# **ВВЕДЕНИЕ**

На данный момент времени сложно представить себе современного человека, который в течение дня не посещает хотя бы единожды тот или иной веб-сайт. Интернет-технологии настолько тесно взаимосвязаны с ежедневным бытом каждого из нас (общение с друзьями и знакомыми в мессенджерах, поиск необходимой информации, заказ продуктов питания из интернет-гипермаркетов, покупка гаджетов и техники) и нашими привычками (чтение интересующих нас новостей, просмотр видеороликов, прослушивание музыки), что никто не усомнится в важности качества веб-сайтов и других интернет-сервисов.

В условиях жесткой конкуренции и постоянной борьбы за аудиторию на рынке веб-технологий, качество **–** критический фактор, влияющий на посещаемость веб-сайта и итоговую прибыль, которую получит бизнес благодаря популярности своего сервиса.

Огромную роль в качестве программного обеспечения занимает тестирование. Оно помогает обеспечить корректную и стабильную работу веб-сайта в различных браузерах и на различных конфигурациях компьютерного «железа», сделать его удобным для пользователя и оптимизировать производительность.

Таким образом, вопрос создания удобного и эффективного средства для тестирования веб-сайтов и приложений является безусловно важным.

Задачей дипломного проектирования является разработка библиотеки, которая позволит автоматизировать тестирование UI (User Interface) и API (Application Programming Interface) веб-сайтов.

Для решения поставленной задачи необходим следующее:

1. Исследовать современные подходы и методы автоматизированного тестирования ПО;
2. Выявить особенности веб-сервисов, влияющие на процесс их тестирования;
3. Рассмотреть и проанализировать современные системы для тестирования UI и API веб-сервисов, провести их сравнительную характеристику, выявить достоинства и недостатки;
4. Освоить технологию создания фреймворков для тестирования веб-сервисов;
5. Выявить необходимость разработки универсальной системы для автоматизации тестирования веб-сайтов;
6. Разработать фреймворк для автоматизации процесса тестирования UI и API веб-сервисов;
7. Протестировать веб-сервис с использованием разработанного фреймворка.

# **АНАЛИЗ ИСХОДНЫХ ДАННЫХ И ПОСТАНОВКА ЗАДАЧ НА ДИПЛОМНОЕ ПРОЕКТИРОВАНИЕ**

## **1.1 Анализ исходных данных к дипломному проекту**

Общая цель веб-тестирования – найти проблемы, также известные как ошибки или баги, проблемы или дефекты, которые могут отрицательно повлиять на работу веб-сайта или приложения. Веб-тестирование также можно использовать, чтобы добавить новую функциональность, обнаружить определенные области или аспекты веб-сайта, которые можно улучшить и, таким образом, получить лучшие результаты, будь то больше запросов, больше продаж, больше повторных посетителей или более счастливые пользователи.

Процедура тестирования чрезвычайно тщательна. Веб-тестировщики проверяют всю веб-систему от начала до конца, прежде чем выпускать веб-сайт на рынок.

По способам тестирование можно подразделить на:

1. Автоматизированное тестирование

Для автоматической проверки веб-сайта тестировщику понадобятся специальные программы. Программа настраивается под нахождение и проверку на соответствия определенных параметров сайта. Такой тип тестирования в наибольшей степени подходит для объемных проектов, нагруженных сложной функциональностью.

1. Ручное тестирование

Ручной тип тестирования программного обеспечения выполняется тестировщиком вручную без использования каких-либо автоматизированных инструментов. Ручное тестирование помогает находить критические ошибки в программном приложении, проверяет реальный клиентский опыт, навигацию и многое другое.

Автоматизация помогает выстроить баланс и проверять вручную то, что требует человеческого внимания (как правило, до 25% кейсов); автоматизировать остальные кейсы.

При этом автоматизация в долгосрочной перспективе снижает как расходы на тестирование, так и риски, связанные с человеческим фактором.

Кроме того, при необходимости можно ускорить релизы. Например, если в спринте разработки нужно проверить около 400 кейсов, то их ручная проверка займет до двух недель, а автоматизированные тесты можно провести ночью и проанализировать за 4 часа.

Бизнес, благодаря автоматизации, получает возможность в любое время убедиться в том, что ключевые функции системы работают правильно и проверить, нет ли ошибок (а если они есть, то в чем заключаются).

## **1.2 Обзор существующих программных средств по теме дипломного проекта**

При выборе фреймворка стоит обращать внимание на его популярность на GitHub, поддержку сообщества, количество открытых issues и совместимость с технологиями, которые используются у вас в системе, а также на особенности каждого инструмента, применяемого для автоматизации.

Например, Protractor хорошо подходит для проектов, написанных на Angular, потому что в нем есть встроенные функции для специфических загрузок Angular-элементов.

Ниже подробнее рассмотрим самые популярные JS-фреймворки, для автоматизации тестирования.

**Protractor**

Достоинства:

* Protractor – это единственный фреймворк, который из коробки поддерживает кастомные определения AngularJS-элементов. Если у вас Angular, используйте Protractor;
* удобная поддержка TypeScript и разных фреймворков для unit testing (Jasmine, Mocha, Cucumber и прочие).

Недостатки:

* нет поддержки мобильного тестирования;
* написан как обертка над WebDriverJS. Если будут проблемы с WebDriverJS, автоматически они будут и в ProtractorJS.

**Nightwatch.js**

Достоинства:

* похож на WebdriverIO и тоже является кастомной имплементацией W3C WebDriver API;
* легко добавить новую функцию;
* не нужно выбирать между Jasmine и Mocha.

Недостатки:

* нет поддержки Mocha и мобильного тестирования;
* меньше поддержки, чем у WebdriverIO и Protractor.

**Cypress**

Достоинства:

* архитектура работы с браузером была написана полностью c нуля. Cypress не запускает удаленные команды через сеть, а работает напрямую с программой;
* быстрая и удобная настройка. Необходимо запустить только одну команду для установки всех пакетов, и нет необходимости ставить все отдельно;
* вся документация в одном месте, не нужно отдельно искать, как правильно делать проверки или как использовать Chai;
* необычный интерфейс запуска с описанием, какие команды выполняются, что позволяет быстрее понимать, что происходит на странице в определенный момент.

Недостатки:

* Cypress не использует Selenium для end-to-end-тестирования. Что порождает много проблем в работе. Работа с браузером не всегда стабильна, и на сегодняшний день на GitHub висят открытыми более чем 900 проблем;
* нет поддержки мобильного тестирования, и, судя по комментариям создателей нативной поддержки, никогда и не будет;
* поддержка ограниченного количества браузеров: Canary, Chrome, Chromium, Electron;
* платная параллелизация тестов. Если необходимо запускать тесты в несколько потоков, то придется платить.

**WebdriverIO**

Достоинства:

* WebdriverIO – это кастомная имплементация W3C WebDriver API. Не нужно привязываться к имплементации WebDriverJS;
* поддержка синхронного кода. Можно забыть об асинхронности JavaScript;
* удобная базовая настройка с помощью встроенного интерфейса для командной строки wdio;
* поддерживает почти все тестовые фреймворки BDD и TDD;
* поддерживает удобную библиотеку ‘webdrivercss’ для сравнения CSS-стилей элементов на странице.

Недостатки:

* большое различие между последними версиями (WebdriverIO 4 и 5);
* хуже кастомизирован для автоматизации AngularJS, чем Protractor;
* синхронность WebdriverIO обеспечивает технология Fibers, иногда ее еще называют coroutines. У этой технологии много плюсов и преимуществ, но существуют подводные камни, например запуск в асинхронном коде сделает весь код асинхронным.

**Selenium**

Selenium – настоящий старожил среди всех инструментов для функционального тестирования. Его история началась в 2004 году, и сейчас он по праву считается самым популярным инструментом своей отрасли. Несмотря на повсеместное применение Selenium такими гигантами как Google, у него все еще остается ряд недостатков. Пожалуй, главным из них является необходимость владения продвинутыми навыками программирования и написания скриптов для полноценной работы с продуктом, что в свою очередь часто затрудняет освоение инструмента начинающими тестировщиками. Зато Selenium распространяется абсолютно бесплатно, поддерживает огромное количество языков для написания скриптов, а также является кроссплатформенным инструментом, и за это его невозможно не любить.

Достоинства:

* бесплатный;
* поддержка всех основных ЯП для сценариев;
* кроссплатформенность.

Недостатки:

* тестирование ограничивается веб приложениями;
* нужны продвинутые навыки программирования;
* ограниченность функционала в сравнении с платными аналогами.

## **1.3  Обоснование и описание выбора языка программирования, средств разработки, используемых технологий и сторонних библиотек**

Приступая к автоматизации, каждая команда принимает решение на основе своих знаний или руководствуясь потребностями заказчика. Сейчас появляется все больше возможностей выбирать язык автоматизации исходя из стека технологий конкретного проекта.

Разница между языками все больше стирается и становится незаметней. Поэтому, просматривая современные тренды в автоматизации, можно заметить, что все больше проектов используют **JavaScript** для разных уровней и видов тестирования. Как пример можно привести статистику Github за 2018 год по популярности языков программирования (Рисунок 1.1).

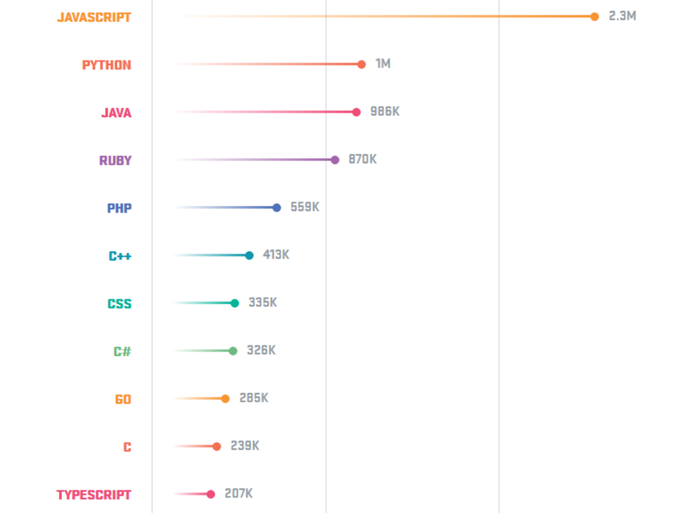


Рисунок 1.1 – Статистику GitHub по популярности языков программирования

Первое, на что нужно обратить внимание при выборе языка для автоматизации, это стек технологий, которые используются в проекте. Так на текущий момент времени практически все актуальные веб-сайты используют JavaScript для так называемого «оживления» веб-страниц, как минимум это весьма существенный плюс в пользу того, чтобы использовать этот же язык программирования при написании автоматизированных тестов.

Помимо вышеозначенного момента, можно выделить следующие преимущества использования JavaScript-фреймворков для автоматизации тестирования:

* скорость написания тестов значительно выше, чем на Java или C#;
* ниже порог входа для старта проекта;
* большое количество готовых решений очень разных проблем, которые возникают.

По вышеозначенным причинам для реализации библиотеки данного дипломного проекта был выбран язык программирования JavaScript.

В качестве среды разработки был выбран **Visual Studio Code** – это сервис, который позиционируется как «легкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений.

VS Code позволяет разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ.

В редакторе присутствуют встроенный отладчик, инструменты для работы с Git и средства рефакторинга, навигации по коду, автодополнения типовых конструкций и контекстной подсказки.

Продукт поддерживает разработку для платформ ASP.NET и Node.js, и считается легковесным решение, которое позволяет обойтись без полной интегрированной среды разработки.

Большим плюсом редактора является поддержка большого количества языков, таких как C++, C#, Python, PHP, JavaScript и других.

Можно выделить следующие преимущества данного редактора кода:

* множество настроек (как всей программы, так и интерфейса);
* расширяемая библиотека дополнений и готовых решений;
* мультифункциональность (редактор поддерживает почти все языки, используемые для создания приложений);
* простота и гибкость.

В качестве инструмента для общения с серверной частью веб-сайтов была выбрана библиотека **Axios** – JavaScript-библиотека, основанная на обещаниях, для выполнения HTTP-запросов.

Некоторые преимущества Axios:

* трансформаторы: позволяют выполнять преобразования данных перед выполнением запроса или после получения ответа;
* перехватчики: позволяют полностью изменить запрос или ответ (в том числе и заголовки). также выполнять асинхронные операции до того, как будет сделан запрос или до того, как Promise выполнит;
* встроенная защита XSRF;
* поддержка всеми актуальными браузерами.

Выбор так называемой assertion библиотеки пал на весьма популярную **Chai**. Можно даже сказать, что большинство разработчиков и тестировщиков используют именно Chai для написания тестовых утверждений.

В Chai сужествует три типа assertions:

* should (должен);
* expect (ожидать);
* assert (утверждать).

Данная библиотека достаточно гибкая и предоставляет большой выбор тестовых утверждений, который способен покрыть фактически любые запросы при написании автоматизированных тестов.

**Mocha** – это JS-фреймворк, облегчающий тестирование асинхронного кода в Node.JS модуле или браузерном приложении. Тесты в Mocha имеют улучшенное качество трассировки исключений и могут прогоняться сериями.

Используя Mocha, можно получить необходимый минимум и ничего лишнего. Это один из популярнейших фреймворков на Node.JS с достаточно активным сообществом и большим количеством документации. Но при всех достоинствах у него есть и недостаток – это отсутствие assert-ов. Однако данный недостаток можно легко назвать и преимуществом, поскольку мы можем подключить фактически любую из имеющихся библиотек утверждений (выбор был сделан в пользу Chai).

## **Постановка задач по разработке программного средства**

Разработка любого программного средства начинается с постановки задач, которые должны быть решены в ходе работы, для того чтобы получить конечный продукт. В ходе дипломного проектирования необходимо разработать библиотеку «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов. Для этого необходимо решить следующие задачи:

1. Проанализировать исходные данные.
2. Произвести анализ рынка библиотек для автоматизированного тестирования веб-сайтов.
3. Выбрать язык программирования, средства разработки и сторонние библиотеки. Привести описание выбранного языка программирования, средств разработки и сторонних библиотек. Обосновать выбор языка программирования, средств разработки и сторонних библиотек.
4. Спроектировать структуру библиотеки.
5. Формализировать предметную область программного средства.
6. Рассмотреть интеграцию разработанной библиотеки с тестируемым веб-сервисом.
7. Описать и реализовать используемые в библиотеке алгоритмы.
8. Протестировать программное средство.
9. Произвести оценку временных показателей программного средства.
10. Произвести оценку ресурсных показателей программного средства.
11. Ввести в эксплуатацию и обосновать минимальные технические требования к оборудованию.
12. Разработать руководство по эксплуатации программным средством.
13. Произвести технико-экономическое обоснование разработки библиотеки «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов.

Таким образом, для получения готового продукта необходимо решить 13 задач.

# **2 ПРОЕКТИРОВАНИЕ, РАЗРАБОТКА И ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА**

## **2.1. Проектирование архитектуры библиотеки**

Для реализации фреймворка будет использоваться объектно-ориентированная архитектура, так как она обладает следующими полезными свойствами:

1. Можно изменять реализацию того или иного объекта или добавлять новые объекты без воздействия на остальную часть системы;
2. Имеющиеся объекты могут быть повторно использованы при создании новых объектов;
3. Фреймворк состоит из независимых компонент, что облегчает процесс разработки и модификации системы.

Для реализации библиотеки «kbv-testdriver» будет использоваться Selenium Webdriver, который позволяет взаимодействовать с браузером и элементами веб-страниц – это популярный инструмент для управления реальным браузером, который можно использовать как для автоматизации тестирования веб приложений, так и для выполнения других рутинных задач, связанных с работой в вебе. Разрабатываемая библиотека будет являться наследником Selenium Webdriver и вбирать в себя все основные и наиболее часто используемые средства для написания и проведения автоматизированного тестирования пользовательского интерфейса приложений. В дополнение к этому в «kbv-testdriver» будет реализована работа с Representational State Transfer Application Programming Interface (REST API). По умолчанию «из коробки», Selenium Webdriver лишен такой возможности. Также будет добавлена возможность загрузки изображений на веб-сервис и последующий анализ их изменений в процессе обработки веб-сервисом.

Создание набора тестов с использованием WebDriver потребует от нас понимания и эффективного использования ряда различных компонентов. Как и все в программном обеспечении, разные люди используют разные термины для обозначения одной и той же идеи. Ниже приведем описание терминов, которые будут использоваться в дальнейшем.

API: интерфейс прикладного программирования. Это набор «команд», которые вы используете для управления WebDriver.

Библиотека: модуль кода, который содержит API и код, необходимый для их реализации. Библиотеки зависят от привязки к каждому языку, например файлы .jar для Java, файлы .dll для .NET и т. Д.

Драйвер: отвечает за управление самим браузером. Большинство драйверов создаются самими производителями браузеров. Драйверы, как правило, представляют собой исполняемые модули, которые запускаются в системе с самим браузером, а не в системе, в которой выполняется набор тестов. (Хотя это может быть одна и та же система.) Некоторые люди называют драйверы прокси-серверами.

Framework: дополнительная библиотека, используемая для поддержки пакетов WebDriver. Эти фреймворки могут быть тестовыми фреймворками, такими как JUnit или NUnit. Они также могут быть фреймворками, поддерживающими функции естественного языка, такие как Cucumber или Robotium. Фреймворки также могут быть написаны и использованы для таких вещей, как манипулирование или настройка тестируемой системы, создание данных, тестовые оракулы и т.д.

Как минимум, WebDriver общается с браузером через драйвер. Связь является двухсторонней: WebDriver передает команды браузеру через драйвер и получает информацию обратно по тому же маршруту. Данный вариант представлен на рисунке 2.1.



Рисунок 2.1 – Простейший вариант взаимодействия Selenium Webdriver с браузером (прямое общение)

Драйвер специфичен для браузера, например ChromeDriver для Google Chrome / Chromium, GeckoDriver для Mozilla Firefox и т. Д. Драйвер работает в той же системе, что и браузер. Это может быть или не быть той же системой, в которой выполняются сами тесты.

Этот простой пример выше - прямое общение. Связь с браузером также может быть удаленной через Selenium Server или RemoteWebDriver. RemoteWebDriver работает в той же системе, что и драйвер и браузер. Данный вариант представлен на рисунке 2.2.

