

SELENIUM WEBDRIVER COM TESTNG EM JAVA

GUIA COMPLETO PARA
AUTOMAÇÃO DE TESTES WEB



Índice

Índice	2
Introdução	4
Capítulo 1 - Pré-Requisitos	5
Capítulo 2 - Configuração	6
2.1 Instalando Java	6
2.1.1 Via Download da Oracle	6
2.1.2 Com Brew via Terminal	8
2.2 Instalando IntelliJ	9
2.2.1 Via Download da JetBrains	9
2.2.2 Com Brew via Terminal	11
2.3 Instalando o Chrome Browser	11
2.4 Instalando o Chrome Driver	12
2.5 Criando um Novo Projeto no IntelliJ	13
Capítulo 3 - WebDriver	16
3.1 Adicionando e Instanciando o WebDriver	19
3.2 Acessando a aplicação a ser testada	21
Capítulo 4 - Encontrando WebElements	25
Capítulo 5 - Interagindo com WebElements	31
5.1 Criando a classe de framework HomePage	32
5.2 Criando a classe de framework LoginPage	36
5.3 Criando a classe de framework SecureAreaPage	40
Capítulo 6 - Criando um Teste	44
6.1 Criando a classe de teste LoginTests	44
Capítulo 7 - Executando o Teste	51
7.1 Via IDE	51
7.2 Via Terminal	52
Capítulo 8 - Interagindo com elementos Dropdown	55
8.1 Criando a classe de framework DropdownPage	55
8.2 Criando a classe de teste DropdownTests	62
Capítulo 9 - Interações avançadas	65
9.1 Hovers	65
Capítulo 10 - Digitando teclas adicionais	75
Capítulo 11 - Alerts, Uploads e Modals	81
11.1 Alerts	81

11.1.1 Executando vários testes em uma classe	88
11.2 Upload	89
11.3 Modals	93
Capítulo 12 - Frames	95
Capítulo 13 - Estratégias de Wait	101
13.1 Esperas Implícitas	101
13.2 Esperas Explícitas	102
13.3 Esperas Fluentes	110
Capítulo 14 - Usando JavaScript	111
Capítulo 15 - Navigation	115
Capítulo 16 - Capturando Screenshots	119
Capítulo 17 - Event Listeners	124
Capítulo 18 - Extras sobre Webdriver	127
18.1 ChromeOptions	127
18.2 Usando Cookies	130

Introdução

O **Selenium WebDriver** é uma **API** de automação orientada a objetos que controla nativamente um navegador como um usuário faria. Ele suporta várias linguagens de programação e, neste curso, focaremos na implementação **Java**.

Neste curso, vamos aprender:

- Como **instalar o Selenium WebDriver** e as outras dependências necessárias
- Como **usar a API WebDriver** para iniciar e interagir com sites
- Várias técnicas para **encontrar elementos** como botões, links, texto, listas suspensas e muito mais **em um aplicativo da web**
- Chamadas de **API do WebDriver** para **interagir com esses elementos**, como clicar, inserir e ler texto, selecionar opções, manipular pop-ups e alertas, fazer upload de arquivos, trabalhar com iframes e muito mais
- **Interações avançadas**, como passar o mouse e enviar teclas alternativas como tab, bem como símbolos
- Como **integrar a API WebDriver** com uma biblioteca de declaração de teste, como **TestNG**
- Como organizar nosso código de teste usando o **Page Object Model**
- Várias **estratégias de espera** para controlar o tempo de nossos scripts e evitar testes com **flakiness** e baixa performance
- Como **tirar screenshots** durante as execuções de teste
- Como fazer com que nosso código ouça eventos específicos do **WebDriver** e execute ações adicionais, como registrar nossa atividade de teste
- Como **personalizar o navegador** usado na execução do teste
- Como **executar testes sem tela (headless)**
- Tópicos avançados, como **navegar em várias guias abertas** em um navegador, bem como **gerenciamento de cookies**
- E para qualquer coisa que não possamos fazer com os principais métodos da **API do WebDriver**, aprenderemos como escrever **ações personalizadas usando JavaScript**

Capítulo 1 - Pré-Requisitos

Se você está acessando este Guia Completo para Automação de Testes Web podemos presumir algumas possíveis hipóteses:

- Está iniciando seus estudos sobre Automação de Testes
OU
- Já possui alguma experiência com Automação de Testes e agora quer aprender uma nova solução técnica ou aprofundar seus conhecimentos sobre ela

Seja qual destes for o seu caso (ou algum outro caso não mencionado aqui), aqui você vai ter acesso à uma jornada passo-a-passo através de um conjunto de funcionalidades e exemplos práticos detalhados de seu uso, o que te permitirá aplicar este conhecimento em seus projetos pessoais de estudo, portfólio profissional, ou em seus projetos reais junto ao seus clientes ou empregadores.

Para isso, utilizaremos uma das linguagens de programação mais difundidas e mais utilizadas no mercado de desenvolvimento de software ao longo de muito tempo e ainda muito relevante e importante: **Java**.

Sabemos que existem muitos outros frameworks e ferramentas para automação que requerem muito menos código escrito, abstraindo uma série de métodos já disponíveis de forma nativa ou através de bibliotecas integradas. Porém, aprender a usar uma linguagem de programação mais poderosa como o **Java** e construir as classes e métodos que precisamos nos proporciona um aprendizado mais sólido sobre a relação que nosso projeto de automação tem com o ambiente de teste e a aplicação que será testada.

Neste material não vamos abordar conceitos básicos sobre lógica de programação, variáveis, tipos de dados, operadores, etc. Para que tenha um melhor aproveitamento deste conteúdo, recomendamos que possua conhecimentos básicos sobre estes conceitos, mas nada lhe impede de seguir os passos e instruções de cada capítulo e, como consequência, aprender um bocado ou identificar dúvidas que podem ser facilmente sanadas com pesquisas e estudos paralelos.

Caso identifique erros de edição deste material ou a ausência de explicações ou detalhamentos mais aprofundados sobre algum tópico, fique à vontade para entrar em contato conosco através dos canais disponíveis em <http://qway.tech>.

Capítulo 2 - Configuração

Para começar a escrever scripts usando *Java* e *Selenium WebDriver*, precisaremos baixar/installar o seguinte:



Java

Para este curso, usaremos a versão 10 ou superior da linguagem de programação *Java*. De qualquer forma, evitamos utilizar recursos do *Java* que não estejam disponíveis em versões anteriores.



IntelliJ

Também vamos usar o Editor Inteligente para escrever nosso código. Você pode usar um editor diferente, se quiser, mas todos os exemplos deste curso serão feitos no *IntelliJ*.



Executável *Chrome Driver*

O *Selenium WebDriver* pode ser executado em todos os principais navegadores, mas neste curso, usaremos o *Chrome*. Então, precisamos do executável *Selenium Chromedriver*.



Chrome Browser

Também precisaremos do próprio navegador *Chrome*.

Obs.: Para a criação deste material estou usando um *MacBook Pro* com processador *M1* da *Apple* rodando *macOS Ventura 13.1*.

2.1 Instalando Java

Para instalar o *Java Development Kit* no *macOS*, você pode optar por fazer via download do site da *Oracle* ou através do gerenciador de pacotes *Brew*.

2.1.1 Via Download da Oracle

Estamos no site da *Oracle* e é aqui que podemos baixar o *Java Development Kit*:

<https://www.oracle.com/technetwork/java/javase/downloads/index.html>

Precisamos do *Java 10* ou superior para este curso. Se você já tem esta versão ou superior, não precisa fazer nada. Se você não tiver nenhum desses, então vamos em frente.

Há alguns links para download do *Development Kit* para *Linux*, *macOS* e *Windows*. Vamos usar a versão 19 para *macOS* disponível em Janeiro/23. Clicamos no nome do arquivo **.dmg**, que inicia o download da imagem de disco para *macOS*:

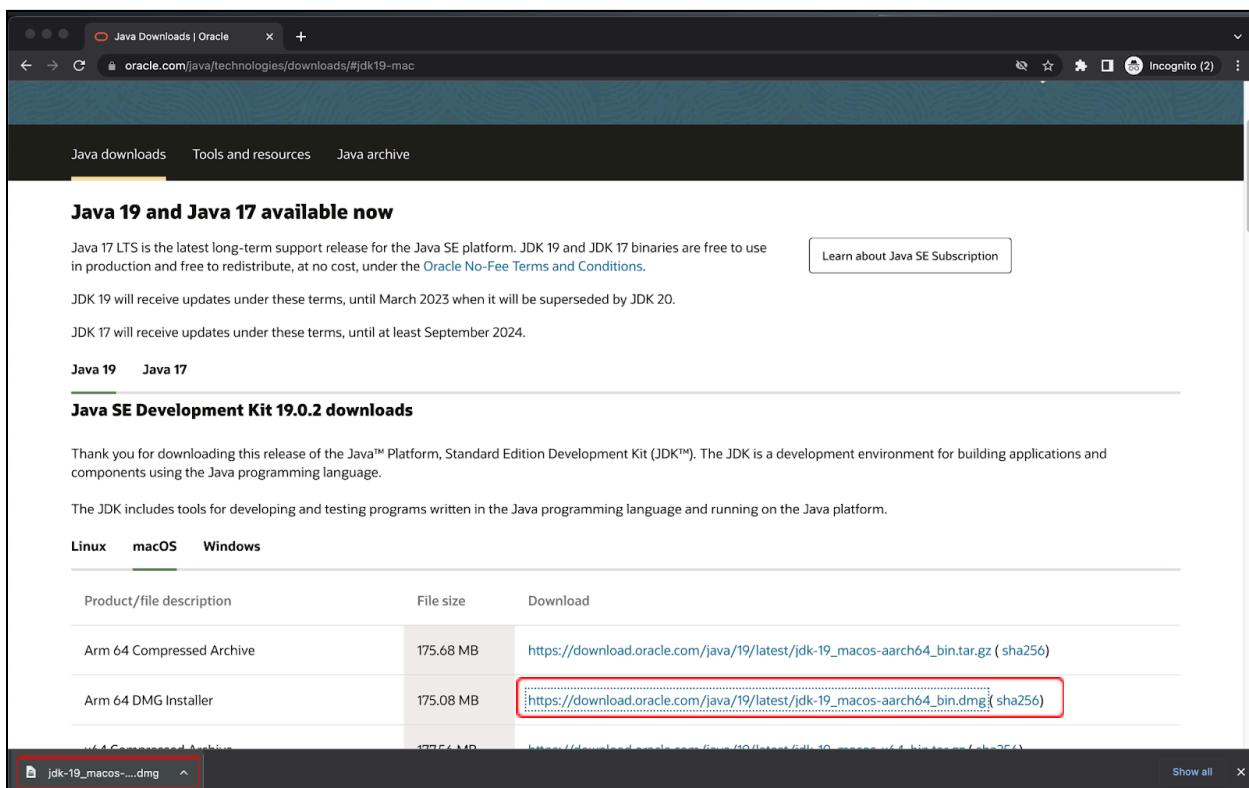


Imagen 1: Download do Java Development Kit no site da Oracle

Após o download, clique no arquivo baixado e uma unidade de disco virtual será criada com a imagem baixada. Dê duplo clique no arquivo **.pkg** e siga o processo de instalação:

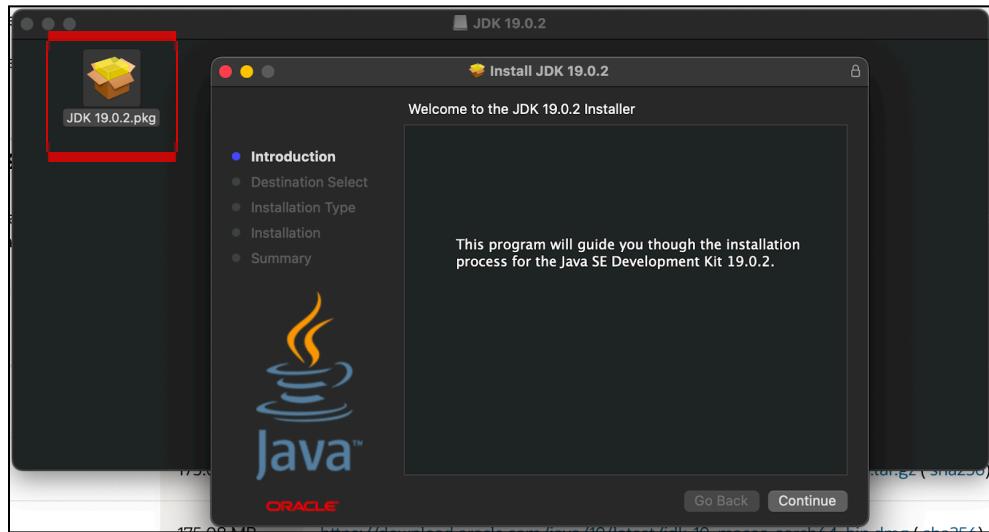


Imagen 2: Executando o instalador do Java Development Kit

Obs.: Para a instalação do JDK em ambiente Windows o instalador deve se comportar de forma similar, com diferenças pontuais em passos como Destination Select.

2.1.2 Com Brew via Terminal

Com o comando abaixo podemos fazer a instalação via Brew no Terminal:

```
[daniel.castro@DanielCastroMBP ~ % brew install openjdk
==> Fetching openjdk
==> Downloading https://ghcr.io/v2/homebrew/core/openjdk/manifests/19.0.2
Already downloaded: /Users/daniel.castro/Library/Caches/Homebrew/downloads/f9e77
637697de0efc73871d6762bbc2219d687cd401686d74f271abbc24beac6--openjdk-19.0.2.bott
le_manifest.json
==> Downloading https://ghcr.io/v2/homebrew/core/openjdk/blobs/sha256:c576dc5ee0
Already downloaded: /Users/daniel.castro/Library/Caches/Homebrew/downloads/4282b
beed9bb944fe4297b99935d05770e99b7356747cffeb29cae7bc19a7cc7--openjdk--19.0.2.arm
64_ventura.bottle.tar.gz
==> Pouring openjdk--19.0.2.arm64_ventura.bottle.tar.gz
==> Caveats
For the system Java wrappers to find this JDK, symlink it with
  sudo ln -sfn /opt/homebrew/opt/openjdk/libexec/openjdk.jdk /Library/Java/JavaV
irtualMachines/openjdk.jdk

openjdk is keg-only, which means it was not symlinked into /opt/homebrew,
because macOS provides similar software and installing this software in
parallel can cause all kinds of trouble.

If you need to have openjdk first in your PATH, run:
  echo 'export PATH="/opt/homebrew/opt/openjdk/bin:$PATH"' >> ~/.zshrc

For compilers to find openjdk you may need to set:
  export CPPFLAGS="-I/opt/homebrew/opt/openjdk/include"

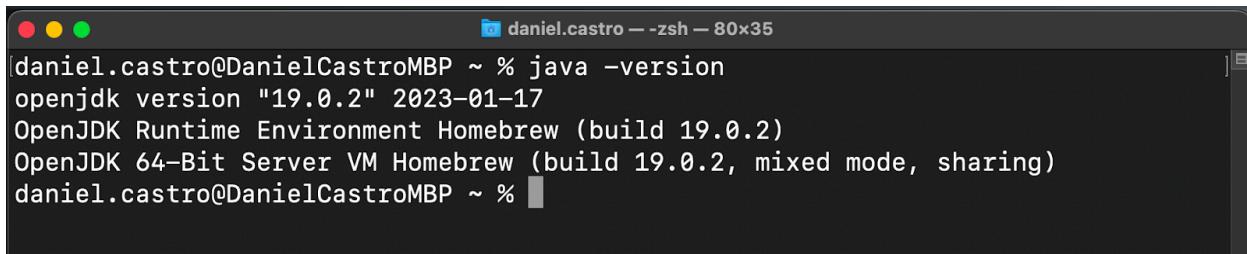
==> Summary
🍺 /opt/homebrew/Cellar/openjdk/19.0.2: 637 files, 320.0MB
==> Running `brew cleanup openjdk`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
daniel.castro@DanielCastroMBP ~ % ]
```

Imagen 3: Instalando o OpenJDK via Terminal com Brew

Após executar a instalação com *Brew* via *Terminal* precisamos criar um link simbólico para que o sistema identifique o *JDK* instalado:

```
sudo ln -sfn /opt/homebrew/opt/openjdk/libexec/openjdk.jdk  
/Library/Java/JavaVirtualMachines/openjdk.jdk
```

Para verificar se o *Java* foi instalado corretamente, execute o seguinte comando:



```
daniel.castro@DanielCastroMBP ~ % java -version  
openjdk version "19.0.2" 2023-01-17  
OpenJDK Runtime Environment Homebrew (build 19.0.2)  
OpenJDK 64-Bit Server VM Homebrew (build 19.0.2, mixed mode, sharing)  
daniel.castro@DanielCastroMBP ~ %
```

Imagen 4: Verificando a instalação do JDK

Obs.:

- Existem *Shell Scripts* para a instalação do *JDK* em ambiente *Windows* utilizando gerenciadores de pacotes similares ao *Brew*, como o *Chocolatey*.
- O comando “*java -version*” também pode ser usado para confirmar a instalação do *JDK* em ambiente *Windows*

2.2 Instalando IntelliJ

A próxima coisa que faremos é instalar o *IntelliJ*, que será o editor no qual escreveremos nosso código.

2.2.1 Via Download da JetBrains

Existem vários sistemas operacionais suportados (*Windows*, *macOS* e *Linux*), então escolha o sistema operacional de sua escolha. Neste caso, farei o download para *macOS*.

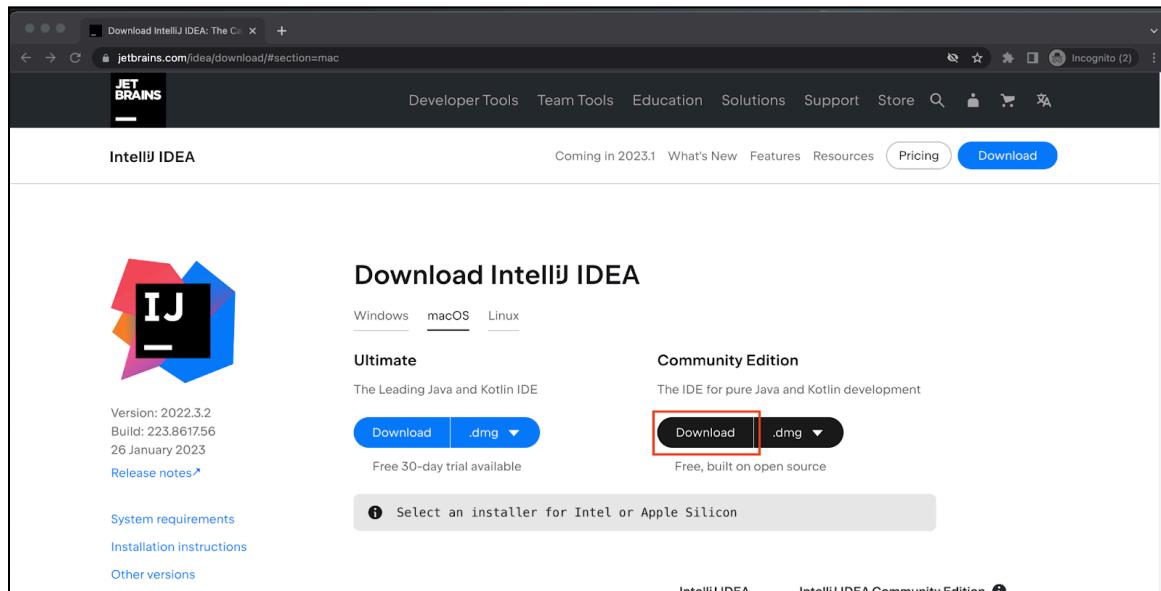


Imagen 5: Download do IntelliJ no site da JetBrains

Há 2 opções aqui: **Ultimate** e **Community**.

A versão *Ultimate* é voltada para desenvolvimento empresarial. Há um teste gratuito, mas depois você precisará comprá-la.

Você pode usar a edição *Community*, que é o que eu uso. Esta é gratuita e de código aberto. Basta clicar em “**Download**”.

Após o download, execute o arquivo **.dmg** e em seguida execute a instalação arrastando o app para a pasta Applications:

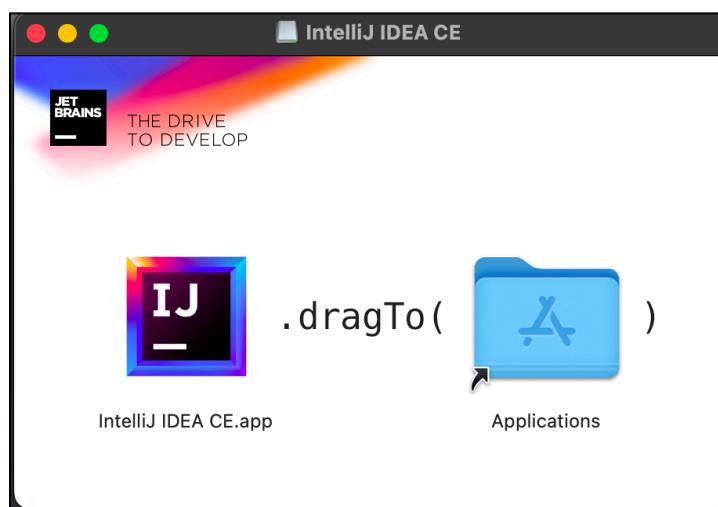
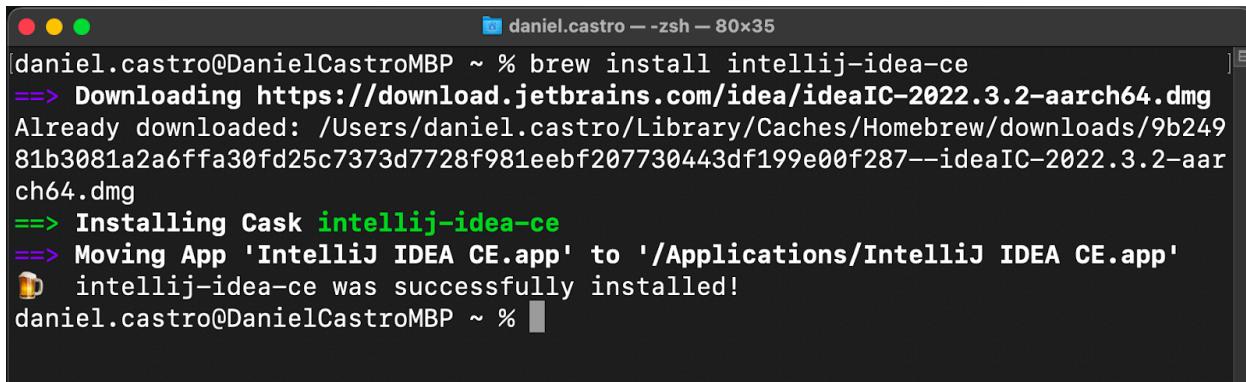


Imagen 6: Instalando o IntelliJ via instalador

2.2.2 Com Brew via Terminal

Alternativamente você pode instalar *IntelliJ* via *Brew* no *Terminal*:



```
daniel.castro@DanielCastroMBP ~ % brew install intellij-idea-ce
==> Downloading https://download.jetbrains.com/idea/ideaIC-2022.3.2-aarch64.dmg
Already downloaded: /Users/daniel.castro/Library/Caches/Homebrew/downloads/9b24981b3081a2a6ffa30fd25c7373d7728f981eebf207730443df199e00f287--ideaIC-2022.3.2-aarch64.dmg
==> Installing Cask intellij-idea-ce
==> Moving App 'IntelliJ IDEA CE.app' to '/Applications/IntelliJ IDEA CE.app'
🍺 intellij-idea-ce was successfully installed!
daniel.castro@DanielCastroMBP ~ %
```

Imagen 7: Instalando o IntelliJ via Terminal com Brew

2.3 Instalando o Chrome Browser

Se você já tem o *Chrome* instalado, não se preocupe com essa parte. Caso contrário, acesse <https://www.google.com/chrome> e clique no botão de “**Download**”:

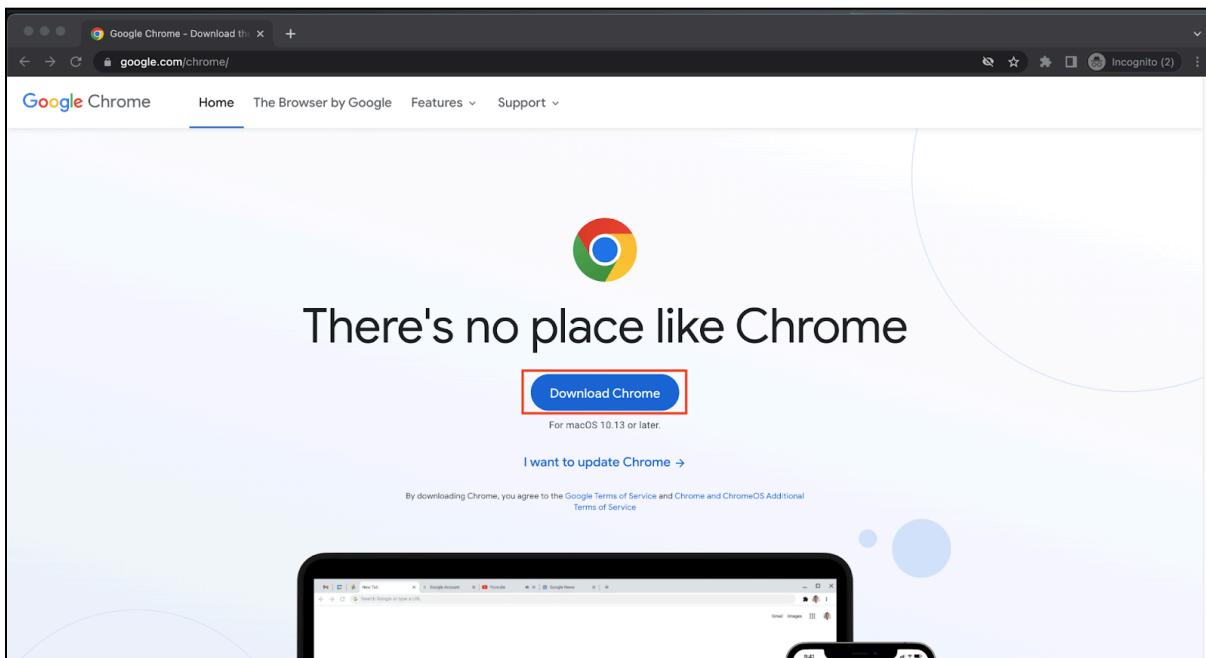


Imagen 8: Download do Google Chrome

Depois que o arquivo for totalmente baixado, basta executar e seguir as instruções para concluir a instalação.

2.4 Instalando o Chrome Driver

Por fim, precisamos baixar o *ChromeDriver*. Este é o *WebDriver* para *Chrome*, o navegador que usaremos neste curso.

A versão do *ChromeDriver* deve corresponder à versão do *Chrome* que você esteja usando.

Para verificar sua versão do *Chrome*, clique no menu disponível no canto superior direito do navegador e depois em “**Ajuda**” >> “**Sobre o Google Chrome**”. Lá podemos ver qual versão do *Chrome* você está usando.

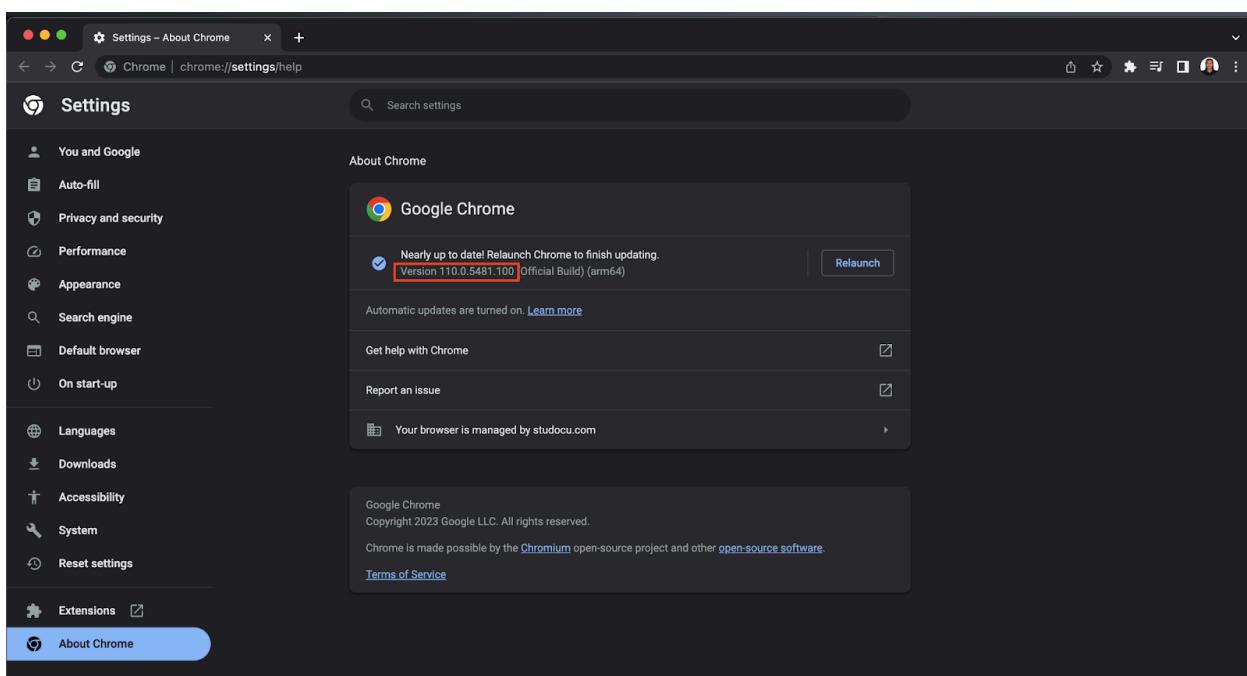


Imagen 9: Verificando a versão do Google Chrome

Estou usando a versão 110 do *Google Chrome*, então vou precisar obter o *ChromeDriver* clicando no link correspondente à esta versão:

<https://chromedriver.chromium.org/downloads>

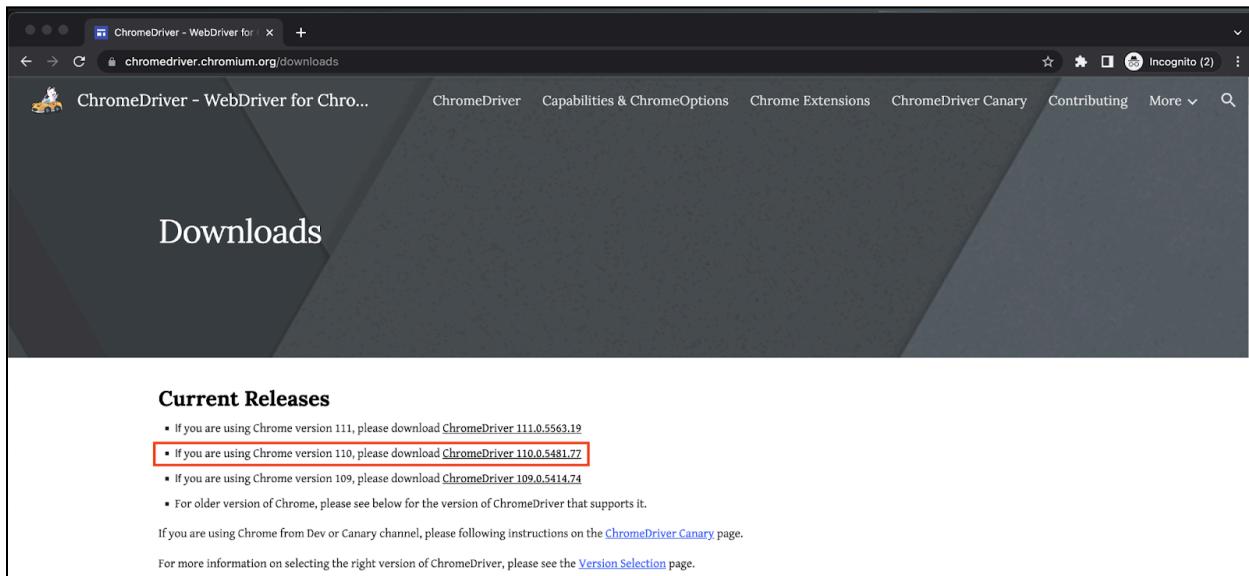


Imagen 10: Download do Google Chrome

Na página seguinte, clicamos no link correspondente ao sistema operacional que estamos utilizando. No meu caso, fiz o download para *macOS* com processador *arm64*, correspondente ao processador *M1* da *Apple*:

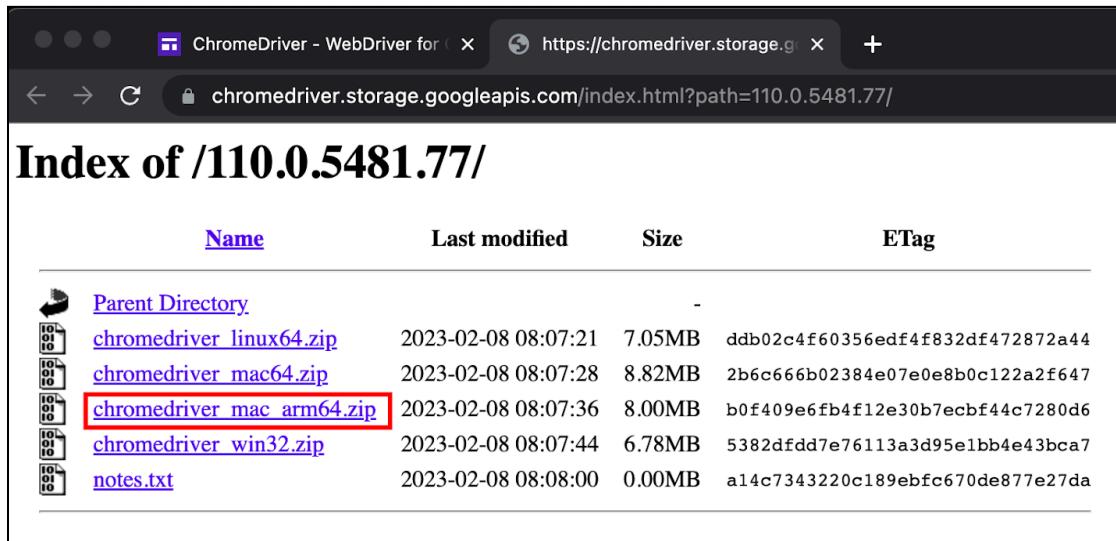


Imagen 11: Download do Chrome Driver para macOS com processador arm64

2.5 Criando um Novo Projeto no IntelliJ

Após instalar o *IntelliJ*, vamos criar um novo projeto clicando em “**New Project**” na tela “**Welcome to IntelliJ IDEA**”:

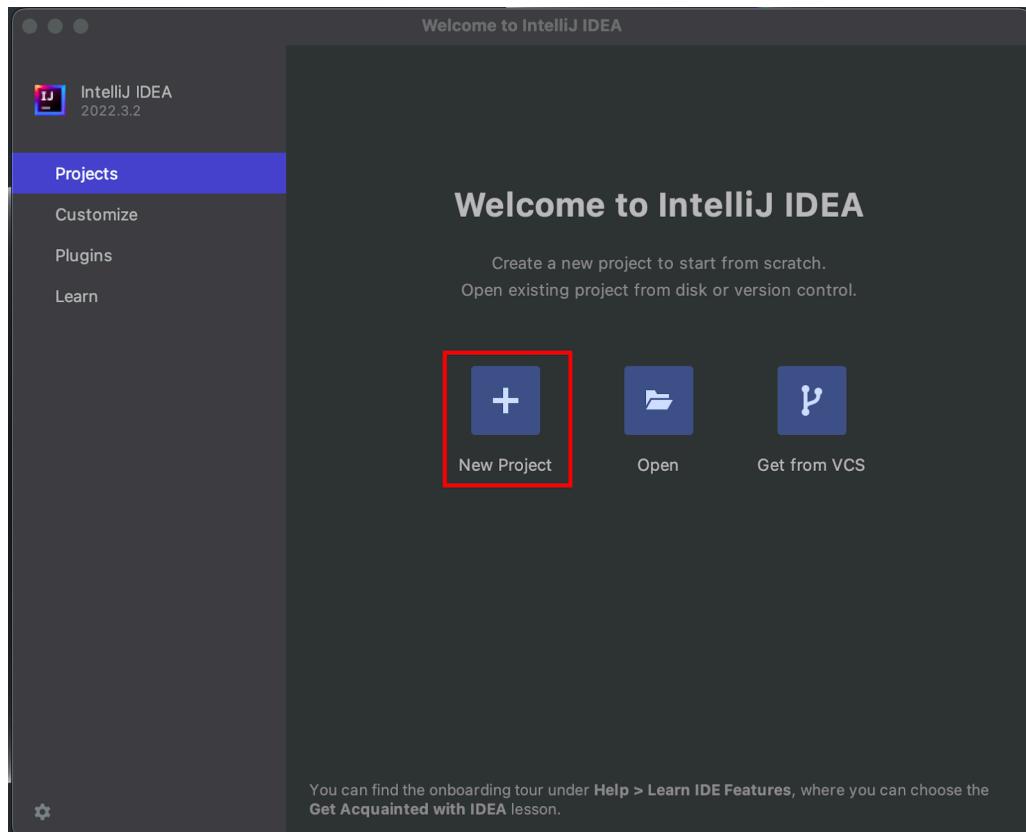


Imagen 12: Welcome screen do IntelliJ

Serão apresentadas algumas opções aqui no menu à esquerda da tela “**New Project**”. Selecione “**New Project**”:

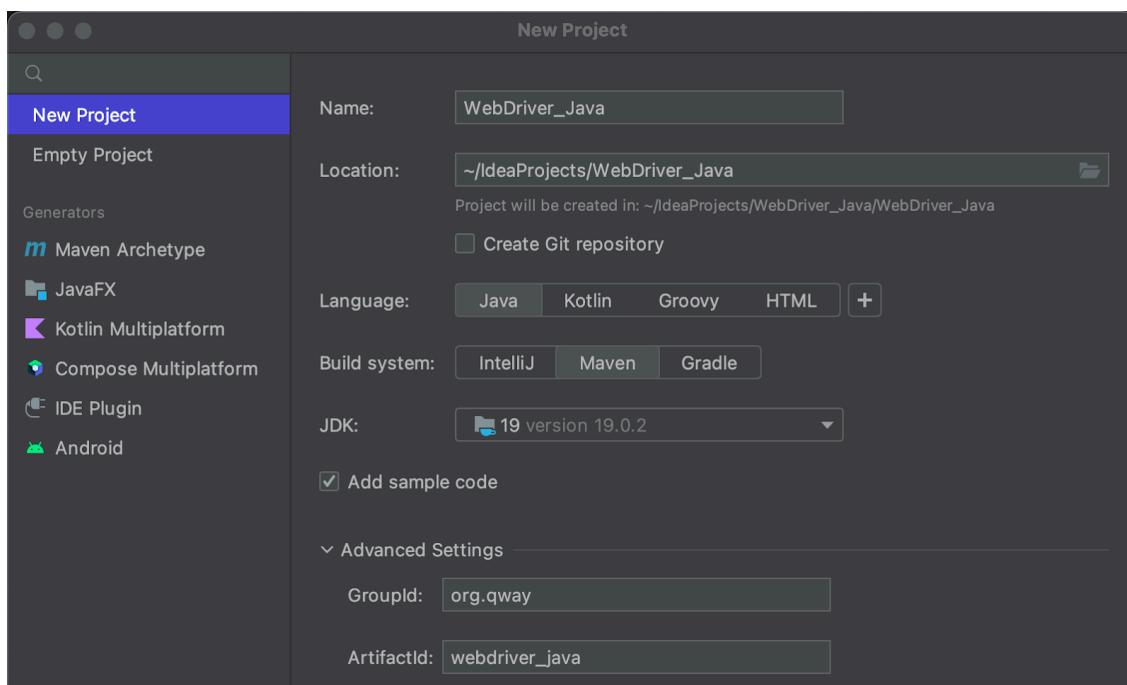
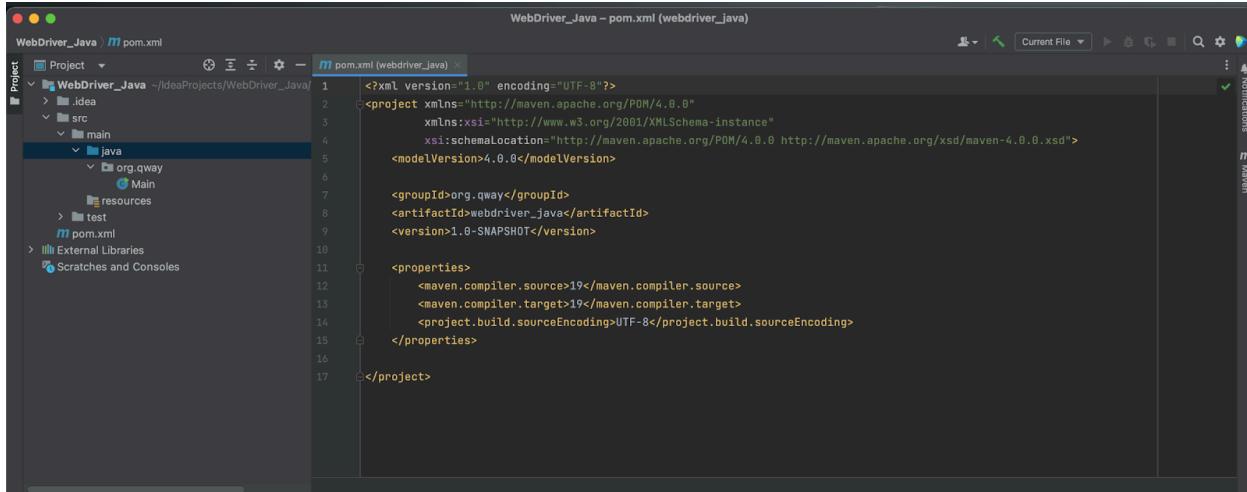


Imagen 13: Criando um novo projeto no IntelliJ

Selecione os campos conforme a imagem acima e clique em “**Create**”.

Em seguida veremos que nosso projeto será aberto apresentando o arquivo **pom.xml**. Neste arquivo adicionaremos as dependências do projeto:



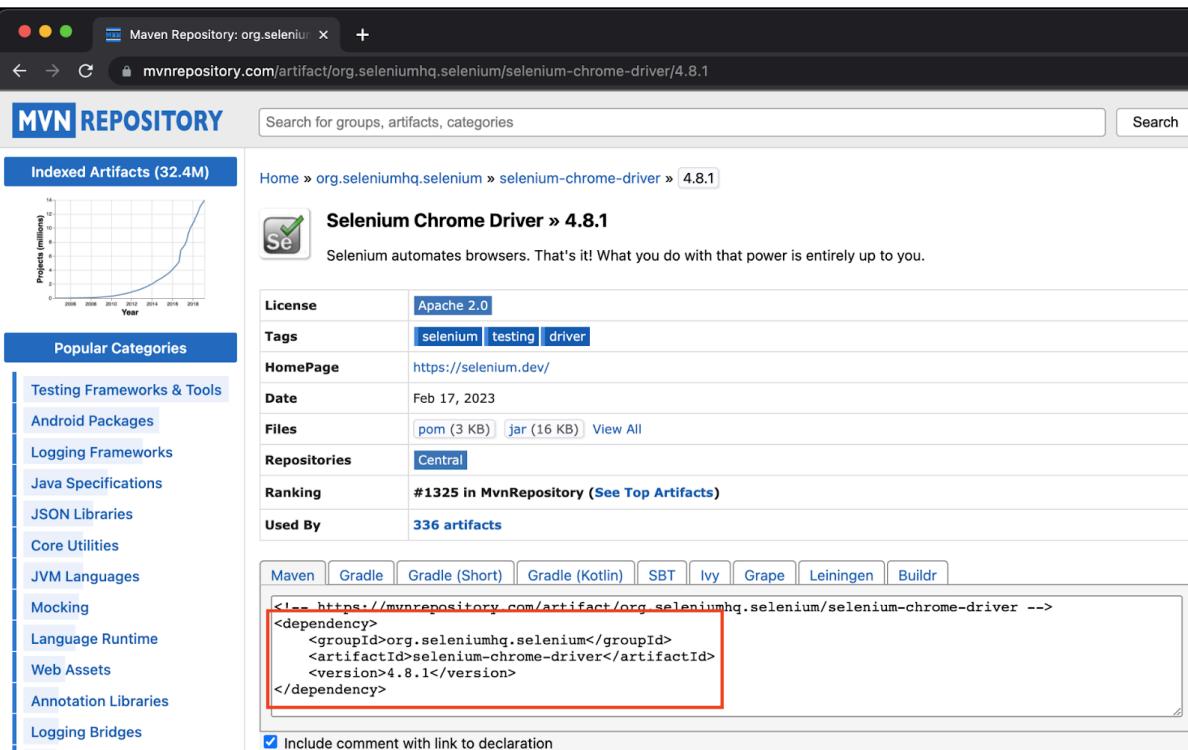
```
<?xml version="1.0" encoding="UTF-8 ?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.qway</groupId>
    <artifactId>webdriver_java</artifactId>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <maven.compiler.source>19</maven.compiler.source>
        <maven.compiler.target>19</maven.compiler.target>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
</project>
```

Imagen 14: Arquivo pom.xml do novo projeto

Capítulo 3 - WebDriver

Há uma dependência que precisamos agora. Estamos usando o *ChromeDriver*, então precisamos da respectiva dependência do *Maven*. Vamos até o repositório para copiar este *snippet*:

<https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-chrome-driver>



```
<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-chrome-driver -->
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-chrome-driver</artifactId>
    <version>4.8.1</version>
</dependency>
```

Imagen 15: Snippet de código de dependência do Selenium Chrome Driver para Maven

Vamos criar uma seção de *dependencies* e colar o *snippet* copiado entre as tags *<dependencies>* e *</dependencies>*.

