GeekBrains

Автоматизация тестирования веб-приложений на Python

Лекция 2
Тестирование Web. Selenium WebDriver

Оглавление

Введение

На второй лекции подробно поговорим о тестировании веб-UI и научимся работать с Selenium Webdriver. Это важный материал — в дальнейших уроках мы будем опираться на него.

На этом уроке мы узнаем:

- Как установить Selenium WebDriver.
- Как взаимодействовать с веб-страницей с использованием Selenium WebDriver и Python.
- Как реализовать кроссбраузерное тестирование.

💡 Для более детального понимания последовательности работы с примерами обязательно посмотрите видеолекцию.

Термины, используемые в лекции

Драйвер — это программная библиотека, которая позволяет другим программам взаимодействовать с устройством.

Кроссбраузерное тестирование — тип тестирования, который проверяет, работает ли приложение так, как ожидается, в нескольких браузерах.

AJAX (Asynchronous Javascript and XML) — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. В результате при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее.

Как тестировать веб-UI

На этой лекции мы будем учиться тестировать веб-интерфейс. Для этого нам нужно взаимодействовать с элементами веб-страницы, открытой в браузере.

В отличие от тестирования API, где мы видим только результат запроса, при тестировании UI нам нужно, чтобы страница открывалась в браузере и мы могли программно взаимодействовать с ней. Для этого понадобится специальный инструмент (не будем называть его библиотекой, почему — поймем позже).

Этот инструмент должен:

- открывать страницы в браузере;
- получать доступ к элементам;
- получать и изменять свойства элементов;
- выполнять обработчики событий (например, click).

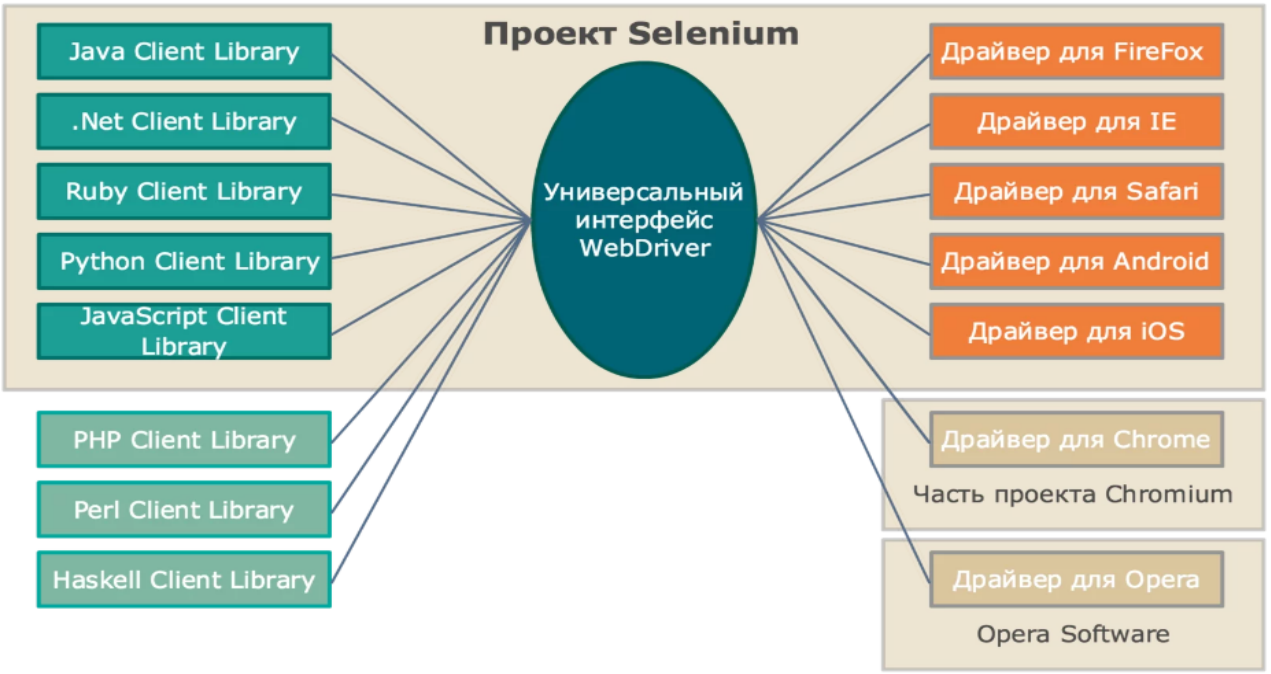
Для этих целей мы будем использовать **Selenium Webdriver**.