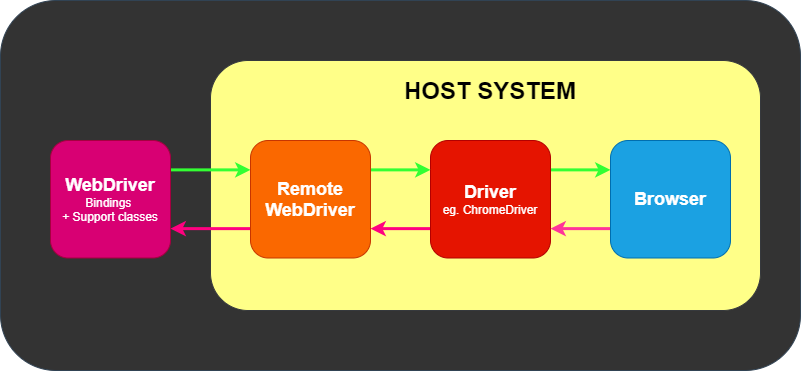


Рисунок 2.2 – Вариант взаимодействия Selenium Webdriver с браузером с использованием RemoteWebDriver

Удаленное общение также может происходить с использованием Selenium Server или Selenium Grid, которые, в свою очередь, взаимодействуют с драйвером в хост-системе. Данный вариант представлен на рисунке 2.3.

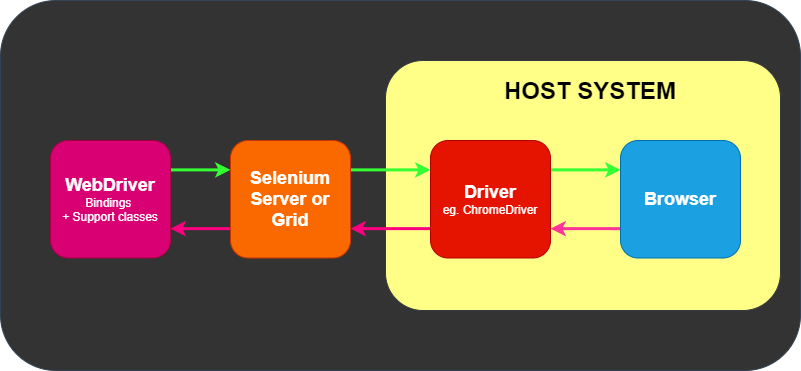


Рисунок 2.3 – Вариант взаимодействия Selenium Webdriver с браузером с использованием Selenium Server или Selenium Grid

Однако при рассмотрении каждого из представленных вариантов взаимодействия Selenium Webdriver с браузером важно понимать то, что у WebDriver одна и только одна задача: общаться с браузером любым из вышеперечисленных методов. WebDriver ничего не знает о тестировании: он не знает, как сравнивать, утверждать, прошел или не прошел тот или иной тест, и, конечно же, ничего не знает об отчетах или грамматике Given / When / Then.

Здесь на помощь приходят различные фреймворки. Как минимум нам понадобится тестовая среда, которая соответствует языковым привязкам, например NUnit для .NET, JUnit для Java, RSpec для Ruby и как уже было описано ранее – Mocha для JavaScript.

Платформа тестирования отвечает за запуск и выполнение вашего WebDriver и связанных шагов в ваших тестах. Таким образом, мы можем это представить следующим образом (Рисунок 2.4).

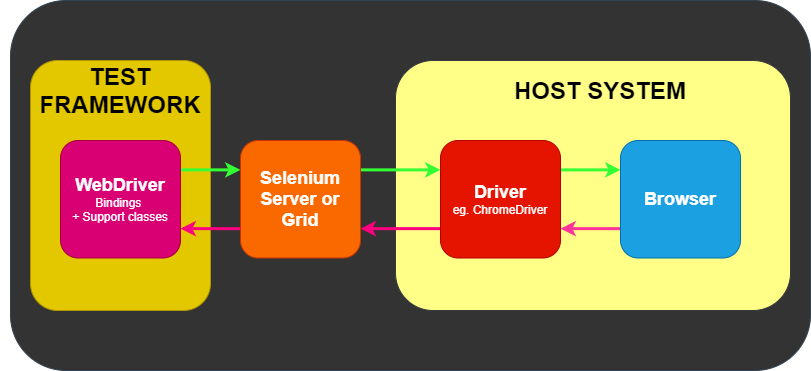


Рисунок 2.4 – Вариант взаимодействия Selenium Webdriver с браузером при условии использования тестового фреймворка

Selenium Webdriver состоит из набора определенных методов, каждый из которых обладает целевым предназначением. Только слаженная работа этих методов предоставляет разработчикам возможность создавать очень широкий набор функциональных тестов пользовательского интерфейса.

Далее приведем список основных методов, предусмотренных в Selenium Webdriver, необходимых для коммуникации с браузером, и краткое описание их предназначения. Разумеется, синтаксис для каждого из поддерживаемых языков программирования будет свой, мы же сосредоточимся на выбранном нами для реализации библиотеки JavaScript.

Первое, что вам нужно сделать после запуска браузера, – это открыть веб-сайт. Это можно сделать одной строкой:

await driver.get('https://selenium.dev')

Мы можем прочитать текущий URL-адрес из адресной строки браузера, используя:

await driver.getCurrentUrl();

Для нажатия кнопки возврата (Back) в браузере:

await driver.navigate().back();

Для нажатия кнопки перехода (Forward) в браузере:

await driver.navigate().forward();

Для обновления текущей страницы:

await driver.navigate().refresh();

Мы можем также прочитать заголовок текущей страницы из браузера:

await driver.getTitle();

Когда мы закончили наш сеанс браузера, мы должны вызвать quit:

await driver.quit();

Метод quit() выполнит следующие действия:

* Закроет все окна и вкладки, связанные с этим сеансом WebDriver;
* Закроет процесс браузера;
* Закроет процесс фонового драйвера;
* Уведомит Selenium Grid о том, что браузер больше не используется, чтобы его можно было использовать в другом сеансе (если мы используем Selenium Grid).

Если же не вызвать quit(), на нашем компьютере будут запущены дополнительные фоновые процессы и порты, что может вызвать проблемы позже.

Также одна из основ при работе с Selenium Webdriver – это поиск (locating) элементов на веб-страницах. WebDriver предлагает ряд встроенных типов селекторов, в том числе поиск элемента по его атрибуту ID. Пример:

const example = driver.findElement(By.id(example'));

Как видно из примера, поиск элементов в WebDriver выполняется на WebDriver-объекте экземпляра. findElement(By) метод возвращает другой фундаментальный тип объекта (WebElement).

WebDriver представляет браузер, в то время как WebElement представляет конкретный узел DOM (элемент управления, например, ссылку или поле ввода и так далее).

Если у нас уже есть ссылка на «найденный» веб-элемент, мы можем сузить область поиска, используя тот же вызов для этого экземпляра объекта:

const example = driver.findElement(By.id(example'));

const partofexample = example.findElement(By.id(partofexample));

Часто возникают ситуации, когда нам необходимо найти сразу все элементы, удовлетворяющие тому или иному условия. Для этого в Selenium Webdriver существует метод driver.findElements(). Этот метод возвращает коллекцию веб-элементов. Если найден только один элемент, он все равно вернет коллекцию (из одного элемента). Если ни один элемент не соответствует локатору, будет возвращен пустой список; данный нюанс в работе метода зачастую используется при формировании тестовых сценариев, так как метод не выдаст undefined ни при каких условиях, а следовательно наш тест не упадет.

В примерах выше мы уже рассмотрели один из возможных способов поиска (по id элемента). Однако таких способов значительно больше и каждый из них удобно использовать в тех или иных обстоятельствах. Рассмотрим данные способы в Таблице 2.1:

Таблица 2.1 – Виды локаторов в Selenium Webdriver

|  |  |
| --- | --- |
| **Локатор** | **Описание** |
| class name | Находит элементы, имя класса которых содержит значение поиска (составные имена классов не допускаются) |
| css selector | Находит элементы, соответствующие селектору CSS |
| id | Находит элементы, атрибут ID которых соответствует поисковому значению |
| name | Находит элементы, атрибут NAME которых соответствует поисковому значению |
| link text | Находит элементы привязки, видимый текст которых соответствует поисковому значению |
| partial link text | Находит элементы привязки, видимый текст которых содержит значение поиска. Если совпадают несколько элементов, будет выбран только первый. |
| tag name | Находит элементы, имя тега которых соответствует поисковому значению |
| xpath | Находит элементы, соответствующие выражению XPath |

В общем, если HTML-идентификаторы доступны, уникальны и постоянно предсказуемы, они являются предпочтительным методом поиска элемента на странице. Они, как правило, работают очень быстро и не требуют большой обработки, связанной со сложным обходом DOM.

Если уникальные идентификаторы недоступны, хорошо написанный селектор CSS является предпочтительным методом поиска элемента. XPath работает так же, как селекторы CSS, но синтаксис сложен и часто бывает трудно отладить. Хотя селекторы XPath очень гибкие, они, как правило, не тестируются на производительность поставщиками браузеров и, как правило, работают довольно медленно. Однако их принято считать самыми надежными.

Стратегии выбора, основанные на linkText и partialLinkText, имеют недостатки в том, что они работают только с элементами ссылки. Кроме того, они вызывают селекторы XPath внутри WebDriver.

Имя тега может быть опасным способом поиска элементов. На странице часто присутствует несколько элементов одного и того же тега. Это в основном полезно при вызове метода findElements (By), который возвращает коллекцию элементов.

Рекомендуется, чтобы локаторы были как можно более компактными и удобочитаемыми. Просить WebDriver обойти структуру DOM - дорогостоящая операция, и чем больше вы сможете сузить область поиска, тем лучше.

Также имеет смысл рассмотреть ожидания в Selenium Webdriver по тому или иному условию. В целом можно сказать, что WebDriver имеет блокирующий API. Поскольку это внепроцессная библиотека, которая указывает браузеру, что делать, и поскольку веб-платформа имеет асинхронную природу, WebDriver не отслеживает активное состояние DOM в реальном времени. Это связано с некоторыми проблемами, которые мы обсудим далее.

По опыту, большинство периодических проблем, возникающих при использовании Selenium и WebDriver, связаны с *условиями*, которые возникают между браузером и инструкциями пользователя. Примером может быть то, что пользователь дает браузеру команду перейти на страницу, а затем получает ошибку отсутствия такого элемента при попытке найти элемент.

Проблема здесь в том, что стратегия загрузки страницы по умолчанию, используемая в WebDriver, ожидает document.readyState изменения «complete» перед возвратом из вызова для перехода. Если элемент добавляется после завершения загрузки документа, этот сценарий в WebDriver может работать с перебоями. Он может быть прерывистым, потому что нельзя дать никаких гарантий относительно элементов или событий, которые запускаются асинхронно без явного ожидания или блокировки этих событий.

Чтобы преодолеть проблему состояния «гонки» между браузером и вашим скриптом WebDriver, большинство клиентов Selenium поставляются с пакетом ожидания. При использовании ожидания мы используем то, что обычно называют явным ожиданием.

Они позволяют нашему коду останавливать выполнение программы или замораживать поток до тех пор, пока условие, которое вы ему передаете, не разрешится. Условие вызывается с определенной частотой, пока не истечет время ожидания. Это означает, что пока условие возвращает ложное значение, оно будет пытаться и ждать.

Поскольку явные ожидания позволяют дождаться возникновения условия, они хорошо подходят для синхронизации состояния между браузером и его DOM, а также вашим скриптом WebDriver.

Мы передаем условие как ссылку на функцию, что ожидание будет выполняться повторно, пока его возвращаемое значение не станет истинным. «Правдивое» возвращаемое значение – это все, что оценивается как логическое истинное на данном языке, например строка, число, логическое значение, объект (включая WebElement) или заполненная (непустая) последовательность или список. Это означает, что пустой список считается ложным. Когда условие является истинным и блокирующее ожидание прерывается, возвращаемое значение из условия становится возвращаемым значением ожидания.

Условие ожидания можно настроить в соответствии с нашими потребностями. Иногда нет необходимости ждать полного тайм-аута по умолчанию, так как штраф за невыполнение успешного условия может быть дорогостоящим.

Ожидание позволяет нам передать аргумент для отмены тайм-аута:

await driver.wait(until.elementLocated(By.id('foo')), 30000)

Условия, доступные в разных языковых привязках, различаются, но это неполный список некоторых из них:

* предупреждение присутствует;
* элемент существует;
* элемент виден;
* заголовок содержит;
* название;
* устаревший элемент;
* видимый текст.

## **2.2. Формализация предметной области программного средства**

Разработанная библиотека «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов будет иметь структуру, как указано на рисунке 2.5.

Как видно из рисунка, директория «framework» включает в себя следующие папки:

* baseElement;
* basePage;
* browser;
* utils.

Непосредственно в данных папках содержаться классы, из которых состоит библиотека «kbv-testdriver».

Для лучшего понимания выбранной структуры библиотеки необходимо разобрать такое понятие как Page object models или Page object pattern.

Объект страницы – это шаблон проектирования, который стал популярным в автоматизации тестирования для улучшения обслуживания тестов и уменьшения дублирования кода. Объект страницы – это объектно-ориентированный класс, который служит интерфейсом для страницы вашего автоматизированного теста. Затем тесты используют методы этого класса объекта страницы всякий раз, когда им нужно взаимодействовать с пользовательским интерфейсом этой страницы. Преимущество состоит в том, что при изменении пользовательского интерфейса для страницы не нужно менять сами тесты, нужно изменять только код внутри объекта страницы. Впоследствии все изменения для поддержки этого нового пользовательского интерфейса расположены в одном месте.

Шаблон дизайна объекта страницы дает следующие преимущества:

* существует четкое разделение между тестовым кодом и кодом страницы, таким как локаторы (или их использование, если вы используете карту пользовательского интерфейса) и макет;
* существует единый репозиторий для сервисов или операций, предлагаемых страницей, вместо того, чтобы эти сервисы были разбросаны по тестам.

В обоих случаях это позволяет вносить любые изменения, необходимые из-за изменений пользовательского интерфейса, в одном месте.

Существует большая гибкость в том, как могут быть спроектированы объекты страницы, но есть несколько основных правил для достижения желаемой ремонтопригодности вашего тестового кода.

Сами объекты страницы никогда не должны делать проверки или утверждения. Это часть вашего теста и всегда должна быть в коде теста, а не в объекте страницы. Объект страницы будет содержать представление страницы и услуги, которые страница предоставляет с помощью методов, но никакой код, связанный с тем, что тестируется, не должен находиться в объекте страницы.

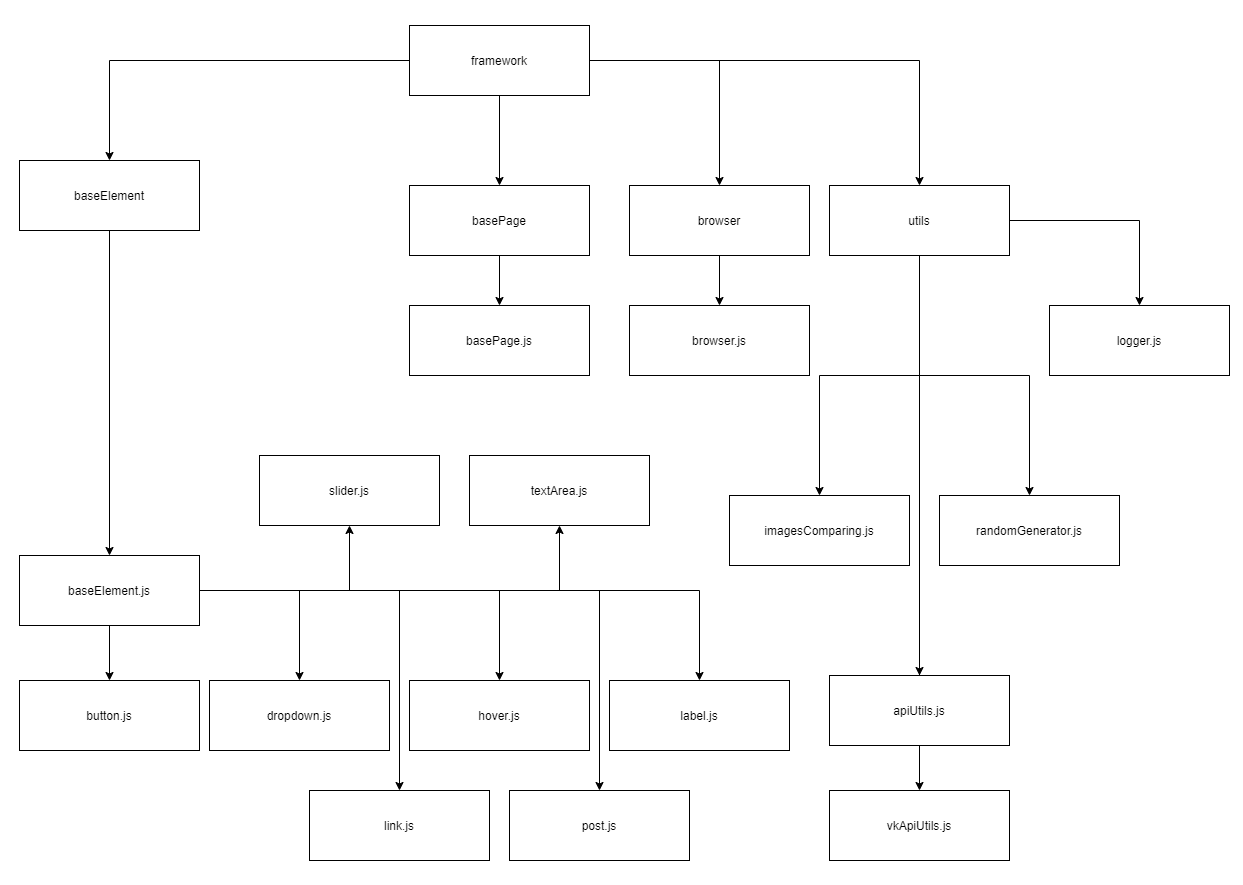


Рисунок 2.5 – Структурное изображение библиотеки «kbv-testdriver»

Существует одна, единственная проверка, которая может и должна быть в объекте страницы, а именно проверка того, что страница и, возможно, критические элементы на странице были загружены правильно. Эта проверка должна выполняться при создании экземпляра объекта страницы.

Объект страницы не обязательно должен представлять все части самой страницы. Те же принципы, что и для объектов страницы, можно использовать для создания «Объектов компонентов страницы», которые представляют дискретные фрагменты страницы и могут быть включены в объекты страницы. Эти объекты-компоненты могут предоставлять ссылки на элементы внутри этих дискретных блоков и методы для использования предоставляемых ими функциональных возможностей. Вы даже можете вкладывать объекты компонентов в другие объекты компонентов для более сложных страниц. Если страница в автоматизированных тестах имеет несколько компонентов или общих компонентов, используемых на всем сайте (например, панель навигации), это может улучшить ремонтопригодность и уменьшить дублирование кода.

Есть и другие шаблоны проектирования, которые также можно использовать при тестировании. Некоторые используют фабрику страниц для создания экземпляров своих объектов страницы.