No canto superior direito clique na view *Maven* e verifique se a dependência do *selenium-chrome-driver* está lá. Caso não esteja, clique no botão de “*Reload*”.

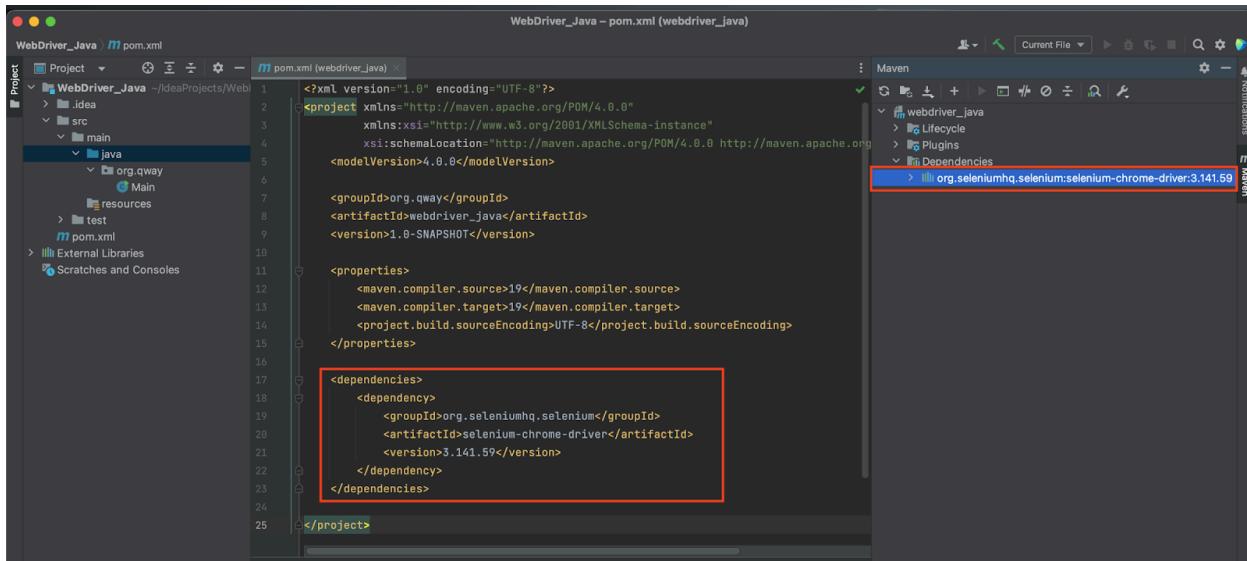


Imagen 16: Dependência do Selenium Chrome Driver adicionada ao arquivo pom.xml

Há apenas mais uma coisa que precisamos fazer para concluir esta configuração inicial. Lembra que baixamos o executável do *ChromeDriver*? Agora colocaremos ele em nosso projeto.

Na view do projeto, posicionada à esquerda da tela, clique com o botão direito no nome do projeto e em seguida escolha a opção “**New**” > “**New Directory**” no menu de contexto apresentado. Vamos chamar este diretório de “**resources**”, com letras minúsculas, e clicar em “**OK**”. Isso criará esse diretório para você. Em seguida, vamos mover o executável do *ChromeDriver* para este diretório.

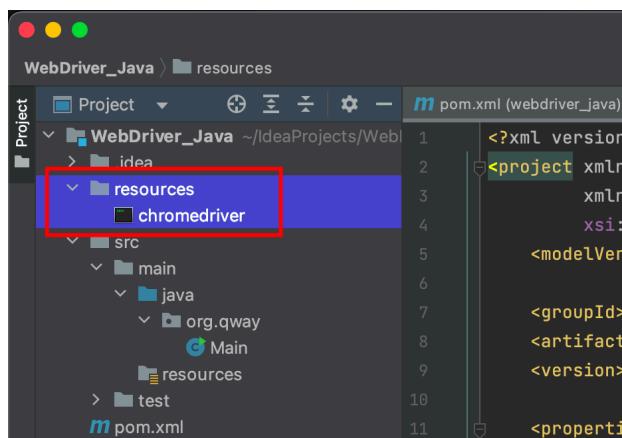
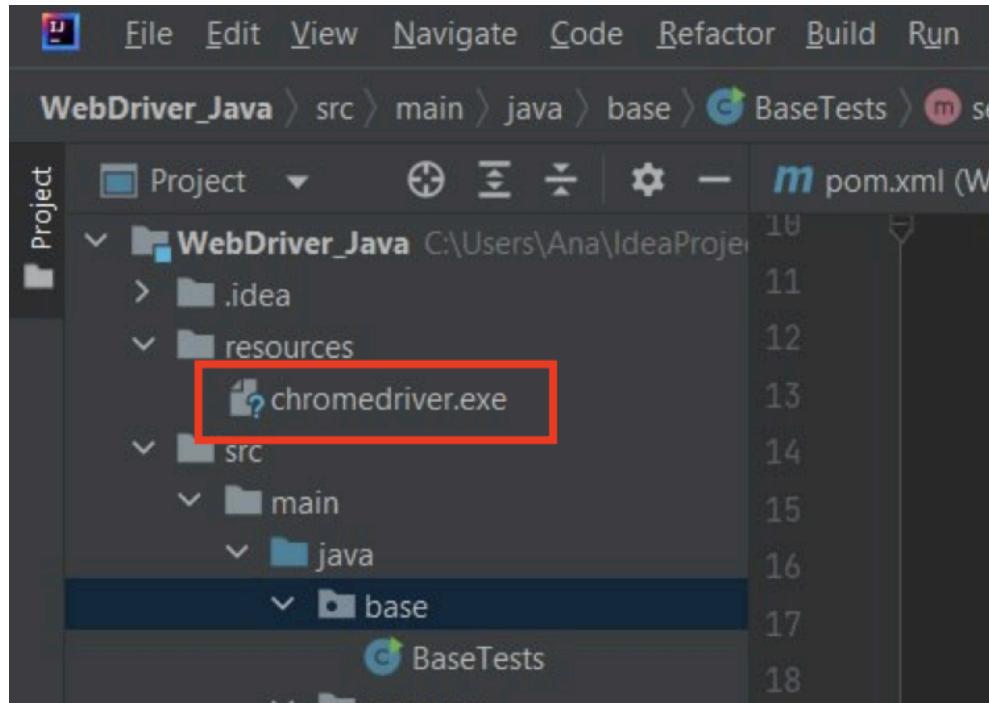


Imagen 17: ChromeDriver adicionado ao projeto

No exemplo acima estou usando o executável para macOS, sem extensão no nome do arquivo, mas se estiver usando Windows o nome do arquivo será “chromedriver.exe”:



Agora usaremos o *Selenium WebDriver* para iniciar um aplicativo, e para isso vamos criar um novo **package** e uma nova **class**.

Sob o diretório “**src**”, temos um diretório “**main**” e um diretório “**test**”, que por sua vez contém outro diretório “**java**”. Vamos criar um **package** dentro de “**java**” clicando com o botão direito do mouse neste diretório e selecionando “**New**” e em seguida “**Package**”:

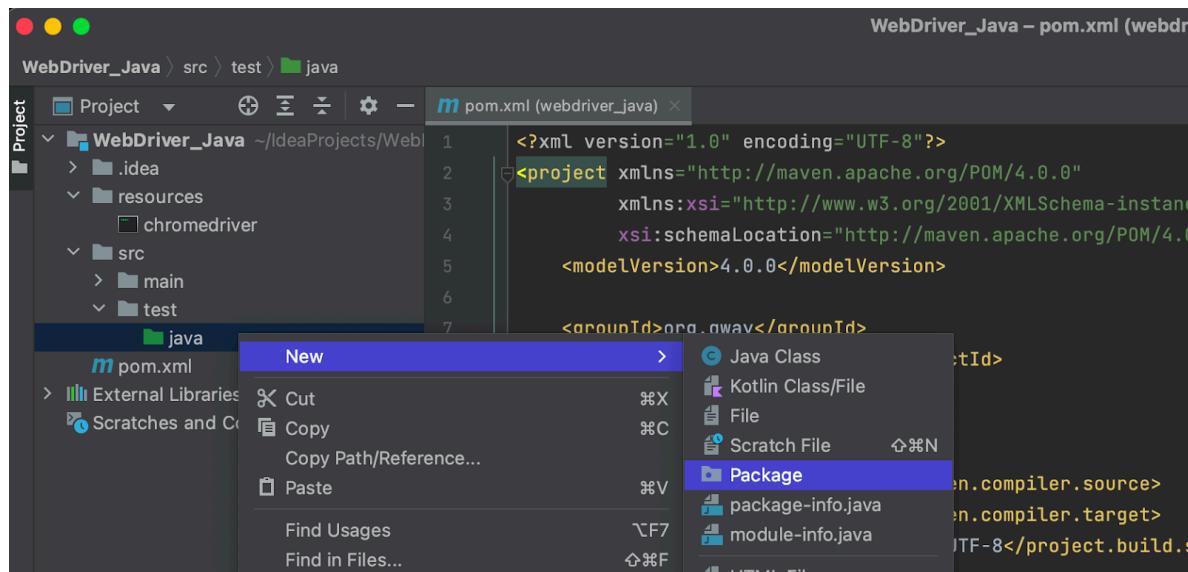


Imagen 18: Criando um novo Package em seu projeto

Vamos criar um pacote aqui chamado “**base**”. Dentro deste novo **package** vamos criar uma nova classe Java, e esta vamos chamar de “**BaseTests.java**”, clicando com o botão direito

do mouse no novo package recém criado e selecionando “**New**” e em seguida “**Java Class**”. A nova classe Java criada será aberta no editor.

3.1 Adicionando e Instanciando o *WebDriver*

A primeira coisa que faremos na classe “**BaseTests.java**” é criar um objeto *WebDriver*.

Ao iniciar a declaração deste objeto digitando **private WebDriver** a função de *AutoComplete* do *IntelliJ* apresentará automaticamente a sugestão *WebDriver* da biblioteca **selenium-chrome-driver** adicionada como dependência durante a configuração do projeto. Tecle “**Enter**” para aceitar a sugestão e importar automaticamente a dependência à classe adicionando a linha **import org.openqa.selenium.WebDriver;**. Acrescente o nome do objeto **driver** e encerre a declaração com “;”.

```

package base;

import org.openqa.selenium.WebDriver;

public class BaseTests {
    private WebDriver driver;
}

```

Imagen 19: Declarando o objeto WebDriver e importando a biblioteca de dependência

Em seguida, criaremos um método que chamaremos de **setUp**.

```

public void setUp() {
    // nosso código seguinte será adicionado aqui
}

```

O Selenium WebDriver precisará saber onde está aquele arquivo executável “**chromedriver**” que você baixou e adicionou ao projeto no diretório “**resources**” que foi criado. Fazemos isso usando um **System.setProperty()** e informando o nome da propriedade e o local onde o ChromeDriver está armazenado. A propriedade que o Selenium irá procurar é chamada **webdriver.chrome.driver**, e deve ser especificada como o primeiro argumento da função **setProperty**. Em seguida, após uma vírgula, o próximo argumento será o local do arquivo “**chromedriver**”. Vai ficar assim:

```
System.setProperty( "webdriver.chrome.driver", "resources/chromedriver" );
```

No exemplo acima estou usando o executável para macOS, sem extensão no nome do arquivo, mas se estiver usando Windows o nome do arquivo será “chromedriver.exe”.

Depois de configurar a propriedade, agora vamos instanciar o objeto **WebDriver**.

Você pode especificar o tipo de driver que deseja usar. Como baixamos o *ChromeDriver*, então é esse que usaremos, mas também existem outras opções para os diferentes navegadores:

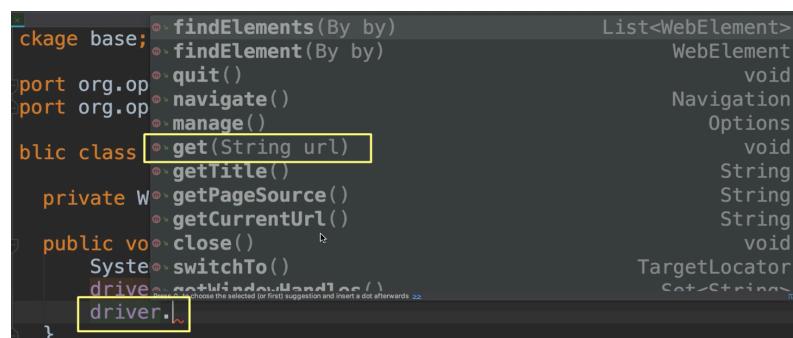
- *ChromeDriver* (que usaremos)
- *EdgeDriver* para o browser Edge
- *FirefoxDriver*
- *InternetExplorerDriver*
- *OperaDriver*
- *SafariDriver*
- *RemoteWebDriver*

Para instanciar o nosso driver como um *ChromeDriver*, usamos o seguinte código:

```
driver = new ChromeDriver();
```

Ao digitarmos esta linha o *IntelliJ* apresentará uma sugestão de dependência que precisa ser importada para a class, bastando pressionar **Option+Enter** para que a linha **import org.openqa.selenium.chrome.ChromeDriver;** seja adicionada.

Esta é a nossa arma secreta para fazer automação de testes. Digitando **driver** vemos que existem muitos métodos diferentes disponíveis e todos eles têm a ver com a interação com o *browser*. Falaremos sobre a maioria deles, mas por enquanto o que queremos fazer é iniciar o navegador.



The screenshot shows a Java code editor with the following code:

```
ckage base;
import org.op...
public class ...
    private W...
    public vo...
        System...
        driv...
    }
}
```

A tooltip is displayed over the `driver.` part of the code, listing several methods:

Method	Description
<code>findElements(By by)</code>	List<WebElement>
<code>findElement(By by)</code>	WebElement
<code>quit()</code>	void
<code>navigate()</code>	Navigation
<code>manage()</code>	Options
<code>get(String url)</code>	void
<code>getTitle()</code>	String
<code>getPageSource()</code>	String
<code>getCurrentUrl()</code>	String
<code>close()</code>	void
<code>switchTo()</code>	TargetLocator
<code>getWindowHandles()</code>	Set<String>

Imagen 20: Acessando os métodos do objeto WebDriver

Para abrir o navegador e acessar uma URL usaremos o método `get` e passaremos uma `String`, que representa a URL.

3.2 Acessando a aplicação a ser testada

A aplicação que queremos testar é um projeto de código aberto usado para automação:

<https://the-internet.herokuapp.com/>

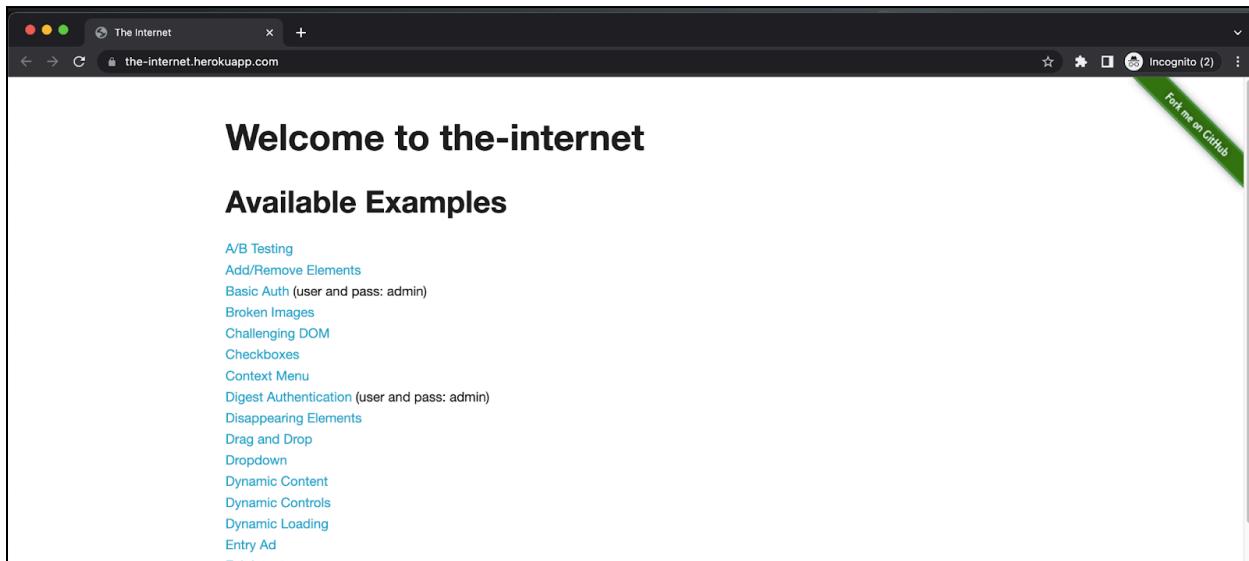


Imagen 21: The Internet é o site que será utilizado para este curso

O aplicativo “**the-internet**” tem todos os tipos de exemplos de diferentes elementos da web e coisas que você costuma ver em sites. Isso nos dá muitas boas práticas para automação de testes. Este é o site que usaremos em nosso curso. Vamos copiar o URL <https://the-internet.herokuapp.com/> e colá-lo dentro do nosso código.

```
driver.get("https://the-internet.herokuapp.com/");
```

Isso é tudo o que precisamos para iniciar o navegador e acessar a URL. Para garantir que o navegador seja iniciado e que ele realmente seja carregado para este aplicativo, vamos escrever mais um comando para imprimir algo no *Console*:

```
System.out.println("==> Page Title: " + driver.getTitle());
```

O comando acima vai concatenar a `String` “`==> Page Title:` ” com o título obtido da página acessada.

Considero uma boa prática inserir outputs como este que podem ser usados para informar ações que serão executadas (costumo iniciar com “`###`”) ou apresentando resultados obtidos (costumo iniciar com “`==>`”).

Da mesma forma, podemos inserir um comando de output antes de acessar a URL, informando o que será feito, utilizando o seguinte comando:

```
System.out.println("### Accessing \"https://the-internet.herokuapp.com/\" ...");
```

Para executar tudo o que implementamos até agora, precisaremos de um método **“main”** apenas porque ainda não configuramos nosso projeto para fazer nenhum teste. Por enquanto, vamos escrever um método **main** temporário que será removido depois.

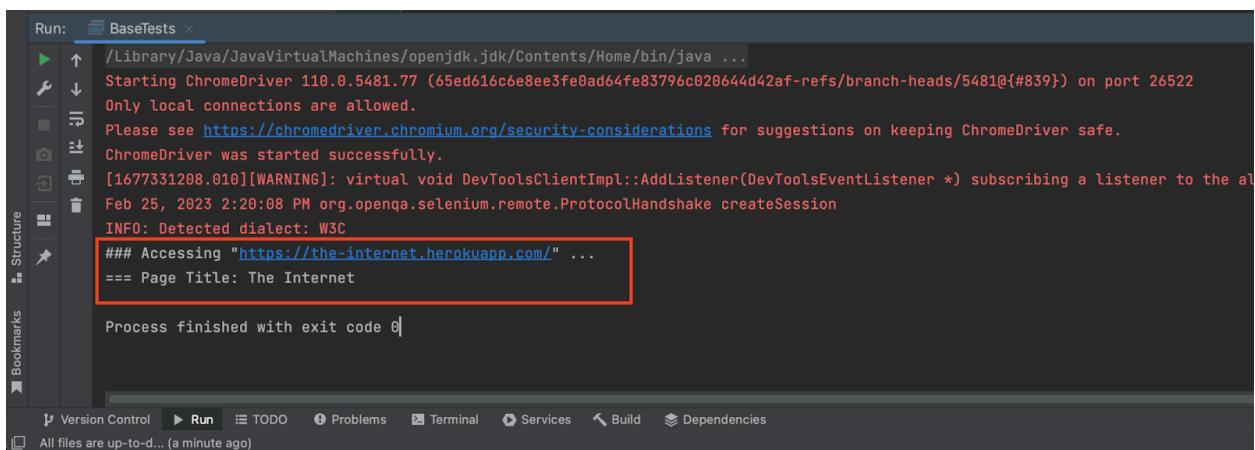
Dentro deste método **main**, criaremos uma instância da classe **BaseTests.java** e invocaremos o método **setUp**.

```
public static void main(String[] args) {
    BaseTests test = new BaseTests();
    test.setUp();
}
```

Para executar este método, clique com o botão direito do mouse em qualquer lugar do código e em seguida clique em “**Run ‘BaseTests.main()’**”.

Veremos que o navegador será carregado e a URL será acessada. Também veremos uma pequena mensagem dizendo **“O Chrome está sendo controlado por um software de testes automatizados”**. Esse é o nosso *Selenium*.

No *IntelliJ*, veremos que os outputs inseridos no código serão impressos no *Console*:



```
Run: BaseTests x
/Library/Java/JavaVirtualMachines/openjdk.jdk/Contents/Home/bin/java ...
Starting ChromeDriver 110.0.5481.77 (65ed616c6e8ee3fe0ad64fe83796c020644d42af-refs/branch-heads/5481@{#839}) on port 26522
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[1677331208.010][WARNING]: virtual void DevToolsClientImpl::AddListener(DevToolsEventListener *) subscribing a listener to the al
Feb 25, 2023 2:20:08 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
### Accessing "https://the-internet.herokuapp.com/"
== Page Title: The Internet

Process finished with exit code 0
```

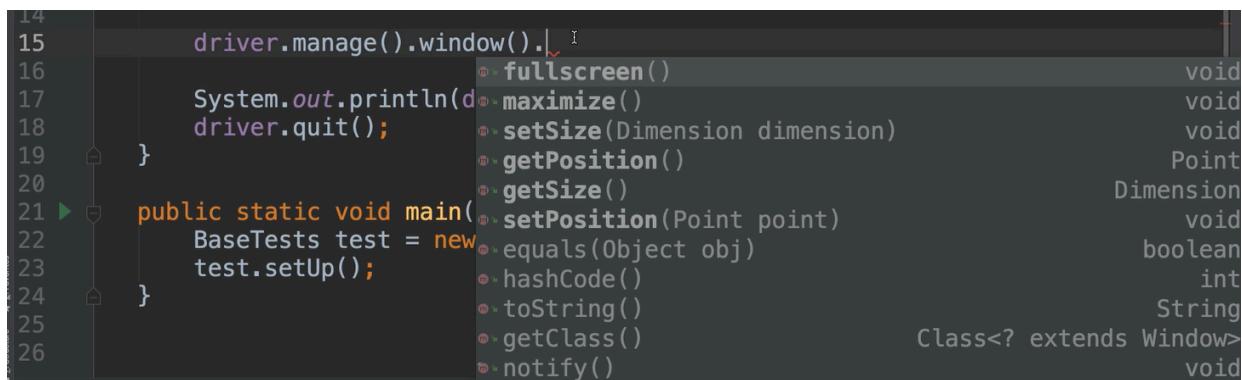
Imagen 22: Output da execução de nosso projeto no console

Existem alguns outros outputs aqui que estão em vermelho. Se esta é a primeira vez que você usa o *ChromeDriver*, isso pode parecer alarmante, mas isso não é um erro, é apenas uma informação sendo impressa e podemos ignorá-la.

Observe que a janela do *browser* foi aberta em um determinado tamanho e não preencheu a tela inteira. Talvez você queira alterar o tamanho da janela antes de começar a

fazer seus testes. Você pode fazer isso de várias maneiras diferentes. Abaixo veremos algumas.

Após a linha de código que cria uma nova instância do *ChromeDriver* e inicia o *browser*, vamos inserir `manage().window()`. Isso nos fornecerá métodos que permitam gerenciar a janela do *browser*.



```

14
15     driver.manage().window().I
16         fullscreen()
17         System.out.println(dI
18         maximize()
19         driver.quit();
20
21 > public static void main(I
22     BaseTests test = newI
23     test.setUp();
24
25 }
26

```

Imagen 23: Acessando os métodos para manipular o tamanho da tela do Chrome Driver

Se usarmos `maximize()`, podemos ver que ele iniciará o aplicativo, maximizará a janela e continuará executando as ações seguintes:

```
driver.manage().window().maximize();
```

Outra opção é o modo de tela cheia utilizando `fullscreen()`:

```
driver.manage().window().fullscreen();
```

E se você tiver um tamanho específico em que deseja redimensionar o navegador? Por exemplo, digamos que queríamos testar nosso aplicativo no tamanho de um dispositivo móvel.

Se quisermos usar o Selenium para executar o teste no modo de exibição de um *iPhone X*, podemos fazê-lo redimensionando para **375x812**. No objeto `window()`, usaremos o método `setSize()` passando como argumento um objeto **Dimension** chamado **size** que foi instanciado recebendo dois inteiros: uma largura e uma altura. Ao incluirmos o objeto **Dimension** no código o *IntelliJ* irá sugerir a importação desta biblioteca. Pressione “**Option+Enter**” para aceitar esta sugestão.

```
Dimension size = new Dimension(375, 812);
driver.manage().window().setSize(size);
```

Vamos executar este código agora e vê-lo no modo de exibição do *iPhone X*. Veremos que agora a janela do *Chrome* estará na largura e na altura que escolhemos.

Agora, para fechar o navegador (pois o deixamos aberto) usaremos o método `quit()` do `WebDriver`.

```
driver.quit();
```

Há também o método `close()` que apenas fechará a janela, mas não a sessão. Usando `quit()`, ele fechará qualquer uma das janelas abertas e encerrará esta sessão.

Ao executar novamente o método `main()` veremos que desta vez a janela do Chrome será apresentada e fechada rapidamente após carregar a URL, antes que possamos ver qualquer coisa. Mas vemos o output no IntelliJ, então sabemos que funcionou.

O código final será o seguinte:

Código Final
Capítulo 3

```
package base;

import org.openqa.selenium.Dimension;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

public class BaseTests {

    private WebDriver driver;

    public void setUp() {
        System.setProperty("webdriver.chrome.driver", "resources/chromedriver");
        driver = new ChromeDriver();

        Dimension size = new Dimension(375, 812);
        driver.manage().window().setSize(size);

        String URL = "https://the-internet.herokuapp.com/";
        System.out.println("### Accessing \'" + URL + "\' ...");
        driver.get(URL);

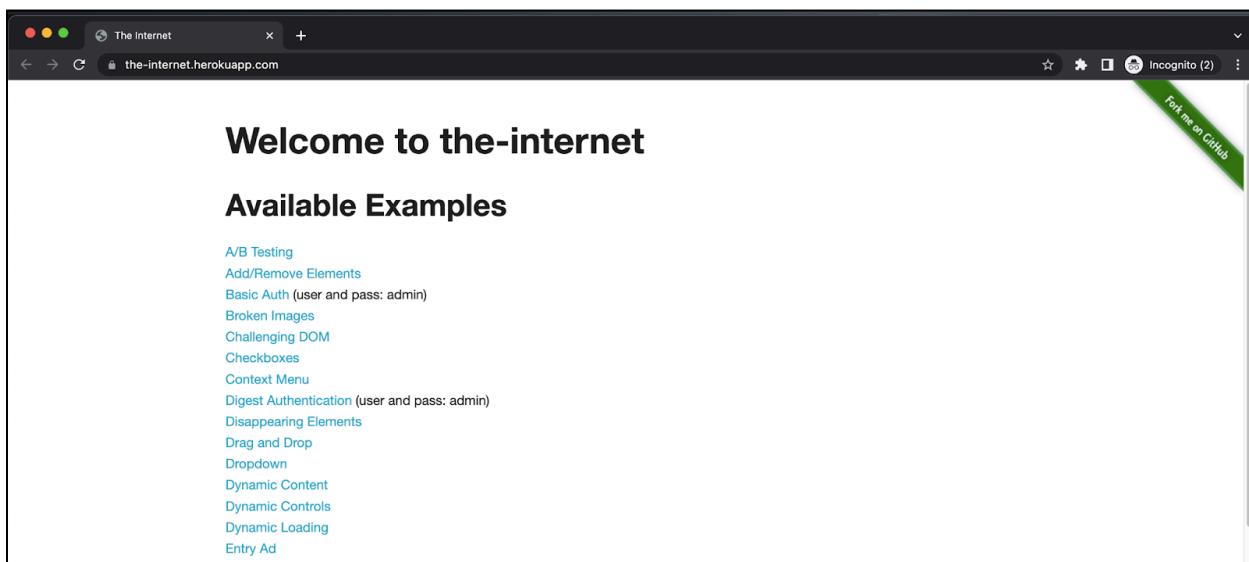
        System.out.println("== Page Title: " + driver.getTitle());
        driver.quit();
    }

    public static void main(String[] args) {
        BaseTests test = new BaseTests();
        test.setUp();
    }
}
```

Capítulo 4 - Encontrando WebElements

Como mencionado no capítulo anterior, o *WebDriver* é usado para interagir com o navegador e, para isso, ele deve ser capaz de localizar os elementos da página acessada.

Vamos analisar a nossa aplicação novamente:



O que compõe esta página são elementos como o título, o subtítulo e todos os links.

Precisamos informar ao *WebDriver*, com qual desses elementos queremos que ele interaja e que ação queremos que ele execute. Primeiro, vamos verificar como identificamos os elementos.

Digamos que queremos automatizar o clique em um desses links. Vamos escolher um, talvez "**Inputs**". Precisamos dizer ao *WebDriver*: "Quero que você encontre este elemento na página".

Para que ele encontre esse elemento, você deve fornecer uma maneira de localizar através do que é fornecido no HTML ou no DOM (*Document Object Model*).

Clique com o botão direito do mouse nesse elemento específico e escolha a opção "**Inspecionar**" do *Chrome*. Fazendo isso teremos acesso ao painel de elementos

O link "**Inputs**" já estará destacado, representado pela tag `<a>` contida por um item de lista composto pela tag ``.

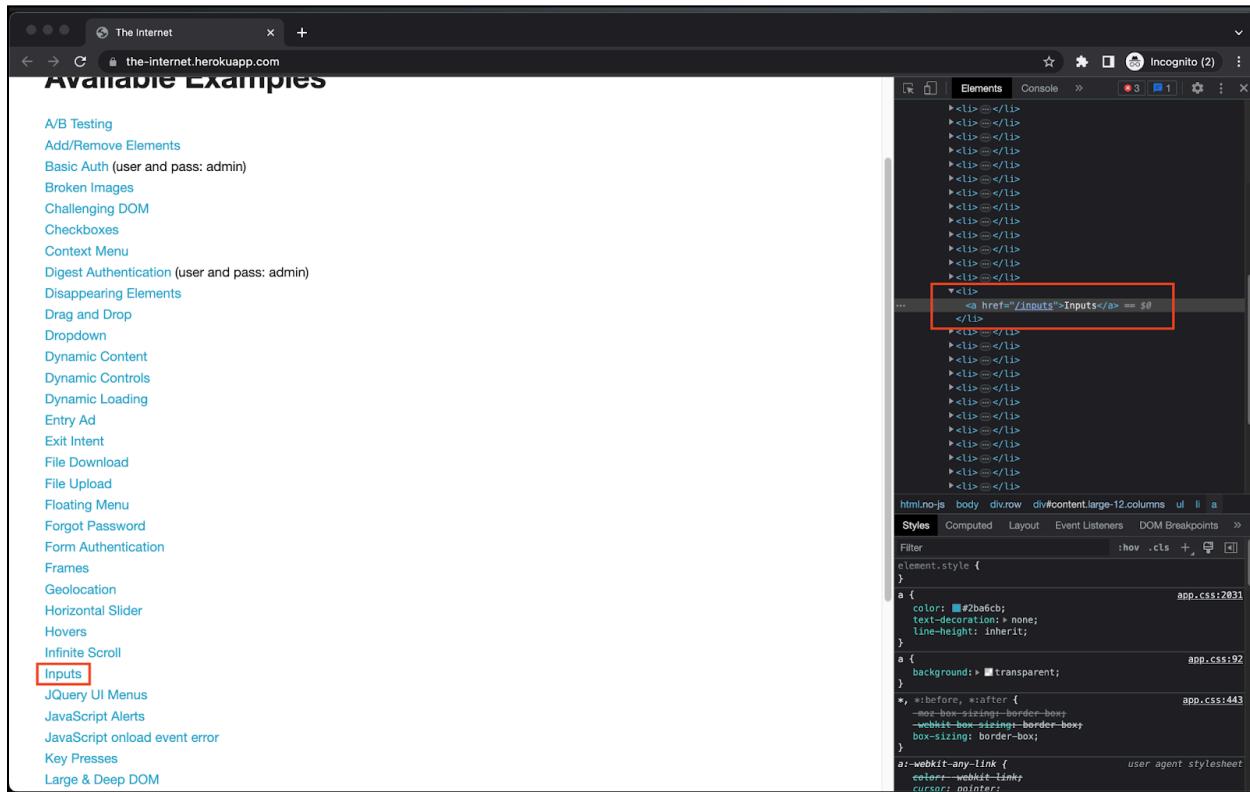


Imagen 23: Inspeccionando o elemento referente ao link Inputs

O método que usaremos para localizar um elemento é o [findElement](#):

```
String URL = "https://the-internet.herokuapp.com/";
System.out.println("### Accessing \" + URL + "\" ...");
driver.get(URL);
```

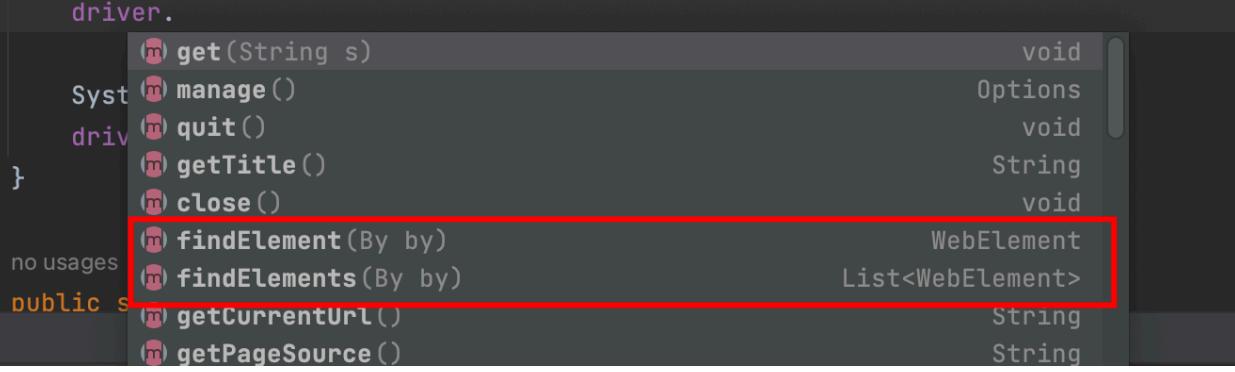
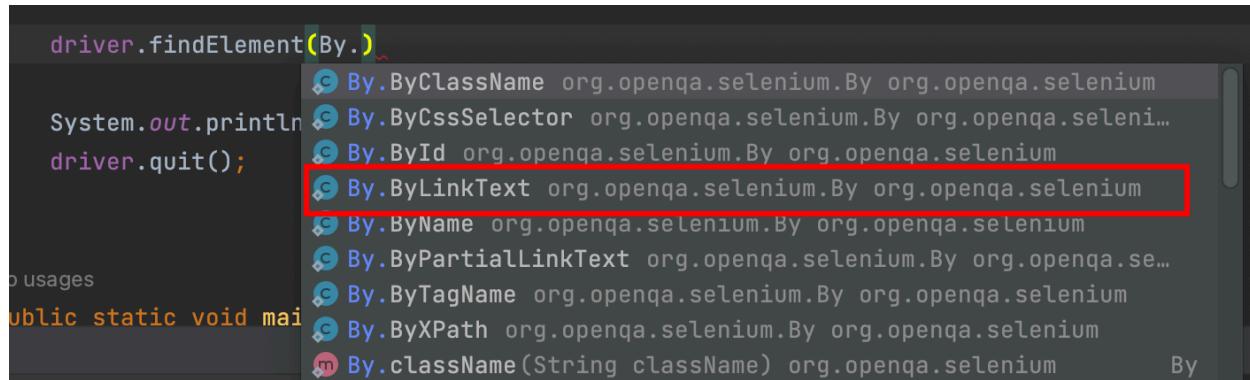


Imagen 24: Métodos do WebDriver para localizar WebElements

Veremos que há dois deles: um é singular e outro é plural. Vamos falar sobre o singular.

O método [findElement](#) retorna um WebElement, a classe que é usada para representar um elemento em uma página. E para descobrir isso, temos que fornecer um objeto [By](#) que pergunta: “como você localiza isso?”.

Digitando `driver.findElement(By.)` veremos muitos métodos estáticos da classe `By`:



```
driver.findElement(By.)
System.out.println("Hello World");
driver.quit();

public static void main(String[] args) {
    // ...
}
```

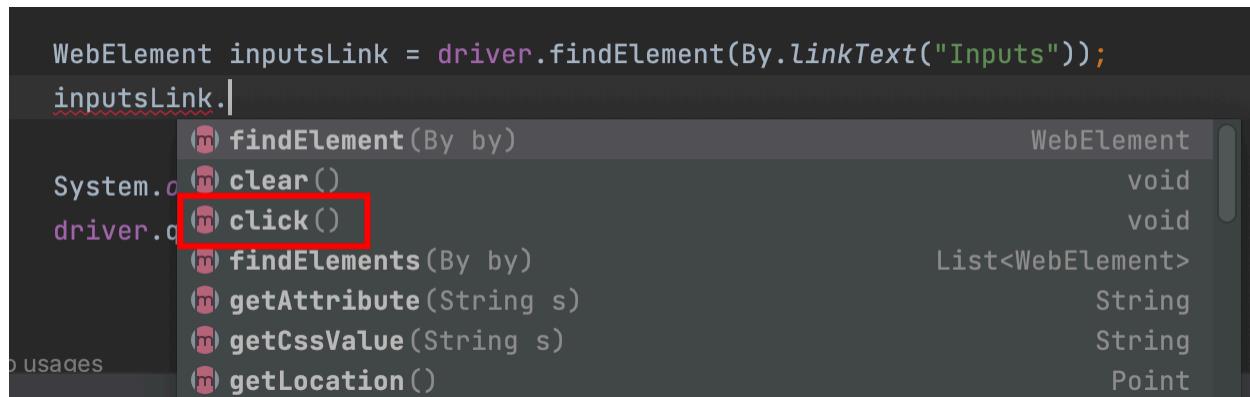
Imagen 25: Métodos da classe By

Estes métodos são várias maneiras de encontrar um elemento pelo texto do link, por seletor CSS, atributo de nome, de classe, de ID, texto parcial do link, nome de tag e xpath.

Neste exemplo, vamos utilizar o método `LinkText()` e o seu argumento será o texto do link que é “**Inputs**”. Isso retornará um `WebElement` com o qual poderemos interagir. Vamos armazenar este `WebElement` em uma variável do tipo `WebElement`.

```
WebElement inputsLink = driver.findElement(By.linkText("Inputs"));
```

Agora, digitando `inputsLink.` veremos que este objeto terá acesso à alguns métodos de interação:



```
WebElement inputsLink = driver.findElement(By.linkText("Inputs"));
inputsLink.

System.out.println("Hello World");
driver.quit();

public static void main(String[] args) {
    // ...
}
```

Imagen 26: Acessando os métodos do WebElement instanciado

Por enquanto, vamos apenas usar o método `click()`. Com as seguintes linhas de código localizarmos o elemento e clicarmos nele:

```
WebElement inputsLink = driver.findElement(By.linkText("Inputs"));
inputsLink.click();
```

Ao executar este código veremos que o *ChromeDriver* realmente clicou no link porque fomos levados para uma nova página, a página de Inputs.

Vejamos também a forma plural deste método: `findElements()`: O método `findElement()` (no singular) retornará apenas um `WebElement` e será o primeiro que encontrar. Se houvessem vários links chamados “`Input`” nessa página, `findElement()` retornaria apenas o primeiro que encontrasse no DOM.

Há casos em que você pode querer obter vários itens com um único localizador. Digamos que quiséssemos encontrar todos os links na página e poder contá-los, por exemplo. Poderíamos fazer algo assim com o método `findElements()` (no plural).

Novamente digitando `driver.findElements(By.)` veremos muitos métodos estáticos. Por exemplo, vamos usar o `tagName()` para encontrar os elementos com a tag `<a>`.



Imagen 27: Tag <a> de um link

Esse método retorna uma lista de *WebElements* que será atribuída à uma variável `List<WebElement> links` e apresentaremos no *Console* o tamanho desta lista:

```
List<WebElement> links = driver.findElements(By.tagName("a"));
System.out.println("==== Amount of links (tags <a>): " + links.size());
```

Precisaremos adicionar este código antes de clicarmos no link “**Inputs**”. A sequência de comandos vai acessar a página principal, contar os links, apresentar esta informação no Console e só então clicar no link “**Inputs**”.

Agora veremos o que acontece se tentarmos encontrar um elemento que não existe, como por exemplo um link que tenha o texto “*Daniel*”.

```
WebElement inputsLink = driver.findElement(By.linkText("Daniel"));
```

```

    List<WebElement> links = driver.findElements(By.tagName("a"));
    System.out.println("== Amount of links found: " + links.size());

    System.out.println("### Searching for link with text \"Inputs\"");
    WebElement inputsLink = driver.findElement(By.linkText("Daniel"));
    System.out.println("### Clicking on link with text \"Inputs\"");
    inputsLink.click();

    driver.quit();
}

```

Run: BaseTests

- ▶ ↑ ### Accessing <https://the-internet.herokuapp.com/> ...
- ⚡ ↓ === Page Title: The Internet
- ⚡ ⚡ === Searching for links (tags <a>)...
- ⚡ ⚡ === Amount of links found: 46
- ⚡ ⚡ ⚡ === Searching for link with text "Inputs"...
- ⚡ ⚡ ⚡ ⚡ Exception in thread "main" org.openqa.selenium.NoSuchElementException: no such element: Unable to locate element: {"method":"link text","selector":"Daniel"} (Session info: chrome=12.0.5615.49)
- ⚡ ⚡ ⚡ ⚡ For documentation on this error, please visit: https://selenium.dev/exceptions/#no_such_element

Imagem 28: Exception ao tentar localizar um elemento que não existe

Vemos aqui que recebemos uma mensagem de erro mencionando a exceção “**NoSuchElementException**”.

Ao fazermos automação de testes com o *WebDriver*, devemos ficar muito confortáveis com essa exceção, porque ocorre com frequência quando tentamos encontrar um elemento e ele não está presente.

Isso pode acontecer por vários motivos. Eventualmente especificamos o localizador incorretamente, ou o elemento ainda não está em um estado que permita ao *WebDriver* localizá-lo. Por exemplo, se clicarmos em um link e tentarmos contar os links ou tentarmos encontrar algo na página que não está lá. Mas falaremos sobre como lidar com todos esses tipos de situações nos próximos capítulos.

