

Министерство науки и высшего образования Российской Федерации  
федеральное государственное автономное образовательное учреждение  
высшего образования

«Пермский национальный исследовательский  
политехнический университет»

Факультет \_\_\_\_\_ прикладной математики и механики  
Направление подготовки \_\_\_\_\_ 09.03.02 Информационные системы \_\_\_\_\_ и  
Профиль \_\_\_\_\_ Информационные системы и технологии  
Кафедра \_\_\_\_\_ Вычислительной математики, механики и \_\_\_\_\_

Допускается к защите  
Зав. кафедрой \_\_\_\_\_  
\_\_\_\_\_ (\_\_\_\_\_) \_\_\_\_\_  
«\_\_» \_\_\_\_\_ 20\_\_ г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

На тему Организация процесса автоматизированного тестирования веб-приложений с  
использованием языка программирования Python  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

Студент \_\_\_\_\_ Соболев Евгений Валерьевич \_\_\_\_\_ (\_\_\_\_\_)

Состав ВКР:

1. Пояснительная записка на 88 стр.

Руководитель ВКР

\_\_\_\_\_ (\_\_\_\_\_)

Консультант

\_\_\_\_\_ (\_\_\_\_\_)

Пермь 2023 г.

Министерство науки и высшего образования Российской Федерации  
федеральное государственное автономное образовательное учреждение  
высшего образования

«Пермский национальный исследовательский  
политехнический университет»

Кафедра \_\_\_\_\_ ВММБ \_\_\_\_\_

«УТВЕРЖДАЮ»

Зав.кафедрой \_\_\_\_\_  
« \_\_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**  
**на выполнение выпускной квалификационной работы бакалавра**

Фамилия И.О. \_\_\_\_\_ Соболев Евгений Валерьевич \_\_\_\_\_

Факультет \_\_\_\_\_ ФПММ \_\_\_\_\_ Группа \_\_\_\_\_ ИСТ-19-1БЗУ \_\_\_\_\_

Начало выполнения работы \_\_\_\_\_ 19 декабря 2022 \_\_\_\_\_

Контрольные сроки просмотра работы кафедрой \_\_\_\_\_ 24 января 2023 \_\_\_\_\_

Сроки представления на рецензию \_\_\_\_\_ 24 января 2023 \_\_\_\_\_

Защита работы на заседании ГЭК \_\_\_\_\_ 31 января 2023 \_\_\_\_\_

1. Форма и наименование работы: (дипломный проект/дипломная работа)

Дипломный проект, Организация процесса автоматизированного тестирования  
веб-приложений с использованием языка программирования Python

2. Исходные данные к работе \_\_\_\_\_

3. Содержание пояснительной записки \_\_\_\_\_

а.) основная часть (конструкторская, технологическая, исследовательская) \_\_\_\_\_

1 Рациональность внедрения системы автоматизации тестирования

2 Анализ и подбор технологий

3 Архитектура разрабатываемого решения

4 Тестируемое приложение

5 Разработка

Заключение

4. Перечень графического материала

Приложение А

Приложение Б

Приложение В

Приложение Г

Приложение Д

Приложение Е

Приложение Ж

Приложение З

5. Дополнительные указания

6. Основная литература

Как тестируют в Google Дж. Уиттакер, Дж. Арбон, Дж. Каролло - СПб.: Питер, 2014.

Тестирование программного обеспечения. Базовый курс. Святослав Куликов - 3-е издание. - : EPAM Systems, 2022.;

Python Testing with pytest Брайан Оккен - 2-е издание. , 2022;

API Testing and Development with Postman Dave Westerveld- 1-е издание. - Packt Publishing, 2021;

Design Patterns for High-Quality Automated Tests: High-Quality Test Attributes and Best Practices Anton Angelov - 1-е издание. - : On Kindle Scribe, 2020;

Learning DevOps: The complete guide to accelerate collaboration with Jenkins, Kubernetes, Terraform and Azure DevOps Mikael Krief - 2-е издание. - : Packt Publishing, 2022;

UI Testing with Puppeteer Dario Kondratiuk - 1-е издание. - Packt Publishing, 2021.

Руководитель выпускной квалификационной работы бакалавра

\_\_\_\_\_( \_\_\_\_\_ )  
(должность, Ф.И.О.)

Консультант \_\_\_\_\_( \_\_\_\_\_ )  
(должность, Ф.И.О.)

Задание получил \_\_\_\_\_( \_\_\_\_\_ )  
(дата и подпись студента)

## Оглавление

Введение.....	6
Постановка задачи.....	7
1 Рациональность внедрения системы автоматизации тестирования .....	8
2 Анализ и подбор технологий .....	13
3 Архитектура разрабатываемого решения.....	23
4 Тестируемое приложение .....	30
5 Разработка .....	32
5.1 Генерация данных .....	32
5.2 Работа с базой данных MySQL.....	36
5.3 Разработка mock .....	40
5.4 Разработка API тестов .....	43
5.5 Разработка UI тестов.....	48
5.6 Отчёты Allure.....	54
5.7 Docker .....	58
5.8 Jenkins CI.....	63
Заключение .....	67
Список литературы .....	68
Приложение А .....	72
Приложение Б .....	73
Приложение В.....	75
Приложение Г .....	76
Приложение Д.....	80
Приложение Е.....	85
Приложение Ж.....	86

Приложение 3 .....	89
--------------------	----

## **Введение**

В современном мире одним из важнейших элементов бизнеса являются веб-приложения. Требования к ним постоянно меняются, и возникает необходимость в их постоянной доработке. А в гибкой разработке, которая внедряется в последнее время в большинстве компаний, процесс создания продукта принимает итеративно – инкрементальный характер, что в свою очередь требует постоянного проведения регрессионного тестирования из-за внедрения новых компонентов. Данное тестирование может занимать большое количество времени квалифицированных специалистов и затягивать время доставки ценности владельцу бизнеса, что приводит к большим финансовым потерям. В данной работе была разработана система, автоматизирующая процесс регрессионного тестирования и позволяющая экономить время и финансы владельцам ИТ-бизнеса.

Объектом исследования являются средства и методы для организации автоматизированного тестирования веб-приложений.

Предметом исследования является функциональные особенности, совместимость средств и методов для использования в процессе организации автоматизированного тестирования веб-приложений.

Целью дипломного проекта является разработка на основе проведённого анализа комплексного решения для автоматизации тестирования веб-приложений.

Так как кроссфункциональность команды является одним из ключевых навыков, то для комфортной разработки основной язык программирования должен совпадать с языком, используемым при разработке веб-приложения. В данной работе таким языком программирования является Python 3.

## **Постановка задачи**

На основе имеющихся средств, инструментов и методов для автоматизации тестирования веб-приложений необходимо разработать систему для автоматизации веб-продукта, представляющую собой готовую инфраструктуру для автотестов и сами тесты, написанные на Python 3.

Для достижения поставленной цели необходимо:

- Оценить рациональность внедрения комплексного решения.
- Проанализировать инструменты, средства и методы, необходимые для разработки комплексного решения.
- Подобрать тестовое приложение для автоматизации тестирования, упакованное в контейнер и имеющее документацию.
- Разработать архитектуру системы автоматизированного тестирования.
- Разработать автотесты с учётом архитектуры и разбиения на уровни тестирования.
- Интегрировать решения для автоматического составления отчётной документации.
- Упаковать проект для дальнейшего развёртывания в CI.
- Интегрировать разрабатываемую систему в CI.

## **1 Рациональность внедрения системы автоматизации тестирования**

Специалисты по автоматизации тестирования SDET (Software Development Engineer in Test) помогают ускорить проведение процессов тестирования, а значит, быстрее выпускать свежие релизы IT-продукта [1]. В основном, это необходимо в масштабных приложениях с большим количеством бизнес-функций. Но часто встаёт вопрос, как бизнесу понять потребность в автоматизации процессов тестирования, с чего начать и как оценить эффективность результатов. Рациональность внедрения такого тестирования рассматривается во многих работах, например, [5, 7, 30].

При работе с масштабными IT-решениями, например, с собственными системами ERP, как на предприятии ООО "Камкабель" важно постоянно тестировать не только работу отдельных функциональностей, но и их взаимодействие. В условиях сжатых сроков, когда каждый месяц обновляются приложения, проверить все вручную невозможно – на это как минимум не хватит времени.

При создании IT-продуктов для бизнеса обычно сочетают два подхода [2]:

- осуществляют проверки вручную с помощью специалистов по обеспечению качества (QA);
- комбинируют ручное тестирование и автоматизацию отдельных тест-кейсов, смоук- и регрессионных тестов.

На предприятии ООО "Камкабель" процесс тестирования построен с использованием концепции, близкой ко второму подходу. Автоматизация тестирования на предприятии вводилась для следующих задач:

- автоматизация рутинных и частых проверок, снижение нагрузки на QA-специалистов;
- контроль основных функций приложения и отслеживание изменений в продукте;



- возможность проводить тестирование с большим количеством мобильных устройств, версий браузеров и операционных систем.

Существовал ряд признаков, указывающих на то, что необходимо задуматься об автоматизации тестирования на проекте. Наличие их являлось маркерами необходимости этого процесса.

Так как разработка проекта рассчитана на постоянную доработку и поддержку, а команда проекта занимается продуктовой разработкой длительное время, то у неё накопилась база стабильного и неизменного функционала, тестирование которого необходимо автоматизировать.

Также в сторону автоматизации процессов тестирования указывало наличие продолжительных регрессионных тестов. Повторные проверки (регрессионные тесты) занимали 3-4 дня и более, автоматизация тестирования помогла ускорить этот процесс за счет параллельного запуска, ночных прогонов и автоматической генерации отчетов.

Большое количество багов выявлялось на поздних этапах тестирования. Внедрение автоматизации не решило такого рода проблемы на 100%, поскольку многое зависит от проекта и процессов в нем. Автоматизации тестирования внесла существенное преимущество – теперь тесты могут запускаться в любое время, в том числе на этапе разработки. Это позволило выявлять возможные проблемы и ошибки раньше, чем задача передаётся в тестирование.

Тест-кейсы требовали создания большого количества тестовых данных и заполнения больших форм. В данном случае автоматизация тестирования решила проблему человеческого фактора. Автотест выполняет каждый раз одинаковую последовательность действий и проверяет один и тот же ожидаемый результат. Кроме того, заполнение и генерация данных в автоматическом режиме выполняется в разы быстрее, чем в ручном.

После того, как решение о внедрении автоматизации было принято, определялась цель внедрения автоматизации тестирования, а также объект тестирования, ресурсы и процессы.

Уменьшение времени от постановки задачи до выпуска приложения на production. SDET-специалисты помогли сократить время на тестирование устоявшейся функциональности приложения. Вместо того, чтобы проходить регрессионные кейсы руками, они автоматизировались. Прогон автотестов занимает в разы меньше времени, чем ручная работа. Кроме того, его можно запланировать на ночное время и запустить в несколько потоков, что позволяли аппаратные ресурсы, имеющихся у компании.

Улучшение качества продукта и оптимизация затрат на тестирование. За счет автоматизации регрессионных кейсов отпала необходимость в найме специалистов по ручному тестированию для прохождения постоянно увеличивающегося на проекте регресса.

Прежде чем приступать непосредственно к автоматизации тестирования, были проанализированы условия, в которых предстоит работать, и уточнено, готовы ли необходимые ресурсы для старта внедрения.

Выяснялось, насколько вписываются ожидания начальства о времени запуска автоматизации в предполагаемые затраты на эти работы. Ожидалось, что затраты на автоматизацию тестирования начнут окупаться спустя полгода после старта. Поэтому уточнялись также масштабы долгосрочности проекта, так как ERP-система является одним из ключевых факторов в работе предприятия, то и решение о необходимости автоматизации тестирования было получено в кратчайшие сроки. Процесс внедрения автоматизации процессов тестирования проходил в течении полугода, данный срок был согласован с руководством и техническим директором.

Состав команды SDET учитывал то, что у компании уже есть свои специалисты по автоматизации тестирования, кроссфункциональность команды позволяла задействовать программистов для внедрения процессов автоматизации, дополнительные специалисты SDET также были нужны и имело место подключения к уже существующей команде “на усиление” дополнительных специалистов SDET по договору краткосрочного найма.

Процесс CI/CD на проекте был выстроен, настройкой окружения для тестирования и стенды для тестирования, доступ к ним осуществлялся специалистами SDET совместно с системными администраторами.

При реализации проекта собирались метрики эффективности автоматизации, чтобы в последствии проанализировать их и рассчитать процент эффективности.

Собираемые метрики можно разделить на две ключевые группы. Это:

- Количество часов, затраченных QA на прохождение тест-кейса.
- Количество часов, затраченных SDET на автоматизацию (актуализацию) тест-кейса.

Упрощенная формула расчета выглядит как [2]:

$$\frac{t_{qa}}{t_{sdet}} * 100\%$$

Так как автоматизация тестирования на проекте внедрялась впервые, то ожидаемая экономия времени и других ресурсов за год в среднем составила 140-150%. При этом эффективность автоматизации нарастала пропорционально времени ее использования, в частности, до 240% к концу второго года.

Показатель экономии ресурсов со временем начал снижаться, поэтому в компании запланирован аудит тестирования и автоматизации тестирования для выявления возможных проблем, ошибок и узких мест.

Для расчета эффективности автоматизации большую роль сыграл источник достоверной информации о временных затратах на автоматизацию тестирования. В частности, источником данных была система таск-трекинга Jira.

На проекте присутствовало порядка 700 тест-кейсов, каждый из них проходили от 70 до 100 раз в год. Возможность автоматизации была у 75% кейсов, остальные требовали проверки вручную.

Затраты времени:

- 30 часов при ручной проверке всех кейсов.

- 8 часов после автоматизации (ночной прогон тестов без участия человека).

Помимо ночного прогона, требовалось около 8 часов на проверку тех кейсов, которые невозможно было покрыть автоматизацией тестирования, и 6 часов – на анализ результатов автотестов и проверку отказов в случае необходимости.

Таким образом, автоматизация тестирования была актуальным и рациональным решением, позволила снизить затраты времени специалистов с 30 до 14 часов. В среднем, этот подход на данный момент позволяет экономить как минимум от 30 до 50% времени и уделить больше внимания развитию и улучшению продукта.

## 2 Анализ и подбор технологий

Команда занимается разработкой web-ориентированной части ERP-системы. Основным языком программирования – Python [24]. Для разработки BE части приложения используется фреймворк Flask [11], для написания front-end части веб-приложения используется React. База данных, используемая в ERP-системе, – MySQL [20].

Само приложение можно разделить на несколько основных компонентов.

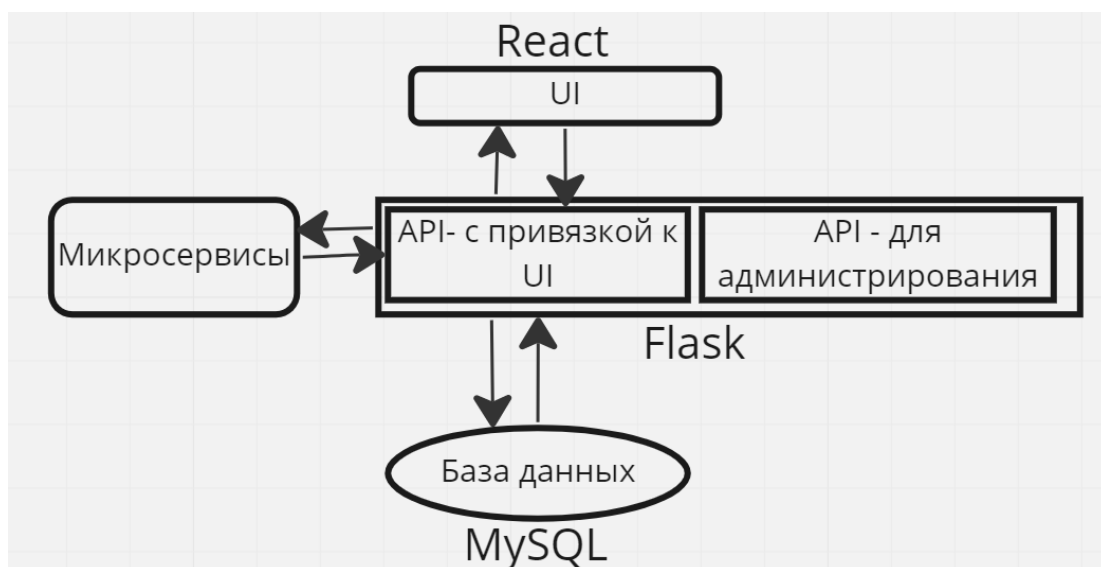


Рисунок 1- Компоненты веб-приложения

- UI интерфейс;
- API, подразделяющееся на используемое в работе веб-приложения и вспомогательное, используемое в основном для администрирования без привязки к конкретному UI интерфейсу;
- микросервисы, из которых приложение получало информацию от контрагентов;
- базы данных с информацией компании.