Теперь мы лучше понимаем с какой целью библиотека «kbv-testdriver» имеет представленную на рисунке 2.5 структуру.

Дадим краткие пояснения относительно того, какие функции возложены на классы, расположенные в данной структуре:

1. baseElement – в данной директории представлены классы, которые ответственны за работу с различными элементами на веб-страницах (кнопки, выпадающие списки, ховер-элементы, надписи, ссылки, публикации (посты), слайдеры, области для ввода текста). Все классы перечисленных элементов наследуются от родительского класса BaseElement.
2. basePage – в данной директории содержится одноименный класс BasePage, который ответственен за работу со веб-страницами. Здесь важно отметить, что у него также есть классы-наследники, однако она располагаются в другой директории, а именно в project, что будет более подробно рассмотрено позже.
3. browser – в данной директории содержится одноименный класс Browser, который отвечает за работу Selenium Webdriver непосредственно с нашим браузером (открытие/закрытие браузера, переход на веб-страницы, увеличение открытого окна и так далее).
4. utils – как логично следует из названия, здесь располагаются все необходимые утилиты, предназначенные для работы с API, логировщик событий, генераторы случайных числе и так далее.

Для более глубокого понимая того какие действия выполняет тот или иной класс, подробно рассмотрим методы, имеющиеся в вышеописанных классах и приведем описание этих методов.

Класс BaseElement содержит следующие методы:

Таблица 2.2 – Описание методов класса BaseElement

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| findElement | Поиск элемента по локатору |
| click | Нажатие курсором мыши на элемент |
| getText | Получение текста элемента |
| getAttribute | Получение одного из атрибутов элемента |
| isElementDisplayed | Проверка отображения элемента |

Как уже было упомянуто ранее, класс BaseElement имеет восемь классов-наследников. Многие из них не содержат в себе специфических методов, а только наследуют базовые из родительского класса BaseElement. Тем не менее, наличие данных классов обусловлено использованием в нашей библиотеке паттерна дизайна Page object pattern. Ниже будут рассмотрены только те классы-наследники, которые содержат в себе характерные только для них методы.

Класс Link содержит следующий специфический метод:

Таблица 2.3 – Описание методов класса Link

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| getHref | Получение атрибута href |

Класс Slider содержит следующий специфический метод:

Таблица 2.4 – Описание методов класса Slider

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| dragAndDropSlider | Перетягивание элемента slider |

Класс TextArea содержит следующие специфические методы:

Таблица 2.5 – Описание методов класса TextArea

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| clearValue | Очистка текстового поля |
| setValue | Ввод клавиш с клавиатуры в текстовое поле |

Далее будут рассмотрены методы, которые имеет родительский класс для работы с веб-страницами BasePage:

Таблица 2.6 – Описание методов класса BasePage

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| findElements | Поиск всех элементов на странице по определенному в методе локатору |
| isDisplayed | Проверка отображение страницы |

Работу и корректный запуск автоматизированных тестов нельзя представить без класса Browser, который отвечает за все манипуляции с веб-браузерами. Рассмотрим методы данного класса в таблице 2.7.

Таблица 2.7 – Описание методов класса Browser

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| initBrowser | Инициализация браузера с опциями |
| quit | Закрытие браузера |
| navigate | Навигация на веб-страницу |
| switchToFrame | Вход во фрейм |
| switchToDefault | Выход к стандартной веб-странице |
| setTimeout | Установка таймаута |
| windowMaximize | Расширение окна браузера |
| wait | Ожидание выполнения условия |
| switchToAlert | Переключение к JavaScript alert-у |
| getCurrentUrl | Получение ссылки на текущую веб-страницу |
| backToPreviousPage | Возврат к предыдущей странице |

Здесь и далее мы будем рассматривать реализуемые утилиты.

Разрабатываемая нами библиотека «kbv-testdriver» на базе Selenium Webdriver предполагает тестирование не только визуальной составляющей веб-сервисов, однако и API. Для этого нам потребуется подключение сторонней библиотеки axios, которая позволяет отправлять различные REST API-запросы на серверную часть веб-сервисов.

Для подключения данной библиотеки для начала необходимо установить npm-пакет (Node Package Manager – менеджер установки пакетов) axios. Для этого в командной строке нашего редактора кода VS Code выполним следующую команду:

npm i --save-dev axios

После чего мы получим сообщение об успешной установке данного пакета. С целью того, чтобы удостовериться в успешном добавлении данного пакета в наш проект, мы можем открыть файл package.json и найти в нем ключ «devDependencies». Здесь отображаются все установленные зависимости (в данном случае пакеты) для нашего проекта. Для корректной работы нашей библиотеки и дальнейшей ее интеграции при написании автоматизированных тестов, нам потребуются и другие npm-пакеты. Рассмотрим все используемые в нашей библиотеке пакеты в таблице 2.8.

Таблица 2.8 – Описание используемых в проекте npm-пакетов

|  |  |
| --- | --- |
| **npm-пакеты** | **Назначение пакета** |
| axios | Выполнение HTTP-запросов |
| chai | Библиотека утверждений (assertions) |
| chromedriver | Драйвер для поддержки Chromium-браузеров |
| geckodriver | Драйвер для поддержки браузера Firefox Mozilla |
| form-data | Работа с формами |
| fs | Работа с файлами |
| image-downloader | Загрузка изображений |
| img-diff-js | Библиотека для сравнения изображений на предмет расхождений |
| lodash | Библиотека, которая предоставляет вспомогательные функции для общих задач программирования с использованием парадигмы функционального программирования |
| mocha | Библиотека, облегчающая тестирование асинхронного кода |
| selenium-webdriver | Фреймворк для взаимодействия с браузером и веб-страницами |

Поскольку мы перечислили все используемые в нашем проекте npm-пакеты, укажем команду для установки всех интересующих нас зависимостей:

npm i --save-dev selenium-webdriver mocha chai axios lodash chromedriver geckodriver form-data fs image-downloader img-diff-js

Итак, после установки зависимостей, мы можем вернуться к рассмотрению реализованной работы с отправкой HTTP-запросов. Поскольку мы уже выбрали и установили необходимую библиотеку, решающую поставленную задачу, нам достаточно написать следующую строчку кода для использования axios в нашем классе ApiUtils:

const axios = require('axios');

Рассмотрим методы класса ApiUtils в таблице 2.9.

Таблица 2.9 – Описание методов класса ApiUtils

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| sendGetRequest | Отправка HTTP-GET запроса |
| sendPostRequest | Отправка HTTP-POSTvзапроса |

Следующей реализуемой нами утилитой будет класс DownloadUtils. Как можно догадаться из его названия, его предназначение – это загрузка изображений на сервер. Для возможности такой загрузки нам необходимо подключить стороннюю библиотеку, которая предоставляется такую возможность. Одним из вариантов подобной библиотеки является npm-пакет image-downloader. Установка и подключение будут аналогичными как было описано выше для библиотеки axios. Класс DownloadUtils включает в себя единственный метод:

Таблица 2.10 – Описание метода класса DownloadUtils

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| downloadImageByUrl | Загрузка изображения |

Для выполнения требования по анализу изменений изображений в процессе обработки веб-сервисом, будет реализован класс ImagesUtils, в основе которого лежит библиотека img-diff-js:

Таблица 2.11 – Описание метода класса ImagesUtils

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| imagesComparing | Сравнение исходного изображения с загруженным |

Для выполнения требования по наличию в нашей библиотеке подсистемы мониторинга работы разрабатываемой библиотеки, или, иными словами, логировщика событий, будет реализован класс Logger:

Таблица 2.12 – Описание метода класса Logger

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| infoLog | Регистрация выполняемых действий в консоли в процессе выполнения тестов с указанием точного времени того или иного события |

Для большинства потенциальных автоматизированных тестов, которые могут быть реализованы с использованием библиотеки «kbv-testdriver», нам могут потребоваться различные генераторы случайных чисел. Предположим, что мы хотим ввести случайное значение в то или иное текстовое поле – это один из возможных вариантов использования нашего класса генерации случайных чисел. Если не прибегнуть к такому подходу, нам придется либо во время каждого тестового прогона вводить случайные значения, либо вовсе «захардкодить» ­­­­­­то или иное значения. Оба перечисленных варианты нельзя считать приемлемыми, поэтому для этих целей нами будет реализован класс RandomGenerators. Рассмотрим его методы:

Таблица 2.13 – Описание методов класса RandomGenerators

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| randomStr | Генерация случайной строки с передаваемым параметром длины |
| getRandomIntInclusive | Генерация случайной строки с передаваемыми параметрами верхней и нижней границ |
| getRandomValueFromArray | Выбор случайного значения из передаваемого параметром массива. Для реализации данного метода используется библиотека lodash |

Поскольку для демонстрации возможностей разрабатываемой нами библиотеки «kbv-testdriver», а также для ее последующего тестирования нами будет разработан проект с реализованными автоматизированными тестами реального веб-сайта, нам необходимо разработать отдельный класс-наследник VkApiUtils, родителем которого является класс ApiUtils. Класс VkApiUtils содержит в себе методы, которые предназначены для работы с открытым API веб-сайта популярной социальной сети vk.com. Рассмотрим подробнее данный класс в таблице 2.14.

Таблица 2.14 – Описание методов класса VkApiUtils

|  |  |
| --- | --- |
| **Метод класса** | **Описание метода** |
| createPost | Создание нового поста со случайно сгенерированным текстом |
| editPost | Редактирование определенного в параметрах поста с помощью случайного текста и загруженного изображения |
| addComment | Добавление случайного комментария к определенному посту |
| getPostLikes | Получение числа лайков и пользователя, совершившего данное действие, для определенного поста |
| deletePost | Удаление определенного поста |
| getWallUploadServer | Получение ссылки для последующей загрузки изображения |
| uploadPhotoToUrl | Загрузка изображения по ссылке |
| saveWallPhoto | Сохранение изображения со стены пользователя |
| getPhotoUrl | Получение ссылки на изображение |

Таким образом, мы рассмотрели все базовые классы, реализованные в нашей библиотеке «kbv-testdriver». Однако, используя данную основу, нам необходимо применить разработанные классы и подходы при тестировании реального веб-сервиса. Для этого следует учитывать, что наша библиотека разработана с применением паттерна дизайна Page Object. Интеграцию библиотеки с тестируемыми веб-сервисами рассмотрим в следующем подпункте.

## **2.3. Интеграция разработанной библиотеки с тестируемым веб-сервисом**

В предыдущем подпункте нами была описана реализации библиотеки автоматизированного тестирования «kbv-testdriver». Все описанные нами классы будут хранится в директории framework, однако для разделения библиотеки и непосредственно тестируемых веб-сервисом, создадим еще одну верхнеуровневую директорию project, где и будет храниться наш проект по тестированию веб-сервисов и все относящиеся к нему классы и модули.

Для демонстрации работы нашей библиотеки было выбрано два веб-сервиса:

1. Веб-сервис «http://the-internet.herokuapp.com». Представляет собой набор веб-страниц, содержащих множество различных веб-элементов. Тем самым является хорошим ресурсом для демонстрации возможностей Selenium Webdriver и непосредственно нашей библиотеки «kbv-testdriver».
2. Веб-сайт «https://vk.com». Популярная социальная сеть с подробной документацией и открытым API. Данный ресурс выбран для демонстрации возможностей разработанной нами библиотеке по совмещению тестирования UI и API веб-сервисов. Более того, мы сможем убедиться в том, что наша библиотека может обрабатывать тестовые случаи с динамической подгрузкой веб-страниц (что зачастую встречается при работе с данным ресурсом).

Для более наглядного представления структуры нашего проекта, рассмотрим рисунок 2.6.

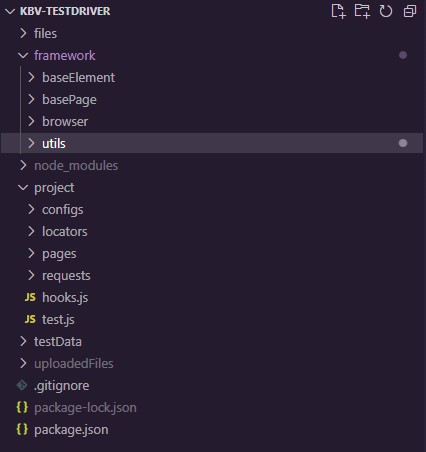


Рисунок 2.6 – Структурное представление проекта

Как видно из рисунка 2.6, наш проект будет включать в себе следующие директории:

* configs;
* locators;
* pages;
* requests.

А также следующие файлы:

* hooks.js;
* test.js.

Рассмотрим подробнее каждый из вышеуказанных элементов.

1. Директория configs. Включает в себя конфигурационный файл с возможностью изменения настроек запуска нашего проекта. Работа с данным конфигурационным файлом будет рассмотрена в разделе по вводу программного средства в эксплуатацию.
2. Директория locators. Простыми словами, локаторы ­– это описанные пути для поиска того или иного веб-элемента. Без них не представляется возможной работа и взаимодействие ни с одним из элементов, поскольку для начала нам необходимо объяснить драйверу как их найти. Существует множество различных способов для поиска веб-элементов, однако самым универсальным и надежным считается использование XPath. XPath – это язык запросов к элементам XML-документа.

В директории locators представлены локаторы для всех тестируемых нами веб-страниц. В качестве примера рассмотрим код локаторов для страницы wallPage, которая в свою очередь относится к ресурсу «https://vk.com»:

const {By} = require('selenium-webdriver');

const wallPageLocators = {

'authorField': By.xpath("//h1[@class='page\_name']"),

findPostWithText: (postId, expectedText) => By.xpath(`//div[contains(@id,'${postId}')]//div[text()='${expectedText}']`),

findNextCommentButton: (postId) => By.xpath(`//div[contains(@class,'replies\_list') and contains(@id,'${postId}')]//a`),

findCommentWithText: (postId, expectedText) => By.xpath(`//div[contains(@id,'${postId}')]//div[@class='wall\_reply\_text' and text()='${expectedText}']`),

findPost: (postId) => By.xpath(`//div[contains(@id,'${postId}')]`),

findPostTextField: (postId) => By.xpath(`//div[contains(@id,'${postId}')]//div[contains(@class,'wall\_post\_text')]`),

findPostAuthorField: (postId) => By.xpath(`//div[contains(@id,'${postId}')]//a[@class='author']`),

findPostCommentField: (postId) => By.xpath(`//div[contains(@id,'${postId}')]//div[@class='wall\_reply\_text']`),

findPostCommentAuthorField: (postId) => By.xpath(`//div[contains(@id,'${postId}')]//a[@class='author']`),

findPostLikeButton: (postId) => By.xpath(`//div[contains(@id,'${postId}')]//a[contains(@class,'like\_btn')]`)

};

module.exports = {wallPageLocators};

Как видно из представленного выше кода, каждый из наших локаторов написан при помощи языка запросов XPath. Кроме того, неспроста был выбран пример локаторов для страницы wallPage: здесь также активно используются динамические локаторы с использованием стрелочных функций. Благодаря этому, при дальнейшем поиске элементов мы можем передавать ту или иную информацию об искомом элементе в качестве параметра нашего локатора, что в свою очередь позволяет нам безошибочно обращаться к нужному элементу, например, к публикации, не переписывая локатор каждый раз.

1. Директория pages. Поскольку библиотека «kbv-testdriver» предполагает использование такого паттерна дизайна создания автоматизированных тестов как Page Object model, нам необходимо учитывать это при формировании структуры проекта. Для этого мы создадим классы-наследники от родительского класса нашей библиотеки (BasePage) для каждой из веб-страниц, встречающихся в рамках тестируемых нами веб-сервисов. Рассмотрим подробнее реализацию класса веб-страницы SliderPage на рисунке 2.7.

Как мы видим из рисунка 2.7, данный класс наследуется от родительского класса BasePage. Здесь используется конструктор для присвоения уникального локатора нашей странице. Это в дальнейшем может использоваться для проверки отображения данной страницы, что, собственно, и происходит в методе sliderPageIsDisplayed(). Также здесь наглядно представлено, что многие из разработанных методов используют локаторы, которые мы описывали в соответствующих веб-страницам классах в директории locators для получения различных веб-элементов. Для взаимодействия с локаторами используется импорт:

const {sliderPageLocators} = require("../locators/sliderPageLocators");

Для возможности последующего обращения к разработанному нами классу SliderPage, необходимо произвести соответствующий экспорт:

module.exports = SliderPage;

1. Директория requests. Данная директория включает в себя переменную, содержащую в себе объект с указанием всех интересующих нас запросов к API веб-сервиса. Как и в случае с классами локаторов, здесь целесообразно использование параметризированных функций, для большей гибкости использования данных запросов.

Рассмотрим код файла requests.js:

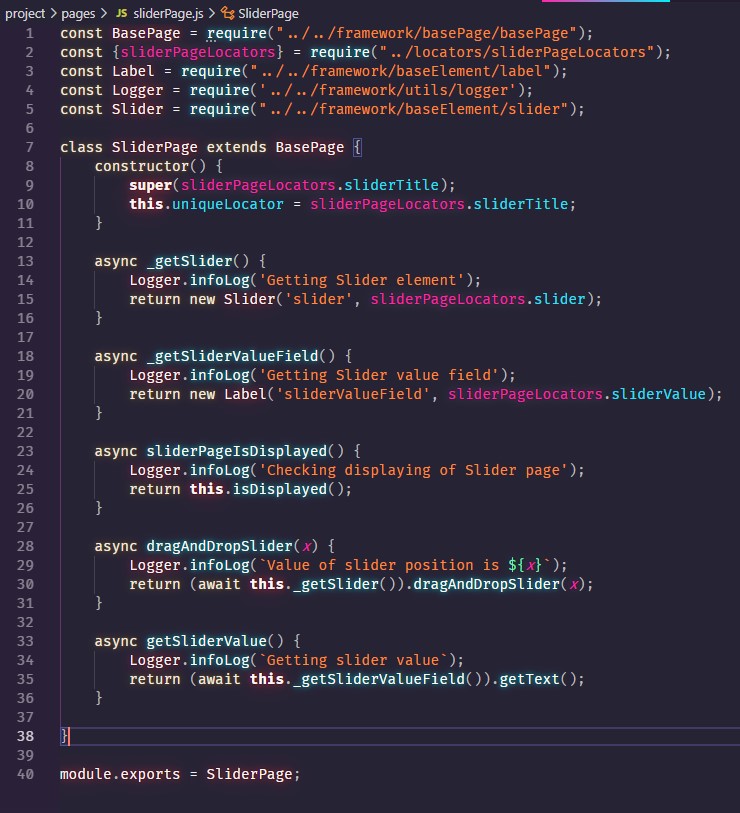


Рисунок 2.7 – Реализация класса веб-страницы SliderPage

const {vkProjectTestData} = require('../../testdata/test.data');

const requests = {

createPost: (randomText) => `${vkProjectTestData.vkApiLink}wall.post?owner\_id=${vkProjectTestData.userId}&message=${randomText}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`,

editPost: (postId, editedText, photoId) => `${vkProjectTestData.vkApiLink}wall.edit?owner\_id=${vkProjectTestData.userId}&post\_id=${postId}&message=${editedText}&attachments=photo${vkProjectTestData.userId}\_${photoId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`,

addComment: (postId, randomComment) => `${vkProjectTestData.vkApiLink}wall.createComment?owner\_id=${vkProjectTestData.userId}&post\_id=${postId}&message=${randomComment}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`,

getPostLikes: (postId) => `${vkProjectTestData.vkApiLink}likes.getList?type=post&owner\_id=${vkProjectTestData.userId}&item\_id=${postId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}&extended=${vkProjectTestData.likesListExt}`,

deletePost: (postId) => `${vkProjectTestData.vkApiLink}wall.delete?post\_id=${postId}&owner\_id=${vkProjectTestData.userId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`,

getWallUploadServer: (postId) => `${vkProjectTestData.vkApiLink}photos.getWallUploadServer?&owner\_id=${vkProjectTestData.userId}?post\_id=${postId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`,

saveWallPhoto: (photo, server, hash) => `${vkProjectTestData.vkApiLink}photos.saveWallPhoto?&user\_id=${vkProjectTestData.userId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}&photo=${photo}&server=${server}&hash=${hash}`,

getPhotoUrl: (photoId) => `${vkProjectTestData.vkApiLink}photos.getById?photos=${vkProjectTestData.userId}\_${photoId}&access\_token=${vkProjectTestData.token}&v=${vkProjectTestData.apiVersion}`

}

module.exports = {requests};

Мы разобрали содержимое всех директорий нашего проекта, однако рассмотрения также требуют следующие файлы: hooks.js и test.js.