Ao final deste capítulo, nosso código ficou assim:

**Código Final
Capítulo 4**

```

package base;

import org.openqa.selenium.By;
import org.openqa.selenium.Dimension;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.List;

public class BaseTests {

    private WebDriver driver;

    public void setUp() {
        System.setProperty("webdriver.chrome.driver",
"resources/chromedriver");
        driver = new ChromeDriver();

        Dimension size = new Dimension(375, 812);
        driver.manage().window().setSize(size);

        String URL = "https://the-internet.herokuapp.com/";
    }
}

```

```
System.out.println("### Accessing \'" + URL + "\' . . .");
driver.get(URL);
System.out.println("== Page Title: " + driver.getTitle());

System.out.println("### Searching for links (tags <a>) . . .");
List<WebElement> links = driver.findElements(By.tagName("a"));
System.out.println("== Amount of links found: " + links.size());

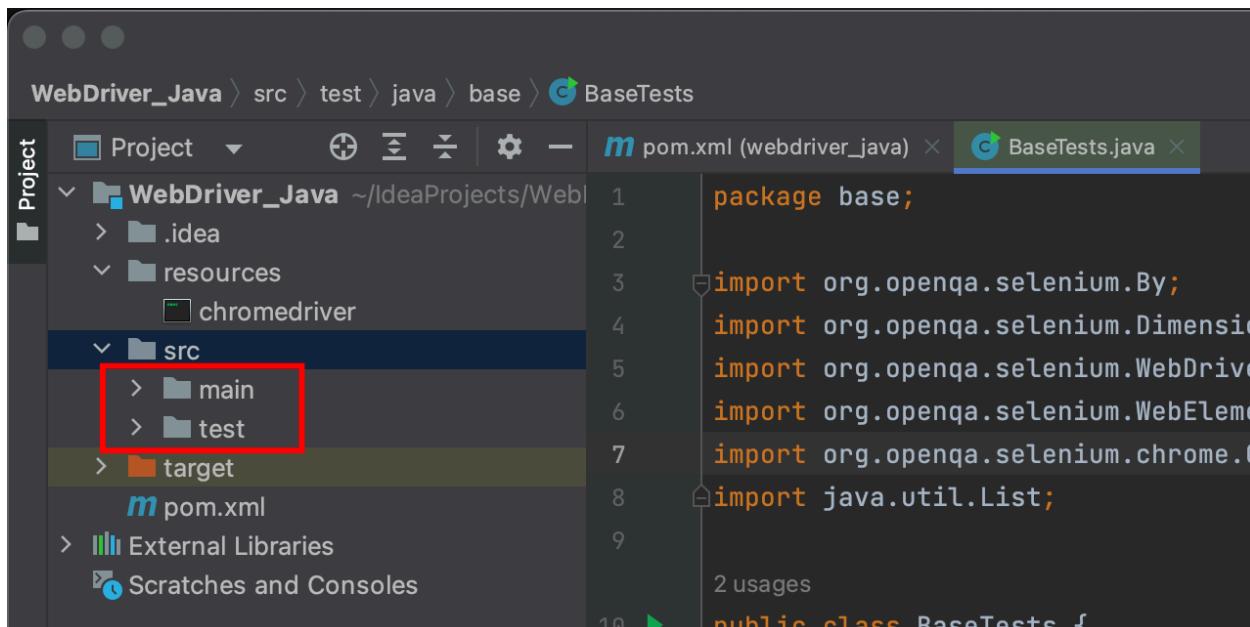
System.out.println("### Searching for link with text \'Inputs\' . . .");
WebElement inputsLink = driver.findElement(By.linkText("Inputs"));
System.out.println("### Clicking on link with text \'Inputs\' . . .");
inputsLink.click();

driver.quit();
}

public static void main(String[] args){
    BaseTests test = new BaseTests();
    test.setUp();
}
}
```

Capítulo 5 - Interagindo com *WebElements*

Em projetos de automação de teste, normalmente há duas camadas: uma é conhecida como camada de framework e a outra é conhecida como camada de teste.



The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** WebDriver_Java > src > test > java > base > BaseTests
- Toolbars:** Project, New, Open, Save, Settings, Minimize, Maximize, Close.
- Code Editor:** pom.xml (webdriver_java) is the active tab, showing XML configuration. BaseTests.java is the current file, showing Java code for a Selenium test base class.
- Project Explorer:** Shows the project structure:
 - WebDriver_Java (Project root)
 - .idea (Idea-specific files)
 - resources (contains chromedriver)
 - src (highlighted with a red box)
 - main
 - test
 - target
 - pom.xml
 - External Libraries
 - Scratches and Consoles

Imagen 29: Camadas de Framework e de Testes

Na visualização do projeto podemos encontrar sob a pasta “**src**” (source) 2 pastas:

- A pasta “**main**” é onde normalmente colocamos o framework
- A pasta “**test**” é onde normalmente colocamos os testes

Todas as interações com o navegador da Web (basicamente toda a codificação que é feita internamente na aplicação) devem estar no framework. Os arquivos de teste devem se concentrar apenas nos testes em si.

Vamos demonstrar isso usando um exemplo:

- Encontrar os elementos e clicar neles são coisas que devem ser feitas no framework. Seu teste em si não deveria focar em encontrar um elemento específico por seu seletor ou localizador.
- Seu teste terá foco em coisas como especificar passos de um fluxo: “**Quero clicar em um link**”.

Para isso, usaremos um design pattern chamado ***Page Object Model***.

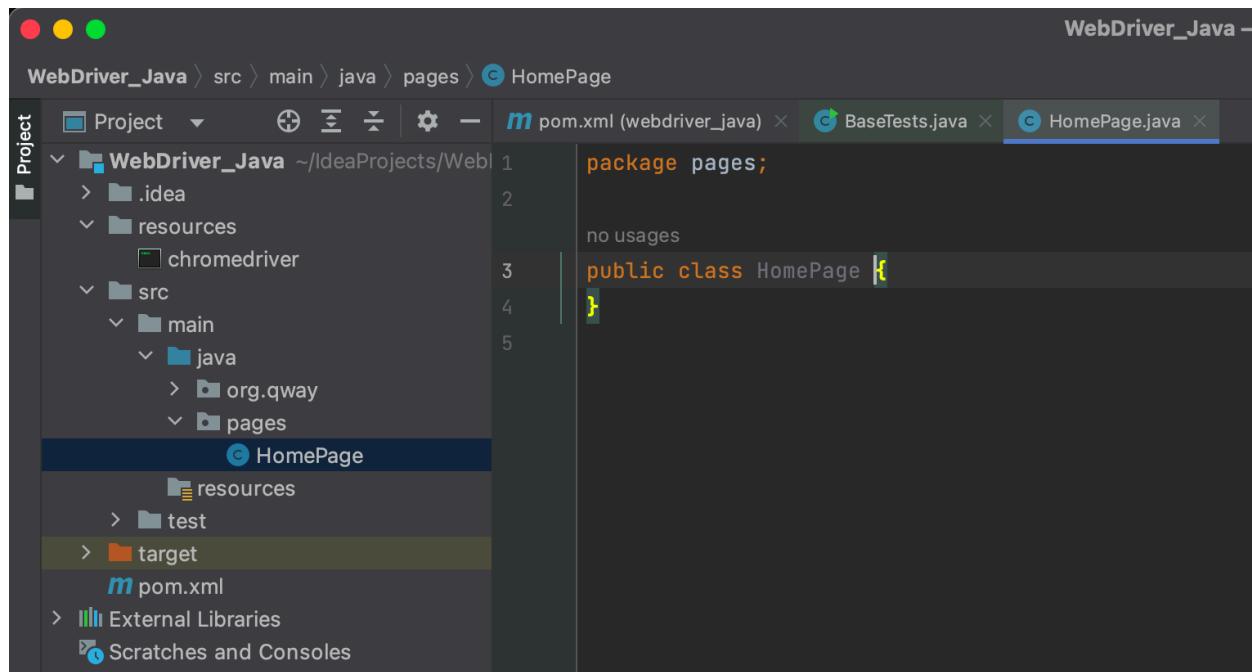
Esta não é a única maneira de projetar seu framework, mas é um dos ***design patterns*** mais populares na automação de testes, então é o que usaremos neste curso.

A maneira como o *design pattern Page Object Model* funciona é que você cria uma classe na seção de framework do projeto que representa uma página em seu aplicativo — e para cada página em seu aplicativo você cria uma nova classe.

5.1 Criando a classe de framework *HomePage*

Vamos criar uma classe para a nossa página inicial na seção de framework do nosso projeto chamada de “**main**”. Aqui temos uma pasta chamada “**java**”, onde criaremos um novo “**package**” clicando com o botão direito do mouse neste diretório e selecionando “**New**” e em seguida “**Package**”. Chamaremos este pacote de “**pages**”, e isso representará nossas classes de *page objects*.

Criaremos uma classe para corresponder à página inicial do aplicativo clicando com o botão direito do mouse neste diretório e selecionando “**New**” e em seguida “**Java Class**”, e chamaremos essa classe de “**HomePage.java**”.



The screenshot shows the IntelliJ IDEA interface with the project 'WebDriver_Java' open. The 'Project' tool window on the left shows the directory structure: 'src/main/java/pages'. A new Java class named 'HomePage' has been created and is currently selected. The code editor on the right contains the following Java code:

```

package pages;

public class HomePage {
}

```

Imagen 30: Nova classe “*HomePage*” criada dentro do package “*pages*”

Nesta classe de *page object* representaremos os elementos da página e criaremos os métodos para interagir com esses elementos.

Observando nossa aplicação novamente, observamos que há muitos elementos, e em teoria poderíamos criar web elements para todos os diferentes links de nossa aplicação, bem como para os títulos, mas isso seria um pouco exagerado. É recomendável que você crie apenas o que precisa no momento e, em seguida, adicione à medida que precisar de mais cenários automatizados. Vamos iniciar com um cenário e construiremos o *Page Object Model* para esse cenário.

Digamos que queremos fazer login no aplicativo — há um link aqui chamado “**Form Authentication**”.

Em nosso cenário, iremos:

- Clicar no link “**Form Authentication**”
- Inserir dados nos campos de “**Username**” e “**Password**”
- Clicar no botão de “**Login**”

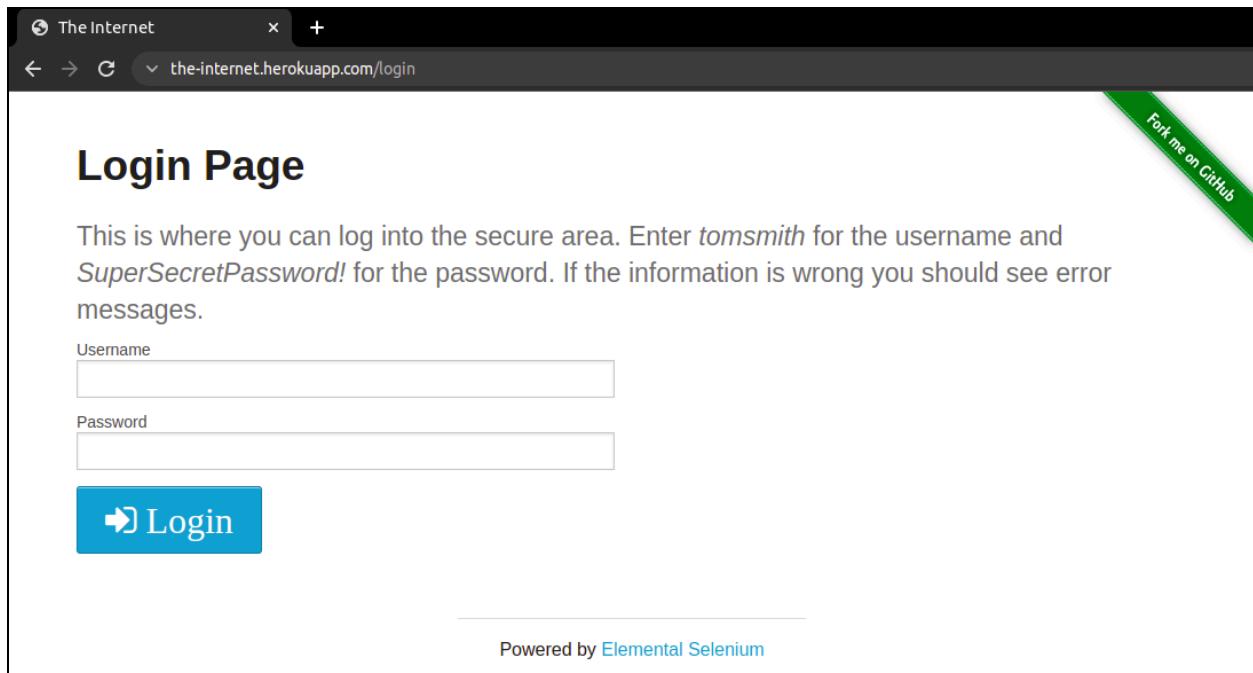


Imagen 31: Form Authentication

Esta página de login é na verdade uma nova página, o que significa que precisaria de uma nova classe se fôssemos seguir o design pattern do Page Object Model, mas não vamos nos preocupar com isso agora.

Na página inicial, tudo o que precisamos é clicar no link “**Form Authentication**”, então vamos apenas criar os web elements e os métodos para fazer isso.

Como aprendemos no capítulo anterior, para interagir com qualquer um dos elementos você deve conhecer o localizador. Então, vamos em frente e obter o localizador para este link.

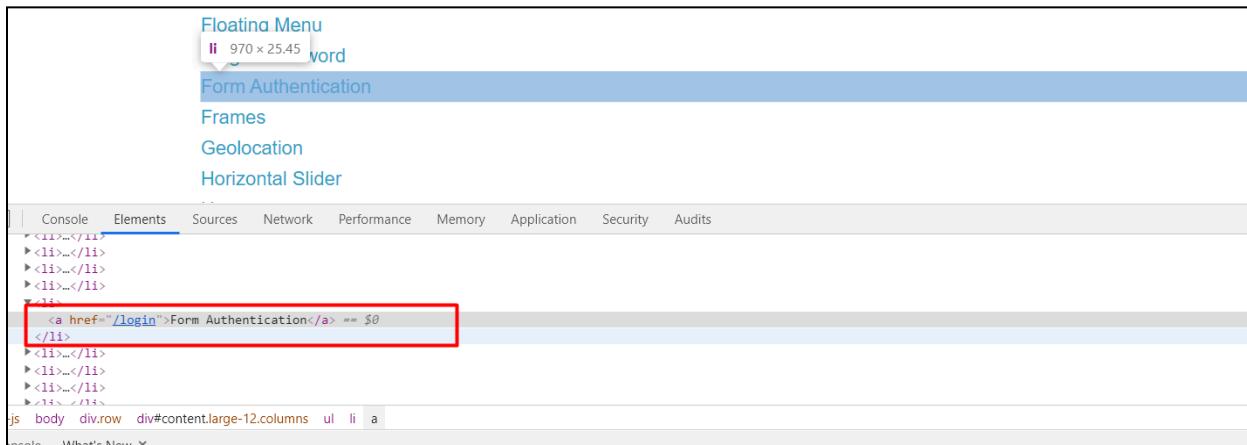


Imagen 32: Link para Form Authentication

Vemos aqui que este é um elemento `<a>`, e tem o texto “**Form Authentication**” como título do link. Vamos em frente e usar isso como nosso identificador:

```
package pages;

import org.openqa.selenium.By;

public class HomePage {
    private By formAuthenticationLink = By.linkText("Form Authentication");
}
```

Vamos quebrar este bloco de código para entender melhor.

Criamos um localizador `By` privado porque é um campo e estamos praticando encapsulamento. Para isso importamos o tipo `By`.

Esse `WebElement` representa o link “**Form Authentication**”, então vamos chamá-lo de `formAuthenticationLink`, e vamos definir isso como `By.linkText()` e, em seguida, passar o texto do link que é “**Form Authentication**”.

Agora criaremos um método para permitir que nossos testes interajam com esse link.

Criaremos um método público e, por enquanto, seu retorno será `void`. É uma boa prática nomear métodos com nomes que expressam o que você está realmente fazendo, então podemos chamar isso de `clickFormAuthenticationLink()`.

Esse método vai localizar o elemento utilizando o método `findElement` que receberá o localizador `By` que foi criado anteriormente. Para utilizar o método `findElement` vamos precisar do objeto `WebDriver` que ainda não foi criado, então vamos criar um método construtor que especificará que ao instanciar essa classe você deve passar o `WebDriver`, e usaremos esse driver para interagir com a página da web.

```
public class HomePage {
    private WebDriver driver;
    private By formAuthenticationLink = By.linkText("Form Authentication");

    public HomePage(WebDriver driver) {
        this.driver = driver;
    }
}
```

O que será passado ao método construtor como argumento é o **driver** que já lançou o browser que possui uma sessão aberta que continuaremos usando para interagir com o site.

Agora que temos nosso **driver**, podemos voltar ao método **clickFormAuthenticationLink()**, e usar o comando **driver.findElement** passando o localizador **formAuthenticationLink** como argumento para encontrar o *WebElement*.

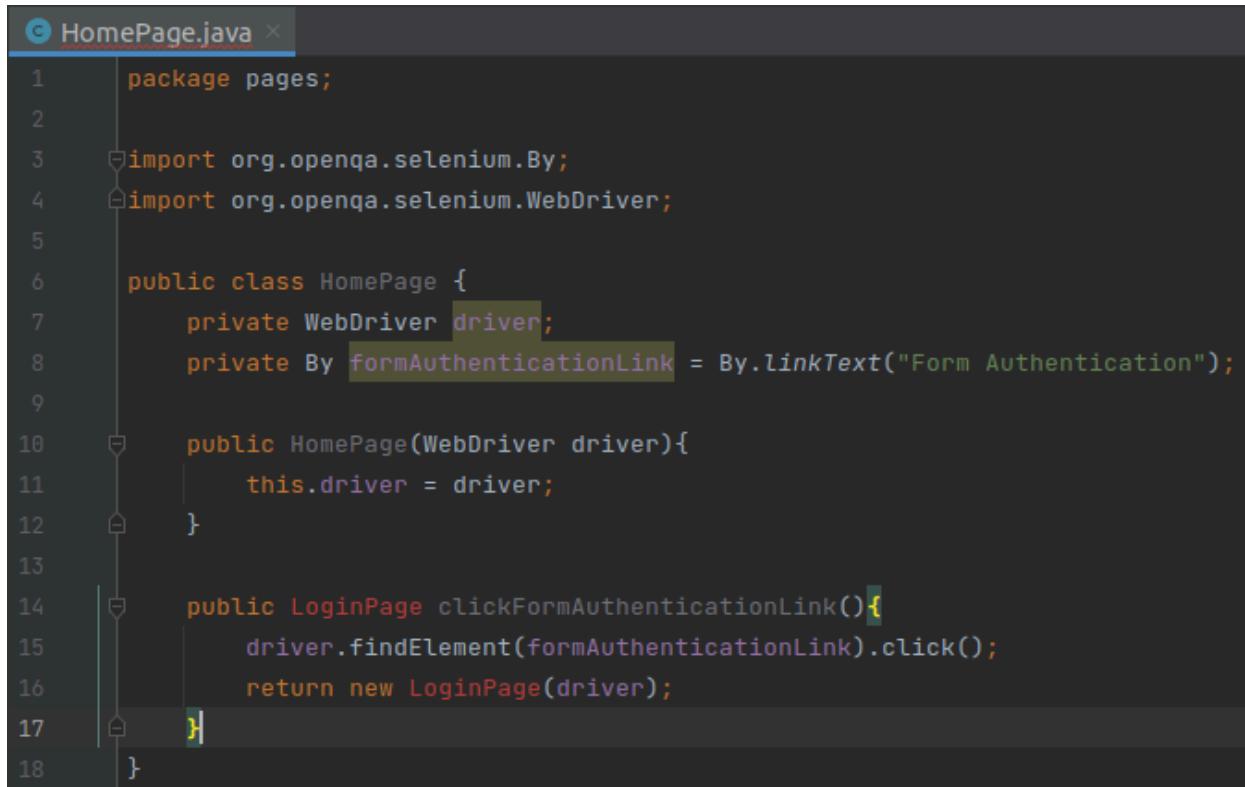
Em seguida, precisamos realizar um clique neste *WebElement* com a função **click()**.

```
public void clickFormAuthenticationLink() {
    driver.findElement(formAuthenticationLink).click();
}
```

Se clicarmos neste link "**Form Authentication**", somos levados para uma nova página. Se sua ação alterar a página, o design pattern Page Object Model recomenda que você deverá retornar um identificador para essa nova página. Assim, a partir do nosso método de clicar nesse link, devemos retornar um objeto **LoginPage**. Então, no lugar de **void**, vamos mudar isso para um retorno do tipo **LoginPage**.

```
public LoginPage clickFormAuthenticationLink() {
    driver.findElement(formAuthenticationLink).click();
    return new LoginPage(driver);
}
```

O **return** desse método também precisará de um driver porque todas as páginas precisam de um driver para interagir com o navegador. Usaremos o mesmo driver que já temos, que já tem uma sessão aberta, e vamos enviar para a **LoginPage** para que ele continue interagindo com o navegador.



```

1 package pages;
2
3 import org.openqa.selenium.By;
4 import org.openqa.selenium.WebDriver;
5
6 public class HomePage {
7     private WebDriver driver;
8     private By formAuthenticationLink = By.linkText("Form Authentication");
9
10    public HomePage(WebDriver driver){
11        this.driver = driver;
12    }
13
14    public LoginPage clickFormAuthenticationLink(){
15        driver.findElement(formAuthenticationLink).click();
16        return new LoginPage(driver);
17    }
18 }

```

Imagen 33: Método `clickFormAuthenticationLink` com retorno de tipo desconhecido

`LoginPage` está vermelho agora porque não criamos esta classe ainda, então vamos criá-la agora.

5.2 Criando a classe de framework `LoginPage`

Onde a nova classe de página de login deve ser criada? Se a classe representa uma página no aplicativo, então sabemos que deverá ser criada na seção do framework, onde temos este pacote chamado “`pages`”.

Vamos criar outra classe abaixo deste pacote, e esta se chamará `LoginPage.java`, e clicamos em OK.

Novamente, já sabemos que isso precisará ter um `WebDriver`. Precisaremos também criar um construtor que receba esse `WebDriver`:

```

public class LoginPage {
    private WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }
}

```

Sabemos que precisamos inserir um nome de usuário, então precisaremos inspecionar este campo "**Username**" e obter o identificador para isso, bem como para o campo "**Password**" e para o botão "**Login**".

São três elementos com os quais temos que interagir para nossos testes, então adicionaremos três campos em nossa classe de page objects e, em seguida, três métodos para interagir com eles.

Login Page

This is where you can log into the secure area. Enter *tomsmith* for the username and *SuperSecretPassword!* for the password. If the information is wrong you should see error messages.

Username

Password

Login

Powered by Elemental Selenium

```
<!DOCTYPE html>
<!--[if IE 8]>         <html class="no-js lt-ie9" lang="en" ><![endif]-->
<!--[if gt IE 8]>!-->
<!--<![endif]-->
<html class="no-js" lang="en">
<!--<![endif]-->
<head></head>
<body data-new-gr-c-s-check-loaded="14.1050.0" data-gr-ext-installed>
  <div class="row"></div>
  <div class="row">
    <div class="example">
      <a href="https://github.com/tourdedave/the-internet"></a>
    </div>
    <div id="content" class="large-12 columns">
      <div class="example">
        <h2>Login Page</h2>
        <h4>Subheader</h4>
        <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 class="large-6 small-12 columns">
              <label for="password">Password</label>
              <input type="password" name="password" id="password" />
            </div>
          </div>
        </form>
      </div>
    </div>
  </div>
</body>
```

Imagen 34: Elementos <input> para Username e Password

O campo de Username é uma tag **<input>** que tem o ***id="username"***. Vamos adicionar isso ao nosso framework. Vamos fazer o mesmo para o campo de Password que tem o ***id="password"***.

Finalmente, vamos obter o botão Login. Nós temos a tag **<button>** mas ela não tem um id. Seu tipo é ***submit***, mas podemos encontrar um localizador melhor usando outra tag.

A tag **<form>** tem um ***id="Login"***, então podemos localizar o elemento button que está contido no formulário. Assim, podemos usar um seletor CSS utilizando a expressão "**#login button**". Usaremos Ctrl+F para testar o seletor.

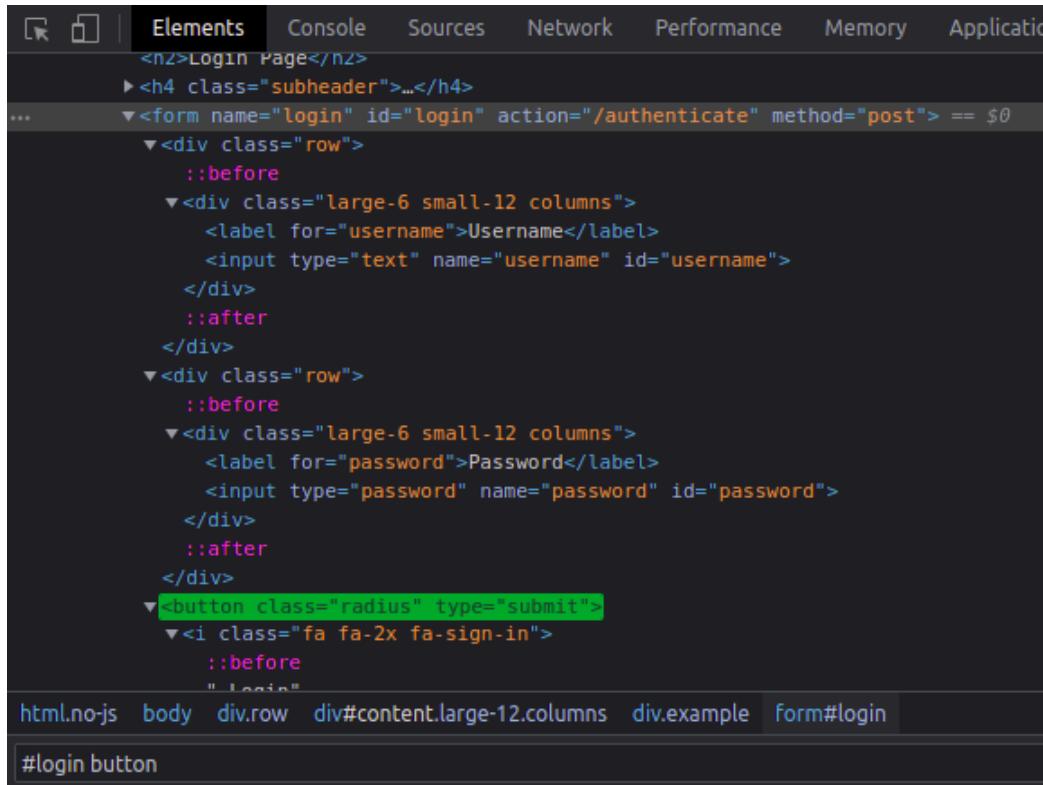


Imagen 35: Localizando o elemento `<button>` contido no `<form>` de id “login”

Usaremos este seletor com a função `By.cssSelector()`. Os 3 elementos serão definidos desta forma:

```

private By usernameField = By.id("username");
private By passwordField = By.id("password");
private By loginButton = By.cssSelector("#login button");

```

Agora criaremos métodos para interagir com esses elementos. O primeiro será chamado de `setUsername()`. Em nossos testes, precisaremos passar o nome de usuário que usaremos, então vamos definir um argumento do tipo `String` chamado `username`.

Localizaremos o campo com o comando `driver.findElement()`, informando o elemento `usernameField` que definimos antes. O método para digitar texto é `sendKeys()`, informando a String que foi recebida como argumento.

Ao fazer isso, não há mudança de página ou algo assim, então não há problema em ter esse tipo de retorno como `void`.

```

public void setUsername(String username) {
  driver.findElement(usernameField).sendKeys(username);
}

```

Agora vamos fazer a mesma coisa para o campo de `passwordField`.

```
public void setPassword(String password) {
    driver.findElement(passwordField).sendKeys(password);
}
```

O próximo passo será criar um método para clicar no botão de Login, que nos levará a uma nova página chamada “**Secure Area**”. Como alteramos a página, é responsabilidade deste método retornar um identificador para essa nova página, então vamos fazer uma nova classe para ela.

Nosso tipo de retorno aqui será **SecureAreaPage**, que ainda não criamos, e esse método será chamado de **clickLoginButton**.

Localizaremos o campo com o comando **driver.findElement()**, informando o elemento **LoginButton** que definimos antes. O método para clicar em um elemento é **click()**, sem a necessidade de informar nenhum argumento.

Sabendo que após o clique a página foi alterada, então retornamos **SecureAreaPage** que receberá como argumento o **driver**.

```
public SecureAreaPage clickLoginButton() {
    driver.findElement(loginButton).click();
    return new SecureAreaPage(driver);
}
```

O resultado da classe “**LoginPage**” será o seguinte:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LoginPage {
    private WebDriver driver;
    private By usernameField = By.id("username");
    private By passwordField = By.id("password");
    private By loginButton = By.cssSelector("#login button");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void setUsername(String username) {
        driver.findElement(usernameField).sendKeys(username);
    }

    public void setPassword(String password) {
        driver.findElement(passwordField).sendKeys(password);
    }

    public SecureAreaPage clickLoginButton() {
        driver.findElement(loginButton).click();
    }
}
```

Código Final
LoginPage.java

```

        return new SecureAreaPage(driver);
    }
}

```

5.3 Criando a classe de framework *SecureAreaPage*

Vamos criar a classe para a **Secure Area** page. Novamente, esta nova classe Java estará no pacote **pages** de nosso framework com o nome “**SecureAreaPage.java**” e, claro, isso precisará do *WebDriver*, bem como de um construtor.

```

package pages;

import org.openqa.selenium.WebDriver;

public class SecureAreaPage {
    private WebDriver driver;

    public SecureAreaPage(WebDriver driver) {
        this.driver = driver;
    }
}

```

Agora vamos retornar à nossa aplicação e verificar o que precisaremos desta página.

Para nosso cenário, após fazer login veremos este banner que diz “**You logged into a secure area!**”

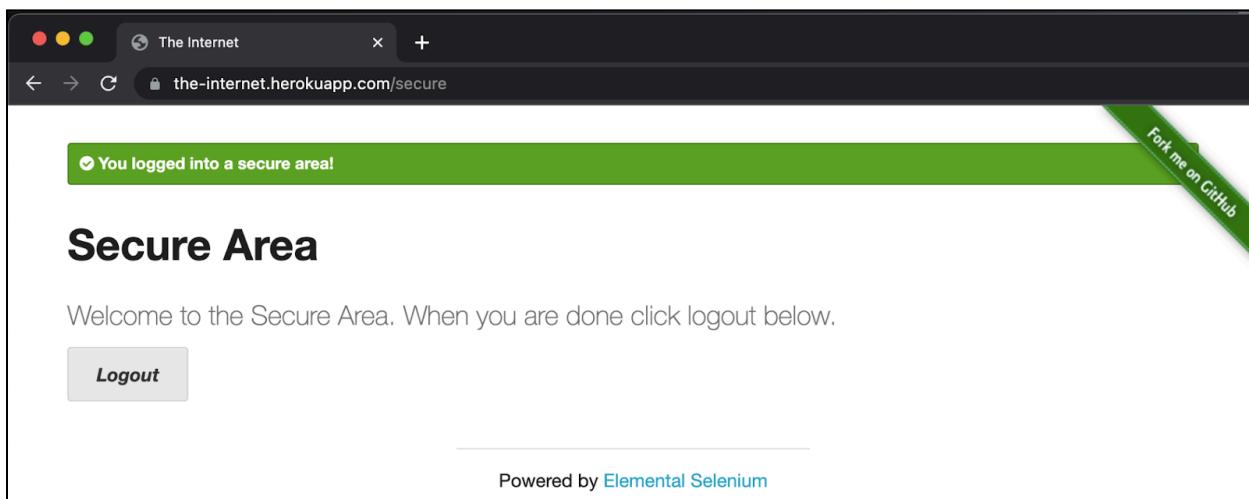


Imagen 36: Banner de confirmação de Login

Podemos checar e confirmar que ele está lá, e para isso precisaremos adicionar este elemento e um método para ler seu conteúdo. Então vamos inspecionar este elemento.

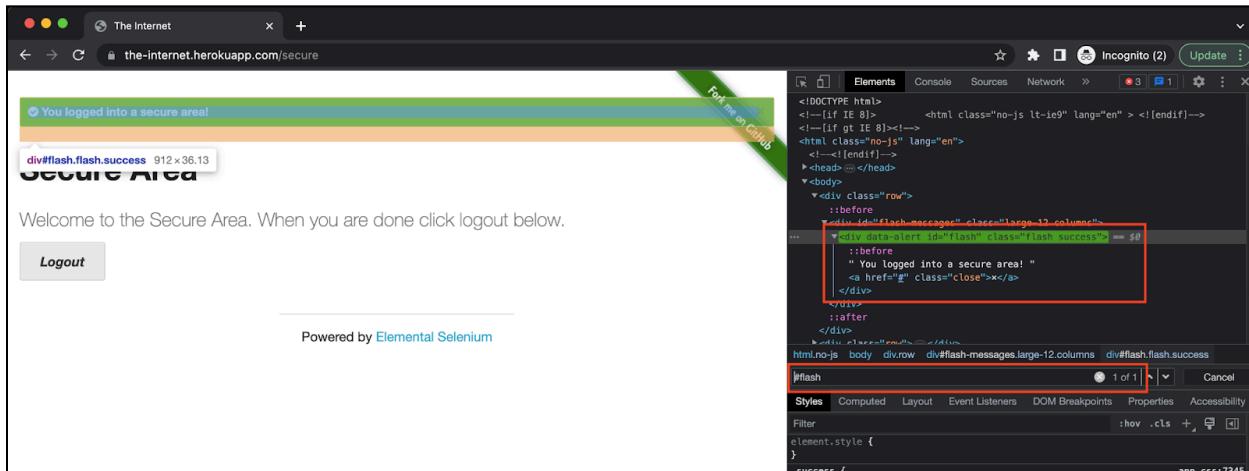


Imagen 37: Elemento `<div>` com id “`flash`”

Aqui vemos que este elemento `<div>` tem um `id =“flash”`. Utilizando o Search (Cmd+F) podemos confirmar que existe apenas 1 elemento com este id. Vamos adicionar este `WebElement` à nossa nova classe e chamá-lo de `statusAlert`.

```
private By statusAlert = By.id("flash");
```

Agora vamos criar um método para interagir com este elemento. Queremos que este método leia o texto do elemento `statusAlert`, então ele precisa retornar uma String com este texto. Para isso, vamos usar `driver.findElement` para localizar o elemento e o método `getText()`, que retornará seu conteúdo:

```
public String getAlertText(){
    return driver.findElement(statusAlert).getText();
}
```

Isto é tudo o que precisamos para concluir nosso cenário de teste, e o resultado da classe `SecureAreaPage.java` será o seguinte:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

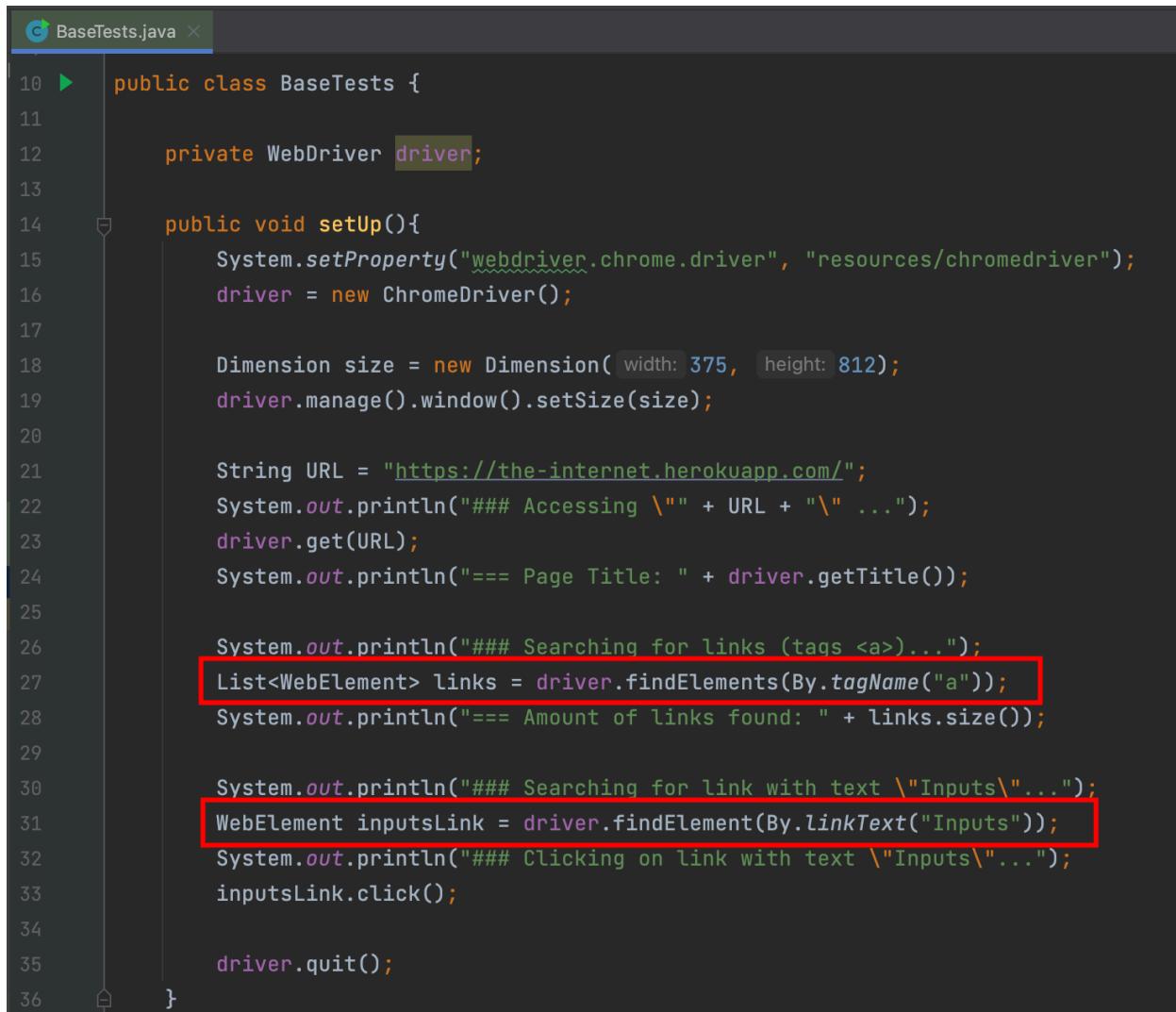
public class SecureAreaPage {
    private WebDriver driver;
    private By statusAlert = By.id("flash");

    public SecureAreaPage(WebDriver driver){ this.driver = driver; }

    public String getAlertText(){
        return driver.findElement(statusAlert).getText();
    }
}
```

Código Final
`SecureAreaPage.java`

Vamos revisar a classe “**BaseTests.java**“ que criamos anteriormente. Nesta classe usamos **findElements** e **findElement**:



```
10  public class BaseTests {  
11  
12      private WebDriver driver;  
13  
14      public void setUp(){  
15          System.setProperty("webdriver.chrome.driver", "resources/chromedriver");  
16          driver = new ChromeDriver();  
17  
18          Dimension size = new Dimension( width: 375, height: 812);  
19          driver.manage().window().setSize(size);  
20  
21          String URL = "https://the-internet.herokuapp.com/";  
22          System.out.println("### Accessing \" + URL + "\" ...");  
23          driver.get(URL);  
24          System.out.println("== Page Title: " + driver.getTitle());  
25  
26          System.out.println("### Searching for links (tags <a>) ...");  
27          List<WebElement> links = driver.findElements(By.tagName("a"));  
28          System.out.println("== Amount of links found: " + links.size());  
29  
30          System.out.println("### Searching for link with text \"Inputs\" ...");  
31          WebElement inputsLink = driver.findElement(By.linkText("Inputs"));  
32          System.out.println("### Clicking on link with text \"Inputs\" ...");  
33          inputsLink.click();  
34  
35          driver.quit();  
36      }  
}
```

Imagen 38: Uso dos métodos **findElement** e **findElements** na classe **BaseTests.java**

Estas são interações com o website. Porém, se olharmos para a estrutura do projeto, veremos que “**BaseTests.java**“ não está na pasta “**framework**”, mas na seção “**test**”:

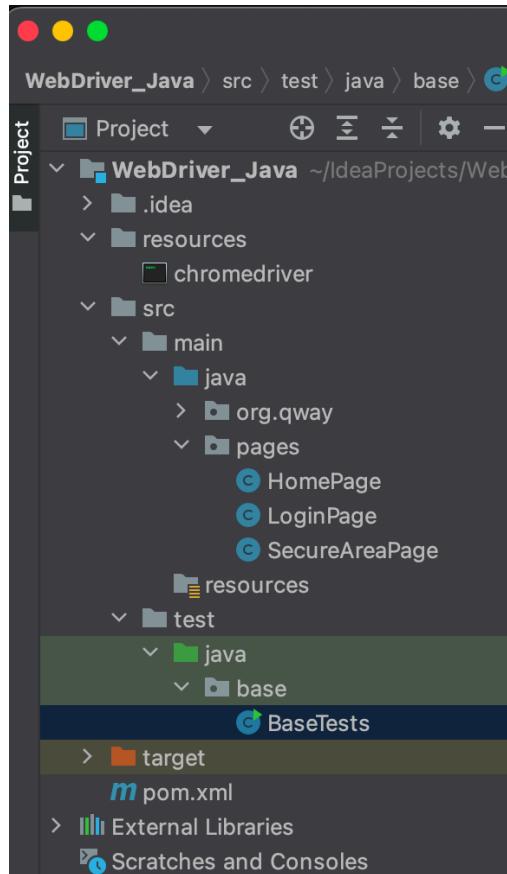


Imagen 39: Classe BaseTests.java na seção de “test”

Portanto, estes tipos de interações não são encorajadas neste nível de “**test**”, mas no nível de “**framework**”. Como estes métodos não são mais necessários para nosso teste, vamos remover este código.

Após abrir o website, vamos criar uma instância da classe “**HomePage.java**” que representa a página atual.

Vamos criar outra variável com acesso **protected** e depois entenderemos o motivo:

```
protected HomePage homePage;
```

Após a abertura da aplicação podemos inicializar esta variável criando uma instância desta página utilizando o mesmo driver utilizado. Isso será feito dentro do método “**setUp**”.

```
homePage = new HomePage(driver);
```

O que fizemos aqui foi fornecer à nossa camada de teste um identificador para a nossa aplicação e no próximo capítulo trataremos mais disso.

Capítulo 6 - Criando um Teste

No capítulo anterior criamos todas as classes que representam as páginas que acessaremos para fazer um teste de Login, identificamos os elementos com os quais teremos que interagir e os métodos que efetuarão as ações necessárias.

Para recordar o teste de Login, vejamos o fluxo seguinte:

1. Acessaremos o website <http://the-internet.herokuapp.com/> e clicaremos no link **Form Authentication**;
2. Preencheremos o username “**tomsmith**” e o password “**SuperSecretPassword!**”, e clicaremos no botão **Login**.
3. Verificaremos se o banner com o texto “**You logged into a secure area!**” foi apresentado

Como já temos todas as partes do framework necessárias para fazer isso, agora precisamos escrever o teste na pasta “**test**”.

6.1 Criando a classe de teste *LoginTests*

Temos o pacote “**base**”, mas criaremos outro pacote que deve representar o tipo de teste que faremos e, por isso, será chamado de “**login**”.

No pacote “**login**” criaremos uma classe chamada “**LoginTests.java**”.

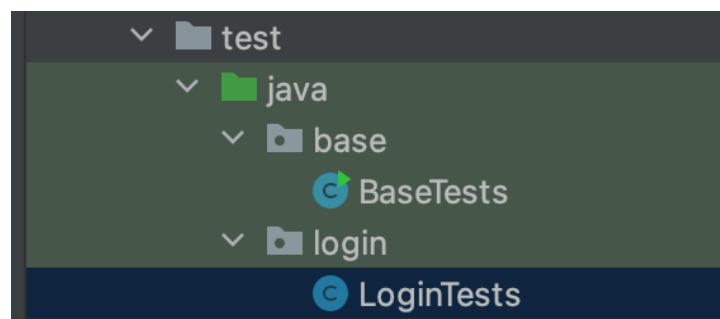


Imagen 40: Novo pacote “login” e classe “LoginTests.java”

Observamos que o nome da classe termina com a palavra “**Test**”. Isto não é necessário, mas um padrão que gosto de usar para sabermos rapidamente que se trata de uma classe de teste.

