work on various browsers.

It's simple enough to write your tests locally against Firefox and assume you're all set. But once you start to run things against other browsers, you may be in for a rude awakening. The first thing you're likely to run into is the speed of execution. A lot of your tests will start to fail when you point them at either Chrome or Internet Explorer, and likely for different reasons.

Chrome execution can sometimes be faster than Firefox, so you could see some odd timeout failures. This is an indicator that you need to add explicit waits to parts of your page objects that don't already have them. And the inverse is true when running things against older version of Internet Explorer. This is an indicator that your explicit wait times are not long enough since the browser is taking longer to respond -- so your tests timeout.

The best approach to solve this is an iterative one. Run your tests in a target browser and see

which ones fail. Take each failed test, adjust your code as needed, and re-run it against the target browser until they all pass. Repeat for each browser you care about until everything is green.

Closing Thoughts

By explicitly waiting to complete an action, our tests are in a much more resilient position because Selenium will keep trying for a reasonable amount of time rather than trying just once. And each action can be tuned to meet the needs of each circumstance. Couple that with the dynamic nature of explicit waits, and you have something that will work in a multitude of circumstances -- helping you endure even the toughest of browsers to automate.

This is one of the most important concepts in testing with Selenium. Use explicits waits often.

Chapter 11
Prepping For Use

Now that we have some tests and page objects, we'll want to start thinking about how to structure our test code to be more flexible. That way it can scale to meet our needs.

Part 1: Global Setup & Teardown

We'll start by pulling the Selenium setup and teardown out of our tests and into a central location.

In pytest there is a central file that we can use for this called `conftest.py` that will automatically be found and used during test execution. So let's create a new file called `conftest.py` in the `tests` directory.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    conftest.py
    dynamic_loading_test.py
    login_test.py
vendor
    geckodriver
```

And here are the contents of the file.

```
# filename: tests/conftest.py
import pytest
from selenium import webdriver


@pytest.fixture
def driver(request):
    driver_ = webdriver.Firefox()

    def quit():
        driver_.quit()

    request.addfinalizer(quit)
    return driver_
```

After importing the necessary classes for pytest and Selenium we specify the fixture setup and teardown code that we used directly in our tests.

Now let's update our tests to clean up the fixtures and remove any unnecessary `import` statements. When we're done our test files should look like this.

```
# filename: tests/login_test.py
import pytest
from pages import login_page


class TestLogin():

    @pytest.fixture
    def login(self, driver):
        return login_page.LoginPage(driver)

    def test_valid_credentials(self, login):
        login.with_("tomsmith", "SuperSecretPassword!")
        assert login.success_message_present()

    def test_invalid_credentials(self, login):
        login.with_("tomsmith", "bad password")
        assert login.failure_message_present()
```

```
# filename: tests/dynamic_loading_test.py
import pytest
from pages import dynamic_loading_page


class TestDynamicLoading():

    @pytest.fixture
    def dynamic_loading(self, driver):
        return dynamic_loading_page.DynamicLoadingPage(driver)

    def test_hidden_element(self, dynamic_loading):
        dynamic_loading.load_example("1")
        assert dynamic_loading.finish_text_present()

    def test_rendered_element(self, dynamic_loading):
        dynamic_loading.load_example("2")
        assert dynamic_loading.finish_text_present()
```

Part 2: Base URL

It's a given that we'll need to run our tests against different environments (e.g., local, test, staging, production, etc.). So let's make it so we can specify a different base URL for our tests at runtime.

First, let's create a file called `config.py` in the `tests` directory.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    config.py
    conftest.py
    dynamic_loading_test.py
    login_test.py
vendor
    geckodriver
```

In it we'll specify an empty placeholder variable for `baseurl` that we'll use in `conftest.py`.

```
baseurl = ""
```

Now let's update our our test fixture in `conftest.py` to enable a command-line argument for a base URL and store the received value in the placeholder variable in `config.py`.

```python
# filename: tests/conftest.py
# ...
def pytest_addoption(parser):
    parser.addoption("--baseurl",
                     action="store",
                     default="http://the-internet.herokuapp.com",
                     help="base URL for the application under test")


@pytest.fixture
def driver(request):
    config.baseurl = request.config.getoption("--baseurl")
# ...
```

There's a nice helper method in pytest called `pytest_addoption(parser)`. It enables us to specify a custom runtime flag and set a sensible default. This value gets passed into the `request` variable that is available in our test fixture. So we can easily pluck the value out of it and store it in `config.baseurl`.

Now we are able to use the base URL in other parts of our test code, like our Base Page Object.

```python
# filename: pages/base_page.py
from tests import config
# ...
    def _visit(self, url):
        if url.startswith("http"):
            self.driver.get(url)
        else:
            self.driver.get(config.baseurl + url)
# ...
```

In `_visit` there could be a case where we'll want to navigate to a full URL so to be safe we've added a conditional check of the `url` parameter to see if a full URL was passed in. If so, we'll visit it. If not, the `config.baserl` will be combined with the URL path that was passed in to `url` to create a full URL.

Now all we need to do is update our page objects so they're no longer using hard-coded URLs when calling `_visit`.

```
# filename: pages/login_page.py
# ...
    def __init__(self, driver):
        self.driver = driver
        self._visit("/login")
        assert self._is_displayed(self._login_form)
# ...
```

```
# filename: pages/dynamic_loading_page.py
# ...
    def load_example(self, example_number):
        self._visit("/dynamic_loading/" + example_number)
        self._click(self._start_button)
# ...
```

Outro

Now when running our tests, we can specify a different base URL by providing an extra command-line flag (e.g., `py.test --baseurl=url`). We're also in a better position now with our setup and teardown abstracted into a central location. Now we can easily extend our test framework to run our tests on other browsers.

Chapter 12
Running A Different Browser Locally

It's straightforward to get your tests running locally against Firefox (that's what we've been doing up until now). But when you want to run them against a different browser like Chrome, Safari, or Internet Explorer you quickly run into configuration overhead that can seem overly complex and lacking in good documentation or examples.