1. hooks.js – файл, который содержит в себе хуки Mocha. Хук – технология, позволяющая изменить стандартное поведение тех или иных компонентов информационной системы. Mocha предоставляет хуки before(), after(), beforeEach() и afterEach(). Их следует использовать для создания предварительных условий и очистки после тестов.

В нашем случае мы будем использовать следующие хуки:

* beforeEach() – запускается перед каждым тестом в конкретном блоке;
* afterEach() – запускается после каждого теста в конкретном блоке;

Итак, приведем листинг кода используемых в наших тестах хуков:

const Browser = require('../framework/browser/browser');

const configs = require('../project/configs/configs.json');

beforeEach(async () => {

await Browser.initBrowser(configs.browserName);

});

afterEach(async () => {

await Browser.quit();

});

Из данного листинга кода можно установить, что хуке beforeEach() будет происходить инициализация и запуск нашего браузера, конфигурация которого будет взята непосредственно из нашего файла configs.json. При вызове хука afterEach() будет происходить закрытие и завершение сеанса открытого ранее браузера.

Таким образом, у пользователя не будет необходимости всякий раз прописывать в тестовом файле открытие браузера, а также необходимости вручную закрывать браузер или же указывать команду закрытия при каждом тестовом прогоне.

1. test.js – файл, содержащий все разработанные нами тестовые сценарии. Для его успешной реализации и последующего запуска, необходимо импортировать все используемые в сценариях страницы, утилиты, тестовые данные. Эта часть нашего тестового файла отображена на рисунке 2.8.

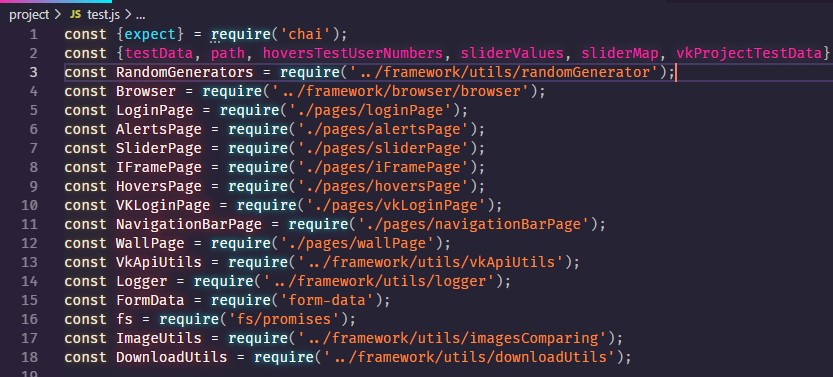


Рисунок 2.8 – Импорт страниц, утилит и тестовых данных в test.js

Для имплементации самих тестовых сценариев нами ранее был выбран тестовый фреймворк Mocha. Для его установки используем следующую команду:

npm install --save-dev mocha

Рассмотрим подробнее базовый синтаксис использования Mocha. Как правило, тесты будут содержать в себе следующие функции:

* describe() – это способ группировать тесты в Mocha.js. Мы можем вкладывать свои тесты в группы по своему усмотрению;
* it() – функция, в которой будут непосредственно располагаться наши тестовые примеры;
* before()/beforeEach()/after()/afterEach() – используемые хуки (было рассмотрено ранее).

Итак, мы разработали структуру нашего проекта, добавили все необходимые нам локаторы, веб-страницы, запросы к API, импортировали все необходимые нам пакеты и классы, установили и настроили тестовый фреймворк. Теперь можем приступить непосредственно к написанию автоматизированных тестов.

Рассмотрим пример написания тестового сценария для веб-сайта «https://vk.com» на рисунке 2.9.

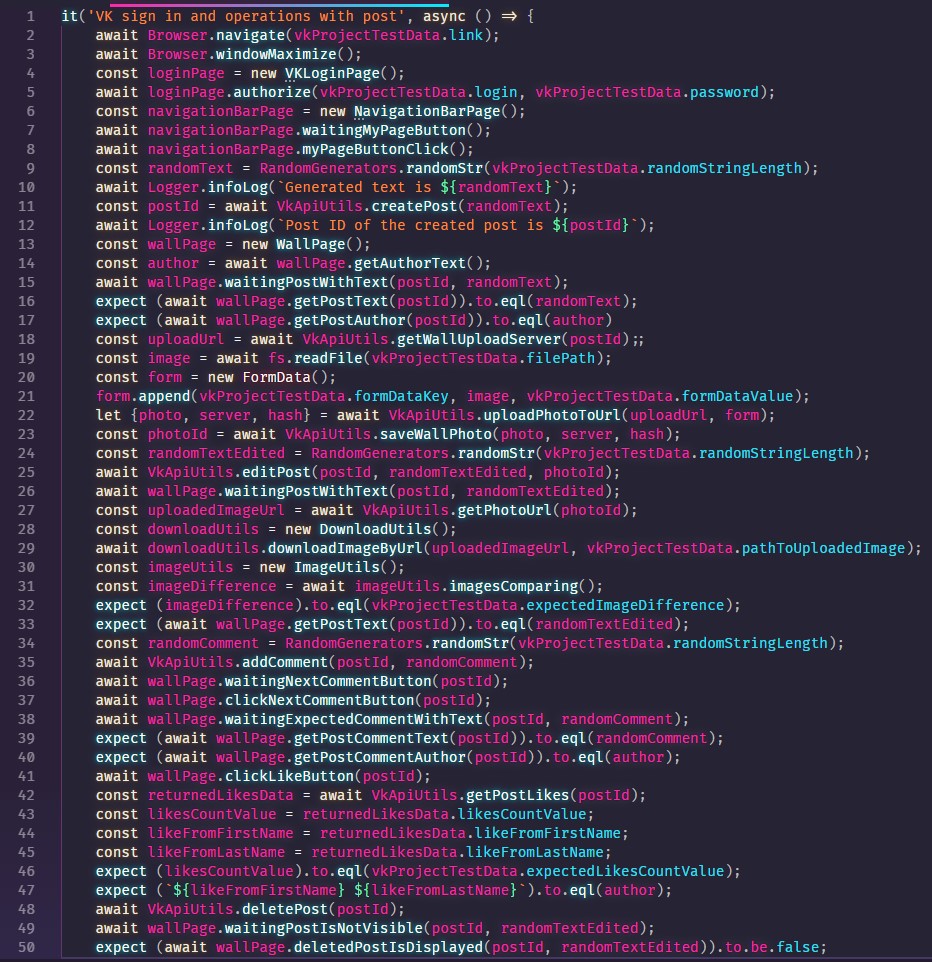


Рисунок 2.9 – Пример тестового сценария для веб-сайта «https://vk.com»

Тестовые сценарии должны быть читабельными и данное правило было применено при их разработке в рамках данного дипломного проекта. Для понимания логики реализации автоматизированного сценария рассмотрим построчно приведенный выше пример.

1. Вызов функции для реализации тестов.
2. Открытие браузера и переход к указанной веб-странице.
3. Развертывание окна браузера на весь экран.
4. Создание нового экземпляра класса VKLoginPage.
5. Прохождение процесса авторизации.
6. Создание нового экземпляра класса NavigationBarPage.
7. Ожидание появления кнопки MyPageButton для последующего обращения к ней.
8. Нажатие левой кнопкой мыши по MyPageButton.
9. Генерация случайного текста.
10. Регистрация случайного текста из шага №9 в консоли.
11. Создание публикации со случайным текстом при помощи вызова соответствующего API.
12. Регистрация идентификатора публикации из шага №11 в консоли.
13. Создание нового экземпляра класса WallPage.
14. Получение автора созданной ранее публикации.
15. Ожидание появления созданной ранее публикации в графическом интерфейсе пользователя.
16. Проверка утверждения: содержание созданной нами публикации соответствует случайно сгенерированному тексту.
17. Проверка утверждения: автор созданной нами публикации соответствует ожидаемому.
18. Получение URL сервера для загрузки изображений через обращение к API.
19. Сохранение в переменную заранее подготовленного изображения.
20. Создание нового экземпляра класса FormData.
21. Заполнение формы с изображением для последующей загрузки на сервер.
22. Загрузка изображения на сервер.
23. Получение идентификатора загруженного изображения.
24. Генерация нового случайного текста для последующего использования при изменении публикации.
25. Изменение ранее созданной публикации: изображение из шага №23 и случайный текст из шага №24.
26. Ожидание появление измененной публикации.
27. Получение URL ранее загруженного изображения из шага №23.
28. Создание нового экземпляра класса DownloadUtils.
29. Сохранение изображения из нашей публикации в локальную директорию.
30. Создание нового экземпляра класса ImageUtils.
31. Сравнение сохраненного изображение в шаге №29 с заранее подготовленным в шаге №19.
32. Проверка утверждения: изображения из шага №31 должны быть эквивалентны.
33. Проверка утверждения: измененный текст публикации эквивалентен ожидаемому.
34. Генерация случайной строки заданной длины.
35. Добавление комментария к созданной нами публикации.
36. Ожидание появления кнопки для получения всех доступных комментариев.
37. Нажатие левой кнопкой мыши по кнопке получения всех доступных комментариев.
38. Ожидание отображения оставленного комментария.
39. Проверка утверждения: комментарий в публикации соответствует ожидаемому.
40. Проверка утверждения: автор комментария в публикации соответствует ожидаемому.
41. Нажатие левой кнопкой мыши по кнопке LikeButton.
42. Получение количества лайков тестируемой нами публикации.
43. Сохранение количества лайков публикации в переменную.
44. Сохранение имени автора лайка публикации в переменную.
45. Сохранение фамилии автора лайка публикации в переменную.
46. Проверка утверждения: количество лайков публикации соответствует ожидаемому.
47. Проверка утверждения: имя и фамилия автора лайка соответствует ожидаемому.
48. Удаление созданной нами публикации для тестирования.
49. Ожидание удаления публикации.
50. Проверка утверждения: удаленная нами тестируемая публикация больше не отображается.

Итак, мы подробно рассмотрели каждую строку кода нашего тестового сценария. Для того чтобы убедиться в достаточной читабельности кода, мы можем сравнить приведенные выше комментарии с представленным кодом нашего тестового сценария на рисунке 2.9.

Стоит отметить, что это достаточно объемный тестовый сценарий в разрезе количества и сложности выполняемых действий, однако сам код получился весьма компактным, что также свидетельствует о правильности выбранного паттерна разработки автоматизированных тестов.

## **2.4. Описание и реализация используемых в библиотеке алгоритмов**

Алгоритм (algorithm) — это любая корректно определенная вычислительная процедура, на вход (input) которой подается некоторая величина или набор величин, и результатом выполнения которой является выходная (output) величина или набор значений. Таким образом, алгоритм представляет собой последовательность вычислительных шагов, преобразующих входные величины в выходные.

На рисунке 2.10 представлена блок-схема алгоритма вызова браузера из разработанной нами библиотеки «kbv-testdriver».

Данный алгоритм реализован непосредственно в Selenium Webdriver и вызывается при инициализации объекта браузера при вызове функции build():

/\*\*

\* Creates a new WebDriver client based on this builder's current

\* configuration.

\*

\* This method will return a {@linkplain ThenableWebDriver} instance, allowing

\* users to issue commands directly without calling `then()`. The returned

\* thenable wraps a promise that will resolve to a concrete

\* {@linkplain webdriver.WebDriver WebDriver} instance. The promise will be

\* rejected if the remote end fails to create a new session.

\*

\* @return {!ThenableWebDriver} A new WebDriver instance.

\* @throws {Error} If the current configuration is invalid.

\*/

build(): ThenableWebDriver;

Поскольку разработанная нами библиотека «kbv-testdriver» способна работать с несколькими браузерами, а именно: Google Chrome и Mozilla Firefox, а также предполагает возможность выбора языка браузера при запуске тестов, нами был реализован алгоритм, который определяет поведение класса Browser при инициализации браузера (здесь имеется в виду метод initBrowser()).

Листинг кода алгоритма проверки и настройки выбранного пользователем браузера и языка браузера:

static async initBrowser(browserName) {

if (browserName == 'chrome') {

Logger.infoLog('Chrome browser initialization');

let chromeCapabilities = Capabilities.chrome();

chromeCapabilities.set("goog:chromeOptions", {args: [`--lang=${configs.browserLanguage}`]});

this.driver = await new Builder().forBrowser(browserName).withCapabilities(chromeCapabilities).build();

}

if (browserName == 'firefox') {

Logger.infoLog('Firefox browser initialization');

let options = new firefox.Options().setPreference('intl.accept\_languages', `${configs.browserLanguage},${configs.browserLanguage}`);

this.driver = await new Builder().forBrowser(browserName).setFirefoxOptions(options).build();

}

}

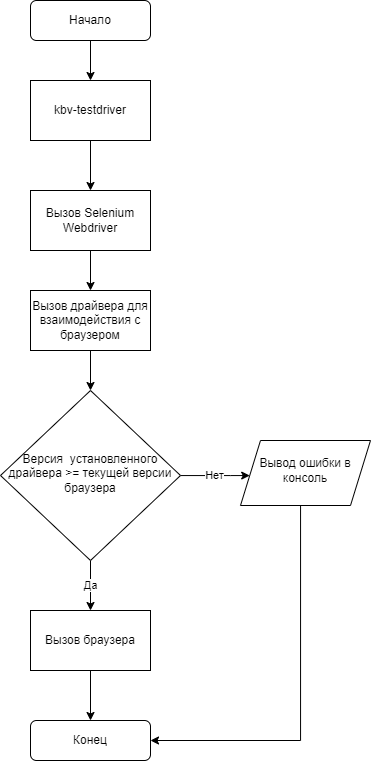


Рисунок 2.10 – Блок-схема алгоритма вызова браузера

На рисунке 2.11 представлена блок-схема алгоритма проверки и настройки выбранного пользователем браузера и языка браузера.

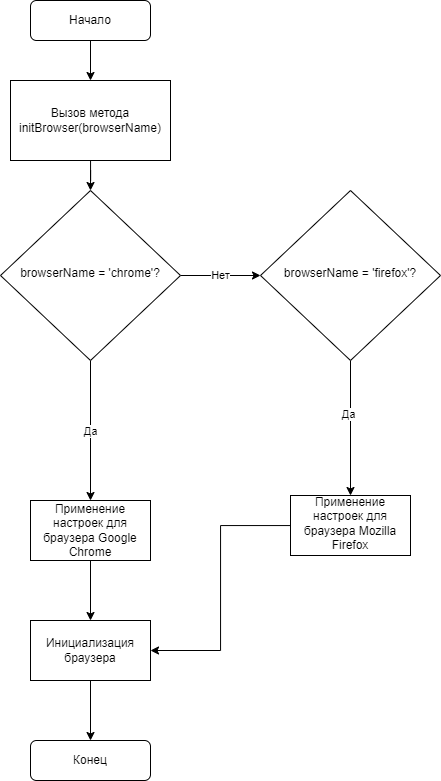


Рисунок 2.11 – Блок-схема алгоритма проверки и настройки выбранного пользователем браузера

Разработанная библиотека «kbv-testdriver» имеет возможность взаимодействия с API-интерфейсами веб-сервисов, посредством отправки HTTP-запросов. Для удобства отслеживания непредвиденных статус-кодов (тех, которые отличаются от 2xx) и ошибок в ответах от сервера, необходимо предусмотреть соответствующий алгоритм-перехватчик.

Листинг кода алгоритма обработки ошибок и непредвиденных статус-кодов в ответах от сервера:

axios.get('/user/number)

.catch(function (error) {

if (error.response) {

// Запрос выполнен, и сервер отправил Вам статус код

// код выпададет из диапазона 2хх (ошибка)

console.log(error.response.data);

console.log(error.response.status);

console.log(error.response.headers);

} else if (error.request) {

// Запрос был сделан, но ответ не получен

// `error.request` - экземпляр XMLHttpRequest в браузере,

// http.ClientRequest экземпляр в node.js

console.log(error.request);

} else {

// Что-то пошло не так, вернулась ошибка

console.log('Error', error.message);

}

console.log(error.config);

});

На рисунке 2.12 представлена блок-схема алгоритма обработки ошибок и непредвиденных статус-кодов в ответах от сервера.

