The power of end-to-end testing

Selenium in .NET for Testers and developers

Adrian Sroka

The power of end-to-end testing

Selenium in .NET for Testers and developers

Adrian Sroka

This book is for sale at http://leanpub.com/end2end-testing-net

This version was published on 2018-01-12

ISBN 978-83-946579-0-1

Tweet This Book!

Please help Adrian Sroka by spreading the word about this book on Twitter!

The suggested tweet for this book is:

@suvrocDev

The suggested hashtag for this book is #seleniumpower.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#seleniumpower

I dedicate this book to my lovely wife Monica. With special acknowledgments for support and help during the creation process. Your faith in me gives me the strength to change the world.

Contents

1. Introduction

In this book I want to show you the power of the end-to-end testing of web applications. The framework which I want to describe here has a huge amount of features, but at the same time it is easy to use and understand. Moreover, the techniques of designing and architecting test cases described here can help us to maintain and use that kind of tests in an easy way.

This book is focused on the Selenium end-to-end tests using .NET environment and C# language. However, the Selenium tool can also be used in JavaScript, Java, PHP, Python and Ruby.

The source code for all the examples presented in this book is available on GitHub[1].

[1] https://github.com/suvroc/SeleniomWebDriver-examples

2. End-to-end testing

What is it and why is it important?

To begin with, we should explain what end-to-end testing actually is and present its role in a testing space. We begin with showing all types of tests, starting from the most basic. Tests should be an important part of Continuous Integration (CI) process. If you don't know what it is, I recommend you to read a short description of this idea. Each type of test has its particular place in the CI process. To help you imagine this concept of testing better I will compare them to car building process.

User manual tests

It is the most important part of the whole testing process. But it takes the greatest amount of time and worse still, it takes the time of the testing team. Every time we want to test something in a manual way, somebody has to sit down and do it step by step. The same applies to the car example. Manual testing is when someone drives a new car through the track and tests how it works.

Automated tests

It is a group of all types of test that we can write and execute many times without effort.

Unit tests

They are fast and test only a single code unit. To separate each tested functionality, we should create only a tested class with its dependencies. A test library should mock these dependencies. All testing data should be placed in mocks. Unit tests shouldn't rely on any data existing in a database, because data like this are not constant so the test result wouldn't be repeatable. This type of tests shouldn't use a database at all. Usually, they should test a single method in isolation from others. All the tests within one project should run no more than 5 minutes together. Thereby, the Continuous Integration tool can execute them after each build.

Unit tests should be the most commonly used in your development. Let's take a look at the analogy to unit tests in car testing industry. They are like testing each wheel and the steering wheel separately (e.g. checking if they are round).

Integration tests

Integration tests can cover a functionality with all its dependencies. Their primary purpose is to test interactions between different parts of the system. Usually, integration tests can access a database, so the running time should be about 5-15 minutes for all the tests together. It might be necessary to separate them from fast unit tests in a CI process, because of the execution time. Of course they are still vulnerable to data changes, but the data set can be recreated before each test to keep test independence. In a well-tested system, integration tests should cover a functionality already tested by unit tests. We don't need to execute them after each build. It is still important to start it as often as necessary i.e. every two hours. In the car construction, integration testing is like software testing. If we want to test whether a steering wheel moves the wheels, we need to check all the components together: the steering wheel, the steering shaft, the steering arms and the wheels. If moving the steering wheel makes the wheel move too, the test is passed.

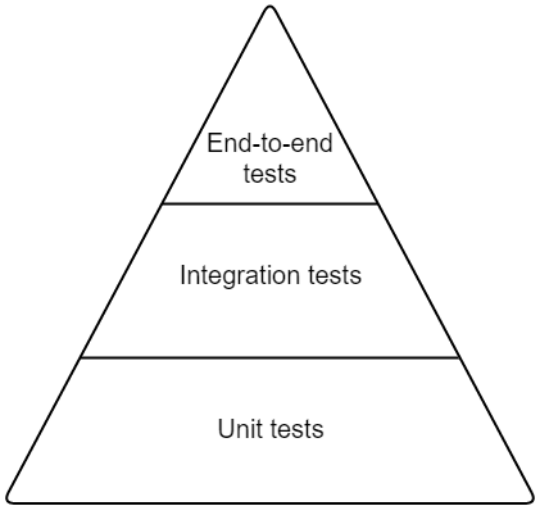
End-to-end tests

End-to-end tests are a kind of acceptance tests and should involve all system functionalities. They operate on the user interface (UI), system logic and database. It should also reflect user acceptance criteria. It is common that executing these tests takes a long time. That's why the Continuous Integration tool shouldn't execute them too often. Once a day is a great compromise, but they should be run before each deployment. We can compare them to the car factory. It is like testing the ability to drive a vehicle on an engine test stand (a tool to characterize engine work). We test the complete functionality of car, and we can automate it.