A Brief Primer On Browser Drivers

With the introduction of WebDriver (circa Selenium 2) a lot of benefits were realized (e.g., more effective and faster browser execution, no more single host origin issues, etc). But with it came some architectural and configuration differences that may not be widely known. Namely -- browser drivers.

WebDriver works with each of the major browsers through a browser driver which is (ideally but not always) maintained by the browser manufacturer. It is an executable file (consider it a thin layer or a shim) that acts as a bridge between Selenium and the browser.

Let's step through an example using [ChromeDriver](#).

An Example

Before starting, we'll need to download the latest ChromeDriver binary executable for our operating system from [here](#) (pick the highest numbered directory) and store the unzipped contents of it somewhere. The simplest thing to do is create a new folder for it (and other things like it) in our test code.

Let's create a `vendor` directory in the root of our project and place the ChromeDriver binary file there.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    config.py
    conftest.py
    dynamic_loading_test.py
    login_test.py
vendor
    chromedriver
    geckodriver
```

In order for Selenium to use this binary we have to make sure it knows where it is. There are two ways to do that. We can add `chromedriver` to the path of our system, or we can pass in the path to `chromedriver` file when configuring Selenium. For simplicity, let's go with the latter option.

NOTE: There is a different ChromeDriver binary for each major operating system. If you're using Windows be sure to use the one that ends with `.exe` and specify it in your configuration. This example was built to run on OSX (which does not have a file extension).

We'll also want to make sure our test suite can run either Firefox or Chrome. To do that, we'll need to make a couple of changes.

First, let's add a `browser` value to our `config.py` file.

```python
# filename: config.py
baseurl = ""
browser = ""
```

Now to update `conftest.py` to receive the browser name as a command-line argument.

```python
# filename: tests/conftest.py
import pytest
from selenium import webdriver
import config
import os


def pytest_addoption(parser):
    parser.addoption("--baseurl",
                     action="store",
                     default="http://the-internet.herokuapp.com",
                     help="base URL for the application under test")
    parser.addoption("--browser",
                     action="store",
                     default="firefox",
                     help="the name of the browser you want to test with")


@pytest.fixture
def driver(request):
    config.baseurl = request.config.getoption("--baseurl")
    config.browser = request.config.getoption("--browser").lower()

    if config.browser == "firefox":
        driver_ = webdriver.Firefox()
    elif config.browser == "chrome":
        _chromedriver = os.path.join(os.getcwd() + 'vendor', 'chromedriver')
        driver_ = webdriver.Chrome(_chromedriver)

    def quit():
        driver_.quit()

    request.addfinalizer(quit)
    return driver_
```

After we specify a new command-line argument and a sensible default (e.g., `"firefox"`) we grab the value for it in our fixture, storing it in `config.browser`. We then use this in a conditional to check which value was passed in. If `"firefox"` was provided, we create a Selenium instance like we've been doing. If `"chrome"` is provided, we grab the full path to the `chromedriver` binary and pass it as an argument during instantiation in order to get at Chrome instance.

Now we can specify Chrome as our browser when launching our tests (e.g., `py.test --browser=chrome`).

It's worth noting that this will only be reasonably performant since it is launching and terminating the ChromeDriver binary executable before and after every test. There are alternative ways to set

this up, but this is good enough to see where our tests fall down in Chrome (and it will not be the primary way we will run our tests a majority of the time anyway -- more on that later in the book).

Additional Browsers

A similar approach can be applied to other browser drivers, with the only real limitation being the operating system you're running. But remember -- no two browser drivers are alike. Be sure to check out the documentation for the browser you care about to find out the specific requirements:

- [ChromeDriver](#)
- [EdgeDriver](#)
- [FirefoxDriver](#)
- [InternetExplorer Driver](#)
- [SafariDriver](#)

Chapter 13
Running Browsers In The Cloud

If you've ever needed to test features in an older browser like Internet Explorer 8 or 9 then odds are you ran a virtual machine (VM) on your computer with a "legit" version of Windows.

Handy, but what happens when you need to check things on multiple versions of IE? Now you're looking at multiple VMs. And what about when you need cover other browser and Operating System (OS) combinations? Now you're looking at provisioning, running, and maintaining your own set of machines and standing up something like Selenium Grid to coordinate tests across them.

Rather than take on the overhead of a test infrastructure you can easily outsource this to a third-party cloud provider like [Sauce Labs](#).

A Selenium Remote, Selenium Grid, And Sauce Labs Primer

At the heart of Selenium at scale is the use of Selenium Grid and Selenium Remote.

Selenium Grid lets you distribute test execution across several machines and you connect to it with Selenium Remote. You tell the Grid which browser and OS you want your test to run on through the use of Selenium Remote's `DesiredCapabilities`.

Under the hood this is how Sauce Labs works. They are ultimately running Selenium Grid behind the scenes, and they receive and execute tests through Selenium Remote and the `DesiredCapabilities` you set.

Let's dig in with an example.

An Example

Part 1: Initial Setup

NOTE: You'll need an account to use Sauce Labs. Their [free trial](#) offers enough to get you started. And if you're signing up because you want to test an open source project, then be sure to check out their [Open Sauce account](#).

With Sauce Labs we need to provide specifics about what we want in our test environment, our credentials, and configure Selenium a little bit differently. Let's start by updating our `config.py` file to store these details.

```python
# filename: config.py
baseurl = ""
host = ""
browser = ""
browserversion = ""
platform = ""
```

In addition to the `baseurl` and `browser` variables, we've added some more (e.g., `host`, `browserversion`, and `platform`).

`host` enables us to specify whether our tests run locally or on Sauce Labs.

With `browser`, `browserveresion`, and `platform` we can specify which browser and operating system combination we want our tests to run on. You can see a full list of Sauce's available platform options here. They also have a handy configuration generator (which will tell you what values to plug into your test) here.

Now we can update our `conftest.py` file to work with Selenium Remote. Let's start by pulling in the new configuration values.