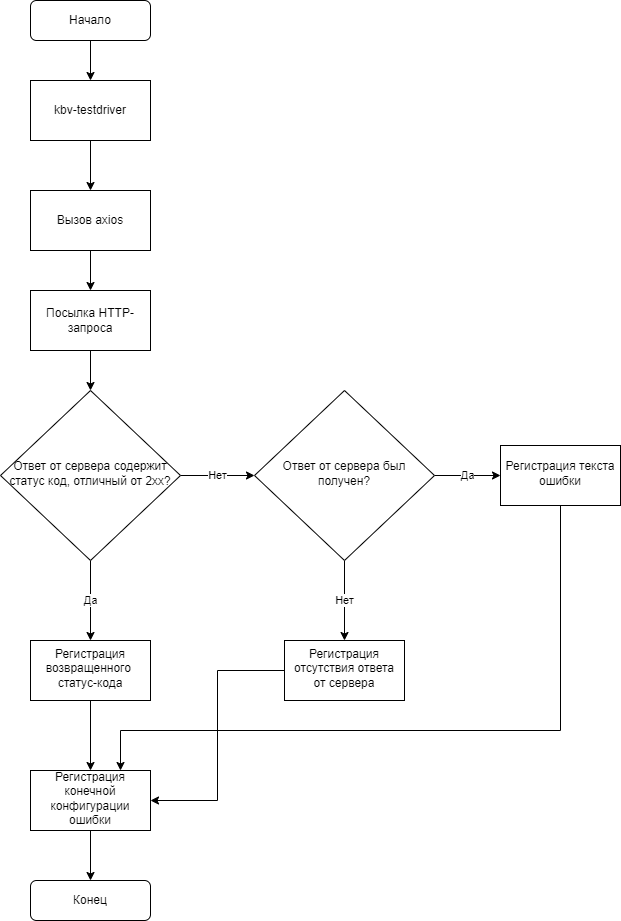


Рисунок 2.12 – Блок-схема алгоритма обработки ошибок и непредвиденных статус-кодов в ответах от сервера

## **2.5 Тестирование программного средства**

При проведении тестирования работоспособности разработанной нами библиотеки «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов, нам необходимо убедиться в том, способно ли наше программное средство выполнять поставленную перед ним задачу, а именно возможность написания автоматизированных тестов пользовательского интерфейса и HTTP-запросов.

Поскольку нами были выбраны следующие веб-сервисы для демонстрации возможностей библиотеки:

* «http://the-internet.herokuapp.com»;
* «https://vk.com».

Нам необходимо разработать тестовые сценарии, которые покрывают максимум возможностей нашей библиотеки «kbv-testdriver» и при этом представляют собой целесообразные, логичные и законченные тестовые случаи.

Начнем с простейшей проверки авторизации пользователя в сервисе:

Таблица 2.15 – Тестовый сценарий «Basic Authorization»

|  |  |
| --- | --- |
| **Шаг** | **Ожидаемый результат** |
| Перейти по ссылке <http://the-internet.herokuapp.com/basic_auth> и пройти базовую авторизацию | Отображается текст: «Congratulations! You must have the proper credentials.» |

Весь интересующий нас код, относящийся к каждому из тестовых сценариев, можно найти в Приложении Б (листинг программного кода). В качестве демонстрации успешности выполнения программы здесь и далее мы будем приводить рисунки вывода консоли нашей программы и все зарегистрированные события во время ее выполнения. Результаты выполнения первого тестового сценария приведем на рисунке 2.13.

Для запуска нашей программы используем следующую команду:

npm test

Как видим, наш тестовый сценарий успешно пройден.

Для проверки корректности работы разработанной нами библиотеки «kbv-testdriver» с JavaScript-оповещениями (alerts), разработаем соответствующий тестовый сценарий:

Таблица 2.16 – Тестовый сценарий «Alerts»

|  |  |
| --- | --- |
| **Шаг** | **Ожидаемый результат** |
| Перейти по ссылке [http://the-internet.herokuapp.com/javascript\_alerts](http://the-internet.herokuapp.com/javascript_alerts%20) | Страница с оповещениями открыта |
| Нажать на кнопку «Click for JS Alert» | Отображается текст «I am a JS Alert» |
| Нажать на кнопку «OK» | Оповещение скрыто.  В секции «Result» отображается текст «You successfully clicked an alert» |
| Нажать на кнопку «Click for JS Confirm» | Отображается текст «I am a JS Confirm» |
| Нажать на кнопку «OK» | Оповещение скрыто.  В секции «Result» отображается текст «You clicked: OK» |
| Нажать на кнопку «Click for JS Prompt» | Отображается оповещение «I am a JS prompt» |
| Ввести случайно сгенерированный текст и нажать кнопку «OK» | Оповещение скрыто.  Отображается текст «You entered: <random text>» |

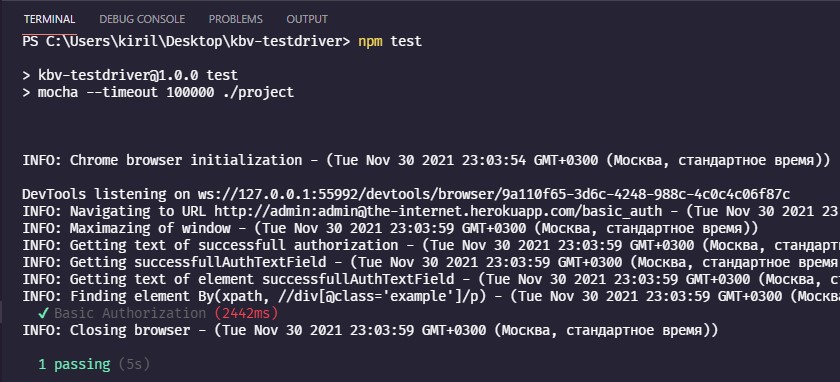


Рисунок 2.13 – Вывод консоли при выполнении тестового сценария «Basic Authorization»

Результаты выполнения второго тестового сценария приведем на рисунке 2.14.

Для проверки корректности работы разработанной нами библиотеки «kbv-testdriver» с JavaScript-cлайдером (slider), разработаем следующий тестовый сценарий:

Таблица 2.17 – Тестовый сценарий «Slider»

|  |  |
| --- | --- |
| **Шаг** | **Ожидаемый результат** |
| Перейти по ссылке <http://the-internet.herokuapp.com/horizontal_slider> | Страница с слайдером открыта |
| Установить для слайдера случайное значение | Значение слайдера установлено верно |

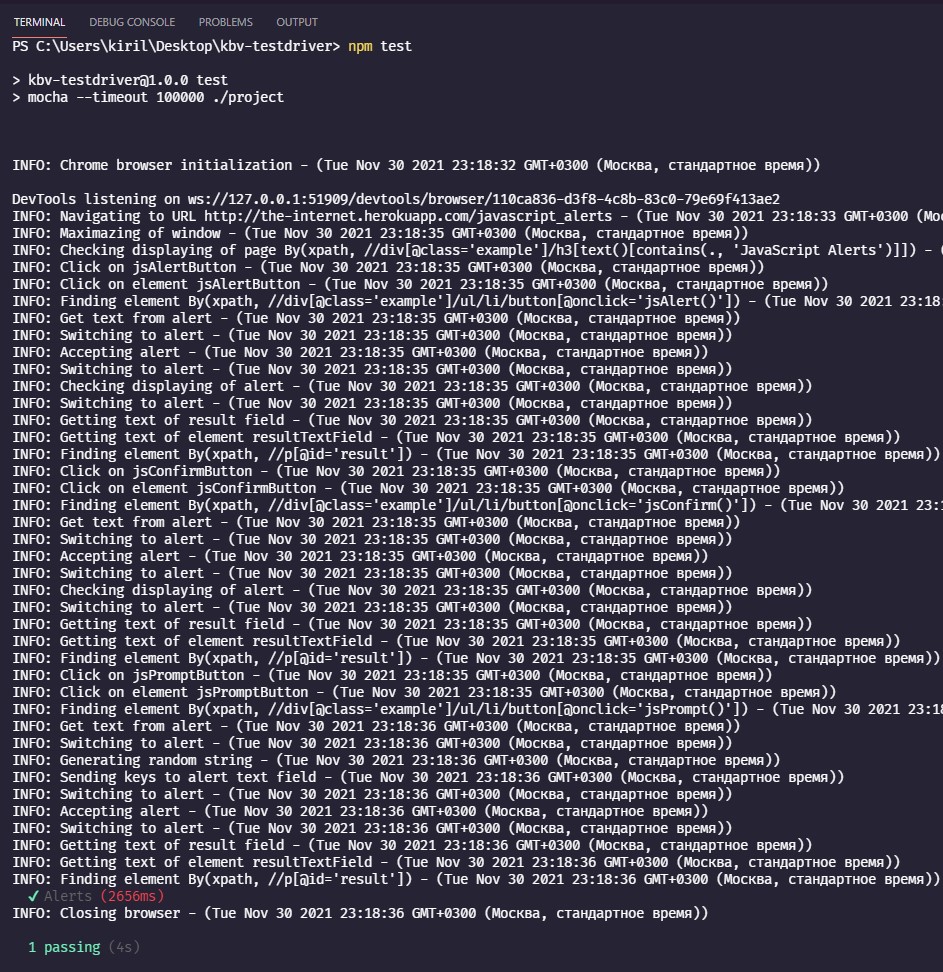


Рисунок 2.14 – Вывод консоли при выполнении тестового сценария «Alerts»

Результаты выполнения третьего тестового сценария приведем на рисунке 2.15.

Для проверки корректности работы разработанной нами библиотеки «kbv-testdriver» с JavaScript-hover элементами, разработаем следующий тестовый сценарий:

Таблица 2.18 – Тестовый сценарий «Hovers»

|  |  |
| --- | --- |
| **Шаг** | **Ожидаемый результат** |
| Перейти по ссылке [http://the-internet.herokuapp.com/hovers](http://the-internet.herokuapp.com/horizontal_slider) | Открыта страница с обложками пользователей |
| Передвинуть курсор на наименование пользователя | Имя пользователя корректно.  Ссылка на профиль пользователя отображается |
| Нажать кнопкой мыши на наименование пользователя | Текущий URL совпадает со ссылкой из предыдущего шага |
| Вернуться на предыдущую страницу | Открыта страница с обложками пользователей |

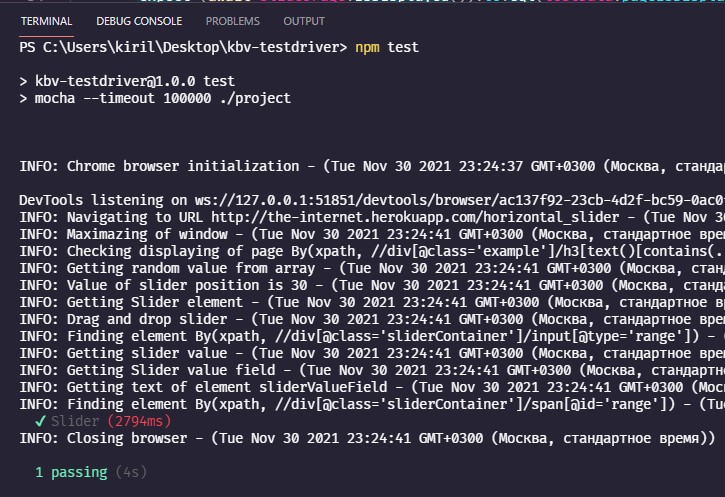


Рисунок 2.15 – Вывод консоли при выполнении тестового сценария «Slider»

Результаты выполнения четвертого тестового сценария приведем на рисунке 2.16.

Для проверки корректности работы разработанной нами библиотеки «kbv-testdriver» с frame-элементами, разработаем следующий тестовый сценарий:

Таблица 2.19 – Тестовый сценарий «Frame»

|  |  |
| --- | --- |
| **Шаг** | **Ожидаемый результат** |
| Перейти по ссылке <http://the-internet.herokuapp.com/iframe> | Открыта страница редактирования текста |
| Выровнять текст по левому краю | Форматирование текста корректно |
| Для половины приведенных текстовых символов изменить размер шрифта | Форматирование текста корректно |
| Нажать на кнопку создания нового документа | Текстовое поле пусто.  Форматирование документа сброшено к стандартному |

Результаты выполнения пятого тестового сценария приведем на рисунке 2.17.

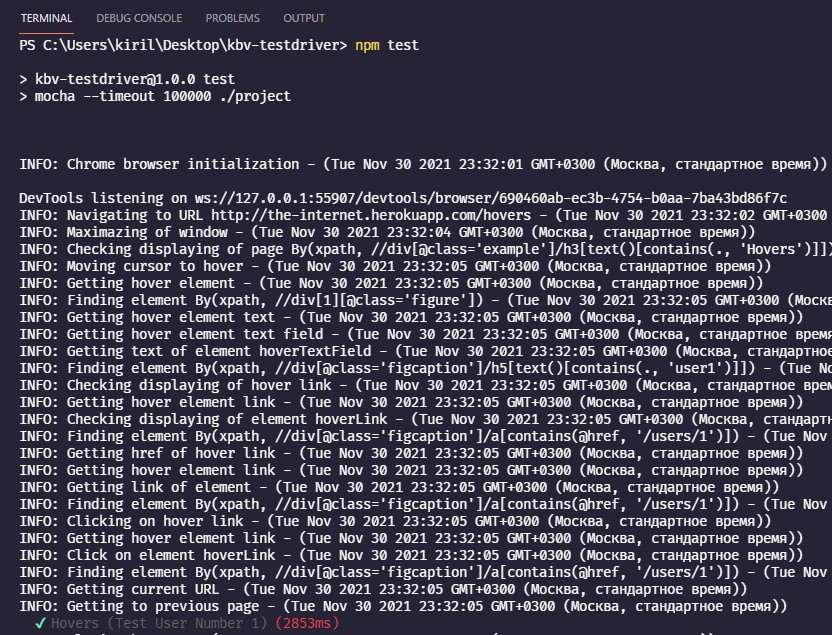


Рисунок 2.16 – Вывод консоли при выполнении тестового сценария «Hovers»

Далее перейдем к проверке работоспособности нашего программного средства при взаимодействии с веб-сервисом «https://vk.com». Данный тестовый сценарий особо интересен нам по трем причинам:

* в данном веб-сервисе активно используется динамическая подгрузка веб-страниц с изменением HTML-структуры документа, что позволит нам проверить на практике реализованную в библиотеке «kbv-testdriver» работу с неявными ожиданиями при помощи реализации метода wait();
* данный веб-сервис имеет открытое API, благодаря чему мы можем проверить реализованную нами в библиотеки работу с запросами;
* данный веб-сервис предполагает возможность загрузки различных изображений, что позволяет нам проверить соответствующую функциональность нашей библиотеки.

Кроме того, тестовый сценарий «VKapi» наиболее объемный из всех ранее нами рассмотренных:

Таблица 2.20 – Тестовый сценарий «VKapi»

|  |  |  |
| --- | --- | --- |
| **Тип** | **Шаг** | **Ожидаемый результат** |
| UI | Перейти по ссылке  [https://vk.com/](http://the-internet.herokuapp.com/javascript_alerts%20) | Соответствующая страница открыта |
| UI | Пройти процесс авторизации | Авторизация успешно пройдена |
| UI | Перейти на веб-страницу «Моя страница» | Отображается веб-страница «Моя страница» |
| API | С помощью запроса к API создать запись со случайно сгенерированным текстом на стене и получить идентификатор записи из ответа | Соответствующая запись со случайно сгенерированным текстом успешно создана |
| UI | Не обновляя страницу убедиться, что на стене появилась запись с нужным текстом от правильного пользователя | Соответствующая запись отображается на веб-странице. Текст и пользователь соответствуют ожидаемым. |
| API | Отредактировать запись через запрос к API – изменить текст и добавить (загрузить) любое изображение | Запись изменена ожидаемым образом: текст и изображение соответствуют ожидаемым |
| UI | Не обновляя страницу убедиться, что изменился текст сообщения и добавилось загруженное изображение (убедиться, что изображения одинаковые) | Обновленный текст соответствует ожидаемому.  Изображение идентично тому, которое было загружено на сервер. |
| API | Используя запрос к API добавить комментарий к записи со случайным текстом | Комментарий успешно добавлен |
| UI | Не обновляя страницу убедиться, что к нужной записи добавился комментарий от правильного пользователя | Добавленный комментарий соответствует ожидаемому |
| UI | Через UI оставить лайк к записи | Лайк успешно оставлен |
| API | Через запрос к API убедиться, что у записи появился лайк от правильного пользователя | В ответе запроса присутствует информация о лайке от ожидаемого пользователя |
| API | Через запрос к API удалить созданную запись | В ответе запроса отмечена успешность выполнения операции |
| UI | Не обновляя страницу убедиться, что запись удалена | Соответствующая больше не представлена на веб-странице |

Результаты выполнения шестого тестового сценария приведем на рисунках 2.18 и 2.19.

Нами было разработано и запущено шесть различных тестовых сценариев, которые тестируют и наглядно демонстрируют работоспособность нашей библиотеки «kbv-testdriver». Как видно из рисунков 2.13 – 2.19 каждый из проверяемых сценариев был успешно пройден (по результатам запуска программного средства мы всякий раз получали зеленую галочку напротив наименования тестового сценария и зарезервированное слово «passing», также окрашенное в зеленый цвет)

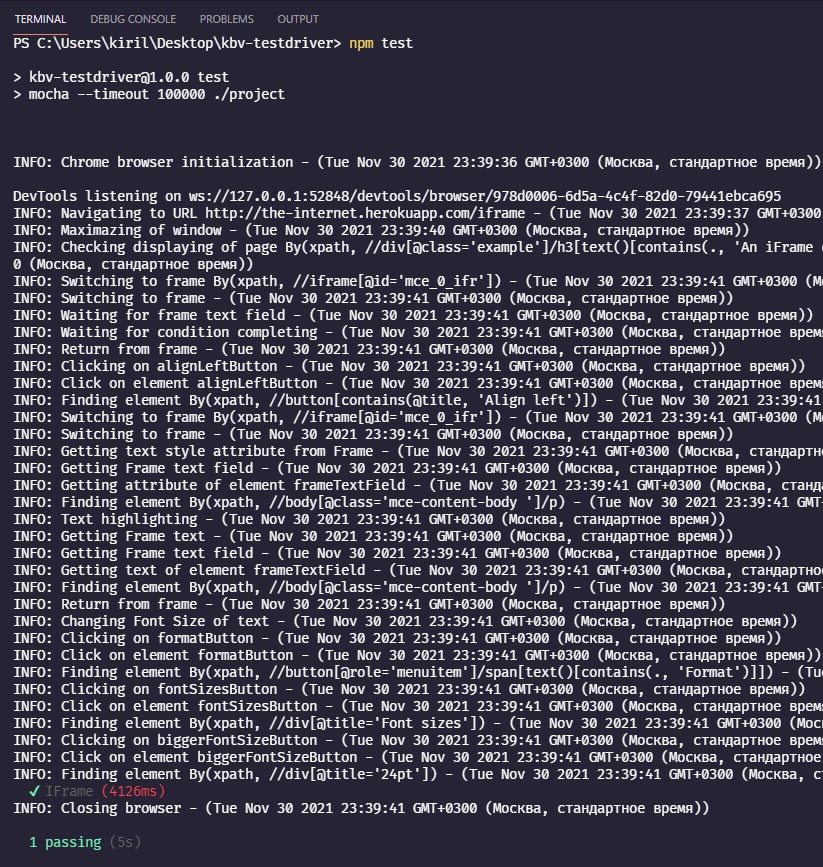


Рисунок 2.17 – Вывод консоли при выполнении тестового сценария «Frame»

Однако, помимо успешных (позитивных) сценариев нам необходимо убедиться в том, что наше программное средство корректно обрабатывает негативные сценарии по результатам тестовых прогонов. В консоль должно выводиться сообщение о негативном результате при выполнении нами проверок, с указанием конкретного тестового сценария, в котором была допущена ошибка. Для этих целей намеренно допустим такую ошибку в тестовом сценарии «Alerts», а также запустим данный тест совместно с «Basic Authorization», для того чтобы убедиться в независимости выполнения нескольких тестовых сценариев при наличии ошибки в одном из них.