Исходя из компонентов веб-приложения, при организации автоматизированного тестирования для каждого из компонентов, необходимо внедрять свои виды тестирования и технологии:

End to end – тесты для покрытия UI части веб-приложения. Тестирование интерфейса помогает проверить функции приложения, имитируя действия пользователей, то есть в процессе тестирования выполняется проверка элементов, проверка на их корректность с помощью ввода данных в приложение через средства автоматизированного UI-тестирования [31]. Данные средства необходимы в разрабатываемом решении и их следует подобрать.

Интеграционные тесты для покрытия API приложения позволяют проверить правильность взаимодействия компонента с внешними ресурсами, например с базой данных [6]. С помощью утверждений можно тестировать API компоненты и побочные эффекты от таких операций, как ввод-вывод в базах данных, ведение журнала и т.д. Обычно для этого требуется только среда тестирования и язык программирования, позволяющий написать клиент для вызова методов API. Из этого возникает необходимость в среде для тестирования на языке программирования Python, умеющей запускать тесты параллельно с возможностью параметризации тестов.

Для изоляции приложения от микросервисов необходимы mock, они помогают имитировать и изучать исходящие взаимодействия. Mock - это вызовы, совершаемые тестируемой системой к ее зависимостям для изменения их состояния [33]. Зачастую они представляют из себя мини-приложения заглушки, написанные с помощью бэкенд-фреймворка.

Для работы с базой данных MySQL необходимо решение, позволяющее в изоляции от особенностей СУБД MySQL использовать разрабатываемое решение не только с приложениями, использующими MySQL, но и с другими (к примеру, PostgreSQL [22]). Для этого необходима технология ORM, с ней намного проще работать и есть возможность находить ошибки статическим анализом, при этом процесс типизирования объектов становится легче. ORM позволяет описывать структуры баз данных и способы взаимодействия с ними на языке Python без использования SQL [29].

Для создания тестовых данных необходима библиотека, позволяющая генерировать объекты разных типов.

Для создания отчетов по результатам проведённых тестов необходим инструмент для генерации отчётов, который позволял бы собирать подробную информацию о тестировании и тестовые артефакты для скорейшего выявления проблем в случае их появления при тестировании.

После создания и разработки решения автоматизированного тестирования его необходимо упаковать для дальнейшего развёртывания и запуска в CI. Для этого необходимо подобрать инструмент, позволяющий сделать это.

Для автоматического запуска и развёртывания тестов необходимо применить систему непрерывной интеграции CI. CI – это практика разработки программного обеспечения [16], в которой участники команды часто объединяют результаты своей работы друг с другом: от одного до нескольких раз в день. Каждая интеграция проверяется автоматической сборкой и тестированием для определения ошибок настолько быстро, насколько это возможно.

Весь разрабатываемый процесс схематично изображен на рис. 2.



Рисунок 2 – Схема разрабатываемого процесса автоматизации тестирования

После определения необходимых для разработки системы инструментов и технологий, к ним необходимо предъявить требования для того, чтобы в ходе конкурентного обзора и анализа можно было выявить наиболее подходящее решение именно для системы автоматизированного тестирования веб-приложений на языке программирования Python с учётом уже используемого при разработке приложения инструментария.

Сначала необходимо подобрать среду для тестирования, так как она используется для запуска всех видов тестов. Критерии для сред тестирования на языке программирования Python:

1. среда тестирования выводит в консоль подробный отчёт о результатах, в том числе и не пройденных тестов с указанием конкретного места;
2. использует максимально похожий на стандартный синтаксис из языка программирования Python;
3. имеет возможность создания фикстур разных уровней (класса, теста, тестового набора);
4. имеет возможность параметризовать тесты;
5. имеет возможность маркировать тесты;
6. имеет возможность параллельного запуска тестов;
7. имеет поддержку, наличие документации
8. лицензия, стоимость продукта

Для языка программирования Python присутствует всего две основные среды тестирования — это unittest и pytest.

Unittest является частью стандартной библиотеки языка Python [32], поэтому дополнительно устанавливать ничего не нужно. Отсюда, казалось бы, следует, что и синтаксис должен быть максимально похожий на Python, но, по сравнению с pytest, для проведения тестирования придётся написать достаточно большое количество кода. Так как разработчики вдохновлялись форматом библиотеки JUnit, названия основных функций написаны в стиле camelCase, например setUp и assertEquals, что не совсем удовлетворяет второму в списке критерию. В pytest синтаксис позволяет писать более компактные по сравнению с unittest наборы тестов, используя при этом стандартный синтаксис Python [12].



```

=====
FAIL: test_on_shuffled_range (test_k_stat.TestStat)
-----
Traceback (most recent call last):
  File "/home/sergey/PycharmProjects/test_k_stat/test_k_stat.py", line 21, in
test_on_shuffled_range
    self.assertEqual(kth_stat(1i, 3), 2)
AssertionError: 3 != 2

```

Рисунок 3 – Вывод логов упавшего теста в unittest

```

example.py ..F                                     |100%|

===== FAILURES =====
_____ test_multiple_root _____

    def test_multiple_root():
        res = square_eq_solver(2, 5, -3)
>       assert len(res) == 3
E       assert 2 == 3
E       + where 2 = len([0.5, -3.0])

example.py:116: AssertionError
===== short test summary info =====
FAILED example.py::test_multiple_root - assert 2 == 3

```

Рисунок 4 – Вывод логов упавшего теста в pytest

Если посмотреть логи, получаемые при тестировании обеими средами, то можно заметить, что у pytest, при указании причины падения теста, помимо самого сравнения, выводится еще и подробная трассировка значений, которая может помочь в решении проблемы, возникшей в ходе падения теста, что говорит в пользу pytest при сравнении по первому критерию.

Возможности маркировать тесты имеется в обеих средах тестирования [12, 32], что делает среды равными при сравнении по пятому критерию.

Фикстуры имеются как в unittest, так и в pytest. В unittest предоставляется три области видимости - фикстуры могут быть определены на уровне отдельного теста, класса или модуля [32]. В pytest областей видимости больше, фикстуры могут быть функциональными, класса, модуля, пакета и сессии [12, 25]. Что при сравнении даёт небольшое преимущество pytest при сравнении по третьему критерию.

Параметризировать тесты умеет только pytest [12, 25], у unittest такой возможности нет [32].

Параллельного запуска теста, встроенного в среду тестирования нет у обеих сред, но у каждой из них есть дополнительные плагины, которые позволяют запускать тесты параллельно. У unittest это плагин concurrencytest, у pytest - pytest-xdist. Работают они одинаково, поэтому по 6 критерию преимуществ у какой либо из сред не выявлено.

Поддержка и документация имеется у обеих сред тестирования. Unittest получает обновления вместе с обновлениями языка программирования Python [32]. Pytest обновляется разработчиками независимо от языка, но также регулярно, обновления всегда можно загрузить с помощью менеджера пакетов pip [12].

Обе среды имеют MIT лицензии, что позволяет свободно и бесплатно использовать их при разработке [12, 32].

В итоге, при сравнении сред тестирования по составленным критериям предпочтение отдаётся pytest.

После подбора среды тестирования можно приступать к выбору инструментов для написания разных видов тестов. Рассмотрим самый высокий уровень тестирования - UI [31]:

Критерии выбора средства автоматизированного UI-тестирования:

1. поддержка языка программирования Python для написания сценариев тестирования;
2. поддержка тестирования веб-приложений;
3. лицензия, стоимость инструмента;
4. возможность кроссбраузерного тестирования;
5. поддержка запуска тестов в CI;
6. одновременное выполнение нескольких тестов;
7. получение детальных отчётов;
8. поддержка, наличие документации.

Средств для автоматизации функционального тестирования UI большое количество, сравнение основных из них удобнее всего представить в виде таблицы, для сравнения будут браться только те средства, которые удовлетворяют первому и второму критерию, так как выполнение этих требований является ключевым фактором для подбора инструмента:

Таблица 1 – Сравнение инструментов автоматизации UI-тестирования

	Eggplant	Robot Framework	Selenium	TestComplete	Tricentis Tosca	Worksoft Certify
Критерий 1	+	+	+	+	+	+
Критерий 2	+	+	+	+	+	+
Критерий 3	Платно	Бесплатно	Бесплатно	Платно	Платно	Платно
Критерий 4	+	+	+	+	+	+
Критерий 5	-	-	+	+	+	+
Критерий 6	-	-	+	-	-	-
Критерий 7	+	-	+	+	+	-
Критерий 8	+	+	+	+	+	+

После анализа и сравнения средств было выявлено, что полностью всем критериям удовлетворяет лишь одно решение – Selenium. Данное решение является бесплатным для коммерческого использования и распространяется свободно [27].

Для генерации фиктивных данных при проведении тестирования на Python есть две основные библиотеки: Faker и Mimesis. Оба решения дают возможность генерации любых персональных данных, как по заданному шаблону, так и по выбранной категории [10, 18]. Поэтому при сравнении ключевым фактором стала популярность данных библиотек среди сообщества программистов. Более популярной библиотекой является Faker [10].

Для написания mock самым подходящим решением является Flask, этот фреймворк уже используется при разработке тестируемого приложения, и решение о его использовании не требует добавления дополнительного

инструмента в стек технологий, что стало ключевым фактором при выборе данного фреймворка.

Для работы с базой данных MySQL с применением технологии ORM в языке программирования Python на данный момент существует библиотека sqlalchemy. Она, по сравнению с аналогами, позволяет использовать преимущества наследования и полиморфизма, при этом не загоняет в рамки задач начального уровня своими архитектурными ограничениями, а наоборот даёт максимум возможностей [28]. Отличная документация и большая поддержка сообщества делает это решение уникальным на фоне других конкурирующих аналогов, таких как Peewee [21].

После выбора всех вспомогательных технологий для написания тестов, необходимо выделить основные критерии для подбора средства составления отчётов по результатам тестирования:

- Возможность интеграции pytest.
- Возможность интеграции с Selenium.
- Наличие плагина для использования в CI.

Под все три критерия подходит три средства: Zephyr, ALM Octane, Allure. Zephyr является плагином для Jira, его использование в отдельности является невозможным, при этом сам плагин является платным, оба этих фактора затрудняют его использование в разрабатываемом решении [34].

ALM Octane позволяет осуществлять планирование, тестирование и контроль на всех этапах разработки [4]. Инструмент имеет много функционала, который в последствии не будет использоваться, но наличие этого функционала оказывает влияние на цену данного продукта и делает его использование в системе автоматизированного тестирования веб-приложения не рациональным, так как необходимо решение только для генерации отчётов о результатах тестирования.

Allure версия Enterprise как и в других сравнимых инструментах платная, но функционал, который она даёт в разрабатываемом решении полностью не

нужен, так как тест-кейсы для ручного тестирования в системе автоматизированного тестирования не используются. В контексте разрабатываемого решения будет достаточно инструмента с открытым исходным кодом Allure report [3]. Данный инструмент - урезанная версия Enterprise, которая специализируется на возможности генерации отчётов по результатам выполнения автотестов, при этом данный инструмент создаёт информативные графические отчеты, что позволяет всем, кто участвует в процессе разработки, извлекать максимум полезной для устранения ошибки информации.

Следующее, что необходимо подобрать и проанализировать — это платформы для упаковки и развёртывания проектов. Критерии выбора для средства упаковки и развёртывания проектов писать не имеет смысла, так как лидирующее место на рынке занимает инструмент Docker [9]. Огромная библиотека образов Docker Hub Container Image Library, в которой содержатся образы для работы с Selenium, такие как selenoid и selenoid-ui, которые позволяют запускать UI-тесты в изолированной среде параллельно и следить за процессом тестирования, при этом поддерживая большие нагрузки без дополнительных затрат ресурсов, поднимая всё необходимое тестовое окружение. Аналогичного решения у других инструментов нет.

Остаётся подобрать инструмент для автоматического запуска разрабатываемого решения. Критерии выбора инструмента для CI:

- лицензия, стоимость продукта;
- поддержка Docker container;
- плагин для работы с allure report.

Под данные критерии было подобрано три инструмента – Jenkins CI, GitLab CI, Go. При выборе инструмента было проведено их сравнение по функциональности, открытости, гибкости, расширяемости и документации. В результате выбор пал на Jenkins CI, в силу того что:

Функциональность – в сравнении ПО [13,14,15] для непрерывной интеграции Jenkins CI является продуктом, обладающим одним из самых широких списков функций и поддерживаемых платформ.

Открытость – Jenkins CI обладает открытой лицензией, что позволяет любому как улучшать саму систему, так и расширять её возможности посредством дополнений.

Гибкость – благодаря своим преимуществам Jenkins CI получил огромную поддержку среди разработчиков ПО с открытым исходным кодом [15]. В частности, для Jenkins CI написано множество дополнений, позволяющих настроить непрерывную интеграцию практически в любом процессе разработки ПО.

Расширяемость – Jenkins CI является, по своей сути, основой для настройки пользовательского процесса непрерывной интеграции [15]. Для того, чтобы Jenkins CI гармонично вписался в пользовательский процесс, необходима либо установка подходящих расширений, либо разработка собственных.

Одним из важнейших параметров при выборе платформы стала хорошая документация, позволяющая быстро найти ответы на вопросы, возникающие во время разработки дополнений, а также содержащая подробные инструкции для тех, кто только начинает разрабатывать под Jenkins CI.

Итогом сравнения и анализа является следующий набор средств и инструментов, необходимых для разработки системы автоматизированного тестирования веб-приложений на языке программирования Python:

- среда тестирования pytest с плагином pytest-xdist;
- средство для функционального UI-тестирования Selenium;
- средство для генерации отчётов allure-report;
- библиотека для генерации тестовых данных Faker;
- фреймворк Flask для написания Mock;
- платформа для упаковки и доставки приложений Docker;
- инструмент для организации процесса CI - Jenkins CI.

### 3 Архитектура разрабатываемого решения

Рассмотрим файловую структуру тестового проекта, который в последствии будет запускаться в CI. В корневой папке 6 разделов: api, generators, mysql, ui, tests, mock. Помимо разделов там также находятся restructions.py, urls.py, start\_test.sh, pytest.ini.

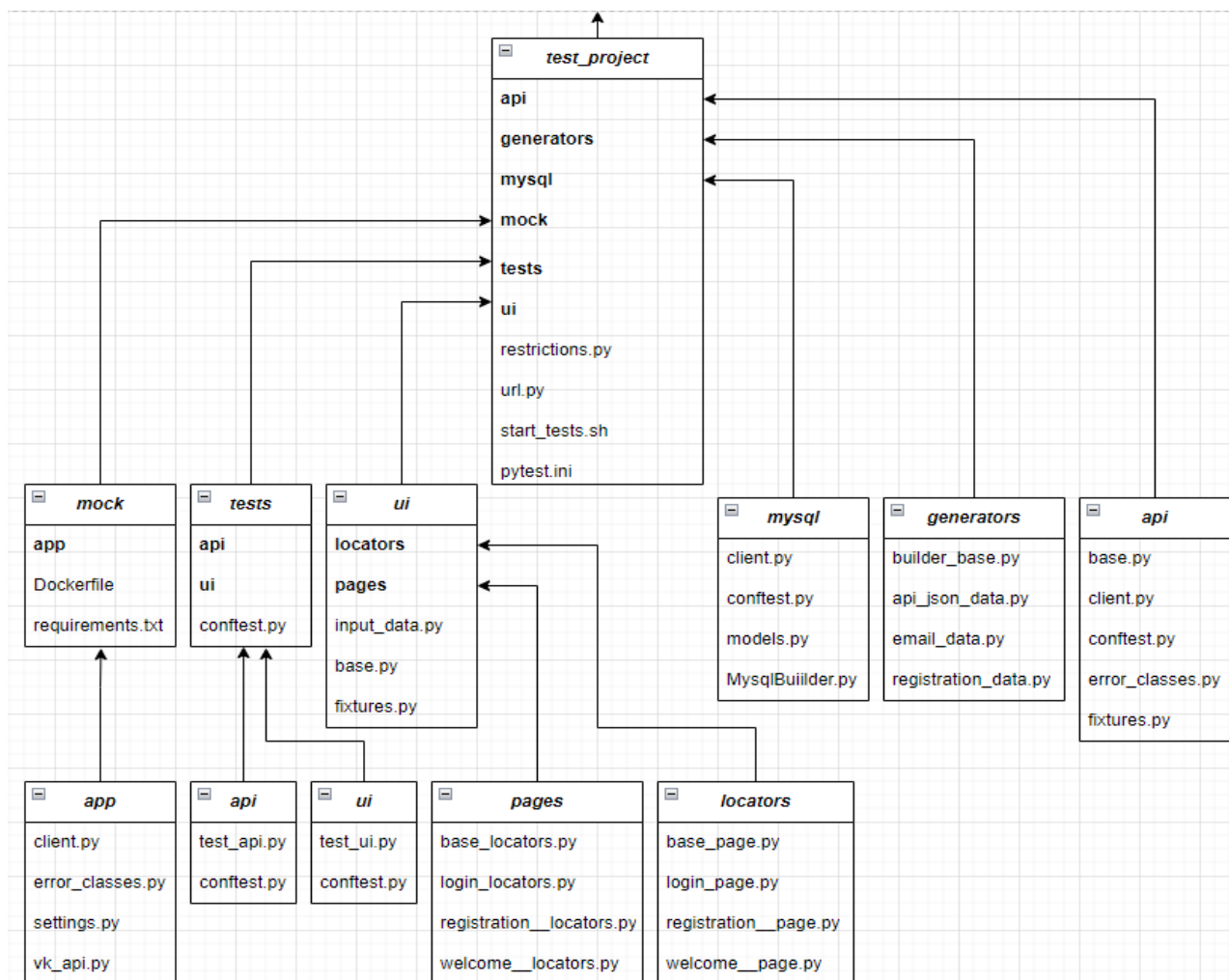


Рисунок 5- Файловая структура тестового проекта

Рассмотрим также некоторые понятия, которые в дальнейшем будут использоваться при описании.