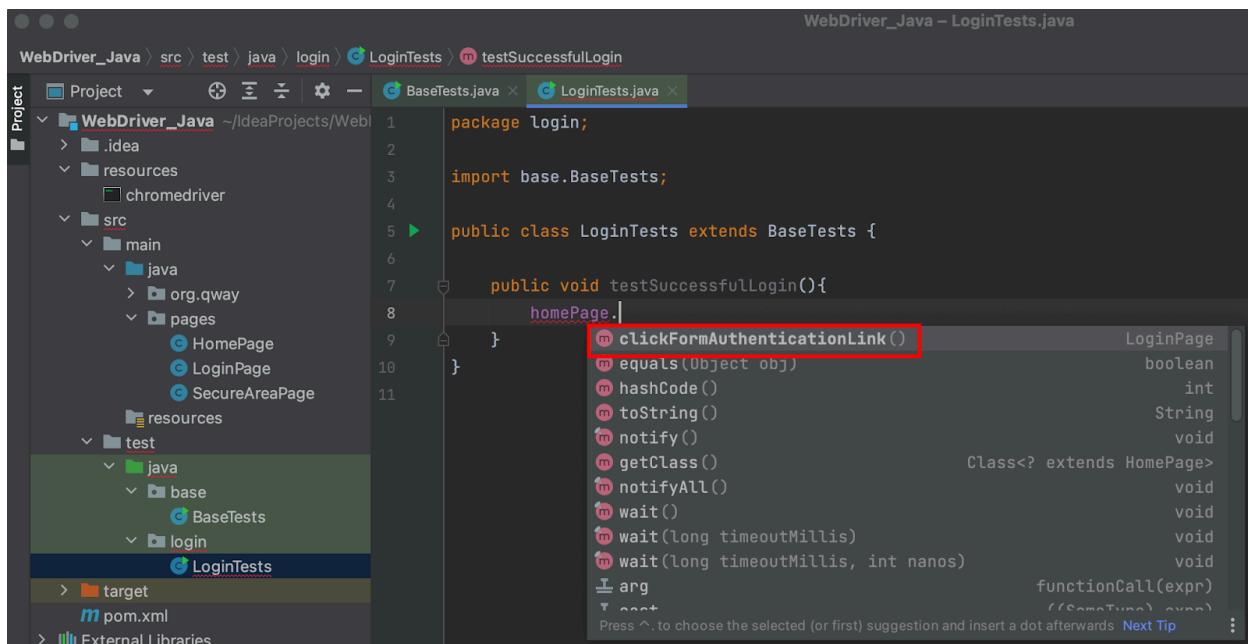
Usaremos esta classe para manter todos os testes relacionados à funcionalidade de login, tornando mais fácil encontrá-los. Novamente, não é uma regra, podemos separar em diferentes classes como preferir, mas para este curso vamos seguir este padrão.

Todas as nossas classes de teste estenderão a classe “**BaseTests.java**” já que é nela que fazemos o setup do browser instanciando o driver, acessando o website e instanciando o objeto **homePage**, e não precisaremos repetir isto em todas as classes de teste que serão criadas.

Vamos começar criando um método com acesso **public** e retorno **void**. Outro padrão que adotaremos será iniciar o nome deste método com o verbo “**test**” e finalizar com o que quer que estejamos testando.

testSuccessfulLogin é um bom nome para indicar o objetivo deste teste.

Como estendemos a classe **BaseTests**, já temos a página **homePage** e é onde está o link em que queremos clicar. Podemos usar este objeto que foi herdado e acessaremos o método **clickFormAuthenticationLink**:



```

WebDriver_Java - LoginTests.java
WebDriver_Java /src/test/java/login/LoginTests.java
Project BaseTests.java LoginTests.java
WebDriver_Java ~/ideaProjects/WebD
  .idea
  resources
    chromedriver
  src
    main
      java
        org.qway
        pages
          HomePage
          LoginPage
          SecureAreaPage
      resources
    test
      java
        base
          BaseTests
        login
          LoginTests
    target
    pom.xml
External Libraries

```

```

1 package login;
2
3 import base.BaseTests;
4
5 public class LoginTests extends BaseTests {
6
7     public void testSuccessfulLogin(){
8         HomePage.| clickFormAuthenticationLink();
9     }
10 }

```

Press ^, to choose the selected (or first) suggestion and insert a dot afterwards **Next Tip**

Imagen 41: Acessando os métodos da classe “HomePage”

Também podemos ver aqui que este método retorna um objeto **LoginPage**, pois quando executarmos este método seremos movidos para a página de Login e teremos acesso aos seus elementos e métodos. Este é o motivo pelo qual retornamos estes objetos.

Agora vamos criar um novo objeto para a página de Login e atribuir o retorno deste método:

```
public void testSuccessfulLogin() {
    LoginPage loginPage = homePage.clickFormAuthenticationLink();
}
```

Após acessarmos a página de Login vamos preencher o username, o password, e clicar no botão de Login. Isto irá retornar uma instância da classe **SecureAreaPage.java** que será atribuído ao objeto **secureAreaPage**:

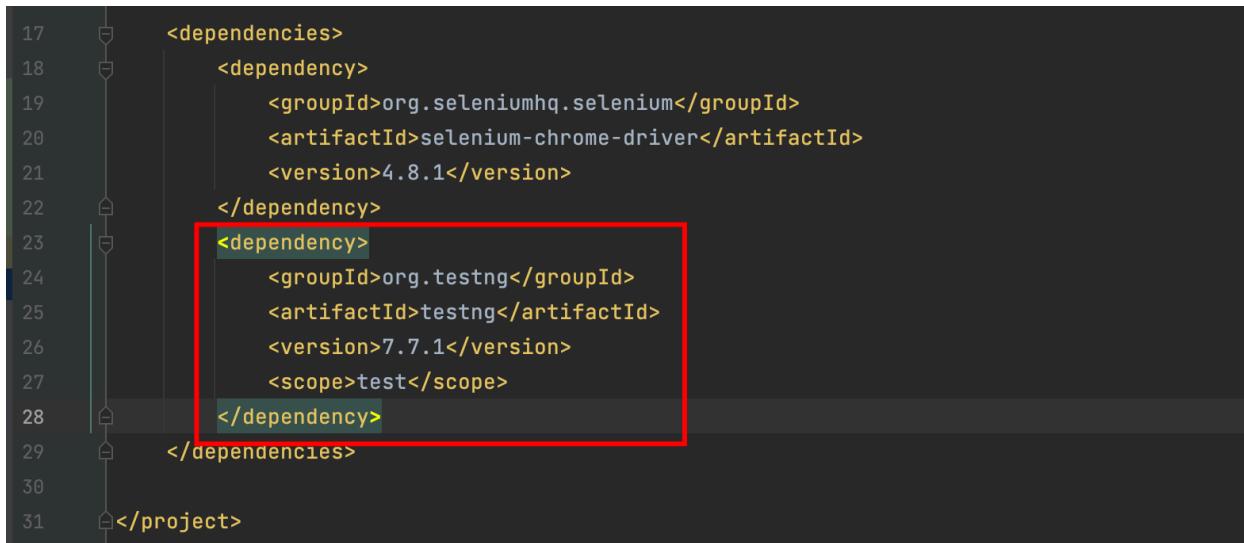
```
public void testSuccessfulLogin() {
    LoginPage loginPage = homePage.clickFormAuthenticationLink();
    loginPage.setUsername("tomsmith");
    loginPage.setPassword("SuperSecretPassword!");
    SecureAreaPage secureAreaPage = loginPage.clickLoginButton();
}
```

Com este objeto poderemos executar o método **getAlertText** e receberemos o texto do alerta, mas ainda não estaremos fazendo nada com isto. Precisamos ser capazes de verificar que este texto é o que esperamos que seja.

Selenium WebDriver não é uma ferramenta de teste! Ele é usado apenas para automatizar as ações no *browser*, então precisamos adicionar uma ferramenta de verificação ao nosso framework e podermos executar estes testes adequadamente.

Retornando ao arquivo **pom.xml**, vamos adicionar outra biblioteca de *assertion* junto à dependência do *Selenium*. Neste curso vamos adotar *TestNG* usando um *snippet* de código do repositório *Maven*:

<https://mvnrepository.com/artifact/org.testng/testng>



```

17 <dependencies>
18     <dependency>
19         <groupId>org.seleniumhq.selenium</groupId>
20         <artifactId>selenium-chrome-driver</artifactId>
21         <version>4.8.1</version>
22     </dependency>
23     <dependency>
24         <groupId>org.testng</groupId>
25         <artifactId>testng</artifactId>
26         <version>7.7.1</version>
27         <scope>test</scope>
28     </dependency>
29 </dependencies>
30
31 </project>

```

Imagen 42: Adicionando *TestNG* como uma dependência do projeto em “pom.xml”

Acesse novamente a view *Maven* do *IntelliJ* e confirme se a biblioteca adicionada está lá e, se necessário, atualize as bibliotecas do projeto:

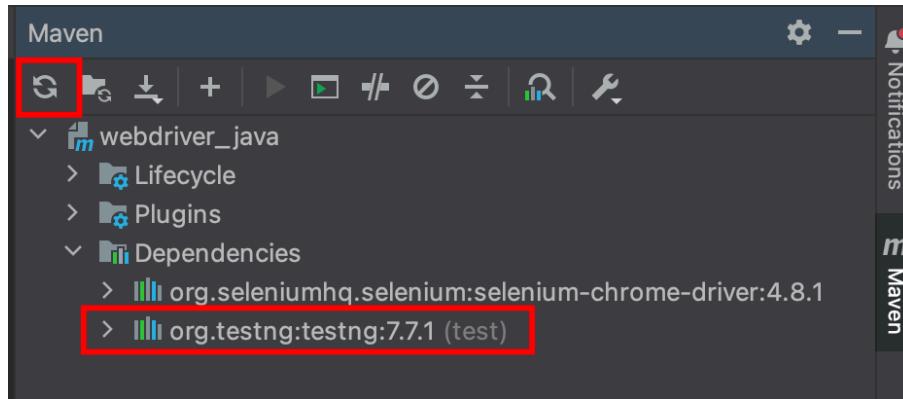


Imagen 43: Atualizando as bibliotecas adicionadas como dependências

Agora podemos adicionar alguns testes ao nosso projeto usando o método `assertEquals` para verificar se o texto recebido do banner é igual ao esperado.

Por fim, adicionamos mais um argumento que é opcional para apresentar uma mensagem caso essa assertion falhe.

```
assertEquals(secureAreaPage.getAlertText(),
    "You logged into a secure area! x",
    "Alert text is incorrect");
```

Veremos que este método também precisa ser importado:



Imagen 44: Importando o método da biblioteca adicionada como dependência

Ao importar este método teremos 2 opções para escolhermos, portanto iremos escolher `Assert.assertEquals`.

Este será o código final de nosso teste:

```

@Test
public void testSuccessfulLogin() {
    LoginPage loginPage = homePage.clickFormAuthenticationLink();
    loginPage.setUsername("tomsmith");
    loginPage.setPassword("SuperSecretPassword!");

    SecureAreaPage secureAreaPage = loginPage.clickLoginButton();
    assertEquals(secureAreaPage.getAlertText(),
        "You logged into a secure area!",
        "Alert text is incorrect");
}

```

Para executar este teste adicionamos uma *annotation* `@Test` do *TestNG*.

Agora que temos uma biblioteca de *assertion* há algumas coisas que também podemos fazer em nossa classe “**BaseTests.java**”, onde temos o método de `setUp`.

Queremos executar o método de `setUp` antes de cada uma das nossas classes de teste para garantir que o *browser* será iniciado e o website foi acessado, além de termos uma instância de uma classe de página para iniciar nossos testes.

Podemos adicionar a *annotation* `@BeforeClass` logo acima do método de `setUp` para executá-lo antes de cada classe de teste.

Também criaremos outro método para encerrar o driver após a execução de cada classe de teste, movendo o método `quit()` que atualmente está no `setUp`, caso contrário os testes nunca serão executados:

```

@AfterClass
public void tearDown() {
    driver.quit();
}

```

Neste método usamos a *annotation* `@AfterClass` que executa este método após cada classe de teste.

A forma como isto vai funcionar será a seguinte:

1. Primeiro será executado o método de `setUp` por causa da *annotation* `@BeforeClass`
2. Então serão executados os métodos que possuem a *annotation* `@Test`
3. Após a execução dos testes, o método de `tearDown` será executado por causa da *annotation* `@AfterClass` para encerrar o driver

Agora podemos remover o seguinte código que não será mais necessário, pois o TestNG fará a execução dos testes:

```
public static void main(String[] args) {
    BaseTests test = new BaseTests();
    test.setUp();
}
```

A classe **BaseTests.java** agora ficou assim:

```
package base;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import pages.HomePage;

public class BaseTests {

    private WebDriver driver;
    protected HomePage homepage;

    @BeforeClass
    public void setUp() {
        System.setProperty("webdriver.chrome.driver",
"resources/chromedriver");

        driver = new ChromeDriver();
        String URL = "https://the-internet.herokuapp.com/";
        System.out.println("### Accessing \"" + URL + "\" ...");
        driver.get(URL);

        homepage = new HomePage(driver);
    }

    @AfterClass
    public void tearDown() {
        System.out.println("### Encerrando o driver...");
        driver.quit();
    }
}
```

- Temos um *driver* e temos um *page object*;
- Temos o método de *setUp* que será executados antes de cada classe de teste e acessará o website a ser testado;
- Após acessar a aplicação, sabemos que estaremos na home page, então criamos uma nova instância dessa classe. Este é o motivo pelo qual o objeto *homepage* tem acesso *protected*, assim as classes de teste que herdaram a classe “**BaseTests.java**” terão acesso à este objeto;
- E então finalmente executaremos o método *tearDown* após cada classe de teste.

Agora vamos ver como ficou a classe **LoginTests.java**:

```
package login;

import base.BaseTests;
import org.testng.annotations.Test;
import pages.LoginPage;
import pages.SecureAreaPage;
import static org.testng.Assert.assertEquals;

public class LoginTests extends BaseTests {

    @Test
    public void testSuccessfulLogin() {
        LoginPage loginPage = homepage.clickFormAuthenticationLink();
        loginPage.setUsername("tomsmith");
        loginPage.setPassword("SuperSecretPassword!");
        SecureAreaPage secureAreaPage = loginPage.clickLoginButton();
        assertEquals(secureAreaPage.getAlertText(),
                    "You logged into a secure area! x",
                    "Alert text is incorrect");
    }
}
```

Código Final
LoginTests.java

- **LoginTests** estende **BaseTests** e herda desta classe;
- Temos um método de teste indicado pela annotation **@Test**;
- Temos o objeto **homepage** criado pela classe **HomePage** herdada da classe **BaseTests** e usaremos este objeto para acessar o método que fará o clique no link “**Form Authentication**”;
- Preenchemos username, password e clicamos no botão Login, nos levando para a página “**Secure Area**”;
- Obtemos o texto do banner e comparamos com o que esperamos que seja. O teste vai passar ou falhar baseado nesta igualdade.

Capítulo 7 - Executando o Teste

7.1 Via IDE

Para executar os testes via IDE, clique com o botão direito do mouse sobre o código e escolha a opção “**Run LoginTests**”. Como resultado desta execução teremos a seguinte falha:

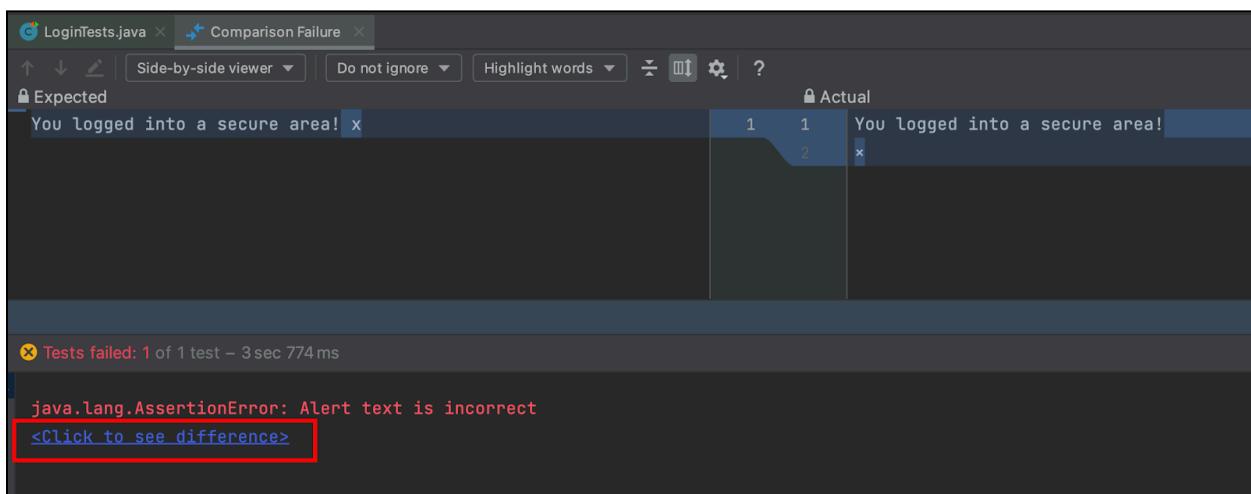


Imagen 45: Verificando a diferença do *AssertionError*

A mensagem de erro diz que temos um erro de *assertion* referente ao texto do banner. Clicando no link “**Click to see the difference**” vemos que esperávamos uma string com um espaço, mas ao inspecionamos este elemento no browser veremos que na verdade são 2 elementos diferentes:



Imagen 46: Conteúdo do banner

Poderíamos simplesmente copiar o conteúdo do resultado atual e usar como argumento do método *assertEquals* como resultado esperado, mas não queremos lidar com isto desta forma, queremos apenas verificar o texto do elemento *<div>* e não incluir o conteúdo da tag *<a>*.

Para resolver isso, mudaremos nossa `assertion`. Não precisamos usar apenas `assertEquals`, há outras `assertions` que podemos usar como `assertTrue`.

Vamos mudar o *import* de `assertEquals` para “`*`” para importar todos os métodos de `assertion`:

```
import static org.testng.Assert.*;
```

A `assertion` será alterada também, usando `assertTrue` para verificar se o texto da `<div>` contém o texto esperado:

```
assertTrue(  
    secureAreaPage.getAlertText()  
    .contains("You logged into a secure area!")  
    ."Alert text is incorrect");
```

Pronto, agora vamos executar o teste novamente e confirmar que ele passará.

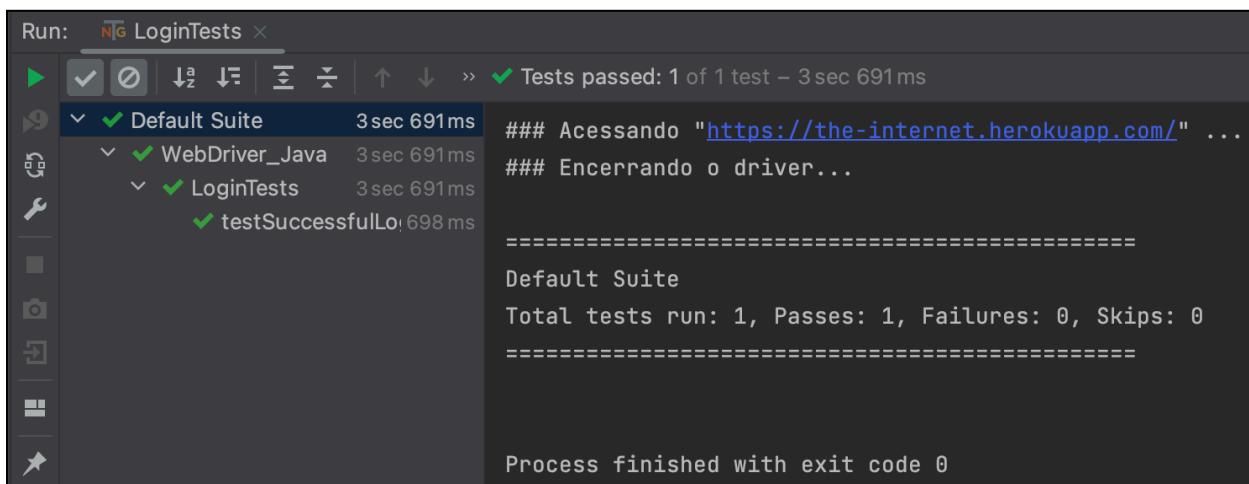


Imagen 47: Resultado da execução do teste

7.2 Via Terminal

Lembra que ao criarmos o projeto escolhemos Maven como Build System? Para executar seus testes via Terminal usaremos o seguinte comando Maven:

mvn test

Porém, dependendo da versão de Chrome que estiver usando, ele não vai funcionar ainda, pois quando nosso método de `setUp` abre o ChromeDriver em `@BeforeClass` e antes de executar o método `goHome` em `@BeforeMethod` ele tenta estabelecer uma conexão de

websocket que gera a exceção “**ConnectionFailedException**” por padrão. No capítulo 18 veremos como contornar isso com “**ChromeOptions**”.

Até aqui vimos os métodos disponíveis no *WebElement* como ***click()*** e ***sendKeys()*** que já utilizamos para interagir ativamente com o browser. Além disso usamos o ***getText()*** que nos permite ler o conteúdo dos elementos, mas não necessariamente executando ações.

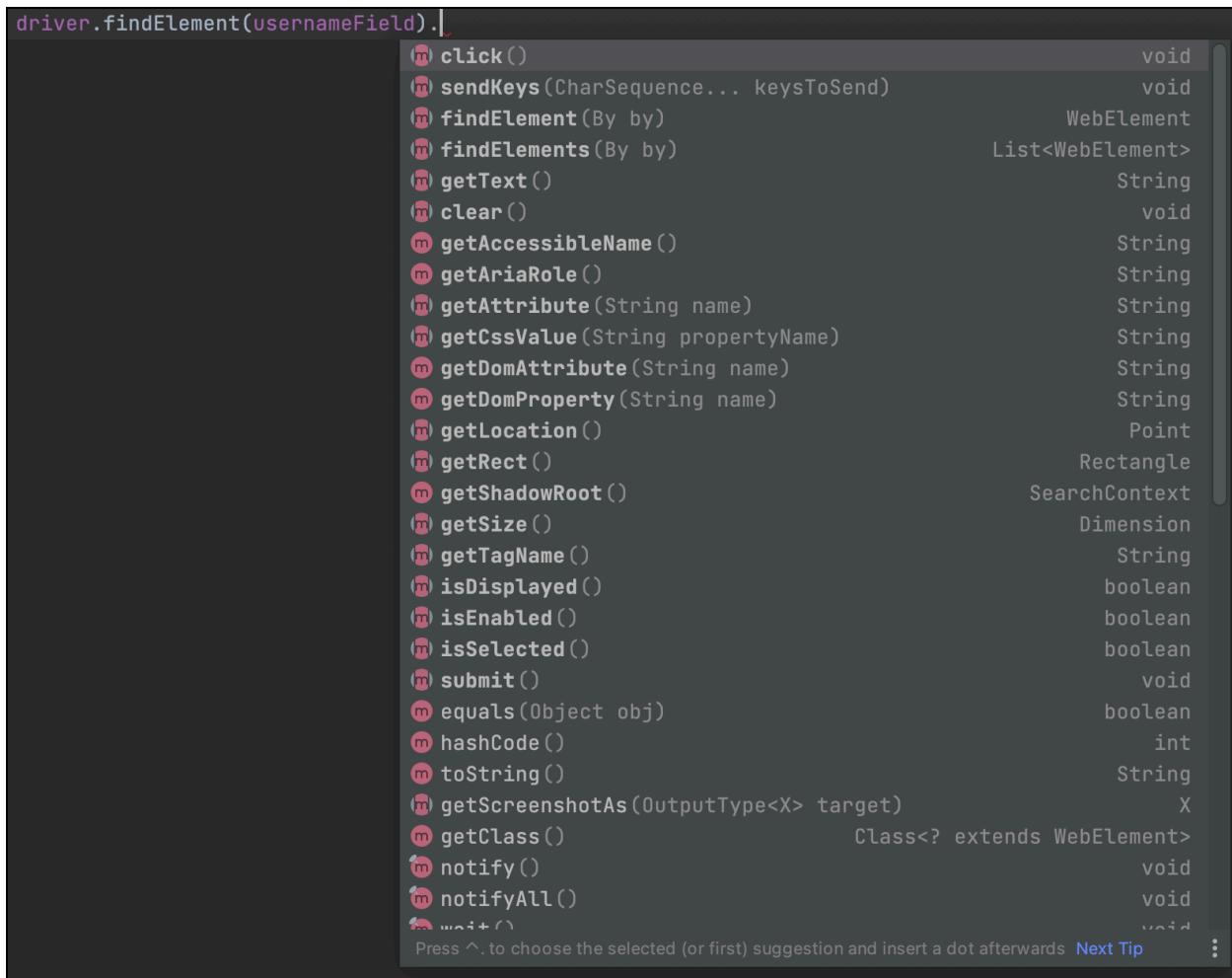


Imagen 48: Métodos de um WebElement

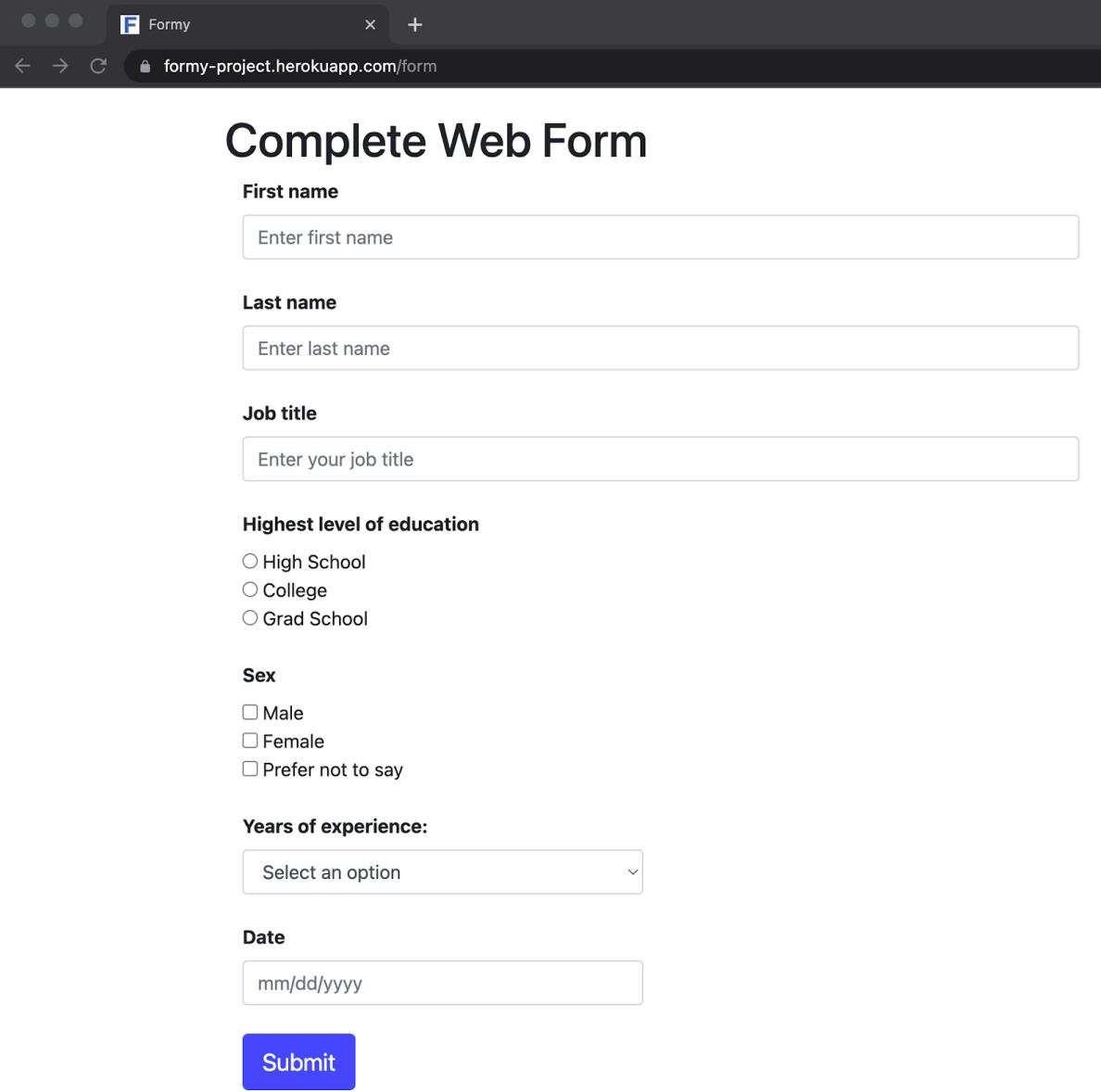
Outros métodos que também estão disponíveis:

- ***isDisplayed()***, que vai nos informar se um elemento está realmente apresentado na página
- ***clear()*** vai limpar um campo
- Alguns métodos que obtém conteúdo como atributos, valor de CSS, localização na página, etc
- ***getRect()***, que retorna um retângulo com a localização e as dimensões de um elemento
- ***getSize()*** que retorna as dimensões
- ***getTagName()*** que retorna o nome da tag

- Podemos verificar se um elemento está *enabled* com `isEnabled()`
- Podemos verificar se um elemento está *selected* com `isSelected()` (funciona com radio *buttons* e *checkboxes*)

Para treinar estes e outros métodos em um form com muitos tipos diferentes de *WebElements*, use o seguinte site:

<https://formy-project.herokuapp.com/form>



The screenshot shows a web browser window titled "F Formy" with the URL "formy-project.herokuapp.com/form". The main content is a form titled "Complete Web Form". It includes the following fields:

- First name:** A text input field with placeholder "Enter first name".
- Last name:** A text input field with placeholder "Enter last name".
- Job title:** A text input field with placeholder "Enter your job title".
- Highest level of education:** A group of three radio buttons labeled "High School", "College", and "Grad School".
- Sex:** A group of three checkbox options labeled "Male", "Female", and "Prefer not to say".
- Years of experience:** A dropdown menu with placeholder "Select an option".
- Date:** A text input field with placeholder "mm/dd/yyyy".

A large blue "Submit" button is located at the bottom left of the form area.

Imagen 49: Formulário para praticar métodos de WebElement

Capítulo 8 - Interagindo com elementos Dropdown

No capítulo anterior conhecemos o site “<https://formy-project.herokuapp.com/form>” que pode ser usado para praticar a seleção e a interação com *WebElements*. Há algo neste `<form>` que é diferente de alguns elementos vistos até agora, o *Dropdown*.

Menus *Dropdown* não são representados pela classe *WebElement*. Eles tem sua própria classe especial que conhiceremos nessa seção.

Acessando novamente o site “**the-internet**” trataremos de um novo cenário que automatizaremos agora:

<https://the-internet.herokuapp.com/dropdown>

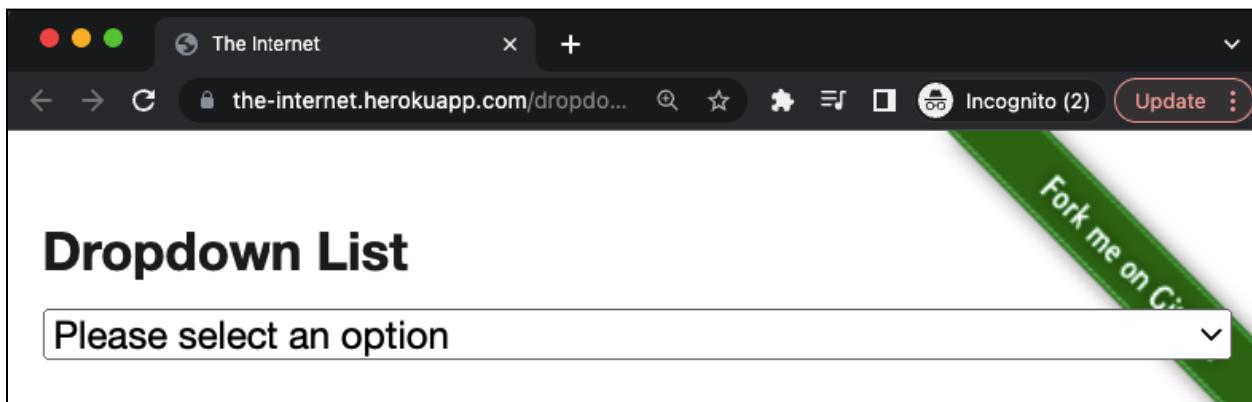


Imagen 50: Dropdown

1. Na Home Page, clicaremos no link “**Dropdown**”;
2. Selecioneamos uma opção no *dropdown*;
3. Então confirmamos que esta opção está selecionada.

Teremos que interagir com 2 páginas, mas como já temos uma classe para a “**Home Page**” apenas precisaremos criar a página “**Dropdown**”.

8.1 Criando a classe de framework *DropdownPage*

Em nosso framework, criaremos dentro do package `pages` uma nova classe Java chamada “**DropdownPage.java**”. Nesta classe criaremos o WebDriver e seu método construtor:

```

public class DropdownPage {
    private WebDriver driver;

    public DropdownPage(WebDriver driver) {
        this.driver = driver;
    }
}

```

Embora já tenhamos uma classe “**HomePage.java**”, ela não tem um método para clicar no link “**Dropdown**”, então precisamos adicioná-lo à esta classe.

Porém, dentro da classe “**HomePage.java**” já existe um localizador de um link e um método para clicar nele. Cada vez que tivermos que criar um localizador de um novo link teremos que criar um outro método, mas como temos muitos links nosso código ficaria muito longo e repetitivo. Aqui podemos começar a tornar nossa classe um pouco mais flexível

Ao invés de criar um novo link, podemos identificar que eles seguem um mesmo padrão, com o texto do link como variável. Então vamos criar um método genérico chamado **clickLink()**.

Ele não irá mais retornar um novo *page object*, o que será explicado em seguida. Este método apenas receberá uma *String* e clicará neste link.

Vamos utilizar **driver.findElement** com o método **By.LinkText()** que receberá a *String* **LinkText** recebida como parâmetro na chamada do método, e por fim adicionamos o método de **click()**:

```

public void clickLink(String linkText) {
    driver.findElement(By.linkText(linkText)).click();
}

```

Agora podemos atualizar o método **clickFormAuthenticationLink()**, onde ao invés de usarmos **findElement().click()** usaremos o novo método **clickLink(String LinkText)** fornecendo a *String* com o texto do link:

```

public LoginPage clickFormAuthenticationLink() {
    clickLink("Form Authentication");
    return new LoginPage(driver);
}

```

Depois de executar o método **clickLink(String LinkText)** ele retornará uma instância da classe “**LoginPage.java**”. Este é o motivo pelo qual não precisamos retornar nada no método, que será genérico para poder ser usado para todos os links. Como não sabemos o que cada link vai retornar, não precisamos nos preocupar com isso neste método.

Se quisermos, podemos modificar o acesso deste método tornando **private** para evitar que nossos testes o utilizem ao invés dos métodos que retornam os *page objects*.

Agora podemos remover o localizador do link criado anteriormente:

```
private By formAuthenticationLink = By.linkText("Form Authentication");
```

E criar o método para clicar no link “**Dropdown**” e retornar a respectiva página.:

```
public DropdownPage clickDropdownLink() {
    clickLink("Dropdown");
    return new DropdownPage(driver);
}
```

Este método resolve a primeira ação a ser tomada neste cenário de teste: clicamos no link “**Dropdown**” e somos levados à página “**Dropdown**”.

Agora, na classe “**DropdownPage.java**” precisamos adicionar os localizadores de mais alguns campos e métodos para executar o restante das ações teste cenário de teste.

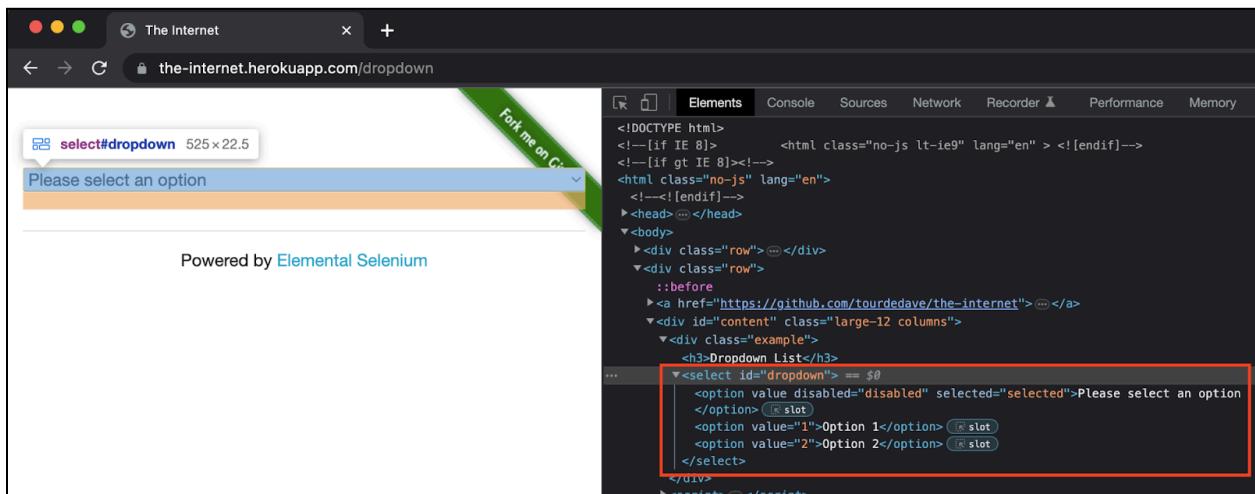


Imagen 51: Elemento dropdown com a tag <select>

Precisamos de um seletor para este elemento *dropdown* que possui um id **dropdown**:

```
private By dropdown = By.id("dropdown");
```

Agora vamos criar um método que vai selecionar uma opção no *dropdown*. Em nosso framework queremos ser capazes de selecionar qualquer opção, então este método não selecionará apenas a primeira ou a segunda opção e, para isso, precisaremos de uma variável.

```
public void selectFromDropDown(String option) {}
```

Ao invés de usarmos dentro deste método `driver.findElement` que retornaria um `WebElement` (mas o dropdown não é um), precisamos adicionar uma classe específica para elementos `dropdown` chamada `Select`. Porém ainda não adicionamos esta dependência ao nosso projeto, por isso vamos voltar ao arquivo `pom.xml` e adicionar outra dependência `Selenium` aqui

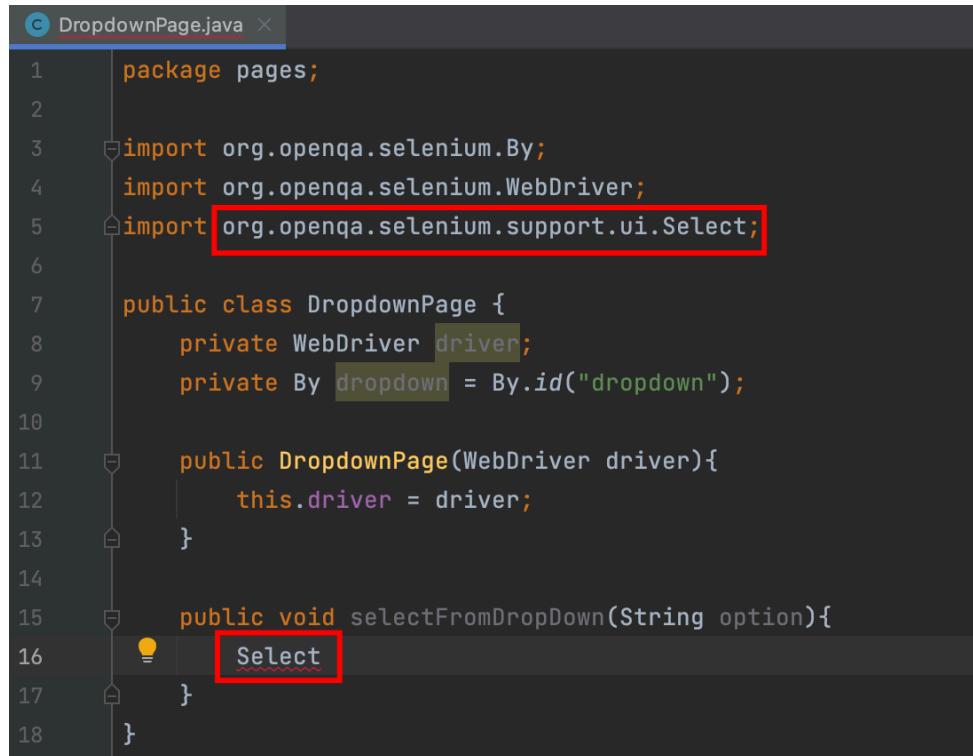
Retornando ao repositório de bibliotecas `Maven` vamos procurar por “`selenium support`” e copiar o seu snippet de código para o arquivo `pom.xml` e adicionar outra dependência `Selenium`:

<https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-support>

```
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-chrome-driver</artifactId>
        <version>4.8.1</version>
    </dependency>
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.7.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-support</artifactId>
        <version>4.8.1</version>
    </dependency>
</dependencies>
```

Imagen 52: Adicionando a biblioteca “selenium-support” ao projeto

De volta a nossa classe “`DropdownPage.java`”, agora podemos criar um objeto `Select` a importar a biblioteca do pacote `org.openqa.selenium.support.ui`:



```

1 package pages;
2
3 import org.openqa.selenium.By;
4 import org.openqa.selenium.WebDriver;
5 import org.openqa.selenium.support.ui.Select;
6
7 public class DropdownPage {
8     private WebDriver driver;
9     private By dropdown = By.id("dropdown");
10
11     public DropdownPage(WebDriver driver){
12         this.driver = driver;
13     }
14
15     public void selectFromDropDown(String option){
16         Select dropdownElement;
17     }
18 }
```

Imagen 53: Criando o objeto “Select”

```

public void selectFromDropDown(String option){
    Select dropdownElement = new Select(driver.findElement(dropdown));
}
```

Agora que temos o objeto `dropdownElement`, podemos ver que esta instância da classe `Select` disponibiliza outros métodos que não estavam disponíveis para os objetos que eram instâncias da classe `WebElement`:



<code>m deselectAll()</code>	<code>void</code>
<code>m deselectByIndex(int index)</code>	<code>void</code>
<code>m getWrappedElement()</code>	<code>WebElement</code>
<code>m deselectByValue(String value)</code>	<code>void</code>
<code>m getAllSelectedOptions()</code>	<code>List<WebElement></code>
<code>m deselectByVisibleText(String text)</code>	<code>void</code>
<code>m getFirstSelectedOption()</code>	<code>WebElement</code>
<code>m getOptions()</code>	<code>List<WebElement></code>
<code>m hashCode()</code>	<code>int</code>
<code>m isMultiple()</code>	<code>boolean</code>
<code>m selectByIndex(int index)</code>	<code>void</code>
<code>m selectByValue(String value)</code>	<code>void</code>
<code>m selectByVisibleText(String text)</code>	<code>void</code>

Imagen 54: Métodos da classe “Select”

Dentro da tag `<select>` temos tags `<options>`, sendo que cada uma delas possui um atributo `value` (1 e 2) e um texto visível (“`Option 1`” e “`Option 2`”). Portanto poderíamos selecionar estas opções por `value`, texto visível, ou por `index`.

Para este método especificamente receberemos como parâmetro a `String option` com o texto visível e usaremos o método `selectByVisibleText()` da classe `Select`.

```
public void selectFromDropDown(String option) {  
    Select dropdownElement = new Select(driver.findElement(dropdown));  
    dropdownElement.selectByVisibleText(option);  
}
```

Isto será o suficiente para selecionar uma opção no dropdown.

Agora precisamos de um método que receba o texto da opção selecionada para que possamos confirmar que ela foi selecionada corretamente. Este método precisará nos retornar uma String:

```
public String getSelectedOption() {}
```

Agora precisamos criar novamente o elemento `dropdownElement`, o que poderemos evitar criando um pequeno método com acesso `private` que retornará este elemento.

O acesso será de acesso `private`, pois não precisará ser acessado pelos testes e apenas será usado como um helper nesta classe.

Este método retornará uma instância da classe `Select` que chamaremos de `findDropDownElement`. Este objeto será utilizado no método `selectFromDropDown()` que foi criado anteriormente e deverá ser atualizado da seguinte forma:

```
public void selectFromDropDown(String option) {  
    findDropDownElement().selectByVisibleText(option);  
}
```

Agora, no novo método `getSelectedOption()` criado podemos utilizar novamente este objeto mas desta vez digitando `get` para verificar alguns dos métodos disponíveis. Vamos obter todas as opções selecionadas com o método `getALLSelectedOptions()`, assim poderemos verificar que apenas a opção que selecionamos está selecionada:

```

public void selectFromDropDown(String option){
    findDropDownElement().selectByVisibleText(option);
}

public String getSelectedOption(){
    findDropDownElement().get
}
private Select findDrop
    return new Select(d
}

```

m **getOptions()** List<WebElement>
m **getAllSelectedOptions()** List<WebElement>
m **getFirstSelectedOption()** WebElement
m **getWrappedElement()** WebElement
m **getClass()** Class<? extends Select>
Press ⇨ to insert, ⌘ to replace Next Tip

Imagen 55: Métodos da classe “Select” iniciados por “get”

Então ao invés de retornar uma **String**, vamos alterar o retorno para uma lista de **Strings**:

```

public List<String> getSelectedOption() {
    findDropDownElement().getAllSelectedOptions();
}

```

Receberemos deste método uma lista de **WebElements**, mas o que queremos são as **Strings** com o texto que deverá ser retornado para o teste realizar a **assertion**. Precisamos acrescentar mais algumas linhas de código para extrair as **Strings** de texto e armazená-las em uma lista, começando com uma lista de **webElement**.