Selenium WebDriver

Драйвер — это программная библиотека, которая позволяет другим программам взаимодействовать с некоторым устройством. Например, драйвер принтера, установленный на устройстве, позволяет печатать на принтере.

Пользователи не работают с драйвером напрямую. Они работают с прикладными программами, которые через драйвер взаимодействуют с устройствами. Пользовательского интерфейса у драйвера нет, но он может быть у программы управления драйвером.

Selenium WebDriver — это драйвер браузера, который позволяет программам взаимодействовать с браузером, управлять его поведением, получать от него данные и заставлять его выполнять команды. Selenium WebDriver существует для разных браузеров и разных языков программирования.



Проект Selenium. Источник: Smartiqa

У проекта Selenium нет единого разработчика. Большинство библиотек и версий разрабатывается командой проекта, однако драйверы для Opera и Chrome разрабатываются как часть проекта этих браузеров. Библиотеки для PHP, Perl и Haskell тоже разрабатываются отдельно.

Существуют и другие драйверы. Внутри многих коммерческих инструментов есть драйверы браузеров, но их, как правило, нельзя использовать отдельно, вне этого инструмента.

WebDriver может использоваться не только при тестировании. Ему безразлично, кто и зачем хочет управлять браузером. Вы можете автоматизировать рутинные задачи, сделать ботов или скрипт, который автоматически снимает скриншоты. Задача WebDriver — предоставить доступ к браузеру.

У Selenium WebDriver есть реализации на разных языках программирования: Java, C#, Ruby, Python.

Можно сказать, что Selenium WebDriver — это не инструмент, а спецификация, документ, стандарт, описывающий, какой интерфейс браузеры должны предоставлять наружу, чтобы через этот интерфейс можно было браузером управлять.

В рамках проекта Selenium было разработано несколько референсных реализаций для разных браузеров, но постепенно эта деятельность переходит в ведение производителей браузеров, поскольку только они досконально знают особенности своего продукта.

Selenium, пожалуй, единственный проект по созданию средств автоматизации управления браузерами, в котором участвуют компании-разработчики браузеров. Это одна из ключевых причин его успеха.

Установка Selenium Webdriver

Установить библиотеки Selenium для Python можно с помощью pip3:

```
pip3 install selenium
```

Далее нужно скачать сам драйвер:

- для Google Chrome: WebDriver for Chrome
- для Mozilla Firefox: WebDriver for Firefox

> 💡 Версия драйвера должна соответствовать версии браузера.
>
> Для скачивания выбирайте подходящую в списке.

Устанавливать драйвер не нужно, достаточно скачать. При инициализации в программе мы просто укажем путь к нему, поэтому удобно сохранить его в папке с программой.

Рассмотрим основные команды:

- **driver.get("https://selenium.dev")** — открыть веб-сайт
- **driver.back()** — кнопка браузера «Назад»
- **driver.forward()** — кнопка браузера «Вперед»
- **driver.refresh()** — обновить страницу
- **driver.quit()** — закончить сеанс браузера

Подключение Selenium и запуск браузера

testdata.yaml

Добавим в конфиг адрес сайта, путь к драйверу и время ожидания запуска браузера.

```yaml
address: https://test-stand.gb.ru
driver_path: D:\chromedriver.exe
sleep_time: 5
```

В отдельном модуле реализуем инициализацию драйвера, открытие сайта и небольшое ожидание.

