Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра управления инновациями (УИ)

К ЗАЩИТЕ ДОПУСТИТЬ
Заведующий кафедрой УИ,
к.ф-м.н., доцент
_____ Г.Н. Нариманова
«___» _____ 20__ г.

ВНЕДРЕНИЕ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ В ПРОЦЕСС РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ

Магистерская диссертация

по направлению подготовки 15.04.06 «Мехатроника и робототехника»

Выполнил:
Студент гр. 038-М
_____ В.А. Никонов
«___» _____ 2020 г.

Руководитель:
Доцент кафедры УИ,
к.ф.-м.н., доцент
_____ М.Е. Антипин
«___» _____ 2020 г.

Томск 2020

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра управления инновациями (УИ)

УТВЕРЖДАЮ
Заведующий кафедрой УИ,
к.ф.-м.н., доцент
_____ Г.Н. Нариманова
«___» _____ 2020г.

ЗАДАНИЕ

на выпускную квалификационную работу (ВКР)

студенту гр.038-М факультета инновационных технологий

Никонову Владиславу Алексеевичу

1. Тема ВКР: Внедрение автоматизированного тестирования в процесс разработки веб-приложений.

2. Цель ВКР: Интеграция автоматизированного тестирования в процесс коммерческой разработки веб-приложений.

3. Задачи ВКР:

1. Исследование предметной области, изучение специальной литературы.
2. Обзор и анализ инструментов автоматизации тестирования.
3. Выбор стека технологий для автоматизации тестирования веб-приложений.
4. Разработка тестового фреймворка на основе выбранного стека технологий для автоматизации тестирования веб-приложений.
5. Внедрение инструмента непрерывной интеграции для автоматизации процессов тестирования.
6. Разработка программного обеспечения для конфигурирования и разворачивания разработанного тестового фреймворка.

7. Разработка регламента процессов автоматизации тестирования.

4. Срок сдачи ВКР в ГЭК «___» _____ 2020 г.

5. Технические требования: Текст ВКР оформляется в соответствии с требованиями стандарта ОС ТУСУР 01-2013, в печатном виде с использованием персонального компьютера на бумаге формата А4 и сдается на кафедру в сброшюрованном виде.

6. Дата выдачи задания: «___» _____ 2020 г.

Руководитель:

Доцент кафедры УИ, к.ф.-м.н., доцент _____ М.Е. Антипин

Задание принял к исполнению:

«____» _____ 2020 г. студент _____ В.А. Никонов

РЕФЕРАТ

Выпускная квалификационная работа объемом 115 страниц состоит из введения, 4 глав, заключения, 44 рисунков, 4 таблицы, 46 литературных источников и 4 приложений.

Ключевые слова: фреймворк для автоматизации тестирования веб-приложений, ПО для автоматизации разворачивания тестового фреймворка, Jest, Puppeteer, TestNG, Selenium, автоматизация тестирования UI, автоматизация тестирования API, непрерывная интеграция с Jenkins.

Работа выполнена на базе компании ООО «Красная рамка» (г. Томск). Компания специализируется на проектировании и разработке веб-проектов со сложным веб-интерфейсом.

Объектом исследования выпускной квалификационной работы является процесс автоматизации тестирования веб-приложений. Предметом исследования – возможность внедрения автоматизации тестирования в процессы разработки веб-приложений.

Цель настоящей работы заключается в организации и интеграции автоматизированного тестирования в процесс разработки веб-приложений.

В процессе работы были выполнены следующие основные задачи: исследована предметная область, выбран стек технологий для автоматизации тестирования веб-приложений, разработан тестовый фреймворк для автоматизации тестирования веб-приложений, внедрен инструмент непрерывной интеграции для автоматизации процессов тестирования, разработано программное обеспечение для конфигурирования и разворачивания тестового фреймворка, а также разработан регламент процессов автоматизации тестирования.

ABSTRACT

The graduation work includes 115 pages and consists of the introduction, 4 chapters and conclusion. Also, this paper includes 44 pictures, 4 tables, 46 literature sources and 4 applications.

Keywords: framework for automated testing of web applications, software for test framework automatic deployment, Jest, Puppeteer, TestNG, Selenium, UI testing automation, API testing automation, continuous integration with Jenkins.

The work was performed on the basis of the company "Red frame" (Tomsk). The company specializes in designing and developing web projects with a complex web interface.

The object of research of the final qualification work is the process of automating testing of web applications. The subject of the research is the possibility of implementing test automation in web application development processes.

The purpose of this work is to organize and integrate automated testing in the process of web applications developing.

During the work, the following main tasks were performed: the subject area was investigated, a stack of technologies for automating web application testing was selected, a test framework for automating web application testing was developed, a continuous integration tool for automating testing processes was implemented, software for configuring and deploying the test framework was developed, and the regulations for testing automation processes were developed.

Оглавление

Введение

Большинство программных продуктов, выпускаемых сегодня, являются веб-ориентированными приложениями, рассчитанными на работу в интернет-браузере. Эффективность тестирования подобных приложений отличается в различных компаниях и организациях. В эпоху высокой интерактивности и взаимодействия в процессе разработки программ, когда многие организации используют методологию Agile в той или иной форме, автоматизация тестирования часто становится необходимостью.

Под автоматизацией тестирования подразумевается использование инструментов для того, чтобы многократно выполнять повторяющиеся тесты для тестируемого приложения. Регрессионное тестирование является наиболее типичным примером применения этого подхода.

Автоматизированный тест – это скрипт или программа, которая имитирует взаимодействия пользователя с приложением для нахождения дефектов в приложении. Данное определение справедливо, пожалуй, только для GUI тестирования.

Не всегда полезно автоматизировать тесты. Иногда ручное тестирование может оказаться более подходящим. Например, если графический интерфейс приложения сильно изменится в ближайшем будущем, автоматизированные тесты придется переписывать. К тому же, иногда попросту не хватает времени на автоматизацию. В краткосрочной перспективе ручное тестирование может быть более эффективно. Если приложение должно быть выпущено в очень сжатые сроки, готовых автоматизированных тестов нет, но протестировать в срок необходимо, то ручное тестирование является лучшим решением.

Автоматизированное тестирование позволяет:

1. Проводить чаще регрессионное тестирование.
2. Быстро предоставлять разработчикам отчет о состоянии продукта.
3. Получить потенциально бесконечное число прогонов тестов.

4. Обеспечить поддержку Agile экстремальными методами разработки.

5. Сохранять строгую документацию тестов.

6. Обнаружить ошибки, которые были пропущены на стадии ручного тестирования.

7. Позволяет существенно повысить надёжность кода и безопасность приложения.

Все вышеперечисленное, обеспечивает преимущества, которые могут повысить эффективность работы отдела тестирования в долгосрочной перспективе [1]. Поэтому разработка крупных и сложных систем непременно требуют привлечения специалистов в области автоматизированного тестирования, что показывает ***актуальность настоящей работы***.

В связи с этим ***цель настоящей работы*** заключается в организации и интеграции автоматизированного тестирования в процесс разработки веб-приложений.

Для достижения поставленной цели необходимо выполнить следующие ***задачи***:

1. Исследование предметной области, изучение специальной литературы.

2. Обзор и анализ инструментов автоматизации тестирования.

3. Выбор стека технологий для автоматизации тестирования веб-приложений.

4. Разработка тестового фреймворка на основе выбранного стека технологий для автоматизации тестирования веб-приложений.

5. Внедрение инструмента непрерывной интеграции для автоматизации процессов тестирования.

6. Разработка программного обеспечения для конфигурирования и разворачивания разработанного тестового фреймворка.

7. Разработка регламента процессов автоматизации тестирования.

Объектом исследования выпускной квалификационной работы является процесс автоматизации тестирования веб-приложений. ***Предметом исследования*** – возможность внедрения автоматизации тестирования в процессы разработки веб-приложений.

На защиту выносятся следующие ***основные положения***:

1. Разработанный тестовый фреймворк для автоматизации тестирования веб-приложений решает все поставленные задачи автоматизации тестирования программного и графического веб-интерфейсов в процессе разработки веб-приложений.

2. Разработанные программное обеспечение и документация обеспечивают быструю интеграцию автоматизированного тестирования в процесс разработки веб-приложений.

В работе использованы методы научного познания: изучение, анализ, исследование и сравнение.

В работе впервые представлено программное обеспечение, позволяющее автоматизировать процесс разворачивания проекта с разработанным тестовым фреймворком, а также конфигурировать его, что определяет ***научную новизну*** настоящей работы.

При этом, данное программное обеспечение позволяет облегчить и ускорить процессы конфигурирования и разворачивания проекта, что стимулирует проектирование и разработку аналогичного программного обеспечения, что несёт в себе ***научную ценность*** данного исследования.

Разработанное программное обеспечение для разворачивания и конфигурирования проекта, а также тестовая документация, облегчит и ускорит процесс разработки автотестов, а внедренная в процесс разработки автоматизация тестирования, безусловно, положительно скажется на времени тестирования веб-проектов и скорости оповещения о возможных дефектах. Все это позволит сфокусироваться на наиболее важных бизнес задачах разработки,

не отвлекая внимания специалистов по качеству на регрессионное тестирование. Это указывает на ***практическую ценность*** настоящей работы.

Работа выполнена на базе компании ООО «Красная рамка», в рамках работы по внедрению автоматизации тестирования в процесс коммерческой разработки веб-приложений, а результаты настоящего исследования используются в практической деятельности предприятия, что подтверждается актом о внедрении, представленным в приложении Г.

По теме научной работы были опубликованы следующие статьи:

1. Никонов В. А. Стратегии автоматизации тестирования // Материалы докладов XVI Международной школы-конференции студентов, аспирантов, молодых учёных «Инноватика-2020», Томск, 23-25.04.2020 г. Секция «Управление качеством»

2. Никонов В. А. Разработка через тестирование. TDD // Материалы докладов XXV Международной научно-технической конференции студентов, аспирантов и молодых учёных «Научная сессия ТУСУР – 2020», Томск, 25-27.05.2020 г. Секция «Молодежные инновационные научные и научно-технические проекты»

1 Обзор процесса автоматизации тестирования

1.1 Развертывание процесса автоматизации тестирования

Автоматизированное тестирование – процесс достаточно сложный как с точки зрения написания кода, так и с точки зрения методологии и организации процессов в команде. К процессу автоматизации нужно подходить обдуманно. В процессе обсуждения важно понять, что стейкхолдеры и технические специалисты ожидают от автоматизации тестирования. Преимущества, риски и издержки должны быть определены заранее. Грамотная стратегия принесет значительные преимущества [2].

1.1.1 Стратегии автоматизации тестирования

Существует несколько общепринятых используемых вариантов стратегии автоматизации тестирования. От выбора конкретной стратегии зависит порядок и интенсивность тех или иных работ по автоматизации. Выбор стратегии – задача не самая важная, но начать процесс развёртывания автоматизации лучше всего именно с неё. Рассмотрим три варианта стратегий, характерных для самого начала развёртывания автоматизации [3].

1) Стратегия «Let's try»

Применяется в том случае, когда автоматизации тестирования, ни на проекте, ни в компании никогда не было, и планируется осторожный старт с умеренным выделением ресурсов.

Стратегию имеет смысл применять в случае, когда:

- Отсутствуют точные цели автоматизации. Например, покрыть 40% кода конкретного модуля к определённой дате, уменьшение расходов на ручное тестирование и т.д.
- Автоматизация тестирования на проекте ранее не применялась.

- У тестировщиков отсутствует или очень мало опыта автоматизации тестирования.
- Выделенные ресурсы умеренные или низкие.

Требования к стратегии:

- Больше внимания уделять подготовительным этапам тестирования таким как: составление тест-планов, тест-кейсов, чек-листов и т.д.
- Больше внимания уделять инструментам, которые можно использовать как помощь в ручном тестировании.
- Больше экспериментировать с технологиями и методологиями автоматизации тестирования. Никто не ждёт срочных результатов и можно экспериментировать.
- Работать с проектом, начиная с верхнего уровня, в начале, не углубляясь в автоматизацию конкретных модулей.

2) Стратегия «Here the target»

Особенностью стратегии служит ориентирование на конкретный результат. Выбирается или определяется цель нового этапа автоматизации тестирования и задачи ориентируются на достижения данного результата.

Стратегию имеет смысл применять в случаях:

- Когда на проекте уже проведена предварительная работа, имеется какой-то бэкграунд в виде тест-планов, тест-кейсов, оптимально – автотестов предыдущего этапа автоматизации.
- Есть конкретная цель. Не глобальная – 80% автотестов за полгода, а скорее 50% авто-тестов конкретного модуля за месяц.
- Для выполнения конкретной цели выбраны конкретные инструменты, оптимально если у специалистов имеется некий технический бэкграунд по работе с инструментами.

Требования к стратегии:

- Поступательная стратегия, чем-то напоминает Agile методологии разработки. Движение вперёд этапами. Покрытие авто-тестами модуля за

модулем, до полного выполнения мета задач вида – 80% тестового покрытия за полгода.

- На каждый этап выставляется новая цель, продолжающая последнюю выполненную цель, но не обязательно, и выбираются инструменты для реализация данной цели.
- Глубокая фокусировка на конкретной цели, написание тест-кейсов, авто-тестов, не для всего проекта, а исключительно под конкретную задачу.

3) Стратегия «Operation Uranum»

Данная стратегия, это постоянная и методичная работа над автоматизацией тестирования по выставляемым раз в 2-3 недели приоритетам. Оптимально, наличие постоянно работающего, именно над автоматизацией, человека, не отвлекающегося на сторонние задачи. Стратегию имеет смысл применять в случае, когда:

- Отсутствуют конкретные цели, есть лишь общее пожелание «чтоб всё было хорошо». Если «Here the target» напоминает по принципу работы Agile, то данная стратегия близка по духу к методологии Waterfall.
- Есть ресурс в виде хотя бы одного постоянно действующего на проекте человека, целенаправленно занятого задачей автоматизации.
- Нет чётко выраженных целей автоматизации тестирования, однако есть пожелания и приоритеты, которые можно выставить на достаточно продолжительный период времени.

Требования к стратегии:

- Постоянная и методичная работа с учётом выставленных приоритетов.
- В начале нужен упор на базовую часть, поскольку так или иначе в рамках данной стратегии автоматизируется весь проект, без полной фокусировки на конкретных модулях.

Обдумывая общую логику и стратегию автоматизации, было принято решение воспользоваться стратегией «Let's try» – подготовить базу для дальнейшей работы, не особо глубоко погружаясь в написание кода самих авто-

тестов. По завершению этого этапа у нас будет готовый базис для дальнейших работ. И дальше действовать в соответствии с используемой в компании методологии разработки.

1.1.2 Создание тест-плана

После выбора стратегии автоматизации тестирования следующим важным пунктом будет создание тест плана. Тест-план должен быть согласован с разработчиками и менеджерами продукта, поскольку ошибки на этапе создания тест плана могут проявиться в поздних этапах разработки [4].

Тест-план нужно составлять для любого относительно крупного проекта, на котором работают тестировщики.

Тест-план состоит из следующих пунктов:

1. Объект тестирования.

Краткое описание проекта, основные характеристики: web/desktop, ui, iOs, Android, работает в конкретных браузерах/ОС и т.д.

2. Состав проекта.

Логически разбитый список отдельных, изолированных друг от друга компонентов и модулей проекта с возможной декомпозицией, но не углубляясь до мелочей, а также функций вне крупных модулей.

В каждом модуле необходимо перечислить набор доступных функций, не вдаваясь в детали. От данного списка будет отталкиваться менеджер и тест-дизайнер при определении задач по тестированию и автоматизации на новый спринт.

3. Стратегия тестирования и планируемые виды тестирования на проекте.

В случае с автоматизацией обычно используется только один вид тестирования – **регрессионное**. Автоматические регрессионные тесты – основа стратегии автоматизации тестирования [5].

«Дымовой» пакет регрессионных тестов нужен для проверки того, что приложение загружается и запускается. В него также входят несколько ключевых сценариев, позволяющих убедиться, что приложение ещё работает.

Цель этого пакета тестов в том, чтобы отловить наиболее очевидные проблемы, например, то, что приложение не загружается или не запускается основной поток взаимодействия пользователя с приложением. Поэтому «дымовые» тесты не должны продолжаться больше 5 минут, их цель – сообщить, что не работает что-то ключевое.

Такие тесты запускаются при каждом развёртывании приложения и могут содержать как API, так и GUI-тесты.

Функциональный пакет регрессионных тестов нужен для более детальной проверки работы приложения, чем это позволяют «дымовые» тесты.

Необходимо создать несколько функциональных пакетов для различных целей. Если есть несколько команд, работающих над различными разделами приложения, то в идеале нужны регрессионные пакеты, покрывающие область работы каждой команды.

Эти пакеты должны запускаться в различных окружениях по мере необходимости и проверять, что поведение приложения остаётся неизменным вне зависимости от окружения. Такие тесты запускаются несколько раз в день и должны продолжаться не дольше 15–30 минут.

Поскольку эти тесты более детализированы и занимают больше времени, важно выносить большую часть функциональных тестов на уровень API, где тестирование проходит быстрее. Это нужно для того, чтобы не выходить за временные рамки в 15–30 минут.

Полный пакет регрессионных тестов позволяет протестировать приложение как целое. Цель этого пакета тестов – проверить, что различные части приложения, которые обращаются к различным базам данных и другим приложениям, работают корректно.

Этот пакет тестов не предназначен для проверки всех возможностей приложения, поскольку их работа уже проверена функциональными регрессионными пакетами. В любом случае, эти тесты более «лёгкие» и проверяют переходы из одного состояния в другое или несколько наиболее популярных сценариев или путей пользователя.

Такие тесты в основном проводятся с использованием GUI, поскольку они проверяют, как пользователь будет взаимодействовать с системой. Время, которое на них затрачивается, может варьироваться в зависимости от приложения, но обычно такие тесты запускаются один раз за день или за ночь.

4. Критерии завершения тестирования

Необходимо кратко описать, когда тестирование считается завершенным в рамках данного релиза. Если есть какие-то специфические критерии – описать их.

1.1.3 Определение первичных задач

После выбора стратегии и составления тест плана стоит выбрать набор задач, с которых начнём автоматизацию тестирования. Наиболее частые типы задач, которые ставятся перед автоматизацией [6]:

- Полная автоматизация приёмочного тестирования (Smoke тестирование) – вид тестирования, проводящийся первым после получения билда отделом тестирования. В рамках smoke-тестирования проверяется та функциональность, который должен работать всегда и в любых условиях, и, если он не работает – по соглашению с разработчиками считается, что билд не может быть принят к тестированию.
- Максимизация количества найденных дефектов. В этом случае надо отобрать сначала те модули или аспекты функциональности системы, которые наиболее часто подвержены изменениям логики работы, а затем выбрать наиболее рутинные тесты, то есть тесты, где одни и те же шаги с небольшими вариациями выполняются на большом объеме данных.

- Минимизация «человеческого фактора» при ручном тестировании. Отбираются наиболее рутинные тесты, требующие наибольшей внимательности от тестировщика, при этом, легко автоматизируемые.
- Нахождение большинства падений системы. Тут можно применять «случайные» тесты.

В самом начале развёртывания автоматизации необходимо поставить задачу автоматизации приёмочного тестирования, как наименее трудоёмкую. При этом решение задачи позволит запускать приёмочное тестирование уже на следующем принятом билде [7].

Основным критерием smoke-тестов должна быть их относительная простота и одновременно обязательная проверка критически важной функциональности проекта.

Также подразумевается, что smoke-тесты будут позитивными – проверяющими корректное поведение системы, в то время как негативные – проверяют, будет ли система работать некорректно, чтобы не тратить время на лишние проверки.

Составляя список первичных задач к автоматизации, логично будет первыми описать и автоматизировать smoke-тесты. В дальнейшем их можно будет включить в проект и запускать при каждой сборке. По причине их ограниченного количества выполнение данных тестов не должно особо затормозить сборку, однако каждый раз, можно будет точно знать, работают ли критически важные функции.

1.1.4 Написание тест кейсов по выбранным задачам

В отношении тест-кейсов принято делить процесс тестирования на две части: тестирование по готовым сценариям (тест-кейсам) и исследовательское тестирование.

В отношении исследовательского тестирования всё достаточно понятно, оно существует в двух вариация, либо исследование новой функциональности

без особой предварительной подготовки, либо в виде банального monkey-тестирования. Тестирование по сценариям подразумевает, что было затрачено время и по функциональности проекта созданы тестовые сценарии, покрывающие максимально большой его объём.

Наиболее разумным является сочетание подходов, при котором новые функции и модули тестируется в исследовательском стиле, стараясь проверить возможные и маловероятные сценарии, и по завершении тестирования создаются тест-кейсы, в дальнейшем используемые для регрессионного тестирования.

Три варианта дальнейшего использования тест-кейсов, кроме очевидного:

- Сформировать из тест-кейсов чек-листы по модулям проекта, так проверка ускорится, но основные проблемные места будут проверены.
- Пришедший на проект тестировщик может изучать проект с точки зрения тест-кейсов, поскольку они захватывают многие не очевидные моменты работы приложения.
- Дальнейшее использование как базис авто-тестов. Если при развёртывании автоматизации тестирования применяется системный подход – то написание и дальнейшее использование тест-кейсов является совершенно логичным – ведь тест-кейс, это уже готовый сценарий авто-теста.

Для дальнейшей автоматизации тестирования нужно написать тест-кейсы по выставленным заранее приоритетным задачам. Они послужат одновременно началом создания нормального регрессионного тестирования и послужит базой для дальнейших авто-тестов.

1.1.5 Отбор тестов для автоматизации

К текущему этапу уже сформирован тест план и часть функциональности модулей описан как тест-кейсы. Следующей задачей будет выбор нужных тестов из имеющегося многообразия тест-кейсов.

На данном этапе, есть только тест-кейсы, подготовленные для smoke-тестирования, однако спустя несколько итераций развития тест-кейсов в проекте станет существенно больше, и далеко не все из них есть смысл автоматизировать [8].

1.1.6 Проектирование тестов для автоматизации

Тест-кейсы выбранные для автоматизации скорее всего будет нужно дописать и поправить, поскольку тест-кейсы, как правило, пишутся на простом человеческом языке, в то время как тест-кейсы для дальнейшей автоматизации должны быть дополнены необходимыми техническими подробностями, для простоты их перевода в код.

Правильно написанный тест-кейс, предназначенный для автоматизации, будет куда более похож на миниатюрное техническое задание по разработке небольшой программы, чем на описание корректного поведения тестируемого приложения, понятное человеку [9].

Собственно, далее следует техническая часть – разработка и запуск авто-тестов. Выполнив все описанные выше шаги, настроив, написав и запустив авто-тесты выполнена важная часть работы – разворачивание автоматизации тестирования на проекте. Далее необходимо решить множество задач по созданию тест-кейсов, настройке инструментария непрерывной интеграции, формированию подходящих и информативных отчётов.

1.2 Рабочее окружение и стек применяемых технологий

Определяющий фактор для успешного применения автоматизации тестирования программного обеспечения - выбор и использование правильного набора средств автоматизации тестирования. Это сложная задача, особенно для тех, кто раньше не сталкивался с автоматизацией тестирования, поскольку на рынке существует очень много инструментов, каждый из которых имеет разные сильные и слабые стороны. Нет инструмента, который бы соответствовал всем

требованиям автоматизированного тестирования. Это затрудняет поиск подходящего решения [10].

1.2.1 Обзор и анализ инструментов автоматизации тестирования веб-приложений

По данным сервиса npmtrends.com [11], который позволяет сравнить количество скачиваний пакетов с течением времени, на ноябрь 2019 года самыми популярными инструментами, предлагающими свои средства для автоматизации браузера являются: Selenium Webdriver, Puppeteer, Cypress, TestCafe. Количество скачиваний за последний год показано на графиках на рисунках 1.1 и 1.2.

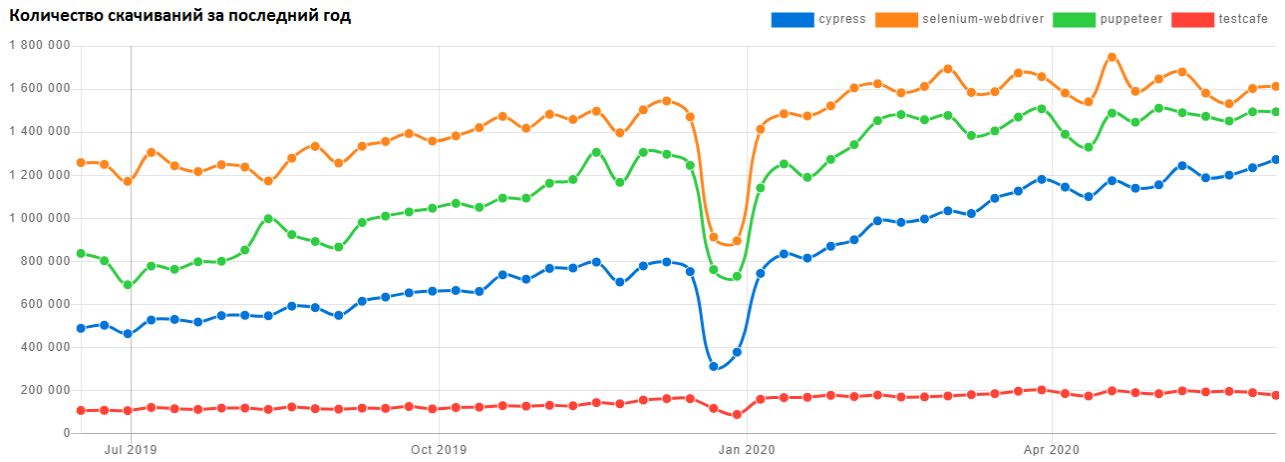

Рисунок 1.1 – Графики количества скачиваний за последний год

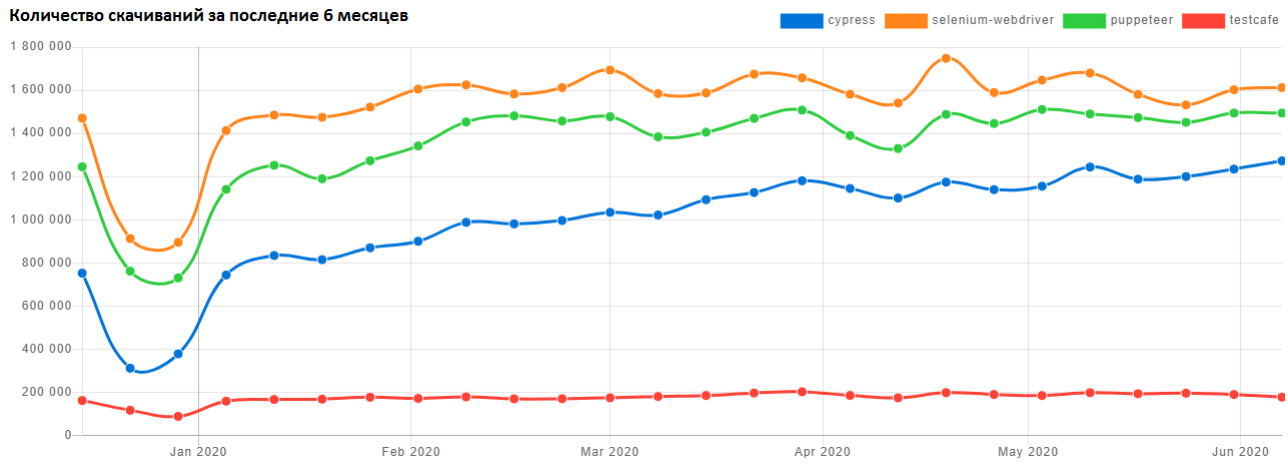

Рисунок 1.2 – Графики количества скачиваний за последние 6 месяцев

Каждый из них реализует свою концепцию управления и автоматизации браузера, отличную друг от друга. Это в свою очередь означает, что используя разные инструменты, разработчики тестов должны учитывать их особенности и подход к построению тестового фреймворка. Инструмент TestCafe является платным, и судя по описанию похож по функционалу на Cypress и Puppeteer, следовательно, для начала необходимо изучить данные инструменты, а затем уже решать вопрос о покупке TestCafe.