Fixtures — это функции, выполняемые pytest до, а иногда и после фактических тестовых функций [25]. Код в фикстуре может делать все, что необходимо для начала или окончания тестирования. К примеру, можно их использовать, чтобы получить набор данных для тестирования или получить

систему в известном состоянии перед запуском теста. Fixtures также используются для получения данных сразу для нескольких тестов. В разрабатываемом решении они всегда хранятся отдельно в файле `fixtures.py` и при необходимости совместного использования подключаются к файлу `conftest.py`

`Conftest.py` – это файл, который `pytest` считывает в первую очередь в зависимости от иерархии каталогов, в котором он находится [25]. В данном файле обычно хранятся настройки, предназначенные для тех тестов, в папке с которыми он находится. Если же необходима настройка сразу для нескольких подкаталогов, то `conftest.py` должен лежать на уровне с данными подкаталогами.

Всё необходимое для разработки `api`-тестов хранится в директории `api`, это:

- `base.py`. Необходим для того, чтобы в дальнейшем вынести базовый функционал клиента `API` и наследоваться от него при разработке других `api` клиентов.
- `client.py` содержит клиент для вызова `api` тестируемого приложения.
- `conftest.py` содержит все настройки и подключенные фикстуры.
- `Error_classes.py` содержит классы с разными видами ошибок, вызываемые при исключениях.
- `fixtures.py` содержит фикстуры, необходимые для разработки `api`-тестов.

В директории `generators` находятся генераторы данных. Каждый из генераторов будет создавать тестовые данные определённого типа:

- `api_json.py` генерирует данные для `api`-тестов;
- `email_data.py` хранит генератор `email` адресов разных типов;
- `registration_data.py` хранит генератор данных, необходимых для регистрации;



- `builder_base.py` хранит базовый класс генератора, в котором определён основной функционал, необходимый для всех генераторов.

В директории `mysql` находится всё, что необходимо для работы с базой данных:

- `client.py` содержит всё необходимое для подключения, настройки и создания базы данных и сессий.

- `models.py` содержит ORM модели таблиц базы данных.

- `MysqlBuilder.py` содержит готовые запросы для работы с базой данных.

- `conftest.py` содержит настройки и фикстуры, необходимые для работы с БД.

- В папке `ui` хранятся всё необходимое для `ui`-тестов, а именно:

- `locators` – директория с локаторами для нахождения `ui` объектов, созданных с разбиением на тестируемые страницы.

- `pages` – директория с функциями, разбитыми на страницы с помощью паттерна `PageObject` (каждый файл будет хранить функциональность необходимую только для работы с определённой страницей веб-приложения).

- `input_data.py` содержит вынесенные из тестов громоздкие входные параметры для тестов, чтобы сохранять хорошую читаемость файлов с тестами.

- `base.py` хранит всю базовую вспомогательную функциональность и часть настроек для `ui`-тестов.

- `fixtures.py` содержит фикстуры, необходимые для разработки `ui`-тестов.

- В директории `tests` хранятся тесты как `api`, так и `ui`:

- `api` – директория с `api`-тестами, содержащая в себе файлы `test_api.py` и `conftest.py` с настройками и фикстурами именно для `api` тестов.

- `ui` – директорией с `ui`-тестами, содержащая в себе файлы `test_ui.py` и `conftest.py` с настройками и фикстурами для `ui`-тестов.

- `conftest.py` содержит настройки и фикстуры, необходимые для всех типов тестов.

Остаётся описать файлы, лежащие в корневой папке с тестовым проектом:

`restruictions.py` – переменные со значениями ограничений полей на UI и ограничений для значений, передаваемых в API.

`Urlis.py` – url-адреса всех страниц веб-приложения.

`pytest.ini` – основной файл конфигурации Pytest, который позволяет изменить поведение настроенного по умолчанию тестового проекта.

`start_test.sh` – скрипт на `bash`, который необходим для запуска тестового проекта в CI.

Помимо самого тестового проекта, в разрабатываемом решении будут директории для конфигурации и поддержки тестового окружения.

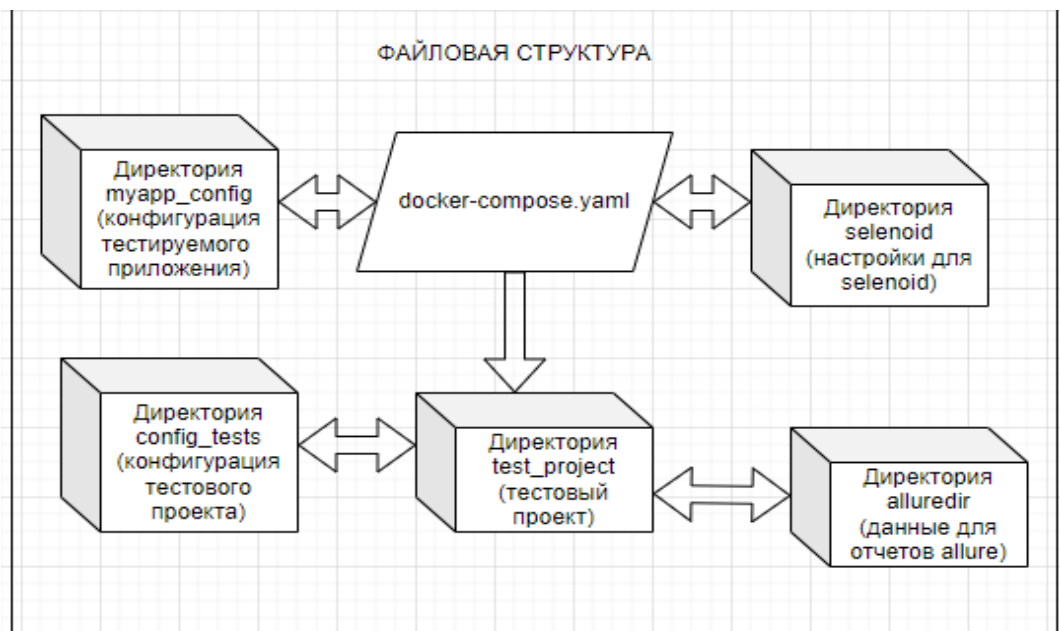


Рисунок 6 – Файловая структура разрабатываемого решения

Файловая структура содержит пять основных директорий:

- `test_project` – тестовый проект, в котором содержатся все рассматриваемые ранее файлы для организации тестирования.

- `config_tests` – содержит всё необходимое для настройки тестового проекта: `dockerfile`, описывающий логику развёртки тестового проекта и `requirements.txt` – список зависимостей, которые необходимы для работы тестового проекта.

- `myapp_config` – в данной директории находится всё нужное для запуска тестируемого приложения, а именно: `app_config.txt` – файл, из которого приложение при запуске берёт все необходимые параметры, директория `database`, которая содержит `dockerfile` для развёртки БД и `init_db.sql` – скрипт, необходимый базе данных для инициализации, и директория `nginx`, в которой хранится конфигурация для `nginx`-сервера, выступающего прокси-сервером для работы внутри созданной в `Docker` сети.

- В `Alluredir` лежат данные, которые сохраняет `allure`, они используются в дальнейшем для генерации отчётов о проведённом тестировании.

- `selenoid` – в данной директории хранится конфигурационный файл для настройки `selenoid`: файл с конфигурацией браузеров и их версий, которые будут использоваться при тестировании.

Рассмотрим архитектуру разрабатываемого решения на рис. 7, на нём также пронумерована последовательность запуска компонентов:

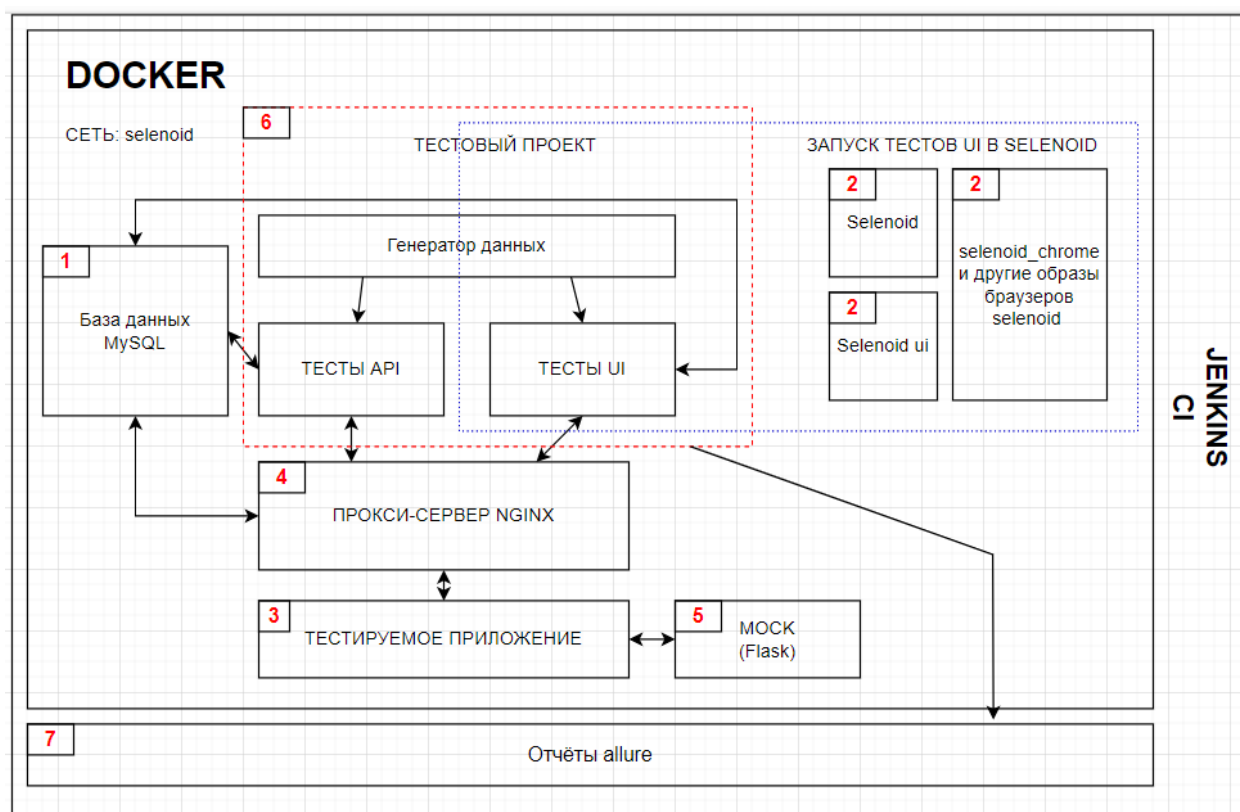


Рисунок 7 – Архитектура разрабатываемого решения

В первую очередь запускается база данных MySQL, так как данный компонент необходим как для работы тестируемого приложения, так и для работы тестов. Сразу за базой данных запускаются компоненты selenoid: сам selenoid, который будет использоваться для поднятия нод и организации функционирования драйверов; selenoid-ui, который будут давать возможность контролировать процесс тестирования в нодах, и образы драйверов браузеров selenoid, из которых будут создаваться ноды. Далее - тестируемое приложение. Так как внутри docker напрямую к нему обратиться не получится, необходимо сразу после него запустить прокси-сервер nginx [9], с помощью которого с приложением можно будет взаимодействовать, в том числе и драйверам, поднятым с помощью selenoid. Затем запускается mock-server, необходимый для полноценной работы приложения. После запуска всего необходимого стартует тестовый проект, в котором UI-тесты выполняются в selenoid, а API - тесты просто в тестовой среде. В процессе тестирования создаются тестовые артефакты, необходимые в дальнейшем для создания allure отчёта о

проведённом тестировании. В самом конце запускается в CI allure и генерирует отчёт по имеющимся тестовым артефактам.

Итогами данной главы являются:

- сформированная и полностью описанная файловая структура;
- архитектура разрабатываемого решения.

## 4 Тестируемое приложение

Для разработки решения требуется веб-приложение, соответствующее архитектуре, описанной в аналитической части данной работы. Так как реальный проект для демонстрации комплексного решения задействовать не удалось в силу действующих NDA, необходимо подобрать максимально похожее на реальный проект приложение, которое позволило бы раскрыть все возможности разрабатываемой системы и при необходимости внедрить её в любой другой проект, написанный на языке программирования Python 3. Оно должно иметь типичный для веб-приложений функционал.

Первый компонент, который обязательно должен присутствовать — это веб-сайт, имеющий UI-интерфейс с возможностью авторизации и регистрации. Практически ни одно веб-приложение не обходится без данных возможностей, поэтому наличие такого функционала становится необходимостью. Также должно быть состояние, в котором пользователь является авторизованным, то есть его личный кабинет, представляющий из себя лендинг с данными пользователя, взятыми из базы данных, и микросервис, который в разрабатываемом решении изолируется с помощью Mock. В личном кабинете должен быть набор UI элементов, таких как выпадающие списки, меню, кнопки для перехода на разные ресурсы, возможность выхода из личного кабинета.

Вторым значимым компонентом является api, позволяющая выполнять функции по администрированию веб-приложения, а именно:

- добавление пользователя;
- удаление пользователя;
- смена пароля пользователя;
- блокировка пользователя;
- разблокировка пользователя;
- статус работы веб-приложения.

Приложение должно быть упаковано в docker-образ, так как именно из него будет запускаться приложение для проведения автоматизированного

тестирования. Все тесты будут выполняться методом серого ящика, то есть для разработки тестов не нужно иметь доступ к исходному коду ПО, достаточно лишь документации и требований к приложению.

В интернете был найден подходящий по архитектуре шаблон веб-приложения и переделан под функциональные требования, необходимые для демонстрации разрабатываемого решения. Весь UI-интерфейс веб-сайта отображен в приложении А. Также была составлена документация для приложения, в которой подробно описаны функциональные особенности и требования к компонентам, в том числе задокументирована API [19].

Итогом данной главы является готовое тестируемое приложение, на котором будет демонстрироваться разрабатываемое решение по автоматизации тестирования веб-приложения и полноценная документация к нему.

## 5 Разработка

Для начала требуется создать решение для генерации данных, так как они потребуются и для UI-, и для API-тестов. Затем следует разработка вспомогательных модулей для работы с БД, эти модули потребуются для занесения сгенерированных тестовых данных в базу, а также для проверки записей и изменений, сделанных в базе данных с помощью функционала веб-приложения. После идёт разработка API-тестов, UI-тестов, а также всё необходимое для их запуска в CI и развёртывания тестовой инфраструктуры.

### 5.1 Генерация данных

Генерация данных помогает сделать тесты автономными. Не нужно заботиться о том, в каком состоянии находятся данные на тестовом стенде или в приложении, так как они всегда генерируются перед тестом заранее. По мере роста приложения вопрос дублирования в тестовом фреймворке может стать большой проблемой. Актуализация тестов с большим количеством дублирования, скорее всего, будет отнимать значительное время, генерация же данных позволяет изменять или добавлять только методы классов, а все созданные объекты изменяются автоматически, при этом практически отсутствует дублирование кода. Также повышается прозрачность тестового фреймворка, так как всегда хорошо видно, что конкретно создаётся в тесте без обращения в дополнительные источники информации.

Для реализации генератора данных была выбрана библиотека Faker. Для создания первого генератора потребуются следующие методы из данной библиотеки:

Метод `lexify` для генерации данных по заданному шаблону, с помощью него будут реализованы все основные данные: имя, фамилия, отчество, логин, пароль и т.д. Данная функция принимает параметры `text` и `letters`, в текст вписывается шаблон для генерации, в `letters` — алфавит, по которому будет заполняться шаблон [10].