Non-functional tests (i.e. performance)

They test a particular aspect of our system and take different time to complete, usually quite long. They are particulary important when we need to focus on a specific aspect of the system during the development (i.e. real-time systems).

The number of manual tests should decrease with the complexity of tests. It is important because complex tests can be much more problematic. The test pyramid visualizes it well.



Testing Pyramid

Our priority during the testing process, is to reduce the human effort with no negative influence on test quality: coverage and completeness. This approach can save our time and increase system reliability. End-to-end testing can help to achieve it.

3. Location of UI elements

> What method is the best for you?

The next essential step in our journey is selecting elements in HTML structure. On each test case, we want to operate on the web page. Before we can do this, we should have an object to work on. Using WebDriver, we can choose specific elements. As we can see in the previous examples, there is a method `FindElement`. It is the most often used method in each testing process.

If you know JavaScript or CSS, you have probably heard about selecting HTML elements. In CSS there is a selector telling us which elements should be affected by the style rule.

```
.header > img {
        width: 100px;
}
```

JavaScript has a similar mechanism to get objects related to HTML elements. We can use that element using a simple object wrapper. It even lets us change the HTML structure.

```
element = document.querySelector(selectors);
```

Selenium WebDriver has the same functionality as these two languages. To perform most of the operations, we will need a .NET representation of page fragments. All HTML elements are objects implementing `IWebElement` interface. Using the `Driver.FindElement()` method, we can find any element in many different ways. This method searches the page for elements matching a given selector and returns the first of them. The element has to be visible but you do not need to have scrolled to this element before. Selenium tries to scroll to the considered element before returning it. We still should be careful while deciding which method to choose. It has significant impact on the future maintenance. We should choose the most appropriate one based on our knowledge about of the possible future changes in code. Let us look at all the possibilities. In the examples below, the first line is an HTML and the second one is a C# command to locate it.

- by ID

```
<div id="uniqueElementId">...</div>

driver.FindElement(By.Id("uniqueElementId"));
```

- by class name

```
<div class="city">
        <span>Warsaw</span>
</div>

driver.FindElement(By.ClassName("city"));
```

- by name

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

driver.FindElement(By.Name("email"));
```

- by css selectors[1]

```
<div id="cities">
        <span class="city">Krakow</span>
        <span class="city capital">Warsaw</span>
</div>

driver.FindElement(By.CssSelector("#cities span.city.capital"));
```

- by tag name

```
<header>...</header>

driver.FindElement(By.TagName("header"));
```

- by XPath[2]

[1] https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors
[2] https://developer.mozilla.org/pl/docs/XPath

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

driver.FindElement(By.XPath("//input"));
```

- by link text

```
<a href="http://www.google.com/search?q=warsaw">Warsaw</a>

driver.FindElement(By.LinkText("Warsaw"));
```

- by partial link text

```
<a href="http://www.google.com/search?q=warsaw">Warsaw city</a>

driver.FindElement(By.PartialLinkText("Warsaw"));
```

- by JavaScript code

```
var element = (IWebElement) ((IJavaScriptExecutor)driver)
        .ExecuteScript("return $('.city')[0]");
```

To be sure that the test will still pass after HTML code changes, we need to make the wise decision. The selector should be specific enough to select the specific element, but it should not be too sensitive for change in the HTML structure.

The most suitable candidates are ids and classes. They are the best choice, but only if they exist in HTML structure. While testing our application, we can add classes and ids to the most important elements on web pages. I recommend this approach because it makes our tests simpler. If we cannot change the application structure, we should try to locate elements using CSS selectors. We should be still pretty careful, while building them. Try to avoid making them too complicated and hard to understand. Remember that you should be able to change them in the future.

XPath selectors are a very powerful method, but at the same time the most complicated one. That is why we should use it only as a last resort. Especially because XPath selectors are much slower than others. Moreover, it is not recommended to use any text based location. Texts on our website can often change, especially if our application supports many languages.