O próximo passo será criar uma **stream** que nos retornará um **map**. Dentro do **map** usaremos uma expressão Lambda dizendo que para cada elemento dessa lista queremos tomar uma ação que será o método **getText()**.

Assim que obtivermos o texto de cada elemento, deverão ser adicionados a uma nova lista de **Strings**. Como agora alteramos nosso código para retornar uma lista e não apenas uma **String**, vamos mudar o nome do método de **getSelectedOption()** para **getSelectedOptions()**.

O código final da classe **DropdownPage.java** ficará assim:

```

package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.Select;
import java.util.List;
import java.util.stream.Collectors;

public class DropdownPage {

```

Código Final
DropdownPage.java

```

private WebDriver driver;
private By dropdown = By.id("dropdown");

public DropdownPage(WebDriver driver) {
    this.driver = driver;
}

public void selectFromDropDown(String option) {
    findDropDownElement().selectByVisibleText(option);
}

public List<String> getSelectedOptions() {
    List<WebElement> selectedElements =
findDropDownElement().getAllSelectedOptions();
    return
selectedElements.stream().map(e->e.getText()).collect(Collectors.toList());
}

private Select findDropDownElement() {
    return new Select(driver.findElement(dropdown));
}
}

```

Agora podemos seguir adiante e criar nosso teste, já que temos todos os métodos de nosso framework disponíveis.

8.2 Criando a classe de teste *DropdownTests*

Já que este não é um teste de *Login* e nem pertence ao escopo da classe de teste “**BaseTests.java**”, vamos criar um novo package dentro do diretório *java* chamado *dropdown*. Em seguida criaremos uma nova classe de teste chamada “**DropdownTest.java**” que vai herdar a classe “**BaseTests.java**” da mesma forma como as demais classes de teste.

Nesta classe usaremos a annotation **@Test** e criaremos um novo método de teste chamado **testSelectOption()**.

Sabemos que temos acesso ao objeto *homePage*, que representa a página onde estamos quando iniciamos um teste antecedido pelo método *setUp*.

Em seguida queremos executar o método **clickDropdownLink()** que nos levará para a página *Dropdown* e nos retornará uma instância da classe “**DropdownPage.java**” que será armazenada em uma variável chamada *dropdownPage*:

```

@Test
public void testSelectOption() {
    var dropdownPage = homePage.clickDropdownLink();
}

```

Agora vamos selecionar a opção “Option 1” em nosso dropdown:

```
@Test
public void testSelectOption() {
    var dropdownPage = homePage.clickDropdownLink();
    dropdownPage.selectFromDropdown("Option 1");
}
```

Agora temos que confirmar que a opção correta foi selecionada. Através do objeto `dropdownPage` podemos obter a opção selecionada com o método `getSelectedOptions()` que vai retornar uma lista.

```
@Test
public void testSelectOption() {
    var dropdownPage = homePage.clickDropdownLink();
    dropdownPage.selectFromDropdown("Option 1");
    var selectedOptions = dropdownPage.getSelectedOptions();
}
```

Agora faremos as seguintes *assertions*.

1. Que a lista contém apenas um item, significando que o dropdown possui apenas uma opção selecionada
2. Que a opção selecionada possui o texto “Option 1”

Para a primeira assertion vamos usar `assertEquals` para garantir que a quantidade de opções selecionadas é igual à 1. Para obter esta quantidade de itens da lista usaremos `selectedOptions.size()`. Em caso de erro, queremos que a mensagem “**Incorrect number of selections**” seja apresentada:

```
assertEquals(selectedOptions.size(), 1, "Incorrect number of selections");
```

Caso essa assertion funcione, se houver apenas uma opção selecionada, queremos fazer a segunda assertion para garantir que esta opção selecionada possui o texto correto. Para isso, precisaremos usar a String de texto novamente, então vamos evitar essa repetição criando uma variável com este texto e modificar o trecho de código que seleciona esta opção:

```
String option = "Option 1";
dropdownPage.selectFromDropdown(option);
```

Agora vamos utilizar esta variável na segunda assertion utilizando `assertTrue` para verificar que esta lista contém o mesmo texto desta variável, e em caso de erro, queremos que a mensagem “**Option not selected**” seja apresentada:

```
assertTrue(selectedOptions.contains(option), "Option not selected");
```

O código final desta classe de teste ficará assim::

```
package dropdown;

import base.BaseTests;
import org.testng.annotations.Test;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertTrue;

public class DropdownTest extends BaseTests {

    @Test
    public void testSelectOption() {
        var dropdownPage = homePage.clickDropdownLink();
        String option = "Option 1";
        dropdownPage.selectFromDropdown(option);
        var selectedOptions = dropdownPage.getSelectedOptions();
        assertEquals(selectedOptions.size(), 1, "Incorrect number of
selections");
        assertTrue(selectedOptions.contains(option), "Option not selected");
    }
}
```

Código Final
DropdownTest.java

Agora vamos executar o teste e verificar seu resultado:

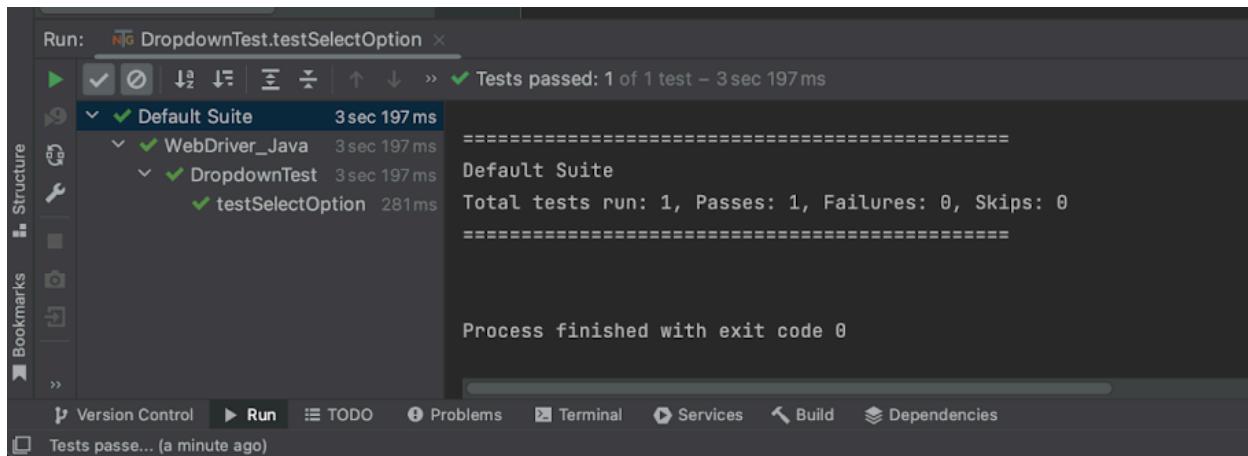


Imagen 56: Teste “testSelectOption” executado com sucesso

Capítulo 9 - Interações avançadas

No último capítulo, aprendemos como interagir com diferentes tipos de elementos da web. Aprendemos como enviar texto para eles, como clicar neles, como obter texto deles, e vimos que haviam vários outros métodos também.

No entanto, existem vários tipos mais avançados de interações que ainda não abordamos, então veremos alguns deles neste capítulo.

9.1 Hovers

Em nosso website de teste temos um link chamado **Hovers**, e nessa página temos 3 imagens:

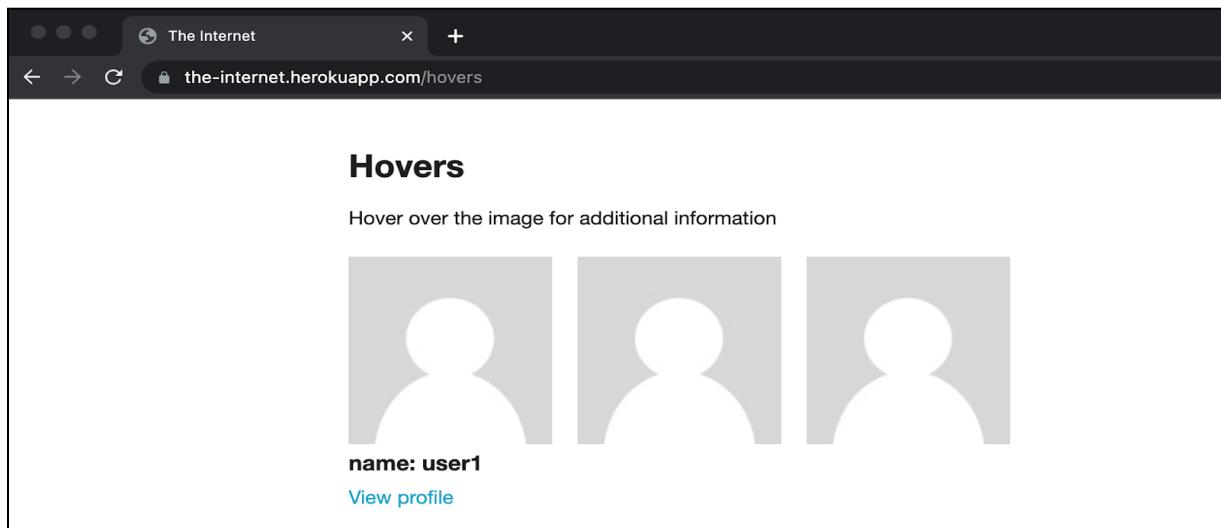


Imagen 57: Página de Hovers

Se passarmos o mouse sobre eles, há um nome e um link que aparecem e cada um é diferente. No entanto, como podemos ver no objeto *WebElement*, dentre os métodos disponíveis não há nenhum para passar o mouse sobre um elemento (fazer *hover*). No entanto, existe uma classe adicional no pacote *Selenium WebDriver* que nos permitirá trabalhar com isso. Vamos criar um novo teste que irá:

1. Clicar no link da página Hovers na página inicial
2. Fazer *hover* sobre a primeira imagem
3. Confirmar que o nome “**user1**” apareça, bem como o link “**View profile**”

O primeiro passo é clicar no link **Hovers** na página inicial. Para isso vamos adicionar um novo método à classe “**HomePage.java**” para clicar no link Hovers.

Vamos criar uma cópia do método ***clickDropdownLink()*** e alterar o seguinte:

- O nome para ***clickHoversLink()***
- O tipo do objeto que vai ser retornado que será uma instância da classe “**HoversPage.java**” que também iremos criar
- O texto do link à ser clicado para “**Hovers**”

```
public HoversPage clickHoversLink() {
    clickLink("Hovers");
    return new HoversPage(driver);
}
```

Em seguida vamos criar a classe java **HoversPage** declarando o WebDriver e com um método construtor que criará uma instância dessa classe:

```
package pages;

import org.openqa.selenium.WebDriver;

public class HoversPage {

    private WebDriver driver;

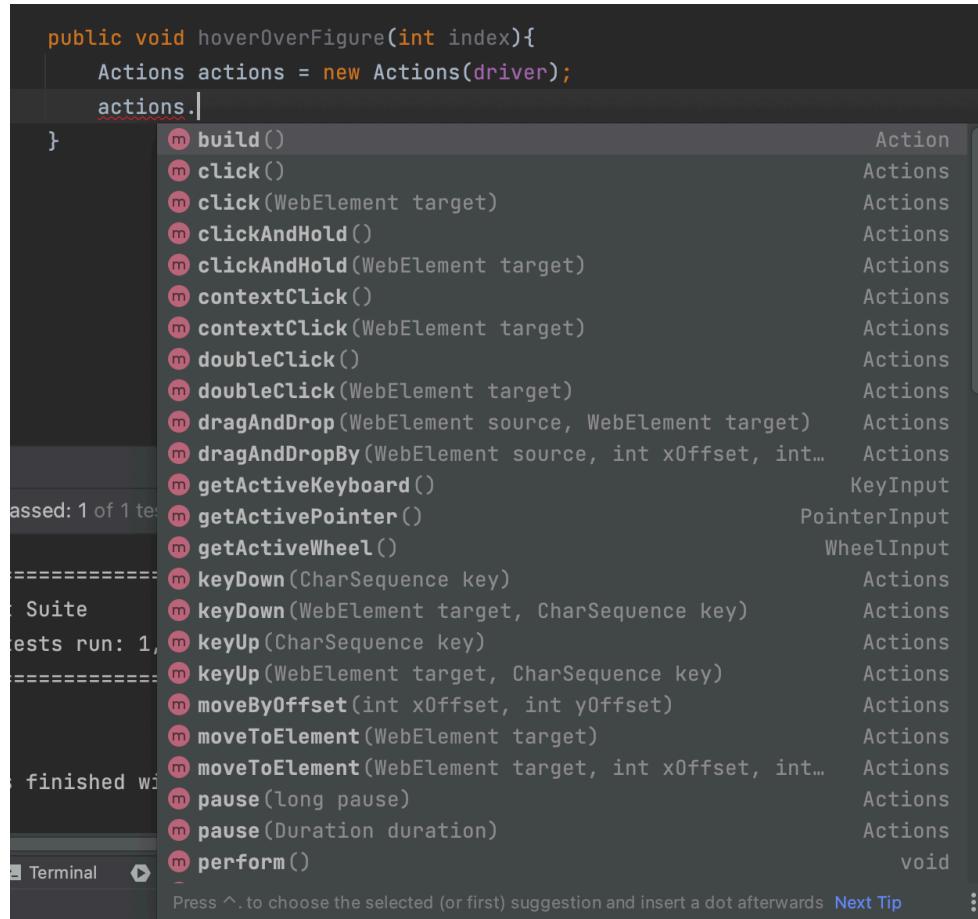
    public HoversPage(WebDriver driver) {
        this.driver = driver;
    }
}
```

Conforme mencionado anteriormente, o *WebElement* não possui um método para fazer *hover*. No entanto, existe uma classe chamada **Actions** com muitas ações avançadas que você pode usar para esse tipo de coisa.

Vamos criar um método que diga ***hoverOverFigure*** e como há três figuras nesta classe, podemos permitir que elas passem em um índice para indicar sobre qual dessas figuras você gostaria de passar o mouse. Neste método precisaremos de um objeto **Actions**. Vamos nomear esse objeto como ***actions*** e importaremos essa biblioteca do pacote **selenium.interactions**. **Actions** receberá como argumento um objeto *WebDriver*, então passaremos o ***driver*** que temos.

```
public void hoverOverFigure(int index) {
    Actions actions = new Actions(driver);
}
```

Agora que temos o objeto **Actions**, veremos que ele possui vários métodos:



```

public void hoverOverFigure(int index){
    Actions actions = new Actions(driver);
    actions.|_
}   m build()                               Action
    m click()                                Actions
    m click(WebElement target)               Actions
    m clickAndHold()                         Actions
    m clickAndHold(WebElement target)         Actions
    m contextClick()                          Actions
    m contextClick(WebElement target)          Actions
    m doubleClick()                           Actions
    m doubleClick(WebElement target)           Actions
    m dragAndDrop(WebElement source, WebElement target) Actions
    m dragAndDropBy(WebElement source, int xOffset, int... Actions
    m getActiveKeyboard()                   KeyInput
    m getActivePointer()                   PointerInput
    m getActiveWheel()                   WheelInput
    m keyDown(CharSequence key)             Actions
    m keyDown(WebElement target, CharSequence key) Actions
    m keyUp(CharSequence key)              Actions
    m keyUp(WebElement target, CharSequence key) Actions
    m moveByOffset(int xOffset, int yOffset) Actions
    m moveToElement(WebElement target)       Actions
    m moveToElement(WebElement target, int xOffset, int... Actions
    m pause(long pause)                   Actions
    m pause(Duration duration)            Actions
    m perform()                            void

```

Press ^ to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

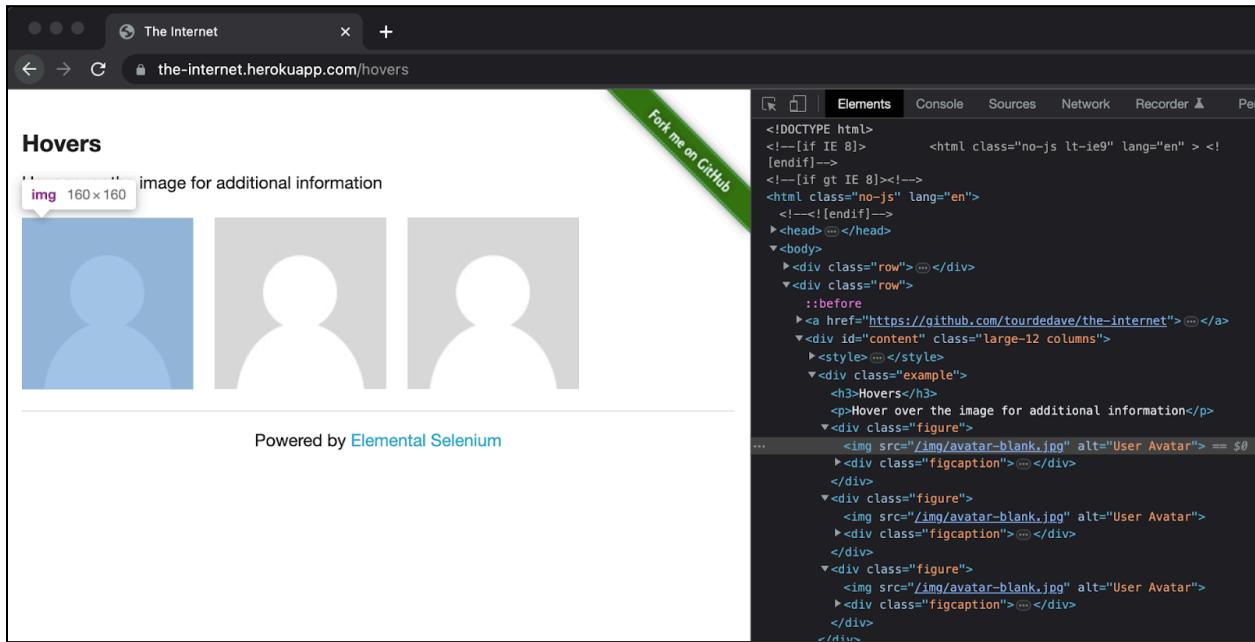
Imagen 58: Métodos do objeto Actions

Um deles é o **moveToElement**, e é este que vamos usar para passar o mouse sobre um elemento. Ele vai mover o mouse para esse elemento.

Outros métodos interessantes que temos são:

- **dragAndDrop**
- **click** e **clickAndHold**
- **contextClick** (para fazer um clique com o botão direito)
- **doubleClick**
- **keyDown** e **keyUp**
- **pausa**
- **release**
- **tick**
- **sendKeys**, etc

Para usar o método **moveToElement** precisamos localizar o elemento sobre o qual faremos *hover*. Vamos inspecionar a tag **** e veremos que ela tem um atributo **alt="User Avatar"**:



como argumento `index`. Porém, o método `get()` deverá receber um índice que começa em 0, portanto precisaremos fazer essa operação de subtração.

```
public void hoverOverFigure(int index) {
    WebElement figure = driver.findElements(figureBox).get(index - 1);
    Actions actions = new Actions(driver);
}
```

Agora vamos usar o método `moveToElement` informando o `WebElement figure`. Apenas isso não fará a ação porque a classe `Actions` usa o que é chamado de *builder pattern*. Este padrão permite o encadeamento de várias chamadas de método e, em seguida, executá-las todas de uma vez. Portanto, para essa ação, adicionamos o método `perform()`.

```
public void hoverOverFigure(int index) {
    WebElement figure = driver.findElements(figureBox).get(index - 1);
    Actions actions = new Actions(driver);
    actions.moveToElement(figure).perform();
}
```

Depois de fazer *hover* sobre a imagem a sua legenda será apresentada com o nome de usuário e com o link “[View profile](#)”. Se utilizarmos o atributo `class="figcaption"` poderemos obter o `WebElement` da `<div>`, mas queremos evitar passar `WebElements` para os métodos de teste, então podemos criar uma classe para modelar este elemento e dar acesso ao texto e ao link.

No framework poderíamos criar outra página para a legenda da figura. No entanto, a legenda da figura não é uma página, então não faz sentido fazer isso. Ainda faz parte da “**HoversPage**”. Poderíamos criar como um modelo ou um componente, de modo que seria como outro pacote sob o framework, e isso pode ser uma boa prática se a legenda da figura aparecer em mais de uma página. Como até agora só está aparecendo aqui, o que faremos é criar uma classe interna dentro da classe “**HoversPage**” para a legenda da figura e chamá-la de `FigureCaption`.

Nesta classe interna será necessário um construtor que aceite este `WebElement`, “`figcaption`”. Vamos adicioná-lo e criaremos um `WebElement` para isso também, e então podemos atribuí-lo.

```
public class FigureCaption{
    private WebElement caption;

    public FigureCaption(WebElement caption) {
        this.caption = caption;
    }
}
```

Quando fizermos *hover* sobre a figura, vamos instanciar este elemento “`caption`” e retornaremos um objeto `FigureCaption` de volta aos nossos testes.

Já temos a `<div>` representada pelo `WebElement figure`, e poderemos encontrar elementos filhos ou descendentes apenas fazendo um `findElement` desse elemento. Então, ao invés de usar `driver.findElement`, que irá buscá-lo em todo o DOM, vamos fazer `figure.findElement`, que irá encontrá-lo apenas dentro desta `<div>`.

Observe que a legenda aparece no DOM, mesmo que não esteja visível antes do `hover`. Em nosso teste, definitivamente queremos garantir que ele esteja visível.

Para encontrar a `<div>` interna que possui a legenda criaremos outro identificador chamado `boxCaption` com `By.className`, e o valor deste atributo:

```
private By boxCaption = By.className("figcaption");
```

Este identificador será usado para encontrar a `<div>` interna, e agora o método `hoverOverFigure` não será mais `void` mas deverá retornar o `WebElement FigureCaption`:

```
public FigureCaption hoverOverFigure(int index) {
    WebElement figure = driver.findElements(figureBox).get(index - 1);
    Actions actions = new Actions(driver);
    actions.moveToElement(figure).perform();
    return new FigureCaption(figure.findElement(boxCaption));
}
```

Agora, dentro dessa classe interna, sabemos que existem dois elementos com os quais nos preocupamos — o header `<h5>` e o link `<a>`. Para localizar eles, vamos criar alguns objetos `By` dentro desta classe `FigureCaption`. Eles não têm um identificador exclusivo, mas podemos usar a `tagName`.

Vamos criar agora alguns métodos. O que queremos saber?

Bem, em nosso teste vamos querer saber se a legenda é exibida, é claro. Isso terá um retorno do tipo boolean, então dirá: “sim, é exibido” ou “não, não é”. Vamos chamar esse método de `isCaptionDisplayed`.

```
public boolean isCaptionDisplayed() {
    return caption.isDisplayed();
}
```

`caption` é um `WebElement` e `isDisplayed()` é outro método de interação disponível no `WebElement` que estamos exercitando agora. Isso será usado para uma de nossas `assertions`.

Também queremos saber qual é o texto do header `<h5>` e verificar se está correto. Para isso criaremos outro método chamado `getTitle`.

```
public String getTitle() {
    return caption.findElement(header).getText();
}
```

Da mesma forma, vamos querer obter o texto do link `<a>`. Neste caso podemos obter o próprio link e também o texto do link. Criaremos outro método chamado `getLink`. A tag `<a>` possui o atributo `href` que tem o link, então podemos usar o método `getAttribute` para obter seu valor:

```
public String getLink() {
    return caption.findElement(link).getAttribute("href");
}
```

Na verdade, não será apenas o “[/users/1](#)” que você vê quando eu passar o mouse sobre isso, que é o link completo aqui, então é isso que será retornado para nós. Vamos apenas fazer essa observação por enquanto.

Criaremos outro método para obter o texto do link.

```
public String getLinkText() {
    return caption.findElement(link).getText();
}
```

Por fim, a classe “**HoverPage.java**” ficará assim:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.interactions.Actions;

public class HoversPage {

    private WebDriver driver;
    private By figureBox = By.className("figure");
    private By boxCaption = By.className("figcaption");

    public HoversPage(WebDriver driver) {
        this.driver = driver;
    }

    public FigureCaption hoverOverFigure(int index) {
        WebElement figure = driver.findElements(figureBox).get(index - 1);
        Actions actions = new Actions(driver);
        actions.moveToElement(figure).perform();
        return new FigureCaption(figure.findElement(boxCaption));
    }
}
```

Código Final
HoverPage.java

```

public class FigureCaption{

    private WebElement caption;
    private By header = By.tagName("h5");
    private By link = By.tagName("a");

    public FigureCaption(WebElement caption) {
        this.caption = caption;
    }

    public boolean isCaptionDisplayed() {
        return caption.isDisplayed();
    }

    public String getTitle() {
        return caption.findElement(header).getText();
    }

    public String getLink() {
        return caption.findElement(link).getAttribute("href");
    }

    public String getLinkText() {
        return caption.findElement(link).getText();
    }
}
}

```

Agora vamos criar nossos testes. No diretório “**test/java**” vamos criar um novo pacote que chamaremos de “**hover**” e nele vamos criar uma nova classe chamada “**HoverTests.java**”.

Da mesma forma como nas demais classes de teste, esta estende a classe “**BaseTests**” e terá um novo teste chamado de **testHoverUser1**.

Em nossa classe “**HomePage**” faremos **clickHoversLink()** que vai retornar um objeto **HoversPage**. Em seguida faremos **hoversPage.hoverOverFigure(1)** para fazer hover sobre a imagem do usuário 1, e isso nos retornará uma instância de **FigureCaption**, aquela classe interna que criamos. Vamos chamar esse objeto de **caption**.

Agora podemos fazer nossas *assertions*. Primeiro vamos confirmar que a legenda é exibida. Podemos usar **assertTrue** e chamar o método **caption.isCaptionDisplayed()** que deverá retornar um boolean **True** e, caso retorne **False**, apresentar uma mensagem no **Console**:

```
assertTrue(caption.isCaptionDisplayed(), "Caption not displayed");
```

Ao adicionarmos a *assertion* anterior tivemos que importar **import static org.testng.Assert.assertTrue;**. Como usaremos diferentes tipos de assertion, vamos alterar o import para importar todos os métodos de assertion usando **import static org.testng.Assert.*;**.

Depois de garantir que a legenda foi realmente exibida, podemos continuar com algumas *assertions* adicionais. Agora queremos ter certeza de que o título da legenda está correto, então vamos fazer isso com *assertEquals* comparando o valor da legenda obtido com *caption.getTitle()* e o valor esperado.

```
assertEquals(caption.getTitle(), "name: user1", "Caption title incorrect");
```

Além de nos certificarmos de que o título da legenda está correto, faremos o mesmo para *caption.getLinkText()* e verificar se o conteúdo recebido é igual à “Visualizar perfil”.

```
assertEquals(caption.getLinkText(), "View profile", "Caption link text incorrect");
```

Por fim, vamos adicionar uma *assertion* final para garantir que o link esteja correto. Lembre-se de que o link obtido será o link absoluto. Para evitar usar a URL completa que poderá ter mudança de protocolo entre *http* e *https*, ou de subdomínio que pode mudar com base no fato de o teste ser executado em ambientes prod, test ou dev, tudo que eu realmente queremos verificar é que o link termina com “[/users/1](#)”.

```
assertTrue(caption.getLink().endsWith("/users/1"), "Link incorrect");
```

O resultado final de nossa classe de teste ficará assim:

Código Final
HoverTests.java

```
package hover;

import base.BaseTests;
import org.testng.annotations.Test;
import static org.testng.Assert.*;

public class HoverTests extends BaseTests {

    @Test
    public void testHoverUser1() {

        var hoversPage = homePage.clickHoversLink();
        var caption = hoversPage.hoverOverFigure(1);
        assertTrue(caption.isCaptionDisplayed(), "Caption not displayed");
        assertEquals(caption.getTitle(), "name: user1", "Caption title incorrect");
        assertEquals(caption.getLinkText(), "View profile", "Caption link text incorrect");
        assertTrue(caption.getLink().endsWith("/users/1"), "Link incorrect");
    }
}
```

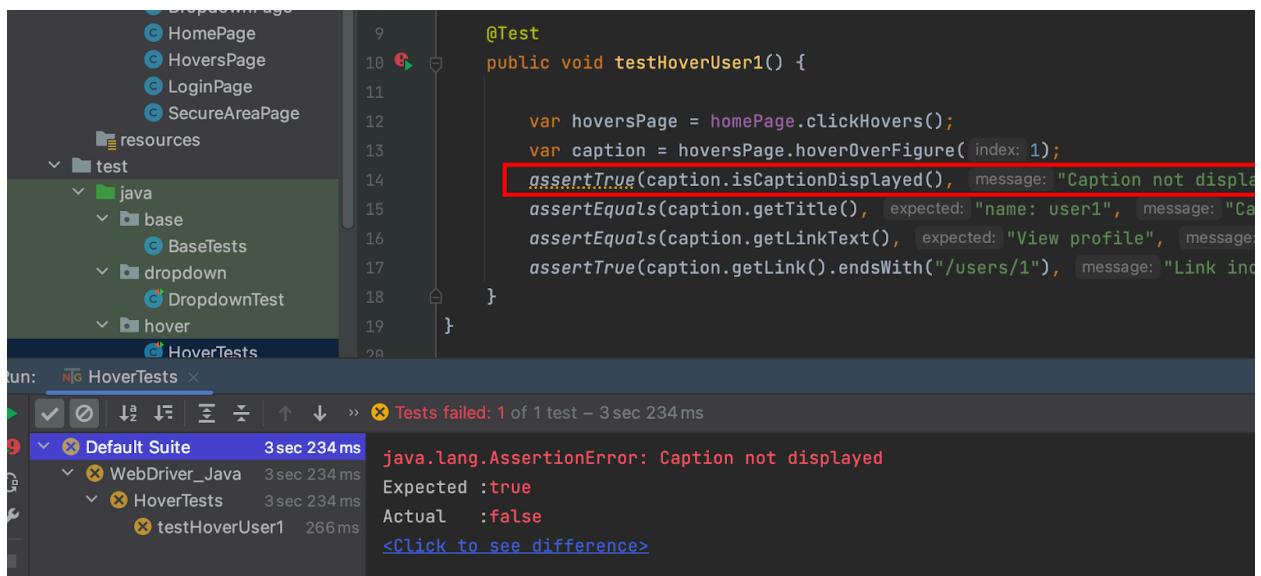
Ao executá-lo, vemos que o teste passa. É uma boa prática também testar seu teste e garantir que ele falhe quando for necessário.

Podemos simular uma falha para testar uma assertion, já que a legenda já existe no DOM mas não temos certeza se `isCaptionDisplayed` sempre retornará true. Para verificarmos isso, podemos apenas “comentar” a linha de código que faria *hover* sobre a imagem.

```
public FigureCaption hoverOverFigure(int index){
    WebElement figure = driver.findElements(textBox).get(index - 1);
    Actions actions = new Actions(driver);
    //actions.moveToElement(figure).perform();
    return new FigureCaption(figure.findElement(boxCaption));
}
```

Imagen 60: Comentando a ação de hover

Após fazermos esta alteração executamos novamente o teste e obtemos o seguinte resultado:



The screenshot shows the IntelliJ IDEA interface. On the left, the project structure displays several pages (HomePage, HoversPage, LoginPage, SecureAreaPage) and a test package containing Java files (BaseTests, DropdownTest, HoverTest). The HoverTests.java file is open in the editor, showing the code for the test method `testHoverUser1`. A red box highlights the line `assertTrue(caption.isCaptionDisplayed(), message: "Caption not displayed")`. In the bottom right, the Test Results window shows the execution of the Default Suite. It lists the WebDriver_Java test and the HoverTests test, both of which failed. The HoverTests test failed with the error message: `java.lang.AssertionError: Caption not displayed`, with the expected value being `:true` and the actual value being `:false`.

Imagen 61: Falha na execução do teste

Além de falhar na execução do teste, a mensagem que definimos para o caso de falha é apresentada dizendo “**Caption not displayed**” e informa o valor que era esperado e o valor que foi obtido.

Capítulo 10 - Digitando teclas adicionais

O método `sendKeys` da classe `WebElement` permite passar uma sequência de caracteres. Até agora, passamos em uma única `String`. No entanto, você também pode passar outros caracteres e pressionar outras teclas, como `backspace`, `tab`, as **teclas de função**, `shift`, `alt` e quaisquer outras teclas do seu teclado.

Na página inicial do nosso site de testes vamos clicar no link “**Key Presses**”. Para isso, vamos criar um novo método em nossa classe “**HomePage**” que clicará nesse link e retornará uma instância da classe “**KeyPressesPage**” que criaremos em seguida:

```
public KeyPressesPage clickKeyPressesLink() {
    clickLink("Key Presses");
    return new KeyPressesPage(driver);
}
```

Em seguida vamos criar a classe “**KeyPressesPage**” dentro do pacote “**pages**” com um método construtor de mesmo nome que receberá o `WebDriver`.

```
package pages;

import org.openqa.selenium.WebDriver;

public class KeyPressesPage {

    private WebDriver driver;

    public KeyPressesPage(WebDriver driver) {
        this.driver = driver;
    }
}
```

Para podermos enviar alguns caracteres para esse campo `<input>`, vamos inspecionar este campo para obter seu localizador e criar um objeto `By`.

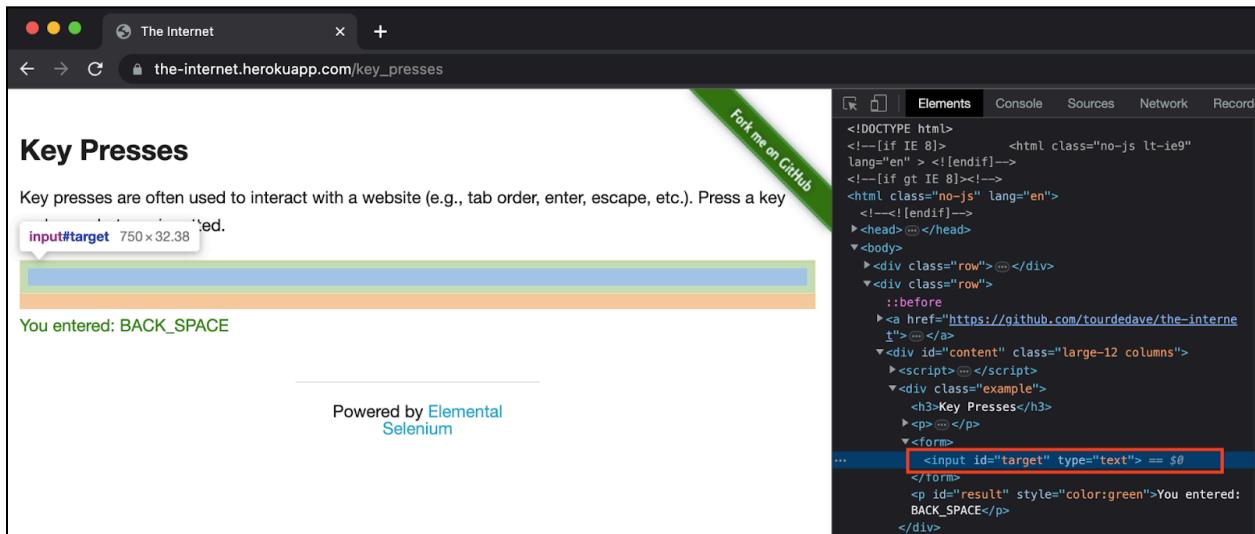


Imagen 62: Inspecionando o campo Input

Vemos que ele tem um *id* "**target**". Criaremos um identificador *By* chamado "**inputField**", e vamos criar um método para digitar caracteres dentro deste campo **<input>**.

```
private By inputField = By.id("target");

public void enterText(String text){
    driver.findElement(inputField).sendKeys(text);
}
```

A outra coisa que precisamos fazer para nosso teste é obter o texto que aparece depois que digitamos algo no campo **<input>**. Inspecionando este elemento vemos que ele tem um id "**result**". Vamos criar um objeto *By* para isso.

```
private By resultText = By.id("result");
```

E também um método para retornar essa String:

```
public String getResult(){
    return driver.findElement(resultText).getText();
}
```

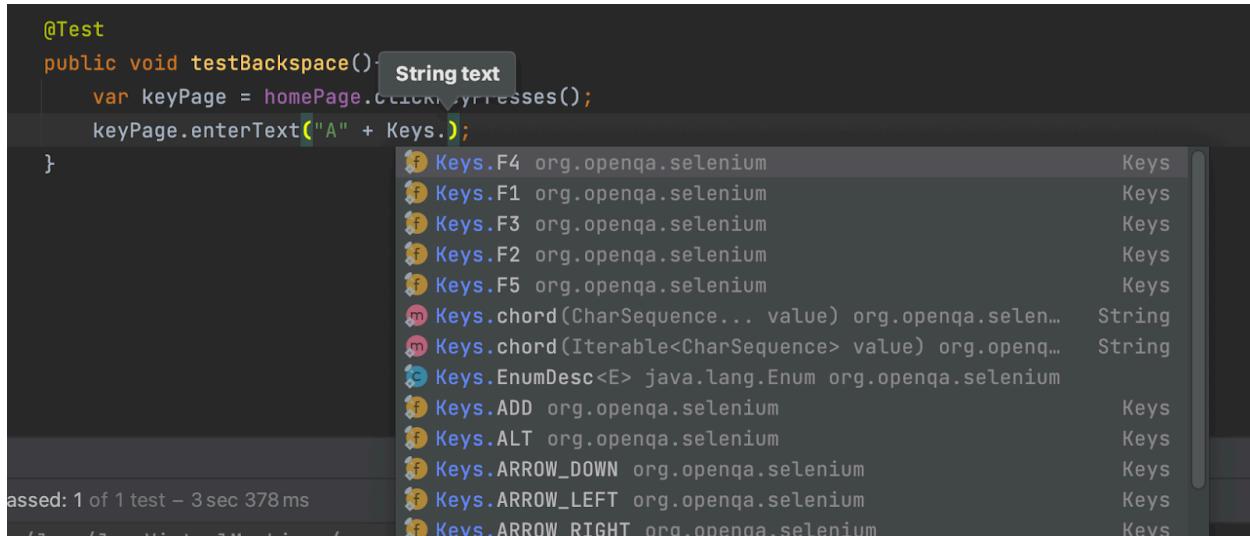
Agora vamos escrever nosso teste. No diretório "**test/java**" vamos criar um novo pacote que chamaremos de "**keys**" e nele vamos criar uma nova classe chamada "**KeysTests.java**".

Da mesma forma como nas demais classes de teste, esta estende a classe "**BaseTests**" e terá um novo teste chamado de **testBackspace**.

Em nossa classe “**HomePage**” faremos `clickKeyPressesLink()` que vai retornar um objeto `KeyPressesPage` recebido por uma variável `keyPage`. Em seguida faremos `keyPage.enterText()`.

Digamos que queremos digitar "A" e então queremos digitar “**backspace**”.

Bem, como você digita “**backspace**”? Não podemos simplesmente enviar a String “**backspace**”, pois isto digitaria esta palavra inteira, então o que podemos usar é essa classe do Selenium chamada “**Keys**”.



```

@Test
public void testBackspace() {
    String text
    var keyPage = homepage.clickKeyPresses();
    keyPage.enterText("A" + Keys.);
}
  Keys.F4 org.openqa.selenium.Keys
  Keys.F1 org.openqa.selenium.Keys
  Keys.F3 org.openqa.selenium.Keys
  Keys.F2 org.openqa.selenium.Keys
  Keys.F5 org.openqa.selenium.Keys
  Keys.chord(CharSequence... value) org.openqa.selenium.String
  Keys.chord(Iterable<CharSequence> value) org.openqa.selenium.String
  Keys.EnumDesc<E> java.lang.Enum org.openqa.selenium.Keys
  Keys.ADD org.openqa.selenium.Keys
  Keys.ALT org.openqa.selenium.Keys
  Keys.ARROW_DOWN org.openqa.selenium.Keys
  Keys.ARROW_LEFT org.openqa.selenium.Keys
  Keys.ARROW_RIGHT org.openqa.selenium.Keys

```

Imagen 63: Classe Keys para digitar outras teclas

Vemos que há várias teclas para selecionar, então vamos escolher o “**BACK_SPACE**”:

```
keyPage.enterText("A" + Keys.BACK_SPACE);
```

Agora queremos verificar se o elemento “**result**” apresenta o valor “**BACK_SPACE**”.

Vamos fazer uma assertion `assertEquals` e usar `keyPage.getResult()` para comparar seu resultado que deverá retornar a string “You entered: BACK_SPACE”.

```

@Test
public void testBackspace() {
    var keyPage = homepage.clickKeyPresses();
    keyPage.enterText("A" + Keys.BACK_SPACE);
    assertEquals(keyPage.getResult(), "You entered: BACK_SPACE");
}

```

Digamos que queiramos digitar um caractere especial, como o caractere “**pi**” (π).

Poderíamos fazer isso no teclado do MacBook pressionando “**Option**” e “**p**” ao mesmo tempo. No Windows, isso é “**Alt**” e “**p**” simultaneamente. Vamos ver como pressionar várias teclas ao mesmo tempo.

Voltando à classe “**KeyPressesPage**” de nosso *framework*, vamos criar um novo método para inserir o caractere “**pi**” chamado “**enterPi**” que vai chamar o método “**enterText**” que temos ali, e vai passar as teclas a serem pressionadas.

A classe “**Keys**” possui um método “**chord**” que nos permite informar uma lista de teclas a serem pressionadas ao mesmo tempo, como “**Keys.OPTION**” e “**p**”.

```
public void enterPi() {
    enterText(Keys.chord(Keys.ALT, "P"));
}
```

Digamos que queremos fazer mais — queremos dizer que “**pi**” é igual a **3.14**. Este método “**chord**” fará pressionar as teclas ao mesmo tempo, mas depois queremos enviar outra *String*.