```python
# filename: tests/conftest.py
# ...
def pytest_addoption(parser):
    parser.addoption("--baseurl",
                     action="store",
                     default="http://the-internet.herokuapp.com",
                     help="base URL for the application under test")
    parser.addoption("--host",
                     action="store",
                     default="saucelabs",
                     help="where to run your tests: localhost or saucelabs")
    parser.addoption("--browser",
                     action="store",
                     default="internet explorer",
                     help="the name of the browser you want to test with")
    parser.addoption("--browserversion",
                     action="store",
                     default="10.0",
                     help="the browser version you want to test with")
    parser.addoption("--platform",
                     action="store",
                     default="Windows 7",
                     help="the operating system to run your tests on (saucelabs only)")
# ...
```

And now to update our fixture.

```python
# filename: tests/conftest.py
# ...
@pytest.fixture
def driver(request):
    config.baseurl = request.config.getoption("--baseurl")
    config.host = request.config.getoption("--host").lower()
    config.browser = request.config.getoption("--browser").lower()
    config.browserversion = request.config.getoption("--browserversion").lower()
    config.platform = request.config.getoption("--platform").lower()

    if config.host == "saucelabs":
        _desired_caps = {}
        _desired_caps["browserName"] = config.browser
        _desired_caps["version"] = config.browserversion
        _desired_caps["platform"] = config.platform
        _credentials = os.environ["SAUCE_USERNAME"] + ":" + os.environ[
"SAUCE_ACCESS_KEY"]
        _url = "http://" + _credentials + "@ondemand.saucelabs.com:80/wd/hub"
        driver_ = webdriver.Remote(_url, _desired_caps)
    elif config.host == "localhost":
        if config.browser == "firefox":
            driver_ = webdriver.Firefox()
        elif config.browser == "chrome":
            _chromedriver = os.getcwd() + "/vendor/chromedriver"
            driver_ = webdriver.Chrome(_chromedriver)
# ...
```

In the beginning of our fixture we grab the command-line values and store them in an instance of `config.py`, which we'll use throughout the fixture. We also amended our conditional flow to check the `host` variable first. We start by checking to see if it's set to `"localhost"` or `"saucelabs"`. If it's set to `"localhost"` we carry on just like before (checking the `browser` value to determine which browser to launch locally).

If it's set to `"saucelabs"` we create a Desired Capabilities object (e.g., `_desired_caps`), populate it with `browser`, `browserversion`, `platform` values and Sauce Labs account credentials (e.g., `"username"` and `"accessKey"` which are being pulled out of environment variables I've configured on my local machine). We then connect to Sauce Labs using Selenium Remote and pass in the `_desired_caps` object. This will return a Selenium WebDriver instance that we can use just like when running our tests locally, except the browser is living on a machine in Sauce Labs' cloud.

If we save everything and run our tests they will execute in Sauce Labs and on the account dashboard we'll see our tests running in Internet Explorer 10 on Windows 7. To run the tests on different browser and operating system combinations, then simply provide their values as

command-line options (e.g., `py.test --browser=name --browserversion=version --platform=os`). For a full list of possible options be sure to check out [the Sauce Labs Platform Configurator](#).

Part 2: Test Name

It's great that our tests are running on Sauce Labs. But we're not done yet because the test name in each Sauce job is getting set to `unnamed job`. This makes it extremely challenging to know what test was run in the job. To remedy this we'll need to pass in the test name in the `_desired_caps` object.

```python
# filename: tests/conftest.py
# ...
    if config.host == "saucelabs":
        _desired_caps = {}
        _desired_caps["browserName"] = config.browser
        _desired_caps["version"] = config.browserversion
        _desired_caps["platform"] = config.platform
        _desired_caps["name"] = request.cls.__name__ + "." + request.function.__name__
        _credentials = os.environ["SAUCE_USERNAME"] + ":" + os.environ[
"SAUCE_ACCESS_KEY"]
        _url = "http://" + _credentials + "@ondemand.saucelabs.com:80/wd/hub"
        driver_ = webdriver.Remote(_url, _desired_caps)
# ...
```

From the `request` object we can pull the name of the test class (e.g., `request.cls.__name__`) and the name of the test method that's currently running (e.g., `request.function.__name__`). We grab this information and store it in the `_desired_caps` object and pass it into the Sauce Labs session.

Now when we run our tests in Sauce Labs, [the account dashboard](#) will show the tests running with a correct name.

Part 3: Test Status

There's still one more thing we'll need to handle, and that's setting the status of the Sauce Labs job after it completes.

Right now regardless of the outcome of a test, the job in Sauce Labs will register as `Finished`. Ideally we want to know if the job was a `Pass` or a `Fail`. That way we can tell at a glance if a test failed or not. And with a couple of tweaks we can make this happen easily enough.

First we need to leverage another helper method in pytest.

```
# filename: tests/conftest.py
# ...
@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    outcome = yield
    result = outcome.get_result()
    setattr(item, "result_" + result.when, result)
```

With `pytest_runtest_makereport` we're able to grab the outcome of a test as it's running and append the result to a node in the request object that we're using in our fixture. There are three phases that are reported on -- `setup`, `call`, and `teardown`. `call` is the one we care about since it's the test run. So the method that we'll use in the request object is `request.node.result_call`.

Now let's update our fixture to use it.