Selenium

Это целый набор инструментов позволяющий осуществлять браузерную автоматизацию. Отличительными чертами Selenium являются возможность написания сценариев на JavaScript, C#, Java, Ruby, Python и поддержки большинства современных браузеров (Chrome, Firefox, Safari, Edge). Ключевым инструментов для работы с браузером является Selenium WebDriver [12].

Автоматизированные сценарии пишутся на одном из предпочитаемых языков, после чего language-binding Selenium для конкретного языка транслирует команду в JSON и посылает ее (через HTTP) на Selenium server [13]. С помощью встроенного набора Browser Drivers осуществляет коммуникацию и контроль над браузером. Схематично это показано на рисунке 1.3.

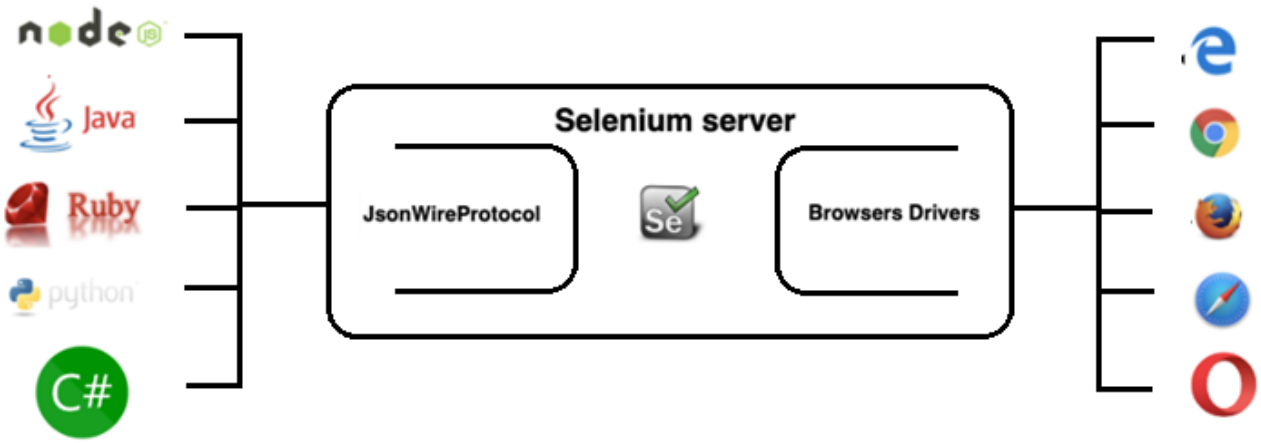

Рисунок 1.3 – Принцип работы Selenium WebDriver

Непосредственно для написания тест-кейс сценариев нужно подключить предпочитаемую библиотеку, фреймворк для тестирования: jasmine, mocha, jest, testNG, jUnit. И в том числе assertion library.

Достоинства:

1. Гибкость в использовании и выборе языка, платформы, браузера.

2. Selenium Webdriver – это стандарт индустрии построенный на утвержденном web стандарте W3C WebDriver.

3. Большая поддержка сообщества. Продукт разрабатывается с 2004 года, поэтому существует множество примеров со многими решенными задачами. Минорные релизы происходят в среднем каждые три месяца, разработчики активно помогают в разрешении проблем.

4. Поддержка параллельного запуска тестов (Selenium Grid).

Недостатки:

1. Весьма нетривиальная установка.

2. Нет встроенного хорошего инструмента для построения развернутого отчета об ошибках. Необходимо обращаться к библиотекам.

3. Нет встроенной возможности для сравнения изображений.

Puppeteer

Это open-source библиотека, которая предоставляет высокоуровневый API для запуска, контроля и управления браузера – Chromium.

В отличии от Selenium Webdriver, коммуникация происходит непосредственно с браузером, хотя и через тот же Chrome DevTools Protocol, который использует ChromeDriver Selenium-a [14]. Особенность здесь тем не менее заключается в том, что Puppereer развивается и обновляется намного динамичнее, чем это происходит с ChromeDriver Selenium-a. Надо заметить, что Google уже достаточно давно не участвует в разработке Selenium, даже в качестве спонсора. И chromedriver обновляет очень редко, в том числе долго не исправляет критические баги. Способствует переходу от кроссбраузерной автоматизации в сторону "chrome only".

Схематично принцип работы Puppeteer показан на рисунке 1.4.

Рисунок 1.4 – Принцип работы Puppeteer

Достоинства:

1. Весьма богатый API для работы с сетевыми запросами, и их перехватами.

2. Наличие возможности существенного ускорения тестов с помощью Puppeteer Browser Context, позволяющей избавится, от затратного по времени, создания новых инстансов браузера, при каждом тест кейсе, путем создания изолированных в рамках браузера сессий независящих друг от друга.

3. Puppeteer в данный момент разрабатывает в том числе и Firefox версию, которая уже на июль 2019 поддерживает до 90% API Puppeteer chromium, но все еще находится в экспериментальном статусе.

4. Наличие API для тестирования приложения в offline-mode.

Недостатки:

1. По факту поддержка работы только с одним единственным браузером – Chomium.

2. Нет встроенного механизма параллелизации тестов.

3. Необходимость подключения библиотек для полноценного, всеобъемлющего тестирования.

Cypress

Это open-source фреймфорк для автоматизации тестирования. Это также как и Puppeteer относительно молодой инструмент, однако он вносит новые концепции и решения в способы осуществления автоматизации и тестирования.

Ключевой особенностью, Cypress является то, что он исполняется внутри самого браузера. Это в том числе означает, что Cypress всегда отслеживает моменты вызова всякого рода событий в браузере и никогда не упустит любые манипуляции с элементами страницы, что намного уменьшает вероятность появления floating-тестов [15].

Достоинства:

1. Встроенный набор инструментов для тестирования построенный на mocha, chai, sinon.

2. Встроенный механизм автоматического ожидания. Это означает, что при написании сценариев нет необходимости писать async/await функции как это делается в Puppeteer и Selenium. Cypress сам подождет, когда появится нужный элемент, подождет, когда закончится анимация, и подождет, когда очередной сетевой запрос завершится.

3. Time machine – фича, которая позволяет в Cypress test runner откатываться на определенные шаги в последовательности выполнения теста.

4. Исчерпывающая документация с большим набором примеров.

5. Возможность написания в том числе и unit тестов.

Недостатки:

1. Нет кросс-браузерной поддержки. Только Chomium, Chrome и Electron.

2. Нет возможности создавать как еще одну вкладку так и еще одно окно.

3. Нет поддержки native events браузера.

4. Не предназначен для performance тестирования.

5. Нет встроенной поддержки Xpath.

Cypress очень сильно отличается от Selenium, поэтому требуется некоторое время, чтобы привыкнуть к расположению и взаимодействию элементов.

Перед тем как начинать использовать Cypress следует в том числе принять во внимание его такую особенность, как встроенный механизм очистки local/session storage, cookie и всего кэша срабатывающий после каждого

тестового сценария. Собственно, это означает, что в Cypress не получится поддерживать такую концепцию в написании тестов, которую можно применять при написании тестов с помощью Selenium и Puppeteer [16], где можно использовать одну и ту же сессию между it-ами, в Cypress каждый it – это отдельная независимая сессия.

Хотелось бы отметить, что выбор инструмента автоматизации зависит прежде всего от потребностей и особенностей проекта, а также от технических навыков разработчиков и, самое главное, тестировщиков, которые и будут писать тесты. Некоторые критерии сравнения, касаемые разработки, приведены в таблице 1.1.

Таблица 1.1 – Результаты сравнения инструментов автоматизации

Фреймворк	**Кроссбраузерная поддержка**	**Headless браузер**	**Встроенная библиотека для тестирования**	**Механизм ожидания**	**Способ поиска элементов DOM**
Selenium	Есть	Есть	Нет	Явный	CSS Selectors, Xpath
Puppeteer	Нет	Есть	Нет	Явный	CSS Selectors, Xpath
Cypress	Нет	Есть	Есть	Авто	JQuery Selectors

Не смотря на плюсы инструмента Cypress, использоваться на проекте он не будет, так как данный инструмент использует другую концепции написания тестов и тестовых наборов, как было сказано выше. В компании имеется небольшой опыт создания тестового фреймворка с использованием Selenium WebDriver и тестового фреймворка TestNG на учебном проекте, поэтому уже сложилось понимание концепции разработки с использованием PageObject паттерна. Данный паттерн можно реализовать и с помощью инструмента Puppeteer [17].

Ограничение кросс-браузерного тестирования и возможность использования Puppeteer только на одном языке, на мой взгляд, хоть и делают Selenium главным инструментом для веб-тестирования на данный момент, но кроссбраузерность не стоит в приоритетах команды, а использование JavaScript наоборот является плюсом, так как команда имеет большой опыт работы с программной платформой Node.js. Также нужно учитывать, что Puppeteer теснее связан с Chrome чем WebDriver, у него выше функциональность и главное стабильность.

Также оба инструмента поддерживают Xpath для поиска узлов в DOM, в то время как Cypress использует только JQuery селекторы, и нет возможности искать элементы перемещаясь по узлам объектной модели в разных направлениях, что очень удобно.

1.2.2 Выбор языка для автоматизации тестирования

Множество компаний считает, что тестировщики обязаны использовать тот язык программирования, на котором написано тестируемое приложение. Если компания пишет на Java, то все тестовые решения обязаны быть именно на Java [18]. По моему мнению это неверный подход к вопросу.

Можно выделить несколько критериев, по которому тестировщик может выбрать тот или иной язык для автоматизации:

1. Наличие надежной и хорошо поддерживаемой версии инструмента для тестирования.

Этот пункт стоит на первом месте. Инструменты и фреймворки действительно появляются очень часто, но при этом некоторые из них регулярно выпускают обновления, поддерживаются большим сообществом и постоянно развиваются, а другие просто опубликованы автором как результат эксперимента или локальной разработки.

В качестве примера можно привести пример – для PHP клиента существует несколько реализаций WebDriver. Все они гораздо сложнее своих

аналогов в других языках программирования и развиваются независимо. Поэтому PHP не стоит использовать для тестирования с WebDriver напрямую, есть библиотеки, которые содержат свои прослойки над WebDriver API.

2. Знание языка членами команды.

В этом пункте речь идет не только о тестировщиках, но и обязательно о разработчиках. У них должна быть возможность поправить тесты в случае изменения деталей реализации приложения: расположение элементов, тип элементов, переходы между страницами и т.д. Причем, это должно быть достаточно легкая операция с точки зрения знаний языка. И вовсе необязательно это должен быть тот язык, на котором разрабатывается само приложение. Если все члены команды разбираются в синтаксисе Python, а приложение пишется на Java, то можно выбрать любой из этих языков.

3. Наличие хорошего IDE.

Этот пункт особенно важен для тех команд, где работать с авто-тестами будут даже неопытные с точки зрения программирования тестировщики. Хорошее IDE помогает многие вещи генерировать вместо того, чтобы набирать руками. Также гораздо ниже вероятность допустить ошибку, потому что IDE подсвечивает потенциальные проблемы и контролирует код по мере его появления.

В этом пункте также стоит подумать над вопросом динамических и статических языков. У динамических обычно код тестов гораздо более читабельный и простой. У статических более громоздкий, но зато меньше подвержен “неожиданным” ошибкам начинающих тестировщиков.

Хорошим примером являются Groovy и Java. Оба языка позволяют писать код с привычным Java синтаксисом, но многие вещи можно сделать значительно проще на уровне языка Groovy.

4. Наличие готовых решений для контекста тестирования.

Иногда в каком-то языке выбор готовых решений на порядок больше, чем в других языках. При прочих равных условиях стоит выбрать тот, в котором

есть готовое решение, ближе всего к контексту тестирования. Это избавит от необходимости писать многие вещи самостоятельно. Больше всего решений в области автоматизации тестирования сделано для Java и Ruby.

5. Опыт использования в компании.

Этот критерий достаточно важен для компаний, в которых автоматизация тестирования делается на многих проектах. Выбор единого решения может решить сразу несколько потенциальных проблем: совершение одних и тех же ошибок в каждом проекте, обмен знанием и опытом по решению задач автоматизации, использование общих библиотек, компонент и наработок, участие в тренингах, семинарах, конференциях. Одним словом, стоит накапливать опыт и экспертизу внутри компании, чтобы каждый раз типовые задачи решались все проще и проще.

6. Простота изучения языка.

Этот пункт последний, потому что для автоматизации тестирования достаточно очень узкого подмножества возможностей языка. Изучаются они обычно недолго, особенно под присмотром разработчиков. Процесс обязательного review кода тестов также помогает в короткие сроки распространить знания об основах языка. Поэтому на первое место выходят возможности писать читабельные и хорошо структурированные тесты.

Основываясь на имеющимся опыте автоматизации тестирования, для первого опыта разработки тестового фреймворка для автоматизации тестирования веб-приложений был выбран язык Java. Данный выбор соответствует критериям: наличие готовых решений для контекста тестирования, наличия надежной и хорошо поддерживаемой версии инструмента для тестирования – Selenium, а также удобная среда разработки.

Но исходя из критериев: знания языка членами команды и простоты изучения, для использования в автоматизации тестирования язык Java не подходит для использования в разработке тестового фреймворка. Язык JavaScript более популярен в разработке веб-приложений, соответственно

основная часть веб-разработчиков владеет навыками разработки с использованием данного языка. Соответственно, использовать язык JavaScript в качестве языка автоматизации тестирования более предпочтительно в плане удобства в команде, но окончательный выбор следует делать после применения инструментов автоматизации тестирования веб-приложений с использованием разных языков программирования, чтобы в полной мере оценить их удобство и эффективность при разработке автотестов.

1.2.3 Обзор и анализ инструментов непрерывной интеграции

Непрерывная интеграция (Continuous Integration) в разработке программ это автоматизированный процесс сборки и тестирования кода в разделяемом репозитории. Когда делаются новые коммиты, они изолируются, собираются и тестируются на соответствие определенным стандартам, прежде чем вольются в основную кодовую базу [19].

Непрерывная интеграция позволяет быстро выявлять поломки, ошибки или баги, при этом все перечисленное не попадает в кодовую базу, а исправляется как можно скорее.

Непрерывная интеграция обеспечивает множество преимуществ, среди которых:

- Действительно раннее обнаружение проблем и исправление их до слияния кода.
- Более короткие и менее напряженные интеграции.
- Благодаря улучшению видимости повышается эффективность коммуникации.
- На поиск багов уходит меньше времени.
- Больше не нужно ждать, пока код тестируется.
- Повышается эффективность быстрой доставки ПО.
- Делается возможным непрерывный фидбэк по изменениям, что со временем может улучшить продукт.

Далее рассмотрим основные системы непрерывной интеграции и особенности их работы [20].

Jenkins

Jenkins это инструмент непрерывной интеграции с открытым исходным кодом. Написан он на Java. Этот проект собрал больше 11 тысяч звезд на GitHub.

Jenkins предоставляет возможность тестирования кода в режиме реального времени, а также дает возможность получать отчеты об отдельных изменениях в обширной кодовой базе. Этот инструмент, главным образом, позволяет разработчикам быстро помечать и исправлять ошибки и баги в коде, а затем автоматически тестировать сборку кода.

Благодаря интуитивному пользовательскому интерфейсу Jenkins очень легко настраивать и конфигурировать. Он доступен на операционных системах Linux, Macintosh и Windows. Jenkins создан для крупномасштабных интеграций, благодаря чему можно легко распределять работу между различными машинами.

Наличие больше 1000 плагинов позволяет автоматизировать практически что угодно. В результате члены вашей команды смогут посвятить свое время исключительно тем задачам, с которыми не способны справиться машины.

TeamCity

TeamCity – сервер непрерывной интеграции корпоративного уровня. Он поддерживает большое количество мощного функционала, а также имеет очень надежную бесплатную версию для маленьких проектов (до 100 конфигураций сборки). TeamCity создан командой JetBrains.

Поставляется он с обширной поддержкой множества плагинов с открытым исходным кодом – как собственных продуктов JetBrains, так и сторонних приложений и инструментов. Также TeamCity предлагает хорошую поддержку .NET.

Благодаря всему этому данный сервер непрерывной интеграции отличается высокой надежностью, не зависящей от запуска сборок. TeamCity имеет очень понятную интеграцию с системами контроля версий. Коммиты можно предварительно тестировать, а команды – запускать удаленно.

Travis

Travis – очень популярный инструмент непрерывной интеграции (больше 7 тысяч звезд на GitHub). Он бесплатен для проектов с открытым исходным кодом. Этот инструмент можно назвать не кроссплатформенным, а платформо-независимым, поскольку что это веб-ресурс.

Travis поддерживает много языков программирования, включая Node и PHP, а также много конфигураций сборки.

Этот инструмент поставляется с очень мощным API и интерфейсом командной строки. Его легко настраивать, а инсталлировать и вовсе не нужно. Travis интегрирован с такими сервисами коммуникации как Slack, HipChat и даже с электронной почтой. Для сборки приложений используются виртуальные машины. Допускается параллельное тестирование.

Gitlab CI

GitLab Continuous Integration это часть GitLab. По сути, это веб-приложение с API, сохраняющим его состояние в базе данных. Используется для менеджмента проектов и предоставляет дружественный к пользователю, интуитивный интерфейс, а также все другие преимущества функционала GitLab.

На данном этапе разработки авто-тестов система непрерывной интеграции не актуальна, но в будущем планируется ее внедрение. Выбор пал на два инструмента: Jenkins, по причине того, что он имеет хорошую интеграцию с языком программирования Java, и Gitlab CI, поскольку компания уже имеет опыт работы с данным инструментом и некоторые проекты уже хранятся в репозиториях Gitlab.

2 Выбор инструментов для автоматизации тестирования веб-приложений

2.1 Краткое описание тестируемого веб-приложения zener.ru

Веб-приложение, выбранное для примера автоматизации тестирования, представляет собой интернет-магазин «zener.ru» показанный на рисунке 2.1. Это клиент-серверное приложение, использующее протокол HTTPS, написанное на php.

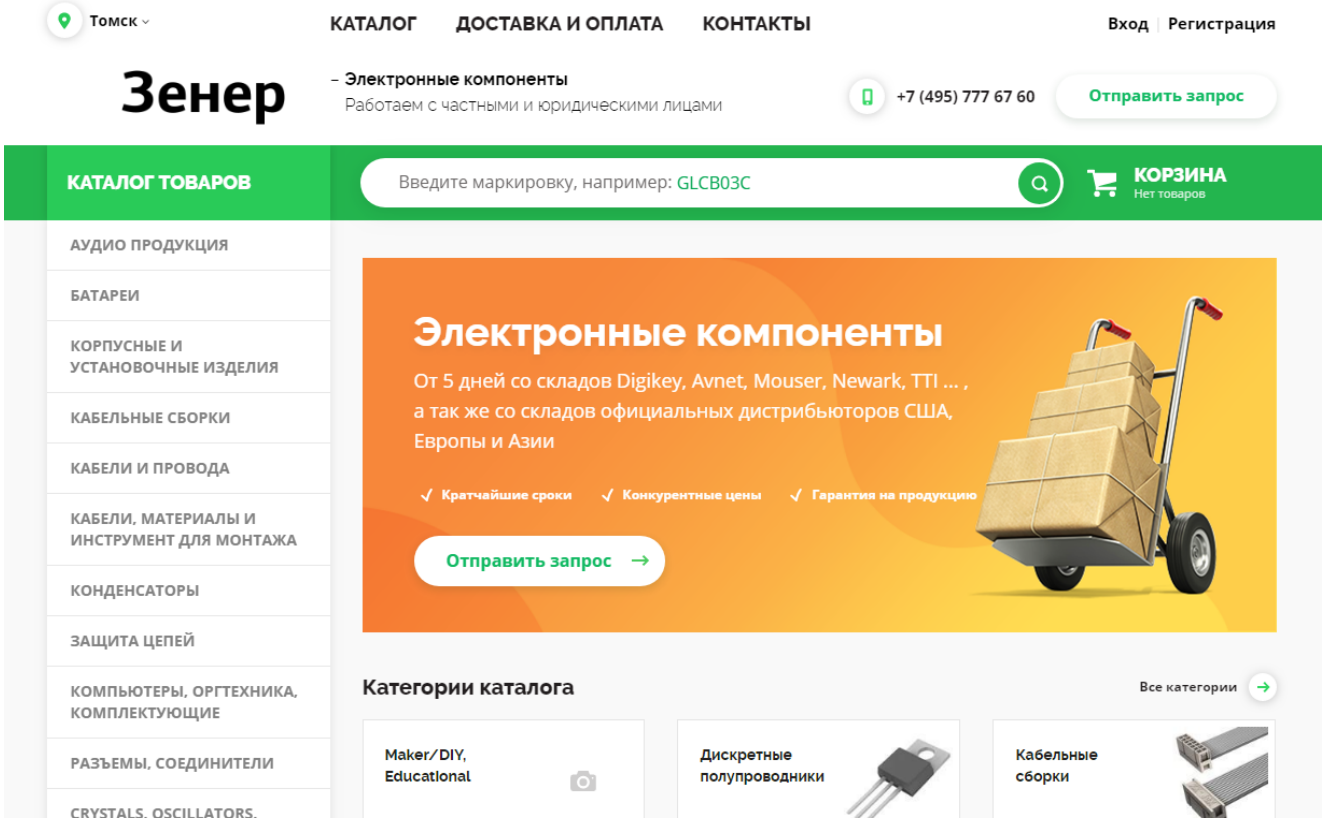

Рисунок 2.1 – Главная страница тестируемого веб-приложения

Соответственно в своей структуре оно имеет: главную страницу, страницу карточки товара – «../product/», страницу корзины – «../cart/», страницу оформления заказа – «../order/», а также страницу информации о формировании заказа. Конечно, имеются и другие страницы в карте веб-приложения, но перечислены только используемые для написания и прохождения тестового сценария.

2.2 Создание тестового сценария для тестируемого веб-приложения

В качестве тестового сценария для первого автотеста был выбран сложный кейс отражающий бизнес логику тестируемого интернет-магазина. Но направленность теста заключается не в проверке успешности формирования заказа, а в корректности формирования заказа. А именно, в сравнении параметров выбранного товара до добавления его в корзину и оформления заказа и после успешного формирования заказа. Это меняет подход к покрытию сценария проверочными условиями. В данном тесте не нужно проверять успешность выполнения этапов формирования заказа.

Тестовый сценарий показан на рисунках 2.2, 2.3, 2.4, 2.5, 2.6 и выглядит следующим образом:

1. Зайти на страницу товара.
2. Выбрать пункт из фильтра.
3. Запомнить необходимые параметры товара.
4. Нажать кнопку «Добавить в корзину».
5. Нажать кнопку «Оформить заказ».
6. Заполнить форму оформления заказа.
7. Нажать кнопку «Оформить заказ».
8. Запомнить номер заказа.
9. Получить данные о заказе, включая параметры товара, по номеру заказа из пункта №8 через REST API.
10. Сравнить параметры товара из пункта №3 с параметрами товара из пункта №9.