Начнем с разработки базового класса генератора. В нём хранятся алфавиты, содержащие разные символы по категориям: кириллица, большие и маленькие буквы, латиница, также большие и маленькие буквы, цифры и спецсимволы. В базовом классе есть инициализатор с объектом класса Faker и переменной результата генерации. Результат генерации всегда словарь, так как именно его можно легко конвертировать в json. Далее необходим метод `update_inner_value`, позволяющий изменять сгенерированные объекты на любом уровне вложенности с возможностью создавать новые, и метод `build`, возвращающий сгенерированный объект. Алгоритм функционирования методов обозначен на рисунке 8.

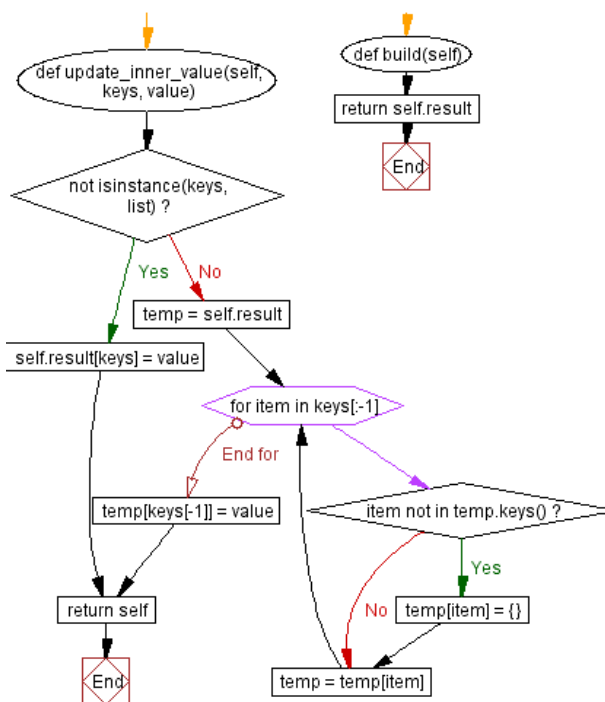


Рисунок 8- Методы build и update\_inner\_value

После создания базового класса генератора нужно создать классы, отвечающие за генерацию объектов для регистрации, авторизации, а также объектов для api запросов.

Класс `BuilderRegData` для генерации данных, необходимых при регистрации, состоит из инициализатора, в котором вызывается конструктор базового класса и метод `reset`, который осуществляет сборку объекта с заданными по умолчанию параметрами, который в последствии можно

изменять с помощью метода `update_inner_value`, унаследованного от базового класса. Основные методы класса отвечают за генерацию данных для полей регистрации, на вход они получают три параметра: `length`, `alphabet` и значение, которое при необходимости пользователь может установить вручную. Всего в классе 7 методов, которые в сумме генерируют 9 видов данных для формы регистрации. Один из методов отображен в виде диаграммы на рисунке 9.

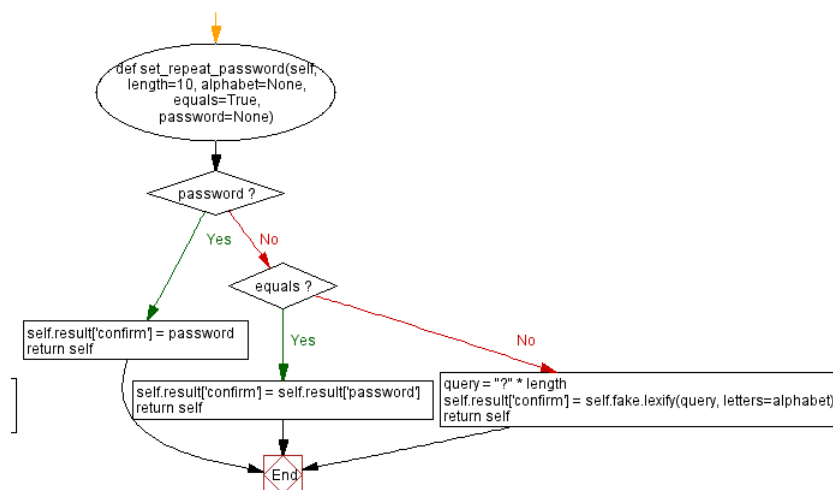


Рисунок 9 – Метод генерации пароля

По точно такому же принципу создаем и класс `BuilderApiData` для генерации объектов для запросов к `api`. В итоге, при их использовании получают объекты, отображенные на рисунке 10.

```

BuilderRegData().set_middle_name(255).set_username(16)
.set_email(email=self.bld_email.email_small_upper(64)).build(), # 1
BuilderRegData().set_name(45).set_surname(255).set_middle_name(255)
.set_username(6).set_email(email=self.bld_email.email_defis_name()).build(), # 2
BuilderRegData().set_name(45).set_middle_name(middle_name="null")
.set_email(6).set_password(255).set_repeat_password().build(), # 3
  
```

Рисунок 10 – Использование генератора регистрационных данных

Результат, полученный при генерации данных объектов, отображен на рисунке 11.

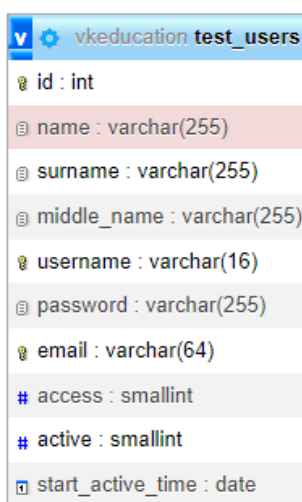
```
[{'name': 't0Df[P 4♦m', 'surname': '3zY)!vYzxa',
  'middle_name': 'DAPu}yg{C 3x▲щnQ1iAE`Cn71юlкрЪBd}FP;
  $PпыIG@wsX9BиJ#э]X&BA▲s!sГjβKE!БЮгрФЯояiТ4Ы06+pzle&aX{9x4
  {ьAU@пжПфюОстаКръьI$хялЦЁкчмсC1гАЗР:QWarцVЙRBh*г)
  Cβ+ЙРъOUэ+!ацо]vды0Ц)@ТβЩFZСжаКТоYОЖюуяю&Йэ8ФWч$клраСъv~XTxT
  кЁ0`ТМЯvЕъ9'УвГфUoPm)BβDo6oш)Щ@BCGYT2э`ол76лvpZ6ЮЁY}',
  'username': 'аилЪе*уБкГщЙ*3Й9', 'email':
  'pdwgbfадgomnposvcbbjfnttlugqESWYQHWEZUSUYEYODDDXODXGZG0V
  @TeSt.Qa', 'password': 'ня(♦ДувESc', 'confirm': 'ня
  (♦ДувESc'}], {'name': '8Фs~д5ЦАщиШqEYУг[Н`qЫn%73Рэм7β6WЧ!Ns
```

Рисунок 11-Визуализации сгенерированных данных для регистрации

Таким образом получен базовый класс, класс для генерации регистрационных данных и класс для генерации тел запросов api. При расширении функционала веб-приложения можно добавлять и другие классы-генераторы данных.

## 5.2 Работа с базой данных MySQL

База данных, согласно документации [19], состоит из одной таблицы, в которой хранятся данные о пользователях веб-приложения. В соответствии с документацией также заданы все ограничения полей. Для инициализации базы требуется скрипт, написанный на языке SQL, см. листинг Б.1 в приложение Б. Данный скрипт используется в docker-контейнере с БД MySQL [20], и запускается после запуска контейнера с БД. Обычно в команде разработки скрипт создаётся программистами и передается вместе с приложением в тестирование. По итогу запуска контейнера со скриптом создаётся база данных vkeducation с таблицей test\_users, см. рисунок 12.



vkeducation test_users	
id	int
name	varchar(255)
surname	varchar(255)
middle_name	varchar(255)
username	varchar(16)
password	varchar(255)
email	varchar(64)
access	smallint
active	smallint
start_active_time	date

Рисунок 12 – Таблица test\_users

После определения структуры рассмотрим разработку ORM моделей для работы с базой данных. Для реализации используется библиотека SQLAlchemy, которая может синхронизировать объекты Python и записи в MySQL.

Создадим ORM модели, которые представляют классы, соответствующие определению таблиц в БД, и объекты, которые хранятся в этих таблицах. Получилась модель TestUsersModel, соответствующая таблице test\_users. Класс модели обязательно наследуется от базового для предоставления ему метакласса, который производит объекты в соответствие с

sqlalchemy.schema.Table [28] и делает вызовы, соответствующие находящимся в sqlalchemy.orm.mapper [28]. В итоге получается модель, которую можно использовать для любых запросов к базе данных, см. листинг Б.2 в приложении Б.

Далее создадим клиент для работы с базой данных. Для этого реализован класс MysqlClient, в конструкторе которого находятся все необходимые для работы с БД значения: название базы данных, адрес и порт для подключения, логин и пароль для авторизации, а также переменные для сессий и соединений [20]. Метод для подключения к базе данных connect принимает на вход параметр db\_created, который сообщает о состоянии БД: создана она или нет.

Рассмотрим, как происходит соединение с БД. Отправной точкой в SQLAlchemy является Engine - это «домашняя база» для фактической и ее DBAPI, доставленная в приложение SQLAlchemy через пул соединений и Dialect [28], который описывает, как общаться с определенным типом комбинации базы данных и DBAPI. Общую структуру можно представить следующим образом, см. рис 13:

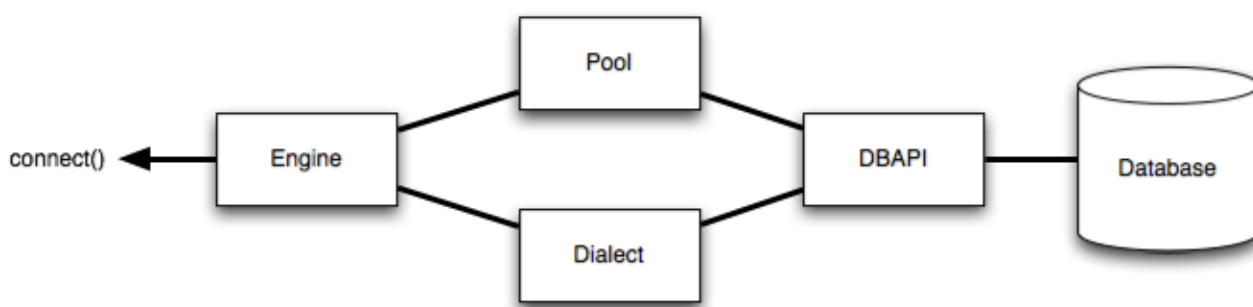


Рисунок 13 – Создание соединения SQLAlchemy

Engine ссылается на Dialect и Pool, которые вместе интерпретируют функции модуля DBAPI, а также поведение базы данных. Для создания движка необходимо выполнить вызов create\_engine(). Функция create\_engine() создает Engine объект на основе URL-адреса. URL-адреса обычно включают имя пользователя, пароль, имя хоста, поля имени базы данных, а также

необязательные аргументы ключевого слова для дополнительной настройки. В некоторых случаях принимается путь к файлу, а в других «имя источника данных» заменяет части «хост» и «база данных». Типичная форма URL-адреса базы данных:

```
dialect+driver://username:password@host:port/database
```

Рисунок 14- Форма URL для create\_engine()

В разрабатываемой функции (см. листинг Б.3 в приложении Б) движок создает Dialect объект, предназначенный для MySQL, а также Pool объект, который указан в конструкторе класса. Устанавливает соединение DBAPI self.host при первом получении запроса на него. Engine и его базовый компонент Pool не устанавливает первое фактическое соединение с DBAPI до тех пор, пока не будет вызван метод Engine.connect() или вызванная операция, зависящая от этого метода, например, Engine.execute(). Таким образом, можно сказать, что поведение инициализации ленивое.

После создания движка и указания всех параметров, с помощью метода Engine.connect() устанавливается соединение с базой данных. Далее с помощью экземпляра Session с переданным внутрь движком можно открывать сеансы для работы с базой данных.

В итоге имеется функция, которая берёт параметры для соединения с базой, устанавливает его и создаёт сессию, см. листинг Б.3 приложения Б.

Для работы с БД нужны методы для инициализации, а также таблицы с пользователями test\_users. Для создания базы и соединения с ней воспользуемся методом connect. Далее в методе для инициализации БД удалим имеющуюся БД, если она есть, создам новую с именем, переданным в класс MysqlClient (см. листинг Б.4 в приложении Б). Создадим функцию инициализации таблицы с пользователями (см. листинг Б.5 в приложении Б), которая проверяет наличие такой таблицы и если её нет, то создаёт её. По

такому же принципу можно добавить создание скольких угодно таблиц при расширении приложения. Также создадим метод `execute_query`, который будет служить обёрткой для метода `execute`, чтобы максимально комфортно передавать запросы к базе и получать ответы от неё.

Далее с помощью написанного клиента можно создавать методы - запросы к базе данных. Их можно разделить на создающие записи и получающие информацию из базы данных.

Для создания записей в базе создадим отдельный класс `MysqlBuilder` (см. листинг Б.6 в приложении Б), в котором с помощью клиента и генератора данных будут реализованы функции, создающие записи в БД и возвращающие данные о созданной записи.

Функции получения данных из БД остаются в классе `MysqlClient`, в них одним из ключевых моментов является метод `session.commit()` экземпляра класса `MysqlClient`, необходимый для сохранения изменений, он же и используется в самом начале функций, чтобы при одновременной работе нескольких пользователей получать самые актуальные данные и вносить изменения с их учётом, пример функции см. листинг Б.7 в приложении Б.

Согласно разработанной документации к тестируемому приложению [19] требуются функции для создания пользователя, получения данных пользователя по значению полей для UI тестов, и запросы на проверку значений полей, заполненных у пользователя в базе для API тестов. Все эти функции реализуются идентично тем, что показаны в листинге Б.6 и Б.7 приложения Б.

### 5.3 Разработка mock

Согласно разработанной документации к тестируемому приложению [19], от микросервиса тестируемое приложение при запросе главной страницы получает информацию о vk id пользователя. Для этого посылается GET-запрос на URL-адрес микросервиса. Для реализации Mock в данном случае потребуется приложение, написанное на Flask. Принцип работы Mock отображён на рис. 15.

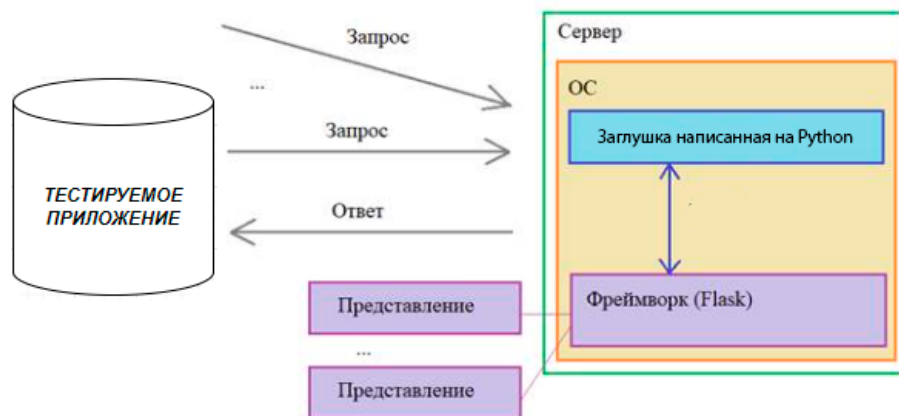


Рисунок 15 – Принцип работы Mock

Импортируем класс-инициализатор Flask. Его экземпляр и будет представлять приложение. Инстанцируем экземпляр, где класс будет принимать первым аргументом имя модуля приложения `__name__`. Это нужно, чтобы Flask знал, где искать шаблоны и статические файлы.

Используем декоратор `route()`, который связывает функции-обработчики с конкретным адресом сайта. Адреса для обработчиков берутся из разработанной документации. Общий принцип работы веб-фреймворка Flask отражает архитектуру HTTP. На каждый адрес задается обработчик - функция, которая выполняет необходимые действия и возвращает ответ [11]. Все входные и выходные данные берутся из запроса согласно документации к тестируемому приложению. Декоратор `@route()` принимает путь или маршрут, для которого вызовется обработчик: каждый путь состоит из сегментов - строк, которые



разделены слэшами ("/"). Функции `/vk_id/add_user` и `/vk_id/<username>` (см. листинг В.1 и В.2 в приложении В) - обработчики, а при каждом применении декоратора `route` происходит добавление нового правила маршрутизации. Во фреймворке принято определять маршрут как комбинацию метода HTTP и адреса, это соответствует идее REST [11]. Метод указывается как аргумент декоратора `route()` — все указанные методы обрабатываются одной функцией, а выбор метода находится в условиях.