```
# filename: conftest.py
# ...
    def quit():
        try:
            if config.host == "saucelabs":
                if request.node.result_call.failed:
                    driver_.execute_script("sauce:job-result=failed")
                    print "http://saucelabs.com/beta/tests/" + driver_.session_id
                elif request.node.result_call.passed:
                    driver_.execute_script("sauce:job-result=passed")
        finally:
            driver_.quit()
# ...
```

We first check to see if our tests are running against Sauce Labs. If so we check to see what the test status was (e.g., `request.node.result_call.failed` or `request.node.result_call.passed`) and execute a snippet of JavaScript to communicate to Sauce what the test outcome was. If the test failed, we also output the URL of the job to the console. We wrap all of this in a `try` block so that if there is an issue `driver_.quit()` will still get executed.

Now when we run our tests in Sauce Labs and navigate to [the Sauce Labs Account dashboard](), we will see our tests running like before. But now there will be a proper test status when they finish (e.g., `Pass` or `Fail`) and we'll see the URL for the job in the console output as well. This enables us to easily jump to the specific job in Sauce Labs.

Part 4: Sauce Connect

There are various ways that companies make their pre-production application available for testing. Some use an obscure public URL and protect it with some form of authentication (e.g.,

Basic Auth, or certificate based authentication). Others keep it behind their firewall. For those that stay behind a firewall, Sauce Labs has you covered.

They have a program called [Sauce Connect](#) that creates a secure tunnel between your machine and their private cloud. With it you can run tests in Sauce Labs and test applications that are only available on your private network.

To use Sauce Connect you need to download and run it. There's a copy for each operating system -- get yours [here](#) and run it from the command-line. In the context of our existing test code let's download Sauce Connect, unzip it's contents, and store it in our `vendor` directory.

```
pages
    __init__.py
    base_page.py
    dynamic_loading_page.py
    login_page.py
tests
    __init__.py
    config.py
    conftest.py
    dynamic_loading_test.py
    login_test.py
vendor
    chromedriver
    geckodriver
    sc-4.3.16-osx
        bin
            sc
            sc.dSYM
                Contents
                    Info.plist
                    Resources
                        DWARF
                            sc
        include
            sauceconnect.h
        lib
            libsauceconnect.a
            libsauceconnect.la
        license.html
```

Now we just need to launch the application while specifying our Sauce account credentials.

```
vendor/sc-4.3.16-osx/bin/sc -u $SAUCE_USERNAME -k $SAUCE_ACCESS_KEY
Sauce Connect 4.3.16, build 2399 c7e5fec
*** WARNING: open file limit 7168 is too low!
*** Sauce Labs recommends setting it to at least 8000.
Starting up; pid 58426
Command line arguments: vendor/sc-4.3.16-osx/bin/sc -u the-internet -k ****
Log file: /tmp/sc.log
Pid file: /tmp/sc_client.pid
Timezone: EDT GMT offset: -4h
Using no proxy for connecting to Sauce Labs REST API.
Resolving saucelabs.com to 162.222.75.243 took 68 ms.
Started scproxy on port 49310.
Please wait for 'you may start your tests' to start your tests.
Starting secure remote tunnel VM...
Secure remote tunnel VM provisioned.
Tunnel ID: 21ff9664b06c4edaa4bd573cdc1fbac1
Secure remote tunnel VM is now: booting
Secure remote tunnel VM is now: running
Using no proxy for connecting to tunnel VM.
Resolving tunnel hostname to 162.222.76.147 took 55ms.
Starting Selenium listener...
Establishing secure TLS connection to tunnel...
Selenium listener started on port 4445.
Sauce Connect is up, you may start your tests.
```

Now that the tunnel is established, we could run our tests against a local instance of our application (e.g., the-internet). Assuming the application was set up and running on our local machine, we run our tests against it by specifying a different base URL at runtime (e.g., `py.test --baseurl=http://localhost:4567`) and they would work.

To see the status of the tunnel, we can view it on the tunnel page of the account dashboard. To shut the tunnel down, we can do it manually from this page. Or we can issue a `Ctrl+C` command to the terminal window where it's running.

When the tunnel is closing, here's what you'll see.

```
Got signal 2
Cleaning up.
Removing tunnel 21ff9664b06c4edaa4bd573cdc1fbac1.
All jobs using tunnel have finished.
Waiting for the connection to terminate...
Connection closed (8).
Goodbye.
```

Chapter 14
Speeding Up Your Test Runs

We've made huge strides by leveraging page objects, a base page object, explicit waits, and connecting our tests to Sauce Labs. But we're not done yet. Our tests still take a good deal of time to run since they're executing in series (e.g., one after another). As our suite grows this slowness will grow with it.

With parallelization we can easily remedy this pain before it becomes acute by executing multiple tests at the same time. And with pytest there's a plugin that makes this turnkey to setup.

Configuration

With [pytest-xdist](#) we get parallel execution. First we need to install it.

```
pip install pytest-xdist
```

Now our tests will be able to run in parallel. We just need to specify the number of processes we want to use which we can do with the `-n` runtime flag.

```
py.test -n 5
```

Alternatively we can specify `-n auto` if we want pytest to detect the number of processors on the machine and spin up an according number of processes.

NOTE: If you're using Sauce Labs be sure to see what your concurrency limit is (e.g., number of available concurrent virtual machines). It's listed on the My Account page in the [Account Dashboard](#). This number will be the limiter to how many parallel tests you can run at once. The recommendation is to set the number of processes for your test runs to equal the concurrency limit. So if you have 3 VMs then set your test runs to `-n 3`.

Random Order Execution

When enabling parallel execution in your tests you may start to see odd, inconsistent behavior that is hard to track down.

This is often due to dependencies between tests that you didn't know were there. A great way to expose these kinds of issues and ensure your tests are ready for prime time is to execute them in a random order. This also has the added benefit of exercising the application you're testing in a random order (which could unearth previously unnoticed bugs).

With the [pytest-randomly plugin](#) this is turnkey to get going.

```
pip install pytest-randomly
```

Now when we run our tests they will run in a random order.

NOTE: Each time a test runs a seed number will be presented in the console output. If we want to rerun the tests using the exact same order to try and reproduce a failure we can provide the seed number using the `--randomly-seed=` flag. And if want to disable random order execution then we specify `-p no:randomly` at run-time.

Chapter 15
Flexible Test Execution

In order to get the most out of our tests we'll want a way to break them up into relevant, targeted chunks. Running tests in smaller groupings like this (along with parallel execution) will help keep test run times to a minimum and help enhance the amount of feedback you get in a timely fashion.

With [pytest markers](#) we're able to easily achieve test grouping.

Let's step how to set this up.

Specifying Markers

In order to use custom markers we first have to specify them a `pytest.ini` file. So let's go ahead and create that in the root of our project.