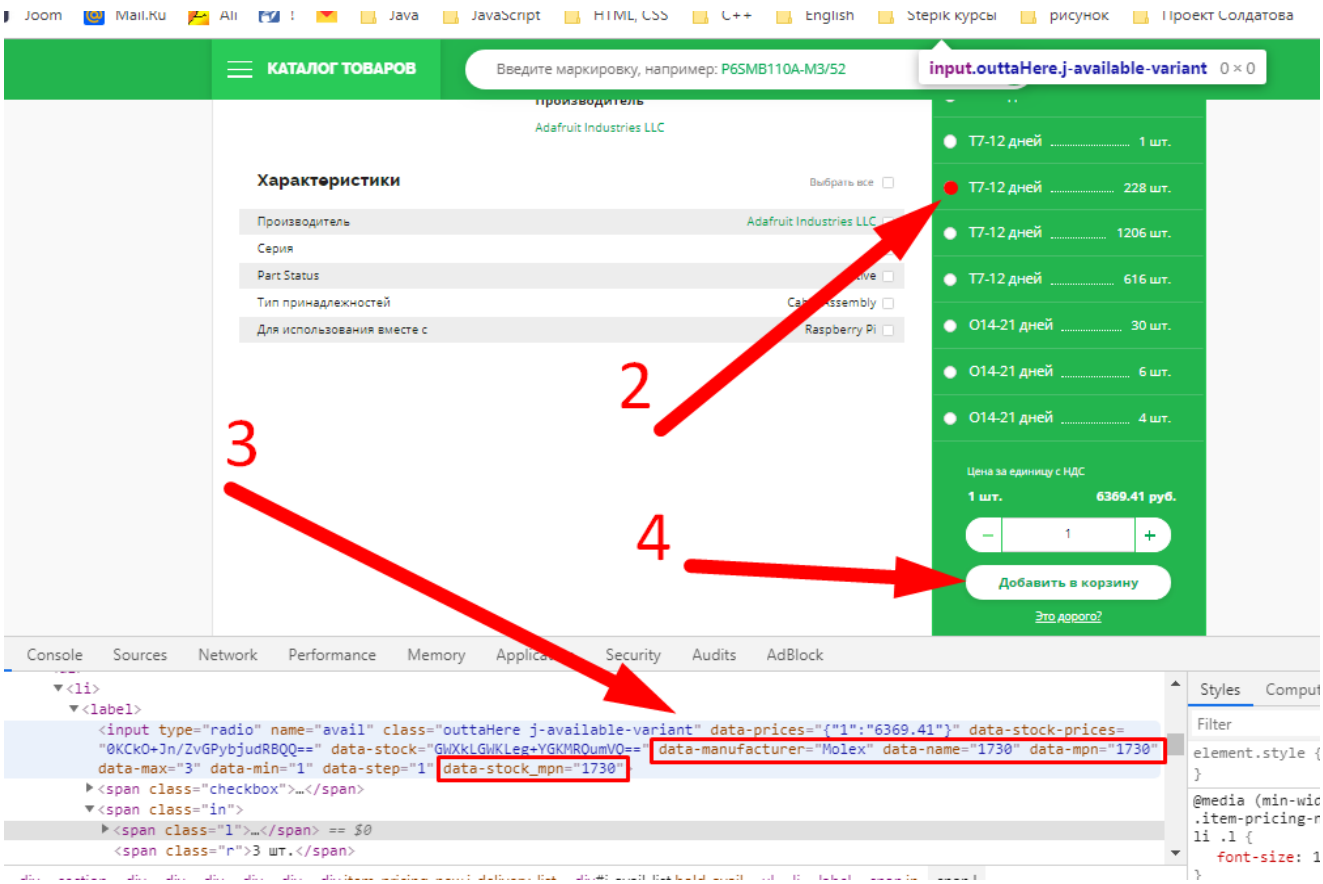

Рисунок 2.2– Шаги № 1, 2, 3 и № 4 тестового сценария

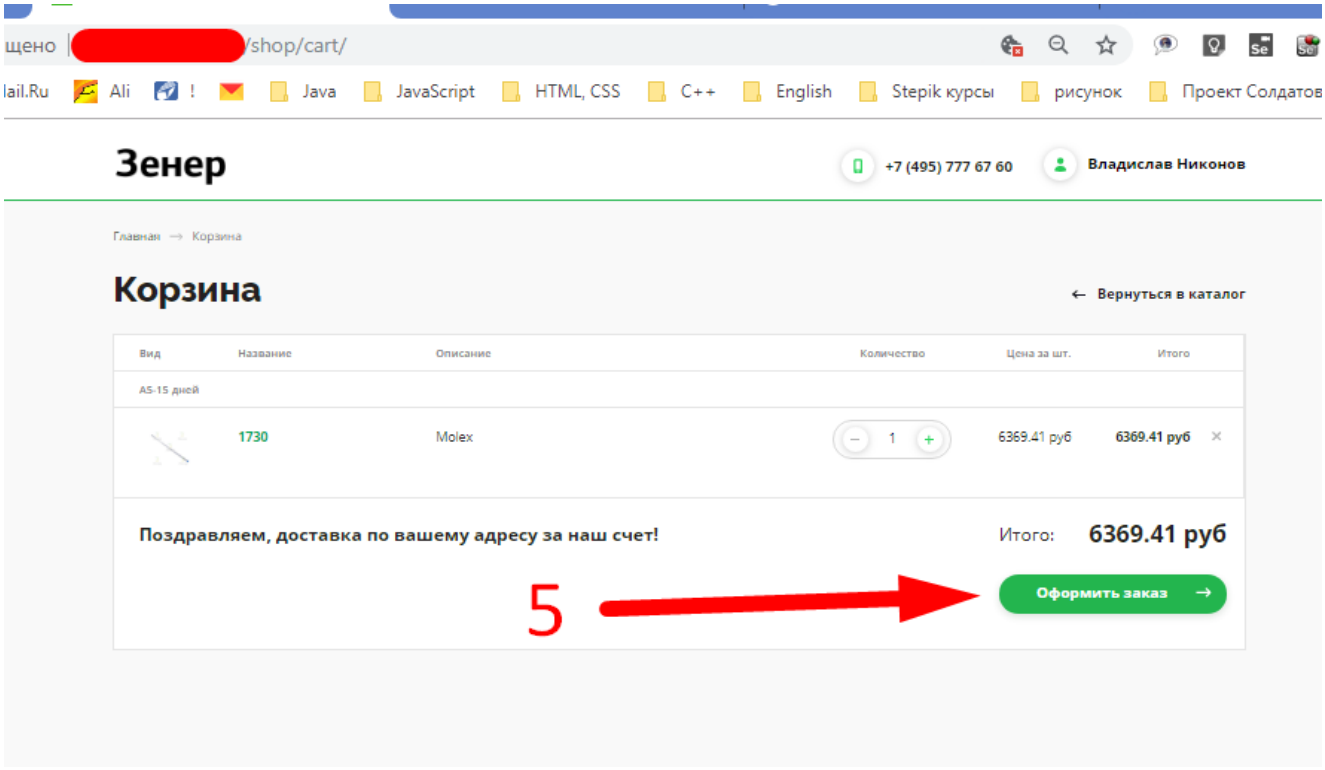

Рисунок 2.3 – Шаги № 5 тестового сценария

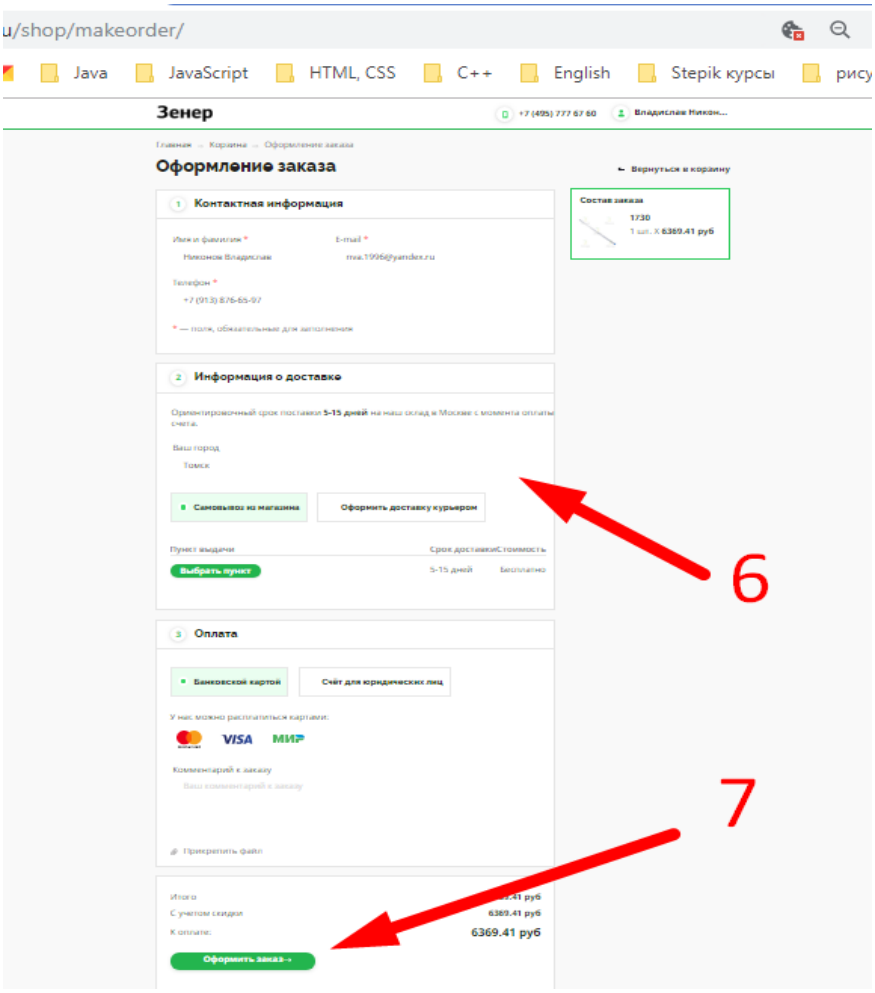

Рисунок 2.4 – Шаги № 6 и № 7 тестового сценария

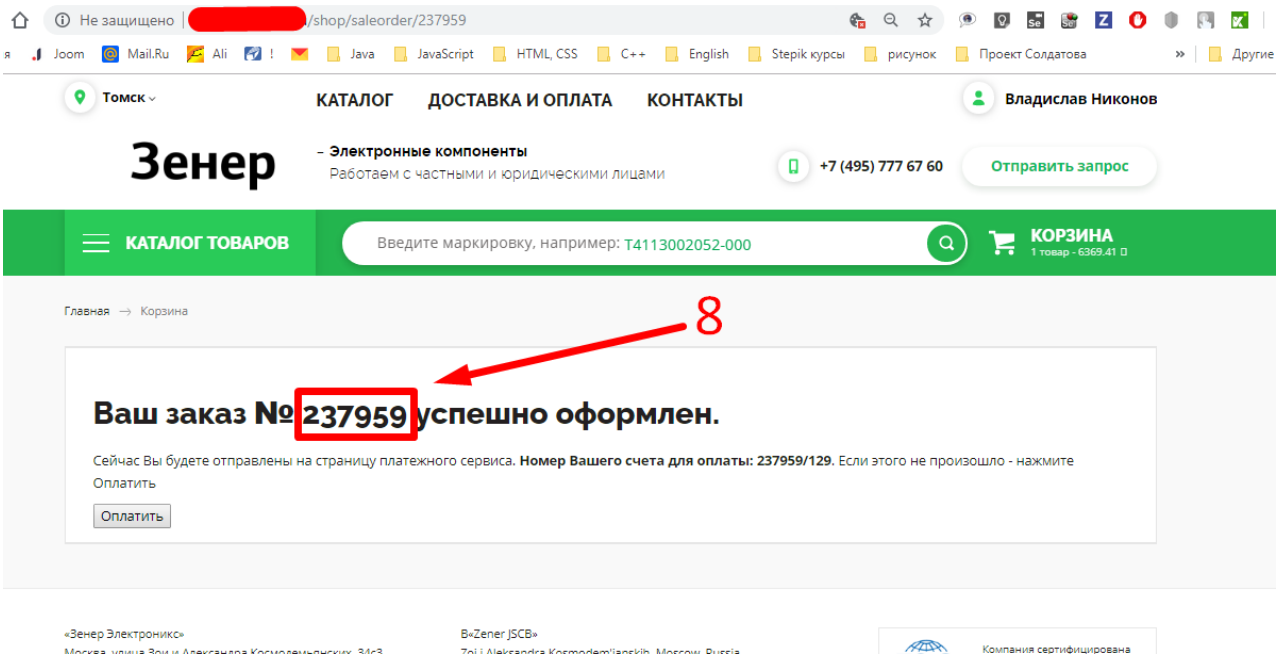

Рисунок 2.5 – Шаг № 8 тестового сценария

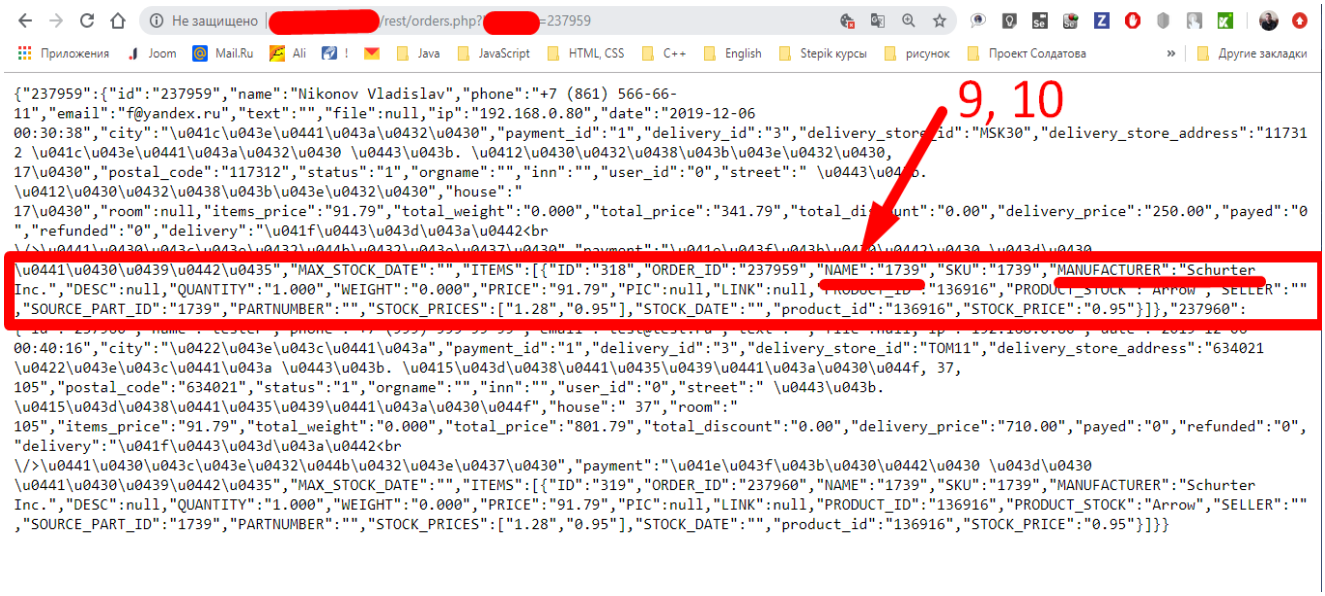

Рисунок 2.6 – Шаги № 9 и № 10 тестового сценария

Это неверный подход со стороны стратегии покрытия приложения автотестами. В первую очередь необходимо автоматизировать позитивные тестовые сценарии успешной авторизации, успешной регистрации, успешного формирования заказа, учитывая все комбинации заполнения формы заказа и т.д., то есть разработать короткие и быстрые автотесты для дымового тестирования. Но на данный момент, команда разработки не преследует цели полного покрытия веб-приложения «zener.ru» автотестами.

Разрабатываемый автотест используется с двумя целями:

1. Проверка и выявление ошибок в функционале серверной части приложения, отвечающего за работу с параметрами товаров при формировании заказа.

2. Сбор информации, для последующего сравнения стеков технологий для автоматизации тестирования. Критерии: скорости разработки автотеста, скорость работы автотеста, поддерживаемость и читаемость кода автотеста.

2.3 Разработка тестового фреймворка с использованием Selenium

Основываясь на имеющимся опыте автоматизации тестирования, для первого опыта разработки тестового фреймворка для автоматизации

тестирования веб-приложений выбран стек технологий для разработки автотестов: Java, Selenium Web Driver и TestNG.

2.3.1 Сравнение тестовых фреймворков Junit и TestNG

Чем раньше начнется процесс тестирования, тем раньше продукт попадет клиенту, который в свою очередь сможет начать использовать его в среде потенциально заинтересованных пользователей. Поэтому при проведении тестов, лучше всего обратиться к современным возможностям автоматизации данного процесса.

TestNG – это весьма популярный фреймворк автоматизации, полностью созданный на языке программирования Java и взявший некоторые вещи с JUnit. Отличается простотой использования, многообразием предоставленных функций, а также большой эффективностью в создании набора тестов для автоматизации веб-продуктов различной направленности [21].

Функциональные возможности TestNG:

- Полная аннотация совершенных операций.
- Наличие XML для полноценной конфигурации созданных тестов.
- Работа с data driven тестами.
- Использование методов для проверки серверных утилит.
- Полноценная техническая поддержка Hudson, Ant, IDEA и Eclipse.
- Многопоточная функция тестирования программного кода.
- Интуитивно понятный интерфейс.

Именно с помощью инструмента TestNG можно запросто реализовать любой типовой вид тестов: от функциональных до интеграционных и модульных тестов. Для подобных целей рекомендуется использовать JDK.

TestNG позволяет достигнуть максимальной параметризации информации.

При работе с данным фреймворком процесс создания наборов тестов состоит из такой последовательности:

- Разработка будущей бизнес-логики теста.
- Процесс внедрения TestNG аннотаций в программный код.
- Детализированное описание теста.
- Старт работы TestNG.

При желании к созданной аннотации можно добавлять вспомогательные параметры. Так как все аннотации являются строго шаблонными, компилятор быстро найдет ошибку и укажет тестировщику, где именно нужно искать баг.

То есть, процесс тестирования программных продуктов с помощью специального инструмента **TestNG** – это верный шаг к максимальному временному сокращению проверки продукта перед финальным выпуском.

JUnit – библиотека для модульного тестирования программ Java. Созданный Кентом Беком и Эриком Гаммой, JUnit принадлежит семье фреймворков xUnit для разных языков программирования, берущей начало в SUnit Кента Бека для Smalltalk. JUnit породил экосистему расширений – JMock, EasyMock, DbUnit, HttpUnit и т. д.

Библиотека **JUnit** была портирована на другие языки, включая PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test::Unit), C++ (CPPUnit), Flex (FlexUnit), JavaScript (JSUnit).

JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов.

И JUnit, и TestNG являются современными инструментами для тестирования в экосистеме Java [22]. Различия между этими инструментами указаны в таблице 2.1.

Таблица 2.1 – Сравнение функционала библиотек TestNG и JUnit

Описание	**TestNG**	**JUnit 4**
Тестовая аннотация	@Test	@Test
Выполняется до вызова первого тестового метода в текущем классе	@BeforeClass	@BeforeClass

Продолжение таблицы 2.1

Описание	TestNG	JUnit 4
Выполняется после всех тестовых методов в текущем классе	@AfterClass	@AfterClass
Выполняется перед каждым методом теста	@BeforeMethod	@Before
Выполняется после каждого метода испытаний	@AfterMethod	@After
Аннотация, чтобы игнорировать тест	@Test(enable=false)	@ignore
Аннотация для исключения	@Test(expectedExceptions = ArithmeticException.class)	@Test(expected = ArithmeticException.class)
Тайм-аут	@Test(timeout = 1000)	@Test(timeout = 1000)
Выполняется перед всеми тестами в наборе	@BeforeSuite	Не реализовано
Выполняется после всех тестов в наборе	@AfterSuite	Не реализовано
Выполняется до запуска теста	@BeforeTest	Не реализовано
Выполняется после запуска теста	@AfterTest	Не реализовано
Выполняется до вызова первого тестового метода, принадлежащего к любой из этих групп	@BeforeGroups	Не реализовано
Запустить после последнего метода теста, который принадлежит к любой из групп здесь	@AfterGroups	Не реализовано

В последующей разработке будет использоваться тестовый фреймворк TestNG. Поскольку TestNG более продвинут в тестировании параметризации, тестировании зависимостей и тестировании комплекта (концепция группирования). TestNG предназначен для высокоуровневого тестирования и комплексного тестирования интеграции. Его гибкость особенно полезна для больших наборов тестов. Кроме того, TestNG также охватывает всю функциональность ядра JUnit4.

2.3.2 Проектирование архитектуры тестового фреймворка

Для высокой поддержки кода и упрощения разработки автотестов в будущем используется двухуровневая архитектура [23]. Автотесты “системного” уровня, в отличие от “unit-тестов”, удобно разделить на два слоя: первый слой, собственно, сами тесты, второй слой – код, ответственный за взаимодействие с тестируемой системой, причём вторая часть, как правило, является более сложной технически.

Был создан общий базовый класс для тестов TestBase, в него перенесены вспомогательные методы, включая запуска и остановки экземпляра браузера. Далее создан класс ApplicationManager и все вспомогательные методы: выбор драйвера браузера, инициализация выбранного драйвера и настройка его параметров, перенесены в данный класс. Это реализовано с помощью механизма делегирования.

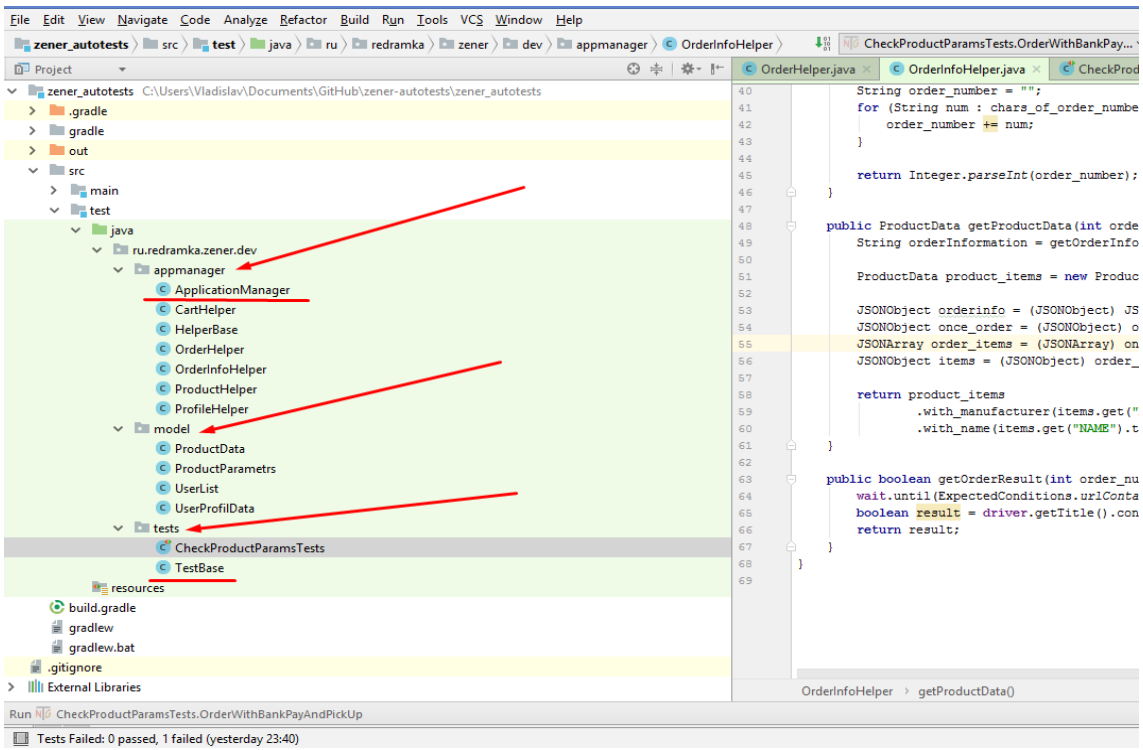

Рисунок 2.7 – Двухуровневая архитектура тестового фреймворка

В результате построения двухуровневой архитектуры, разрабатываемая система автоматизации тестирования приобрела вид, показанный на рисунке 2.7. В директории «appmanager» содержится код, отвечающий за управление

автотестами, в директории «model» содержатся объекты параметров методов, две этих части составляют первый уровень архитектуры, а в директории «tests» нет ничего лишнего, кроме самих тестов, он составляет второй уровень двухуровневой архитектуры.

2.3.3 Применение паттерна Page Object и разработка дополнительных методов

Page Object – один из наиболее полезных и используемых архитектурных решений в автоматизации. Данный шаблон проектирования помогает инкапсулировать работу с отдельными элементами страницы, что позволяет уменьшить количество кода и упростить его поддержку. Если, к примеру, дизайн одной из страниц изменён, то нам нужно будет переписать только соответствующий класс, описывающий эту веб-страницу.

Основные преимущества Page Object в разделении кода тестов и описания страниц, а также в объединение всех действий по работе с веб-страницей в одном месте.

Для реализации данного паттерна следует выделить из ApplicationManager специализированные классы-помощники: ProductHelper, OrderHelper, CartHelper, OrderInfoHelper, ProfileHelper, и перенести в них из ApplicationManager соответствующие вспомогательные методы, как показано на рисунке 2.8. Как видно из названий, каждый класс-помощник соответствует определенной веб-странице и содержит в себе методы, необходимые для работы с элементами и логикой, реализованной на данной странице. Для этих вспомогательных классов создан общий базовый класс BaseHelper, и в него перенесены низкоуровневые вспомогательные методы, такие как заполнение отдельного поля – type(), нажатие на кнопку или ссылку– click(), и т.д.

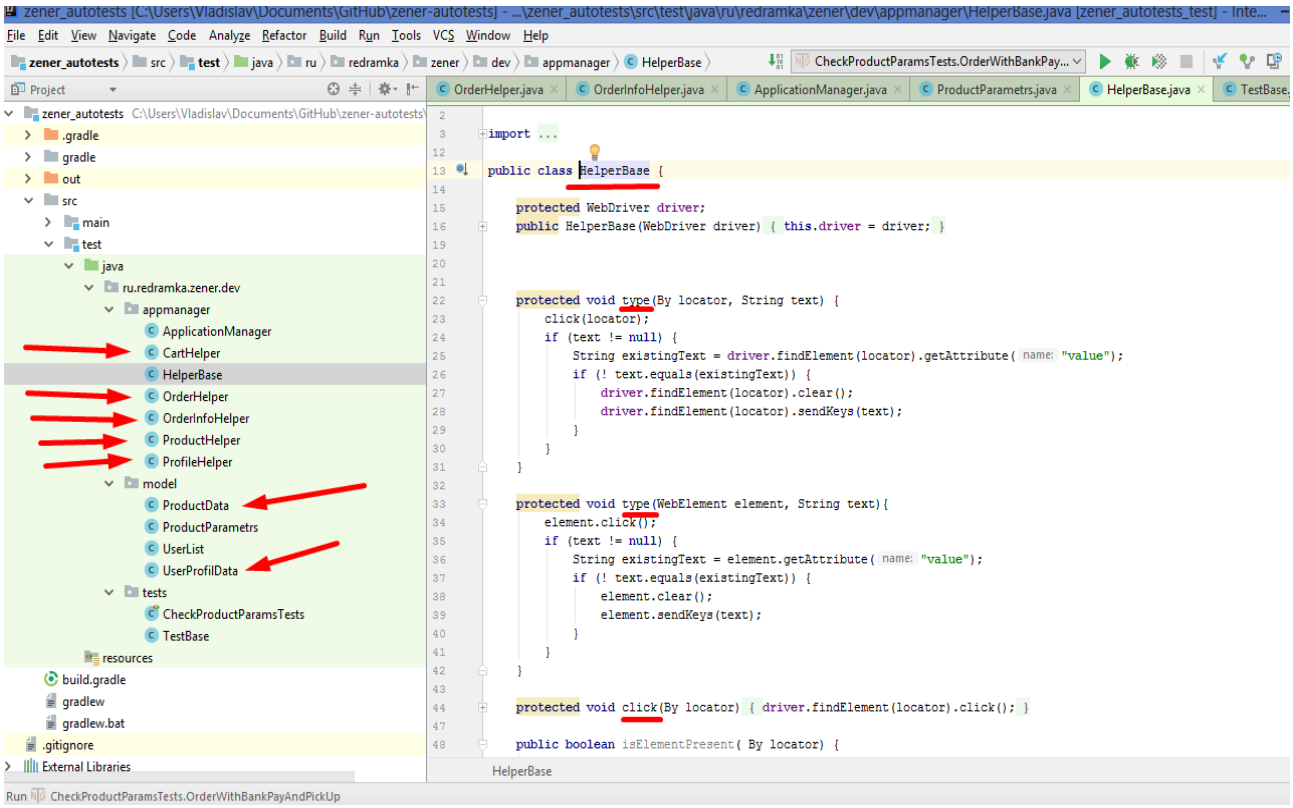

Рисунок 2.8 – Применение паттерна Page Object

Помимо классов реализованных для управления, также реализованы классы UserProfilData и ProductData, для описания таких сущностей тестируемого веб-приложения, как «пользователь» и «продукт», они также показаны на рисунке 3.4. Для хранения списков с данными объектами используются классы ProductParametrs и UserList, которые расширяют абстрактный класс ForwardingList, который в свою очередь наследуется от Collections и реализует все его методы.

В качестве ответа, на запрос информации о заказе, через REST API, приходит текст, описанный как объект, в формате json. Для работы с json-объектами в текущем проекте, используется библиотека json-simple от разработчиков Google. Текстовая строка преобразуется в объект json, а далее разбивается на соответствующие массивы данных и объекты, как показано на рисунке 3.5. Данная реализация используется в методе getProductData(), который принимает на вход параметр orderNumber – номер заказа, а возвращает описанный объект класса ProductData.

2.3.4 Автоматизация тестового сценария

Опираясь на тестовый сценарий, в классах-помощниках были реализованы методы, с помощью которых описываются шаги тестового сценария, а также необходимые проверки, явные и неявные ожидания [24].

В классе ProductHelper реализованы методы:

- getProductParametrs(String productName) – данный метод позволяет получить параметры указанного наименования товара из фильтра на странице товара, через атрибуты.
- isFilterPresent() – позволяет дождаться появления списка фильтров, в противном случае программа сообщит, что элемент отсутствует.
- selectPointfromFilter(int point_number) – метод позволяет выбрать элемент фильтра по номеру.

В классе OrderHelper реализованы методы для оформления заказа:

– makeOrder(String payment, String delivery, UserProfileData user) – данный метод позволяет полностью осуществить оформление заказа, исходя из входных данных. Метод адаптивен к выбору разных способов доставки и оплаты, а данные для заполнения полей клиента, получены из объекта UserProfileData.

– fillUserForm(), fillBillForm(),fillAddressForm() – данные методы позволяют заполнить все формы на странице оформления заказа. Они используются в методе makeOrder().

В классе OrderInfoHelper реализованы методы, которые позволяют работать с уже сформированным заказом:

– getOrderInfo(int orderNumber) – данный метод использует REST API и получает информацию о заказе в текстовом виде. Номер заказа указывается как входной параметр.

– getOrderNumber() – метод позволяет, в случае успешного формирования заказа, получить номер заказа.

– getOrderResult() – метод проверяет успешное формирование заказа.

– getProductData() – основной метод класса OrderinfoHelper. Возвращает описанный объект класса ProductData.

Как видно из названий и реализации методов, все они описывают определенный шаг автоматизируемого тестового сценария. Для написания автотеста необходимо создать класс CheckProductParamsTests в директории tests, который описывает тестовый набор. Каждый тест в нем реализуется в виде метода с соответствующей аннотацией @test, как показано на рисунке 2.9.

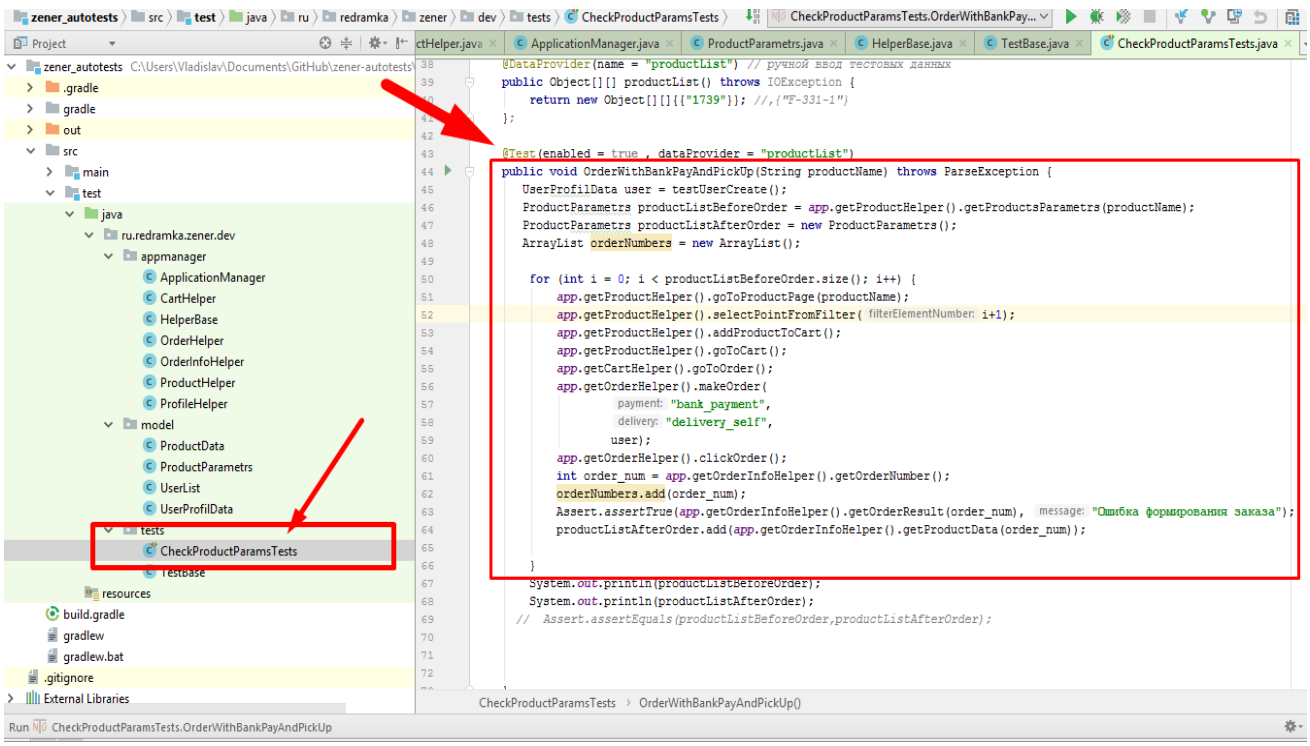

Рисунок 2.9 – Реализация тестового набора checkProductParamsTests

Из реализованных методов из классов-помощников выстраивается цепочка методов, в соответствии с шагами, указанными в тестовой сценарии.