Процесс поиска нужного обработчика называется диспетчеризацией. Она выполняется в два этапа: до входа во фреймворк и после входа в приложение. До входа во фреймворк: клиент выполняет запрос к веб-серверу, который расположен на сервере. Клиент — это не обязательно браузер, в данном случае клиент — это реализованный класс `MockClient`, в котором с помощью библиотеки `requests` разрабатываемое решение может посылать запрос к `Mock`. Веб-сервер перенаправляет запрос приложению и устанавливает правильные параметры запроса.

После входа в приложение происходит диспетчеризация, фреймворк анализирует параметры запроса и пытается сопоставить маршруты, которые добавлены в объект `app` с тем, что пришло. Он сравнивает комбинацию адреса и метода. Если метод не указан в правиле, то по умолчанию предполагается `GET`. Этот процесс называется роутингом или маршрутизацией. А место, где внутри хранятся все добавленные маршруты, называют роутером. Если в процессе роутинга нашелся соответствующий маршрут, то вызывается его обработчик. Ответ, который сформировал обработчик, отправляется обратно клиенту.

Логика работы обработчика `/vk_id/add_user` (см. листинг В.1 в приложении В) заключается в том, чтобы добавить в память приложения пользователя с закреплённым за ним `VK id`, если за пользователем он ещё не закреплён.

Логика работы обработчика `/vk_id/<username>` (см. листинг В.1 в приложении В) заключается в просмотре наличия в памяти приложения пользователя с закреплённым за ним `VK ID`.

По такому же принципу в данное Mosk приложение можно добавлять неограниченное количество обработчиков и создавать неограниченное количество подобных mosk-приложений при необходимости.

## 5.4 Разработка API тестов

Для разработки API тестов требуется настройка тестового окружения и создание API-клиента для отправки HTTP-запросов тестируемому приложению. Общая схема того, что необходимо реализовать до разработки таких тестов, отображена на рисунке 16.

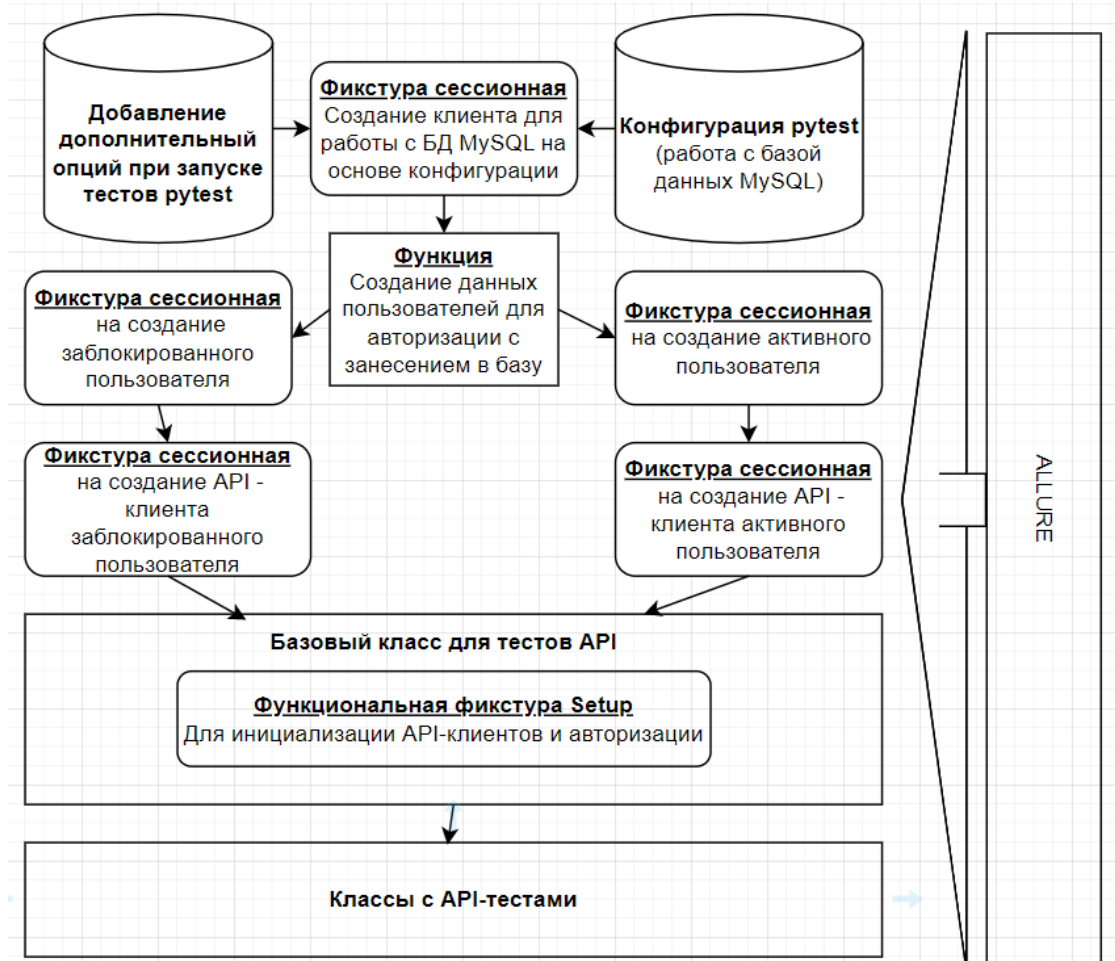


Рисунок 16 – Компоненты необходимые для разработки API тестов

Первое, с чем приходится сталкиваться при настройке — это необходимость в передаче разных значений тестовой функции в зависимости от параметров командной строки. Для этого была использована встроенная в тестовую среду pytest-функция — `pytest_addoption`, принимающая параметр `parser`. Данная функция позволяет добавлять параметры, которые pytest будет считывать из командной строки при запуске тестов. Для API тестов требуется дополнительная опция `url`, которая отвечает за URL-адрес тестируемого

приложения. Именно его будут использовать API-клиенты для отправки запросов.

Рассмотрим конфигурацию `pytest` для запуска тестов. За неё отвечает функция `pytest_configure`, которая принимает объект конфигурации `config` [12, 25]. Данная функция выполняет роль хука, который вызывается для каждого плагина и начального файла `conftest` после анализа параметров командной строки. Он разрешает плагинам и файлам `conftest.py` выполнять первоначальную настройку, то есть всё, что выполняется в данной функции, выполняется до запуска `conftest.py`. В данной функции также создан MySQL клиент с данными `root` пользователя, и реализован алгоритм, позволяющий запускать тесты параллельно, избегая конфликтов с инициализацией БД и таблиц, а также доступа к файлам, реализацию см. в листинге Г.1 приложения Г.

Далее реализуем фикстуру, которая позволила бы создавать MySQL клиент и предоставлять его к использованию, а по окончании автоматически закрывать соединение с БД. Сделать это можно с помощью созданного в конфигурационной функции `pytest`, клиента базы данных и оператора `python – yield`, который позволяет возвратиться из функции с сохранением её локальных переменных, а при повторном вызове выполнять продолжение (см. листинг Г.2 в приложении Г).

Создадим функцию, которая будет генерировать данные пользователя, как заблокированного, так и активного. Для этого используем созданную фикстуру `mysql_client` и `MysqlBuilder` для занесения сгенерированных данных в БД, а также генератор данных для генерации данных пользователя. Функция возвращает логин и пароль пользователя, сгенерированного и занесённого в БД (реализацию см. в листинг Г.3. В приложении Г). С помощью данной функции создаются фикстуры для генерации разных типов пользователей (см. листинг в приложении Г), по такому принципу можно создать сколько угодно пользователей с разными параметрами.

После того, как все необходимые данные для создания API-клиента имеются, можно приступать к его разработке. Для этого была использована библиотека `requests`, позволяющая отправлять HTTP-запросы [26]. Сам API клиент представляет из себя класс с набором методов для запросов к веб-приложению. Согласно документации приложения, запросы используют методы GET, POST, DELETE и PUT. Помимо функций из документации требуется реализовать функцию авторизации.

При инициализации необходимо запросить URL-адрес, логин и пароль пользователя, от имени которого API клиент будет делать HTTP-запросы. Также в конструкторе класса создаётся сессия, в которой будут храниться cookies, в API клиенте все запросы будут выполняться именно от данной сессии. Cookies — это хранящиеся на компьютерах и гаджетах клиентов данные веб-приложения, с помощью которых оно запоминает информацию о пользователях. Создадим функцию авторизации (см. листинг Г.4 в приложении Г), в которой составляются заголовки для запроса, тело запроса с данными пользователя, а также сам POST запрос в созданной сессии. После этого в сессию будут добавлены cookies, которые позволят подтверждать личность пользователя при отправке запросов тестируемому приложению.

Также создадим обёртку для метода `request` (см. листинг Г.5 в приложении Г), которая позволит сократить количество дублирующего кода в приложении. В неё вынесен функционал проверки ожидаемого статус кода ответа тестируемого приложения, а также функция для проверки тела запроса на наличие сообщений об ошибке. После этого создадим методы для запросов к приложению согласно документации: для создания и удаления пользователя, для смены пароля, для блокировки и разблокировки пользователя, а также для проверки статуса работоспособности приложения, пример реализации см. листинг Г.6 в приложении Г.

После создания API клиента реализуем фикстуры, которые будут его создавать в зависимости от пользовательских данных, переданных в них (см. листинг Г.7 в приложении Г). Это необходимо для того, чтобы в API-тестах

можно было выполнять запросы от имени разных пользователей, наделённых разными правами.

Далее приступим к написанию API-тестов. Создадим базовый класс `BaseApi` с фикстурой `setup`, отвечающей за первоначальную настройку класса с тестами. В ней создадим экземпляры API-клиентов заблокированного и активного пользователя в зависимости от флага `authorize`. API-клиенты производят авторизацию (см. листинг Г.8 приложения Г). Это позволяет делить тесты на классы для авторизованных и неавторизованных пользователей, по умолчанию флаг принимает значение `True`.

Сами тесты делятся на тестовые классы в зависимости от тестируемой функциональности и необходимости быть авторизованным. Все API-тесты помечаются маркировкой `API`, чтобы их можно было запускать отдельно по данной маркировке. Все классы обязательно наследуются от базового класса `BaseApi`.

Тесты бывают как параметризованные, так и обычные. Параметризованные тесты дают возможность прогнать большое количество тестовых случаев в виде одного теста с помощью подстановки разных входных значений. Такие тесты помечаются декоратором `@pytest.mark.parametrize`. В данном случае такой тип тестов используется для тестирования функции создания пользователей, так как на каждый параметр, передаваемый в запросе должны накладываться точно такие же ограничения, что и в UI интерфейсе. Также при этом есть много негативных сценариев, при которых выполняется один и тот же тест, но с разными входными параметрами.

В разработке тестов используется библиотека `pytest_check`, которая является заменой стандартным `assert` [23]. Главным её преимуществом является то, что при возникновении ситуации, в которой тестовое условие не выполняется, она не останавливает тест до тех пор, пока не проверит все тестовые условия в отличие от стандартных `assert`, которые заканчивают выполнение при первом найденном невыполненном условии.

API-тесты в основном состоят из 3 частей: в первой части выполняется запрос к API приложения с помощью API-клиента определенного пользователя, ответ, полученный от приложения, сравнивается с ожидаемым. Во второй части делается запрос в БД с помощью MySQL-клиента, чтобы проверить верность внесённых с помощью запроса изменений. В третьей части происходит очистка БД от тестовых данных, созданных тестом, это необходимо для того, чтобы избежать конфликтов между тестами. Часть реализации тестов см. в листинге Г.9 в приложении Г.

Итогом данной главы является:

- создание конфигурации тестовой среды;
- разработка вспомогательных функций и фикстур;
- создание API-клиента для HTTP-запросов;
- разработка тестовых классов API с разными типами тестов.

## 5.5 Разработка UI тестов

Разработка UI-тестов, как и разработка API-тестов, начинается с создания вспомогательных функций и фикстур, а также конфигураций для запуска тестов. Общая схема реализации отображена на рис. 17.

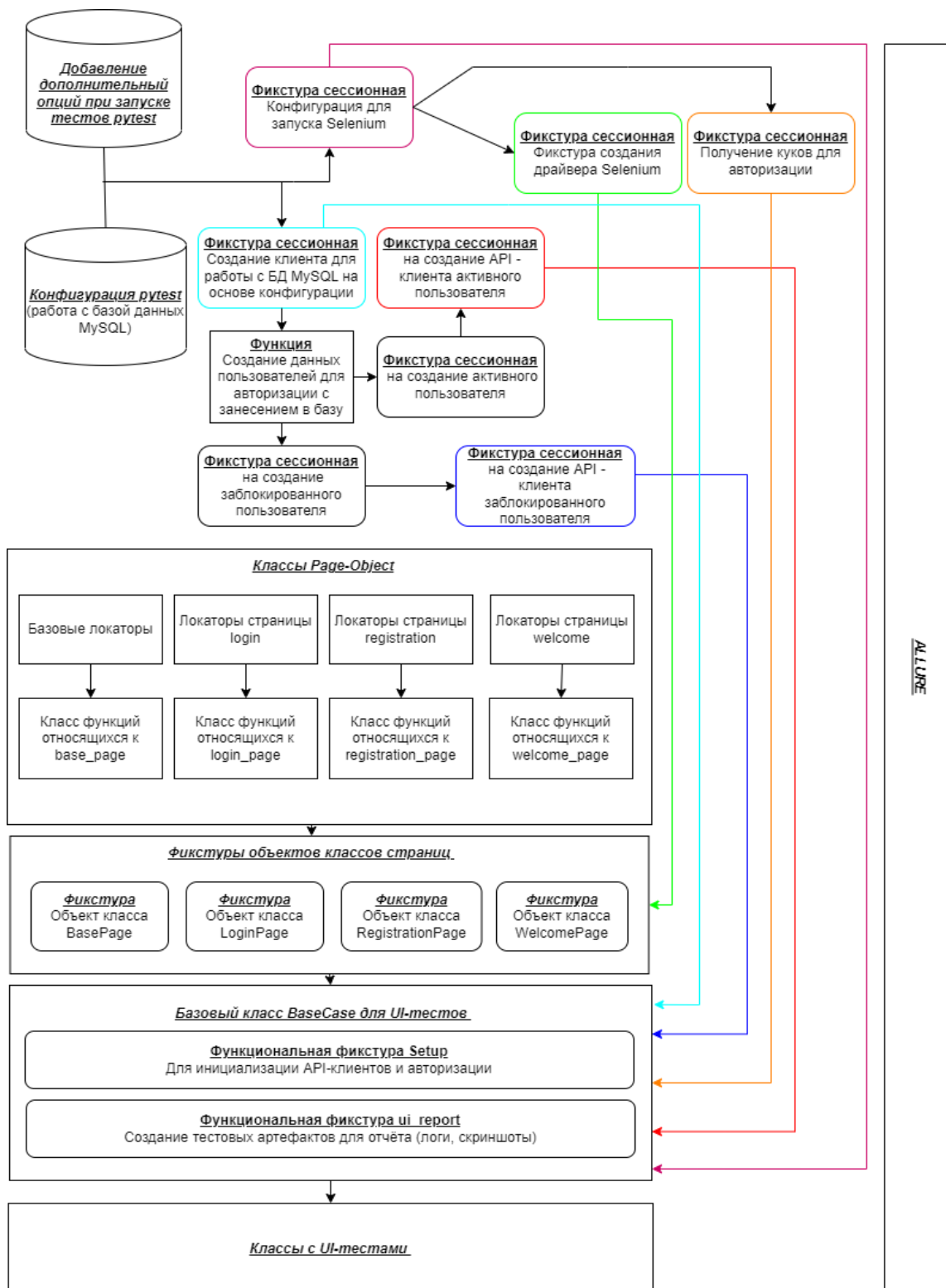


Рисунок 17 – Диаграмма взаимосвязи компонентов UI - тестов



Для начала необходимо добавить в функцию `pytest_addoption` считывание параметров, таких как `--browser` (отвечает за название браузера), `--debug` (сохранение подробной информации об ошибках в лог файл), `--selenoid` (для настройки selenoid), `--vnc` (включение возможности наблюдения за тестами внутри selenoid).

Конфигурационная функция `pytest_configure` остаётся без изменений, так как настройки, сделанные для API тестов, удовлетворяют потребностям UI.

Все фикстуры, отвечающие за инициализацию API-клиентов, генерацию данных и создание пользователей, остаются без изменений. Они используются в базовом классе UI-тестов, в точно таком же виде, что и в базовом классе API.