```
  pages
      __init__.py
      base_page.py
      dynamic_loading_page.py
      login_page.py
  pytest.ini
  tests
      __init__.py
      config.py
      conftest.py
      dynamic_loading_test.py
      login_test.py
  vendor
       chromedriver
       geckodriver
       sc-4.3.16-osx
           bin
               sc
               sc.dSYM
                   Contents
                       Info.plist
                       Resources
                           DWARF
                               sc
           include
               sauceconnect.h
           lib
               libsauceconnect.a
               libsauceconnect.la
           license.html
```

In `pytest.ini` file we'll add two no markers and a description for them.

```
# filename: pytest.ini
[pytest]
markers =
    shallow: Run the most critical tests. Quick to run and exercise the top layer of
functionality that matters most to the business
    deep: Run non-critical tests which may take longer to run
```

`shallow` tests are roughly equivalent to "smoke" or "sanity" tests. These should pass before you can consider running other tests which aren't as mission critical and may take longer to run (e.g., `deep` tests).

Now we just need to add decorators for the markers to the tests. They can either be applied at the

class level or at the test method level. If they're applied to the class, then all test methods in the class will receive it.

```python
# filename: tests/login_test.py
# ...
    @pytest.mark.shallow
    def test_valid_credentials(self, login):
        login.with_("tomsmith", "SuperSecretPassword!")
        assert login.success_message_present()

    @pytest.mark.deep
    def test_invalid_credentials(self, login):
        login.with_("tomsmith", "bad password")
        assert login.failure_message_present()

# ...
```

In `login_test.py` we marked the happy path test as `shallow` and the invalid credentials test as `deep`. Now let's apply the `deep` marker to the entire class in `dynamic_loading_test.py`.

```python
# filename: tests/dynamic_loading_test.py
# ...
@pytest.mark.deep
class TestDynamicLoading():
# ...
```

Markers are powerful since they can be applied across different test files, enabling you to create a dynamic grouping of tests at runtime.

Running Markers

With pytest we can specify which marker to launch at runtime. This is handled as another runtime flag on the command-line using `-m`.

Here are the available execution commands given for our current markers:

```
py.test -m shallow
py.test -m deep
```

For more info on this functionality and other available options, check out the [pytest marker documentation](#).

Chapter 16
Automating Your Test Runs

You'll probably get a lot of mileage out of your test suite in its current form if you just run things from your computer, look at the results, and tell people when there are issues. But that only helps you solve part of the problem.

The real goal in test automation is to find issues reliably, quickly, and automatically. We've built things to be reliable and quick. Now we need to make them run on their own, and ideally, in sync with the development workflow you are a part of.

To do that we need to use a Continuous Integration server.

A Continuous Integration Server Primer

A Continuous Integration server (a.k.a. CI) is responsible for merging code that is actively being developed into a central place (e.g., "trunk", "head", or "master") frequently (e.g., several times a day, or on every code commit) to find issues early so they can be addressed quickly — all for the sake of releasing working software in a timely fashion.

With CI we can automate our test runs so they can happen as part of the development workflow. The lion's share of tests that are typically run on a CI Server are unit (and potentially integration) tests. But we can very easily add in our Selenium tests too.

There are numerous CI Servers available for use today, most notably:

- Bamboo
- Jenkins
- Solano Labs
- TravisCI

Let's pick one and step through an example.

A CI Example

Jenkins is a fully functional, widely adopted, open-source CI server. Its a great candidate for us to try.

Lets start by setting it up on the same machine as our test code. Keep in mind that this isn't the "proper" way to go about this — its merely beneficial for this example. To do it right, the Jenkins server (e.g., master node) would live on a machine of its own.

Part 1: Quick Setup
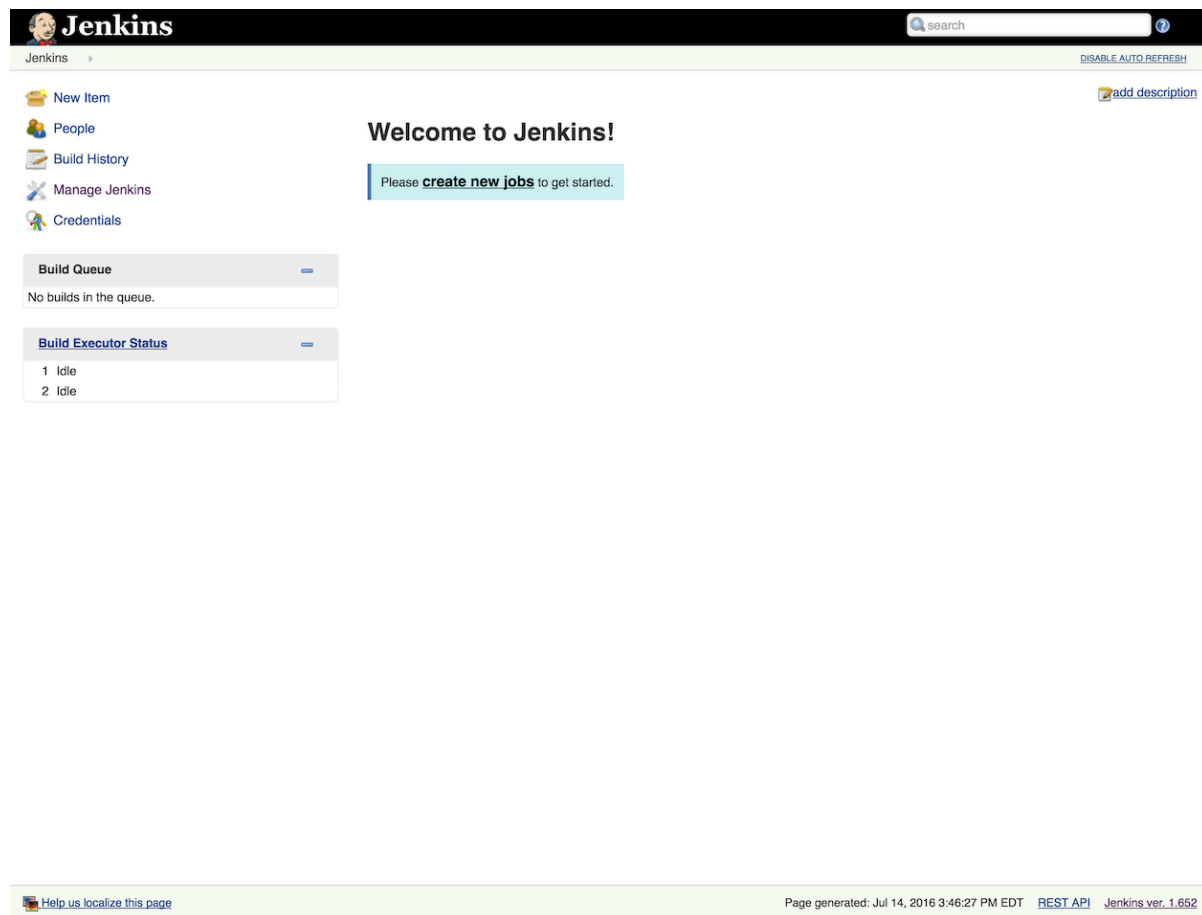
A simple way to get started is to grab the latest Jenkins war file. You can grab it from the [Jenkins homepage](#), or from [the direct download link on the homepage](#).

Once downloaded, launch it from the command-line.