Также добавляется логика и проверки. TestNG позволяет делать проверки с помощью класса Assert. Как показано на рисунке 3.5, данный класс используется для проверки успешного формирования заказа. Далее следует добавить необходимые проверки на сравнение списка товаров productListBeforeOrder, с параметрами до оформления заказа, со списком

товаров productListAfterOrder с параметрами после оформления заказа, с помощью метода equalTo() класса Assert.

2.4 Разработка тестового фреймворка с использованием Puppeteer

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 [25], как показано на рисунке 2.10.

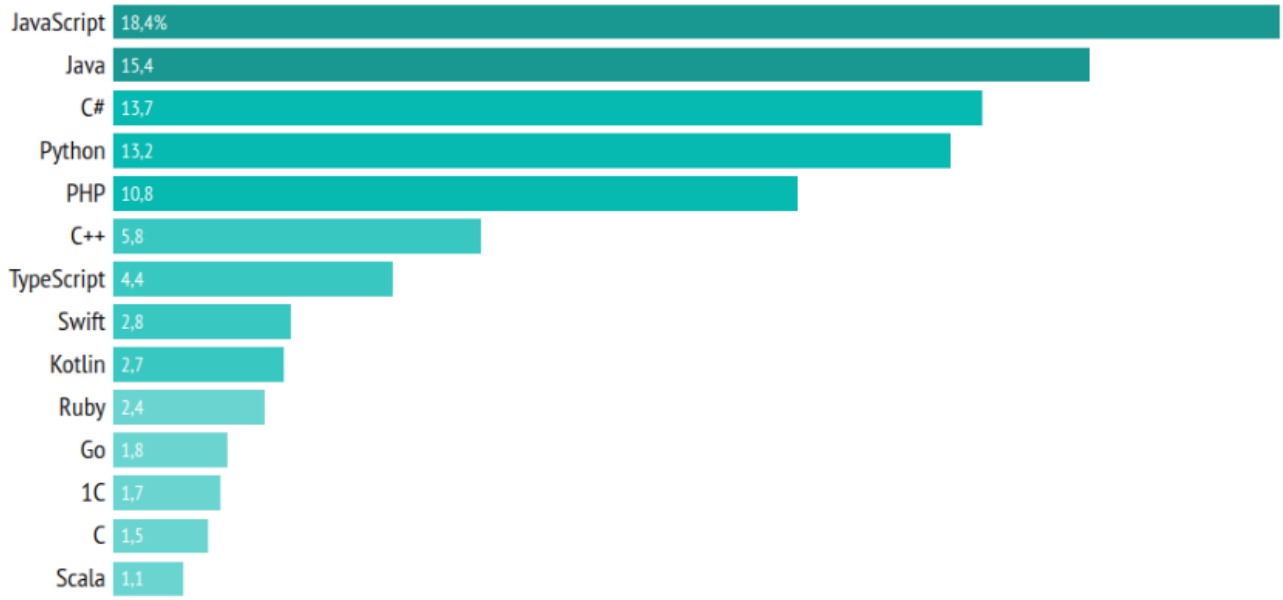

Рисунок 2.10 – Статистика популярности ЯП на начало 2020 г.

Основными факторами перехода на JavaScript стали: скорость написания, настраивания; наличие знаний и опыта в команде и более быстрый вход в проект для QA специалистов, которые только начинают изучать автоматизацию. Для того чтобы начать автоматизировать на JS, необходимо изучить базовые основы языка, выбрать тестовый фреймворк, а также выбрать инструмент для автоматизации. Для построения фреймворка для автоматизации тестирования пользовательского интерфейса веб-приложений были выбраны: тестовый фреймворк Jest и библиотека для браузерной автоматизации Puppeteer.

2.4.1 Обзор платформы Node

Node или 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, подключать другие внешние библиотеки, написанные на разных языках, обеспечивая вызовы к ним из JavaScript-кода. Node.js применяется преимущественно на сервере, выполняя роль веб-сервера [26].

Пакетная экосистема Node.js, npm, является самой большой экосистемой библиотек с открытым исходным кодом в мире. Это библиотеки, построенные сообществом, которые решают большинство часто встречающихся проблем разработчиков. npm, или менеджер пакетов Node, содержит пакеты, которые разработчики используют в своих приложениях, чтобы сделать разработку более быстрой и эффективной [27].

Для создания проекта необходимо открыть терминал в необходимой директории и ввести команду создания проекта «npm init». Менеджер пакетов создаст в директории папку «node_modules», где находятся все необходимые для базовой разработки библиотеки, а также конфигурационный файл *package.json*, как показано на рисунке 2.7, в котором помимо описания проекта, находятся необходимые разработчику зависимости.

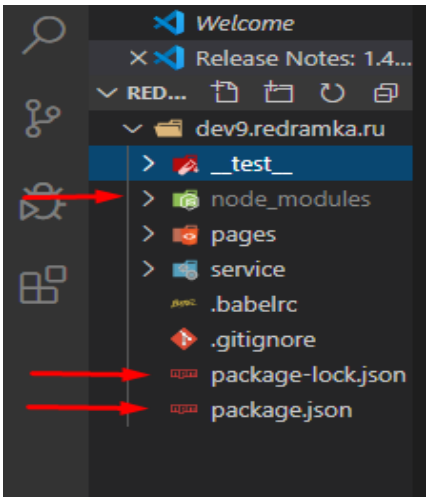

Рисунок 2.7 – Начальная структура проекта

2.4.2 Обзор тестового фреймворка Jest

Jest – это JavaScript test runner, то есть библиотека JavaScript для создания, запуска и структурирования тестов. Jest распространяется в виде пакета npm. Jest – один из самых популярных тестовых фреймворков в настоящее время. Он быстрый и легко настраиваемый. Jest активно разрабатывается и используется Facebook для тестирования всех своих приложений React, а также многими другими разработчиками и компаниями [28].

Для интеграции Jest в проект необходимо ввести в терминале команду «npm i jest --save-dev», менеджер автоматически загрузит в папку необходимые зависимости и модули. Затем, необходимо указать псевдоним для запуска тестов в *package.json*, как показано на рисунке 2.11.

Рисунок 2.11 – Создание команды запуска тестового набора

2.4.3 Проектирование структуры директорий

Для высокой поддержки кода и упрощения разработки автотестов необходимо грамотно подойти к вопросу проектирования архитектуры тестового фреймворка. С одной стороны необходимо отойти от сложных реализаций, с другой, необходимо грамотно делегировать методы, избежать дублирование кода, и сделать тесты легко поддерживаемыми и читаемыми [29].

Автотесты “системного” уровня, в отличие от “unit-тестов”, удобно разделить на два слоя: первый слой, собственно, сами тесты, второй слой – код, ответственный за взаимодействие с тестируемой системой, причём вторая часть, как правило, является более сложной технически. В результате построения двухуровневой архитектуры, разрабатываемая система автоматизации тестирования приобрела вид, показанный на рисунке 2.12.

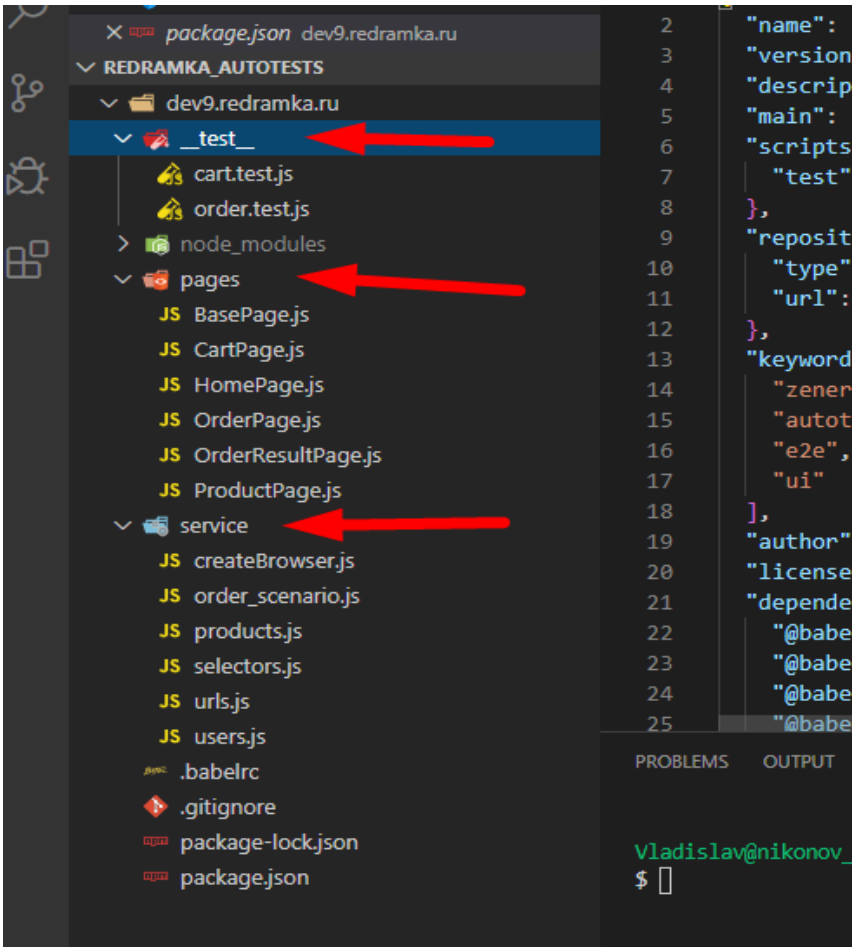

Рисунок 2.12 – Двухуровневая архитектура проекта

В директории «service» содержится код, отвечающий за управление, создаваемыми в процессе тестирования, экземплярами браузеров, а также модули, которые описывают модели пользователя, поведения и селекторы веб-страниц. В директории «pages» содержатся модули, содержащие в себе методы взаимодействия со страницами веб-приложения, две этих части составляют

первый уровень архитектуры, а в директории «test» нет ничего лишнего, кроме самих тестов, он составляет второй уровень двухуровневой архитектуры.

2.4.4 Применение паттерна Page Object и разработка дополнительных методов

Для этого проекта также было решено использовать паттерн Page Object. Основные преимущества PageObject в разделении кода тестов и описания страниц, а также в объединение всех действий по работе с веб-страницей в одном месте.

Для реализации данного паттерна следует для каждой страницы веб-приложения разработать соответствующий модуль, как показано на рисунке 2.13.

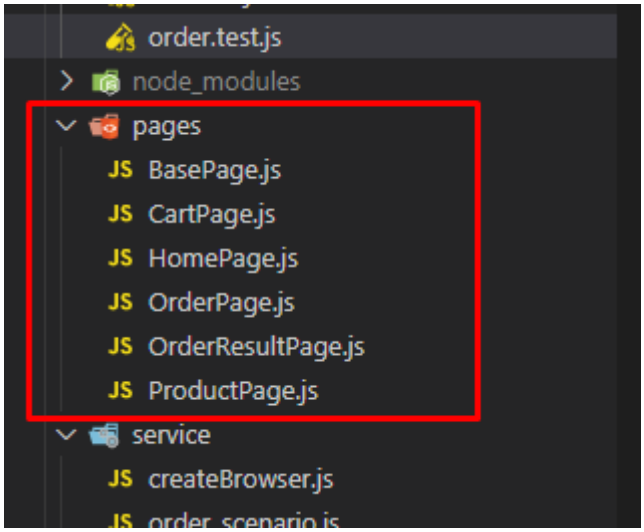

Рисунок 2.13 – Модули описывающие веб-страницы

Опираясь на тестовый сценарий, в директории «pages» были реализованы модули с соответствующими методами, с помощью которых описываются шаги тестового сценария, а также необходимые проверки, явные и неявные ожидания.

В модуле ProductPage.js реализованы методы:

– getEachProductsParams() – данный метод позволяет получить параметры товара из фильтра на странице товара, через атрибуты.

– checkProductWasLoaded()– позволяет дождаться появления списка фильтров, в противном случае программа сообщит, что у данного товара отсутствует список цен.

– selectElemFromProductList(elemNumber) – метод позволяет выбрать элемент фильтра по номеру.

В модуле OrderPage.js реализованы методы для оформления заказа:

– fillOrderForm(user, scenario) – данный метод позволяет полностью заполнить необходимые для осуществления заказа формы, исходя из входных данных. Метод адаптивен к выбору разных способов доставки и оплаты, а данные для заполнения полей клиента, получены из модуля users.js из директории «service».

– isOrderCorrect() – данный метод позволяет удостовериться, что все поля формы были заполнены корректно, и всплывающие уведомления об ошибке отсутствуют.

В модуле OrderResultPage реализованы методы которые позволяют работать с уже сформированным заказом:

– getInfoByOrderNumbers(from, to) – данный метод использует REST API и получает информацию о заказе в текстовом виде, в формате json. Для работы с json-объектами в Java Script, используется метод JSON.parse(). Объект json преобразуется в текстовую строку, а далее разбивается на соответствующие массивы данных и объекты. Номера заказов указываются как входные параметры.

– getOrderNumber() – метод позволяет, в случае успешного формирования заказа, получить номер заказа.

– checkSuccess() – метод проверяет успешное формирование заказа.

Как видно из названий и реализации методов, все они описывают определенный шаг автоматизируемого тестового сценария.

2.4.5 Автоматизация тестового сценария

Для написания автотеста необходимо создать модуль в директории «test», который описывает тестовый набор. Из реализованных методов из директории «pages», в разрабатываемом тесте order.test.js, выстраивается цепочка методов, в соответствии с шагами, указанными в тестовой сценарии. Благодаря логически подобранным названиям методов, код автотеста легко читается и поддерживается, как показано на рисунке 2.14.

```
test.each(products)(
  `Test for %s product`,
  async product => {
    await product_page.openProduct(product);
    await product_page.checkProductWasLoaded();
    const condition = await product_page.ifProductListPresent();
    if (condition) {
      let dataBeforeOrder = await product_page.getEachProductsParams();
      let orderNumbers = [];
      const list_length = await product_page.getListLength();
      for (let i = 1; i <= list_length; i++) {
        await product_page.openProduct(product);
        await product_page.checkProductWasLoaded();
        await product_page.selectElemFromProductList(i);
        await product_page.addToCart();
        await product_page.continueToCard();
        await cart_page.goToOrder();
        await order_page.fillOrderForm(tester_user, scenar);

        expect(await order_page.isOrderCorrect()).toBe(true);

        await order_page.waitSuccessPage();

        expect(await order_result_page.checkSuccess()).toBe(true);

        orderNumbers.push(await order_result_page.getOrderNumber());
      }
      let dataAfterOrder = await order_result_page.getInfoByOrderNumbers(
        orderNumbers[0],
        orderNumbers[orderNumbers.length - 1]
      );

      // сравнение массива с параметрами продукта до заказа и с массивом с параметрами продукта после заказа
      expect(dataAfterOrder).toEqual(dataBeforeOrder);
```

Рисунок 2.14 – Код автоматизированного тестового сценария

Также добавляется логика и проверки. Jest позволяет делать проверки с помощью метода expect(). Как показано на рисунке 3.3, данный метод используется для проверки успешного формирования заказа. Далее следует добавить необходимые проверки на сравнение списка товаров dataBeforeOrder,

с параметрами до оформления заказа, со списком товаров datatAfterOrder с параметрами после оформления заказа, с помощью метода toEqual().

2.5 Выбор стека технологий для браузерной автоматизации

Приступая к автоматизации, каждая команда принимает решение на основе своих знаний или руководствуясь потребностями заказчика. Для автоматизации тестирования пользовательского интерфейса веб-приложений в компании, изначально были выбраны два инструмента: Selenium Web Driver и Puppeteer.

Следует отметить, что оба инструмента справляются с поставленными компанией задачами автоматизации тестирования пользовательского интерфейса, но необходимо, основываясь на имеющимся опыте работы с технологиями и на имеющихся предпочтениях команды разработки, выбрать основной инструмент для использования в разработанном тестовом фреймворке для автоматизации тестирования веб-приложений.

Если сравнивать два инструмента по скорости работы автотестов, то при анализе скорости работы самих автотестов с использованием обоих инструментов было выявлено, что скорость практически не отличается. Это объясняется тем, что Puppeteer использует тот же протокол общения с браузером, что и Selenium Server, это DevTools Protocol. Но скорость работы тестовой сборки с момента запуска до начала работы первого тестового сценария заметно отличается, у Selenium она составляет около десяти секунд, как видно из рисунка 2.15, в то время как при использовании Puppeteer данное время отсутствует.

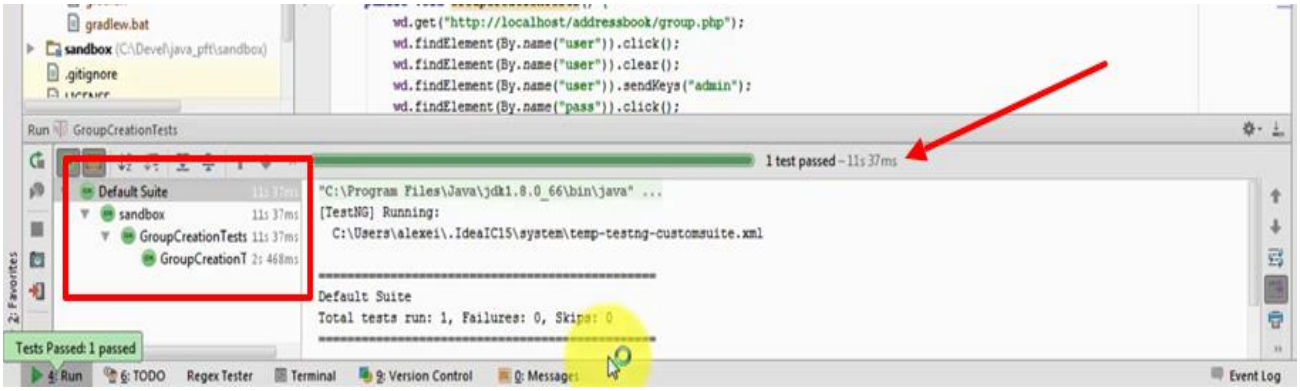

Рисунок 2.15 – Время работы тестового сценария

Это связано с тем, что Puppeteer не использует в схеме своей работы промежуточный элемент – сервер, как это делает Selenium. Схема работы Selenium Web Driver показана на рисунке 1.3.

Такой принцип работы делает Selenium отличным инструментом для кроссбраузерного тестирования [30], что нельзя сказать про Puppeteer, схема работы которого показана на рисунке 1.4. В отличии от Selenium Webdriver, коммуникация происходит непосредственно с браузером, хотя и через тот же Chrome DevTools Protocol, который использует ChromeDriver Selenium.

Особенность заключается в том, что Puppereer развивается и обновляется намного динамичнее, чем это происходит с ChromeDriver Selenium. Надо заметить, что Google уже достаточно давно не участвует в разработке Selenium, даже в качестве спонсора. И ChromeDriver обновляет очень редко, в том числе долго не исправляет критические баги. Способствует переходу от кроссбраузерной автоматизации в сторону "chrome only", что, необходимо заметить, поддерживается в текущей компании.

Выбор первого инструмента Selenium, основывался на имеющимся опыте автоматизации тестирования на языке Java с использованием тестового фреймворка TestNG и сборщика проектов Gradle. Данный стек технологий плохо отвечал требованиям компании таким как: поддерживаемость кода автотестов и быстрое введение неопытных специалистов в разработку автотестов.

Первое несоответствие объясняется тем, что компания не использует Java в своих проектах и нет разработчика, который смог бы поддерживать код автотестов в случае отсутствия специалиста. Также следует учесть, что имеющийся опыт разработки не большой и в случае возникновения проблем, разработчику будет тяжело искать ответы у коллег.

Второе несоответствие объясняется спецификой самого языка. Синтаксис языка программирования Java нельзя назвать сложным, но он громоздкий и имеет длинные конструкции кода, что увеличивает время на разработку, а также требует хорошего понимания ООП для написания качественного кода. Начинающему специалисту будет сложно быстро внедрится в процесс разработки.

Из положительных сторон следует выделить хорошую поддержку ООП языка Java, что позволяет спроектировать и реализовать более гибкую и модульную двухуровневую архитектуру разрабатываемого тестового фреймворка, с другой стороны усложняющего его. Это сказывается на том, что время на подготовку структуры проекта тратится больше, чем на написание самих тестов.

Выбор второго инструмента Puppeteer и использование платформы Node.js объясняется тем, что команда имеет опыт работы с данной платформой, а также использованием JavaScript одним из основных языков разработки в компании, что полностью соответствует такому требованию компании, как поддерживаемость кода.

Что касается быстрого внедрения начинающих QA специалистов в процессы написания автотестов, то использования инструментов на платформе Node.js показало положительные результаты. Структура директорий и модулей разрабатываемого фреймворка значительно проще чем на стеке Java, что упрощает автоматизацию разворачивания и конфигурирования разрабатываемого фреймворка. Начинающему специалисту требуется лишь запустить bash-скрипт, заполнить необходимые данные в конфигурационном

файле и приступить к разработке автотестов по уже имеющимся шаблонам и регламенту.

В результате можно сделать вывод, что для автоматизации тестирования в компании будет использоваться платформа Node.js и такие инструменты для автоматизации тестирования как Puppeteer и тестовый фреймворк Jest, так как они соответствуют основным требованиям компании: простоте поддерживаемости кода автотестов и быстрому внедрению в процесс разработки QA специалистов с небольшим опытом автоматизации. Помимо этого, данные инструменты автоматизации имеют положительные оценки скорости и стабильности работы самих автотестов и тестовых сборок.

Но при необходимости кроссбраузерного тестирования, целесообразно использование инструмента Selenium WebDriver на платформе Node.js, что позволит, не меняя платформу и язык разработки проводить автоматизированное тестирование продуктов компании, используя различные браузеры. Одним из инструментов, позволяющих это реализовать, это Protractor, который является надстройкой над Selenium. Но данный вариант является второстепенным.

2.6 Внедрение инструмента непрерывной интеграции Jenkins

Jenkins – это инструмент автоматизации с открытым исходным кодом, написанный на Java, с плагинами, созданными для непрерывной интеграции. Jenkins используется для непрерывной сборки и тестирования программных проектов, что облегчает разработчикам интеграцию изменений в проект и облегчает пользователям получение новой сборки. Это также позволяет вам непрерывно поставлять программное обеспечение, интегрируя с большим количеством технологий тестирования и развертывания [31].

С помощью Jenkins организации могут ускорить процесс разработки программного обеспечения за счет автоматизации. Jenkins объединяет процессы жизненного цикла разработки всех видов, включая сборку,

документацию, тестирование, пакет, этап, развертывание, статический анализ и многое другое.

Несмотря на то, что для хранения проектов в компании используется сервис Gitlab, который уже включает в себя инструмент непрерывной интеграции, было решено вынести автоматизацию проектов в отдельный модуль. Но это не основная причина, дело в том, что в Gitlab создание сборки определенного этапа разработки происходит с помощью конфигурационного файла «.gitlab-ci.yml». В данном файле, описываются необходимые для сборки команды и пути к директориям, с использованием специфичной логики и структуры. Начинающего специалиста будет сложно быстро ввести в проект. В Jenkins конфигурационный файл не используется, создание и настройка сборки происходит через интерфейс сервиса, интуитивно понятного для специалиста любого уровня. Также, из преимуществ, Jenkins включает в себя:

- Большая поддержка сообщества, так как это инструмент с открытым исходным кодом.
- Тривиальная установка и первичная настройка.
- Имеет более тысячи плагинов для облегчения работы. Если плагин не существует, есть возможность написать плагин под свои нужды и поделиться с сообществом.
- Jenkins бесплатен.
- Он кроссплатформенный и работает на всех основных платформах.

В текущем процессе разработки, Jenkins используется только для непрерывной интеграции автоматизированного тестирования. Планируется создание нескольких основных сборок для каждого проекта:

1. Сборка для автоматизированного smoke тестирования.
2. Сборка для автоматизированного регрессионного тестирования.
3. Сборки для специфичных автоматизированных тестовых задач, которые запускаются вручную.

Условия запуска сборок, следующие:

1. Изменение в репозитории тестируемого проекта.

2. Изменение в репозитории, где находится код автотестов для тестируемого проекта.

3. Запуск по расписанию.

2.5.1 Конфигурирование Jenkins

Jenkins достигает непрерывной интеграции с помощью плагинов. Плагины позволяют интегрировать различные этапы DevOps. Если необходимо интегрировать определенный инструмент, нужно установить плагины для этого инструмента. Для установки плагинов следует перейти на страницу «Настроить Jenkins», далее в «Управление плагинами» и во вкладке «Дополнительные» выбрать необходимые для установки плагины. Изменения, вызванные установкой новых плагинов, вступят в силу после перезагрузки Jenkins.

Для создания сборки для текущего проекта необходимо установить «NodeJS Plugin» для возможности сборки проекта на платформе Node.js. Для настройки данного плагина следует перейти на страницу «Конфигурация глобальных инструментов», если на сервере, где установлен Jenkins уже есть установленный node, и данная версия совместима с автотестами, то можно в конфигурации указать путь к текущей установленной версии в качестве локальной. Также имеется возможность создать несколько конфигураций для разных версий, необходимая версия выбирается в селекте, как показано на рисунке 2.16.