module.py

```python
import time
import yaml
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
service = Service(testdata["driver_path"])
options = webdriver.ChromeOptions()

class Site:
    def __init__(self, address):
        self.driver = webdriver.Chrome(service=service, options=options)
        self.driver.maximize_window()
        self.driver.get(address)
        time.sleep(testdata["sleep_time"])
```

В главном файле пока ничего делать не будем, просто создадим объект класса — отработает конструктор и откроется сайт.

main.py

```python
import yaml
from module import Site

with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
site = Site(testdata["address"])

if __name__ == "__main__":
    pass
```

Поиск элементов

Selenium предоставляет следующие методы для поиска элемента на странице:

- **find_element_by_id** — поиск по id
- **find_element_by_name** — поиск по имени
- **find_element_by_xpath** — поиск по xpath-локатору
- **find_element_by_link_text** — поиск по тексту ссылки
- **find_element_by_partial_link_text** — поиск по части текста ссылки
- **find_element_by_tag_name** — поиск по имени тега
- **find_element_by_class_name** — поиск по имени класса
- **find_element_by_css_selector** — поиск по css-селектору

Чтобы найти все элементы, удовлетворяющие условиям поиска, используйте следующие методы (возвращается список):

- find_elements_by_name
- find_elements_by_xpath
- find_elements_by_link_text
- find_elements_by_partial_link_text
- find_elements_by_tag_name
- find_elements_by_class_name
- find_elements_by_css_selector

Ответ: потому что идентификаторы элементов страницы всегда уникальны.

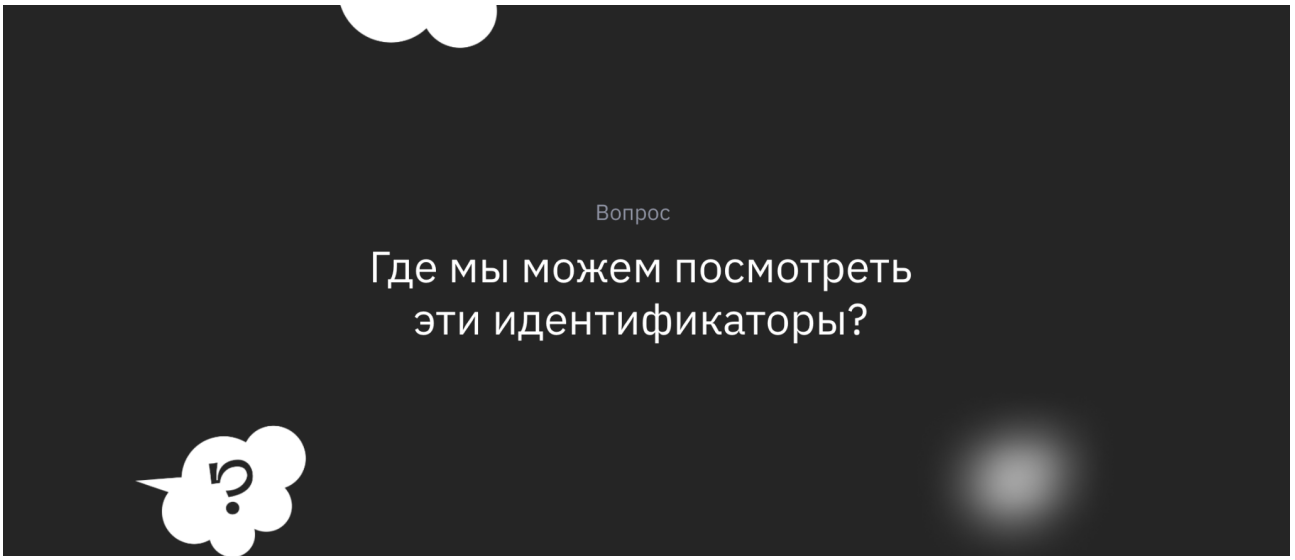
Также можно использовать методы **find_element** и **find_elements**, передавая им тип поиска через аргумент. Возможность передать тип поиска в аргументе очень удобна, поэтому чаще всего мы будем использовать именно эти методы.