Также необходима фикстура, которая позволит сохранять в лог-файл данные, выводимые тестами в командную строку, и отображать их по оставленным в программном коде меткам в соответствии с хронологией. Для реализации такой фикстуры используем библиотеку `logging`, предоставляющую функции и классы, реализующие гибкую систему регистрации событий, разделение сообщений по степени критичности, а также фиксирование времени события [17]. Ключевым преимуществом данной библиотеки является то, что все модули Python могут участвовать в ведении журнала, поэтому в него могут включаться собственные сообщения и интегрированные с сообщениями из сторонних модулей.

В фикстуре зададим формат лога, который используется при сохранении записей приложения. Для создания уникальных путей каждого файла используется отдельная фикстура `temp_dir` (см. листинг Д.1 в приложении Д), которая в зависимости от названия класса и теста возвращает уникальный путь до файла с логами каждого теста. При этом базовый каталог остаётся неизменным, так как он из функции конфигурации `pytest_configure`.

Все параметры форматирования устанавливаются в объекте класса `logging` и возвращаются из фикстуры с помощью оператора `yield`, который после возвращения позволит закрыть и записать логи в файл `.log`. Реализацию см. листинг Д.2 в приложении Д.

Для запуска selenium создадим конфигурацию, которая возвращает все нужные параметры: название браузера, URL-адрес тестируемого приложения, URL-адрес с портом для хостинга selenium, а также информация о необходимости наблюдать за ходом выполнения тестов в selenium с помощью VNC [27]. VNC — это широко распространенный метод удаленного доступа к рабочему столу компьютера по сети. В нашем случае доступ будет к контейнеру selenoid. Всё это реализуется в сессионной фикстуре config, а данные берутся из командной строки с помощью pytest\_addoption, на основе которых составляется словарь с параметрами (см. листинг Д.3 в приложении Д).

После настройки конфигурации создадим фикстуру, инициализирующую Selenium Webdriver с заданными настройками. Selenium Webdriver - это драйвер браузера, то есть не имеющая пользовательского интерфейса программная библиотека, позволяющая другим программам взаимодействовать с браузером, управлять его поведением, получать от браузера какие-либо данные и заставлять браузер выполнять команды [27]. Схематично процесс взаимодействия с браузером отображён на рис. 18.



Рисунок 18 – Взаимодействие Selenium Webdriver с браузером.

Для создания такой фикстуры получим из config всю необходимую информацию для конфигурации драйвера. Зададим путь до каталога, сформированного с помощью temp\_dir. В данный каталог поступают все файлы, сохраняемые в процессе тестирования Selenium Webdriver. При наличии флага selenoid осуществляется настройка его использования в контейнере: указывается название и версия контейнера браузера, возможность наблюдения

за процессом тестирования с помощью VNC (selenium-ui) и адрес контейнера хаба, в котором запущен браузер с заданными настройками. При отсутствии этого флага происходит выбор драйвера в соответствии с указанным названием в параметре browser, и затем открывается домашняя страница, указанная в параметре url фикстуры config. Готовый к использованию экземпляр драйвера возвращается из фикстуры с помощью оператора yield, что даёт возможность автоматически завершать использование драйвера. Реализацию см. листинг Д.4 в приложении Д.

Стоит поговорить об авторизации в приложении без помощи UI-интерфейса. Это необходимо для тестов, проверяющих функции, доступные авторизованным пользователям. Для этого воспользуемся API клиентом и генерируемыми пользователями фикстурами. С помощью них проведем авторизацию, полученные cookies возвратим в виде словаря со значениями. Реализацию см. в листинге Д.5 в приложении Д.

Далее рассмотрим паттерн PageObject, который является де-факто стандартом в автоматизации тестирования веб-продуктов [8]. Основная идея состоит в том, чтобы разделить логику тестов от реализации. Каждая веб-страница проекта описывается в виде объекта класса, взаимодействие пользователя - в методах класса, а в тестах остается только бизнес-логика. Данный подход помогает избежать проблем при изменении верстки веб-приложения, так как нужно будет править только один класс, описывающий страницу. Page-Object содержит следующие части:

- Locator Object \ Locator Class.
- Page Object \ Page Class.
- Base Page \ Base Class
- Tests.

Locator Object \ Locator Class хранит в себе локаторы. Локатор - это строка, уникально идентифицирующая UI-элемент. Когда человек делает клик мышкой, вводит текст и выполняет прочие действия, то они выполняются над

конкретным объектом. Selenium поступает также, однако ему надо указать объект, для которого нужно применить действие. Для создания локатора используется XPATH. Xpath - это язык запросов к элементам xml или xhtml документа. Также, как и SQL, xpath является декларативным языком запросов. Поэтому за счёт его универсальности для составления локаторов был выбран именно он. Пример класса с локаторами см. в листинге Д.6 в приложении Д.

Page Object \ Page Class реализует методы для работы с элементами на веб-страницах. Методы разделены на классы по функциональной принадлежности к той или иной веб-странице. В Page Class также передаются локаторы. Пример реализации класса см. в листинге Д.7 в приложении Д.

Base Page \ Base Class реализует в себе необходимые методы для работы с webdriver, они доступны для всех классов. В данном случае это функциональные фикстуры setup и ui\_report.

В фикстуре setup инициализируются webdriver, конфиг, логгер, клиент и билдер MySQL, все классы из Page Object. Затем проходит авторизация с помощью подстановки cookies, полученных из фикстуры в случае установки флага authorize, что предоставляет всем тестовым классам полный функционал разрабатываемых ранее функций, фикстур и классов (реализацию см. в листинге Д.8 в приложении Д).

Фикстура ui\_report используется для получения тестовых артефактов из браузера. Всё, что выводит браузер она пишет консоль, а также делает скриншот окна браузера в случае нахождения тестом неисправности. Все данные записываются по пути, полученному с помощью фикстуры temp\_dir. Скриншот делается с помощью метода get\_screenshot\_as\_file, который предоставляет webdriver. Реализацию см. в листинге Д.9 в приложении Д.

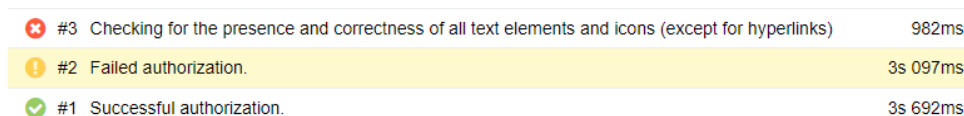
Рассмотрим разработку классов с UI-тестами. Все они наследуются от базового, с помощью чего получают основную функциональность для написания тестов. Это позволяет получать их описанными бизнес-логикой тест-кейса. Тестовые классы разделяются по признаку принадлежности к веб-странице и по необходимости в авторизации. Их вид точно такой же, что и в

API-тестах. Все UI-тесты помечаются специальным декоратором `@pytest.mark.UI`, который позволяет в случае необходимости запускать только их. Реализацию см. в листинге Д.10 в приложении Д.

## 5.6 Отчёты Allure

Чтобы Allure мог собирать результаты во время выполнения теста, добавим параметр `–alluredir` с указанным путем к папке, в которой должны храниться результаты. Для фактического отчёта после завершения тестов используем утилиту командной строки Allure для создания отчета по результатам.

Все статусы pytest по умолчанию имеются в отчете Allure: тесты, которые не были выполнены из-за ошибок утверждения, будут помечены как не пройденные, любое другое исключение приведет к тому, что тест будет иметь сломанный статус.



✖ #3	Checking for the presence and correctness of all text elements and icons (except for hyperlinks)	982ms
⚠ #2	Failed authorization.	3s 097ms
✔ #1	Successful authorization.	3s 692ms

Рисунок 19- статусы тестов в Allure

Фикстуры и финализаторы — это служебные функции, которые Pytest будет вызывать до начала и после завершения теста соответственно [25]. Allure отслеживает вызовы каждой фикстуры и подробно показывает, какие методы и с какими аргументами были вызваны, сохраняя правильную последовательность.



Set up	
✔ _session_faker	95ms
> config 1 sub-step	1ms
✔ temp_dir	1ms
✔ mysql_client	0s
✔ driver	6s 355ms
✔ logger	1ms
✔ setup	145ms
✔ login_page	0s
✔ base_page	0s
✔ welcome_page	0s
✔ registration_page	0s
✔ ui_report	0s

Рисунок 20-Отображение фикстуры и конфигураторов

Первый и, вероятно, самый важный аспект отчета Allure заключается в том, что он позволяет получить пошаговое представление каждого запуска теста. Это возможно благодаря `@allure.step` декоратору, который добавляет в

отчет вызов аннотированного метода или функции с предоставленными аргументами. Именно он используется в большинстве разработанных функций и фикстур (см. листинги Д.5 и Д.7 в приложении Д).

Аннотированные методы `@step` хранятся отдельно и импортируются при необходимости (см. листинг Е.1 в приложении Е). Пошаговые методы могут иметь сколь угодно глубоко вложенную структуру. Статус каждого шага отображается маленькой иконкой справа от названия. Вложенные шаги организованы в виде древовидной складной структуры. Они могут иметь строку описания, которая поддерживает заполнители для переданных позиционных и ключевых аргументов. Также захватываются параметры по умолчанию аргументов ключевого слова, см.рис 21.

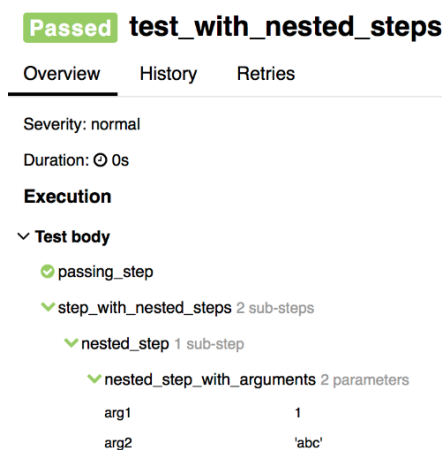


Рисунок 21- Хранимые шаги, вложенная структура.

Отчеты отображают множество различных типов предоставленных вложений, которые дополняют результаты теста и шага. Вложения созданы с помощью `allure.attach(body, name, attachment_type, extension)`, где:

- **body** - необработанный контент для записи в файл;
- **Name** - строка с именем файла;
- **attachment\_type** - одно из `allure.attachment_type` значений;
- **extension** - используется в качестве расширения для созданного файла;

А также с помощью `allure.attach.file(source, name, attachment_type, extension)`. Реализацию см. листинг Д.9 в приложении Д. Вложения отображаются в контексте тестового объекта, которому они принадлежат см. рис. 22.

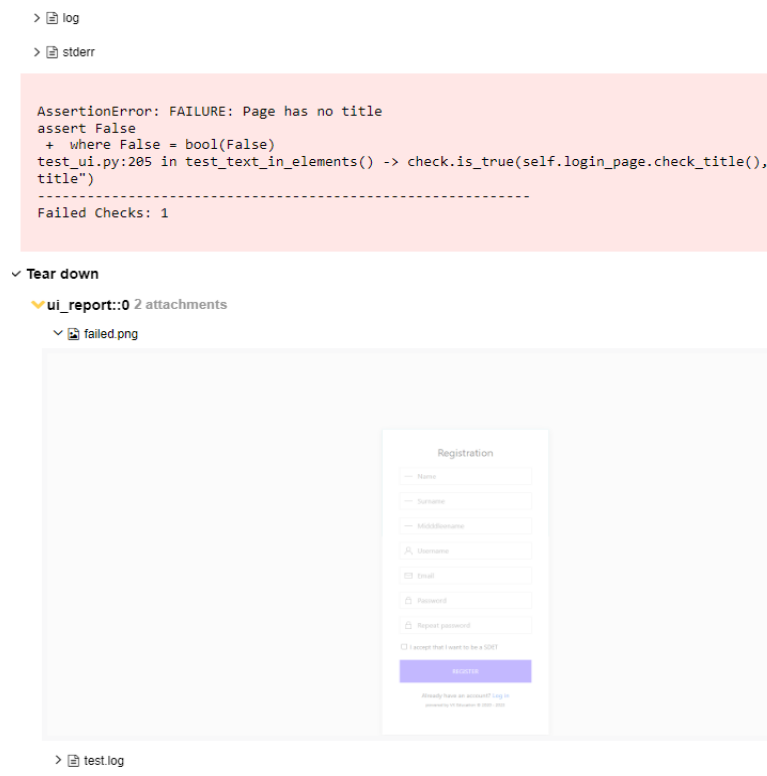


Рисунок 22 – Вложения Allure

Заголовки тестов сделаны более читабельными с помощью специального декоратора `@allure.title`. Они поддерживают аргументы и динамическую замену (см. листинг Д.10 в приложении Д и рис. 19).

Для интеграции отчета с системой отслеживания ошибок или системой управления тестированием у Allure есть дескрипторы `@allure.link`, `@allure.issue`. `@allure.link` предоставляет кликабельную ссылку на указанный URL-адрес в разделе «Ссылки» (см. рис 24).

## Links

[Click me](#)

Рисунок 23 – Отображение ссылки



@allure.issue предоставляет ссылку со значком ошибки. Этот дескриптор принимает идентификатор тестового примера в качестве входного параметра, чтобы использовать его с предоставленным шаблоном ссылки для типа ссылки задачи (см. рис 25).

### **Links**

✖ Pytest-flaky test retries shows like test steps

Рисунок 24 – Отображение issue

## 5.7 Docker

После готовности тестового проекта следует задуматься об его упаковке и дальнейшем использовании в CI. Для этого потребуется собрать образ docker-контейнера для некоторых из разработанных компонентов. В каждом образе содержится образ базовой операционной системы, код приложения и библиотеки, от которого он зависит. Всё это скомпоновано в виде единой сущности, на основе которой можно создать контейнер.

Файл под названием Dockerfile содержит набор инструкций, следуя которым Docker собирает образ контейнера [9]. Данный файл нужен для упаковки тестового проекта, Mock и БД. У остальных компонентов имеется готовый образ, который можно настроить без создания Dockerfile уже в Docker Compose.

Самым крупным хранилищем образов является репозиторий Docker Hub, он используется при работе с Docker по умолчанию [9]. В образ контейнера, поверх базового образа, можно добавлять дополнительные слои, это делается в соответствии с инструкциями из Dockerfile. Помимо организации запаковки проекта также потребуется и его развёртывание. Для этого используется инструментальное средство Docker Compose, который используется для одновременного управления несколькими контейнерами, входящими в состав приложения [9]. Этот инструмент предлагает те же возможности, что и Docker, но позволяет работать с более сложными приложениями.

Начнем рассмотрение с упаковки Mock. Создадим Dockerfile в папке рядом с его файлами (см. рис 5). В качестве родительского образа выступает Python 3.8. Создадим список зависимостей в файле requirements.txt и добавим его в корень контейнера, с помощью pip3 они установятся внутри него. Далее перенесем папку с файлами Mock в проект, укажем её рабочим каталогом. Добавим в Dockerfile команду, с помощью которой будет стартовать mock-server, и укажем используемый для него порт. Реализацию см. листинг Ж.1 в приложении Ж.

Теперь нужно создать Dockerfile для БД MySQL. В нём, поверх родительского образа, скопируем в контейнер скрипт для инициализации БД. Все остальные действия для работы с БД описываются в Docker-compose (см. листинг Ж.2 в приложении Ж).

Затем необходимо упаковать сам тестовый проект. В Dockerfile используем родительский образ python 3.9, а также добавим зависимости, хранящиеся в requirements.txt, и установим их внутри контейнера. После этого рабочей папкой внутри контейнера назначим src. Все остальные действия описываются в Docker-compose. Реализацию см. листинг Ж.3 в приложении Ж.