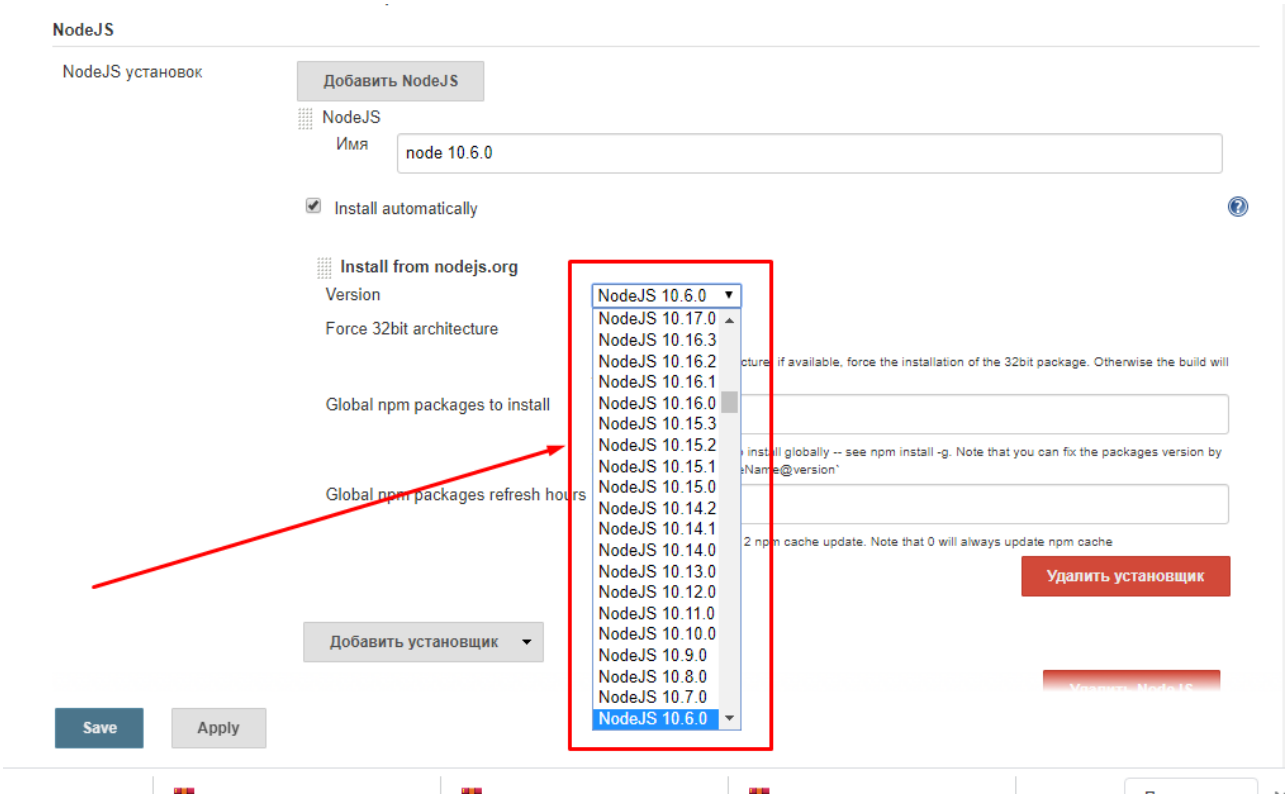

Рисунок 2.16 – Конфигурация плагина NodeJS

Также необходим плагин «Allure Jenkins plugin» для создания отчетов о пройденных сборках. Его настройка не требуется, он сразу готов к использованию при конфигурации сборки.

Для сборки требуется удаленно стянуть проект из репозитория Gitlab. В Jenkins есть несколько плагинов для работы с git репозиториями: «Git plugin», «Gitlab Plugin», «Github Plugin». Два последних более специфичны в настройках, для конкретных сервисов. Для корректной работы git требуется плагин «Git», а для задач, связанных с обменом данными с сервисом Gitlab, используется плагин «Gitlab Plugin». «Git plugin» настраивается аналогично, необходимо указать локальную версию git, заранее установленную на сервере, как показано на рисунке 2.17.

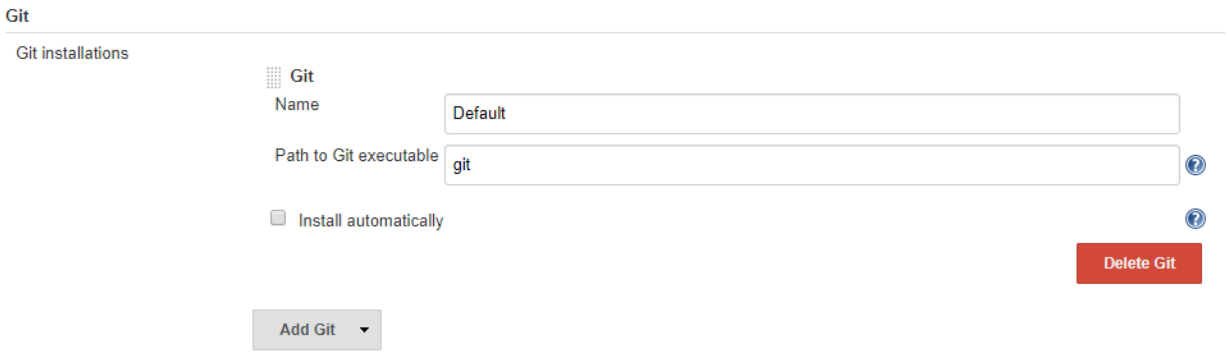

Рисунок 2.17 – Настройка плагина Git

2.5.2 Создание в Jenkins задач сборки и запуска разрабатываемых автотестов

После установки всех необходимых плагинов и настройке их конфигураций можно приступать к созданию сборки.

Необходимо перейти на страницу создания сборки, где будет предложено ввести имя сборки и выбрать вариант конфигурации. Для текущих проектов выбрана конфигурация «Создать задачу со свободной конфигурацией», как показано на рисунке 2.18.

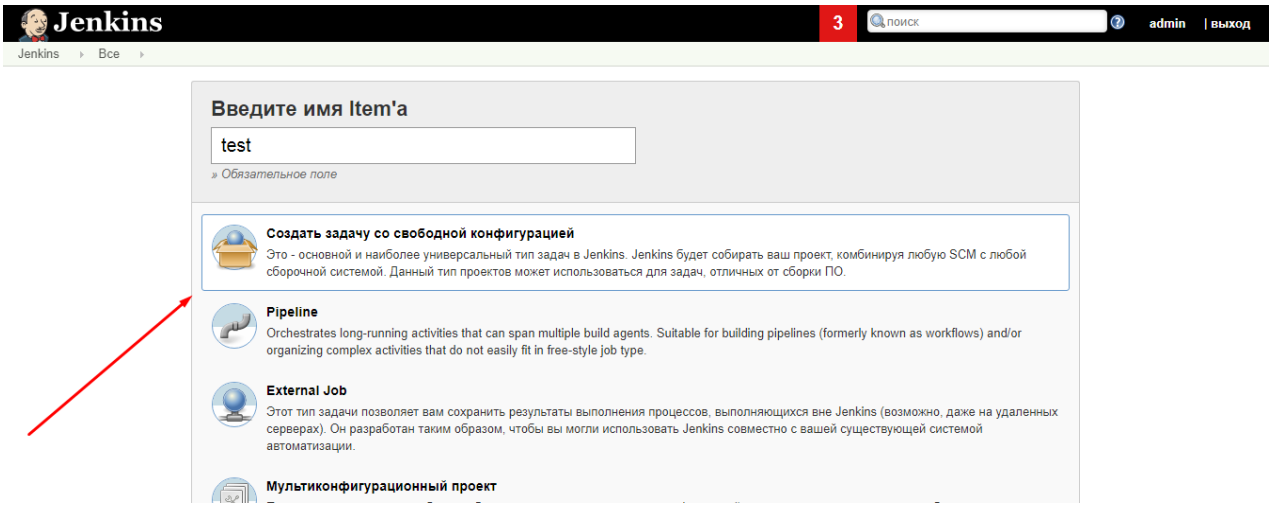

Рисунок 2.18 – Меню создания сборки в Jenkins

Сама конфигурация сборки делится на несколько частей: «Общая конфигурация», «Управление исходным кодом», «Триггеры сборки», «Среда сборки», «Сборка», «Послесборочные операции», как показано на рисунке 2.19.

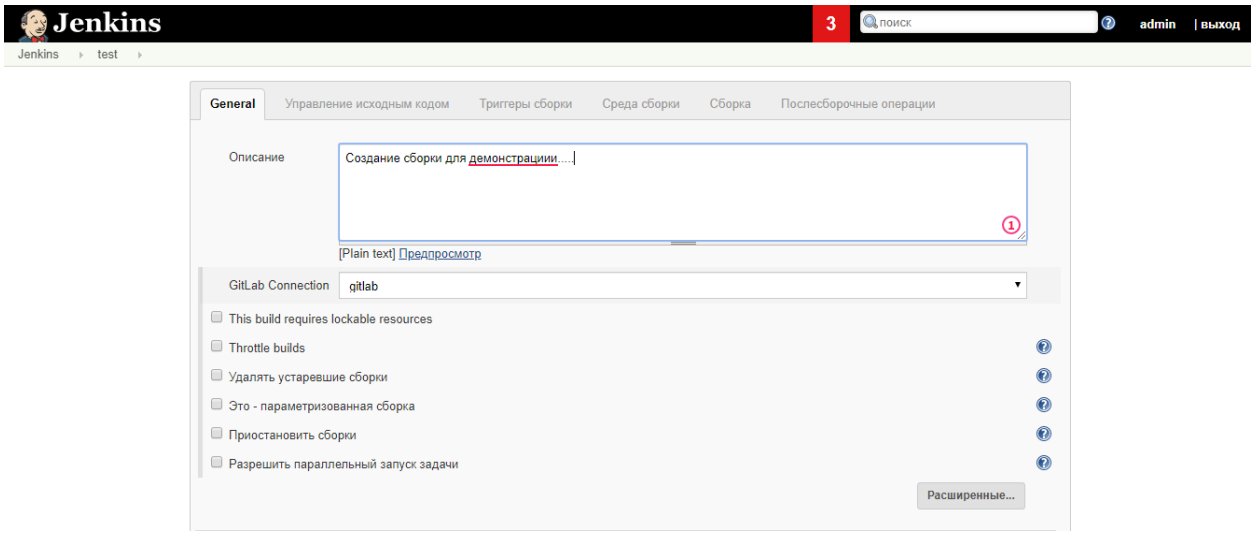

Рисунок 2.19 – Форма общей конфигурации сборки

В «Общей конфигурации» следует указать описание сборки, а также настроить удаление устаревших сборок, чтобы не копить их и не занимать место на жёстком диске сервера.

Далее следует «Управление исходным кодом». В данной вкладке указывается название удаленного репозитория, а также, если данный репозиторий приватный, следует указать данные для доступа в поле «Credentials».

Как правило проект имеет несколько веток в репозитории, необходимо указать какая именно ветка будет стянута и впоследствии протестирована, данную информацию указывают в поле «Branch Specifier», как показано на рисунке 2.20.

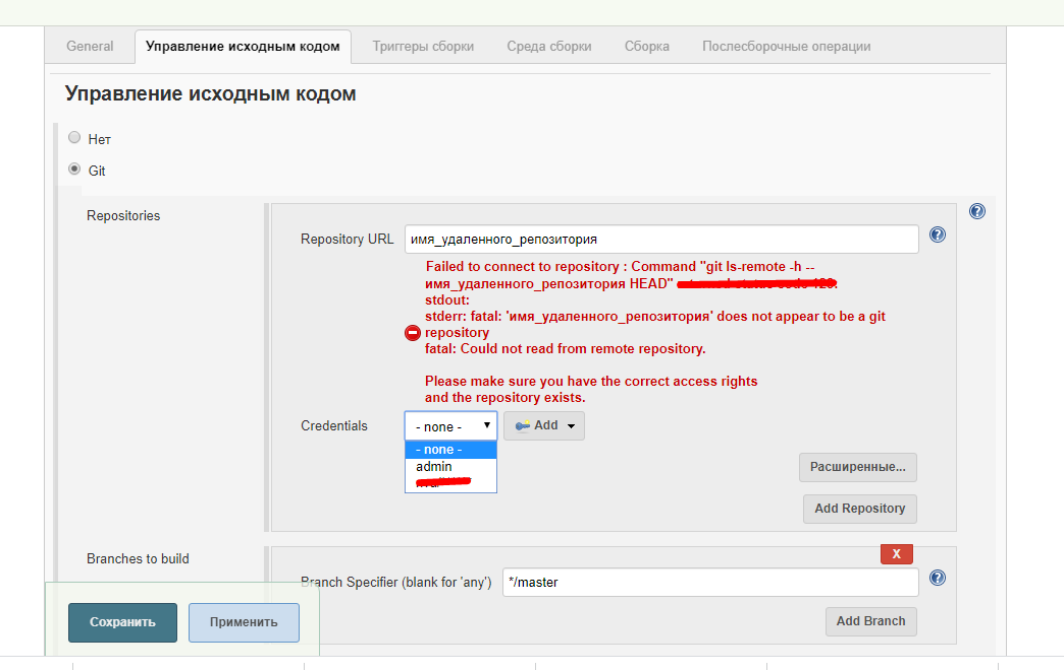

Рисунок 2.20 – Форма конфигурации управления исходным кодом

Далее следует вкладка «Триггеры сборки». Как было сказано выше сборка будет начинаться при изменении проекта в удаленных репозиториях Gitlab. Для обеспечения связи используется технология «Webhooks» [32]. Как показано на рисунке 2.21, следует скопировать предложенный Jenkins webhook.

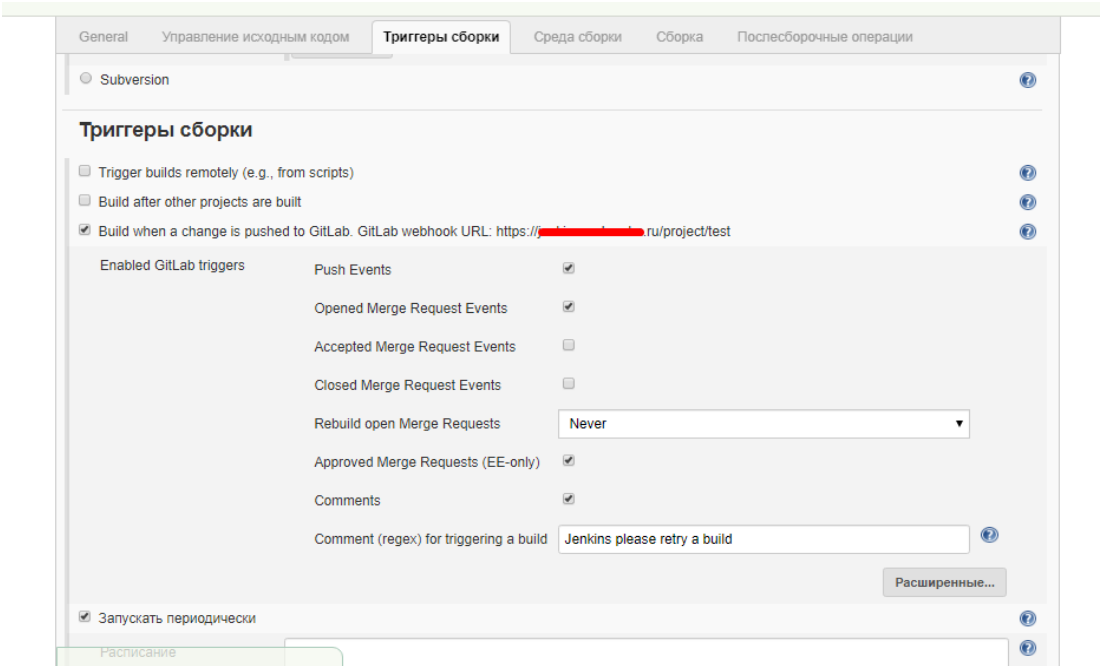

Рисунок 2.21 – Форма конфигурации триггеров сборки

Далее уже в самом сервисе Gitlab, на странице «Integrations», создается webhook в поле «url» которого вставляется, скопированное из Jenkins значение, как показано на рисунке 2.22. Также следует выбрать событие, которое будет выступать в качестве триггера для Jenkins.

Для данного проекта выбран «Push events», это значит, что сборка в Jenkins запустится, как только разработчик выполнит команду «push» для того, чтобы сохранить сделанные им изменения в репозиторий проекта.

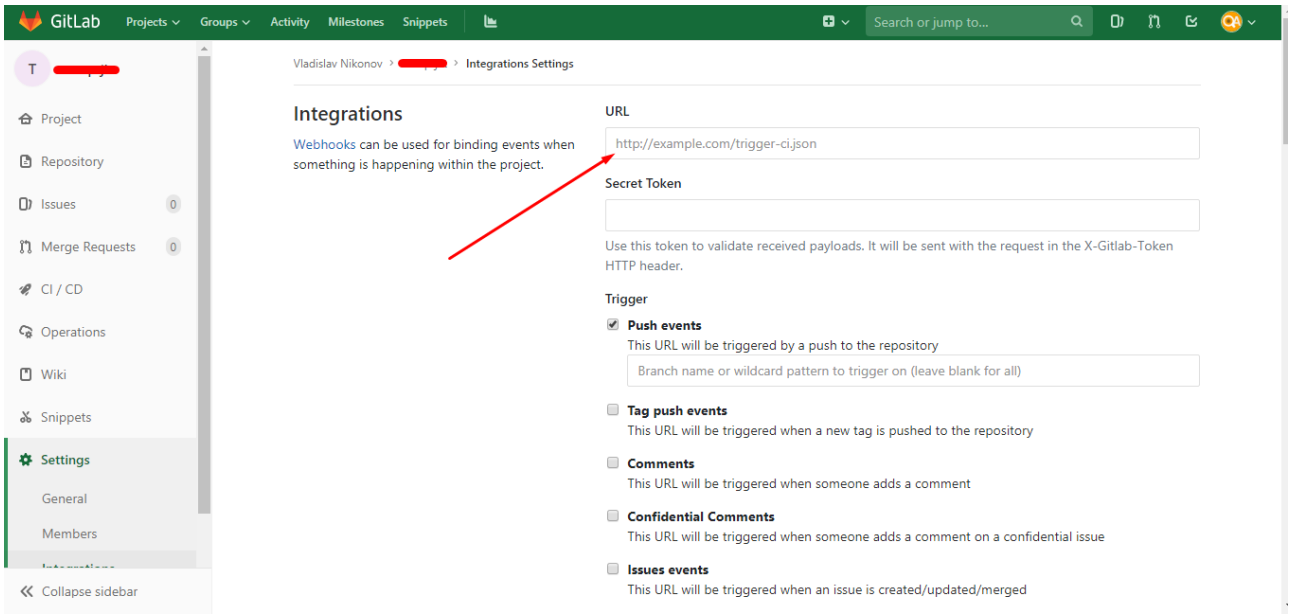

Рисунок 2.22 – Страница создания webhook в сервисе Gitlab

Следующий шаг – это «Среда сборки». Так как был установлен плагин «NodeJS Plugin», в списке доступных сред для сборки появился пункт «Provide Node & npm bin/folder to PATH». Далее выбирается подходящая под текущий проект версия node, как показано на рисунке 2.23.

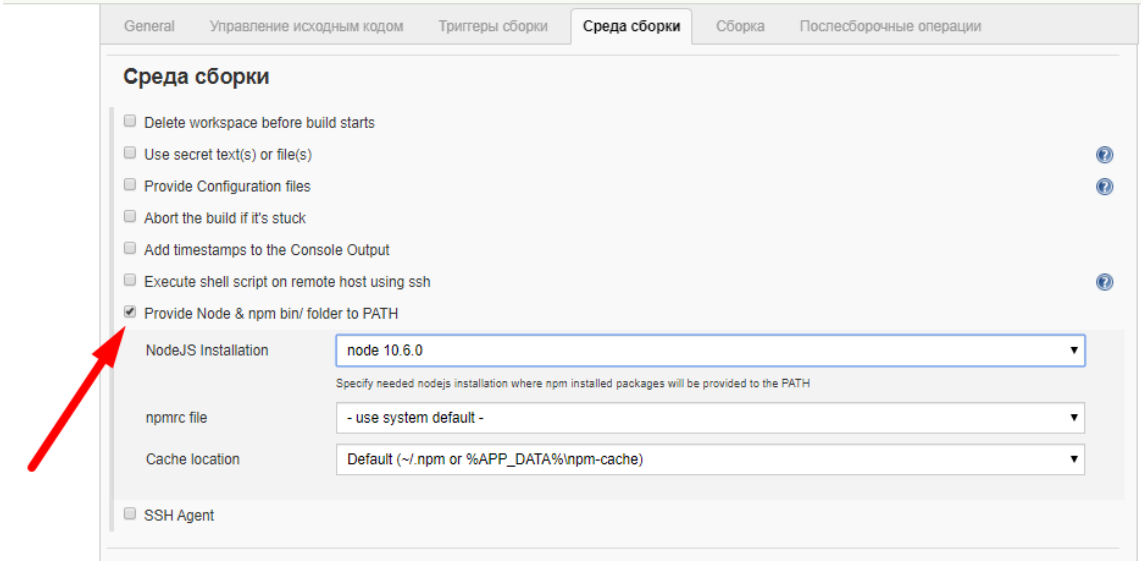

Рисунок 2.23 – Форма конфигурации среды сборки

На этапе «Сборка» прописываются команды, необходимые для работы уже с загруженным с репозитория проектом. Так как в проекте используется пакетный менеджер npm, то для работы с ним необходим терминал shell, и из предложенного списка выбираем именно его. В поле ввода прописываем необходимые команды для загрузки необходимых библиотек, headless браузера и запуска автотестов, как показано на рисунке 2.24.

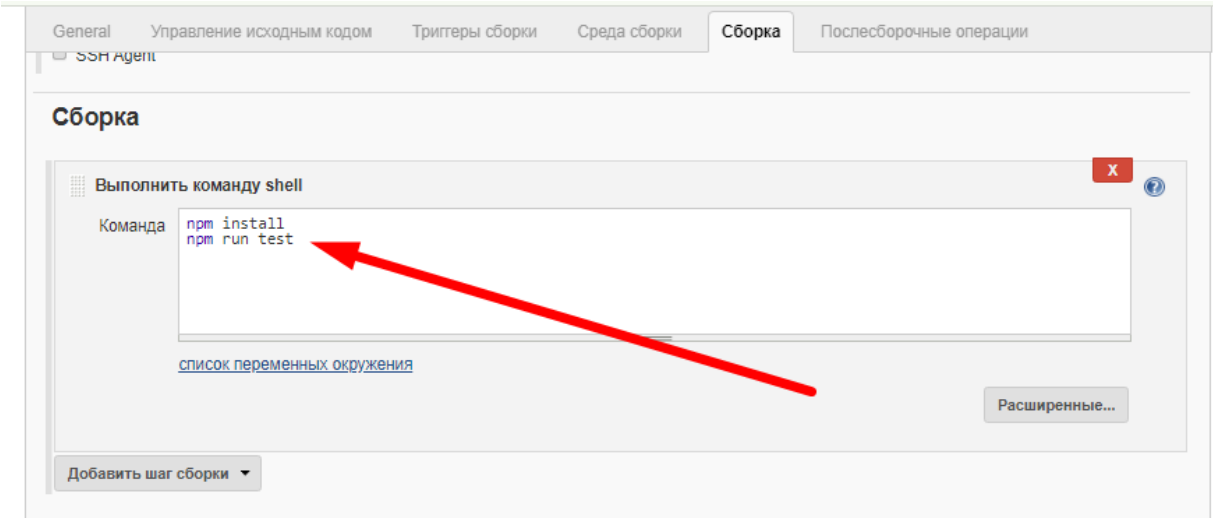

Рисунок 2.24 – Командная строка для сборочных операций проекта

И последним этапом является «Послесборочные операции», где есть возможность выбора различных событий, которые будут происходить если

сборка пройдет неудачно. Планируется использование рассылки электронных писем специалистам по качеству, в случае если сборка завершиться с ошибкой. Также можно создавать очередь сборок, где запуск каждой последующих сборок, зависит от предыдущей. Данные функции показаны на рисунке 2.25.

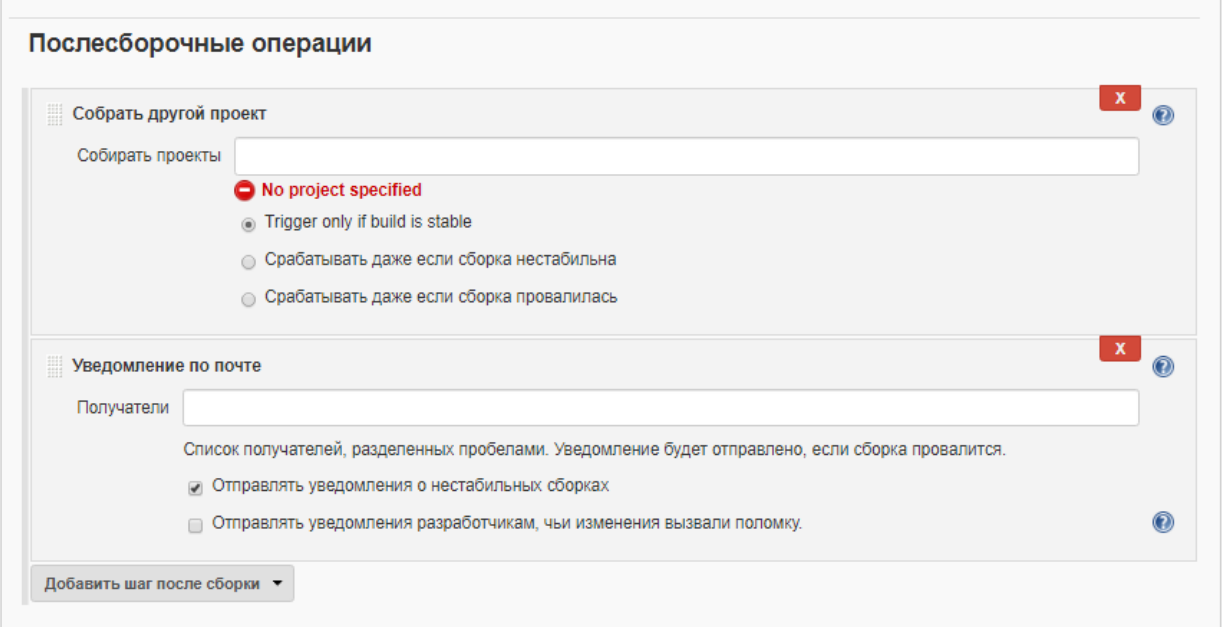

Рисунок 2.25 – Форма настройки послесборочных операций

2.5.3 Автоматическая генерации отчета о пройденных сборках с использованием Allure

Allure Framework – популярный инструмент построения отчётов автотестов, упрощающий их анализ. Это гибкий и легкий инструмент, который позволяет получить не только краткую информацию о ходе выполнения тестов, но и предоставляет всем участникам производственного процесса максимум полезной информации из повседневного выполнения автоматизированных тестов [33].

Разработчикам и тестировщикам использование отчетов Allure позволяет сократить жизненный цикл дефекта: падения тестов могут быть разделены на дефекты продукта и дефекты самого теста, что сокращает затраты времени на анализ дефекта и его устранение. Также к отчету могут быть прикреплены логи, обозначены тестовые шаги, добавлены вложения с разнообразным контентом,

получена информация о таймингах и времени выполнения тестов. Кроме того, Allure-отчеты поддерживают взаимодействие с системами непрерывной интеграции и баг-трекинговыми системами, что позволяет всегда держать под рукой нужную информацию о прохождении тестов и дефектах.

Для использования Allure в Jenkins необходим плагин «Allure Jenkins plugin». Он не требует глобальных конфигураций, необходимые опции появляются при создании сборки на этапе «Послесборочные операции», как показано на рисунке 2.26.

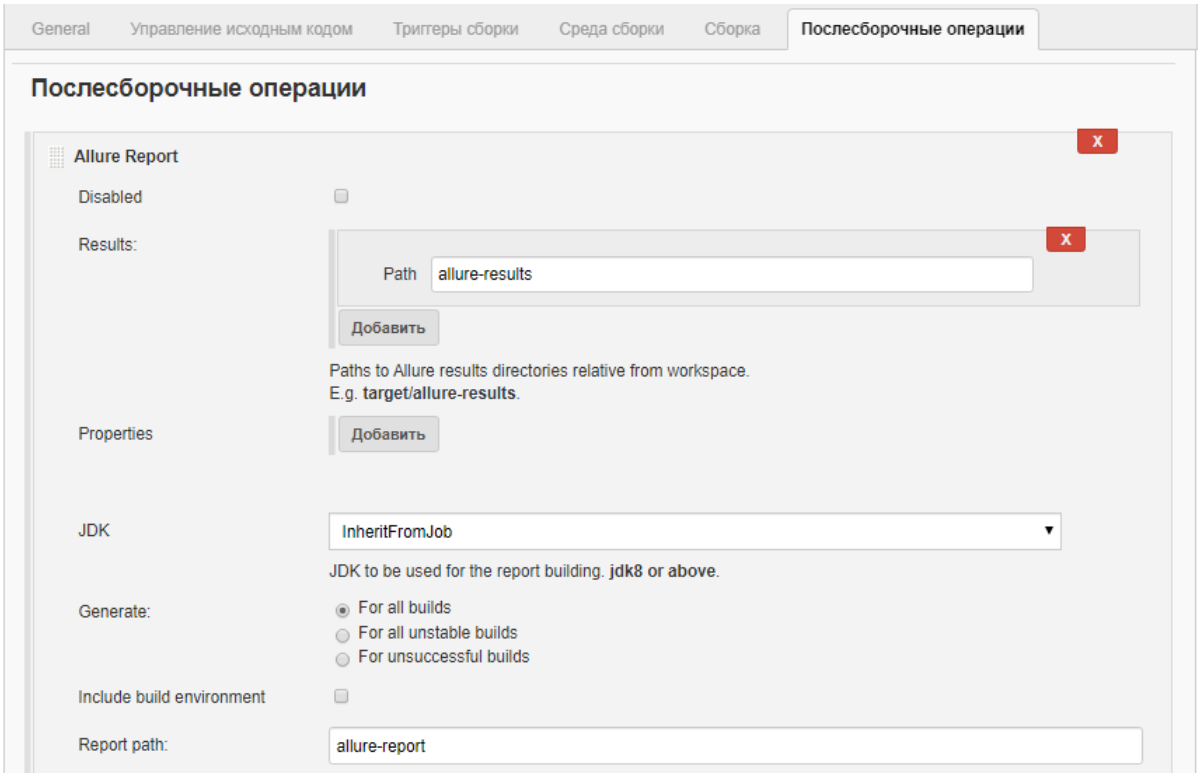

Рисунок 2.26 – Настройка Allure во время конфигурации сборки

Для конфигурации создания отчета требуется указать директорию куда будут создаваться сами отчеты о пройденных автотестах, а также, директорию, где будет сохранятся история отчетов. Пример отчета показан на рисунке 2.27.