There are some cases when we want to test the correctness of a text. For example, to check if changing the language functionality works fine. Even in this case, it will be better to choose more specific selectors (id or class) than to rely on text values. When we find that element, we can check its `Text` property to find, what it contains.

Below there is a list of selectors in the order of recommendation. The most convenient are on the top and the bottom, there are ones that we rather shouldn't use.

- id
- class name
- name
- CSS selector
- tag name
- XPath
- link text
- partial link text
- JavaScript code

Selecting embedded elements

We can select elements in many ways. We can also search for them in different scopes, in the whole page structure (like in the previous chapter) or inside other elements. The method `FindElement` exists also for a single `IWebElement`.

```
1  [Test]
2  public void ShouldSelectHousesList()
3  {
4          var driver = new ChromeDriver();
5          driver.Navigate()
6                  .GoToUrl("https://commons.wikimedia.org/wiki/Category:Houses");
7          var houseTile = driver.FindElement(By.ClassName("gallerybox"));
8          var link = houseTile.FindElement(By.CssSelector(".gallerytext > a"));
9          link.Click();
10         driver.Quit();
11 }
```

In line 8 we can see that we are trying to find an element inside another element which we have already found. It is a useful method. Especially if we want to iterate through list items and take some action on each element from this list item.

Working with lists

That leads us straight to selecting many elements. The previous paragraph shows that the `FindElement` method can select one element. To select many elements, we should use a similar method.

```
public ReadOnlyCollection<IWebElement> FindElements(By by);
```

It means, that if we have the following HTML we can select these elements as follows.

```
<div id="cities">
        <span class="city">Krakow</span>
        <span class="city capital">Warsaw</span>
</div>

IEnumerable<IWebElement> cities = driver.FindElements(By.ClassName("city"));
```

This way we would gain a collection of elements matching a selector that we have used (class name in this example).

Sometimes, we may want to create a test case, within which we have a list on the page. We may want to iterate through its items to take some actions. We can click the link inside a list item or check, if any element contains proper text. To achieve this, we can make a loop and work with a list of elements. We can also create a new structure to increase readability and understanding of the code. In our case, we assume, that we will have the following structure on the page.

```
<ul class="gallery">
        <li class="gallerybox">
                <div class="gallerytext">
                        <a href="...">Image 1 name</a>
                        <a class="thumb"><img src="..."/></a>
                </div>
        </li>
        <li class="gallerybox">
                <div class="gallerytext">
                        <a href="...">Image 2 name</a>
                        <a class="thumb"><img src="..."/></a>
                </div>
        </li>
        <li class="gallerybox">
                <div class="gallerytext">
                        <a href="...">Image 3 name</a>
                        <a class="thumb"><img src="..."/></a>
                </div>
        </li>
        ...
</ul>
```

Additionally, let us assume that image name is the identifier of a picture in a gallery. So we use it to select a gallery item, that we want to search for. We will need the following structure.

```
1  public class ListItem
2  {
3          public ListItem(string name, IWebElement element)
4          {
5                  this.Name = name;
6                  this.Element = element;
7          }
8
9          public string Name { get; set; }
10         public IWebElement Element { get; set; }
11 }
```

This class will store an identifier of the list item and element assigned to it. Now we can select `ListItems` instead of `IWebElement`.

```
1  [Test]
2  public void ShouldSelectHousesListItems()
3  {
4          var driver = new ChromeDriver();
5
6          driver.Navigate()
7                  .GoToUrl("https://commons.wikimedia.org/wiki/Category:Houses");
8
9          var houseTiles = driver.FindElements(By.ClassName("gallerybox"))
10                 .Select(x => new ListItem(
11                         x.FindElement(By.CssSelector(".gallerytext > a")).Text, x));
12
13         var houseTile = houseTiles.First();
14
15         var link = houseTile.Element
16                 .FindElement(By.ClassName("thumb"));
17
18         link.Click();
19
20         driver.Quit();
21 }
```

The most important part of this example happens in line 9. We select a list of gallery items. Then we convert them to `ListItem` type using handy LINQ Select method[3]. Inside we create objects of a final type and also select a name for each gallery item.

[3] https://msdn.microsoft.com/en-gb/library/bb548891(v=vs.110).aspx

The variable `houseTiles` is an `IEnumerable<ListItem>` type. Thereby we can use Linq methods to select and filter objects. Our new class gives us a useful interface to filter list items in the next line.