```
public void enterPi() {
    enterText(Keys.chord(Keys.ALT, "P") + "=3.14");
}
```

Em nosso teste, vamos fazer uma chamada para este método:

```
@Test
public void testPi() {
    var keyPage = homePage.clickKeyPressesLink();
    keyPage.enterPi();
}
```

Para acompanhamos a execução de nosso teste e observarmos o que acontece em uma determinada linha de código, vamos adicionar um *Breakpoint* clicando ao lado do número da linha 14:



Imagen 64: Adicionando um Breakpoint

Para executar o teste e parar no breakpoint, vamos executar em modo de *Debug*:

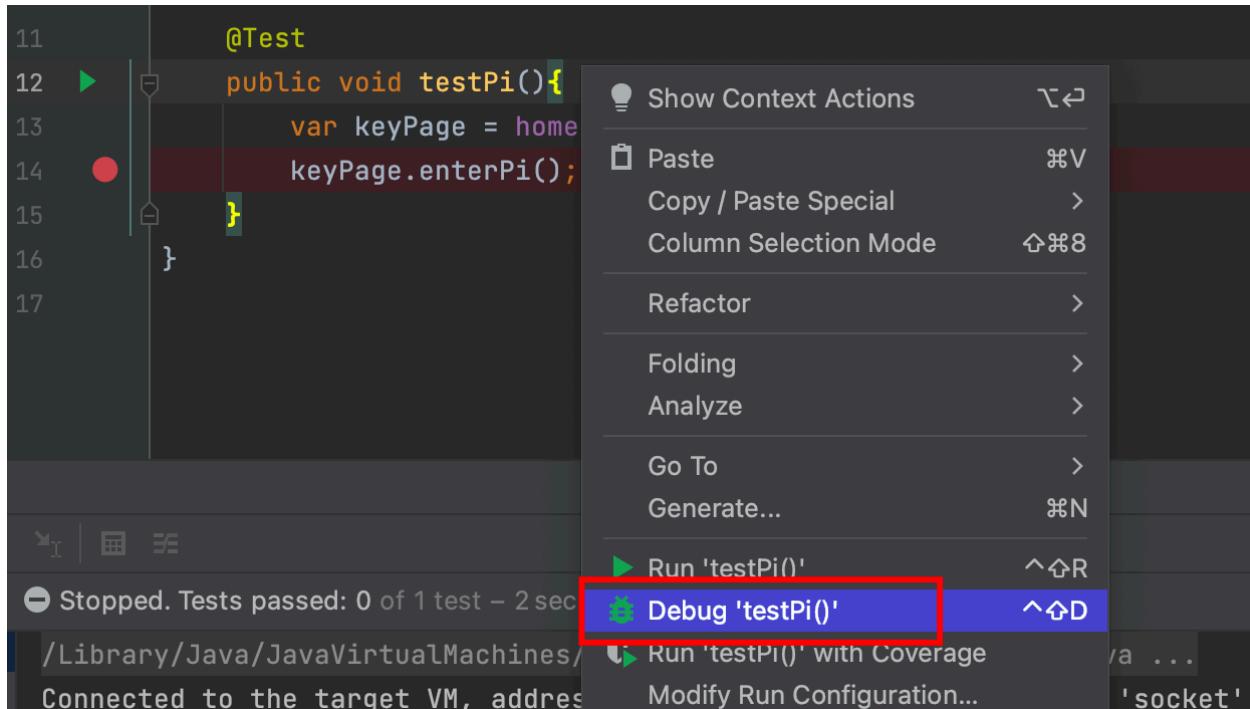


Imagen 65: Executando o teste em modo de Debug

Ao interromper a execução do teste em modo de *Debug* ele será interrompido antes de executar as instruções da linha de código onde inserimos o Breakpoint, e veremos que não há nada digitado no campo <input> do site.

Vamos entrar neste método clicando em “**Step Into**” e vamos permitir que ele insira este texto:

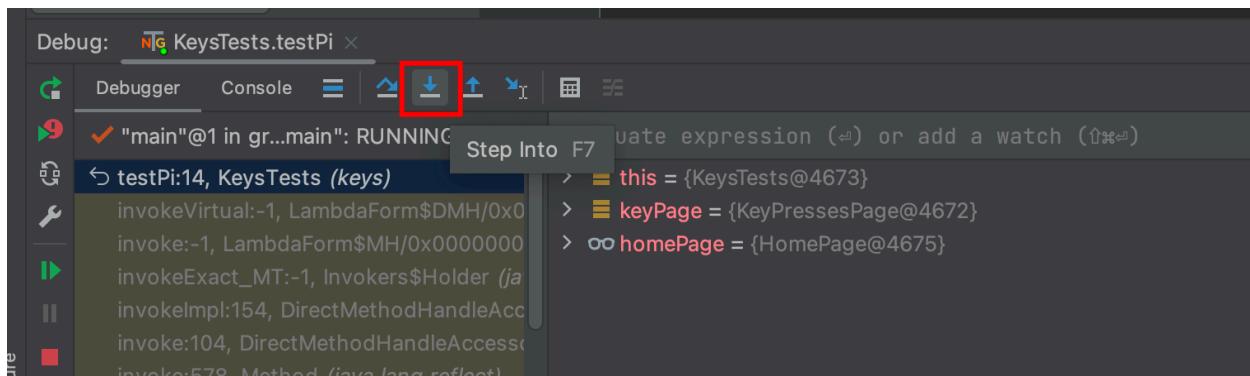


Imagen 66: Clicando em “Step Into” para executar o método da linha que contém o Breakpoint

Ao clicar em “**Step Into**” a execução vai avançar passo a passo, acessando cada método da classe *Selenium* utilizados. Temos que repetir o “**Step Into**” até que ele execute a digitação e vejamos o seguinte resultado:

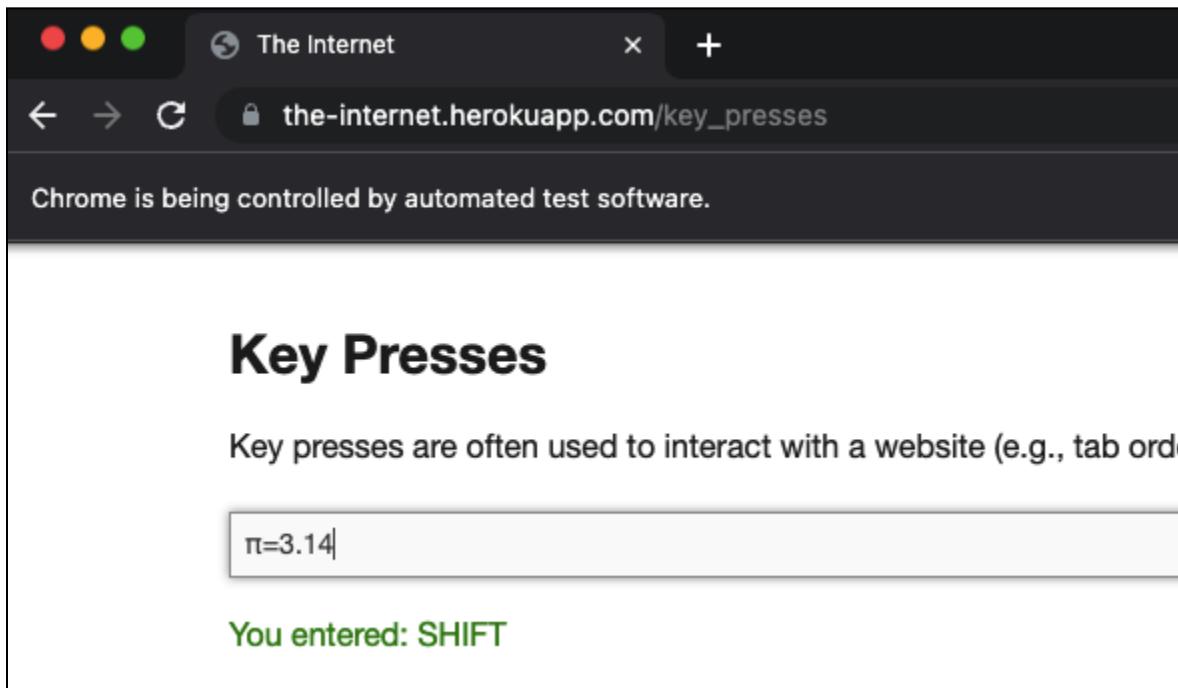


Imagen 67: Texto preenchido no campo <input>

Para finalizar este teste, poderíamos fazer o seguinte:

- Criar um método para obter o último caractere do conteúdo preenchido no campo <input>;
- Criar uma *assertion* para verificar que este texto obtido pelo método `getResults()` termina com este caractere.

Capítulo 11 - Alerts, Uploads e Modals

Neste capítulo, veremos os pop-ups e como interagir com eles. Veremos 3 tipos diferentes de pop-ups: **alerts**, **uploads** de arquivos e também **modals**:

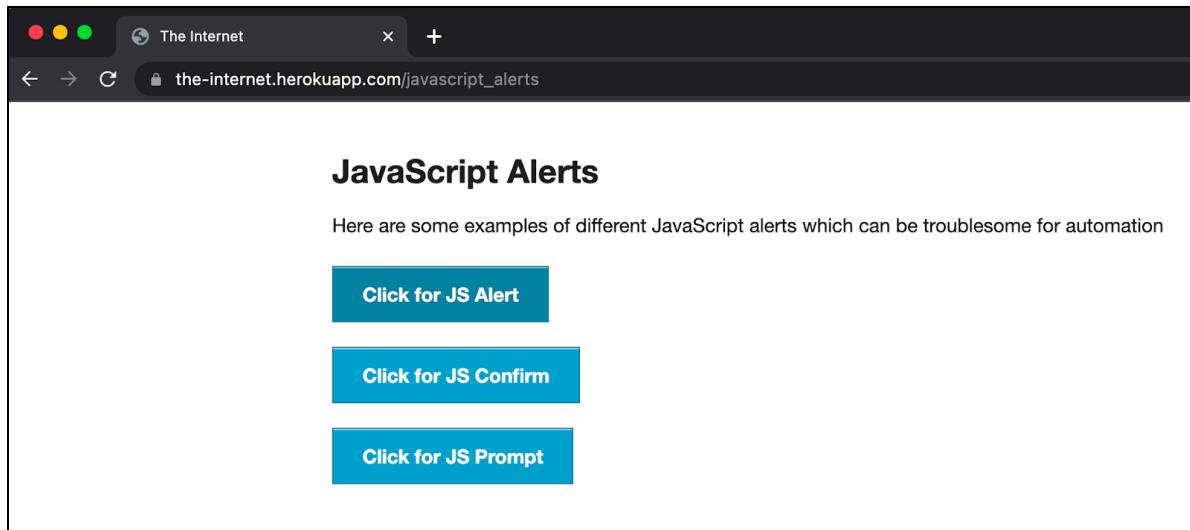


Imagen 68: Página de JavaScript Alerts

11.1 Alerts

Em nosso website de teste temos um link chamado **JavaScript Alerts**. O primeiro passo é clicar neste link, e para isso vamos adicionar um novo método à classe “**HomePage.java**”.

Vamos criar uma cópia do método **clickDropdownLink()** e alterar o seguinte:

- O nome para **clickJavaScriptAlerts()**
- O tipo do objeto que vai ser retornado que será uma instância da classe “**AlertsPage.java**” que também iremos criar
- O texto do link à ser clicado para “**JavaScript Alerts**”

```
public AlertsPage clickJavaScriptAlerts() {  
    clickLink("JavaScript Alerts");  
    return new AlertsPage(driver);  
}
```

Também já criei a classe "**AlertsPage.java**" com seu método construtor:

```
package pages;

import org.openqa.selenium.WebDriver;

public class AlertsPage {

    private WebDriver driver;

    public AlertsPage(WebDriver driver) {
        this.driver = driver;
    }
}
```

Vamos criar agora o elemento **By** para o primeiro botão. Ao inspecionarmos este botão veremos que não existe um ***id*** ou qualquer outro identificador. O único atributo aqui é o ***onclick="jsAlert()"***. Mas isso realmente não parece seguro porque o nome dessa função pode mudar.

Então, vamos utilizar o texto do botão e podemos escrever um XPath para isso: ***.//button[text()="Click for JS Alert"]***. Vamos testar este seletor no inspetor de elementos:

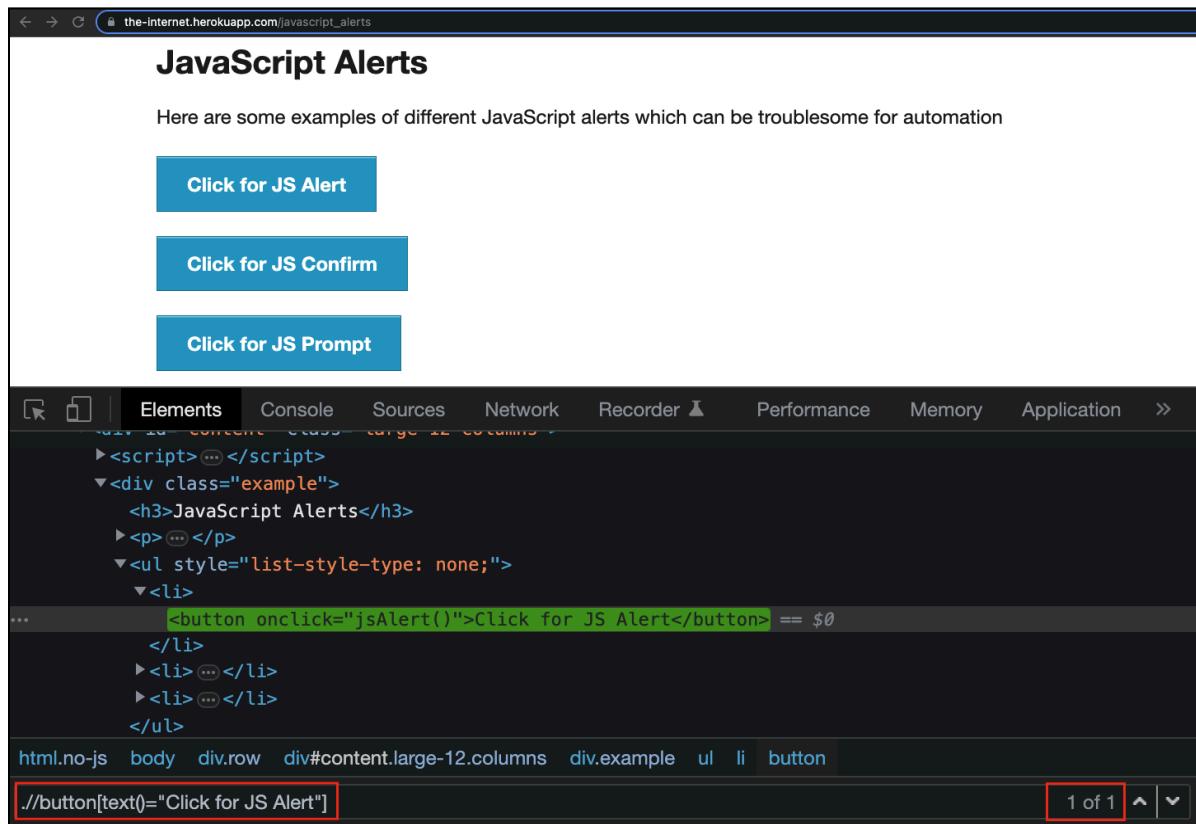


Imagen 69: Testando o seletor XPath

Em seguida criaremos o novo elemento **By** que busca pelo XPath:

```
private By triggerAlertButton = By.xpath("//button[text()='Click for JS Alert']);
```

Eu ia chamar esse objeto de **alertView**, mas isso parece um pouco confuso, porque mais tarde quando olharmos para isso podemos pensar que é um botão que está dentro do alerta. Portanto, sejamos muito claros em nossa nomenclatura e chamemos isso de **triggerAlertButton**.

Agora criaremos o método que clica neste botão:

```
public void triggerAlert(){
    driver.findElement(triggerAlertButton).click();
}
```

Depois de clicarmos nisso, sabemos que receberemos um *alert*. Vamos criar um método para clicar no “**OK**” desse *alert*.

Novamente, não podemos usar **driver.findElement** para interagir com o *alert* porque não é um *WebElement*. O que vamos precisar fazer é usar o método “**switchTo**” do WebDriver que mudará do contexto atual do DOM para o *alert*.

Então, dizemos **switchTo** e, em seguida, vemos mais algumas opções para as quais podemos alternar, sendo uma delas um alerta. Isto será feito dentro de outro método chamado **acceptAlert**:

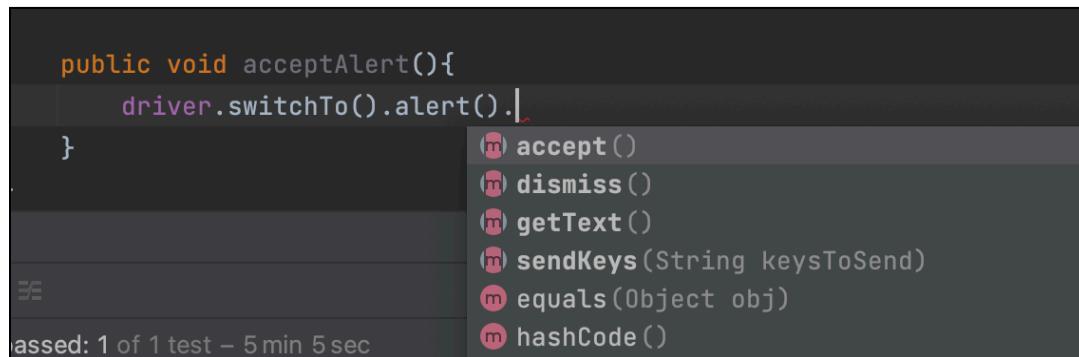


Imagen 70: Opção `accept()` para o objeto `alert()` do método `switchTo()`

```
public void acceptAlert(){
    driver.switchTo().alert().accept();
}
```

Agora vamos escrever nosso teste. No diretório “**test/java**” vamos criar um novo pacote que chamaremos de “**alerts**” e nele vamos criar uma nova classe chamada “**AlertTests.java**”.

Da mesma forma como nas demais classes de teste, esta estende a classe **“BaseTests.java”** e terá um novo teste chamado de **testAcceptAlert**.

Fazendo **homePage.clickJavaScriptAlerts()** teremos como retorno um objeto **AlertsPage** recebido por uma variável **alertsPage**. Em seguida, em **alertsPage** podemos dizer: **“Acionar o alert”** e, em seguida, podemos dizer: **“Aceitar o alert”**.

```
@Test
public void testAcceptAlert() {
    var alertsPage = homePage.clickJavaScriptAlerts();
    alertsPage.triggerAlert();
    alertsPage.acceptAlert();
}
```

Agora, para testar isso, lembre-se de que temos este texto resultante **“You successfully clicked an alert”**. Então, vamos adicionar este objeto **By** à nossa classe **“AlertsPage.java”**:

```
private By results = By.id("result");
```

E um método para obter seu texto:

```
public String getResult() {
    return driver.findElement(results).getText();
}
```

Retornando ao nosso teste, precisamos de uma **assertion** para comparar o resultado obtido com a String esperada:

```
assertEquals(alertsPage.getResult(), "You successfully clicked an alert",
            "Results text incorrect");
```

Vamos criar mais alguns testes para os outros tipos de alerts. Para o nosso próximo teste, precisamos clicar no botão **“JS Confirm”**, então vamos adicionar outro objeto **By** à nossa classe **“AlertsPage.java”**:

```
private By triggerConfirmButton = By.xpath("//button[text()='Click for JS
Confirm']");
```

Para clicar neste botão criaremos outro método para **triggerConfirm**:

```
public void triggerConfirm() {
    driver.findElement(triggerConfirmButton).click();
}
```

Desta vez vamos validar o texto do *alert* e depois clicar em ***Cancel***:

```
public void dismissAlert() {
    driver.switchTo().alert().dismiss();
}
```

Neste teste faremos o seguinte:

- Executando `homePage.clickJavaScriptAlerts()` teremos como retorno um objeto `AlertsPage` recebido por uma variável `alertsPage`;
- Em seguida, faremos `alertsPage.triggerConfirm()` para abrir o *alert*;
- Com o *alert* aberto, faremos `alertsPage.getAlertText()` para obter seu texto e armazenar esta String em uma variável chamada de `text`;
- Clicarmos no botão “***Cancel***” para fechar o *alert*;
- Uma `assertion` vai comparar esta `String` com o texto esperado.

```
@Test
public void testGetTextFromAlert() {
    var alertsPage = homePage.clickJavaScriptAlerts();
    alertsPage.triggerConfirm();

    String text = alertsPage.getAlertText();
    alertsPage.dismissAlert();

    assertEquals(text, "I am a JS Confirm", "Alert text incorrect");
}
```

Por fim, para o nosso último teste de *alert*, precisamos clicar no botão “***JS Prompt***”, então vamos adicionar outro objeto `By` à nossa classe “`AlertsPage.java`”:

```
private By triggerPromptButton = By.xpath("//button[text()='Click for JS
Prompt']");
```

Para clicar neste botão criaremos outro método para `triggerPrompt`:

```
public void triggerPrompt() {
    driver.findElement(triggerPromptButton).click();
}
```

Neste teste faremos o seguinte:

- Executando `homePage.clickJavaScriptAlerts()` teremos como retorno um objeto `AlertsPage` recebido por uma variável `alertsPage`;
- Em seguida, faremos `alertsPage.triggerPrompt()` para abrir o *alert*;
- Com o *alert* aberto, digitaremos a `String` “**Q.Way!**” com um novo método que criaremos chamado `alertsPage.setAlertInputText()`;

```
public void setAlertInputText(String text){
    driver.switchTo().alert().sendKeys(text);
}
```

- Clicaremos em “OK” para fechar o *alert*;
- Uma *assertion* vai comparar a *String* obtida com o método *getResult()* e o texto digitado.

```
@Test
public void testSetInputInAlert() {
    var alertsPage = homePage.clickJavaScriptAlerts();
    alertsPage.triggerPrompt();

    String text = "Q.Way!";
    alertsPage.setAlertInputText(text);
    alertsPage.acceptAlert();

    assertEquals(alertsPage.getResult(), "You entered: " + text, "Results
text incorrect");
}
```

No final, a classe “*AlertsPage.java*” ficará assim:

Código Final
AlertsPage.java

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class AlertsPage {

    private WebDriver driver;
    private By triggerAlertButton = By.xpath("//button[text()='Click for JS
Alert']");
    private By triggerConfirmButton = By.xpath("//button[text()='Click for JS
Confirm']");
    private By triggerPromptButton = By.xpath("//button[text()='Click for JS
Prompt']");
    private By results = By.id("result");

    public AlertsPage(WebDriver driver) {
        this.driver = driver;
    }

    public void triggerAlert() {
        driver.findElement(triggerAlertButton).click();
    }

    public void triggerConfirm() {
        driver.findElement(triggerConfirmButton).click();
    }

    public void triggerPrompt() {
```

```

        driver.findElement(triggerPromptButton).click();
    }

    public void acceptAlert() {
        driver.switchTo().alert().accept();
    }

    public void dismissAlert() {
        driver.switchTo().alert().dismiss();
    }

    public String getAlertText() {
        return driver.switchTo().alert().getText();
    }

    public void setAlertInputText(String text) {
        driver.switchTo().alert().sendKeys(text);
    }

    public String getResult() {
        return driver.findElement(results).getText();
    }
}

```

E a classe de testes “**AlertTests.java**” ficará assim:

Código Final
AlertTests.java

```

package alerts;

import base.BaseTests;
import org.testng.annotations.Test;
import static org.testng.Assert.assertEquals;

public class AlertTests extends BaseTests {

    @Test
    public void testAcceptAlert() {
        var alertsPage = homePage.clickJavaScriptAlerts();
        alertsPage.triggerAlert();
        alertsPage.acceptAlert();
        assertEquals(alertsPage.getResult(), "You successfully clicked an alert", "Results text incorrect");
    }

    @Test
    public void testGetTextFromAlert() {
        var alertsPage = homePage.clickJavaScriptAlerts();
        alertsPage.triggerConfirm();

        String text = alertsPage.getAlertText();
        alertsPage.dismissAlert();

        assertEquals(text, "I am a JS Confirm", "Alert text incorrect");
    }

    @Test
    public void testSetInputInAlert() {

```

```

        var alertsPage = homePage.clickJavaScriptAlerts();
        alertsPage.triggerPrompt();

        String text = "Q.Way!";
        alertsPage.setAlertInputText(text);
        alertsPage.acceptAlert();

        assertEquals(alertsPage.getResult(), "You entered: " + text, "Results
text incorrect");
    }
}

```

11.1.1 Executando vários testes em uma classe

Esta foi a primeira vez que criamos vários testes em uma mesma classe. Se tentarmos executar toda essa classe, isso falharia porque cada um desses testes está tentando clicar no link “**JavaScript Alerts**”, assumindo que estamos na página inicial apresentada após executarmos o método `setUp()` marcado como `@BeforeClass`.

Porém, não estaremos na página inicial quando terminarmos esses testes, então eles estão deixando o site na página errada. O que podemos fazer é adicionar algo em nossa classe “**BaseTests**” para dizer, “antes de qualquer teste, certifique-se de que está realmente na página inicial”.

Adicionaremos um método com a annotation `@BeforeMethod` chamado `goHome()` que nos levará à página inicial.

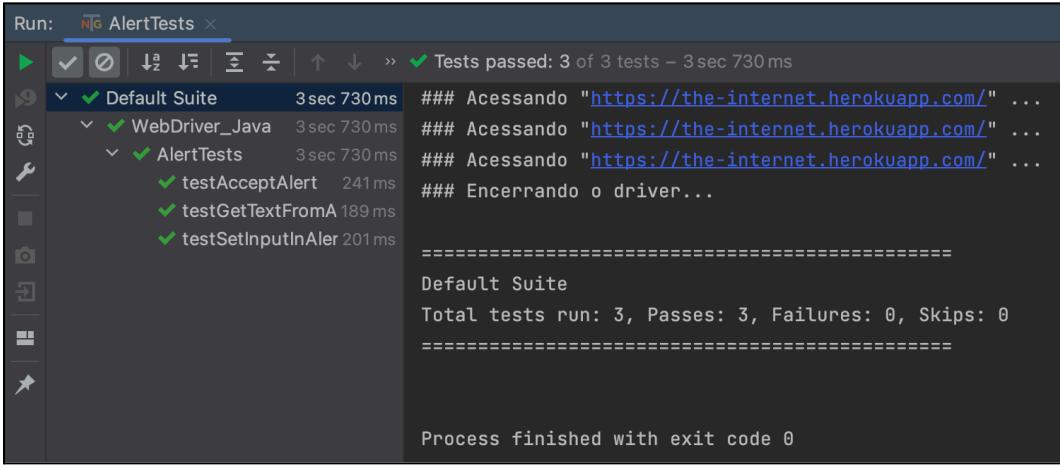
```

@BeforeClass
public void setUp() {
    System.setProperty("webdriver.chrome.driver", "resources/chromedriver");
    driver = new ChromeDriver();
}

@BeforeMethod
public void goHome() {
    String URL = "https://the-internet.herokuapp.com/";
    System.out.println("### Acessando \"" + URL + "\" ...");
    driver.get(URL);
    homePage = new HomePage(driver);
}

```

E agora vamos executar toda a classe e garantir que isso realmente funcione.



```

Run: NT AlertTests x
    Tests passed: 3 of 3 tests - 3 sec 730 ms
    Default Suite      3 sec 730 ms
        WebDriver_Java 3 sec 730 ms
            AlertTests   3 sec 730 ms
                testAcceptAlert 241 ms
                testGetTextFromA 189 ms
                testSetInputInAlert 201 ms
                ====
                Default Suite
                Total tests run: 3, Passes: 3, Failures: 0, Skips: 0
                ====
Process finished with exit code 0

```

Imagen 71: Resultado da execução de uma classe de testes com 3 métodos de teste

11.2 Upload

Vamos falar sobre outro tipo de pop-up. Vamos clicar no link **File Upload** na página inicial do aplicativo:

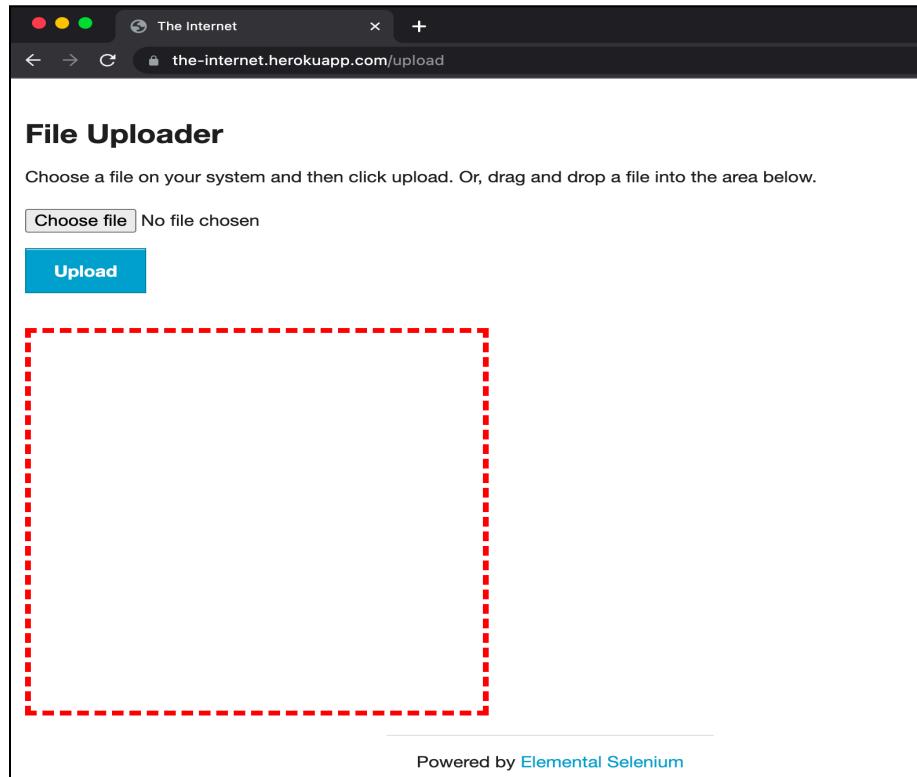


Imagen 72: Página de File Upload

Quando clicamos no botão "**Choose file**", obtemos um pop-up. Mas este é um pop-up do gerenciador de arquivos do sistema operacional, portanto, é diferente do *alert* de JavaScript:

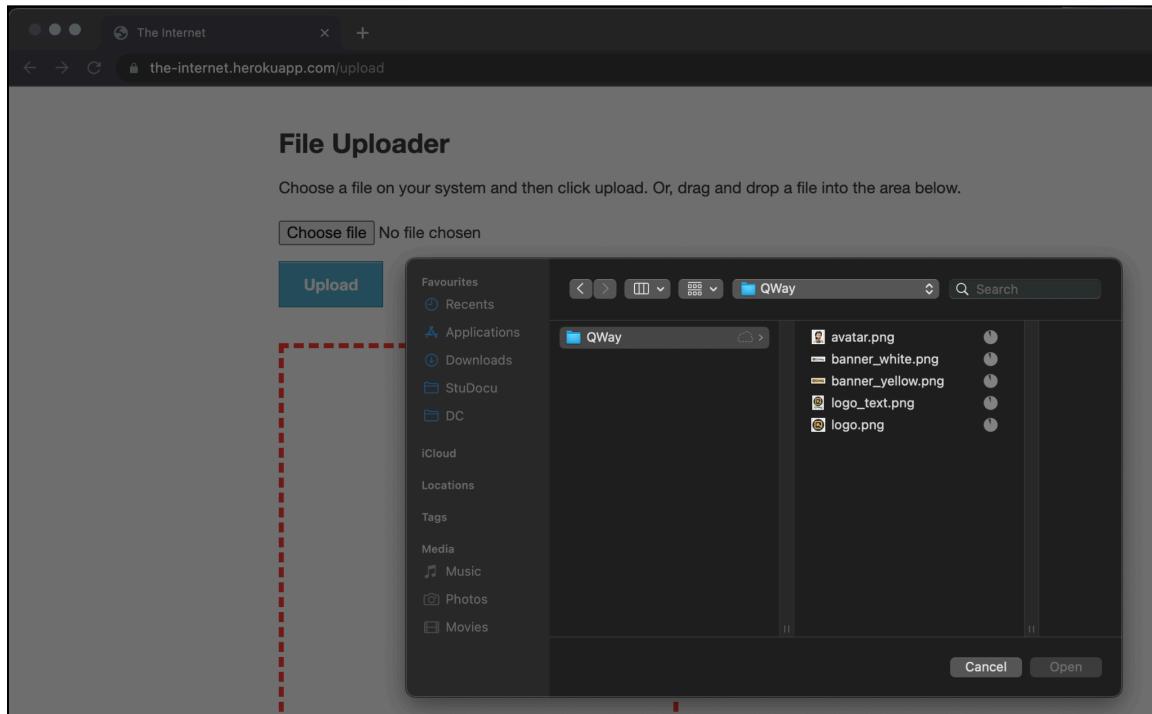


Imagen 73: Pop-up de Upload

Você não poderá usar a classe *Alert* dentro do *Selenium* para interagir com este pop-up. Para arquivos, fazemos algo diferente: ignoramos toda essa operação. Em vez disso, passamos o caminho absoluto do arquivo para este formulário.

Vamos começar adicionando um método à classe “**HomePage.java**” para clicar no link de acesso à esta página criando uma cópia do método **clickDropdownLink()** e alterando o seguinte:

- O nome para **clickFileUploadLink()**
- O tipo do objeto que vai ser retornado que será uma instância da classe que também iremos criar
- O texto do link à ser clicado para “**File Upload**”

```
public FileUploadPage clickFileUploadLink() {
    clickLink("File Upload");
    return new FileUploadPage(driver);
}
```

Para fazer um teste rápido e carregar nosso executável *chromedriver* aqui, vamos clicar com o botão direito para inspecionar este elemento:

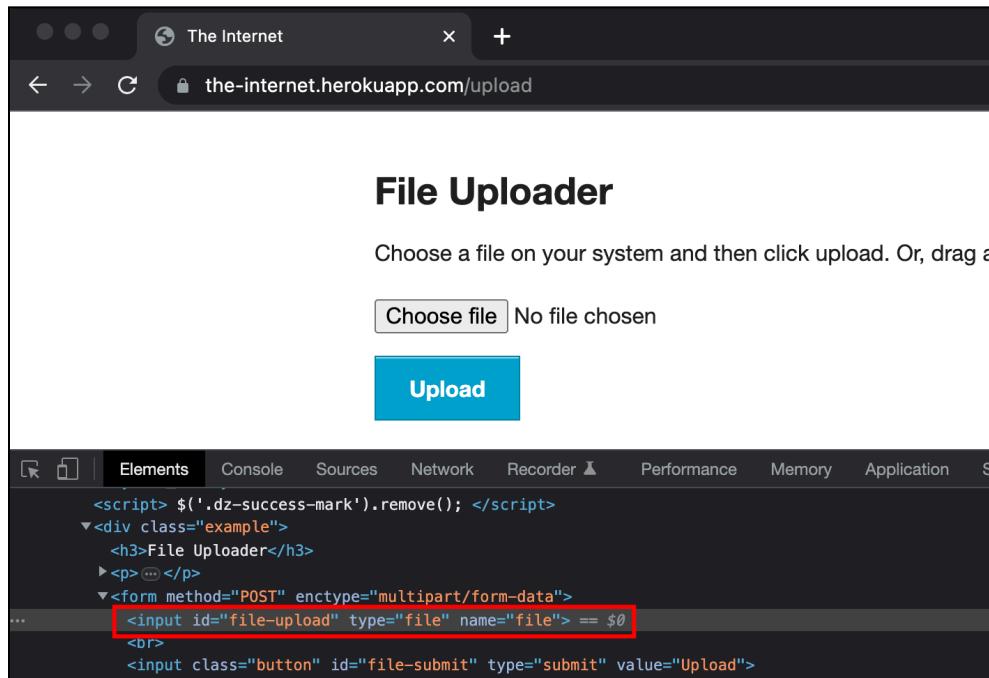


Imagen 74: Campo de Input do tipo “file”

Vemos aqui que temos um formulário e o formulário tem um “**<input>**” do tipo “**file**”. Vamos enviar o caminho absoluto do arquivo para este campo que tem um atributo ***id="file-upload"***.

Eu criei uma nova classe de *Page Objects* chamada “**FileUploadPage.java**” dentro do pacote “**pages**”:

Código Final
FileUploadPage.java

```

package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class FileUploadPage {

    private WebDriver driver;
    private By inputField = By.id("file-upload");
    private By uploadButton = By.id("file-submit");
    private By uploadedFile = By.id("uploaded-files");

    public FileUploadPage(WebDriver driver) {
        this.driver = driver;
    }

    public void clickUploadButton() {
        driver.findElement(uploadButton).click();
    }

    public void uploadFile(String absolutePathOfFile) {
        driver.findElement(inputField).sendKeys(absolutePathOfFile);
        clickUploadButton();
    }
}

```

```

    public String getUploadedFiles() {
        String uploadedFileText = driver.findElement(uploadedFile).getText();
        return uploadedFileText;
    }
}

```

Temos o WebDriver e o método construtor. Também temos dois elementos **By**: um para o **<input id=inputField>** e outro para o botão de upload **uploadButton**, além da **<div id="uploaded-files">** que contém o texto com o nome do arquivo carregado.

Para fazer o upload do arquivo, o teste precisa nos passar o caminho absoluto do arquivo, e então enviaremos esse caminho para o campo **<input>**, e então clicaremos no botão de upload.

Agora vamos escrever nosso teste. No pacote “**alerts**” que já criamos anteriormente vamos criar uma nova classe chamada “**FileUploadTests.java**”.

Da mesma forma como nas demais classes de teste, esta estende a classe “**BaseTests.java**” e terá um novo teste chamado de **testFileUpload**.

Neste teste faremos o seguinte:

- Executando **homePage.clickFileUploadLink()** teremos como retorno um objeto **FileUploadPage** recebido por uma variável **uploadPage**;
- Em seguida, faremos **uploadPage.uploadFile()** informando o path absoluto do arquivo e clicando no botão de *Upload*;
- Uma **assertion** vai comparar a **String** obtida com o método **getUploadedFiles()** e o texto digitado:

Código Final
FileUploadTests.java

```

package alerts;

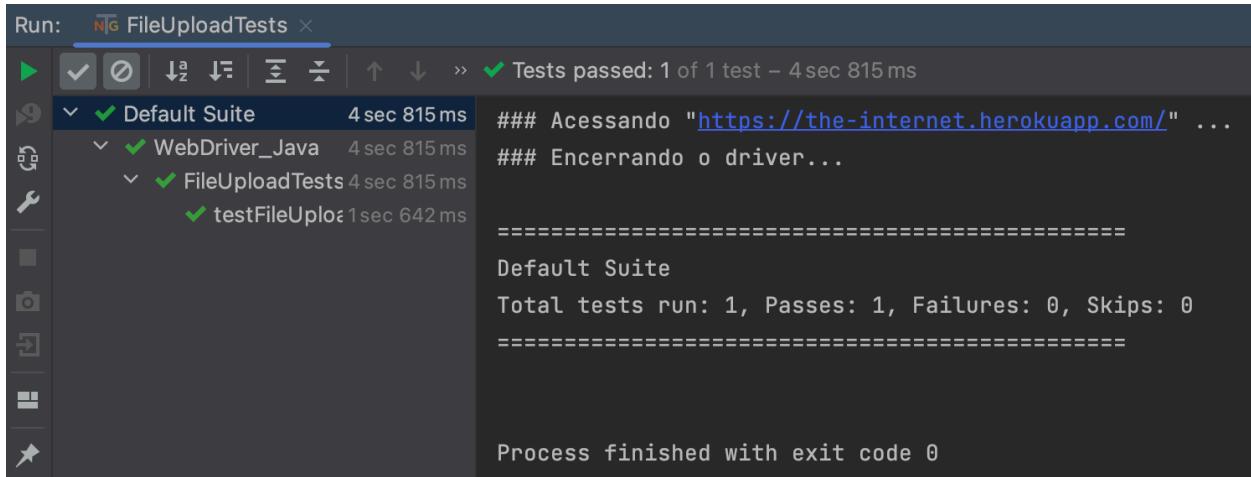
import base.BaseTests;
import org.testng.annotations.Test;
import static org.testng.Assert.assertEquals;

public class FileUploadTests extends BaseTests {

    @Test
    public void testFileUpload() {
        var uploadPage = homePage.clickFileUploadLink();

        uploadPage.uploadFile("/Users/daniel.castro/IdeaProjects/WebDriver_Java/WebDriver_Java/resources/chromedriver");
        assertEquals(uploadPage.getUploadedFiles(), "chromedriver", "Uploaded files incorrect");
    }
}

```



The screenshot shows the TestNG Results window with the following details:

- Run:** FileUploadTests
- Tests passed:** 1 of 1 test – 4 sec 815 ms
- Suite:** Default Suite (4 sec 815 ms)
 - Test:** WebDriver_Java (4 sec 815 ms)
 - Method:** FileUploadTests (4 sec 815 ms)
 - Test Case:** testFileUpload (1 sec 642 ms)
- Logs:**
 - ### Acessando "<https://the-internet.herokuapp.com/>" ...
 - ### Encerrando o driver...
- Total:** Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
- Process finished:** Process finished with exit code 0

Imagen 75: Resultado da execução do teste de Upload

11.3 Modals

Outro tipo de pop-up é um **Modal**. Vamos clicar no link **Entry Ad** na página inicial do aplicativo:

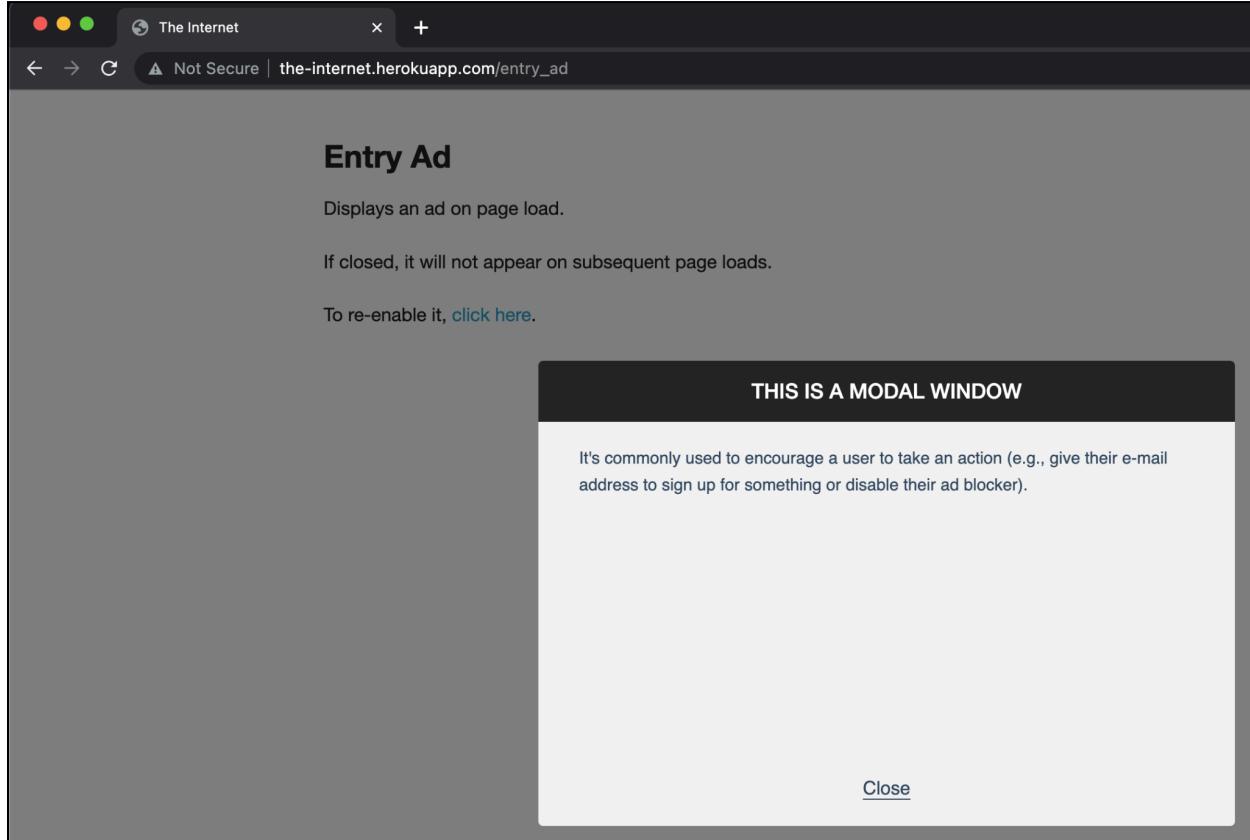


Imagen 76: Página de Modal

Observe que este *Modal* aparece e, no fundo, a página principal foi esmaecida. Uma das grandes vantagens dos *Modals* é que eles realmente aparecem no DOM, ao contrário dos alertas. Assim podemos interagir com os modais da mesma forma que fazemos com a própria página.

Quando estiver nesta página, não há nada de especial que você precise fazer. Você apenas cria objetos para os WebElements do que quiser no *Modal*.

Por exemplo, se você quiser clicar neste botão “***Close***”, você simplesmente obterá um identificador para esta div class=“modal-footer” com um seletor CSS utilizando a expressão “***.modal-footer***” e depois para a tag ***<p>*** contendo o texto “***Close***”.

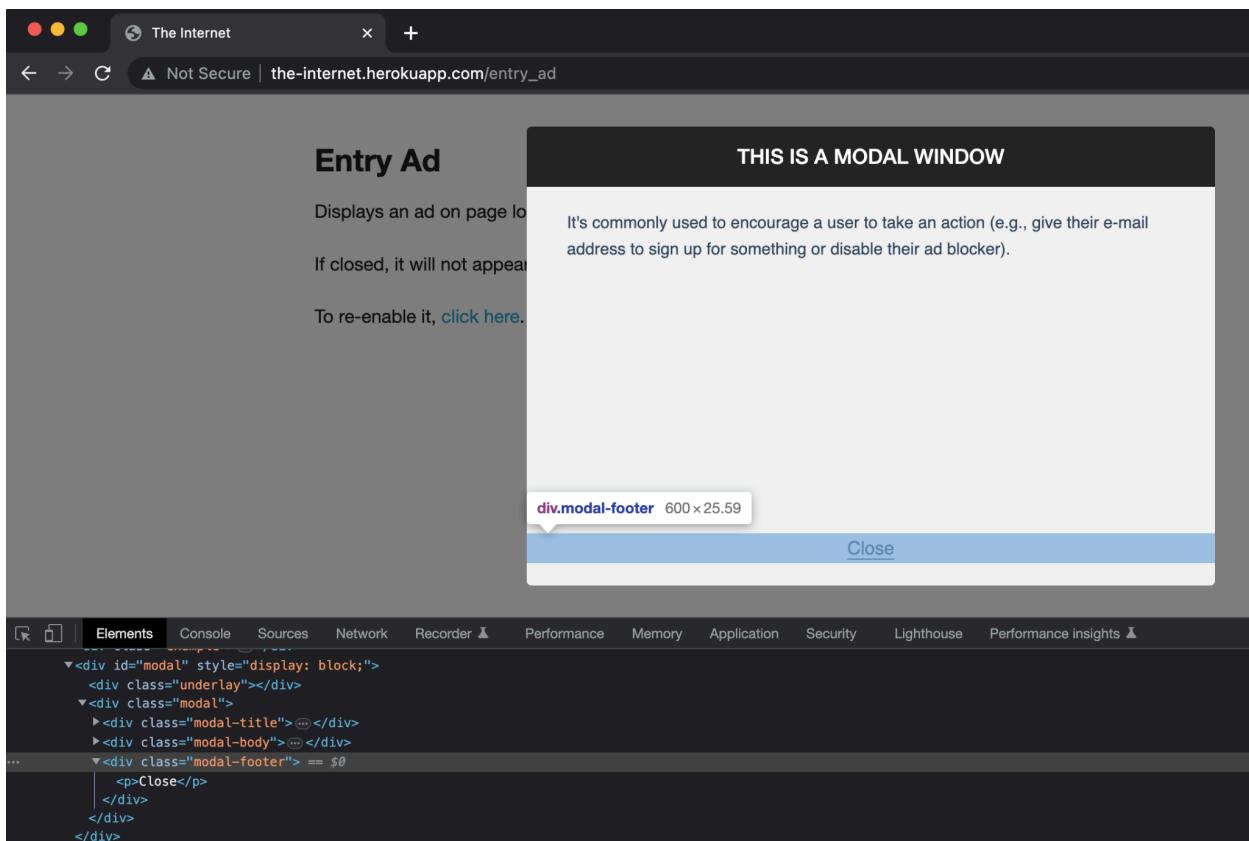


Imagen 77: Elementos do Modal no DOM

Capítulo 12 - Frames

Neste capítulo, veremos os *Frames* que são documentos HTML incorporados dentro de outro documento HTML. Em nosso website de teste temos um link chamado ***WYSIWYG Editor***.