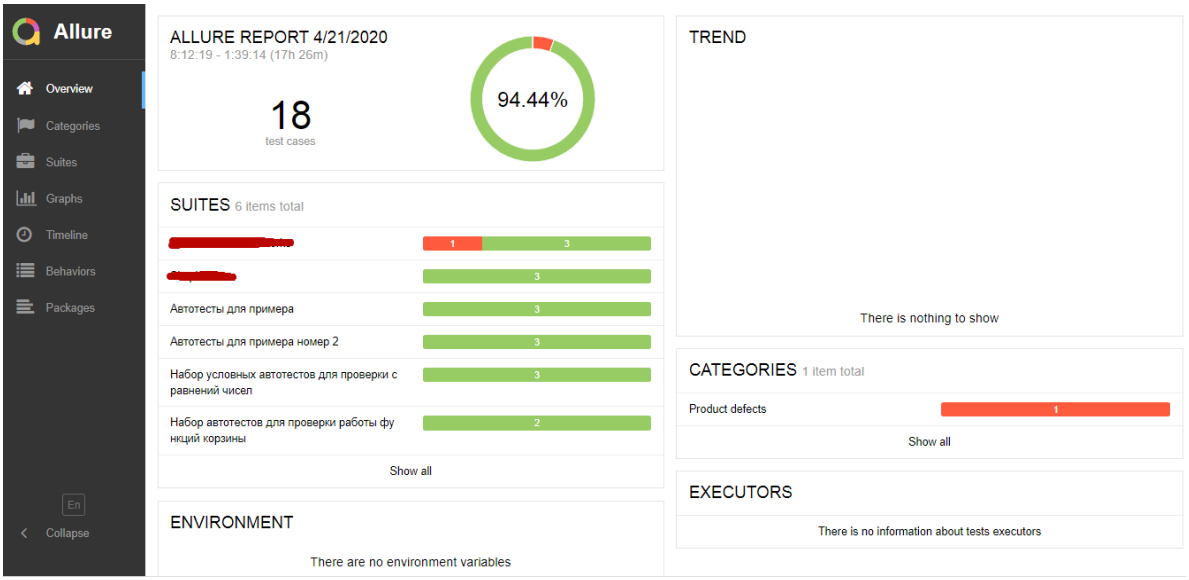

Рисунок 2.27 – Пример сгенерированного отчета Allure

Генерация отчетов будет происходить для всех сборок, независимо от успеха. Это позволит отслеживать наиболее уязвимые модули, что позволить контролировать покрытие разрабатываемое приложение автотестами, а также наглядно видеть историю стабильности, как показано на рисунке 2.28.

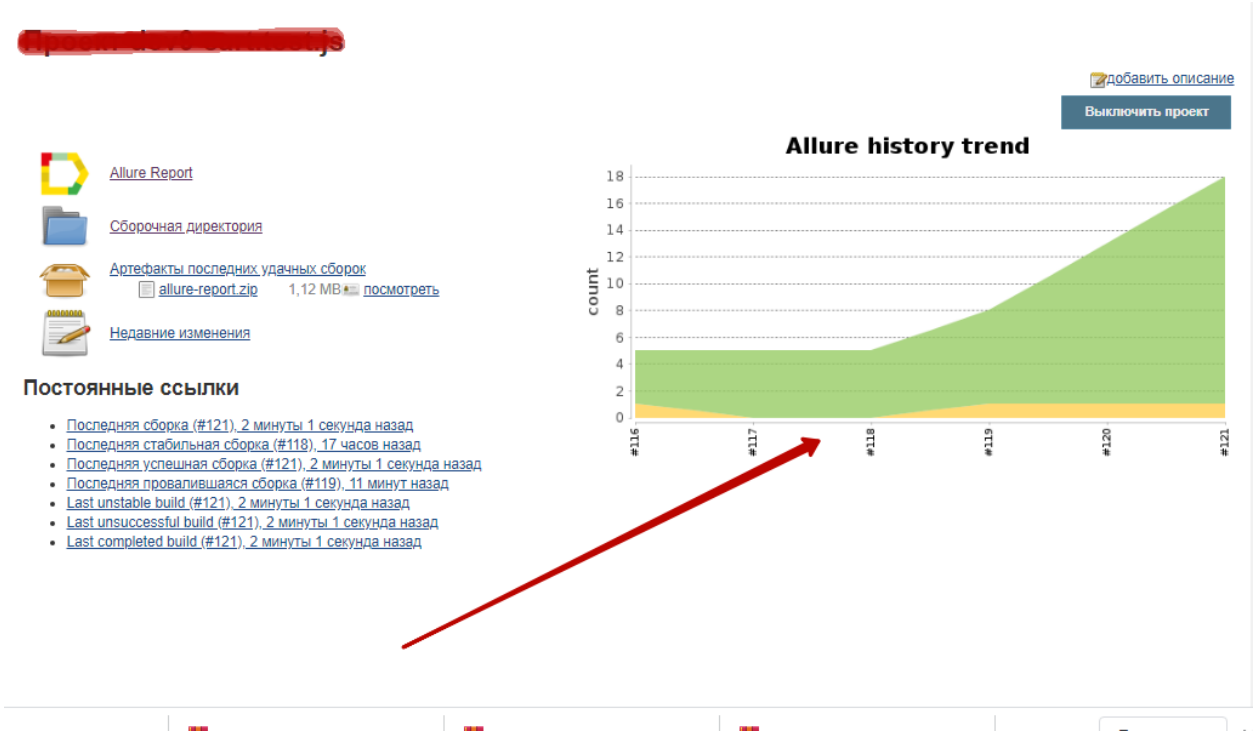

Рисунок 2.28 – Статистика прохождения тестовых сборок

2.5.4 Создание аннотаций и описания в коде автотестов

С использованием тестового фреймворка Jest появляется возможность описывать автотесты не только на уровне разбиения по тест-suit и по тест-кейсам. Данный фреймворк позволяет использовать глобальную переменную «reporter» которая в модуле «jest-allure» [34]. Для того что бы использовать данную переменную необходимо импортировать модуль следующим образом – «import "jest-allure/dist/setup"».

```
61
62  describe("Автотесты для примера", () => {
63    let l = 5;
64    test("fПервый тест", async () => {
65      reporter.feature("Здесь название тестируемой функциональности номер 1");
66      reporter.description("Здесь описание тестового сценария");
67      reporter.severity(Severity.Trivial);
68      reporter.addLabel("Status","BUG"); // можно добавить лейбл
69      reporter.startStep("Название шага тестового сценария: 5 должно быть больше чем 4");
70      expect(l > 4).toBe(true);
71      reporter.endStep("Конец шага");
72    }, 15000);
73    test("Второй тест", async () => {
74      reporter.feature("Здесь название тестируемой функциональности номер 2");
75      reporter.startStep("Название шага тестового сценария: 5 - 4 должно быть равно 1");
76      expect(l - 4).toEqual(1);
77      reporter.endStep("Конец шага");
78    }, 15000);
79    test("Третий тест", async () => {
80      reporter.feature("Здесь название тестируемой функциональности номер 3");
81      reporter.startStep("Название шага тестового сценария: 5 +1 должно быть равно 6");
82      expect(l + 1).toEqual(6);
83      reporter.endStep("Конец шага");
84    }, 15000);
85  });
```

Рисунок 2.29 – Пример добавления аннотаций и описания автотеста в коде

Переменная «reporter» позволяет описать автотест как обычный тест-case. Как показано на рисунке 2.29, в автотест добавлены: описание функциональности которую проверяет автотест, с помощью функции «feature», с помощью функции «description» добавлено описание самого автотеста, возможность добавить серьезность и приоритет автотеста с помощью функций «severity» и «priority», также есть возможность разделения кода отдельных смысловых блоков автотеста на отдельные описываемые шаги.

Все эти функции позволяют отобразить полноценный allure-отчет, который будет понятен уже не только техническим специалистам, но также менеджерам и специалистам по качеству низкого уровня подготовки. В конечном итоге, на основе данных описания автотестов, генерируемый отчет приобретает вид, показанный на рисунке 2.30.

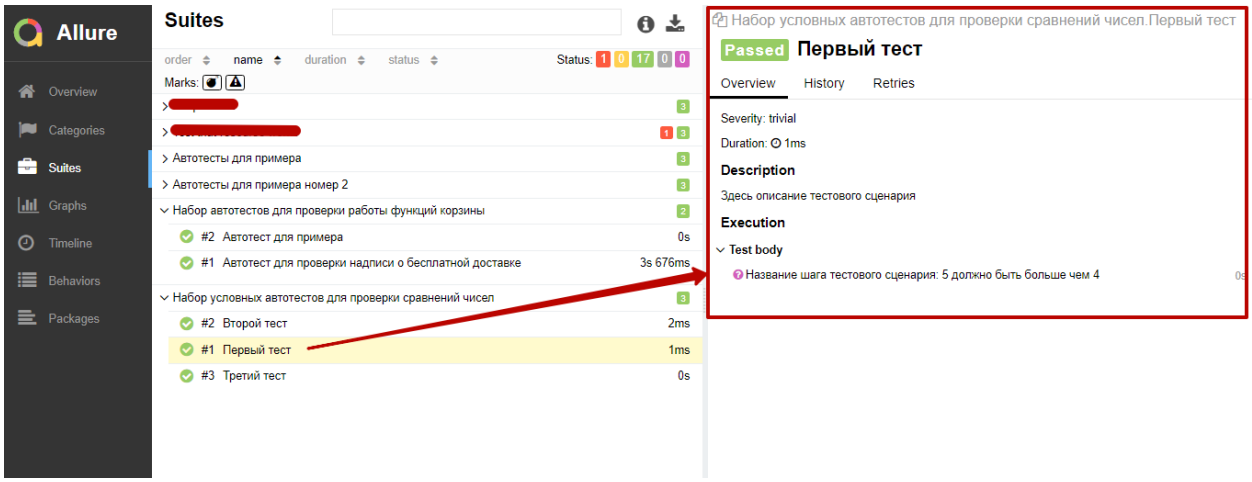

Рисунок 2.30 – Подробная информация о тесте в генерируемом отчете

3 Автоматизация разрабатываемого тестового фреймворка

Разработанный тестовый фреймворк представляет собой определенную структуру директорий и базовых модулей, изображенную на рисунке 3.1, которую можно использовать как шаблон для автоматизации отдельных проектов. Для полноценной работы с фреймворком, на данный момент, необходимо скопировать структуру и минимально переработать ее под конкретный проект. Эта тривиальная задача для опытного разработчика, но специалист без определенных знаний и опыта не справиться с ней.

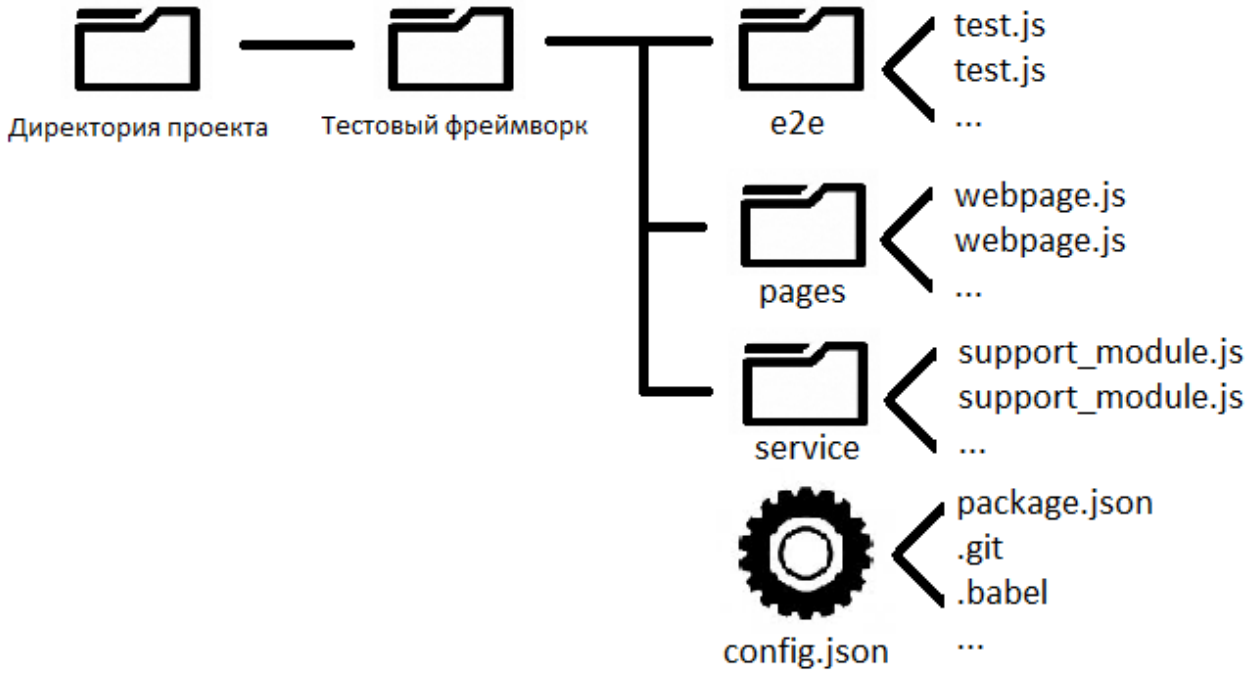

Рисунок 3.1 – Структура разрабатываемого тестового фреймворка

Безусловно, необходимо стремиться к развитию навыков и умений начинающих специалистов, но также требуется создать условия, в которых начинающий специалист сможет в кратчайшие сроки начать разрабатывать автотесты. Для этого была поставлена задача автоматизации процесса конфигурирования и разворачивания проекта, а также создания регламента разработки и описания автотестов.

3.1 Разработка bash-скрипта для автоматизации разворачивания тестового фреймворка

Для автоматизации конфигурирования и разворачивания разрабатываемого тестового фреймворка используется bash-скрипт [35]. Это актуальное решение, так как в основном разработчики используют Linux в качестве операционной системы, но даже если используется Windows, в большинстве интегрированных сред разработки есть встроенный bash терминал.

Bash-скрипты – сценарии командной строки, написанные для оболочки bash. Существуют и другие оболочки, например – zsh, tcsh, ksh, но выбор был сделан в пользу bash, из-за его распространенности.

Сценарии командной строки – это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии – это мощный способ автоматизации часто выполняемых действий.

Для начальной конфигурации и автоматического создания структуры директорий разрабатываемого тестового фреймворка был создан сценарий, код которого приведен в приложении А.

Строка «#!/bin/bash» указывает системе на то, что сценарий создан именно для bash. В других строках этого файла символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает.

Далее скрипт используя команды «echo -n "Enter project name:" / read project» предлагает пользователю ввести название проекта для которого будут разрабатываться автотесты.

Команды «mkdir $project, cd $project, mkdir ./test, mkdir ./test/e2e, test/pages, test/service, test/rest» позволяют создать общую папку название

которой определяется пользователем, а также структуру директорий тестового фреймворка.

Далее командами «touch ./test/service/urls.js, touch ./test/service/selectors.js» создаются стандартные модули для описания необходимых селекторов, используемых при разработке и адресов страниц веб-приложения. Как видно из кода bash-скрипта, данные файлы заполняются необходимой информацией.

Также создается файл «.gitignore», в который также записываются начальные директории и файлы, которые не должны попасть в репозиторий в целях безопасности.

В заключении срабатывают самые важные команды, отвечающие за инициализацию и конфигурацию проекта с помощью npm. В коде скрипта указаны все модули, необходимые для дальнейшей разработки, которые будут автоматически загружены, также будет загружен headless-браузер для работы с инструментом Puppeteer.

В итоге, после успешного окончания работы bash-скрипта, создана структура проекта для автоматизации тестирования с необходимыми модулями и конфигурационными файлами, схематично изображенная на рисунке 3.1. Данное решение позволяет взглянуть на разрабатываемый тестовый фреймворк, ни как на шаблон структуры проектов и файлов, который применяется из проекта к проекту, а как на полноценный инструмент автоматизации тестирования, применяемый в компании.

3.2 Добавление других видов тестов в разрабатываемый тестовый фреймворк

Помимо UI тестов, необходимо добавить в тестовый фреймворк возможность разработки автотестов для тестирования API, а также добавления, так называемых тестов общего назначения.

Для реализации API тестов, или тестов общего назначения, как правило не нужно взаимодействовать с элементами графического интерфейса веб-

приложений. Необходимы возможности создания запросов к серверу веб-приложения, а также возможность парсинга html кода страницы приложения [36].

Для парсинга была выбрана библиотека Cheerio. Cheerio позволяет работать со скачанными из сети данными, используя синтаксис, аналогичный jQuery. Это быстрый, гибкий и надёжный порт jQuery, разработанный специально для сервера [37]. Использование Cheerio позволяет сконцентрироваться непосредственно на работе с полученными данными, а не на их парсинге.

3.2.1 Обзор библиотек для осуществления REST запросов

Одна из важнейших задач, которую приходится решать разработчику при работе с веб-проектами, заключается в организации обмена данными между клиентскими и серверными частями таких проектов. Для данных целей существуют такие библиотеки как: Axios, Request, Fetch.

Библиотека **Axios**, предназначенная для выполнения HTTP-запросов, основана на промисах [38]. Она подходит для использования в среде Node.js и в браузерных приложениях. Библиотека поддерживает все современные браузеры, и, в том числе, IE8+.

Положительные стороны данной библиотеки:

- Работает в среде Node.js и в браузерах.
- Поддерживает промисы.
- Позволяет выполнять и отменять запросы.
- Позволяет задавать тайм-аут ответа.
- Поддерживает защиту от XSRF-атак.
- Позволяет перехватывать запросы и ответы.
- Поддерживает индикацию прогресса выгрузки данных.
- Широко используется в проектах, основанных на React и Vue.

Но данной библиотекой довольно сложно пользоваться.

Библиотека **Request**, представляет собой упрощённое средство для выполнения HTTP-запросов [39]. При использовании этой библиотеки приходится писать меньше кода, чем при работе с другими библиотеками. Она не использует промисы, но, если эта возможность необходима, можно воспользоваться библиотекой Request-Promise [40], реализующей обёртку вокруг библиотеки Request и позволяющей работать с промисами. Имеет API, которым легко пользоваться.

Fetch – это, в отличие от других средств, не библиотека [41]. Это стандартное браузерное API, являющееся альтернативой XMLHttpRequest.

Положительные стороны данной библиотеки:

- Гибкость и простота в использовании.
- Применение промисов.
- Поддержка всеми современными браузерами.
- Следование подходу «запрос – ответ».
- Простой и приятный синтаксис.
- Поддерживается в React Native.

Отрицательные стороны:

- Не работает в серверной среде.
- Не реализует некоторые возможности, имеющиеся в HTTP-библиотеках, такие, как отмена запроса.
- Не содержит встроенной поддержки параметров, задаваемых по умолчанию, наподобие режима запроса, заголовков, учётных данных.

В результате работы с данными библиотеками, было выявлено, что в контексте поставленных задач, библиотеки не отличаются по функциональности. Поэтому для дальнейшей работы была выбрана библиотека Axios, как, субъективно, наиболее удобная в использовании.

3.2.2 Разработка API теста

При разработке API тестов используется библиотека для http/https запросов Axios [42]. Создаётся директория «api» и поддиректория «service», как показано на рисунке 3.2.

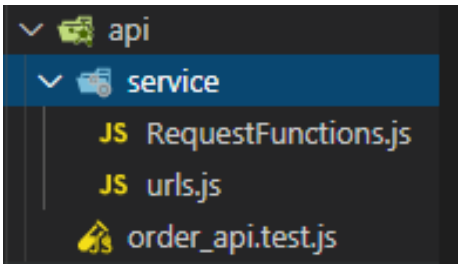

Рисунок 3.2 – Структура директории «api»

В директории «service» содержаться: модуль «urls.js» - данный модуль содержит ссылки применимые для определенного проекта, и дополнительные модули, которые содержат классы с функционалом для соответствующих API тестов.

Для разработки автотеста для тестирования API было выбрано API интернет-магазина. Используя данное API, можно получить от сервера ответ в формате json [43], с информацией о заказах клиентов, где отображены данные о покупателе, данные о содержимом заказа, а также данные о состоянии заказа. Разработанный тест направлен на проверку того, что API возвращает не пустые данные, а также что сами данные корректны. Код самого теста показан в приложении Б.

Как видно из кода теста, данные о позиции из корзины заказа, получены с помощью функции «getOrderItemInfo(url, order_number)». Именно в этой функции используется API интернет-магазина. Для разработки функций получения данных по API, был создан класс «RequestFunctions.js», в котором описано взаимодействие с API интернет-магазина:

«getOrdersList(url,order_num1,order_num2 = order_num1)» – данная функция возвращает список заказов в виде обьектов с информацией. Входные

параметры: url – ссылка ресурса, API которого используется; order_num1 и order_num2 – номера заказов, с какого по какой заказ необходимо вывести данные;

«getOrderItemInfo(url,order_num1)» – данная функция, используя функцию «getOrdersList()» получает список заказов и возвращает информацию о позициях в корзине заказа. Входные параметры аналогичны, только используется один конкретный номер заказа;

«getOrderInfo(url,order_num1)» – данная функция, используя функцию «getOrdersList()» получает список заказов и возвращает информацию о конкретном заказе. Входные параметры аналогичны, только используется один конкретный номер заказа.

3.2.3 Разработка теста общего назначения

Под определением “Тесты общего назначения” подразумеваются тестовые сценарии, которые встречаются в каждом проекте, и не зависят от проекта. Такие тесты, при минимальных правках, можно применять для тестирования разных проектов. Такие сценарии встречаются не часто, но один из таких сценариев, это проверка нерабочих ссылок на странице.

При разработке тестов общего назначения используются библиотеки, для реализации http/https запросов Axios и библиотека для парсинга веб-страниц Cheerio [44]. Создаётся директория «common_tests» и поддиректория «service», как показано на рисунке 3.3.

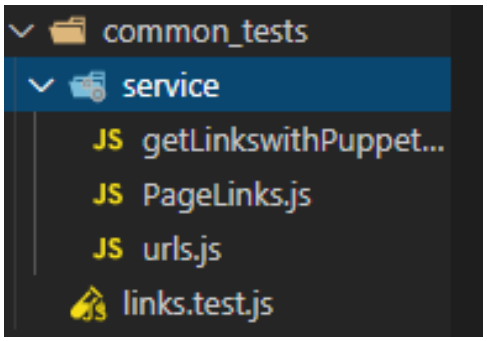

Рисунок 3.3 – Структура директории «common_tests»

В директории «service» содержаться: модуль «urls.js» - данный модуль содержит ссылки применимые для определенного проекта, и дополнительные модули, которые содержат классы с функционалом для соответствующих тестов.

Автотест для проверки наличия некорректных ссылок на странице выглядит следующим образом:

```
test("Автотест для проверки наличия битых ссылок на
production",async()=>{
    let all_links = await pl.getLinksFromPage(url.ZENER_PAGE)
    let bad_links = await pl.getBadLinks(all_links,url.ZENER_PAGE)
    expect(bad_links.length==0).toEqual(true)
},5000000)
```

Как видно из кода теста, для получения всех ссылок на странице используется функция «getLinksFromPage(url)». Входным параметром данной функции является url тестируемой страницы.

Функцией для проверки некорректных ссылок является «getBadLinks(all_links, url)». Входными параметрами данной функции является url тестируемой страницы, а также список всех ссылок старницы, который был получен с помощью функции «getLinksFromPage(url)»

Данные функции являются функциями класса «PageLinks.js», находящийся в директории «service». В нем собраны функции для работы с ссылками страницы.

3.3 Разработка приложения для конфигурирования и разворачивания тестового фреймворка

На этапе доработки bash-скрипта, была поставлена задача сделать утилиту более конфигурируемой:

1. Добавить возможность выбора вида тестов для проекта. Необходимо предоставлять пользователю на выбор добавления в проект трех возможных вида тестов: «API», «UI», «Тесты общего назначения».

2. Добавить возможность выбора создания шаблонов тестов. При выборе данной опции, для пользователя в созданных директориях создаются шаблоны-примеры тестов со структурой тестов и примерами их описания, а также подсказки в виде комментариев. Это позволит начинающему разработчику быстрее сориентироваться в стиле написания автотестов в данном разработанном фреймворке.

3. Добавить возможность конфигурации для опытного разработчика и для начинающего разработчика.

4. Делегирование сущностей, используемых в скрипте, по отдельным файлам: списки шаблонов конфигурационных файлов, списки используемых Node.js библиотек, шаблоны примеров автотестов.

Данные пожелания привели к необходимости отойти от идеи консольного приложения и постановки задачи разработки GUI приложения, так как пользователю проще воспринимать визуально большой поток информации, а также все варианты конфигураций будут отображены в одном месте, что также улучшает восприятие.

Приложение было решено реализовать на языке программирования Python [45]. Python – высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода. Высокая читаемость кода делает этот язык простым и быстрым, что позволит сократить сроки разработки приложения и облегчит техническую поддержку при эксплуатации.

3.3.1 Обзор библиотеки Tkinter

Для реализации графического интерфейса приложения было решено использовать библиотеку Tkinter [46]. Tkinter – это графическая библиотека, позволяющая создавать программы с оконным интерфейсом. Эта библиотека является интерфейсом к популярному языку программирования и инструменту создания графических приложений tcl/tk.

Tkinter, является кроссплатформенной библиотекой и может быть использована в большинстве распространённых операционных систем: Windows, Linux, Mac OS X. Это делает разрабатываемое приложение кроссплатформенным и независящим от операционной системы разработчика.

Эта библиотека не имеет в своем арсенале больших возможностей в плане создания сложных интерфейсов, но в разрабатываемом приложении не требуется сложный дизайн. Она проста в освоении и решает все поставленные задачи.

3.3.2 Дизайн окна разрабатываемого приложения

С учетом поставленных задач, был разработан макет окна разрабатываемого приложения, показанный на рисунке 3.4.