Перейдем к рассмотрению файла Docker Compose. Схема зависимости контейнеров отображена на рисунке 25:

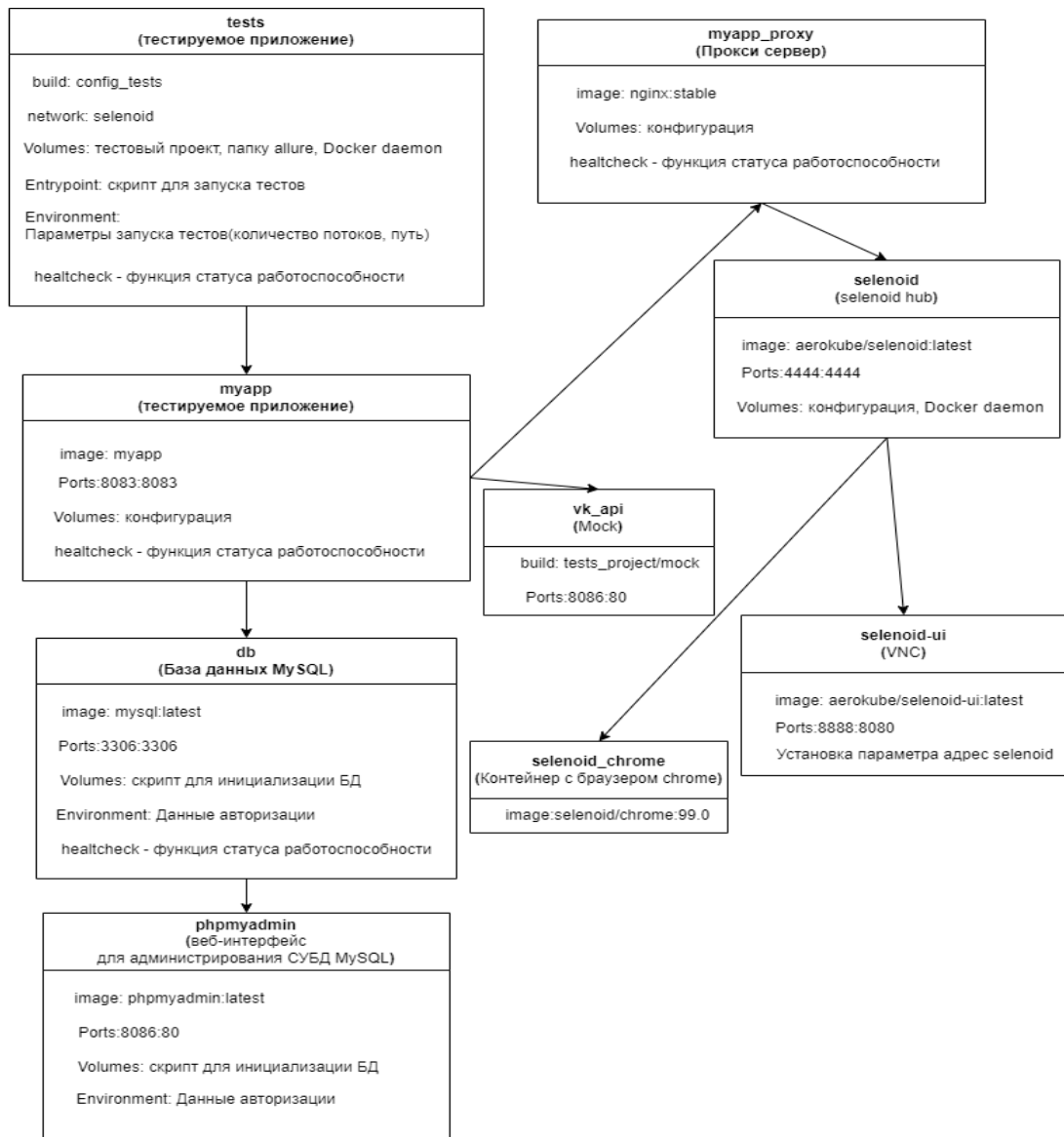


Рисунок 25- зависимости Docker- контейнеров друг от друга

Для начала создадим сеть, в которой будут находиться все контейнеры: в networks укажем название selenium, а параметр external установим равным False, чтобы Docker Compose понимал, что сеть была создана снаружи.

Сервисный контейнер tests содержит тестовый проект, myapp – тестируемое приложение, myapp\_proxy – прокси сервер для взаимодействия с тестируемым приложением внутри сети Docker, db – база данных MySQL, phpadmin – веб-интерфейс для администрирования СУБД MySQL, vk\_api – mock-server, selenium\_chrome – контейнер с браузером chrome для selenium, selenium – контейнер с selenium, selenium-ui – VNC для selenium. Для всех этих контейнеров необходимо описать образы, из которых они будут запускаться.

Это делается с помощью указания команды `image` для всех контейнеров за исключением проекта с тестами и `mock-server`, так как в них развёртывание происходит по инструкциям из `Dockerfile` с помощью команды `build`, в которой указан путь до конфигурации сборки образа контейнера из исходного кода `Dockerfile`. При развёртке контейнеров `tests`, `myapp`, `myapp_proxy`, `db` и `selenium` в параметре `volumes` укажем пути монтирования хоста или наименование томов, которые должны быть доступны сервисным контейнерам. Если монтирование является путем к хосту и используется только одной службой, оно может быть объявлено как часть определения службы вместо ключа верхнего уровня, как, например, сокет Unix в сервисных контейнерах `tests` и `selenium`, который `Docker Daemon` прослушивает по умолчанию. Это используется для связи с `Docker Daemon` из контейнера.

Команда `tty` настраивает сервисный контейнер для работы с TTY, при её включении консоль открывается сразу после запуска контейнера. Команда `working_dir` переопределяет рабочий каталог контейнера в соответствии с указанным в образе. Команда `ports` задаёт контейнерные порты, которые необходимо открыть.

С помощью `command` в контейнерах `myapp`, `selenium` и `selenium-ui` переопределяется команда по умолчанию, объявленная образом контейнера, либо же задаются начальные параметры, как в случае с контейнером `selenium`.

С помощью команды `entrypoint` в контейнере `tests` переопределяется точка входа по умолчанию для образа `Docker`, то есть удалятся обе команды `ENTRYPOINT` и `CMD` в инструкции `Dockerfile`, если они настроены файлом `Compose`. В данной команде указывается скрипт, написанный на языке `bash`, который копируется по пути `/bin/bash` в контейнер. Оттуда он автоматически запускается при запуске контейнера. Сам скрипт представляет из себя набор команд для указания прав тестовому проекту и запуска тестов (см. листинг Ж.4 в приложении Ж).

Команда `environment` определяет переменные среды, установленные в контейнерах, в данном случае это либо параметры для запуска тестов, либо данные для подключения к БД и авторизации.

Команда `depends_on` указывает зависимости запуска и завершения работы между контейнерами, а с помощью команды `healthcheck` объявляется проверка, которая выполняется для определения работоспособности контейнера.

В итоге получаем Docker Compose файл в расширении `.yaml` (см. листинг Ж.5 в приложении Ж), который успешно запускает всё тестовое окружение и прогоняет тесты. Следующим шагом является автоматический запуск данной тестовой инфраструктуры с помощью средства CI.

## 5.8 Jenkins CI

Цель методологии "Continuous Integration" – построение максимально быстрого, надежного и повторяемого процесса внедрения изменений в программный продукт [16]. В качестве рабочих микропроцессов рассматриваются коммит, тестирование, сборка, регистрация пакета и развертывание (см. рис 26).

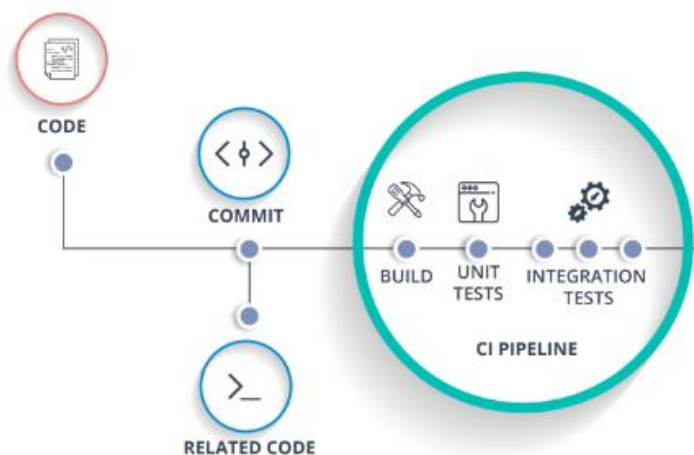


Рисунок 26- Процессы CI

Изменение программного кода, зарегистрированного системой контроля версий и отправленное для проверки в интеграционное тестирование, называется коммитом. После процесса сборки, описанного в предыдущей главе, продукт проходит процедуру регистрации в виде отдельного релиза. Процесс непрерывной интеграции завершается развертыванием на эксплуатируемой платформе (обычно это тестовый сервер).

Для управления этим процессом была установлена и сконфигурирована CI-консоль Jenkins, являющаяся бесплатным и открыто распространяемым программным обеспечением. Обычно в продуктовой команде каждый разработчик может получить доступ к Jenkins через веб-браузер для отслеживания результатов последней сборки, а также истории внедрения изменений.

В данном случае рассмотрим только микропроцесс CI по организации процесса тестирования в Jenkins. В процессе установки были интегрированы

все плагины, которые предлагались по умолчанию, так как данный набор максимально закрывал потребности для организации CI. Дополнительно был установлен плагин Allure Jenkins Plugin, позволяющий просматривать полученные отчеты о прохождении тестов непосредственно в Jenkins.

Следуя подходу «инфраструктура как код», по которой процесс её настройки аналогичен процессу программирования ПО, мы используем Jenkins Pipeline, предоставляющий набор подключаемых модулей, поддерживающих реализацию и интеграцию конвейеров непрерывной доставки в Jenkins. В основе Pipeline лежит Jenkinsfile – это простой текстовый файл с кодом на языке Groovy, который используется для его конфигурации [13]. Данный файл располагают в корень проекта в репозитории, здесь это будет Github. Если отобразить схематично, то Pipeline в данном проекте выглядит следующим образом (см. рис. 27):

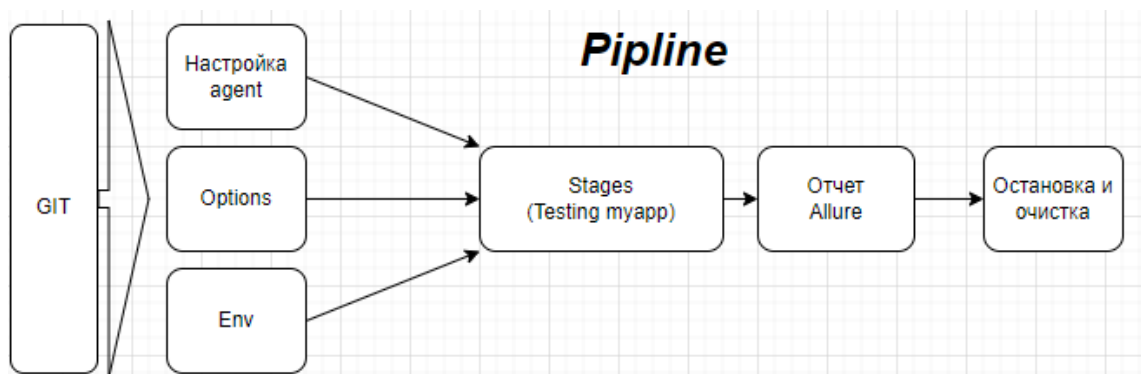


Рисунок 27 – Pipeline

В настройках pipeline указывается система контроля версий, которая используется на хосте и репозиторий с веткой, из которого будут браться файлы проекта, а также путь до скрипта groovy.



Рисунок 28 – Настройка SCM

Рассмотрим сам скрипт `simple.groovy` (см. листинг 3.1 в приложении 3). Он реализует базовый трехэтапный конвейер непрерывной доставки, однако два этапа из них являются заглушками, так как в данной работе рассматривается только тестирование. Не все Pipeline имеют эти три стадии, но это хорошая отправная точка для определения большинства проектов.

В директиве `agent` указывается исполнитель и рабочая область для конвейера. Без этой директивы декларативный конвейер не только недействителен, но и не способен выполнять какую-либо работу. Она по умолчанию гарантирует, что исходный репозиторий извлечен и доступен для шагов на последующих этапах, а также указывает, где будет выполняться весь конвейер или определенный этап в среде Jenkins в зависимости от того, где находится `agent` раздел.

Затем указываются переменные окружения с помощью директивы `environment`. Она используется в блоке верхнего уровня `pipeline`, то есть применяется ко всем шагам в конвейере. Из таких переменных для тестирования необходим только путь до `docker-compose`.

Директивы `stage` и `steps` необходимы для декларативного конвейера, так как они указывают Jenkins, что и на каком этапе должно быть выполнено. В данной работе выполняется переход в папку с тестовым проектом и открывается `docker-compose`, который запускает всё тестовое окружение и проводит тестирование.

В декларативе `post` указываются события, которые должны произойти после завершения действий, описанных в `stages`, а также сами действия, которые выполняются при ошибках и сбоях. В данном случае такими действиями являются составление отчёта `allure` и остановка `docker-compose`. Они указываются также под декларативой `always`, то есть происходят в любом случае после завершения `pipeline`.

В конце с помощью функции `cleanWs()` удаляется рабочее пространство, выделенное под сборку `pipeline`.

Итогом данной главы является настроенный процесс тестирования в CI Jenkins с помощью Pipeline. Тесты теперь могут запускаться как при совершении нового коммита, так и автоматически по таймеру.

## Заключение

В рамках данной работы была достигнута поставленная цель, а именно: разработка на основе проведённого анализа комплексного решения для автоматизации тестирования веб-приложений.

В теоретической части дипломной работы была показана актуальность и рациональность внедрения системы автоматизированного тестирования веб-приложений, произведена оценка пользы от внедрения данной системы. Также произведен анализ и подбор технологий для разработки данной системы.

В практической части произведена разработка архитектуры системы автоматизированного тестирования, подобрано веб-приложение для демонстрации разрабатываемой системы.

Разработаны вспомогательные модули:

- генерации данных,
- работы с БД MySQL,
- Mock-server.

Разработаны основные модули, содержащие автотесты, фикстуры, конфигурации:

- API-тесты,
- UI-тесты.

Осуществлена интеграция Allure Framework для автоматической генерации отчётной документации.

Произведена упаковка разрабатываемой системы с помощью платформы контейнеризации Docker.

Произведена интеграция разработанной системы в процесс CI с помощью программной системы Jenkins.

## Список литературы

1. Как тестируют в Google: [книга]/ / Дж. Уиттакер, Дж. Арбон, Дж. Каролло - СПб.: Питер, 2014. - 320с.
2. Тестирование программного обеспечения. Базовый курс. : [Книга]/ / Святослав Куликов - 3-е издание. - : EPAM Systems, 2022 - 301 с.
3. Allure Documentation. — Текст: электронный // Docs Allure: [сайт]. — URL: [https://docs.qameta.io/allure/#\\_pytest](https://docs.qameta.io/allure/#_pytest) (дата обращения: 1.08.2022).
4. ALM Octane Documentation. — Текст: электронный // Docs ALM Octane: [сайт]. — URL: <https://zebrunner.com/documentation integrations/octane/> (дата обращения: 1.08.2022).
5. An approach to creating concretized test scenarios within test automation technology for industrial software projects: [Статья]/ Kolchin, A, Letichevsky, A, Peschanenko, V, Drobintsev, P, Kotlyarov, V // Automatic Control and Computer Sciences, 2013. — URL: <https://doi.org/10.3103/S0146411613070213> (дата обращения: 1.08.2022).
6. API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing [книга]/ Dave Westerveld- 1-е издание. - Packt Publishing, 2021 - 340 с.
7. A Survey of the Software Test Methods and Identification of Critical Success Factors for Automation: [Статья]/ Bindu Bhargavi, S. M., Suma, V // SN Computer Science, 2022. — URL: <https://doi.org/10.1007/s42979-022-01297-5> (дата обращения: 1.08.2022).
8. Design Patterns for High-Quality Automated Tests: High-Quality Test Attributes and Best Practices: [Книга]/ / Anton Angelov - 1-е издание. - : On Kindle Scribe, 2020 - 316 с.
9. Docker Documentation. — Текст: электронный // Docs Docker: [сайт]. — URL: <https://docs.docker.com/> (дата обращения: 1.08.2022).

10. Faker documentation. — Текст: электронный // Docs Faker: [сайт]. — URL: <https://faker.readthedocs.io/en/> (дата обращения: 1.08.2022).
11. Flask documentation. — Текст: электронный // Docs Flask: [сайт]. — URL: <https://flask.palletsprojects.com/en/> (дата обращения: 1.08.2022).
12. Full pytest documentation. — Текст: электронный // Docs Pytest: [сайт]. — URL: <https://docs.pytest.org/en/> (дата обращения: 1.08.2022).
13. Gitlab documentation. — Текст: электронный // Docs Gitlab: [сайт]. — URL: <https://docs.gitlab.com/> (дата обращения: 1.08.2022).
14. GoCD User Documentation. — Текст: электронный // Docs GoCD: [сайт]. — URL: <https://docs.gocd.org/current/> (дата обращения: 1.08.2022).
15. Jenkins User Documentation. — Текст: электронный // Docs Jenkins: [сайт]. — URL: <https://www.jenkins.io/doc/> (дата обращения: 1.08.2022).
16. Learning DevOps: The complete guide to accelerate collaboration with Jenkins, Kubernetes, Terraform and Azure DevOps: [Книга] / / Mikael Krief - 2-е издание. - : Packt Publishing, 2022 - 560 с.
17. logging — Logging facility for Python— Текст: электронный // Docs Logging: [сайт]. URL: <https://docs.python.org/3/library/logging.html> (дата обращения: 1.08.2022).
18. Mimesis: Fake Data Generator. — Текст: электронный // Docs Mimesis: [сайт]. — URL: <https://mimesis.name/en/> (дата обращения: 1.08.2022).
19. My APP Documentation. — Текст: электронный // Docs MyApp: [сайт]. URL: [https://github.com/EVSobol/Autotest\\_python/blob/main/Docs%20MyApp.pdf](https://github.com/EVSobol/Autotest_python/blob/main/Docs%20MyApp.pdf) (дата обращения: 1.08.2022).
20. MySQL documentation. — Текст: электронный // Docs MySQL: [сайт]. — URL: <https://dev.mysql.com/doc/> (дата обращения: 1.08.2022).