Se inspecionarmos este editor, veremos que ele está dentro de uma tag HTML, que está embutida nesta página. Este documento HTML é filho deste **<iframe>** que representa quadros.

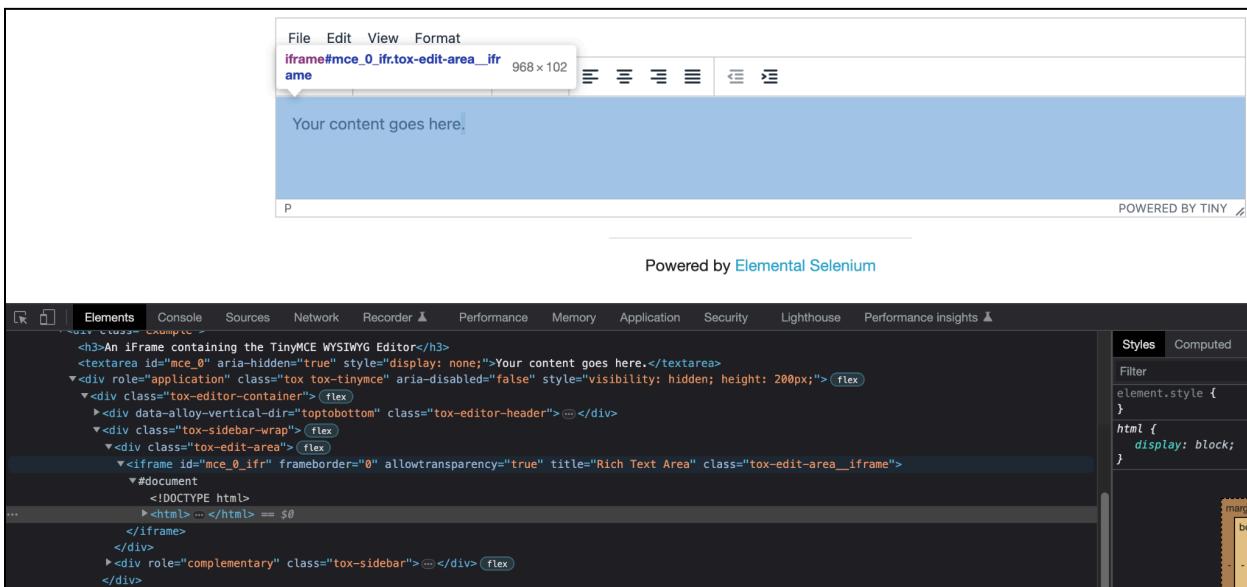


Imagen 78: Documento HTML dentro de um <iframe>

Se quisermos entrar em um documento embutido, ou um **<iframe>** neste caso, precisaremos mudar o contexto para esse quadro, semelhante ao que fizemos em um capítulo anterior com *alerts*.

Vamos criar um cenário de teste para entendermos como isso funciona:

1. Na página inicial, clicaremos no link ***WYSIWYG Editor***,
2. Limparemos este campo antes de tentar digitar outro texto;
3. Digitaremos "**Hello**" e depois voltaremos ao cabeçalho do editor, que está em um quadro diferente
4. Clicaremos em um dos botões da barra de ferramentas
5. Digitaremos a palavra "**World**"
6. Verificaremos o texto digitado

Em nossa classe “**HomePage.java**” criei o método `clickWysiwygEditor()`, que irá clicar no link **WYSIWYG Editor** e retornar o objeto “**WysiwygEditorPage**”.

```
public WysiwygEditorPage clickWysiwygEditor() {
    clickLink("WYSIWYG Editor");
    return new WysiwygEditorPage(driver);
}
```

Também já criei a classe “**WysiwygEditorPage.java**” com seu método construtor:

```
package pages;

import org.openqa.selenium.WebDriver;

public class WysiwygEditorPage {

    private WebDriver driver;

    public WysiwygEditorPage(WebDriver driver) {
        this.driver = driver;
    }
}
```

Uma vez que chegamos a página **WYSIWYG Editor**, a primeira coisa que queremos fazer é limpar este editor, e para isso precisamos encontrar o localizador para este editor. Como ele está dentro de um `<iframe>`, antes teremos que mudar o contexto do WebDriver para este `iframe`.

Na classe criada acima vamos adicionar um método chamado `switchToEditArea()` que será privado. O usuário não tem ideia de que isso está dentro de um `iframe`, portanto, nosso teste não precisa se preocupar com esses tipos de detalhes de implementação.

Dentro deste método digitaremos `driver.switchTo()`. e veremos um quadro com algumas opções para frame, mas utilizaremos aquela que receberá uma String com o *id* do `iframe` que será mapeado em seguida:

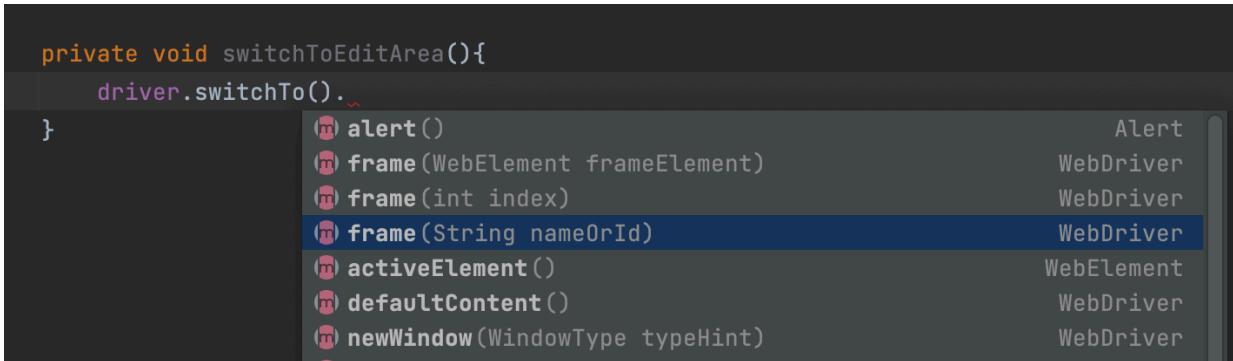


Imagem 79: Métodos para `switchTo()`

O id que usaremos será o seguinte:

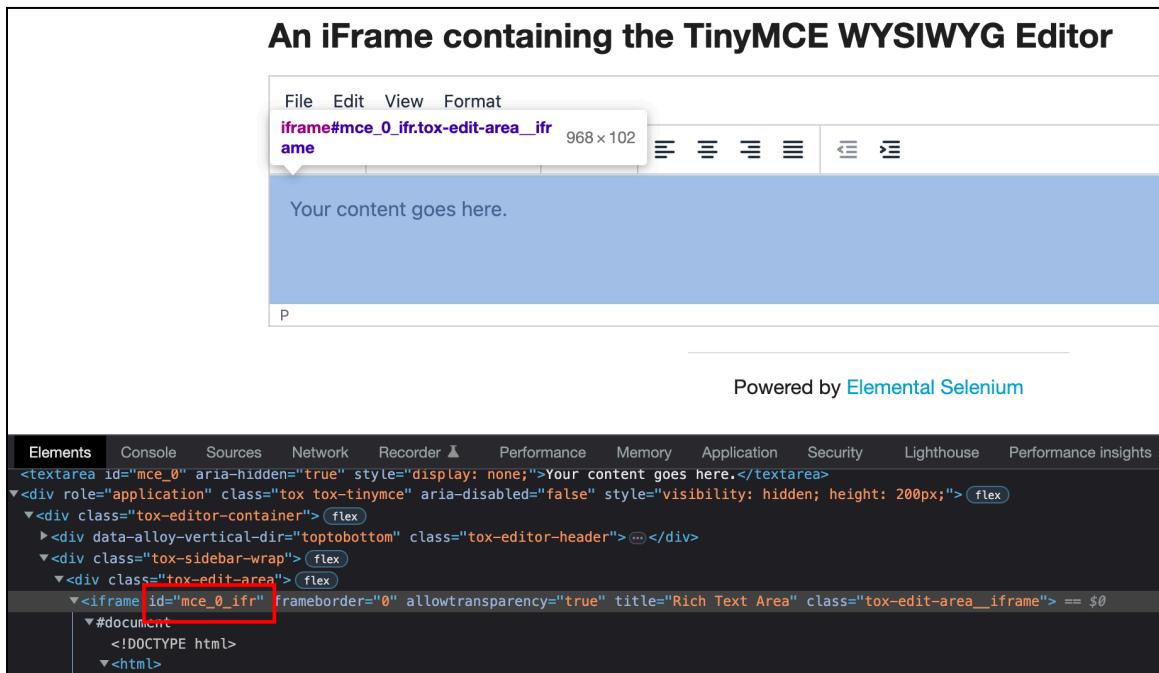


Imagen 80: id do iframe

Vamos criar um localizador sem usar o objeto **By**. Não precisamos de um `WebElement`, pois o método `driver.switchTo().frame()` recebe uma String de um id.

```
private String editorIframeId = "mce_0_ifr";
```

E então, de volta ao nosso método:

```
private void switchToEditArea() {
    driver.switchTo().frame(editorIframeId);
}
```

Isso mudará o contexto do DOM da página para este DOM do `iframe`. Assim que estivermos dentro desse iframe, as únicas coisas que podemos acessar são os elementos dentro dessa tag HTML.

Dentro deste documento HTML temos uma tag `<body>` que tem um atributo `id="tinymce"`, então vamos usá-lo como seletor para criar um `WebElement` By em nossa classe de *Page Object*.

```
private By textArea = By.id("tinymce");
```

Lembre-se de que precisamos limpar essa área antes de podermos fazer qualquer coisa nela, então vamos criar um método para isso. Este método será chamado de

`clearTextArea()` é o primeiro vai mudar o contexto para o *iFrame*, em seguida fazer clear do elemento `textArea`, e depois mudar o contexto de volta para o DOM principal:

```
public void clearTextArea() {
    switchToEditArea();
    driver.findElement(textArea).clear();
    switchToMainArea();
}
```

Acima usamos um método chamado `switchToMainArea()` que ainda não havia sido criado. Ele vai mudar o contexto de volta para o DOM principal que é o *parentFrame* deste *iFrame*:

```
private void switchToMainArea() {
    driver.switchTo().parentFrame();
}
```

O próximo passo é criar um método para enviar texto para a `textArea`. Vai ser parecido com o `clearTextArea`, mas ao invés de fazer `clear()` ele vai enviar uma String de texto recebida pelo método. Vamos chamá-lo de `setTextArea(String text)`:

```
public void setTextArea(String text) {
    switchToEditArea();
    driver.findElement(textArea).sendKeys(text);
    switchToMainArea();
}
```

O quarto passo de nosso teste será um clique em um botão da barra de ferramentas, como por exemplo aumentar a indentação.

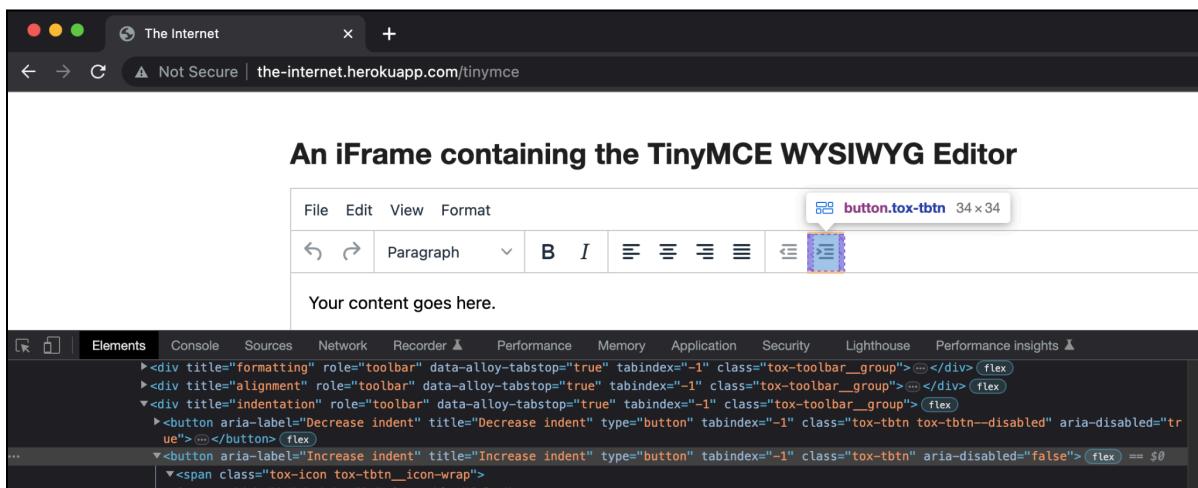


Imagen 81: Button para “Increase Indent”

Ao inspecionarmos este botão veremos que ele não está dentro do iframe, mas dentro do DOM principal, então não precisamos nos preocupar com mudanças de contexto. Ele não possui um id que possa ser usado como seletor, mas podemos usar um de seus atributos como um seletor CSS:

```
private By increaseIndentButton = By.cssSelector("button[title='Increase indent']);
```

Agora vamos criar um método para clicar nesse botão:

```
public void increaseIndention() {
    driver.findElement(increaseIndentButton).click();
}
```

O último passo de nosso teste será a verificação do conteúdo digitado. Para isso, precisamos criar um método que nos retorne o texto preenchido em `textArea`. Da mesma forma como no método `setTextArea` precisaremos mudar os contextos para o iFrame e retornar o contexto para o DOM principal antes de retornar o valor armazenado na String `text`. O resultado final de nossa classe de *Page Object* será o seguinte:

Código Final
WysiwygEditorPage.java

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class WysiwygEditorPage {

    private WebDriver driver;
    private String editorIframeId = "mce_0_ifr";
    private By textArea = By.id("tinymce");
    private By increaseIndentButton = By.cssSelector("button[title='Increase indent']);

    public WysiwygEditorPage(WebDriver driver) {
        this.driver = driver;
    }

    private void switchToEditArea(){
        driver.switchTo().frame(editorIframeId);
    }

    private void switchToMainArea(){
        driver.switchTo().parentFrame();
    }

    public void clearTextArea(){
        switchToEditArea();
        driver.findElement(textArea).clear();
        switchToMainArea();
    }

    public void setTextArea(String text){
```

```

        switchToEditArea();
        driver.findElement(textArea).sendKeys(text);
        switchToMainArea();
    }

    public String getTextFromEditor() {
        switchToEditArea();
        String text = driver.findElement(textArea).getText();
        switchToMainArea();
        return text;
    }

    public void increaseIndention(){
        driver.findElement(decreaseIndentButton).click();
    }
}

```

Agora vamos escrever nosso teste. No diretório “**test/java**” vamos criar um novo pacote que chamaremos de “**frames**” e nele vamos criar uma nova classe chamada “**FrameTest.java**”.

Da mesma forma como nas demais classes de teste, esta estende a classe “**BaseTests.java**” e terá um novo teste chamado de **testWysiwyg**. Este teste vai executar os passos descritos anteriormente através dos métodos que criamos em nossa classe de *Page Object*:

Código Final
FrameTest.java

```

package frames;

import base.BaseTests;
import org.testng.annotations.Test;

import static org.testng.Assert.assertEquals;

public class FrameTest extends BaseTests {

    @Test
    public void testWysiwyg(){
        WysiwygEditorPage editorPage = homepage.clickWysiwygEditor();
        editorPage.clearTextArea();
        String text1 = "hello ";
        String text2 = "world";
        editorPage.setTextArea(text1);
        editorPage.decreaseIndention();
        editorPage.setTextArea(text2);
        assertEquals(editorPage.getTextFromEditor(), text1+text2, "Text from
editor is incorrect");
    }
}

```

Capítulo 13 - Estratégias de *Wait*

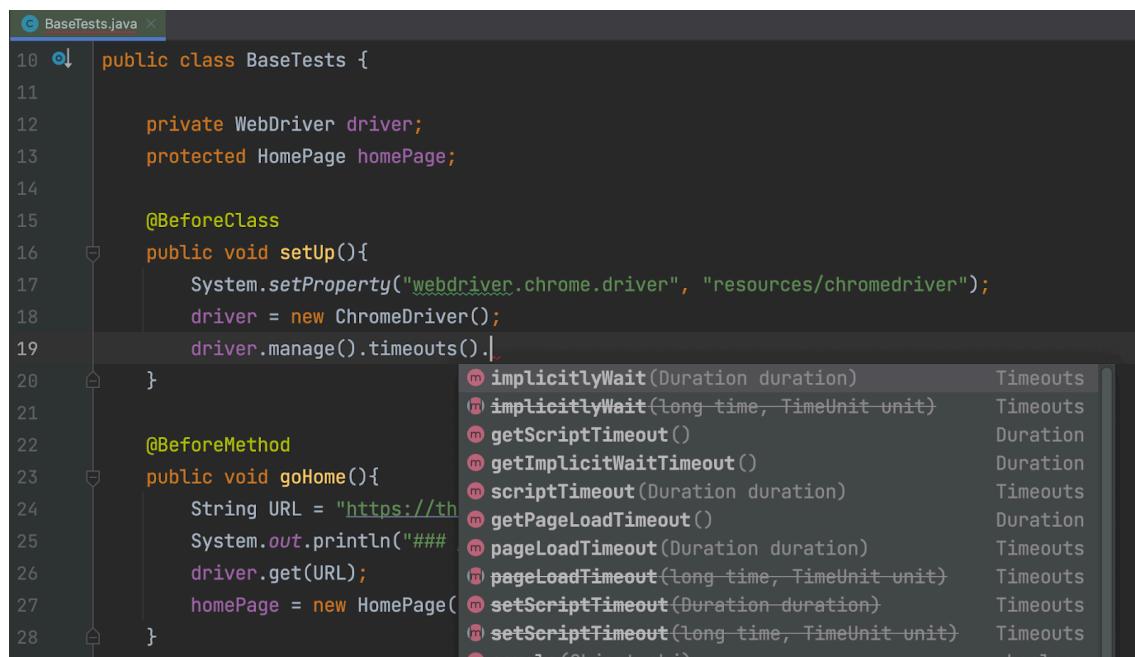
Neste capítulo, vamos discutir Estratégias de Espera (*Wait*).

Às vezes testamos um site que pode demorar um pouco para carregar seus elementos. Ou eles podem carregar dinamicamente com base no que você está fazendo.

Por exemplo, você pode clicar em um botão que gera alguma ação em segundo plano e a interface do usuário pode mostrar um *spinner* ou algum tipo de indicador informando que uma operação está sendo processada. Seu script não saberá lidar com isso, a menos que você diga a ele para fazer isso, e existem várias maneiras de fazer isso.

13.1 Esperas Implícitas

Vamos primeiro falar sobre a espera implícita. Você pode adicionar uma instrução em seu script para dizer: "Quero que você espere até esse período de tempo". Por exemplo, depois de criar nosso *ChromeDriver*, poderíamos criar um objeto *Timeouts* escrevendo o comando `driver.manage().timeouts()`, e veremos que existem vários métodos disponíveis. Um deles é o `implicitlyWait`:



```

10  public class BaseTests {
11
12      private WebDriver driver;
13      protected HomePage homePage;
14
15      @BeforeClass
16      public void setUp(){
17          System.setProperty("webdriver.chrome.driver", "resources/chromedriver");
18          driver = new ChromeDriver();
19          driver.manage().timeouts().|
```

Imagen 82: Métodos disponíveis em “`timeouts()`”

Este método precisa receber como argumento um objeto Duration que define a unidade de tempo e recebe um valor de duração:

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(30));
```

Isso significa que sempre que o *WebDriver* precisar interagir com um elemento, ele deve checar o site (fazer *polling*) por até 30 segundos até encontrar esse elemento.

- Se encontrar o elemento antes de 30 segundos, o *WebDriver* interage com ele.
- Caso contrário, o *WebDriver* continuará checando até encontrar o elemento ou até que os 30 segundos se esgotem.
- Se os 30 segundos se esgotarem e o *WebDriver* não encontrar o elemento, ele lançará uma exceção **NoSuchElementException**.

Os 30 segundos que temos aqui são apenas um exemplo. Você poderia usar outro valor e outra unidade de tempo.

13.2 Esperas Explícitas

Existe outra abordagem que nos permite fazer esperas explícitas, que usaremos apenas quando precisamos que nosso aplicativo espere uma determinada quantidade de tempo fixa.

Em nossa página inicial existe um link chamado **Dynamic Loading**, onde teremos 2 exemplos:

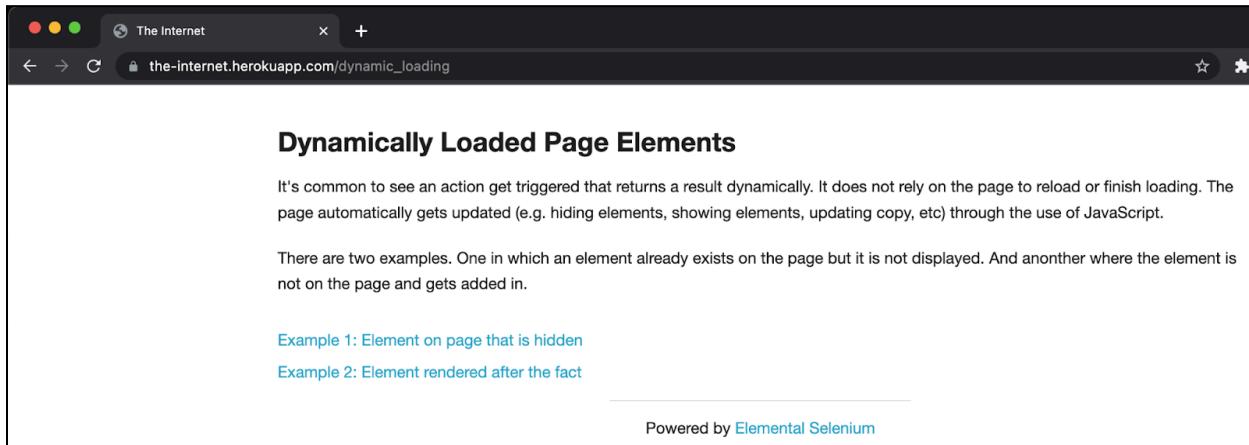


Imagen 83: Página Dynamic Loading

Para nosso próximo teste usaremos a primeira opção “**Example 1: Element on page that is hidden**” que nos levará à outra página:

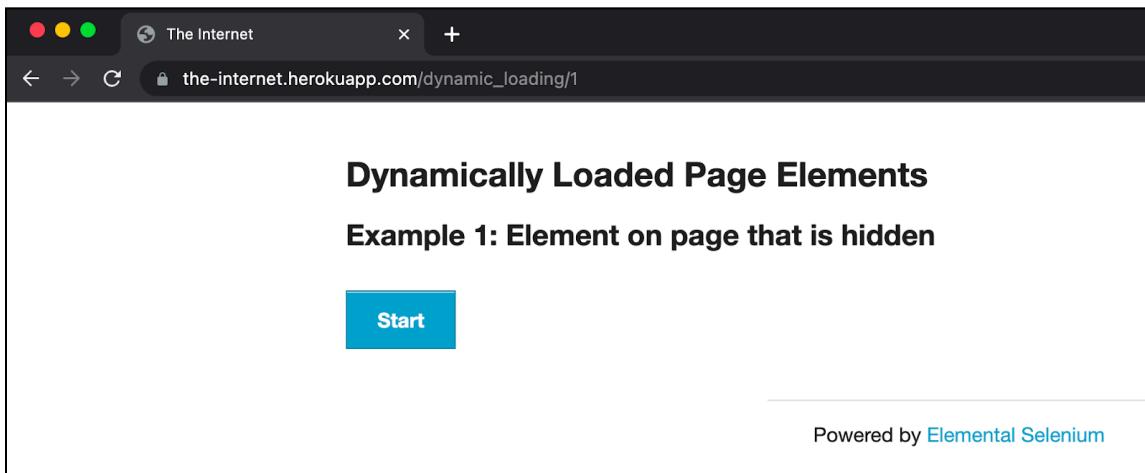


Imagen 84: “Example 1: Element on page that is hidden”

Nesta página clicaremos no botão “**Start**” e será apresentado uma barra de carregamento “**Loading**”.

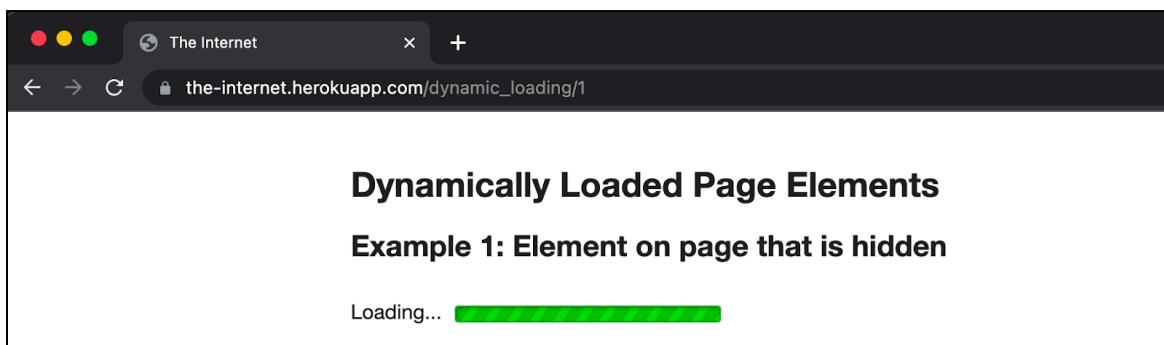


Imagen 85: Loading...

Assim que terminar de carregar, ele apresenta a mensagem “**Hello World!**”

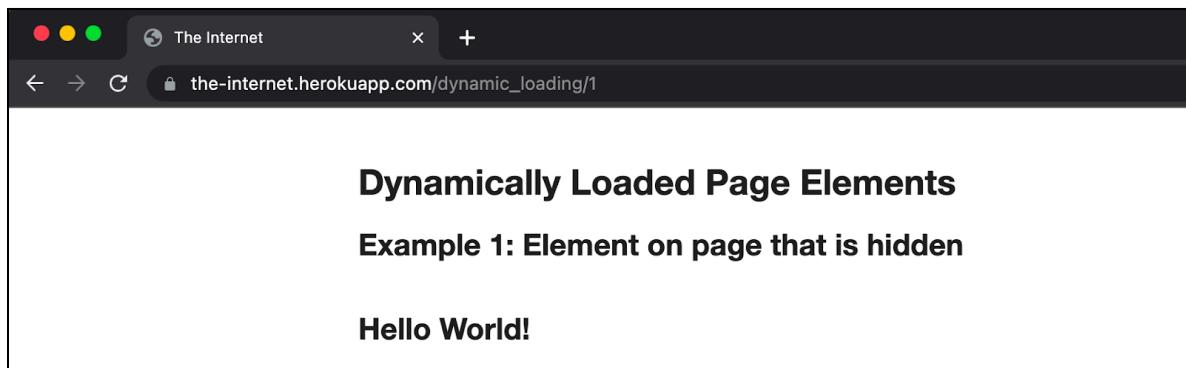


Imagen 86: Hello World!

Portanto, precisamos criar um teste que possa verificar o carregamento e, em seguida, afirmar que a mensagem “**Hello World!**” foi apresentada com o texto correto.

Adicionei um novo método `clickDynamicLoadingLink()` que irá clicar no link **Dynamic Loading** e retornar o objeto “**DynamicLoadingPage**”:

```
public DynamicLoadingPage clickDynamicLoadingLink() {
    clickLink("Dynamic Loading");
    return new DynamicLoadingPage(driver);
}
```

Em seguida criamos a classe “**DynamicLoadingPage.java**”:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class DynamicLoadingPage {

    private WebDriver driver;

    private String linkXpath_Format = ".//a[contains(text(), '%s')}";
    private By link_Example1 = By.xpath(String.format(linkXpath_Format,
    "Example 1"));

    public DynamicLoadingPage(WebDriver driver) {
        this.driver = driver;
    }

    public DynamicLoadingExample1Page clickExample1Link() {
        driver.findElement(link_Example1).click();
        return new DynamicLoadingExample1Page(driver);
    }
}
```

Lembre-se de que havia dois links de exemplo nesta página da web. Criamos um objeto **By** para ser o seletor do primeiro exemplo.

Os links de ambos os exemplos tinham a mesma fórmula, não havia identificação específica neles. Vamos utilizar o texto, mas não queremos colocar o texto inteiro lá porque é longo e pode mudar.

Então, criaremos uma expressão Xpath para dizer, “*procure uma tag <a> que contenha esta String*”, e então, criaremos o elemento **By Link_Example1**, para o elemento que possui o formato informado e a String com o texto “**Example 1**”. Se quisermos adicionar um link para o “**Example 2**”, podemos fazer isso facilmente seguindo a mesma fórmula.

Além disso, nesta classe temos nosso construtor, bem como um método `clickExample1`, que clicará no primeiro link de exemplo e nos levará ao para a página representada pela classe “**DynamicLoadingExample1Page.java**” que criaremos agora:

```

package pages;

import org.openqa.selenium.WebDriver;

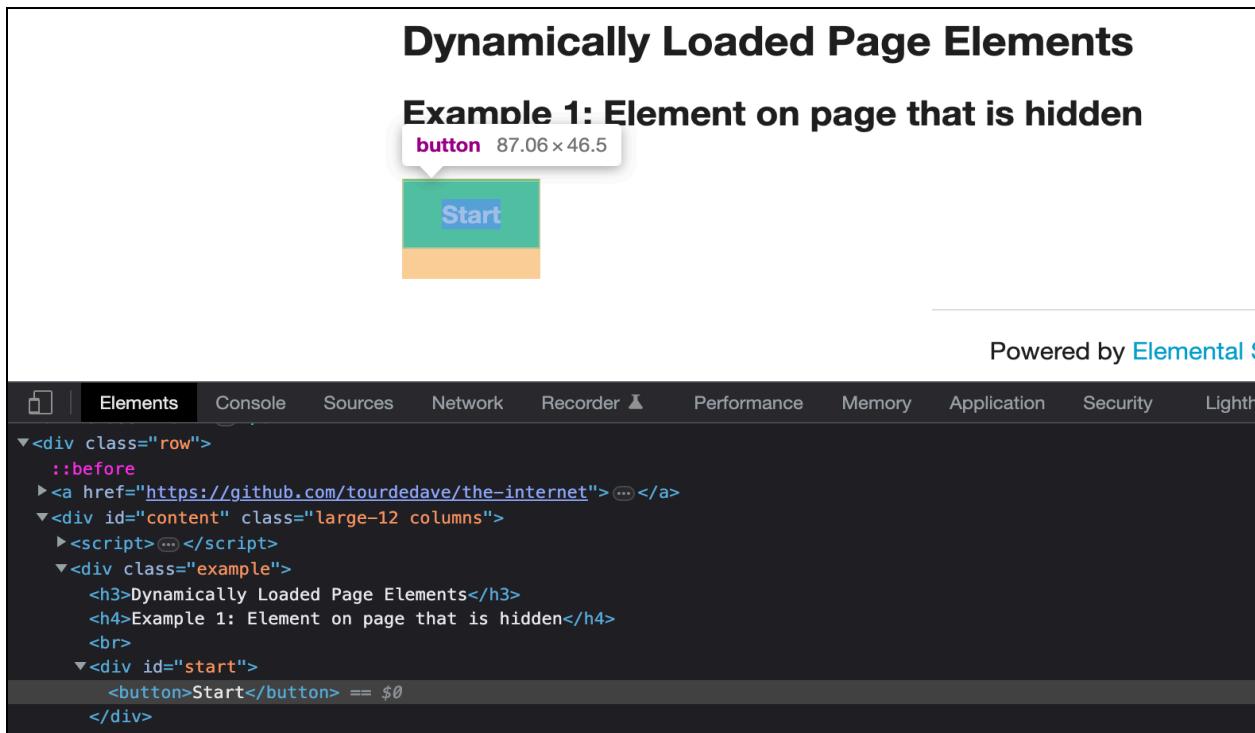
public class DynamicLoadingExample1Page {

    private WebDriver driver;

    public DynamicLoadingExample1Page(WebDriver driver) {
        this.driver = driver;
    }
}

```

Vamos agora adicionar os elementos que precisamos para nosso teste. A primeira coisa que queremos fazer é clicar no botão “**Start**”:



The screenshot shows the Chrome DevTools Elements tab. At the top, it says "Dynamically Loaded Page Elements" and "Example 1: Element on page that is hidden". Below that is a tooltip for a button element with the text "button 87.06 x 46.5". The button itself is green with the word "Start" in white. In the bottom left corner of the screenshot, there is a watermark that says "Powered by Elemental S". The main content area shows the DOM structure with a

element having an id of "start" containing a element.

```

<div class="row">
  <div id="content" class="large-12 columns">
    <div id="start">
      <button>Start</button> == $0
    </div>
  </div>
</div>

```

Imagen 87: Inspecionando o botão “Start”

Não há identificadores específicos no botão, mas a tag `<div>` na qual ele está inserido tem um `id="iniciar"`. Vamos adicionar isso ao nosso framework:

```

private By startButton = By.cssSelector("#start button");

```

E vamos criar um método para clicar nesse botão:

```

public void clickStart() {
    driver.findElement(startButton).click();
}

```

Quando clicarmos neste botão não seremos levados a uma nova página, então não precisamos nos preocupar em retornar uma nova página.

No entanto, ele mostrou no inspetor de elementos uma nova tag <div>, e quando passamos o mouse sobre ele vemos a barra de loading em destaque e sabemos que é o elemento certo, e ele possui um ***id="Loading"***:

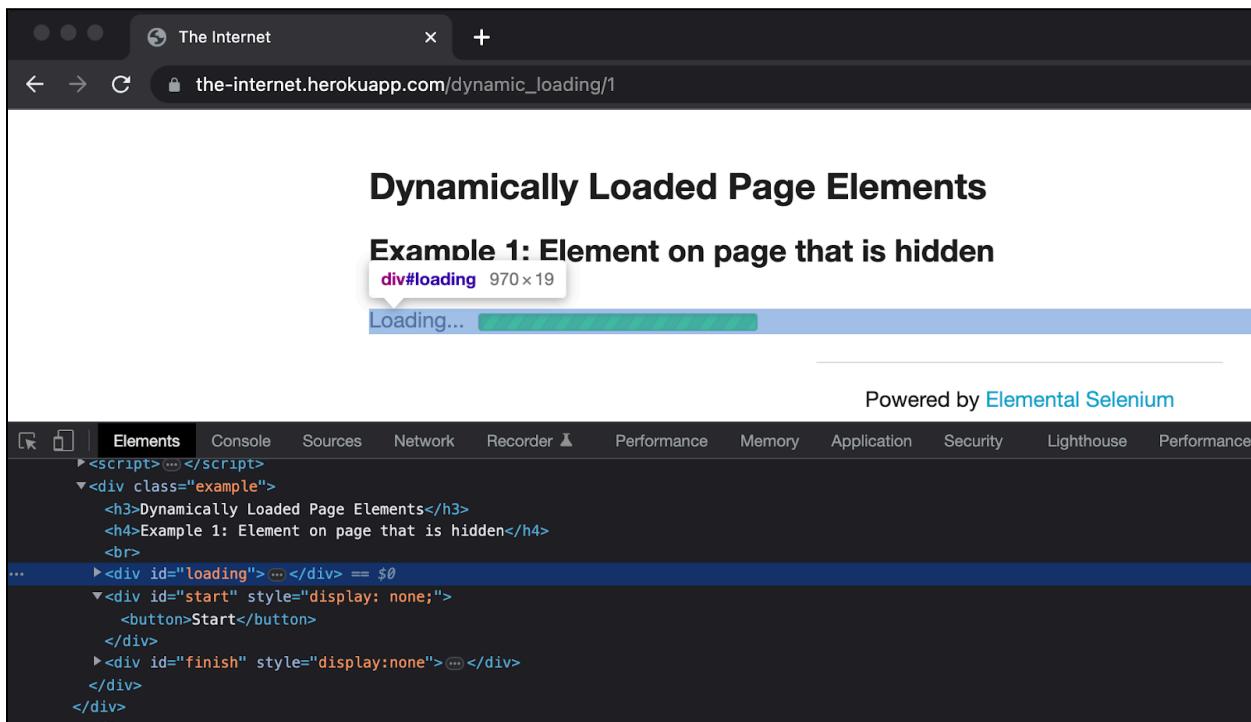


Imagen 88: Inspecionando o elemento Loading

Vamos adicionar isso ao nosso framework também:

```
private By loadingIndicator = By.id("loading");
```

E, finalmente, queremos obter o texto “**Hello World!**”. Vamos criar o seletor para este elemento com o ***id="finish"***:

```
private By loadedText = By.id("finish");
```

Assim, após clicarmos no botão “**Start**”, sabemos que o indicador de carregamento estará lá, e teremos que esperar que ele termine.

Como apresentamos isso nesta classe? Uma boa maneira de fazer isso é no próprio método ***clickStart()***.

Então, depois de clicar, ele precisa esperar até que o aplicativo esteja pronto antes de devolver o controle ao teste. Portanto, não queremos que nossos testes tenham que pensar em chamar um método para esperar por isso.

Quando ele clica em “**Start**”, o teste apenas precisa saber que o controle não será devolvido até que o carregamento seja concluído. Então, neste método, depois de clicarmos vamos adicionar a funcionalidade para a espera explícita.

O *WebDriver* tem uma classe chamada *WebDriverWait*, e isso está no pacote de suporte de UI. O construtor *WebDriverWait* aceita dois argumentos — o *driver* e também o tempo limite em segundos com *Duration*. Digamos que queremos esperar 5 segundos:

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
```

O tempo limite em segundos também fará uma checagem, assim como a espera implícita que criamos antes. No entanto, a espera implícita era para todo o aplicativo, enquanto este *wait* é apenas para este método específico. Então, estamos dizendo ao *WebDriver* para checar o aplicativo por até 5 segundos.

Esta instrução por si só não fará nada — é apenas uma instrução de configuração para criar este objeto de espera. Agora temos que dizer *wait.until* e definirmos alguma condição esperada com a classe *ExpectedConditions* que também está no pacote “*support.ui*”:



```
public void clickStart(){
    driver.findElement(startButton).click();
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
    wait.until(ExpectedConditions.);
}
```

Imagen 89: Classe “ExpectedConditions”

A classe *ExpectedConditions* contém todos os tipos de métodos que permitem esperar que alguma condição seja atendida antes de prosseguir. Esta classe é muito completa e oferece muitas condições diferentes para você escolher.

Então, temos que nos perguntar, o que estamos esperando? Vamos olhar para o aplicativo novamente.

Quando clicamos no botão “**Start**”, vemos que o ícone de carregamento está aqui e antes de prosseguirmos, precisamos esperar até que ele desapareça. Vamos ver se podemos

encontrar uma condição para dizer, “espere até que aquele elemento de carregamento não esteja mais lá”.

Vemos que existe um método chamado `invisibilityOf`, então podemos usar este. `invisibilityOf` recebe um `WebElement`, então passaremos a ele nosso objeto `By` que criamos, o “`loadingIndicator`”:

```
wait.until(ExpectedConditions.invisibilityOf(
    driver.findElement/loadingIndicator)));
```

Este método clicará no botão “`Start`”. Em seguida, aguardará até 5 segundos para que o indicador de carregamento fique invisível. Se após 5 segundos ele não estiver invisível, isso lançará uma exceção.

Novamente, o tempo limite é a quantidade máxima de tempo que ele esperará. No entanto, se descobrir que o indicador de carregamento está invisível após 1 segundo, ele continuará após 1 segundo.