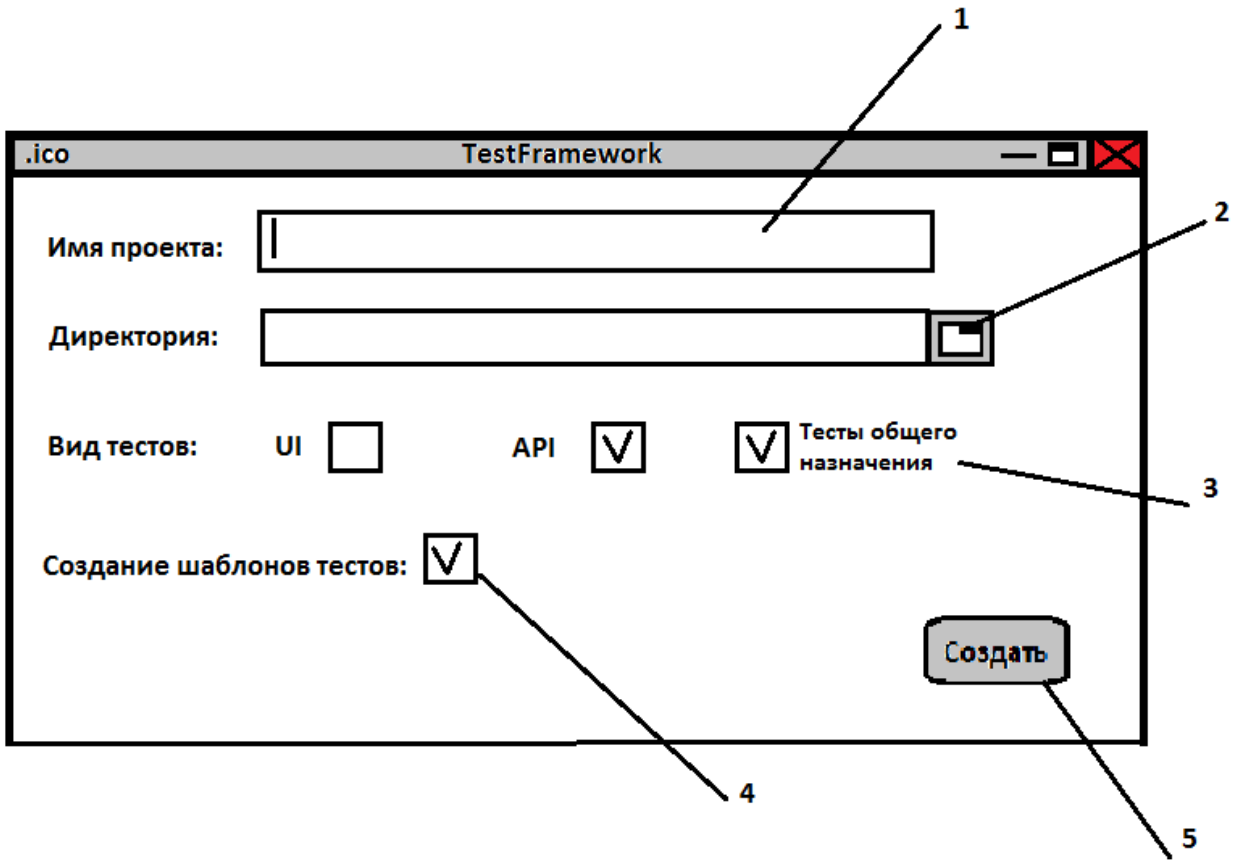

Рисунок 3.4 – Макет разрабатываемого приложения TAFC

На данном макете изображены следующие элементы графического интерфейса:

1. Поле ввода названия проекта.

2. Поле ввода директории. В выбранной директории будет создан каталог с проектом в соответствии с разработанным тестовым фреймворком.

3. Чекбоксы выбора вида тестов, которые будут использоваться в проекте. Здесь решается задача возможности добавления в проект разных видов тестов по выбору.

4. Чекбокс выбора создания шаблонов тестов. Здесь решается задача разделения разработчиков по опыту.

5. Кнопка «Создать» для применения выбранной конфигурации и создания проекта.

3.3.3 Структура разрабатываемого приложения

При планировании структуры разрабатываемого приложения решается задача делегирования сущностей, используемых в скрипте, по отдельным файлам: списки шаблонов конфигурационных файлов, списки используемых Node.js библиотек, шаблоны примеров автотестов, как показано на рисунке 3.5

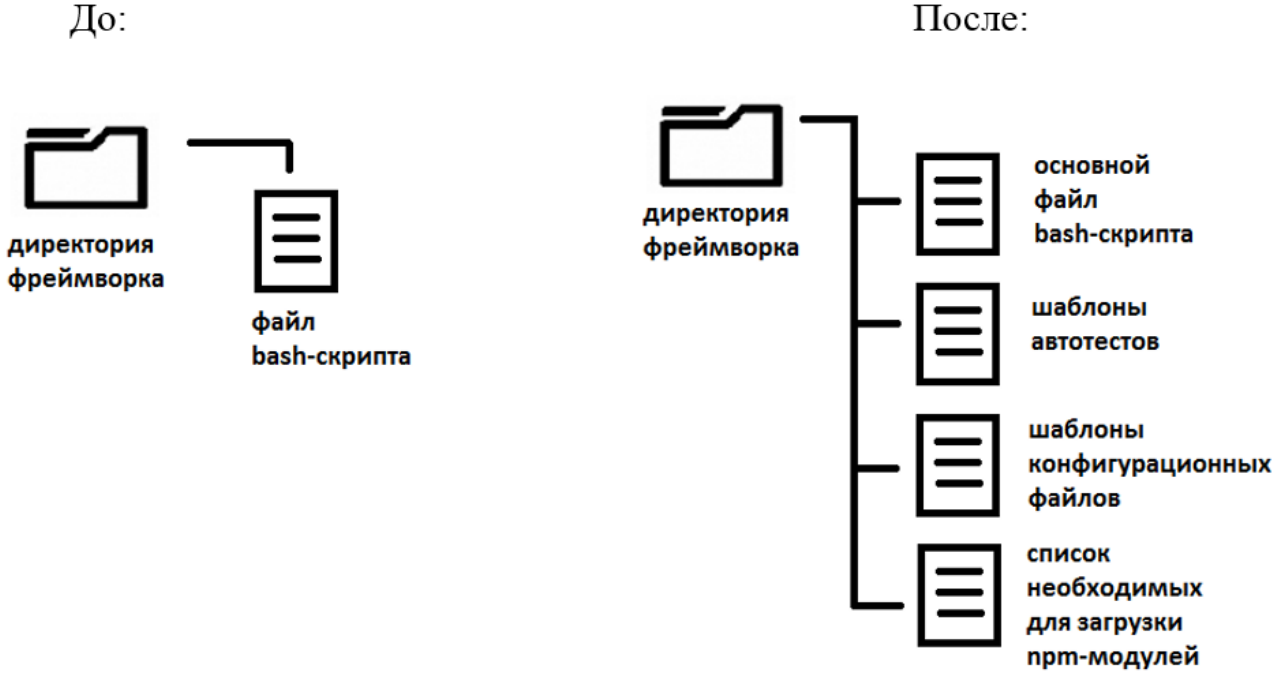

Рисунок 3.5 – Структура разрабатываемой программы

В результате, приложение “TestAutomationFrameworkCreate” состоит из нескольких модулей, показанных на рисунке 3.6:

1. Модуль “com_test_templates.py” содержит в себе шаблоны тестов общего назначения. Данные тесты разрабатываются независимыми от проекта и проверяют общую для большинства проектов функциональность.

2. Модуль “main.py” содержит основной код программы: функции создания необходимых директорий и графический интерфейс.

3. Модуль “npm_libraries.py” содержит списки node.js библиотек.

4. Модуль “templates.py” содержит шаблоны конфигурационных файлов, а также шаблоны структур автотестов с описанием и подсказками.

5. Иконка программы.

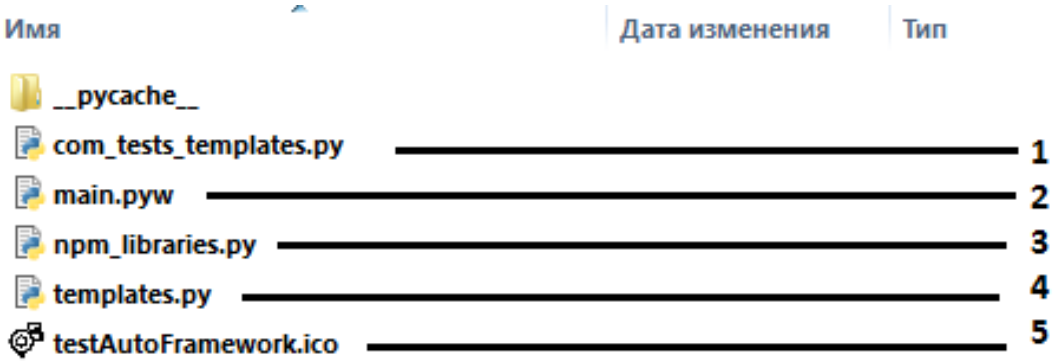

Рисунок 3.6 – Структура программы “TAFC”

3.3.4 Разработка приложения «TAFC»

Код разрабатываемого приложения, представленный в приложении В, был поделен на три части:

1. Создание и отображение элементов графического интерфейса.

В данной части используется функционал библиотеки Tkinter для создания элементов графического интерфейса. К примеру, данный код создает экземпляр окна графического приложения; добавляет заголовок, размеры и выводит окно по центру экрана:

```
root =Tk()
UI_var=IntVar()
API_var=IntVar()
```

```
Temp_var=IntVar()
NPM_var=IntVar()
COMMON_var=IntVar()
root.title("TestAutomationFrameworkCreate TAFC")
x = (root.winfo_screenwidth() - root.winfo_reqwidth()) / 2
y = (root.winfo_screenheight() - root.winfo_reqheight()) / 2
x=x-200
y=y-200
root.wm_geometry("+%d+%d" % (x, y))
root.geometry("680x500")
```

Следующий фрагмент кода демонстрирует создание элемента «Label» и создание поля ввода «Entry». «Label» служит для вывода текста на экран. Данный код создает поле ввода имени проекта:

```
name = Label(root,text="Введите имя проекта:")
entryName = Entry(root,width=50)
name.grid(column=0,row=0,sticky=W,ipadx=10)
entryName.grid(column=1,row=0,pady=10,sticky=W)
```

Для создания чекбокса необходимо создать элемент «Checkbutton». В данном коде демонстрируется создание выбора видов тестов для проекта:

```
testChk= Label(root,text="Виды тестов для проекта:")
chkUI = Checkbutton(root, text="UI тесты", variable=UI_var)
chkApi = Checkbutton(root, text="API тесты", variable=API_var)
chkCOMMON = Checkbutton(root, text="Тесты общего\n назначения",
variable=COMMON_var)
testChk.grid(column=0,row=2,sticky=W,ipadx=10)
chkUI.grid(column=1,row=2,sticky=W+E)
chkApi.grid(column=1,row=2, sticky=W)
chkCOMMON.grid(column=1,row=2, sticky=E)
UI_var.set(1)
```

Таким образом на данных тривиальных элементах строиться весь необходимы графический интерфейс, в соответствии с макетом, показанным на рисунке 3.4.

2. Разработка функций событий для элементов интерфейса.

На такие элементы интерфейса как кнопки, в Tkinter это «Button», навешиваются события. В данном коде на кнопку «Создать», навешивается событие «createFolder»:

```
btnCreate = Button(root, text="    Создать    ", font=("Arial
Bold",11),command=createFolder)
btnCreate.grid(column=2, row=8,sticky=S).
```

«CreateFolder» – это основная функция, где происходит создание проекта с фреймворком.

В данной функции используются вспомогательные функции, показанные в таблице 3.1, которые принимают в качестве входного параметра директорию текущего проекта.

Таблица 3.1 – Описание основных функций приложения «TAFC»

Название функции	**Описание**
npmApiCreate(dir)	Данная функция необходима для установки соответствующих Node.js модулей в проект, при разработке API тестов, без необходимости устанавливать лишние модули и перегружать проект.
npmUICreate(dir)	Данная функция необходима для установки соответствующих Node.js модулей в проект, при разработке UI тестов, без необходимости устанавливать лишние модули и перегружать проект.
commonCreate(dir)	Данная функция создает необходимые директории для разработки тестов общего назначения.
apiCreate(dir)	Данная функция создает необходимые директории для разработки API тестов.
e2eCreate(dir)	Данная функция создает необходимые директории для разработки UI тестов.
baseCreate(dir)	Данная функция создает основные директории проекта, а также добавляет конфигурационные файлы “.babelrc” и “.gitignore” в проект.

Продолжение таблицы 3.1

Название функции	Описание
commonModulsCreate(dir)	Данная функция создает необходимые файлы в поддиректории «service» директории «common_tests» и записывает в них данные шаблонов из файла «templates.py».
apiModulsCreate(dir)	Данная функция создает необходимые файлы в поддиректории «service» директории «api» и записывает в них данные шаблонов из файла «templates.py».
e2eModulsCreate(dir)	Данная функция создает необходимые файлы в поддиректории «service» директории «e2e» и записывает в них данные шаблонов из файла «templates.py».

3. Разработка дополнительных функций.

Также в коде приложения разработаны дополнительные функции, которые в свою очередь обслуживают основные функции, описанные в таблице 3.2.

Таблица 3.2 – Описание дополнительных функций приложения «TAFC»

Название функции	Описание
openFolder(dir)	Данная функция открывает директорию в новом окне. Используется в функции «createFolder», по завершению процесса создания проекта с фреймворком, открывается папка с готовым проектом.
get_npm_ver()	Данная функция позволяет получить текущую версию npm на данной машине.

Продолжение таблицы 3.2

Название функции	Описание
get_node_ver()	Данная функция позволяет получить текущую версию Node.js на данной машине.
getThisDir()	Данная функция позволяет получить текущую директорию в виде строки.
insertDirName()	Данная функция позволяет вставить либо заменить значение в поле ввода текста.
exit()	Осуществляет выход из программы.
writeInFile(dir,content)	Данная функция позволяет записывать данные в файл. Имеет два входных параметра: dir – путь к файлу, в который нужно записать данные, если файла нет, то он будет создан; content – данные для записи в файл. Данная функция используется в таких функциях как, «writeRequestFuncModule(dir)», «writeUrlFile(dir)», «writeBabelrc(dir)», «writeCreateBrowserModule(dir)», «writeUITestExampleModule(dir)», и другие.

3.3.5 Описание разработанной программы TAFC

Данная программа, после предварительной конфигурации, создает необходимую директорию “./имя_проекта/test”, с поддиректориями “api”, “e2e”, “common_tests”, в зависимости от выбранного варианта.

Также в директории “./имя_проекта” создается стандартный npm проект (команда “npm init”) с файлом package.json, директорией “node_modules” и конфигурационными файлами “.babelrc”, ”.gitignore”, описанные в файле templates.py. В зависимости от конфигурации пользователем, в проект устанавливаются необходимые для разработки node модули, хранящиеся в файле npm_libraries.py.

Для использования утилиты необходимо запустить bash-терминал в директории тестового фреймворка “./TestAutomationFramework” и выполнить команду “python main.pyw”, в центре экрана появиться окно программы, показанное на рисунке 3.7.

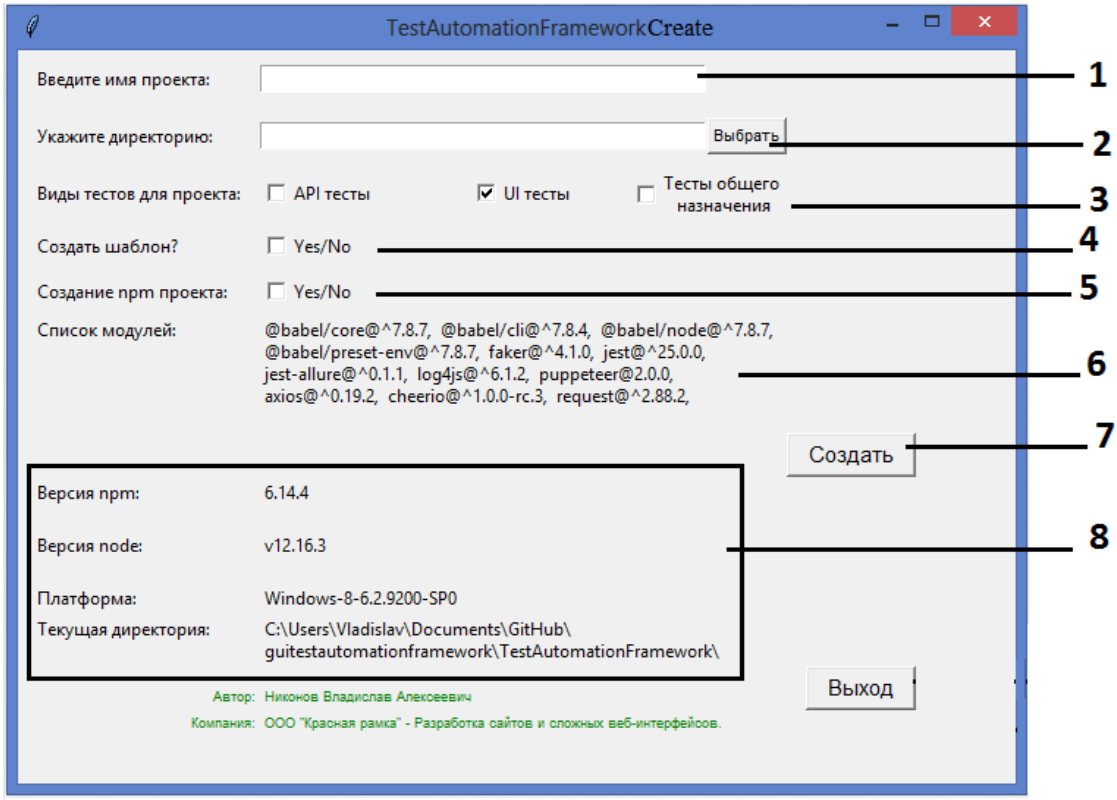

Рисунок 3.7 – Окно утилиты “TestAutomationFrameworkCreate”

Как показано на рисунке 3.7, программа имеет следующие элементы интерфейса:

1. Поле ввода названия проекта. Введенное значение используется в названии создаваемой под проект директории и, как правило, должно соответствовать названию тестируемого проекта.

2. Кнопка выбора директории «Выбрать». При нажатии кнопки, появляется системное окно выбора директории, как показано на рисунке 3.8, в которой будет создана папка под проект.

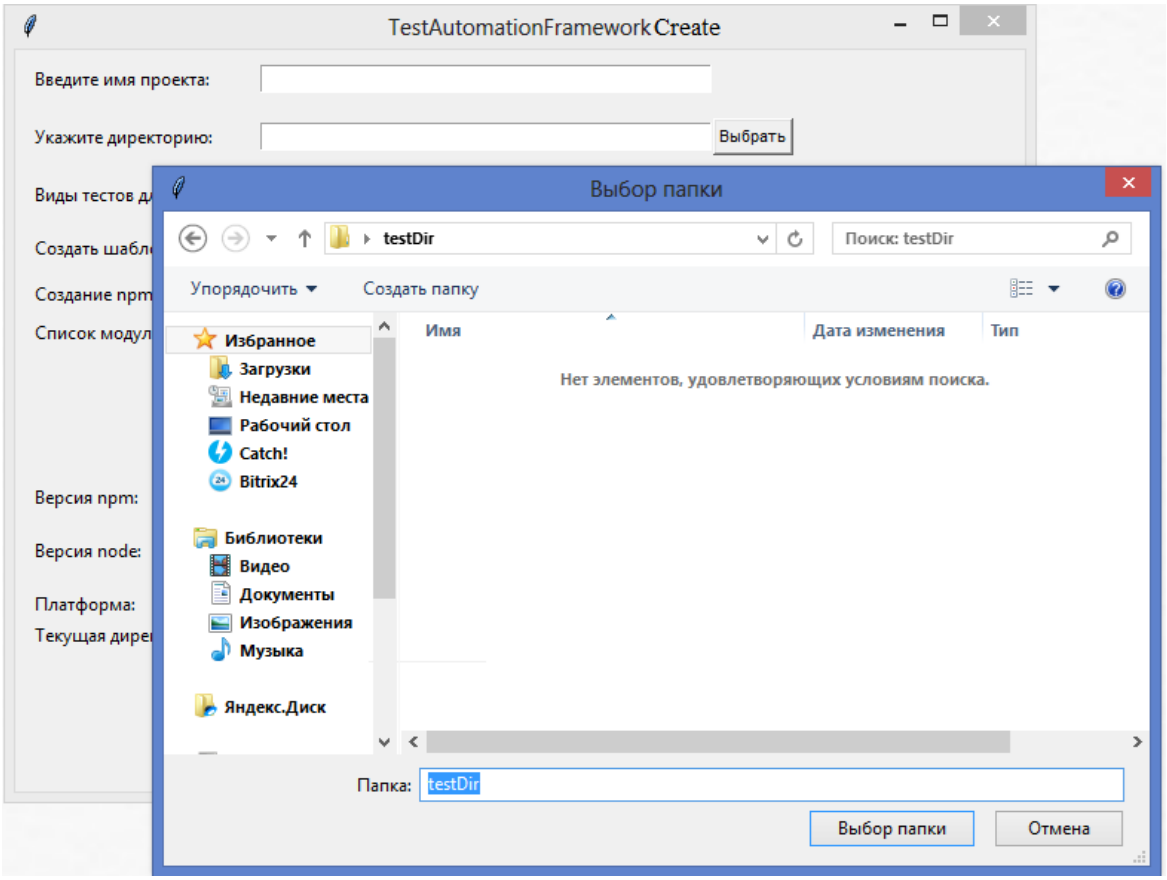

Рисунок 3.8 – Окно выбора директории после нажатия «Выбрать»

3. Чекбоксы для выбора видов тестов. Позволяют выбрать вид тестов, которые, предположительно, будут разрабатываться в проекте. На выбор предоставляются три вида: «api», «ui», «общего назначения». Будут созданы соответствующие выбору подкаталоги и модули.

4. Чекбокс выбора создания шаблонов. При выборе данного чекбокса, в каталогах под разные виды автотестов, будут созданы шаблоны автотестов. Шаблон содержит необходимую структуру тестов, соответствующую Jest фреймворку, примеры правильного описания автотеста с помощью глобальной переменной «reporter», а также необходимые пояснения в виде комментариев, как показано на рисунке 3.9.

```
import "@babel/polyfill";
import "@babel/preset-env";
import "jest-allure/dist/setup";
import { Severity } from "jest-allure/dist/Reporter";
var log4js = require("log4js");
let createBrowser = require("../e2e/service/createBrowser");
let { HomePage } = require("../e2e/pages/HometPage");
let logger = log4js.getLogger();
logger.level = "ALL";  // уровень логирования

describe("Набор автотестов для примера, для проверки сравнений чисел:", () => {  // название тестового набора
  let l = 5;

  test("Первый тест", async () => {  // название теста
    reporter.feature("Здесь название тестируемой функциональности номер 1");  // название тестируемой функциональности
    reporter.description("Здесь описание тестового сценария"); // описание тестируемой функциональности
    reporter.severity(Severity.Trivial);  // severity
    reporter.addLabel("Status","BUG"); // добавление лейбла
    reporter.startStep("Название шага тестового сценария: 5 должно быть больше чем 4"); // начало шага
    expect(l > 4).toBe(true);
    reporter.endStep("Конец шага"); // конец шага
  }, 15000);

  test("Второй тест", async () => {  // название теста
    reporter.feature("Здесь название тестируемой функциональности номер 2"); // название тестируемой функциональности
    reporter.startStep("Название шага тестового сценария: 5 - 4 должно быть равно 1"); // начало шага
    expect(l - 4).toEqual(1);
    reporter.endStep("Конец шага"); // конец шага
  }, 15000);

});
```

Рисунок 3.9 – Пример шаблона автотеста

5. Чекбокс выбора создания npm проекта. При выборе данного чекбокса, в директории проекта создается файл package.json, который содержит описание проекта по дефолту (соответствует команде «npm init --yes»). Также, в соответствии с выбором вида тестов, устанавливаются необходимые node-модули (соответствует команде «npm install названия_библиотек»), которые хранятся в файле npm_libraries.py.

6. Список стандартных модулей. В данном блоке отображаются все node.js библиотеки, которые используются в проекте, как стандартные. Данные список храниться в файле npm_libraries.py.

7. Кнопка «Создать». По нажатию данной кнопки, происходит создание проекта в соответствии с выбранной заранее конфигурацией.

8. Информация о системе. В данном блоке отображается информация о версиях node.js и npm, установленных на текущей платформе. В случае проблем с вышесказанными программами, в данном блоке будет отображена ошибка, что позволит разработчику устранить проблемы до работы, непосредственно, с функционалом утилиты “TAFC”. Также, для удобства, отображена версия текущей платформы и текущая директория, в которой запущена утилита.

4 Разработка регламента процессов автоматизации тестирования

Для разработки регламента был создан документ содержание которого показано на рисунке 4.1.

Содержание

Рисунок 4.1 – Содержание регламента автоматизации тестирования

Данный документ разработан, в первую очередь, для неопытных специалистов для быстрого введения их в процесс автоматизации тестирования в компании, знакомства со структурой разработанного тестового фреймворка, используемого в автоматизации тестирования и приучения их к однообразному стилю разработки автотестов, принятому в компании.

4.1 Разработка регламента использования программы TAFC

Как показано на рисунке 3.1, данная часть регламента состоит из одной главы «Утилита “TestAutomationFrameworkCreate”» и подглав «Описание программы “TAFC”», «Структура программы “TAFC”».

Данная часть состоит из подробного описания программы “TAFC”, описания элементов интерфейса данной программы, описания функционала и

структуры. После ознакомления начинающий разработчик будет уверенно владеть данной утилитой и сможет конфигурировать и разворачивать с помощью нее проекты с разработанным тестовым фреймворком.

4.2 Разработка регламента процесса разработки автотестов

Как показано на рисунке 4.1, данная часть регламента состоит из четырех глав: «Разработка автотестов», «Создание item в Jenkins», «Allure отчеты» и «Ссылки на документацию».

В главе «Разработка автотестов» содержится описание всех моментов связанных с разработкой и оформлением автотестов. Данная глава состоит из подглав:

«UI тесты», «API тесты», «Тесты общего назначения» – здесь описаны структуры директорий для разработки соответствующих тестов, необходимые модули и библиотеки для разработки.

«Логирование» – в данной подглаве упоминается библиотека для логирования в Node.js проекте. Начинающий разработчик должен понимать, что логирование важная часть разработки, так как при помощи логирования можно увеличить поток отображаемых данных при сборке проекта в среде непрерывной интеграции Jenkins, что позволит ускорить поиск причины ошибок, либо наоборот, подтвердит корректность данных.

«Описание автотеста с использованием reporter» – в данной подглаве для начинающего специалиста, рассказано о переменной «reporter», которая позволяет описать автотест как обычный тест-case. В автотест добавляются: описание функциональности которую проверяет автотест, с помощью функции «feature», с помощью функции «description» добавлено описание самого автотеста, возможность добавить серьезность и приоритет автотеста с помощью функций «severity» и «priority», также есть возможность разделения кода отдельных смысловых блоков автотеста на отдельные описываемые шаги.

Все эти функции позволяют отобразить полноценный allure-отчет, который будет понятен уже не только техническим специалистам, но также менеджерам и специалистам по качеству низкого уровня подготовки.

В главе «Создание item в Jenkins» подробно описано создание задачи в среде непрерывной интеграции Jenkins. Данная информация поможет начинающему специалисту быстро разобраться в данном сервисе.

Глава «Allure отчеты» описывает настройку автоматического создания отчета о пройденных задачах в среде непрерывной интеграции Jenkins. Наглядно показаны примеры генерируемых отчетов.

В главе «Ссылки на документацию» приведены ссылки на документацию библиотек и сервисов, используемых в разработке, для помощи начинающему разработчику. На данных ресурсах находится полная, необходимая для полноценной разработки автотестов, информация.

Заключение

В ходе выполненной работы были получены следующие ***результаты и выводы***:

1. Установлены и описаны основные преимущества и недостатки инструментов и технологий для автоматизации тестирования веб-приложений. На основе полученной информации, был выбран оптимальный стек технологий для разработки фреймворка для автоматизации тестирования веб-приложений.

2. Разработан фреймворк для автоматизации тестирования веб-приложений, позволяющий решить все необходимые задачи автоматизации в рамках процесса разработки веб-приложений.

3. В процесс разработки внедрена концепция непрерывной интеграции для автоматизированного тестирования, что позволяет специалистам по контролю качества ускорить процессы обнаружения и анализа найденных дефектов и соответственно сократить жизненный цикл дефекта.

4. Разработано программное обеспечение для конфигурирования и автоматического разворачивания тестового фреймворка для автоматизации тестирования веб-приложений, позволяющее сократить время на создание и настройку проекта и в кратчайшие сроки начать разработку автотестов.

5. С целью быстрого введения специалистов в процесс автоматизации тестирования в компании, знакомства со структурой разработанного тестового фреймворка, используемого в автоматизации тестирования и приучения их к однообразному стилю разработки автотестов, принятому в компании, был создан регламент процессов автоматизации тестирования веб-приложений.