Пример использования:

```
1 from selenium.webdriver.common.by import By
2 driver.find_element(By.XPATH, '//button[text()="Some text"]')
3 driver.find_elements(By.XPATH, '//button')
```

Для класса **By** доступны следующие атрибуты:

- ID = "id"
- XPATH = "xpath"
- LINK_TEXT = "link text"
- PARTIAL_LINK_TEXT = "partial link text"
- NAME = "name"
- TAG_NAME = "tag name"
- CLASS_NAME = "class name"
- CSS_SELECTOR = "css selector"



Ответ: в инструментах разработчика в браузере.

Поиск элементов и получение их свойств

module.py

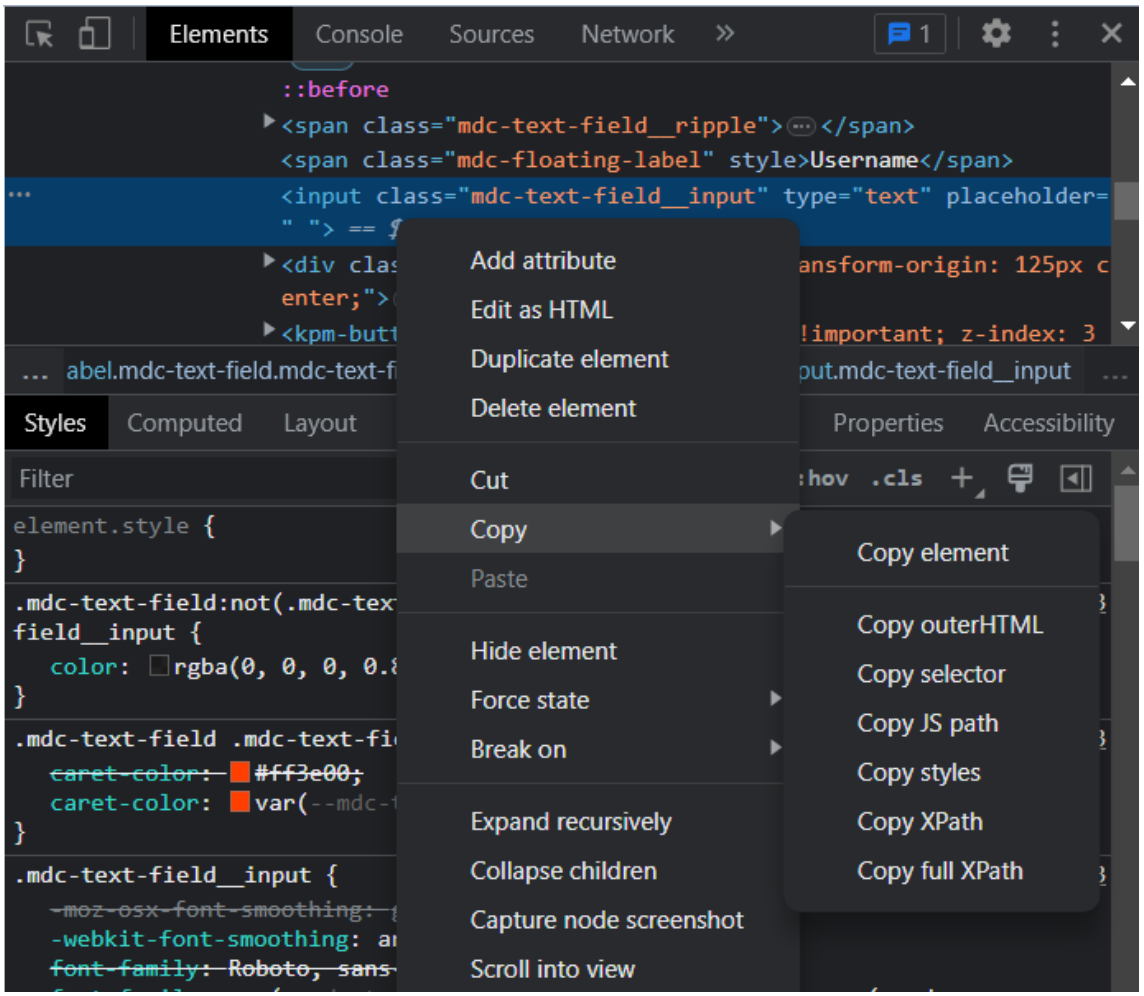
Доработаем модуль, добавив методы поиска элемента и получения его атрибутов. Реализуем поиск только по xpath и по css-селектору. Эти методы поиска самые универсальные.

```python
import time
import yaml
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
service = Service(testdata["driver_path"])
options = webdriver.ChromeOptions()

class Site:
    def __init__(self, address):
        self.driver = webdriver.Chrome(service=service, options=options)
        self.driver.maximize_window()
        self.driver.get(address)
        time.sleep(testdata["sleep_time"])