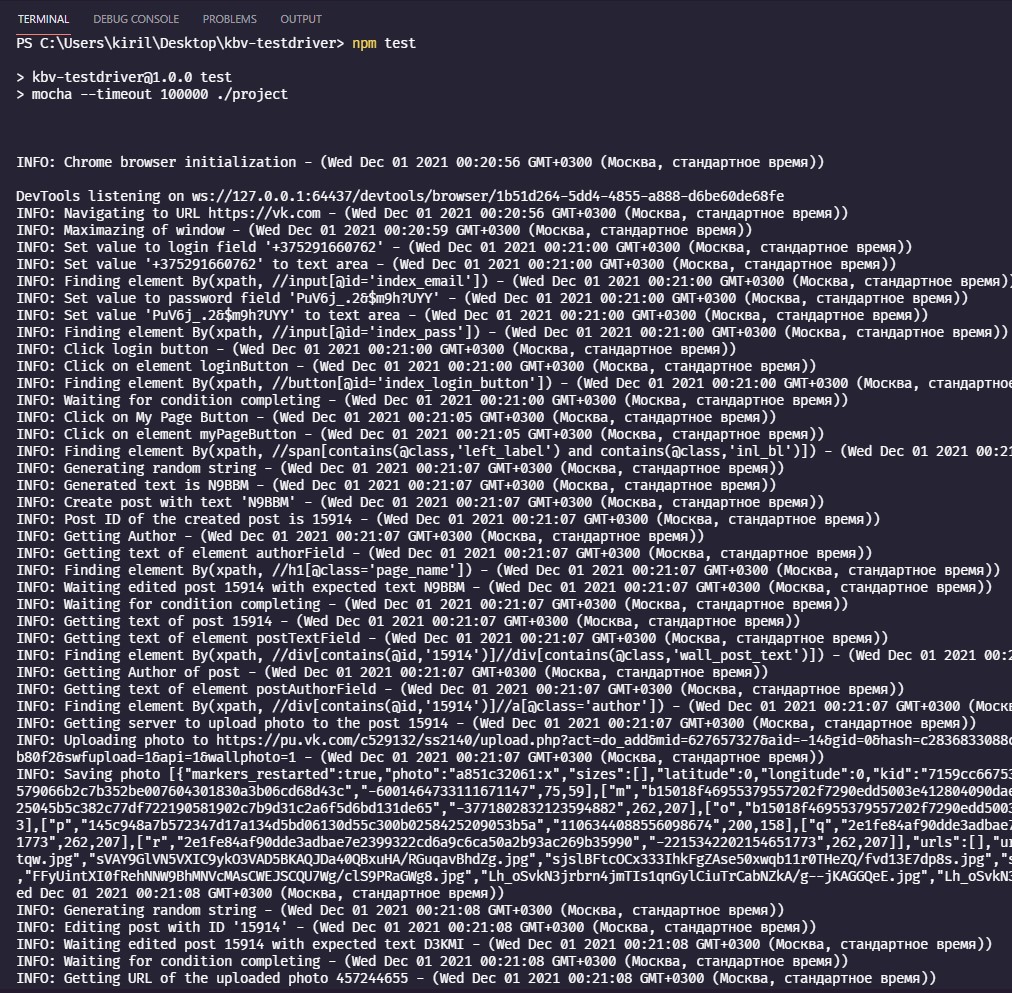


Рисунок 2.18 – Вывод консоли при выполнении тестового сценария «VKapi»

Итак, добавим заведомо ложный ожидаемый результат в следующую строку:

expect(await(alertsPage.getResultText()).to.eql(testData.resultPromptText("negative\_test")));

Теперь запустим тест заново и проанализируем его результаты. Результаты вывода данного теста в консоль представлены на рисунке 2.20.

По результатам прохождения нашего теста мы можем наблюдать следующее сообщение:

1 passing (7s)

1 failing

Оно свидетельствует о том, что наш первый тестовый сценарий был выполнен с положительным результатом за временной интервал равный 7 секундам. Второй же тестовый сценарий был завершен с негативным результатом.

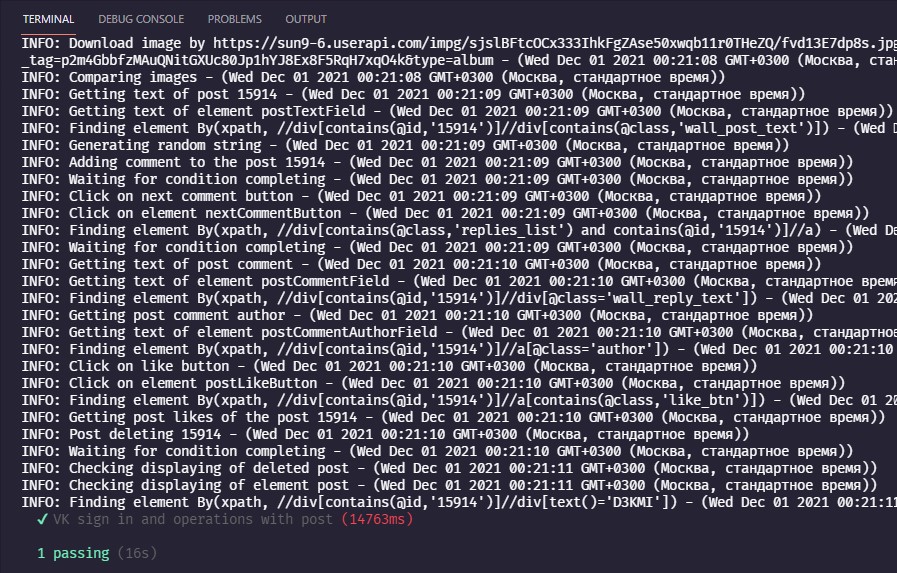


Рисунок 2.19 – Вывод консоли при выполнении тестового сценария «VKapi»

Также мы можем наблюдать развернутое сообщение о том, чем конкретно вызван негативный результат нашего тестового прогона «Alerts»:

AssertionError: expected 'You entered: FFHA079' to deeply equal 'You entered: negative\_test'

+ expected - actual

-You entered: FFHA079

+You entered: negative\_test

Как мы видим из данного консольного сообщения, наш фактический результат не соответствует ожидаемому, а именно:

* фактический (актуальный результат) – negative\_test;
* ожидаемый результат – FFHA079, что эквивалентно сгенерированному нами ранее для проверки случайному тексту.

По итогам тестирования работоспособности разработанной нами библиотеки «kbv-testdriver», было установлено, что библиотека способна корректно функционировать в связке с выбранным нами тестовым фреймворком Mocha, а также выбранный паттерн дизайна тестов удачно вписывается в общую концепцию. Результатом вывода работы программного средства в консоль может быть как позитивный, как и негативный сценарии, что также было установлено с приведением конкретных примеров.

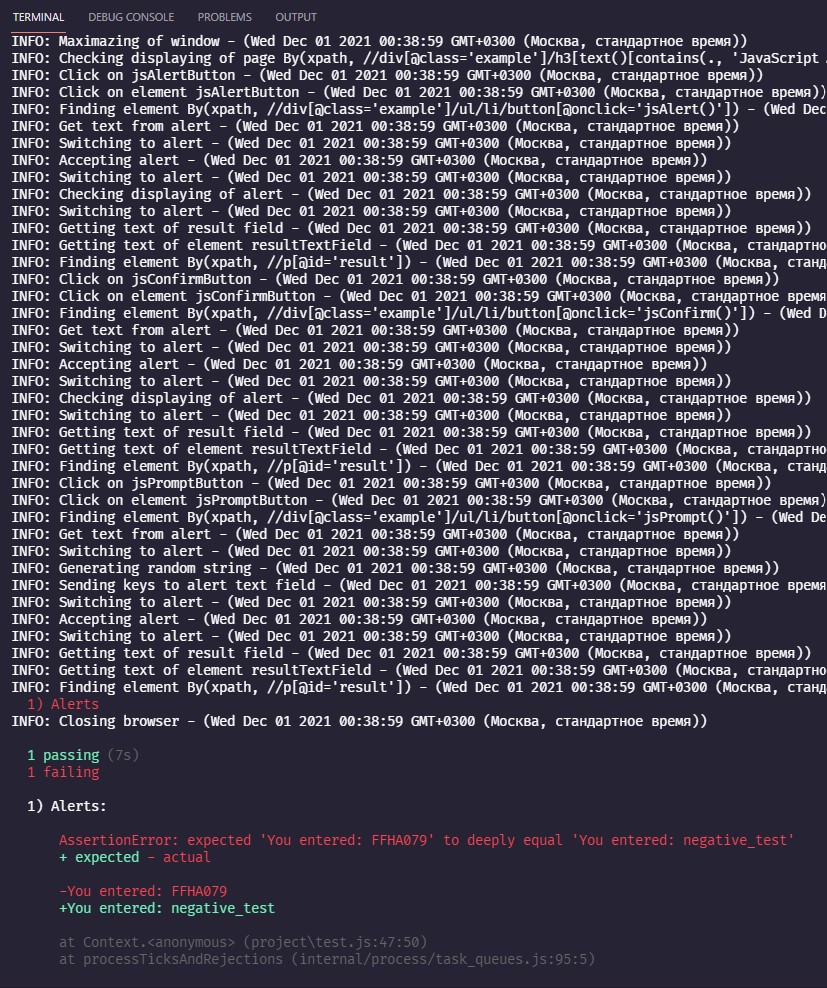


Рисунок 2.20 – Вывод консоли при выполнении тестового сценария «Alerts», с учетом наличия заведомого ложного результата в тестах

# **ОЦЕНКА КОЛИЧЕСТВЕННЫХ ПОКАЗАТЕЛЕЙ ФУНКЦИОНИРОВАНИЯ ПРОГРАММНОГО СРЕДСТВА**

## **Оценка временных показателей программного средства**

Временные показатели являются крайне важным параметром, когда речь заходит об автоматизации тестирования, поскольку это один из определяющих факторов внедрения автоматизации тестирования в тот или иной проект. Качественно разработанные автоматизированные тесты в долгосрочной перспективе экономят много времени и человеческих ресурсов, поскольку мы можем автоматизировать часто повторяющиеся тестовые сценарии в рамках регрессионного тестирования.

Для того чтобы оценить временные показатели при запуске разработанных нами автоматизированных тестов, а также стабильность их функционирования при запусках, нам необходимо произвести запуски одних и тех же тестовых сценариев на обоих поддерживаемых браузерах, а затем внести полученные данные в таблицу 3.1. Для оценки был выбран веб-сервис «http://the-internet.herokuapp.com», поскольку в рамках его тестирования нами уже ранее были разработаны пять различных тестовых сценариев. Благодаря своей изолированности, их запуск не займет много времени, при этом мы сможем оценить временные показатели при прогонах тестов, которые взаимодействуют с большим количеством разнообразных веб-элементов.

Используемый нами тестовый фреймворк Mocha содержит в себе функциональность по выводу в консоль времени, затрачиваемого на каждый тестовый сценарий по отдельности (пример на рисунке 3.1) и на весь тестовый прогон в совокупности (рисунок 3.2).

Таблица 3.1 – Временные показатели во время тестовых прогонов веб-сервиса «http://the-internet.herokuapp.com».

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Итерация, номер | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Google Chrome, с | 28 | 27 | 28 | 27 | 27 | 25 | 25 | 27 | 27 | 26 |
| Mozilla Firefox, с | 48 | 48 | 46 | 46 | 55 | 54 | 54 | 53 | 53 | 51 |

Стоить отметить, что все 20 запущенных подряд тестовых прогонов были успешно выполнены, что, в свою очередь, свидетельствует о высокой степени стабильности разработанных нами автоматизированных тестов и надежности при работе с библиотекой «kbv-testdriver».

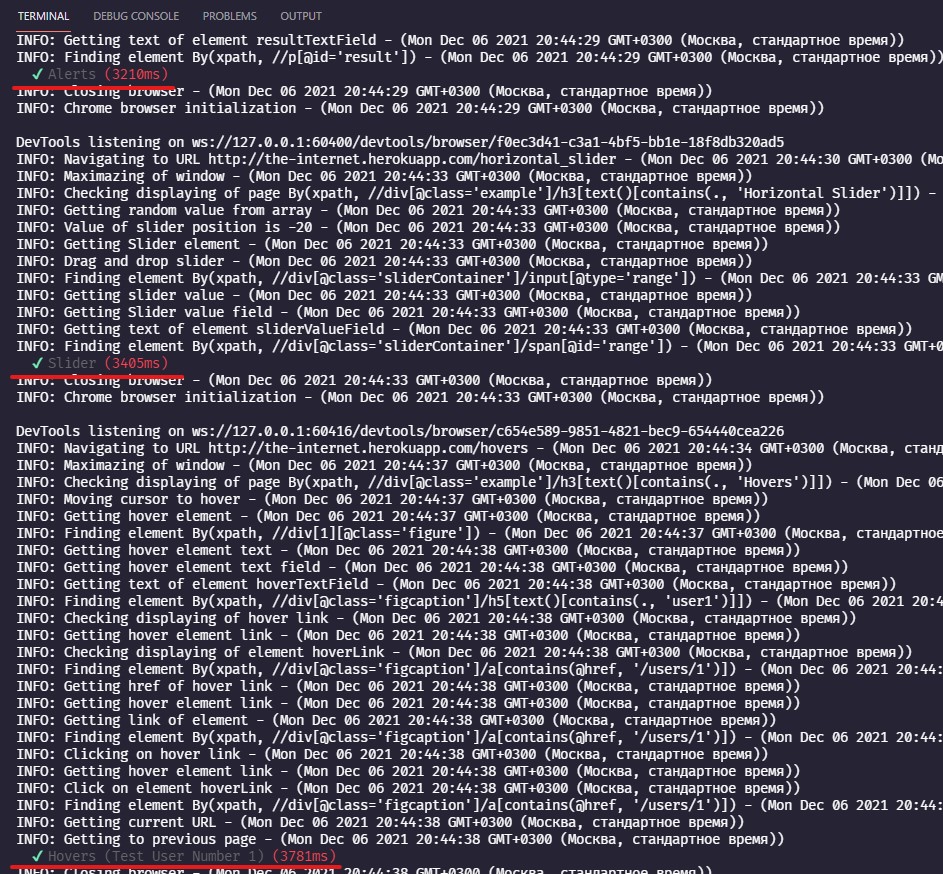


Рисунок 3.1 – Пример вывода в консоль времени, затрачиваемого на каждый тестовый сценарий по отдельности

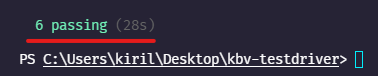


Рисунок 3.2 – Пример вывода в консоль времени, затрачиваемого на весь тестовый прогон в совокупности

На основании таблицы 3.1 построим график (рисунок 3.3), наглядно демонстрирующий временные показатели при выполнении разработанных нами тестовых сценариев для веб-сервиса «http://the-internet.herokuapp.com».

Из данного графика можно однозначно заключить, что работа с браузером Google Chrome в Selenium Webdriver оптимизирована на порядок лучше в сравнении с Mozilla Firefox. Помимо очевидно лучших временных показателей, запуски на Google Chrome также отличаются меньшей разбежкой во времени.

Для расчета разницы между самыми быстрым и самыми медленными тестовыми прогонами для обоих браузеров, выраженными в процентах, воспользуемся формулой 3.1:

| (a – b) / [(a + b) / 2] | \* 100 %, (3.1)

где a – первое число для расчетов, в нашем случае самый медленный тестовый запуск, с; b – самый быстрый тестовый запуск, с.

Рассчитаем разницу в процентах для тестовых запусков для браузера Google Chrome:

| (a – b) / [(a + b) / 2] | \* 100% = | (28 – 25) / [(28 + 25) / 2] | \* 100% = 11,32%

Теперь проделаем ту же самую операцию, только теперь для тестовых запусков на браузере Mozilla Firefox:

| (a – b) / [(a + b) / 2] | \* 100% = | (55 – 46) / [(55 + 46) / 2] | \* 100% = 17,82%

Как мы видим из расчетов, представленных выше, время запуска тестовых сценариев на браузере Google на 6,5% стабильнее, чем при запуске на браузере Mozilla Firefox.

Рисунок 3.3 – График, отображающий временные показатели при выполнении тестовых сценариев на обоих браузерах

Далее рассчитаем среднее время, необходимое для запуска рассматриваемых тестовых сценариев для каждого из браузеров.

Для этого воспользуемся формулой расчета среднего арифметического значения временных показателей (3.2):

Рассчитаем среднее арифметическое для браузера Google Chrome:

Рассчитаем среднее арифметическое для браузера Mozilla Firefox:

Теперь мы можем выразить разницу между средним временем тестового запуска на браузере Google Chrome и Mozilla Firefox. Для этого снова воспользуемся формулой 3.1:

| (a – b) / [(a + b) / 2] | \* 100 % = | (50,8 – 26,7) / [(50,8 + 26,7) / 2] | \* 100% = 31,09%

Благодаря вышеприведенным расчетам, мы можем сделать следующие выводы:

* разбежка во времени при запуске тестовых сценариев будет меньше при использовании браузера Google Chrome;
* тестовые запуски совершаются в среднем на 30 процентов быстрее на браузере Google Chrome.

Учитывая, что Google Chrome является самым популярным браузером на текущий момент времени, его использование является более предпочтительном, если в отдельно взятом случае нет необходимости тестирования на браузере Mozilla Firefox.

## **3.2.  Оценка ресурсных показателей программного средства**

Для оценки ресурсных показателей при запуске автоматизированных тестов, созданных на основе разработанной нами библиотеки «kbv-testdriver» необходимо провести оценку:

* нагрузки на центральный процессор персонального компьютера;
* потребления оперативной памяти персонального компьютера.

Для начала необходимо обозначить характеристики центрального процессора оперативной памяти нашего ПК. Подробную информацию касательно характеристик аппаратной части ПК можно почерпнуть в AIDA64 – это утилита для тестирования и идентификации компонентов персонального компьютера под управлением операционных систем Windows, предоставляющая детальные сведения об аппаратном и программном обеспечении.