В дальнейшем планируется доработка разработанного программного обеспечения для конфигурирования и разворачивания тестового фреймворка для автоматизации тестирования веб приложений, а именно разработка дополнительных модулей, позволяющих расширить стек используемых технологий и инструментов для автоматизации тестирования веб-приложений.

Сокращения, обозначения, термины и определения

В данной выпускной квалификационной работе были использованы следующие сокращения, обозначения, термины и определения.

Agile – это философия, семейство гибких подходов к разработке программного обеспечения.

Monkey-тестирование – это метод, при котором пользователь тестирует приложение или систему, предоставляя случайные входные данные и проверяя поведение или наблюдая, произойдет ли сбой приложения или системы.

Open-source – проект с открытым исходным кодом.

Smoke-тестирование – минимальный набор тестов на явные ошибки.

Билд – подготовленный для использования информационный продукт, в виде исполняемого, двоичного файла, содержащий исполняемый код программы или библиотеки.

Кроссбраузерность – свойство веб-приложения отображаться и корректно функционировать во всех часто используемых браузерах идентично.

Паттерн – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Регрессионное тестирование – собирательное название для всех видов тестирования программного обеспечения, направленных на обнаружение ошибок в уже протестированных участках исходного кода.

Сервер – программный компонент вычислительной системы, выполняющий обслуживающие функции по запросу клиента, предоставляя ему доступ к определённым ресурсам или услугам.

Скрипт – это последовательность действий, описанных с помощью скриптового языка программирования для автоматического выполнения определенных задач.

Тест-кейс – это формально описанный алгоритм тестирования программы, специально созданный для определения возникновения в программе определённой ситуации, определённых выходных данных.

Тест-план – это документ, описывающий весь объем работ по тестированию: описание объекта тестирования, стратегии, расписания, критерии начала и окончания тестирования, необходимое оборудование, специальные знания, а также оценки рисков с вариантами их решения.

Фреймворк — в самом общем случае данное слово обозначает структуру, содержащую некоторую информацию.

API – Application Programming Interface.

CSS – Cascading Style Sheets.

DevOps – Development and operations.

DOM – Document Object Model.

GUI – Graphical User Interface.

HTTP – HyperText Markup Language.

JSON – JavaScript Object Notation.

NPM – Node Package Manager.

QA – Quality Assurence.

REST – Representational State Transfer.

UI – User Interface.

XML – eXtensible Markup Language.

Xpath – XML Path Language.

ООП – Объектно-ориентированный подход.

ОС – Операционная система.

ЯП – Язык программирования.

Список использованных источников

1. Советы и рекомендации по развёртыванию процесса автоматизация тестирования с нуля [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/275171/ - свободный (дата обращения 10.9.2018г.).

2. Элфрид Дастин, Джефф Рэшка, Джон Пол Автоматизированное тестирование программного обеспечения. – Москва: Лори, 2003. – 592 с.

3. Стратегия автоматизации тестирования для Agile-проектов. [Электронный ресурс]. – Режим доступа: https://tproger.ru/translations/test-automation-strategy-for-agile-projects/ - свободный (дата обращения 10.9.2018г.).

4. Гленфорд Майерс, Том Баджетт, Кори Сандлерб Искусство тестирования программ. – Москва: Вильямс, 2012. – 271 с.

5. Святослав Куликов Тестирование программного обеспечения. Базовый курс. – Минск: Четыре четверти, 2017. – 314 с.

6. Лиза Криспин, Джанет Грегори Гибкое тестирование. – Москва: Вильямс, 2016. – 464 с.

7. Linda G. Hayes Automated Testing Handbook. – Software Testing Inst, 2004. – 182 с.

8. Как выбрать тесты для автоматизации? [Электронный ресурс]. – Режим доступа: https://quality-lab.ru/blog/how-to-select-test-to-be-automated/ - свободный (дата обращения 10.9.2018г.).

9. Автоматизация тестирования: минусы [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/111292/ - свободный (дата обращения 10.9.2018г.).

10. E2E автоматизация веб-приложений 2019: выбери свой инструмент. [Электронный ресурс]. – Режим доступа: https://cutt.ly/xe3yc1i - свободный (дата обращения 10.9.2018г.).

11. npmtrends: cypress vs testcafe vs puppeteer vs selenium-webdriver. [Электронный ресурс]. – Режим доступа: https://www.npmtrends.com/cypress-vs-

testcafe-vs-puppeteer-vs-selenium-webdriver - свободный (дата обращения 10.9.2018г.).

12. Selenium 2.0 и WebDriver. [Электронный ресурс]. – Режим доступа: https://selenium2.ru/docs/webdriver.html - свободный (дата обращения 10.9.2018г.).

13. Использование Selenium сервера для автоматизации работы с внешними ресурсами [Электронный ресурс]. – Режим доступа: https://www.azoft.ru/blog/selenium/ - свободный (дата обращения 10.9.2018г.).

14. Puppeteer documentation [Электронный ресурс]. – Режим доступа: https://github.com/puppeteer/puppeteer - свободный (дата обращения 10.9.2018г.).

15. Cypress documentation [Электронный ресурс]. – Режим доступа: https://docs.cypress.io/guides/overview/why-cypress.html - свободный (дата обращения 10.9.2018г.).

16. Cypress vs. Selenium. [Электронный ресурс]. – Режим доступа: https://automationrhapsody.com/cypress-vs-selenium-end-era/ - свободный (дата обращения 10.9.2018г.).

17. Page Object – путь к совершенным автотестам [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/wapstart/blog/138674/ - свободный (дата обращения 10.9.2018г.).

18. Какой язык выбрать для тестирования? [Электронный ресурс]. – Режим доступа: https://xpinjection.com/articles/what-language-to-choose-for-testing/ - свободный (дата обращения 10.9.2018г.).

19. Хамбл Джез, Фарли Дэвид Непрерывное развертывание ПО. Автоматизация процессов сборки, тестирования и внедрения новых версий программ. – Москва: Вильямс, 2016. – 425 с.

20. 10 систем непрерывной интеграции, о которых стоит знать. [Электронный ресурс]. – Режим доступа: https://techrocks.ru/2019/03/11/10-continuous-integration-systems/ - свободный (дата обращения 10.9.2018г.).

21. Тестирование с помощью TestNG в Java [Электронный ресурс]. Режим доступа: https://devcolibri.com/тестирование-с-помощью-testng-в-java/ - свободный (дата обращения 10.9.2018г.).

22. JUnit 4 Vs TestNG - Сравнение. [Электронный ресурс]. – Режим доступа: https://www.mkyong.com/unittest/junit-4-vs-testng-comparison/ - свободный (дата обращения 10.9.2018г.).

23. Роль архитектуры в автоматизации тестирования. [Электронный ресурс]. – Режим доступа: https://automated-testing.info/t/rol-arhitektury-v-avtomatizaczii-testirovaniya/3037/1 - свободный (дата обращения 10.9.2018г.).

24. Явные и неявные ожидания в Selenium WebDriver [Электронный ресурс]. – Режим доступа: https://software-testing.ru/library/testing/testing-automation/2679-selenium-webdriver - свободный (дата обращения 10.9.2018г.).

25. Рейтинг языков программирования 2020 [Электронный ресурс]. – Режим доступа: https://techrocks.ru/2020/02/08/programming-languages-rank-2020/ - свободный (дата обращения 10.9.2018г.).

26. About Node.js [Электронный ресурс]. – Режим доступа: https://nodejs.org/en/about/ - свободный (дата обращения 10.9.2018г.).

27. About npm [Электронный ресурс]. – Режим доступа: https://docs.npmjs.com/about-npm/ - свободный (дата обращения 10.9.2018г.).

28. Jest. Getting Started [Электронный ресурс]. – Режим доступа: https://jestjs.io/docs/en/getting-started - свободный (дата обращения 10.9.2018г.).

29. Паттерны для тестировщиков [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/414253/ - свободный (дата обращения 10.9.2018г.).

30. Кросс-браузерное тестирование [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Learn/Tools_and_testing/Cross_browser_testing - свободный (дата обращения 10.9.2018г.).

31. Три способа поднять Jenkins CI для ваших автотестов [Электронный ресурс]. – Режим доступа: http://automation-remarks.com/tri-sposoba-podniat-jenkins-ci-dlia-vashikh-avtotiestov/ - свободный (дата обращения 10.9.2018г.).

32. Continuous Integration with Jenkins and GitLab [Электронный ресурс]. – Режим доступа: https://medium.com/@teeks99/continuous-integration-with-jenkins-and-gitlab-fa770c62e88a - свободный (дата обращения 10.9.2018г.).

33. Allure Test Report [Электронный ресурс]. – Режим доступа: http://allure.qatools.ru/ - свободный (дата обращения 10.9.2018г.).

34. Jest-Allure reporting plugin [Электронный ресурс]. – Режим доступа: https://www.npmjs.com/package/jest-allure - свободный (дата обращения 10.9.2018г.).

35. Основы BASH [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/47163/ - свободный (дата обращения 10.9.2018г.).

36. Node JS Web Scraper [Электронный ресурс]. Режим доступа: https://gist.github.com/corinneling/3a59ca3585eac261682e26ef8888b221 - свободный (дата обращения 10.9.2018г.).

37. Cheeriojs/cheerio [Электронный ресурс]. – Режим доступа: https://github.com/cheeriojs/cheerio - свободный (дата обращения 10.9.2018г.).

38. Npm/axios [Электронный ресурс]. – Режим доступа: https://www.npmjs.com/package/axios - свободный (дата обращения 10.9.2018г.).

39. Request [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Web/API/Request - свободный (дата обращения 10.9.2018г.).

40. request-promise [Электронный ресурс]. – Режим доступа: https://www.npmjs.com/package/request-promise - свободный (дата обращения 10.9.2018г.).

41. Использование Fetch [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Web/API/Fetch_API/Using_Fetch - свободный (дата обращения 10.9.2018г.).

42. Используем Axios для доступа к API [Электронный ресурс]. – Режим доступа: https://ru.vuejs.org/v2/cookbook/using-axios-to-consume-apis.html - свободный (дата обращения 10.9.2018г.).

43. Работа с JSON [Электронный ресурс]. – Режим доступа: https://developer.mozilla.org/ru/docs/Learn/JavaScript/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D1%8B/JSON - свободный (дата обращения 10.9.2018г.).

44. Scrape a site with Node and Cheerio [Электронный ресурс]. – Режим доступа: https://medium.com/@dylan.sather/scrape-a-site-with-node-and-cheerio-in-5-minutes-4617daee3384 - свободный (дата обращения 10.9.2018г.).

45. Beginner's Guide to Python. [Электронный ресурс]. – Режим доступа: https://wiki.python.org/moin/BeginnersGuide - свободный (дата обращения 10.9.2018г.).

46. Tkinter – Python interface to Tcl/Tk [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/library/tkinter.html - свободный (дата обращения 10.9.2018г.).

Приложение А

(обязательное)

Bash-скрипт автоматизации разворачивания тестового фреймворка для автоматизации тестирования веб-приложений

```
#!/bin/bash
echo -n "Enter project name:"
read project
mkdir $project
cd $project
mkdir ./test
mkdir ./test/e2e test/pages test/service test/rest
touch ./test/service/urls.js
echo 'module.exports = { /* В данном модуле перечисляются необходимые url, используемые в проекте. */ }' > ./test/service/urls.js
echo 'Module "urls.js" for web project urls description was created..'
touch ./test/service/selectors.js
echo 'Module "selectors.js" for web project selectors description was created..'
echo '{ "presets": ["@babel/preset-env"] }' > .babelrc
echo "Babel config file was created.."
echo 'node_modules\nallure-results' > .gitignore
echo "Git ignore file was created.."
npm init
npm install @babel/core@^7.8.7 @babel/cli@^7.8.4 @babel/node@^7.8.7 @babel/preset-env@^7.8.7
npm install faker@^4.1.0 jest@^25.0.0 jest-allure@^0.1.1 log4js@^6.1.2 puppeteer@2.0.0
```

Приложение Б

(обязательное)

Код автотеста для тестирования API

```
test(`Test`,async()=>{
let order_item_info = await rf.getOrderItemInfo(url.DEV_PAGE,238029)
let is_ID_present = order_item_info[0].ID!=undefined &&
order_item_info[0].ID!="";
let is_NAME_present = order_item_info[0].NAME!=undefined &&
order_item_info[0].NAME!="";
let is_MANUFACTURER_present =
order_item_info[0].MANUFACTURER!=undefined &&
order_item_info[0].MANUFACTURER!="";
let is_PRODUCT_STOCK_present =
order_item_info[0].PRODUCT_STOCK!=undefined &&
order_item_info[0].PRODUCT_STOCK!="";
let is_STOCK_PRICE_present = order_item_info[0].STOCK_PRICE!=undefined
&& order_item_info[0].STOCK_PRICE!="";
let is_PRICE_present = order_item_info[0].PRICE!=undefined &&
order_item_info[0].PRICE!="";
expect(order_item_info.length!=0).toEqual(true) // проверить что api возвращает
не пустые данные о корзине
// проверить, что данные о продуктах в корзине приходят корректно
expect(is_ID_present).toEqual(true)
expect(is_NAME_present).toEqual(true)
expect(is_MANUFACTURER_present).toEqual(true)
expect(is_PRODUCT_STOCK_present).toEqual(true)
expect(is_STOCK_PRICE_present).toEqual(true)
expect(is_PRICE_present).toEqual(true)       }, 30000000);
```

Приложение В

(обязательное)

Код приложения «TestAutomationFrameworkCreate» («TAFC»)

```
from tkinter import *
from tkinter import filedialog
from tkinter.messagebox import *
import subprocess
import os
import platform
import npm_libraries
import templates
import com_tests_templates

# работа с файлами
def writeInFile(dir,content):
    f = open(dir,'w')
    f.write(content)
    f.close()

def writeRequestFuncModule(dir):
    writeInFile(dir,templates.request_func_template)

def writeUrlFile(dir):
    writeInFile(dir,templates.url_template)

def writeSelectorsModule(dir):
    writeInFile(dir, templates.selectors_templates)
```

```
def writeCreateBrowserModule(dir):
    writeInFile(dir,templates.createbrowser_template)

def writeUsersModule(dir):
    writeInFile(dir,templates.users_template)

def writeBabelrc(dir):
    writeInFile(dir,templates.babelrc_template)

def writeGitignore(dir):
    writeInFile(dir,templates.gitignore_template)

def writePageObjectBaseModules(dir):
    writeInFile(dir + "/BasePage.js", templates.basepage_template)
    writeInFile(dir + "/HomePage.js", templates.homepage_template)

def writeUITestExampleModule(dir):
    writeInFile(dir + "/test/e2e/example.test.js", templates.ui_test_example_template)

def writeAPITestExampleModule(dir):
    writeInFile(dir + "/test/api/example.test.js", templates.api_test_example_template)

# добавлять тесты общего назначения тут ->
def writeCommonTests(dir):
    writeInFile(dir + "/test/common_tests/service/PageLinks.js",
com_tests_templates.page_link_class)
    writeInFile(dir + "/test/common_tests/link.test.js",
com_tests_templates.bad_links_test_template)
```

```
# создание файлов и папок
def e2eModulsCreate(dir):
    subprocess.run(["touch",
        dir+"/test/e2e/service/urls.js",
        dir+"/test/e2e/service/selectors.js",
        dir+"/test/e2e/service/users.js"])
    writeUrlFile(dir + "/test/e2e/service/urls.js")
    writeUsersModule(dir + "/test/e2e/service/users.js")
    writeSelectorsModule(dir + "/test/e2e/service/selectors.js")
    writeCreateBrowserModule(dir + "/test/e2e/service/createBrowser.js")
    writePageObjectBaseModules(dir + "/test/e2e/pages")

def apiModulsCreate(dir):
    subprocess.run(["touch",
        dir+"/test/api/service/urls.js"])
    writeRequestFuncModule(dir + "/test/api/service/RequestFunctions.js")

def commonModulsCreate(dir):
    subprocess.run(["touch",
        dir+"/test/common_tests/service/urls.js"])
    writeUrlFile(dir + "/test/common_tests/service/urls.js")

def baseCreate(dir):
    subprocess.run(["mkdir",
        dir,
        dir+"/test"])
    writeBabelrc(dir+"/test/.babelrc")
    writeGitignore(dir+"/test/.gitignore")
```

```
def e2eCreate(dir):
    subprocess.run(["mkdir",
        dir+"/test/e2e/",
        dir+"/test/e2e/pages",
        dir+"/test/e2e/service"])

def apiCreate(dir):
    subprocess.run(["mkdir",
        dir+"/test/api/",
        dir+"/test/api/service"])

def commonCreate(dir):
    subprocess.run(["mkdir",
        dir+"/test/common_tests/",
        dir+"/test/common_tests/service"])

def npmUICreate(dir):
    # if "Windows" in platform.platform():
        # subprocess.run("cd "+dir, shell=True)
        subprocess.run("cd "+dir+" && npm init -yes", shell=True)
        for key in npm_libraries.ui_libraries:
            subprocess.run("cd "+dir+ " && npm install "
                +npm_libraries.ui_libraries.get(key), shell=True)

def npmApiCreate(dir):
    # if "Windows" in platform.platform():
        # subprocess.run("cd "+dir, shell=True)
        subprocess.run("cd "+dir+" && npm init -yes", shell=True)
        for key in npm_libraries.api_libraries:
```

```
            subprocess.run("cd "+dir+ " && npm install "
                +npm_libraries.api_libraries.get(key), shell=True)

# Создание фреймворка, основная логика
def createFolder():
    if len(entryName.get())!=0:
        if len(entrydirName.get())!=0:
            PROJKT_NAME=entryName.get()
            DIR_NAME=entrydirName.get()
            FULL_DIR_NAME=DIR_NAME+"/"+PROJKT_NAME

            if not os.path.exists(FULL_DIR_NAME):

                if (UI_var.get()==0) & (API_var.get()==0) & (COMMON_var.get()==0):
                    showinfo("Ошибка!", "Выбирите необходимый вид автотестов!")

                if (UI_var.get()==1) & (Temp_var.get()==1):
                    if not os.path.isdir(FULL_DIR_NAME):
                        baseCreate(FULL_DIR_NAME)
                    e2eCreate(FULL_DIR_NAME)
                    e2eModulsCreate(FULL_DIR_NAME)
                    writeUITestExampleModule(FULL_DIR_NAME)
                    showinfo("Создание шаблона автотеста.","Шаблон UI автотеста
создан!")
                elif (UI_var.get()==1) & (Temp_var.get()==0):
                    if not os.path.isdir(FULL_DIR_NAME):
                        baseCreate(FULL_DIR_NAME)
                    e2eCreate(FULL_DIR_NAME)
                    e2eModulsCreate(FULL_DIR_NAME)
```

```
            # elif (UI_var.get()==0) & (Temp_var.get()==1):
            #     showinfo("Ошибка!", "Выбирите создание UI тестов!")

            if (API_var.get()==1) & (Temp_var.get()==1):
                if not os.path.isdir(FULL_DIR_NAME):
                    baseCreate(FULL_DIR_NAME)
                apiCreate(FULL_DIR_NAME)
                apiModulsCreate(FULL_DIR_NAME)
                writeAPITestExampleModule(FULL_DIR_NAME)
                showinfo("Создание шаблона автотеста.","Шаблон API автотеста
создан!")
            elif (API_var.get()==1) & (Temp_var.get()==0):
                if not os.path.isdir(FULL_DIR_NAME):
                    baseCreate(FULL_DIR_NAME)
                apiCreate(FULL_DIR_NAME)
                apiModulsCreate(FULL_DIR_NAME)
            # elif (API_var.get()==0) & (Temp_var.get()==1):
            #     showinfo("Ошибка!", "Выбирите создание API тестов!")

            if (COMMON_var.get()==1):
                if not os.path.isdir(FULL_DIR_NAME):
                    baseCreate(FULL_DIR_NAME)
                commonCreate(FULL_DIR_NAME)
                commonModulsCreate(FULL_DIR_NAME)
                writeCommonTests(FULL_DIR_NAME)

            if (NPM_var.get()==1) & (UI_var.get()==1):
                npmUICreate(FULL_DIR_NAME)
```

```
            if (NPM_var.get()==1) & (API_var.get()==1):
                npmAPICreate(FULL_DIR_NAME)

            if os.path.isdir(FULL_DIR_NAME):
                openFolder(FULL_DIR_NAME)

        else:
            showinfo("Ошибка!", "Директория с таким именем существует!")
            entryName.insert(0,"")
            entryName.focus()

    else:
        showinfo("Ошибка!", "Выбирите директорию!")
        entrydirName.focus()
else:
    showinfo("Ошибка!", "Введите название проекта!")
    entryName.focus()

# дополнительные функции
def openFolder(dir):
    # if "Windows" in platform.platform():
        subprocess.run(['explorer', os.path.realpath(dir)])
        # os.system("explorer "+dir)

def get_npm_ver():
    # if "Windows" in platform.platform():
        npm_ver=subprocess.run("npm -v",stdout=subprocess.PIPE, text=True,
shell=True)
        return npm_ver.stdout
```

```
def get_node_ver():
    # if "Windows" in platform.platform():
        node_ver=subprocess.run("node -v",stdout=subprocess.PIPE, text=True,
shell=True)
        return node_ver.stdout

def getThisDir():
    # if "Windows" in platform.platform():
        # this_dir=subprocess.run("pwd",stdout=subprocess.PIPE, text=True,
shell=True)
        # this_dir.stdout
        full_dir = os.getcwd()
        if ("\\" in full_dir):
            full_dir_folders = full_dir.split("\\")
        elif ('/' in full_dir):
            full_dir_folders = full_dir.split("/")
        i = 0
        result_dir = ""
        for folder in full_dir_folders:
            i=i+1
            result_dir = result_dir + folder
            result_dir = result_dir +"\\"
            if (i%5==0):
                result_dir = result_dir + "\n"
        return result_dir

def insertDirName():
    entrydirName.insert(0, filedialog.askdirectory())
```

```
def exit():
    if askyesno("Выход", "Выйти из программы?"):
        root.destroy()

def showLibsList():
    all_libs=[]
    answ=""
    for key in npm_libraries.all_libraries:
            all_libs.append( npm_libraries.all_libraries.get(key))
    j=0
    for i in all_libs:
        j=j+1
        answ = answ + i + ",  "
        if (j%3==0):
            answ = answ+"\n"
    return answ

# код графического интерфейса
root =Tk()
UI_var=IntVar()
API_var=IntVar()
Temp_var=IntVar()
NPM_var=IntVar()
COMMON_var=IntVar()
root.title("TestAutomationFrameworkCreate TAFC")
x = (root.winfo_screenwidth() - root.winfo_reqwidth()) / 2
y = (root.winfo_screenheight() - root.winfo_reqheight()) / 2
x=x-200
```

```
y=y-200
root.wm_geometry("+%d+%d" % (x, y))
root.geometry("680x500")
# root.iconbitmap('./testAutoFramework.ico')
name = Label(root,text="Введите имя проекта:")
entryName = Entry(root,width=50)
name.grid(column=0,row=0,sticky=W,ipadx=10)
entryName.grid(column=1,row=0,pady=10,sticky=W)
dirName = Label(root,text="Укажите директорию:")
dirName.grid(column=0,row=1,sticky=W,ipadx=10)
entrydirName = Entry(root,width=50)
entrydirName.grid(column=1,row=1,pady=10,sticky=W)
dirName_btn = Button(root, text="Выбрать", font=("Arial Bold",8),
command=insertDirName)
dirName_btn.grid(column=1, row=1,sticky=E)
testChk= Label(root,text="Виды тестов для проекта:")
chkUI = Checkbutton(root, text="UI тесты", variable=UI_var)
chkApi = Checkbutton(root, text="API тесты", variable=API_var)
chkCOMMON = Checkbutton(root, text="Тесты общего\n назначения",
variable=COMMON_var)
testChk.grid(column=0,row=2,sticky=W,ipadx=10)
chkUI.grid(column=1,row=2,sticky=W+E)
chkApi.grid(column=1,row=2, sticky=W)
chkCOMMON.grid(column=1,row=2, sticky=E)
UI_var.set(1)
templatesUi = Label(root,text="Создать шаблон?")
chkTempUi = Checkbutton(root, text="Yes/No", variable=Temp_var)
templatesUi.grid(column=0,row=4, pady=5,sticky=W,ipadx=10)
chkTempUi.grid(column=1,row=4,sticky=W)
```

```
pkgjsnlb = Label(root, text="Создание npm проекта:")
chkpkgjsn = Checkbutton(root, text="Yes/No", variable=NPM_var)
pkgjsnlb.grid(column=0,row=5, pady=5,sticky=W,ipadx=10)
chkpkgjsn.grid(column=1,row=5,sticky=W)
allLibslb = Label(root, text="Список модулей:")
allLibslb.grid(column=0, row=6,sticky=W+N,ipadx=10)
if  len(showLibsList())==0:
    all_libs = Label(root,text="Ошибка!Проверьте файл со списком
модулей!",fg="red")
else:
    all_libs = Label(root,text=showLibsList(),justify=LEFT)
all_libs.grid(column=1,row=6,sticky=W+N)


btnCreate = Button(root, text="  Создать  ", font=("Arial
Bold",11),command=createFolder)# создание фреймворка
btnCreate.grid(column=2, row=8,sticky=S)
if len(get_npm_ver())>10 & len(get_npm_ver())==0:
    npmlb =Label(root,text="Версия npm : Ошибка!",fg="red")
else:
    npmlb =Label(root,text="Версия npm: ")
    npmlb_var = Label(root,text=get_npm_ver())
npmlb.grid(column=0,row=9,sticky=W+N,ipadx=10)
npmlb_var.grid(column=1,row=9,sticky=W)
nodelb =Label(root,text="Версия node: ")
if len(get_node_ver())>10 & len(get_node_ver())==0:
    nodelb_var =Label(root,text="Ошибка!",fg="red")
else:
    nodelb_var =Label(root,text=get_node_ver())
```

```
nodelb.grid(column=0,row=10,sticky=W+N,ipadx=10)
nodelb_var.grid(column=1,row=10,sticky=W)
pltflb = Label(root,text="Платформа: ")
pltflb_var= Label(root,text=platform.platform())
pltflb.grid(column=0, row=11,sticky=W,ipadx=10)
pltflb_var.grid(column=1, row=11,sticky=W)
thisDirlb = Label(root,text="Текущая директория: ")
thisDirlb_var = Label(root,text=getThisDir(),justify=LEFT)
thisDirlb.grid(column=0, row=12, sticky=W+N,ipadx=10)
thisDirlb_var.grid(column=1, row=12, sticky=W)
exitBtn = Button(root, text="  Выход  ", font=("Arial Bold",11),command=exit)
exitBtn.grid(column=2,row=13,sticky=N+E)
infAuth = Label(root,text="Автор:", font=("Arial Bold",7),fg="green")
infAuth_var = Label(root, text="Никонов Владислав Алексеевич", font=("Arial
Bold",7),fg="green")
infCom = Label(root,text="Компания:", font=("Arial Bold",7),fg="green")
infCom_var = Label(root, text="ООО \"Красная рамка\" - Разработка сайтов и
сложных веб-интерфейсов.", font=("Arial Bold",7),fg="green")
infAuth.grid(column=0, row=13, sticky=E+S)
infAuth_var.grid(column=1, row=13, sticky=W+S)
infCom.grid(column=0, row=14, sticky=E+S)
infCom_var.grid(column=1, row=14, sticky=W+S)
root.mainloop()
```

Приложение Г

(справочное)

Акт о внедрении

Утверждаю:

Генеральный директор ООО
«Красная рамка»

_____ В. А. Кречетов

05 июня 2020 года

АКТ

о внедрении результатов производственной практики Никонова Владислава Алексеевича в практическую деятельность предприятия

Настоящий акт подтверждает использование результатов работы Никонова В. А. при разработке фреймворка для автоматизации тестирования веб-приложений, программного обеспечения для конфигурирования и разворачивания фреймворка для автоматизации тестирования веб-приложений, а также сопутствующей документации: регламент использования разработанного программного обеспечения и регламент автоматизации тестирования в процессе разработки веб-приложений.

Актуальность работ вызвана необходимостью внедрения автоматизированного тестирования в процесс разработки веб-приложений, с целью повышения качества разработки и эффективности отдела тестирования в компании ООО «Красная рамка».

Руководитель проекта
Генеральный директор ООО «Красная рамка» _____ В. А. Кречетов