```
> java -jar jenkins.war
// ...
hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

You will now be able to use Jenkins by visiting http://localhost:8080/ in your browser.

NOTE: Before moving to the next step, click `ENABLE AUTO-REFRESH` at the top right-hand side of the page. Otherwise you'll need to manually refresh the page (e.g., when running a job and waiting for results to appear).

Part 2: Job Creation And Configuration

Now that Jenkins is loaded in the browser, let's create a Job and configure it to run our `shallow` tests against Chrome on Windows 10.

- Click `New Item` from the top-left of the Dashboard

- Give it a name (e.g., `Shallow Tests Chrome Windows 10`)
- Select the `Freestyle project` option
- Click `OK`

This will load a configuration screen for the Jenkins job.

- In the `Advanced Project Options` section select the `Advanced` button
- Choose the checkbox for `Use custom workspace`
- Provide the full path to your test code
- Leave the `Display Name` field blank

NOTE: Ideally, your test code would live in a version control system and you would configure your job (under `Source Code Management`) to pull it in and run it. To use this approach you may need to install a plugin to handle it. For more info on plugins in Jenkins, go here.

- Scroll down to the `Build` section and select `Add build step`
- Select `Execute shell`
- Specify the commands needed to launch the tests

```
py.test -n 5 -m shallow --browser=chrome --browserversion=50 --platform="Windows 10"
--junitxml=result.xml
```

All of the runtime flags should look familiar except for `--junitxml`. We want to have our test run output into a standard format our CI server can consume. JUnit XML is the defacto standard format and with pytest it's available as an output when you specify a runtime flag and a filename (e.g., `result.xml`).

Now let's configure the job to consume the test results.

- Under `Post-build Actions` select `Add post build action`
- Select `Publish JUnit test result report`
- Add the name of the result file specified in the command -- `result.xml`
- Click `Save`

Discard Old Builds ❓

This build is parameterized ❓

Disable Build (No new builds will be executed until the project is re-enabled.) ❓

Execute concurrent builds if necessary ❓

Advanced Project Options

Advanced...

Source Code Management

⦿ None
○ CVS
○ CVS Projectset
○ Subversion

Build Triggers

☐ Build after other projects are built ❓

☐ Build periodically ❓

☐ Poll SCM ❓

Build

⦀ **Execute shell** ❓

Command `py.test -n 5 -m shallow --browser=chrome --browserversion=50 --platform="Windows 10" --junitxml=result.xml`

| Aggregate downstream test results |
| Archive the artifacts |
| Build other projects |
| **Publish JUnit test result report** |
| Publish Javadoc |
| Record fingerprints of files to track usage |
| E-mail Notification |

ables

Delete

Add post-build action ▾

Save   Apply

○ CVS Projectset
○ Subversion

Build Triggers

☐ Build after other projects are built ❓

☐ Build periodically ❓

☐ Poll SCM ❓

Build

⦀ **Execute shell** ❓

Command
```
py.test -n 5 -m shallow --browser=chrome --browserversion=50 --platform="Windows 10" --junitxml=result.xml
```

See the list of available environment variables

Delete

Add build step ▾

Post-build Actions

⦀ **Publish JUnit test result report** ❓

Test report XMLs   `result.xml`

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is the workspace root.

☐ Retain long standard output/error ❓

Health report amplification factor   `1.0` ❓

1% failing tests scores as 99% health. 5% failing tests scores as 95% health

Advanced...

Delete

Add post-build action ▾

Save   Apply

Now our tests are ready to be run, but before we do, let's go ahead and add a failing test so we can demonstrate the test report.

Part 3: Force A Failure

Let's add a new test method to `login_test.py` that will fail every time we run it.

```python
# filename: tests/login_test.py
# ...
    @pytest.mark.shallow
    def test_forced_failure(self, login):
        login.with_("tomsmith", "bad password")
        assert login.success_message_present()
```

This test mimics our `BadPasswordProvided` test by visiting the login page and providing invalid credentials. The difference is in the assertion. It will fail since a success message won't be present after attempting to login with bogus credentials.