```

```python
    def find_element(self, mode, path):
        if mode == "css":
            element = self.driver.find_element(By.CSS_SELECTOR, path)
        elif mode == "xpath":
            element = self.driver.find_element(By.XPATH, path)
        else:
            element = None
        return element

    def get_element_property(self, mode, path, property):
        element = self.find_element(mode, path)
        return(element.value_of_css_property(property))

    def close(self):
        self.driver.close()

```

main.py

В основном файле проверим работу этих методов, добавив css и xpath-селекторы поля ввода со страницы логина. Скопируем их из браузера в режиме разработчика:




```python
import yaml
from module import Site

with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
site = Site(testdata["address"])

if __name__ == "__main__":
    css_selector = "span.mdc-text-field__ripple"
    print(site.get_element_property("css", css_selector, "height"))
    xpath = """//*[@id="login"]/div[1]/label/span[1]"""
    print(site.get_element_property("xpath", xpath, "color"))
```

Ожидания

Сейчас большинство веб-приложений используют AJAX-технологии. Когда страница загружена в браузере, элементы на ней могут подгружаться с разными временными интервалами. Это затрудняет поиск элементов.

Проблему решают ожидания. Они задают временной интервал, чтобы элемент успел появиться на странице. После этого можно будет найти его или выполнить другую операцию.

В Selenium WebDriver есть два типа ожиданий — неявное и явное.

- Явное заставляет WebDriver ожидать возникновение определенного условия до произведения действий.
- Неявное заставляет WebDriver опрашивать модель определенное время и пытаться найти элемент.

Явное ожидание — это код, которым вы определяете, какое условие должно произойти, чтобы дальнейший код исполнился. Не очень хороший пример такого кода — команда time.sleep(), которая ожидает определенное точное время. Есть и более удобные методы: например, можно использовать WebDriverWait в комбинации с ExpectedCondition следующим способом:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Firefox()
driver.get("http://somedomain/url_that_delays_loading")
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "myDynamicElement"))
    )
finally:
    driver.quit()
```

Этот код будет ждать 10 секунд до того, как отдаст исключение TimeoutException или, если найдет элемент за эти 10 секунд, вернет его. WebDriverWait по умолчанию вызывает ExpectedCondition каждые 500 миллисекунд до тех пор, пока не получит успешный return. Успешный return для ExpectedCondition имеет тип Boolean и возвращает значение true либо not null для всех других ExpectedCondition типов.

Неявное ожидание заставляет драйвер опрашивать страницу определенное время, пока он пытается найти элементы, недоступные в тот момент. Значение по умолчанию равно 0. После установки неявное ожидание устанавливается для всего экземпляра объекта WebDrive.

Пример реализации неявного ожидания:

```python
from selenium import webdriver
driver = webdriver.Firefox()
driver.implicitly_wait(10) # seconds
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element_by_id("myDynamicElement")
```

Взаимодействие с элементами

Для проведения полноценного тестирования нам важно уметь получать свойства элементов: ширину (width), высоту (height), цвет (color) и подобные. Для этого используется метод **element.get_attribute()**.

Пример использования:

```python
# Navigate to the url
driver.get("https://www.selenium.dev/selenium/web/inputs.html")
# Identify the email text box
email_txt = driver.find_element(By.NAME, "email_input")
# Fetch the value property associated with the textbox
value_info = email_txt.get_attribute("value")
```

Часто бывает нужно кликнуть по элементу. Для этого используется метод **element.click()**.

Чтобы ввести текст в текстовое поле, нужно выполнить **element.send_keys("some text")**.

В этом же методе можно имитировать нажатие специальных клавиш клавиатуры. Для этого используется класс Keys: **element.send_keys(" and some", Keys.ARROW_DOWN).**

Метод send_keys можно вызвать для любого элемента, который позволяет проверить сочетания клавиш. Но есть побочный эффект: ввод в текстовое поле не очищает его автоматически. То, что вы набираете на клавиатуре, будет дописываться к уже записанному в поле.

Очистить содержимое текстового поля или текстовой области можно с помощью метода **element.clear()**.

Для работы с элементами форм SELECT есть специальный класс — Select. Он предоставляет удобные способы взаимодействия, например, выбор по некоторым условиям:

```
from selenium.webdriver.support.ui import Select
select = Select(driver.find_element_by_name('name'))
select.select_by_index(index)
select.select_by_visible_text("text")
select.select_by_value(value)
```

Также WebDriver предоставляет возможность снять выделение со всех элементов выпадающего списка с помощью метода **deselect_all()**:

```
select = Select(driver.find_element_by_id('id'))
select.deselect_all()
```

Этот код снимет выделение со всех тегов OPTION первого тега SELECT на странице.

Если для теста вам нужен список всех выделенных опций, то свойство **all_selected_options** класса Select его вернет:

```
select = Select(driver.find_element_by_xpath("xpath"))
all_selected_options = select.all_selected_options
```

А для получения всех доступных опций можно использовать свойство **options**:

```
options = select.options
```

Чтобы сохранить изменения в форме, можно найти кнопку Submit и кликнуть по ней, а можно использовать метод **submit()**, доступный для каждого элемента.

```
element.submit()
```

Selenium Webdriver поддерживает два варианта перетаскивания» элементов: перемещение элемента на определенную величину либо перетаскивание его на другой элемент. Вот пример перетаскивания одного элемента на другой:

```
element = driver.find_element_by_name("source")
target = driver.find_element_by_name("target")
from selenium.webdriver import ActionChains
action_chains = ActionChains(driver)
action_chains.drag_and_drop(element, target)
```

Перетаскивание по координатам выполняется аналогично, но с использованием метода **drag_and_drop_by_offset()**:

```
drag_and_drop_by_offset(source1, 100, 100)
```

Selenium WebDriver поддерживает управление всплывающими диалоговыми окнами. Например, после того как вы инициируете запуск alert и откроется соответствующее окно, управлять им можно так:

```
alert = driver.switch_to_alert()
```

Код вернет объект текущего открытого окна. Мы можем принять или отклонить вопрос, а также прочитать его содержимое. Интерфейс взаимодействия со всплывающими окнами реализован для предупреждений (alerts), запросов к подтверждению (confirms) и приглашений к вводу (prompts).

В процессе автоматизации нам может потребоваться сделать скриншот всей страницы или определенного веб-элемента. Это можно сделать с помощью метода **save_screenshot()**:

```
capture_path = 'C:/capture/your_desired_filename.png'
driver.save_screenshot(capture_path)
```

Итак, мы разобрали основные методы поиска, ожидания и взаимодействия с элементами. Этого достаточно, чтобы начать писать тесты веб-интерфейса. С помощью изученных методов мы можем писать тесты верстки и функциональные тесты.

Тесты верстки пишутся стандартно и однообразно: мы можем перебрать все элементы на странице, получить их свойства и проверить, что они соответствуют заранее заданным (их можем получить от дизайнеров и занести в conftest.py).

Больший интерес представляют функциональные тесты: чтобы написать их, мы изучаем функциональные требования продукта, на их основе сочиняем тестовый сценарий и реализуем его проверку. Сценарий может включать, например, проверку функционала авторизации. В этом случае мы должны проверить реакцию системы на успешную и неуспешную авторизацию. Для реализации таких тестов нам нужно активно взаимодействовать со страницей: кликать, вводить текст, отправлять формы и анализировать реакцию на эти события.

Кроссбраузерное тестирование

Кроссбраузерное тестирование — это тип тестирования, который проверяет, работает ли приложение так, как ожидается, в нескольких браузерах.

Создание и выполнение индивидуального сценария тестирования для уникальных сценариев занимает много времени. Поэтому наши тестовые сценарии создаются с тестовыми данными для использования их комбинаций. Один и тот же сценарий тестирования может выполняться на Chrome и Windows для первой итерации, затем на Firefox и Mac для второй итерации, а затем на других сценариях для последующих итераций.

Это экономит время, поскольку мы создаем только один тестовый сценарий, а не несколько.

Но для этого нам нужно использовать Selenium WebDriver не на локальной машине, а обращаться к нему с использованием специального менеджера Selenium Webdriver Manager.

Реализация кроссплатформенного теста ошибки при авторизации

Добавим параметр, отвечающий за то, какой браузер запустится.

testdata.yaml

```
address: https://test-stand.gb.ru
sleep_time: 1
browser: firefox
```

Добавим в модуль импорт библиотеки webdriver_manager и работу с разными браузерами с ее помощью. Скачивать драйвер самим теперь не нужно.

module.py

```python
import time
import yaml
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.chrome.service import Service
with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
    browser = testdata["browser"]
from webdriver_manager.firefox import GeckoDriverManager
from webdriver_manager.chrome import ChromeDriverManager

```

```python
class Site:
    def __init__(self, address):
        if browser == "firefox":
            service = Service(executable_path=GeckoDriverManager().install())
            options = webdriver.FirefoxOptions()
            self.driver = webdriver.Firefox(service=service, options=options)
        elif browser == "chrome":
            service = Service(executable_path=ChromeDriverManager().install())
            options = webdriver.ChromeOptions()
            self.driver = webdriver.Chrome(service=service, options=options)
        self.driver.implicitly_wait(3)
        self.driver.maximize_window()
        self.driver.get(address)
        time.sleep(testdata["sleep_time"])
```

Оформим тест привычным образом в формате pytest. Реализуем ввод в поля неправильных имени пользователя и пароля, клик по кнопке логина и проверку того, что появился элемент, содержащий код ошибки 401.

test_1.py

```python
import yaml
from module import Site

with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
site = Site(testdata["address"])

def test_step1():
    x_selector1 = """//*[@id="login"]/div[1]/label/input"""
    input1 = site.find_element("xpath", x_selector1)
    input1.send_keys("test")
    x_selector2 = """//*[@id="login"]/div[2]/label/input"""
    input2 = site.find_element("xpath", x_selector2)
    input2.send_keys("test")
    btn_selector = "button"
    btn = site.find_element("css", btn_selector)
    btn.click()
    x_selector3 = """//*[@id="app"]/main/div/div/div[2]/h2"""
    err_label = site.find_element("xpath", x_selector3)
    assert err_label.text == "401"
```

Итоги

На этом уроке мы узнали:

- Как установить Selenium WebDriver.
- Как взаимодействовать с веб-страницей с использованием Selenium WebDriver и Python.
- Как реализовать кроссбраузерное тестирование.

Что можно почитать еще?

1. Selenium для Python
2. Кроссбраузерное тестирование в Selenium

Используемая литература

1. Что такое Selenium WebDriver?
2. Selenium with Python
3. Selenium Webdriver для начинающих