Характеристики центрального процессора представлены на рисунке 3.4.

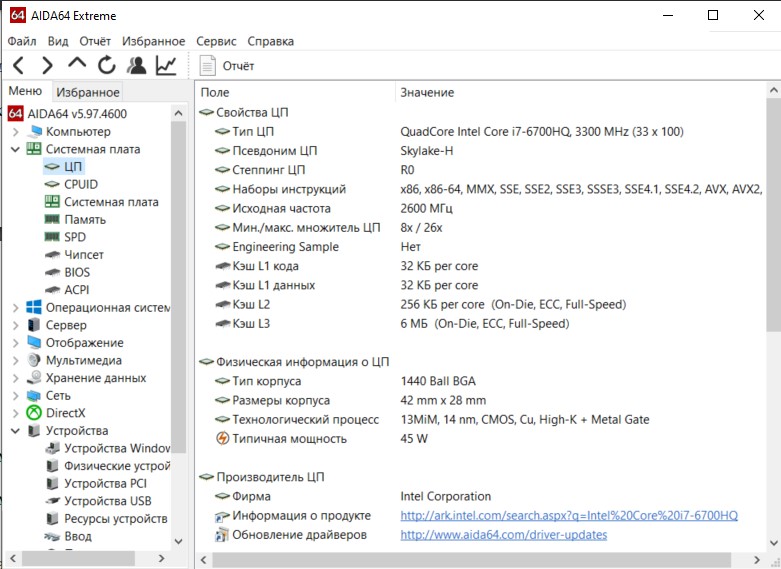


Рисунок 3.4 – Аппаратные характеристики центрального процессора в AIDA64

Аналогичным образом рассмотрим характеристики оперативной памяти нашего персонального компьютера на рисунке 3.5.

Для того чтобы проанализировать нагрузку на центральный процессор и потребление оперативной памяти при запуске автоматизированных тестов, воспользуемся встроенном в Windows Монитором ресурсов, который позволяет в реальном времени отслеживать состояние наших ЦП и ОП.

Приведем исходное состояние данных показателей (до запуска тестового прогона) на рисунках 3.6 и 3.7 соответственно.

Для применения в дальнейших расчетах, проследим за изменением показателей нагрузки на центральный процессор и потребления оперативной памяти, после чего отметим пиковые и минимальные нагрузки.

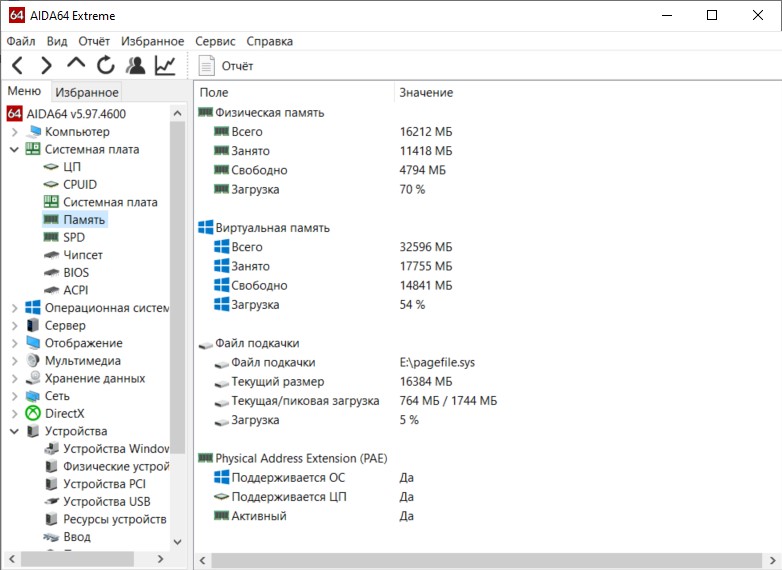


Рисунок 3.5 – Аппаратные характеристики оперативной памяти в AIDA64

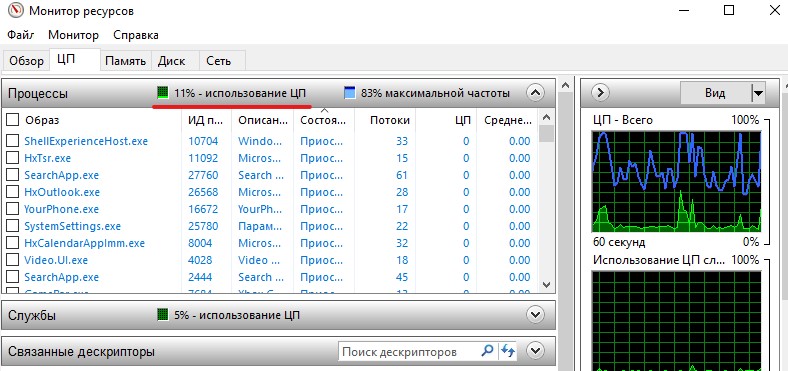


Рисунок 3.6 – Исходные показатели нагрузки на ЦП в Мониторе ресурсов

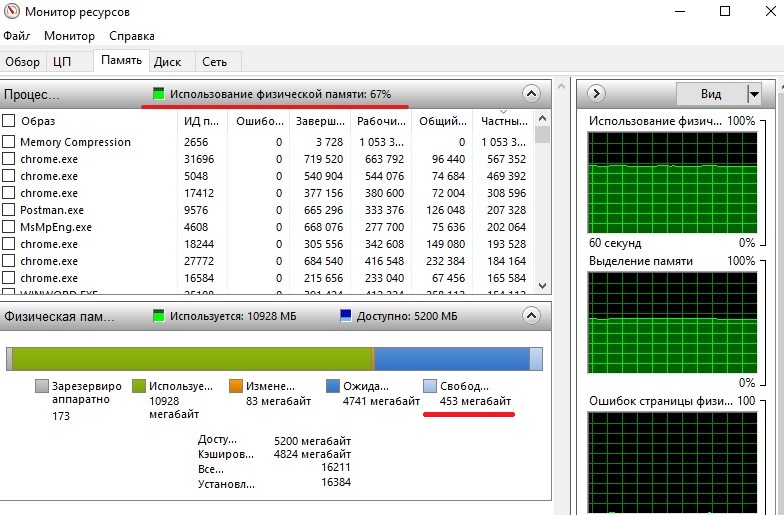


Рисунок 3.7 – Исходные показатели потребления ОП в Мониторе ресурсов

После проведенных наблюдений было установлены следующие показатели, отображенные в таблице 3.2:

Таблица 3.2 – Зарегистрированные показатели нагрузки на ЦП и потребления ОП в ходе тестового запуска

|  |  |  |
| --- | --- | --- |
|  | ЦП, % | ОП, МБ (количество свободной памяти) |
| Максимальное значение | 74 | 39 |
| Минимальное значение | 8 | 295 |

Воспользуемся формулой 3.3 для расчета среднего значения нагрузки на центральный процессор:

где – средняя нагрузка на процессор, %; – максимальная нагрузка на процессор, %; – минимальная нагрузка на процессор, %.

Исходя из результатов, представленных в таблице 3.2, рассчитаем искомое значение средней нагрузки на процессор:

Таким образом, средняя нагрузка на процессор, создаваемая тестовыми запусками на основе разработанной нами библиотеки «kbv-testdriver», составляет 41%, без учета погрешности в виде постоянно изменяющегося начального значения нагрузки на центральный процессор в размере ~ 5%.

Произведем расчеты для анализа потребления оперативной памяти, используя формулу 3.4:

где – среднее потребление оперативной памяти, МБ; – максимальное значение свободной оперативной памяти, МБ; – минимальное значение свободной оперативной памяти, МБ; – значение свободной оперативной памяти перед запуском тестового прогона.

Исходя из полученных ранее результатов, рассчитаем среднее значение потребления оперативной памяти используя формулу 3.4:

Таким образом, потребление оперативной памяти, создаваемое тестовыми запусками на основе разработанной нами библиотеки «kbv-testdriver», составляет 286 МБ.

В конечном итоге, основываясь на полученных данных, можно сделать вывод о том, что нагрузка, создаваемая программным средством, не является критичной. Если рассматривать отдельно нагрузку на центральный процессор и оперативную память: процессор в пиковые моменты нагружается достаточно сильно за счет взаимодействия с многими экземплярами объекта браузера; оперативная же память расходуется более чем умеренно.

# **ЭКСПЛУАТАЦИЯ ПРОГРАММНОГО СРЕДСТВА**

## **4.1 Ввод в эксплуатацию и обоснование минимальных технических требований к оборудованию**

В данном подразделе будут приведены основные сведения по работе с библиотекой «kbv-testdriver», применяемой для автоматизации тестирования, а именно будут перечислены конкретные действия, необходимые для полноценного и ввода в эксплуатацию и дальнейшего применения рядовыми пользователями.

Для полноценного использования нашей библиотеки нам потребуется следующее программное окружение:

* 1. Операционная система Windows 10 – как наиболее распространенная на текущий момент времени и имеющая широчайшую совместимость с огромным количеством актуального программного обеспечения.
  2. Node.js – программная платформа, основанная на движке V8 (транслирующем JavaScript в машинный код), превращающая JavaScript из узкоспециализированного языка в язык общего назначения. Node.js добавляет возможность JavaScript взаимодействовать с устройствами ввода-вывода через свой API, написанный на C++, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js совершенно необходим нам, поскольку языком программирования для создания нашей библиотеки был выбран JavaScript ES6 и выше. Node.js же выступает средой выполнения JavaScript.
  3. Selenium Webdriver 4.0 и выше, поскольку разработанная нами библиотека будет общаться с браузерами посредством данного фреймворка. Версия 4.0 была выбрана как последняя стабильная мажорная версия фреймворка.
  4. Браузер Google Chrome версии 93.0 и выше, для запуска разработанных нами автоматизированных тестов.
  5. Браузер Mozilla Firefox версии 91.0 и выше, поскольку разработанная нами библиотека «kbv-testdriver» имеет возможность запуска автоматизированных тестов в том числе на данном браузере.
  6. Драйвер chromedriver версии 93 и выше. Является необходимой прослойкой между фреймворком Selenium Webdriver и браузером Google Chrome для их корректного взаимодействия.
  7. Драйвер geckodriver версии 2 и выше. Является необходимой прослойкой между фреймворком Selenium Webdriver и браузером Mozilla Firefox для их корректного взаимодействия.
  8. Библиотека axios версии 0.21. Данная библиотека выступает популярным и лаконичным решением для реализации взаимодействия нашей библиотеки «kbv-testdriver» с API веб-сервисов и таким образом будет использована для отправки HTTP-запросов.
  9. Библиотека chai версии 4.2 и выше. Необходима для создания тестовых утверждений (assertions) в наших тестовых сценариях.
  10. Библиотека Mocha версии 9 и выше. Используется для запуска и имплементации тестов, написанных на языке программирования JavaScript.
  11. Также для сборки разработанного нами проекта и последующего запуска тестов рекомендуется использовать редактор кода Visual Studio Code (VS Code), поскольку он хорошо оптимизирован под операционную систему Windows, является достаточно легковесным и нетребовательным к ресурсам. В целом можно утверждать, что большинство JavaScript-разработчиков выбирают именно данный редактор кода.

Итак, мы подготовили всю вышеописанную программную среду. Для начала откроем папку нашего проекта в редакторе VS Code и пропишем в консоле следующую команду:

npm init

для того чтобы настроить и инициализировать новый npm-пакет (Node Package Manager) для нашего проекта. Во время выполнения данной команды, нам будут предложена череда вопросов в консоли, на которые необходимо дать ответы. После чего данная информация будет отображена в файле package.json.

Package.json представляет собой нечто вроде файла-манифеста для проекта. Он даёт в наше распоряжение множество разноплановых возможностей. Например, он представляет собой центральный репозиторий настроек для инструментальных средств, используемых в проекте. Кроме того, он является тем местом, куда npm записывает сведения об именах и версиях установленных пакетов.

После инициализации нашего npm-пакета, файл package.json будет выглядеть следующим образом (см.рисунок 4.1).

Далее необходимо установить все зависимости с другими npm-пакетами, с которыми будет взаимодействовать наша библиотека. Для этого воспользуемся следующей командой:

npm i --save-dev selenium-webdriver mocha chai axios lodash chromedriver geckodriver form-data fs image-downloader img-diff-js

Поскольку изначально библиотека «kbv-testdriver» разрабатывалась нами с расчетом на то, что вывод результатов тестовых запусков будет происходить не только в консоль, но и в отдельный файл, самое время реализовать такую возможность перед непосредственным вводом в эксплуатацию.

Для этого воспользуемся npm-пакетом под названием mochawesome. Это надстройка над используемом нами тестовом фреймворком Mocha, которая позволяет генерировать автоматические репорты (отчеты) по результатам наших тестовых запусков. Данные отчеты будут представлены в виде HTML-страниц, что позволяет открыть их в любом браузере.

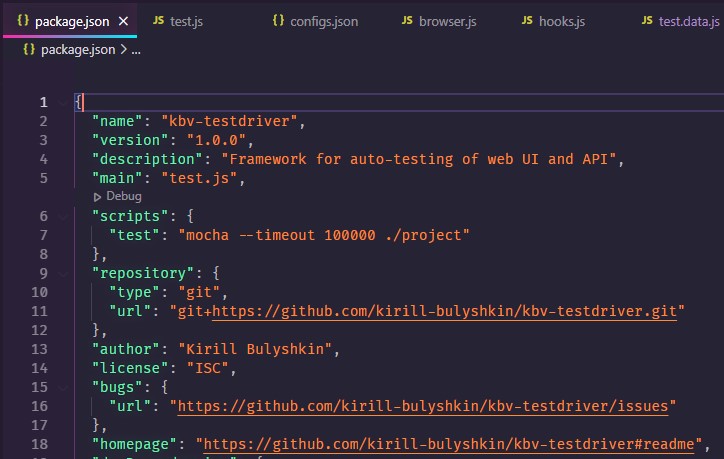


Рисунок 4.1 – Файл package.json после инициализации нашего npm-пакета

Итак, для подключения mochawesome воспользуемся следующей командой:

npm install --save-dev mochawesome

Чтобы убедиться в том, что генератор HTML-отчетов был подключен успешно, достаточно повторно запустить любой тестовый прогон. После чего следует обратить внимание на структуру нашего проекта, а именно, его директории. Здесь должна появиться новая директория с именем «mochawesome-report». Она будет иметь следующую структуру:

* assets (папка, содержащая все необходимые ассеты для построения итогового отчета);
* mochawesome.html (именно этот файл мы и будем открывать в дальнейшем для изучение отчетов по итогам тестовых запусков);
* mochawesome.json (JSON-файл, содержащий информацию на основании которой и будет строиться наш отчет).

Таким образом, после подключения пакета для генерации отчетов по результатам тестовых запусков, итоговая корневая структура нашего проекта будет выглядеть как представлено на рисунке 4.2.

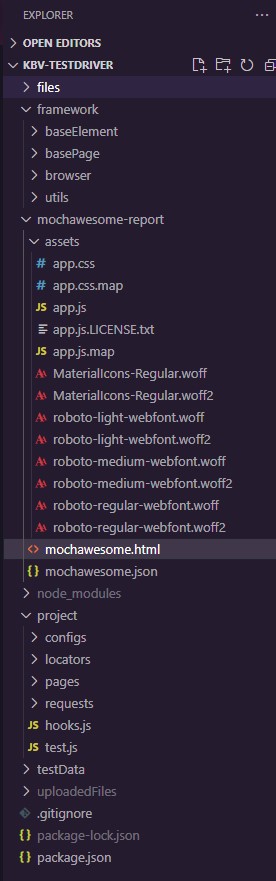


Рисунок 4.2 – Итоговая корневая структура проекта

Пример сгенерированного отчета рассмотрим позже в руководстве по использованию.

Поскольку мы полноценно завершили разработку нашего проекта, а также подключили все необходимые зависимые npm-пакеты, рассмотрим также итоговый вид файла package.json, на основе которого будет строится наш собственный npm-пакет в дальнейшем (см. рисунок 4.3).

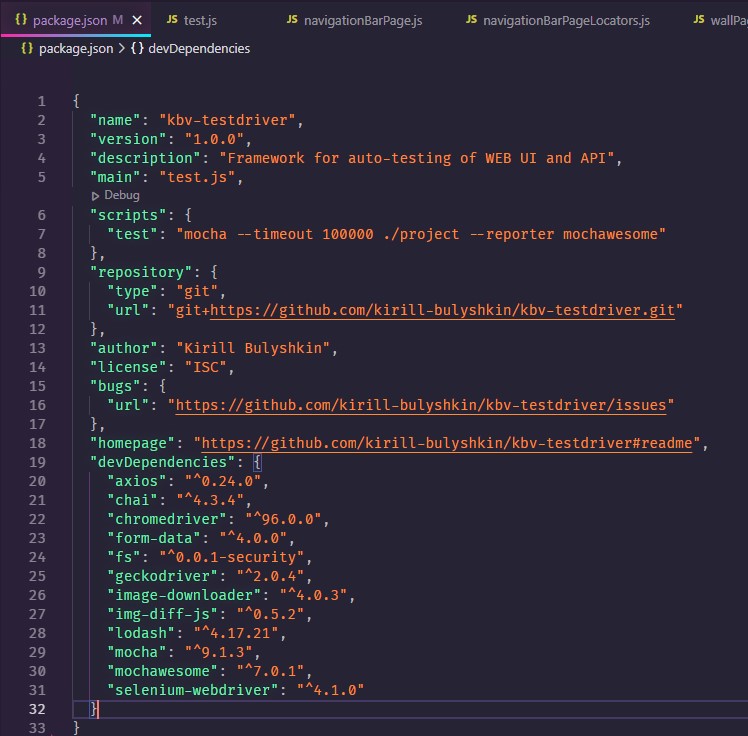


Рисунок 4.3 – Итоговый вид файла package.json после подключения всех зависимостей

Для удобства поиска и загрузки разработанной нами библиотеки «kbv-testdriver» остается лишь загрузить наш пакет в npm-репозиторий. Сделать это достаточно просто. Опишем пошагово необходимые действия:

1. Регистрируемся на ресурсе npmjs.com с помощью команды «npm adduser».
2. Далее необходимо авторизоваться. Для этого вводим команду «npm login», после чего указываем свой логин и пароль.
3. Вводим команду «npm publish».

Итого, с помощью трех простых команд наш npm-пакет должен оказаться в соответствующем репозитории. Для проверки этого залогинимся на веб-сайте npmjs.com с логином и паролем пользователя, от имени которого был загружен наш пакет. Поскольку все три шага были пройдены успешно, мы будем наблюдать следующую веб-страницу, представленную на рисунке 4.4.