One more thing we'll want to do is update how we're outputting the Sauce Labs job URL when there's a test failure. Right now we're outputting it to the console, but with the pytest XML report generation this information will get lost when in our Jenkins job. So let's make sure it shows up in the stack trace.

```python
# filename: tests/conftest.py
# ...
    def quit():
        try:
            if config.host == "saucelabs":
                if request.node.result_call.failed:
                    driver_.execute_script("sauce:job-result=failed")
                    raise AssertionError("http://saucelabs.com/beta/tests/" + driver_.
session_id)
# ...
```

In the `quit()` method of fixture we make it so we throw an exception with the Sauce Labs job URL when there's a test failure.

Now let's run our Jenkins job by clicking `Build Now` from the left-hand side of the screen.

NOTE: You can peer behind the scenes of a job while it's running (and after it completes) by clicking on the build you want from `Build History` and selecting `Console Output`. This output will be your best bet in tracking down an unexpected result.

When the test completes, it will be marked as failed.

When we click on `Latest Test Result` we can see the test that failed (e.g., `Tests.LoginTest.ForcedFailure`).

Jenkins

🔍 search ❓

Back to Project

🔍 Status

Changes

Console Output

Edit Build Information

History

Test Result

Test Result

1 failures

2 tests
Took 22 sec.
📝 add description

All Failed Tests

Test Name	Duration	Age
➕ tests.login_test.TestLogin.test_forced_failure ▾	10 sec	1

All Tests

Package	Duration	Fail	(diff)	Skip	(diff)	Pass	(diff)	Total	(diff)
tests.login_test	22 sec	1	+1	0		1	+1	2	+2

And if we click on the failed test, we can see the failure message along with a URL to the job in Sauce Labs.

Jenkins

Back to Project
Status
Changes
Console Output
Edit Build Information
History
Test Result

Failed

tests.login_test.TestLogin.test_forced_failure (from pytest)

Failing for the past 1 build (Since ● #1)
Took 10 sec.
add description

Error Message

```
test setup failure
```

Stacktrace

```
def quit():
        try:
            if config.host == "saucelabs":
                if request.node.result_call.failed:
                    driver_.execute_script("sauce:job-result=failed")
>                   raise AssertionError("http://saucelabs.com/beta/tests/" + driver_.session_id)
E                   AssertionError: http://saucelabs.com/beta/tests/2ffb08a91c904af2bad3752d6ea3b64a

tests/conftest.py:74: AssertionError
```

When we follow the URL to the Sauce Labs job we're able to see what happened during the test run (e.g., we can replay a video of the test, see what Selenium commands were issued, etc.).

Notifications

In order to maximize your CI effectiveness, you'll want to send out notifications to alert your team members when there's a failure.

There are numerous ways to go about this (e.g., e-mail, chat, text, co-located visual cues, etc). And thankfully there are numerous, freely available plugins that can help facilitate whichever method you want. You can find out more about Jenkins' plugins here.

For instance, if you wanted to use chat notifications and you use a service like HipChat or Slack, you would do a plugin search and find the following plugins:

After installing the plugin for your chat service, you will need to provide the necessary information to configure it (e.g., an authorization token, the channel/chat room where you want notifications to go, what kinds of notifications you want sent, etc.) and then add it as a `Post-build Action` to your job (or jobs).

After installing and configuring a plugin, when your CI job runs and fails, a notification will be sent to the chat room you configured.

Ideal Workflow

In the last chapter we covered test grouping with categories and applied some preliminary ones to our tests (e.g., "Shallow" and "Deep"). These categories are perfect for setting up an initial acceptance test automation workflow.

To start the workflow we'll want to identify a triggering event. Something like a CI job for unit or integration tests that the developers on your team use. Whenever that runs and passes, we can trigger our "Shallow" test job to run (e.g., our smoke or sanity tests). If the job passes then we can trigger a job for "Deep" tests to run. Assuming that passes, we can consider the code ready to be promoted to the next phase of release (e.g., manual testing, push to a staging, etc.) and send out a relevant notification to the team.

NOTE: You may need to incorporate a code deployment action as a preliminary step before your "Shallow" and "Deep" jobs can be run. Consult a developer on your team for help if that's the case.

Outro

By using a CI Server you're able to put your tests to work by using computers for what they're good at -- automation. This frees you up to focus on more important things. But keep in mind that there are numerous ways to configure your CI server. Be sure to tune it to what works best for you and your team. It's well worth the effort.

Chapter 17
Finding Information On Your Own

There is information all around us when it comes to Selenium. But it it can be challenging to sift through it, or know where to look.

Here is a list breaking down a majority of the Selenium resources available, and what they're useful for.

Documentation & Tips

- [Selenium HQ](#)

This is the official Selenium project documentation site. It's a bit dated, but there is loads of helpful information here. You just have to get the hang of how the navigate site to find what you need.

- [The Selenium Wiki](#)

This is where all the good stuff is -- mainly, documentation about the various language bindings and browser drivers. If you're not already familiar with it, take a look.

- [Elemental Selenium Archives](#)

Every tip I've written is freely available on the tips archive page. There are over 70 different Selenium problems and solutions covered.

Blogs

- [The official Selenium blog](#)

This is where news of the Selenium project gets announced, and there's also the occasional round-up of what's going on in the tech space (as it relates to testing). Definitely worth a look.

- [A list of "all" Selenium WebDriver blogs](#)

At some point, someone rounded up a large list of blogs from Selenium practitioners and committers. It's a pretty good list.

Other Books

- [Selenium 2 Testing Tools](#)

This book is by [David Burns](#) and it is a solid resource. It outlines how to use Selenium, top-to-bottom. It's definitely worth having on your shelf.

- [Selenium Testing Tools Cookbook](#)

This is another good book outlining some great ways to leverage Selenium. While I haven't had a chance to finish reading it, it's clear that Gundecha has a very pragmatic approach that will yield great results.