A última coisa que precisamos fazer para esta classe de *Page Object* é fornecer um método que retorne o texto carregado:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import java.time.Duration;

public class DynamicLoadingExample1Page {

    private WebDriver driver;
    private By startButton = By.cssSelector("#start button");
    private By loadingIndicator = By.id("loading");
    private By loadedText = By.id("finish");

    public DynamicLoadingExample1Page(WebDriver driver) {
        this.driver = driver;
    }

    public void clickStart() {
        driver.findElement(startButton).click();
        WebDriverWait wait = new WebDriverWait(driver,
Duration.ofSeconds(5));
        wait.until(ExpectedConditions.invisibilityOf(
            driver.findElement(loadingIndicator)));
    }

    public String getLoadedText() {
        return driver.findElement(loadedText).getText();
    }
}
```

Agora vamos escrever nosso teste. No diretório “**test/java**” vamos criar um novo pacote que chamaremos de “**wait**” e nele vamos criar uma nova classe chamada “**WaitTests.java**”.

Da mesma forma como nas demais classes de teste, esta estende a classe “**BaseTests.java**” e terá um novo teste chamado de **testWaitUntilHidden**. Este teste vai executar os passos descritos anteriormente através dos métodos que criamos em nossas classes de Page Object **clickDynamicLoadingLink()** e **clickExample1Link()**:

```
public class WaitTests extends BaseTests {

    @Test
    public void testWaitUntilHidden() {
        var dynamicLoadingExample1Page =
            homePage.clickDynamicLoadingLink().clickExample1Link();
    }
}
```

Vamos agora clicar no botão “**Start**” que irá iniciar e aguardar o carregamento ser concluído. Em seguida faremos uma *assertion* para verificar se o texto de confirmação foi apresentado corretamente. O resultado final da classe de testes ficará assim:

Código Final
WaitTests.java

```
package wait;

import base.BaseTests;
import org.testng.annotations.Test;

import static org.testng.Assert.assertEquals;

public class WaitTests extends BaseTests {

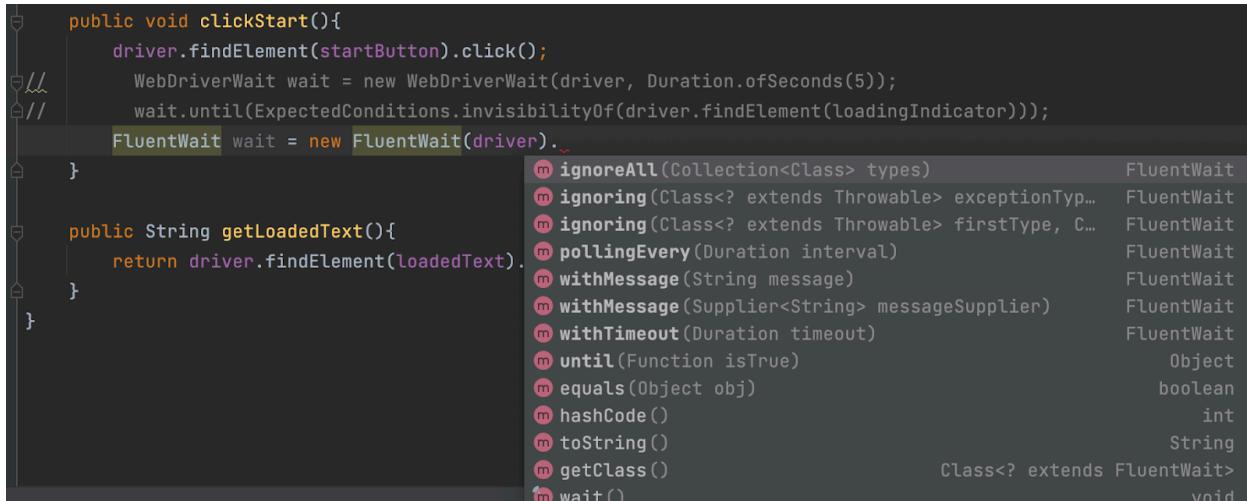
    @Test
    public void testWaitUntilHidden() {
        DynamicLoadingExample1Page dynamicLoadingExample1Page =
            homePage.clickDynamicLoadingLink().clickExample1Link();
        dynamicLoadingExample1Page.clickStart();
        assertEquals(dynamicLoadingExample1Page.getLoadedText(), "Hello
World!", "Loaded text incorrect");
    }
}
```

13.3 Esperas Fluentes

Há também o conceito de Fluent Waits, que oferecem um pouco mais de flexibilidade na criação desse objeto de *wait* do WebDriver. Além de indicar o tempo limite, você também pode

informar com que frequência ele deve fazer *polling* checando a condição esperada, e também você pode especificar quaisquer exceções que deseja ignorar.

Vejamos este mesmo exemplo com *FluentWait* no método `clickStart()`:



```

public void clickStart(){
    driver.findElement(startButton).click();
    WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
    wait.until(ExpectedConditions.invisibilityOf(driver.findElement/loadingIndicator)));
}
public String getLoadedText(){
    return driver.findElement(loaderText);
}

```

Imagen 90: Alterando a estratégia de wait do método `clickStart()`

Primeiro vamos definir o timeout com `.withTimeout()` e passando Duration com 5 segundos.

Em seguida vamos definir o intervalo de *polling* para que a checagem seja feita à cada 1 segundo, e em seguida instruir o comando para que a exceção NoSuchElementException seja ignorada. Você pode especificar quaisquer outras exceções que desejar ignorar, e isso é equivalente à criação do objeto *wait* para a espera explícita feita anteriormente. Para que ele realmente faça alguma coisa, precisamos repetir o *wait.until* com a mesma condição esperada:

```

FluentWait wait = new FluentWait(driver)
    .withTimeout(Duration.ofSeconds(5))
    .pollingEvery(Duration.ofSeconds(1))
    .ignoring(NoSuchElementException.class);
wait.until(ExpectedConditions.invisibilityOf(driver.findElement/loadingIndicator)));

```

Capítulo 14 - Usando *JavaScript*

Embora a biblioteca *WebDriver* tenha muitos métodos integrados, às vezes há ações necessárias que não são suportadas diretamente. Para esses casos, o *WebDriver* fornece uma maneira de permitir a execução do *JavaScript* no navegador. Todos os principais navegadores da Web possuem mecanismos *JavaScript* dedicados. Portanto, qualquer coisa que não possamos fazer diretamente com a API do *WebDriver*, podemos escrever nosso próprio código *JavaScript*.

Um exemplo comum do uso de *JavaScript* em projetos de automação de teste é rolar uma página (fazer *scroll*). Surpreendentemente, não há uma funcionalidade interna para fazer isso com o *WebDriver*, então precisaremos escrever nosso próprio *JavaScript* para fazer isso.

Vejamos um exemplo clicando no link ***Large & Deep DOM*** da página inicial de nosso site de testes:

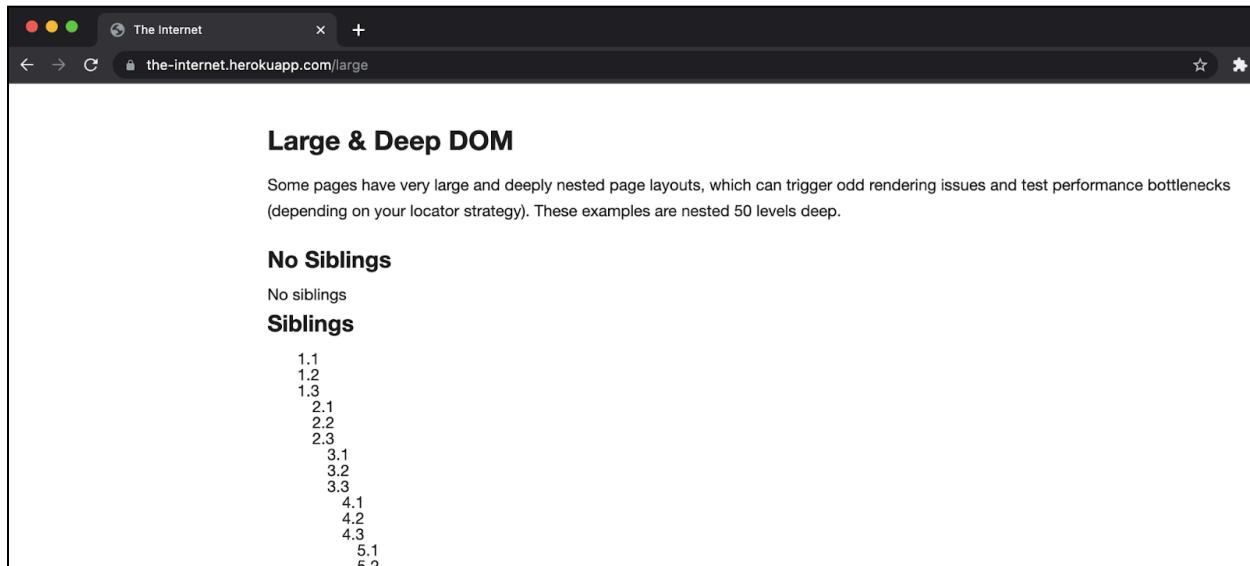


Imagen 91: Página *Large & Deep DOM*

Observe nas barras de rolagem que esta página é longa nos sentidos horizontal e vertical:

Imagen 92: Rolagem Horizontal e Vertical

Vamos escrever um código que fará scroll para baixo até que o elemento da tabela esteja visível.

Antes temos que criar um método dentro da nossa classe “**HomePage.java**” cara clicar no link “**Large & Deep DOM**”:

```
public LargeAndDeepDomPage clickLargeAndDeepDom() {  
    clickLink("Large & Deep DOM");  
    return new LargeAndDeepDomPage(driver);  
}
```

Dentro da classe `LargeAndDeepDomPage.java` que será instanciada criamos um construtor e um objeto `By` para a tabela:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;

public class LargeAndDeepDomPage {

    private WebDriver driver;
    private By table = By.id("large-table");

    public LargeAndDeepDomPage(WebDriver driver) {
        this.driver = driver;
    }
}
```

Agora criaremos um método que usará *JavaScript* para fazer *scroll* até esta tabela. A primeira coisa que precisamos fazer é criar o *WebElement* usando o objeto ***Bv*** dessa tabela:

```
public void scrollToTable(){
    WebElement tableElement = driver.findElement(table);
}
```

Agora que temos a tabela, queremos fazer *scroll* até ela. É aqui que precisaremos do *JavaScript*, pois não há nenhum método neste elemento de tabela que permita fazer *scroll* até ele. Então, o que faremos é usar essa classe *JavascriptExecutor*, e vemos que ela também pertence à biblioteca *Selenium*:

```
public void scrollToTable(){
    WebElement tableElement = driver.findElement(table);
    JavascriptExecutor
} I JavascriptExecutor org.openqa.selenium
```

Press ^ to choose the selected (or first) suggestion and insert a dot afterwards **Next Tip**  

Imagen 93: Classe JavascriptExecutor

Esta é uma classe que o *Selenium* fornece para nos permitir executar o *JavaScript*. Para utilizar isso, temos que transmitir nosso driver para *JavascriptExecutor* colocando ele entre parênteses e, em seguida, adicionando nosso driver depois dele. Isso lançará este objeto. Vamos colocar tudo entre parênteses para que possamos usá-lo e acessar o método *executeScript*:

```
public void scrollToTable(){
    WebElement tableElement = driver.findElement(table);
    ((JavascriptExecutor)driver).|
```

m executeScript(String script, Object... args)	Object
m executeScript(ScriptKey key, Object... args)	Object
m wait(long timeoutMillis)	void
m wait(long timeoutMillis, int nanos)	void

Imagen 94: Método executeScript da classe JavascriptExecutor

Antes disso, vamos criar uma String com o *JavaScript* que queremos executar, e depois passar este script e o *WebElement* como argumento:

```
package pages;

import org.openqa.selenium.By;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

public class LargeAndDeepDomPage {
```

Código Final
LargeAndDeepDomPage.java

```
public LargeAndDeepDomPage(WebDriver driver) {
    this.driver = driver;
}

public void scrollToTable() {
    WebElement tableElement = driver.findElement(table);
    String script = "arguments[0].scrollIntoView();";
    ((JavascriptExecutor)driver).executeScript(script, tableElement);
}
```

Agora, vamos criar um método de teste que rolará até que esta tabela esteja visível. Em nossa seção de teste, criei um novo pacote chamado “**javascript**” e uma classe chamada “**JavaScriptTests.java**”.

Em seguida criaremos o método de teste **testScrollToTable**:

```
package javascript;

import base.BaseTests;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

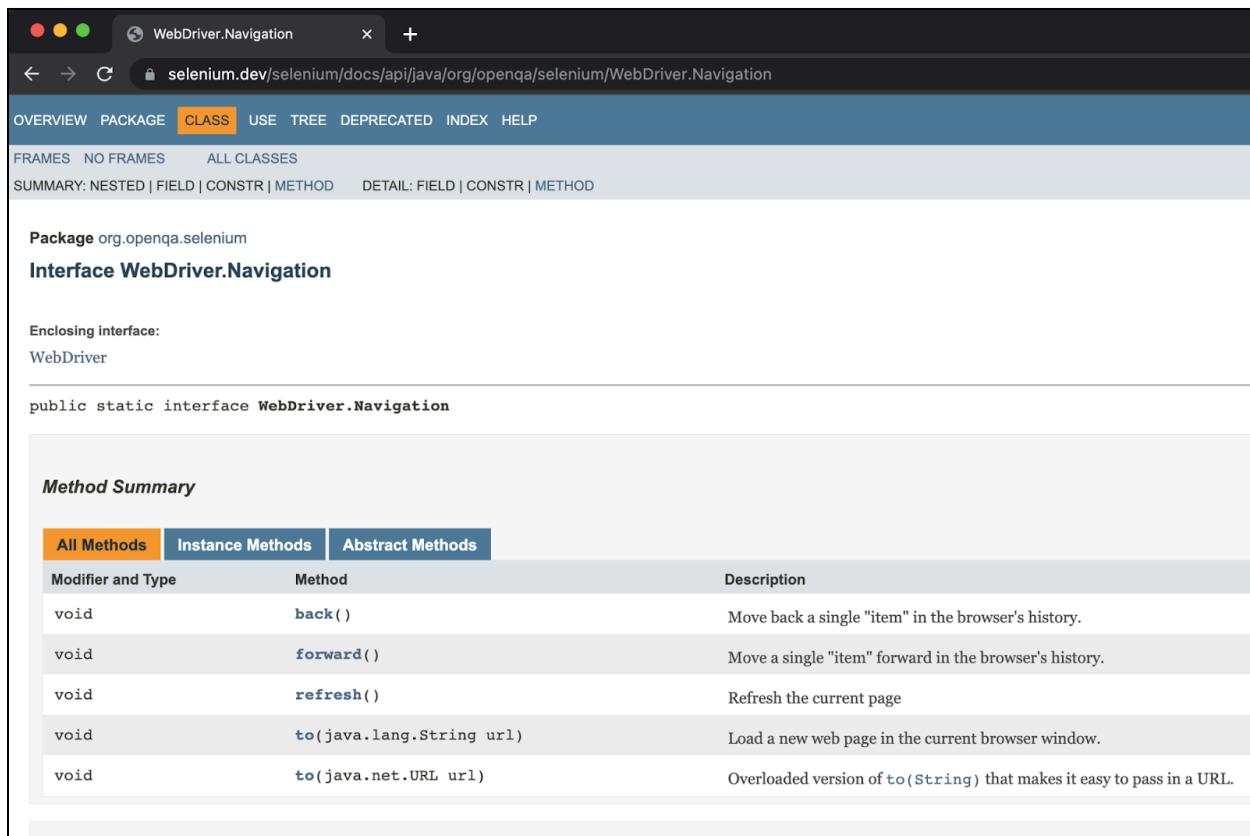
public class JavaScriptTests extends BaseTests {

    @Test
    public void testScrollToTable() {
        homePage.clickLargeAndDeepDom().scrollToTable();
    }
}
```

Neste teste não precisamos adicionar uma assertion pois o scroll só será feito corretamente se o elemento estiver visível.

Capítulo 15 - Navigation

Até agora clicamos nos links fornecidos para navegar pelo aplicativo. No entanto, o *WebDriver* também fornece outros métodos de navegação. O *WebDriver* fornece uma interface chamada ***Navigation*** e, nessa interface, existem vários métodos:



The screenshot shows the Java API documentation for the `WebDriver.Navigation` interface. The URL is `selenium.dev/selenium/docs/api/java/org/openqa/selenium/WebDriver.Navigation`. The interface is part of the `org.openqa.selenium` package. It has five methods: `back()`, `forward()`, `refresh()`, `to(java.lang.String url)`, and `to(java.net.URL url)`.

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method	Description
void	<code>back()</code>	Move back a single "item" in the browser's history.
void	<code>forward()</code>	Move a single "item" forward in the browser's history.
void	<code>refresh()</code>	Refresh the current page
void	<code>to(java.lang.String url)</code>	Load a new web page in the current browser window.
void	<code>to(java.net.URL url)</code>	Overloaded version of <code>to(String)</code> that makes it easy to pass in a URL.

Imagen 95: Métodos da interface Navigation

- `back` que navegará para a última página visitada no histórico do navegador
- `forward` que avançará no histórico do navegador em uma página
- `refresh` irá recarregar a página atual
- `to` pode receber uma String ou um objeto URL. Ambos vão diretamente para o URL especificado. O método `to` é muito semelhante ao método `get` que estamos usando, no entanto, `driver.get` aguarda o carregamento da página, mas este método não.

Vamos criar uma classe que nos permita utilizar esses métodos de navegação. Isso entrará na camada de *framework*, pois está realmente interagindo com o navegador. No entanto, não devemos colocá-lo no pacote “pages” porque ele se destina aos objetos de página que modelam páginas reais em nosso aplicativo.

Este é mais um método utilitário, então vamos criar outro pacote no mesmo nível do pacote “**pages**”, e vamos chamá-lo de “**utils**”.

Em seguida vamos criar uma que chamaremos de “**WindowManager.java**”. Agora, é claro, para que qualquer classe interaja com o navegador, ela precisa do *driver*. Vamos criar este objeto.

```
package utils;

import org.openqa.selenium.WebDriver;

public class WindowManager {
    private WebDriver driver;

    public WindowManager(WebDriver driver) {
        this.driver = driver;
    }
}
```

Agora vamos criar um método que irá voltar um item no histórico do navegador. A maneira de chegar à interface **Navigation** é dizer *driver.navigate()*, e então vemos os métodos que vimos no *JavaDoc*.

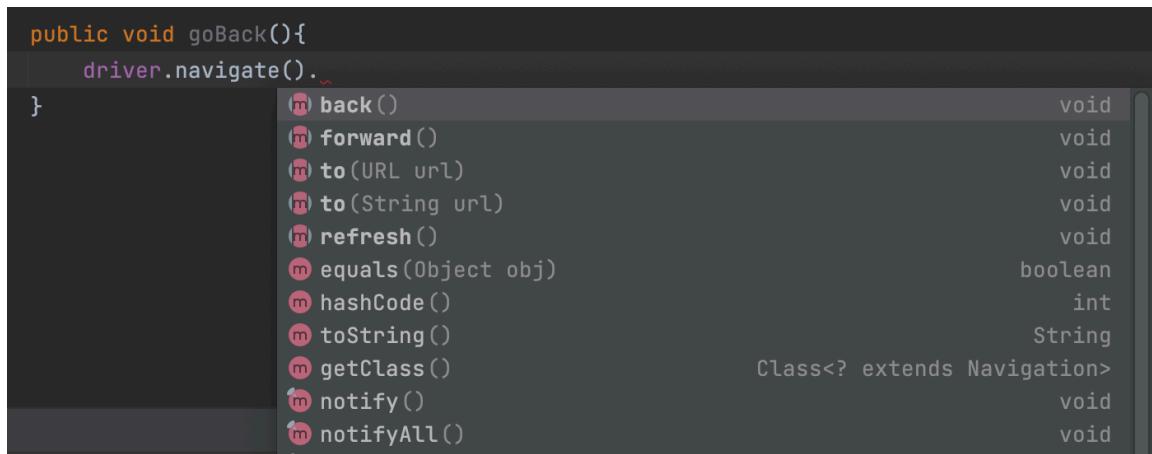


Imagen 96: Métodos da interface Navigation

E escolheremos o método **goBack**:

```
public void goBack() {
    driver.navigate().back();
}
```

Agora, se eu quiser criar um método para avançar ou atualizar, precisarei usar *driver.navigate()*. novamente, então podemos evitar isso criando um objeto para representá-lo e tornar as coisas mais simples. Em nosso método construtor ele será instanciado assim que tivermos o nosso *driver*.

```

private WebDriver driver;
private WebDriver.Navigation navigate;

public WindowManager(WebDriver driver) {
    this.driver = driver;
    navigate = driver.navigate();
}

```

Agora podemos alterar o método `goBack` para usar este objeto `navigate`:

```

public void goBack() {
    navigate.back();
}

```

Depois de criar os demais métodos de navigation, este será o resultado final da classe “`utils.java`”:

Código Final
WindowManager.java

```

package utils;

import org.openqa.selenium.WebDriver;

public class WindowManager {

    private WebDriver driver;
    private WebDriver.Navigation navigate;

    public WindowManager(WebDriver driver) {
        this.driver = driver;
        navigate = driver.navigate();
    }

    public void goBack() {
        navigate.back();
    }

    public void goForward() {
        navigate.forward();
    }

    public void refreshPage() {
        navigate.refresh();
    }

    public void goTo(String url) {
        navigate.to(url);
    }
}

```

Em nossas classes de *Page Object*, quando movemos para uma nova página com nossa ação, nosso método retornará um objeto apropriado como uma instância da classe da

página seguinte. No entanto, quando olhamos para os métodos implementados aqui, todos eles retornam void, então não há nenhum objeto de página para retornar.

Portanto, já que este é um método utilitário, caberá aos nossos testes acompanhar onde eles estão. Agora vamos escrever um pequeno método de teste que irá exercitá-los. Criaremos um novo pacote chamado “***navigation***” e uma nova classe chamada “***NavigationTests.java***”.

Já temos um *framework* que percorre algumas páginas. Podemos fazer ***homePage.clickDynamicLoading***, que nos retornará a página *Dynamic Loading*, e então poderíamos fazer ***clickExample1***, o que nos retornará uma instância da classe ***DynamicLoadingExample1Page***:

```
package navigation;

import base.BaseTests;
import org.testng.annotations.Test;

public class NavigationTests extends BaseTests {

    @Test
    public void testNavigator() {
        homePage.clickDynamicLoadingLink().clickExample1Link();
    }
}
```

Até aqui nosso teste navegou por algumas páginas. Depois de fazer isso, podemos querer navegar de volta para uma página anterior.

Para usar o ***WindowManager*** precisamos de um *driver*, porém nos testes não temos acesso ao *driver*, mas nosso ***BaseTests*** tem, onde vamos criar um método que nos retornará um objeto para o ***WindowManager***:

```
public WindowManager getWindowManager() {
    return new WindowManager(driver);
}
```

Vamos agora usar alguns métodos desta classe em nosso teste:

```
@Test
public void testNavigator() {
    homePage.clickDynamicLoadingLink().clickExample1Link();
    getWindowManager().goBack();
    getWindowManager().refreshPage();
    getWindowManager().goForward();
    getWindowManager().goTo("https://google.com");
}
```

Capítulo 16 - Capturando Screenshots

O Selenium WebDriver também fornece uma funcionalidade para tirar *screenshots* do navegador. Vamos adicionar o código para tirar uma captura de tela quando um teste for concluído.

Em nossa classe “**BaseTests.java**”, vamos criar outro método chamado **takeScreenshot** usando a annotation **@AfterMethod** e isso será executado após a execução de cada teste.

```
@AfterMethod  
public void takeScreenshot() {  
}
```

A primeira coisa que precisamos fazer é converter nosso *driver* para uma classe Selenium diferente chamada **TakesScreenshot** que está no pacote Selenium:

```
var camera = (TakesScreenshot)driver;
```

Agora temos um objeto para fazer a captura de tela e podemos salvar esta captura como um arquivo usando o pacote “**java.io**”.

Chamaremos essa captura de tela e a definiremos igual a **camera.getScreenshotAs()**, que usa um *OutputType* que definiremos como **FILE**:

```
TakesScreenshot camera = (TakesScreenshot)driver;  
File screenshot = camera.getScreenshotAs(OutputType.FILE);
```

Este é todo o código que você precisa para salvar a captura de tela como um arquivo, mas vamos em frente e imprimir o caminho onde essa captura de tela está armazenada.

```
System.out.println("Screenshot taken: " + screenshot.getAbsolutePath());
```

Agora vamos executar qualquer um dos nossos testes e devemos ter uma captura de tela logo após. Vamos executar este método **testBackspace()**:

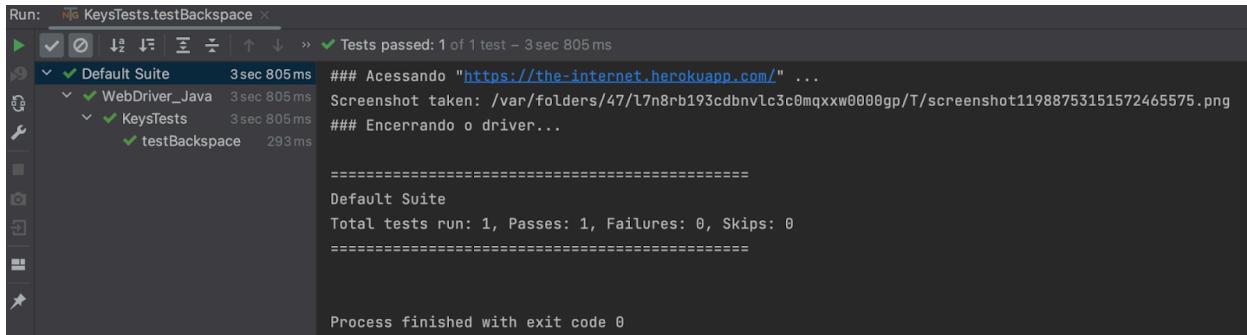


Imagen 97: Executando teste com captura de tela

Veremos que a captura de tela foi feita e o arquivo está sob esta pasta fora de nosso projeto, então podemos escrever mais uma linha de código para colocar a captura de tela em outra pasta que definiremos.

Voltando à nossa classe “**BaseTests.java**” e ao método ***takeScreenshot***, vamos mover o arquivo. Ao digitarmos `Files`. Veremos algumas opções para importar. Vamos escolher a disponível em `com.google.io`:

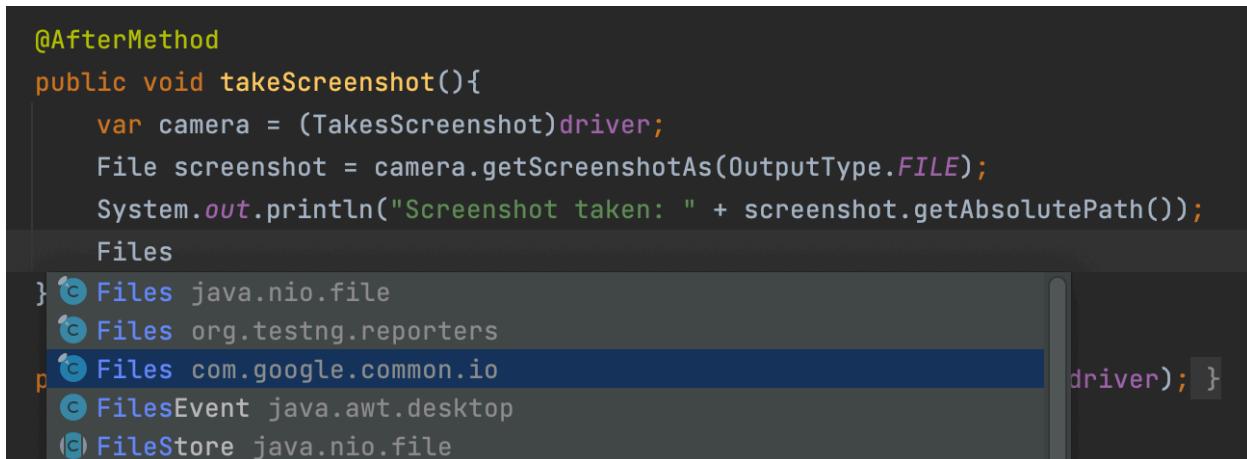


Imagen 98: Importando `com.google.common.io.Files`

Com a classe `Files` podemos usar o método ***Files.move*** informando que arquivo queremos mover (neste caso o objeto `screenshot` do tipo `File`) e para qual destino. Queremos movê-lo para dentro deste diretório “**resources**” dentro do nosso projeto.

Vou criar outro diretório aqui chamado “**screenshots**”, e é para onde gostaria de mover a imagem. Digamos “**resources/screenshots/**” e vamos dar um nome para o arquivo. Por enquanto, vamos apenas dizer “**test.png**”. Este será o objeto `File` de destino.

```

    @AfterMethod
    public void takeScreenshot(){
        var camera = (TakesScreenshot)driver;
        File screenshot = camera.getScreenshotAs(OutputType.FILE);
        System.out.println("Screenshot taken: " + screenshot.getAbsolutePath());
        Files.move(screenshot, new File( pathname: "resources/screenshots/test.png"))
    }

```

Unhandled exception: java.io.IOException
Add exception to method signature ⌂ More actions... ⌂

Imagen 99: IOException

Agora observe aqui que isso não está compilando porque diz que lançará uma exceção **IOException**. Para contornar isso, vamos envolver este código em um try catch que capture a exceção caso ela ocorra:

```

try{
    Files.move(screenshot, new File("resources/screenshots/test.png"));
} catch(IOException e){
    e.printStackTrace();
}

```

Executamos o teste novamente e veremos o arquivo de screenshot gerado movido para o diretório definido em nosso projeto:

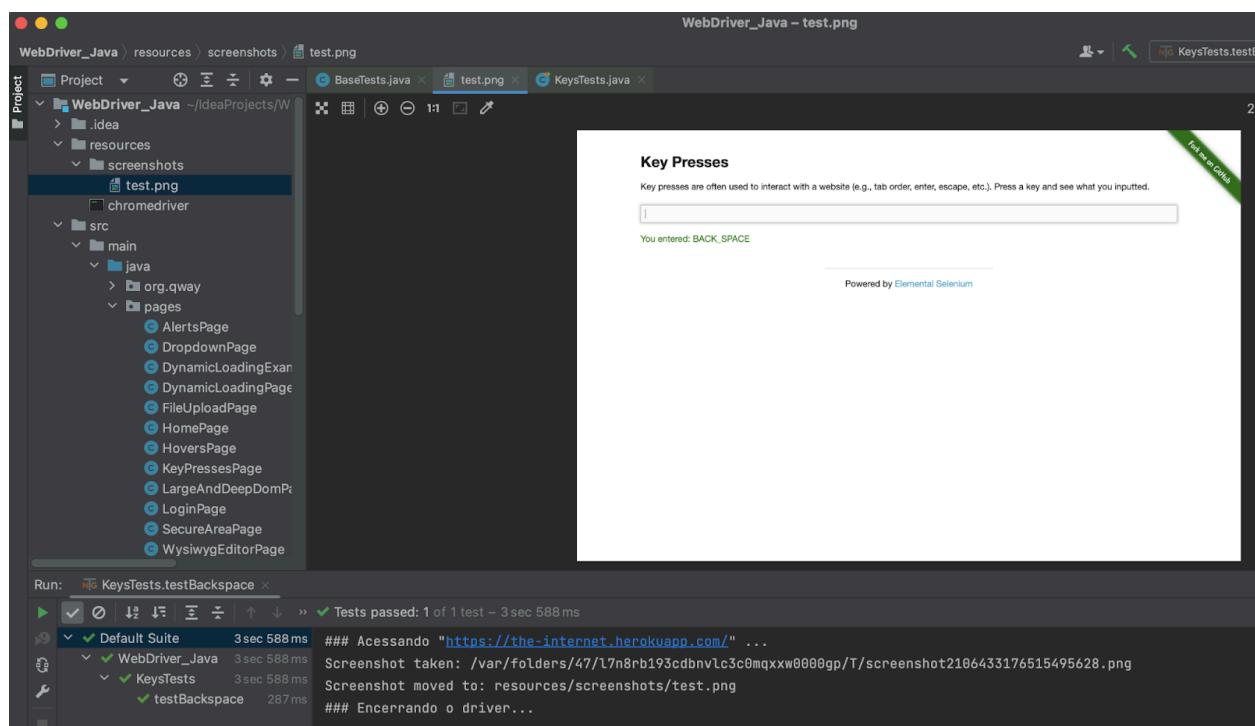


Imagen 100: Screenshot salvo em nosso projeto

E se quiséssemos apenas capturar um *screenshot* nos testes que falham?

Vamos voltar para a classe “**BaseTests.java**” e renomear nosso “**@AfterMethod**“ de **takeScreenshot** para **recordFailure**, para que saibamos que ele só será executado em caso de falha.

Vamos também adicionar um parâmetro do tipo **ITestResult** para este método. E este é o resultado de um teste. Chamaremos esse parâmetro de **result** e então podemos adicionar uma condição dizendo: “se o teste falhou, então queremos fazer tudo isso”.

A lógica da condição será: se **ITestResult.FAILURE** for igual ao status dos resultados que foi passado como parâmetro (o que significa que o teste falhou), então queremos executar a captura do **screenshot**.

```
@AfterMethod
public void recordFailure(ITestResult result) {

    if(result.getStatus() == ITestResult.FAILURE)
    {
        var camera = (TakesScreenshot)driver;
        File screenshot = camera.getScreenshotAs(OutputType.FILE);

        try{
            Files.move(screenshot, new
File("resources/screenshots/test.png"));
        }catch(IOException e){
            e.printStackTrace();
        }
    }
}
```

Para testes bem-sucedidos, ele ainda chamará esse método, mas essa condição falhará e nada acontecerá. Portanto, isso só será executado se o teste não for bem-sucedido.

Vamos excluir este arquivo de imagem de nossa última execução e vamos executar o teste novamente. Se o teste for aprovado, não teremos uma captura de tela. Verificamos novamente e o diretório **screenshots** ainda está vazio.

Além disso, em nosso “**BaseTests.java**”, em vez de gerar um arquivo chamado “**test.png**”, vamos ter o nome do caso de teste que falhou:

```
Files.move(screenshot, new File("resources/screenshots/" + result.getName()
+ ".png"));
```

result.getName() obterá o nome do teste que falhou. E então precisamos adicionar uma extensão também (.png).

Agora vamos fazer esse teste falhar. Esperávamos que o resultado dissesse: "Você digitou: BACK_SPACE". Vamos adicionar um ponto de exclamação aqui e isso deve falhar no teste.

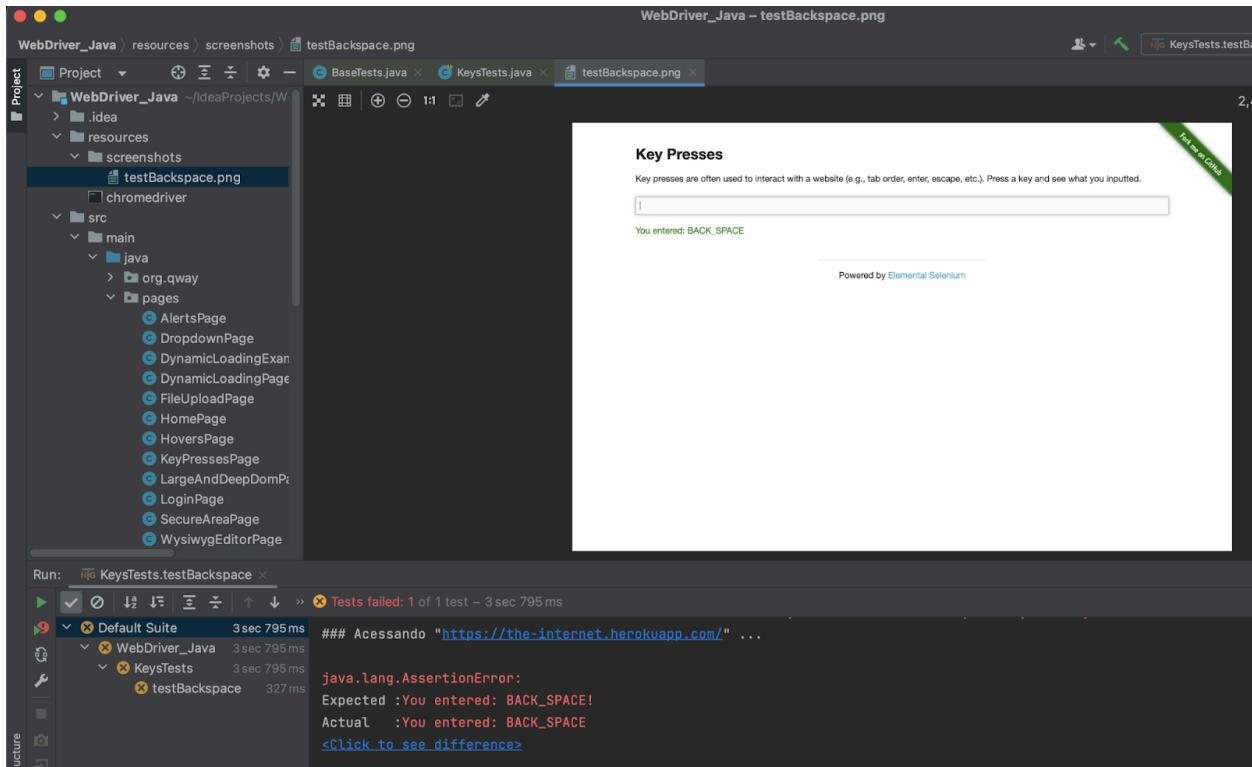


Imagen 101: Screenshot salvo em nosso projeto com o nome do teste que falhou

Capítulo 17 - Event Listeners

O *Selenium* oferece uma interface chamada **WebDriverEventListener**, que fornece métodos que escutam eventos do *Selenium* e permitem que você adicione funcionalidades quando esses eventos ocorrerem.

Por exemplo, e se quisermos escrever relatórios que capturem as ações que foram executadas na UI durante nosso teste? Poderíamos fazer isso implementando a interface **WebDriverEventListener**.

Em nossa classe “**BaseTests.java**”, vamos alterar nosso tipo de *driver* de um *WebDriver* para uma instância específica do *WebDriver* chamada **EventFiringWebDriver**. Em vez de ser uma instância do *ChromeDriver*, vamos torná-la uma instância do *EventFiringWebDriver*, que usa outra instância do *WebDriver*.

Portanto, podemos dizer que é um *ChromeDriver*, mas queremos agrupar isso neste **EventFiringWebDriver** que escutará os eventos.

A maneira como dizemos para ouvir eventos é registrando uma classe que está implementando a interface **WebDriverEventListener**. Então, dizemos **driver.register** e, em seguida, fornecemos a instância da classe *listener*, que ainda não criamos.

```
private EventFiringWebDriver driver;
protected HomePage homePage;

@BeforeClass
public void setUp() {
    System.setProperty("webdriver.chrome.driver", "resources/chromedriver");
    driver = new EventFiringWebDriver(new ChromeDriver());
    driver.register();
}
```

Voltaremos a isso em seguida. Agora vamos criar esta classe — em nosso projeto queremos criar este *listener* no pacote “**utils**” em uma nova classe que chamaremos de “**EventReporter.java**”.

Agora, para que seja um *listener*, ele deve implementar a interface **WebDriverEventListener**. Observe que temos um erro porque precisamos implementar esses métodos herdados.



```

package utils;

import org.openqa.selenium.support.events.WebDriverEventListener;
public class EventReporter implements WebDriverEventListener {
}
  
```

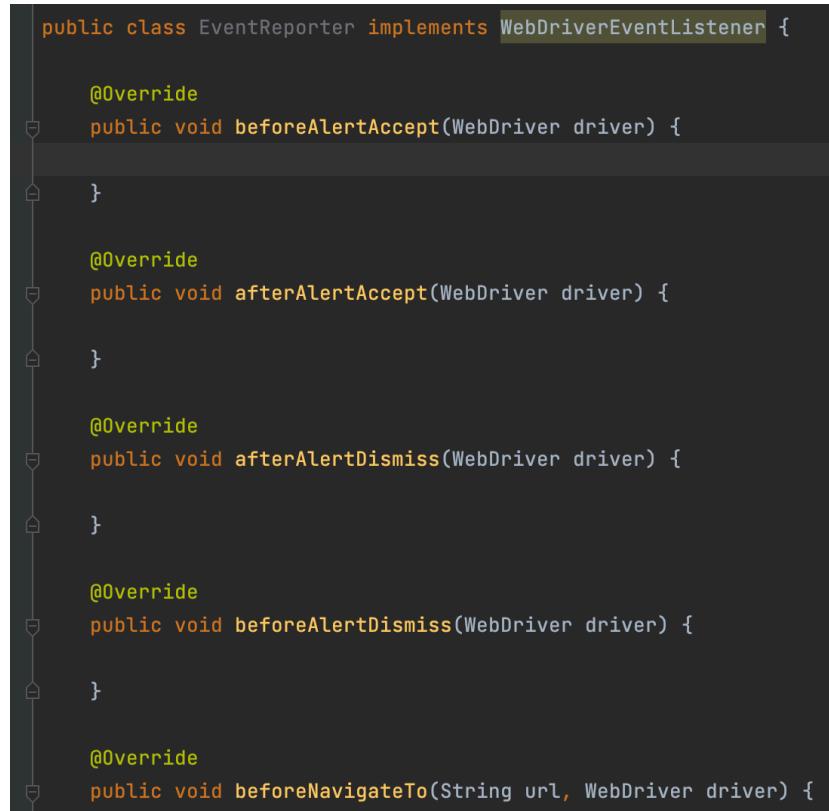
Class 'EventReporter' must either be declared abstract or implement abstract method 'beforeAlertAccept()'.

Implement methods More actions... ↗

org.openqa.selenium.support.events

Imagen 102: Implementação da interface `WebDriverEventListener` na classe `EventReporter.java`

Clicarmos em “**Implement methods**” e adicionaremos todos eles:



```

public class EventReporter implements WebDriverEventListener {

    @Override
    public void beforeAlertAccept(WebDriver driver) {
    }

    @Override
    public void afterAlertAccept(WebDriver driver) {
    }

    @Override
    public void afterAlertDismiss(WebDriver driver) {
    }

    @Override
    public void beforeAlertDismiss(WebDriver driver) {
    }

    @Override
    public void beforeNavigateTo(String url, WebDriver driver) {
    }
}
  
```

Imagen 103: Métodos herdados do `WebDriverEventListener`

Observe que todos os seus corpos estão vazios, nenhum deles faz nada agora. Mas esses métodos permitirão que você adicione o que quiser para qualquer uma das ações — antes e depois de todas as ações.

Vamos escolher apenas um para que possamos vê-lo em ação, como por exemplo antes de clicarmos em algo.

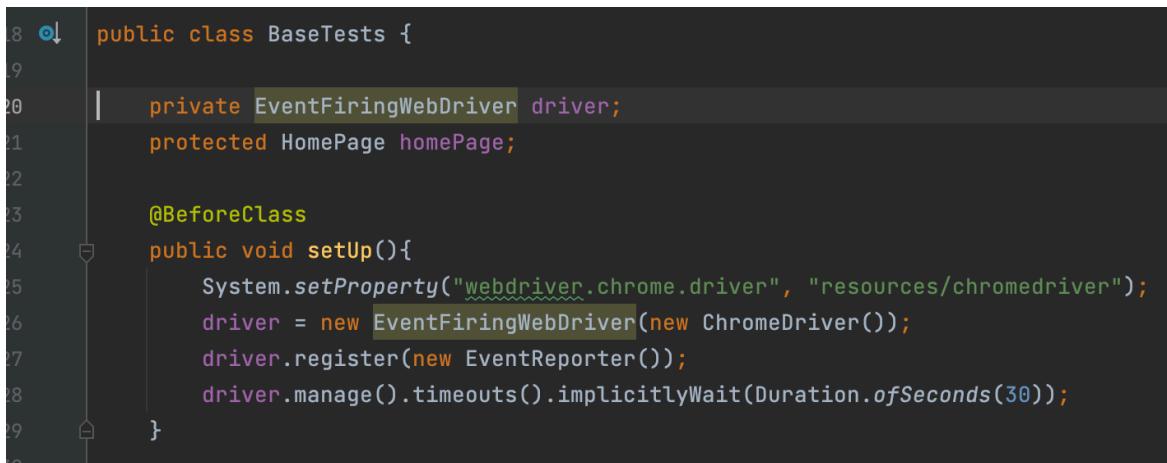
Observe que muitos desses métodos usam um `WebElement` e também um `WebDriver`. Alguns deles levam apenas o `WebDriver`. Para o nosso caso, vamos usá-lo apenas para relatar o que está acontecendo.

Vamos adicionar algum código aqui apenas para imprimir no console o que estamos fazendo. É uma declaração de `println` simples que apenas diz que estamos clicando e fornecerá o texto do elemento no qual estamos clicando:

```
@Override
public void beforeClickOn(WebElement webElement, WebDriver webDriver) {
    System.out.println("### Clicking on " + webElement.getText() + "...");
}
```

Portanto, sempre que estivermos prestes a clicar em qualquer coisa em nosso projeto, esse método será chamado primeiro e imprimirá isso.

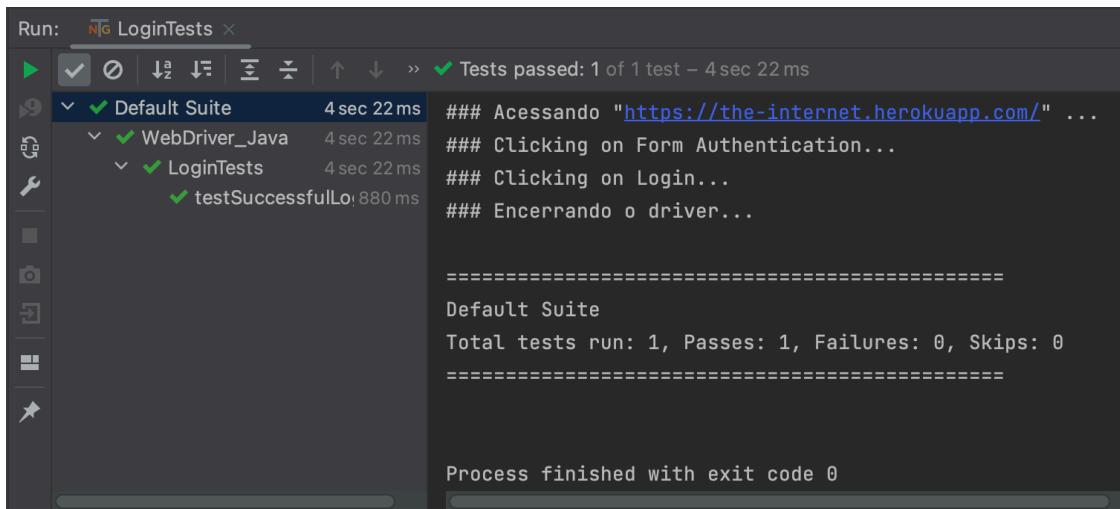
Vamos voltar a classe “**BaseTests.java**” para podermos registrar esta classe.



```
1.8  public class BaseTests {
1.9
20     |     private EventFiringWebDriver driver;
21     |     protected HomePage homepage;
22
23     @BeforeClass
24     public void setUp(){
25         System.setProperty("webdriver.chrome.driver", "resources/chromedriver");
26         driver = new EventFiringWebDriver(new ChromeDriver());
27         driver.register(new EventReporter());
28         driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(30));
29     }
}
```

Imagen 104: Registrando a classe Eventreporter como interface do driver.

Vamos executar qualquer teste, um que faça alguns cliques, talvez o **LoginTests**.



```
Run: NG LoginTests x
[Run] [Stop] [↓a] [↓e] [☰] [✖] [↑] [↓] [▶] [Tests passed: 1 of 1 test – 4 sec 22 ms]
[Default Suite] 4 sec 22 ms
  [WebDriver_Java] 4 sec 22 ms
    [LoginTests] 4 sec 22 ms
      [testSuccessfulLogout] 880 ms
      ====
      Default Suite
      Total tests run: 1, Passes: 1, Failures: 0, Skips: 0
      ====
      Process finished with exit code 0
```

Imagen 105: Resultado do teste com o output do event listener.