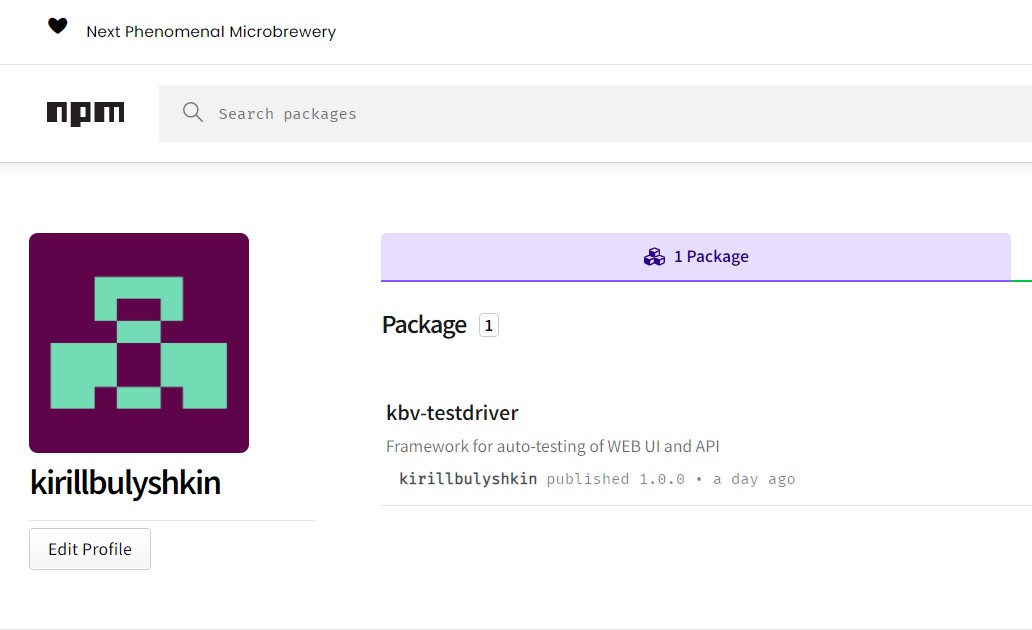


Рисунок 4.4 – Успешно загруженный npm-пакет библиотеки «kbv-testdriver» в общедоступный npm-репозиторий

Таким образом, после проделанных нами действий, библиотека «kbv-testdriver» станет общедоступна посредством ввода одной-единственной команды в консоль, поскольку наш репозиторий публичный.

Что касается технических требований, предъявляемых к оборудованию для работы с нашей библиотекой. В предыдущем разделе нами уже были рассмотрены нагрузка на центральный процессор нашей системы, а также потребление оперативной памяти. На основании этих данных можно сделать следующий вывод: для комфортного использований библиотеки будет достаточно аналогичного аппаратного обеспечения, которое указано в минимальных системных требованиях для использования операционной системы Windows 10, поскольку использование нашей библиотеки предполагается именно на данной ОС. Для этого перейдем на официальный веб-сайт корпорации Microsoft. Минимальные системные требования Windows 10 представлены на рисунке 4.5.

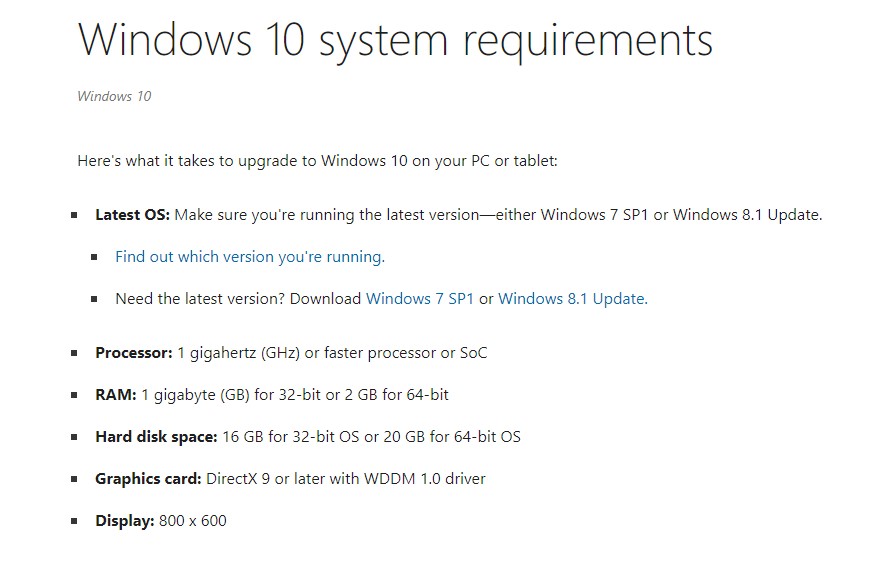


Рисунок 4.5 – Минимальные системные требования ОС Windows 10

Итак, системные требования представленные выше будут совпадать с минимальными техническими требованиями к оборудованию, необходимыми для работы разработанной нами библиотеки «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов.

## **Руководство по эксплуатации программным средством**

Руководство по эксплуатации представляет собой перечень команд, необходимых для начала использования библиотеки «kbv-testdriver» со всеми необходимыми пояснениями, а также поясняющие комментарии относительно интегрированного средства для визуализации результатов тестовых прогонов.

Для начала использования библиотеки «kbv-testdriver» достаточно ввести следующую команду в консоли нашего проекта:

npm i --save-dev kbv-testdriver

Таким образом будет установлен соответствующий npm-пакет из общедоступного репозитория. При успешном выполнении установки пользователь увидит следующее сообщение в консоли, как представлено на рисунке 4.6.

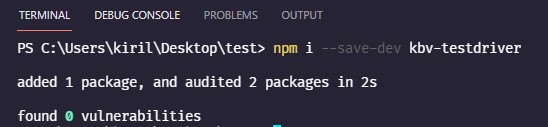


Рисунок 4.6 – Вывод консоли при успешной установке npm-пакета «kbv-testdriver»

После данного действия пользователю станут доступны все методы, реализованные в рамках библиотеки. Для подробного изучения всех реализованных методов и классов пользователь может перейти по следующему пути в корневом каталоге своего проекта по тестированию:

node\_modules –> kbv-testdriver –> framework

где и будут находиться все классы. Например, таким образом были найдены методы класса baseElement, для работы с базовыми элементами веб-страниц, как представлено на рисунке 4.7.

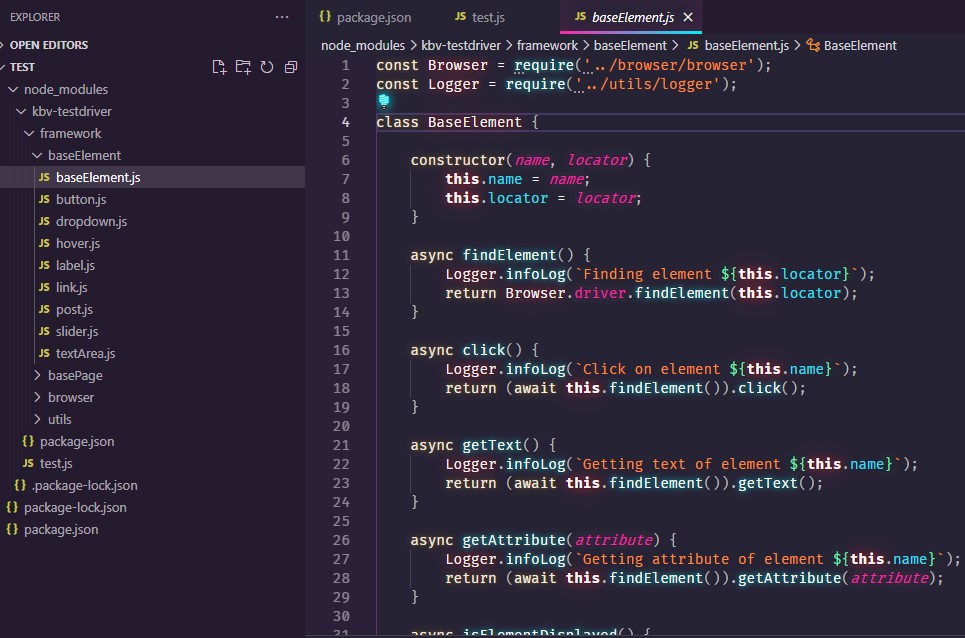


Рисунок 4.7 – Доступ к методам класса BaseElement

Следующим шагом будет вызов библиотеки «kbv-testdriver» при написании пользователем своих собственных тестовых сценариев. Для этого воспользуемся синтаксисом CommonJS и функцией require().

Node.js следует модульной системе CommonJS, а встроенная require функция – самый простой способ включить модули, которые существуют в отдельных файлах. Основная функциональность require заключается в том, что он читает файл JavaScript, выполняет файл, а затем переходит к возврату exports объекта.

Для удобного вызова всех методов, относящихся к нашей библиотеке, сохраним данный вызов в константу «kbv». К примеру, запись обращения к методам базового класса Browser, с сохранением в переменную, будет выглядеть следующим образом:

const kbv = require('kbv-testdriver/framework/browser/browser');

Далее на примере все того же обращения к классу Browser, разработанного в рамках библиотеки «kbv-testdriver», рассмотрим вызов методов на рисунке 4.8.

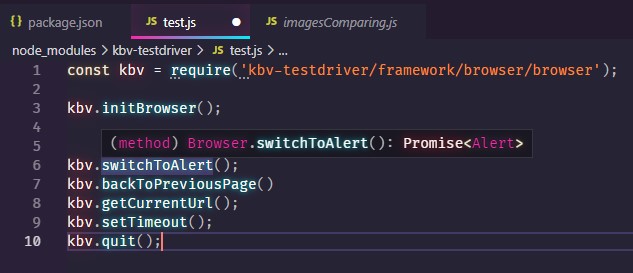


Рисунок 4.8 – Вызов методов библиотеки «kbv-testdriver»

Как видно из рисунка 4.8, редактор кода VS Code выводит подсказку к какому классу относится тот или иной метод.

В библиотеке предусмотрена возможность взаимодействия с браузерами Google Chrome и Mozilla Firefox, а также выбора их языка: русский или английский. Для изменения этих параметров перейдем к файлу config.json. Как видно из рисунка 4.9, в конфигурационном файле представлено два параметра:

* browserLanguage;
* browserName.

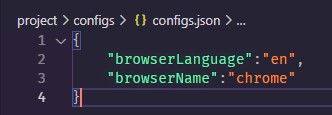


Рисунок 4.9 – Конфигурационный файл config.json

В таблице 4.1 представлены возможные значения конфигурируемых параметров.

Таблица 4.1 – Возможные значения параметров в файле config.json

|  |  |  |
| --- | --- | --- |
| **Ключ** | **Возможные значения** | |
| browserLanguage | en | ru |
| browserName | chrome | firefox |

Соответственно, существует четыре возможных конфигурации запуска браузера во время тестовых прогонов.

При создании тестов пользователю рекомендуется разделять файлы с непосредственно тестами и тестовыми данными. В качестве примера приведем файл test.data.js, представленный на рисунке 4.10.

В библиотеке «kbv-testdriver» реализована возможность генерации отчетов по результатам тестовых запусков. Отчеты будут хранится в следующей директории:

mochawesome-report –> mochawesome.html

При открытии сгенерированного отчета, пользователь может изменять следующие параметры, как указано на рисунке 4.11.

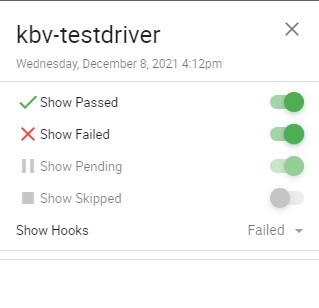


Рисунок 4.11 – Конфигурирование сгенерированного отчета

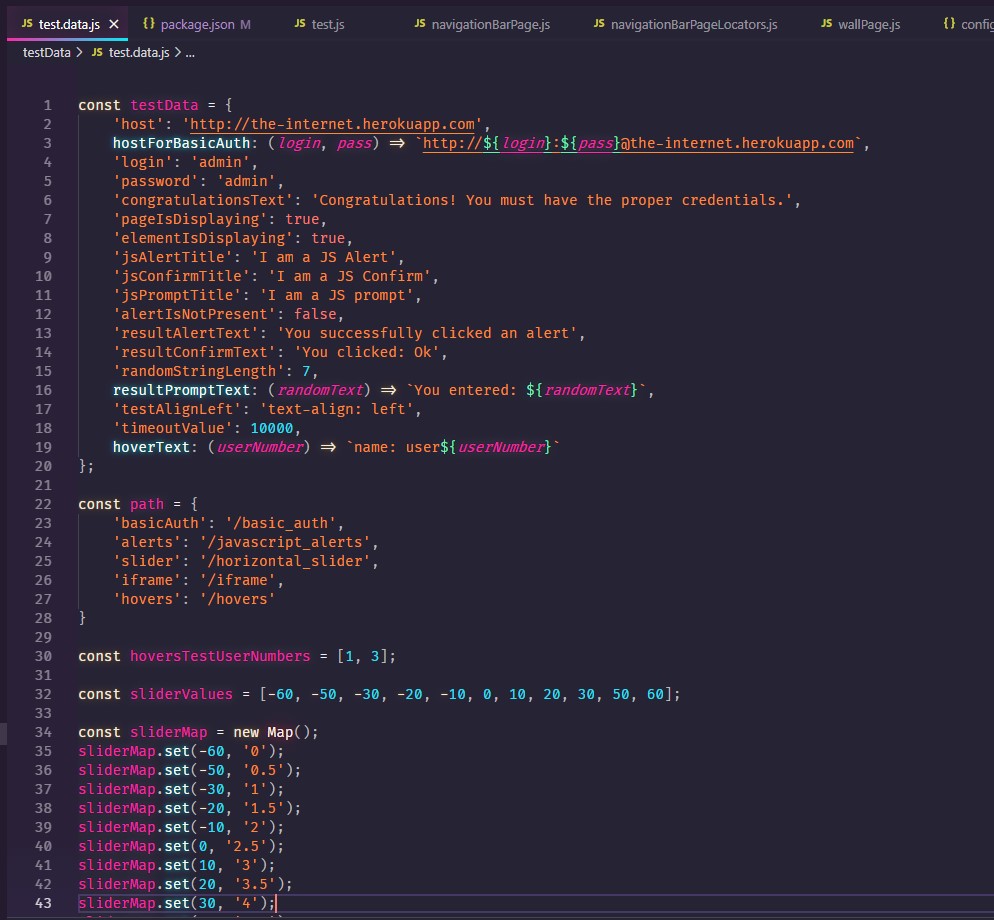


Рисунок 4.10 – Пример файла с тестовыми данными «test.data.js»

Как видно из рисунка 4.11, пользователь может выбрать какие именно результаты тестов отображать (все результаты, только положительные результаты, только отрицательные результаты, отложенные тесты, пропущенные тесты и их различные комбинации.

Пример сгенерированного отчета по результатам запущенных автоматизированных тестов представлен на рисунке 4.12.

Как видно из рисунка 4.12, пользователю предоставлена исчерпывающая информация о результатах тестового запуска, а именно:

* общее время, затраченное на тестовый запуск;
* общее количество запущенных тестов;
* количество тестов с позитивным результатом;
* количество тестов с негативным результатом;
* время, затраченное на каждый из тестов по отдельности;
* полная расшифровка каждого из тестов;
* ошибка, появившаяся в консоли при негативном результате теста.

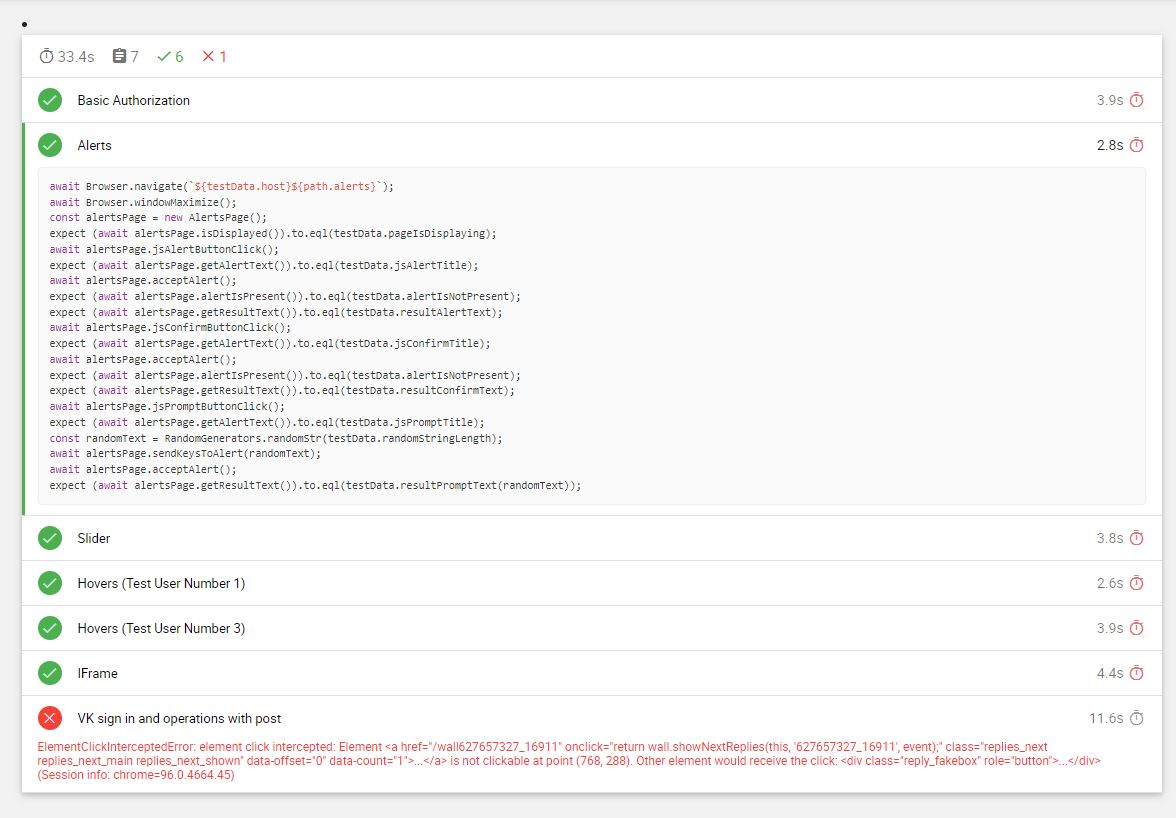


Рисунок 4.12 – Пример сгенерированного отчета

Таким образом, руководствуясь представленной выше информацией пользователь может полноценно эксплуатировать библиотеку «kbv-testdriver» на базе Selenium Webdriver для автоматизации тестирования UI и API веб-сайтов.