- [Selenium Design Patterns and Best Practices](#)

Dima Kovalenko's book covers useful tactics and strategies for successful test automation with Selenium. I was a technical reviewer for the book and think it's a tremendous resource. The book covers Ruby, but he has ported the examples to Java. You can find them [here](#).

Meetups

- [The Selenium Hangout](#)

This is an entirely online meetup that is run by me and [David Burns](#) where we talk about the latest in the Selenium community (e.g., upcoming conferences, the status of Selenium 3, the W3C spec, etc.) and answer questions from the community. Videos are recorded and made available (along with notes) on [the official Selenium blog](#).

- [All Selenium Meetups listed on Meetup.com](#)

A listing of all in-person Selenium Meetups are available on Meetup.com. If you're near a major city, odds are there's one waiting for you.

- [How to start your own Selenium Meetup](#)

If there's not a Selenium Meetup near you, start one! Sauce Labs has a great write up on how to do it.

Conferences

- [Selenium Conf](#)

This is the official annual conference of the Selenium project where practitioners and committers gather and share their latest knowledge and experiences with testing. The conference location changes every year (e.g., it's been in San Francisco, London, Boston, Bangalore, and soon -- Portland).

- [Selenium Camp](#)

This is an annual Selenium conference in Eastern Europe (in Kiev, Ukraine) organized by the folks

at [XP Injection](#). It's a terrific conference. If you can make the trip, I highly recommend it.

- [List of other testing conferences](#)

A helpful website that lists all of the testing conferences out there.

Videos

- [Selenium Conference Talks](#)

All of the talks from The Selenium Conference are recorded and made freely available online. This is a tremendous resource.

- [Selenium Meetup Talks](#)

Some of the Selenium Meetups make it a point to record their talks and publish them afterwards. Here are some of them. They are a great way to see what other people are doing and pick up some new tips.

- [Selenium Hangout](#)

All of the Selenium Hangout Meetups are recorded and made available. A lot of great stuff is discussed in them.

Mailing Lists

- [Selenium Developers List](#)

This is where developers discuss changes to the Selenium project.

- [Selenium Users Google Group](#)
- [Selenium LinkedIn Users Group](#) The signal to noise ratio in these groups can be challenging at times. But you can occasionally find some answers to your questions.

Forums

- [Stack Overflow](#)
- [Quora](#)
- [Reddit](#)

These are the usual forums where you can go looking for answers to questions you're facing (in addition to the mailing lists above).

Issues

- [Selenium Issue Tracker](#)

If you're running into a specific and repeatable issue that just doesn't make sense, you may have found a bug in Selenium. You'll want to check the Selenium Issue Tracker to see if it has already been reported. If not, then create a new issue. But be sure to read [this post](#) before you do (so you can be sure that you have provided enough information).

Chatting With the Selenium Community on IRC

The Selenium IRC Chat Channel is arguably the best way to connect with the Selenium community and get questions answered. This is where committers and practitioners hang out day-in and day-out.

Brief Intro To IRC

IRC (short for Internet Relay Chat) is a protocol that freely enables live chatting (both in groups and person to person) and file sharing. It's been around for a while (circa 1988) and is the preferred mode of communication among certain tech circles.

Within the realm of IRC there are numerous networks you can connect to. Each one containing people and bots logged in and joined to one or more chat rooms talking, sharing files, etc.

One of the beautiful things about IRC is that there is no registration required to join the party. You just need to download a client that handles the IRC protocol (there's at least one available for every platform), point it at a network, and pick a unique nickname for yourself on that network.

Once you're on you can join a chat room and start jib-jabbing.

How To Get Connected

Step 1: Get An IRC Chat Client

First thing's first, get a chat client that supports IRC.

You may already have one and not even know. For example, [Adium](#) (for OSX) supports a staggering number of chat protocols. If you already have it (or something like it) then use that to connect. If you don't, then you'll need to download one that supports it (or one that is built specifically for IRC).

Here are some worthwhile IRC chat clients (broken out by platform -- and are free unless otherwise noted).

OSX

- [LimeChat](#)
- [Textual](#) ($5.99 to buy)

Windows

- [mIRC](#) (free for 30 days, $20 for a single-user license)

Linux

- [Irssi](#)

Web-only

- [Webchat](#)
- [IRCCloud](#) (free with paid tier as well, Android and iOS apps available as well)

Step 2: Connect To The Proper Server

The Selenium chat channel lives on the Freenode network. To connect to it directly you would use `irc.freenode.net`.

Before connecting you should be able to set a nickname and perhaps even specify which channel you would like to connect to after connecting (e.g., `#selenium`). If you don't see these bits, don't sweat it. Connect and proceed to the next step.

NOTE: If you're using Webchat, it will automatically connect you to Freenode.

Step 3: Join The Chat Channel

In IRC parlance channels are prepended with a `#` and are lower-case. So the Selenium channel is `#selenium`.

If you were able to configure your chat client to join the channel for you, then proceed to the next step. Otherwise, you'll need to issue a command in the status window. There are a series of commands you can issue in IRC. They are all prepended with a `/`. To join a chat channel type `/join #selenium` and press `Enter`.

This will open a new chat window to the Selenium chat channel. Woohoo!

Step 4: Talk And Hangout

Feel free to say hello and introduce yourself. But more importantly, ask your question. If it looks like no one is chatting, ask it anyway. Someone will see it and eventually respond. They always do.

In order to get your answer, you'll probably need to hang around for a bit. But the benefit of being a fly on the wall is that you gain insight into other problems people face, possible solutions, and the current state of the Selenium project and its various pieces.

Chapter 18
Now You Are Ready

The journey for doing Selenium successfully can be long and arduous. But by adhering to the principals in this book, you will avoid a majority of the pitfalls around you. You're also in a better position now -- armed with all of the information necessary to continue your Selenium journey.

You are ready. Keep going, and best of luck!

If you have any questions, feedback, or want help -- get in touch!

Cheers,
Dave H