21. Peewee documentation. — Текст: электронный // Docs Peewee: [сайт]. — URL: <https://docs.peewee-orm.com/en/latest/> (дата обращения: 1.08.2022).
22. PostgreSQL. — Текст: электронный // Docs PostgreSQL: [сайт]. URL: <https://www.postgresql.org/docs/> (дата обращения: 1.08.2022).
23. Pytest-check. — Текст: электронный // Docs pytest-check: [сайт]. URL: <https://pypi.org/project/pytest-check/> (дата обращения: 1.08.2022).
24. Python. — Текст: электронный // Docs python: [сайт]. URL: <https://docs.python.org/3/> (дата обращения: 1.08.2022).
25. Python Testing with pytest: [книга]// Брайан Оккен - 2-е издание, 2022 - 216 с.
26. Requests: HTTP for Humans. — Текст: электронный // Docs Requests: [сайт]. URL: <https://requests.readthedocs.io/en/latest/> (дата обращения: 1.08.2022).
27. Selenium with Python. — Текст: электронный // Docs Selenium: [сайт]. — URL: <https://selenium-python.readthedocs.io/> (дата обращения: 1.08.2022).
28. SQLAlchemy documentation. — Текст: электронный // Docs SQLAlchemy: [сайт]. — URL: <https://www.sqlalchemy.org/> (дата обращения: 1.08.2022).
29. SQLAlchemy ORM Tutorial for Python Developers . — Текст: электронный // Auth0 blog: [сайт]. — URL: <https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/> (дата обращения: 1.08.2022).
30. Test Automation: [Статья] / De Grood, Derk-Jan // TestGoal: Result-Driven Testing, 2008. — URL: [https://doi.org/10.1007/978-3-540-78829-4\\_16](https://doi.org/10.1007/978-3-540-78829-4_16) (дата обращения: 1.08.2022).
31. UI Testing with Puppeteer [книга]/ Dario Kondratiuk - 1-е издание. - Packt Publishing, 2021 - 316 с.

32. Unit testing framework. — Текст: электронный // Docs.python: [сайт]. — URL: <https://docs.python.org/3/library/unittest.html> (дата обращения: 1.08.2022).

33. Using the Autotest Mock Library for testing. — Текст: электронный // Autotest readthedocs:[сайт]. — URL: <https://autotest.readthedocs.io/> (дата обращения: 1.08.2022).

34. Zephyr Project Documentation. — Текст: электронный // Docs Zephyr: [сайт]. — URL: <https://flask.palletsprojects.com/en/> (дата обращения: 1.08.2022).

## Приложение А

### Welcome to the TEST SERVER

LOGIN

Not registered? [Create an account](#)

powered by VK Education © 2020 - 2022

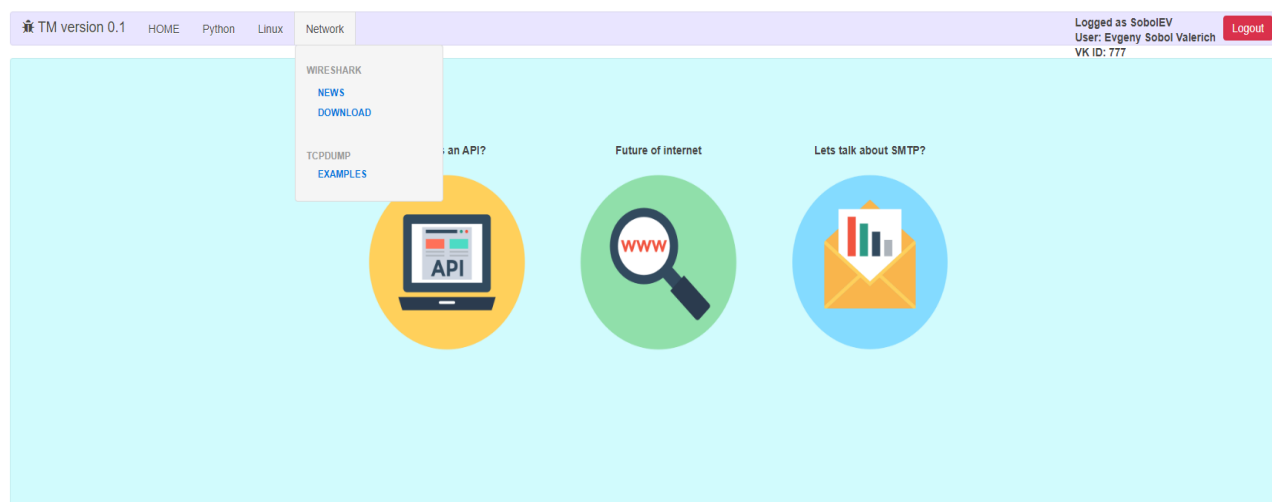
### Registration

☐ I accept that I want to be a SDET

REGISTER

Already have an account? [Log in](#)

powered by VK Education © 2020 - 2022



If the implementation is hard to explain, it's a bad idea

Powered by VK Education © 2020 - 2021



## Приложение Б

### Листинг Б.1 – SQL скрипт для инициализации базы данных

```
CREATE TABLE `test_users` (  
  `id` int NOT NULL AUTO_INCREMENT,  
  `name` varchar(255) NOT NULL,  
  `surname` varchar(255) NOT NULL,  
  `middle_name` varchar(255) DEFAULT NULL,  
  `username` varchar(16) DEFAULT NULL,  
  `password` varchar(255) NOT NULL,  
  `email` varchar(64) NOT NULL,  
  `access` smallint DEFAULT NULL,  
  `active` smallint DEFAULT NULL,  
  `start_active_time` datetime DEFAULT NULL,  
  PRIMARY KEY (`id`),  
  UNIQUE KEY `email` (`email`),  
  UNIQUE KEY `ix_test_users_username` (`username`)  
);
```

### Листинг Б.2 – Модель SQLAlchemy

```
from sqlalchemy.ext.declarative import declarative_base  
from sqlalchemy import Column, Integer, SmallInteger, Date, VARCHAR  
Base = declarative_base()  
  
class TestUsersModel(Base):  
    __tablename__ = 'test_users'  
    __table_args__ = {'mysql_charset': 'utf8'}  
  
    def __repr__(self):  
        return f"<TestUsers: id={self.id}, name={self.name},  
surname={self.surname}, " \br/>            f" middle_name={self.middle_name}, username={self.username},  
password={self.password}, " \br/>            f"email={self.email}, access={self.access},  
active={self.active}, " \br/>            f" start_active_time = {self.start_active_time}>"  
  
    id = Column(Integer, primary_key=True, autoincrement=True)  
    name = Column(VARCHAR(255), nullable=False)  
    surname = Column(VARCHAR(255), nullable=False)  
    middle_name = Column(VARCHAR(255))  
    username = Column(VARCHAR(16), nullable=False, unique=True)  
    password = Column(VARCHAR(255), nullable=False)  
    email = Column(VARCHAR(64), nullable=False, unique=True)  
    access = Column(SmallInteger, nullable=False)  
    active = Column(SmallInteger, nullable=False)  
    start_active_time = Column(Date)
```

### Листинг Б.3 – Функция connect в MySQL клиенте

```
def connect(self, db_created=True):  
    db = self.db_name if db_created else ''  
    url =  
f'mysql+pymysql://{self.user}:{self.password}@{self.host}:{self.port}/{db}'  
    print(url)  
    self.engine = sqlalchemy.create_engine(url)  
    self.connection = self.engine.connect()
```

```

session = sessionmaker(bind=self.connection.engine)
self.session = session()

```

#### Листинг Б.4 – Функции создания базы данных при инициализации

```

def create_db(self):
    self.connect(db_created=False)
    self.execute_query(f'DROP database IF EXISTS {self.db_name}')
    self.execute_query(f'CREATE database {self.db_name}')

```

#### Листинг Б.5 – Функции создания таблицы при инициализации

```

def create_table_test_users(self):
    if not inspect(self.engine).has_table('test_users'):
        Base.metadata.tables['test_users'].create(self.engine)

```

#### Листинг Б.6 – Часть класса с функциями создания данных в БД

```

from mysql.models import TestUsersModel
from generators.registration_data import BuilderRegData

class MysqlBuilder:
    def __init__(self, client):
        self.client = client
        self.builder = BuilderRegData()

    def create_new_user(self):
        user_data = self.builder.build()
        new_user = TestUsersModel(
            name=user_data['name'],
            surname=user_data['surname'],
            middle_name=user_data['middle_name'],
            username=user_data['username'],
            password=user_data['password'],
            email=user_data['email'],
            access=1,
            active=0,
            start_active_time=None
        )
        self.client.session.add(new_user)
        self.client.session.commit()

        return {
            "info": new_user,
            "user_data": user_data
        }

```

#### Листинг Б.7 – функция получения данных о пользователе из БД

```

def get_created_user(self, **filters):
    self.client.session.commit()
    res = self.client.session.query(TestUsersModel).filter_by(**filters)
    res = res.all()[0]
    return res

```

## Приложение В

### Листинг В.1 – Mock написанный с помощью Flask

```
import json
import signal
from flask import Flask, jsonify, request
from error_classes import ServerTerminationError
from settings import MOCK_HOST, MOCK_PORT

app = Flask(__name__)

user_data = {}

def exit_gracefully(signum, frame):
    print("Exit Gracefully called")
    raise ServerTerminationError()

signal.signal(signal.SIGINT, exit_gracefully)
# sigterm отправляется командой docker stop
signal.signal(signal.SIGTERM, exit_gracefully)

@app.route('/vk_id/add_user', methods=['POST'])
def add_user():
    username = json.loads(request.data)['username']
    vk_id = json.loads(request.data)['vk_id']
    if username not in user_data:
        user_data[username] = vk_id
        return jsonify({'user_id': user_data[username]}), 201
    else:
        return jsonify(f'User {username} already '
                        f'exists: id: {user_data[username]}'), 400

@app.route('/vk_id/<username>', methods=['GET'])
def get_vk_id(username):
    if user_id := user_data.get(username):
        payload = {'vk_id': str(user_id)}
        return jsonify(payload), 200
    else:
        return jsonify({}), 404

if __name__ == '__main__':
    try:
        app.run(MOCK_HOST, MOCK_PORT)
    except ServerTerminationError as e:
        print("Stopped Server")
```

## Приложение Г

### Листинг Г.1 – Конфигурация pytest

```
def pytest_configure(config):
    mysql_client = MysqlClient(
        user='root',
        password='toor',
        db_name='vkeducation'
    )
    if sys.platform.startswith('win'):
        base_dir = 'C:\\\\tests'
    else:
        base_dir = '/tmp/tests'
    if not hasattr(config, 'workerinput'):
        mysql_client.create_db()
    mysql_client.connect(db_created=True)
    if not hasattr(config, 'workerinput'):
        mysql_client.create_table_test_users()
    if not hasattr(config, 'workerinput'):
        if os.path.exists(base_dir):
            shutil.rmtree(base_dir)
        os.makedirs(base_dir)

    config.mysql_client = mysql_client
    config.base_temp_dir = base_dir
```

### Листинг Г.2 – Фикстура MySQL клиента

```
@allure.step("Creating a MySQL Client")
@pytest.fixture(scope='session')
def mysql_client(request) -> MysqlClient:
    client = request.config.mysql_client
    yield client
    client.connection.close()
```

### Листинг Г.3 – Функция создания пользователя в БД

```
def credentials(block: bool, mysql_client):
    m_builder = MysqlBuilder(mysql_client)
    access = 0 if block else 1
    result = m_builder.create_new_user(access, 0)
    return result["user_data"]["username"], result["user_data"]["password"]
```

### Листинг Г.4 – Метод API-клиента для авторизации

```
@allure.step(f"Authorization ")
def authorize(self):
    url_auth = urljoin(self.base_login_url, '/login')
    with allure.step(f"URL:{url_auth} - {self.user} - {self.password}"):
        pass
    headers = {
        'Referer': self.base_url
    }
    data = {
        "username": self.user,
        "password": self.password,
        "submit": "Login"
    }
    response = self.session.post(url=url_auth, data=data, headers=headers)
```

```
return response
```

### Листинг Г.5 – Метод-обертка функции request для API-клиента дл

```
def _request(self, method, location, headers=None, data=None,
             expected_status=200, jsonify=False, params=None, files=None):

    url = urljoin(self.base_url, location)
    response = self.session.request(method=method, url=url, headers=headers,
                                   data=data, params=params, files=files)

    if response.status_code != expected_status:
        raise error_cls.ResponseStatusCodeException(
            f'Got {response.status_code} {response.reason} for URL "{url}"'
        )

    if jsonify:
        json_response = response.json()
        if json_response.get('error', False):
            error = json_response['error']
            raise error_cls.RespondErrorException(
                f'Request {url} returned error {error["message"]}!'
            )
        return json_response
    return response
```

### Листинг Г.6 – Метод API-клиента для создания пользователя

```
@allure.step("User creation")
def post_create_user(self, payload, expected_status):
    location = '/api/user'
    payload = json.dumps(payload)
    headers = {
        'Content-Type': "application/json"
    }

    return self._request(
        method="POST", location=location, headers=headers, data=payload,
        expected_status=expected_status
    )
```

### Листинг Г.7 – Фикстура создания API-клиента заблокированного пользователя

```
@allure.step("Create a client API for a blocked user")
@pytest.fixture(scope="function")
def api_client_user_block(request, credentials_user_block) -> ApiClient:
    url = request.config.getoption('--url')
    username, password = credentials_user_block
    api_client = ApiClient(url, username, password)
    return api_client
```

### Листинг Г.8 – Базовый класс для API тестов

```
class BaseApi:

    authorize = True

    @pytest.fixture(scope="function", autouse=True)
    def setup(self, api_client_user_block, api_client_user_unlock):
```

```

self.api_client_user_block = api_client_user_block
self.api_client_user_unlock = api_client_user_unlock

if self.authorize:
    self.api_client_user_block.authorize()
    self.api_client_user_unlock.authorize()

```

## Листинг Г.9 – Классы с API тестам

```

import pytest
import allure
import pytest_check as check
from api.base import BaseApi
from api.error_classes import ResponseStatusCodeException
from api.input_data.api import InputData

class TestApiAddUserAuth(BaseApi):

    input_data = InputData()

    @pytest.mark.API
    @allure.title("Positive test for user creation and constraint check.")
    @pytest.mark.parametrize("user_data",
input_data.positive_test_cases_add_user())
    def test_positive_cases(self, user_data, mysql_client):
        response = self.api_client_user_unlock.post_create_user(user_data, 201)
        check.equal(
            response.status_code, 201,
            f"Status code {response.status_code}, expected 201!"
        )
        response_db =
mysql_client.get_created_user(username=user_data['username'])
        check.is_true(response_db, "User not created!")
        for key_user_data in user_data.keys():
            check.equal(
                user_data[key_user_data],
                eval(f"response_db.{key_user_data}"),
                f"Field value {key_user_data} transmitted incorrectly!"
            )
        if response_db:
            mysql_client.delete_user(username=user_data['username'])

    @pytest.mark.API
    @allure.title("Negative user creation test and constraint check.")
    @pytest.mark.parametrize("user_data",
input_data.negative_test_cases_add_user())
    def test_negative_cases(self, user_data, mysql_client):
        response = self.api_client_user_unlock.post_create_user(user_data, 400)
        check.equal(
            response.status_code, 400,
            f"Status code {response.status_code}, expected 400!"
        )
        response_db =
mysql_client.get_created_user(username=user_data['username'])
        check.is_false(response_db, "The user was created bypassing
restrictions!")
        if response_db:
            mysql_client.delete_user(username=user_data['username'])

```

```

    @pytest.mark.API
    @allure.title("Negative test for creating a user with an already existing
username.")
    def test_negative_username_already_use(self, mysql_client):
        user_data = self.input_data.user_username_already_use()
        response = self.api_client_user_unlock.post_create_user(user_data[0],
201)
        check.equal(
            response.status_code, 201,
            f"Status code {response.status_code}, expected 201!"
        )
        response = self.api_client_user_unlock.post_create_user(user_data[1],
400)
        check.equal(
            response.status_code, 400,
            f"Status code {response.status_code}, expected 201!"
        )
        response_db = mysql_client.get_created_user(email=user_data[1]["email"])
        check.is_false(response_db, "Created an entry with the same username!")
        if response_db:
            mysql_client.delete_user(username=user_data[1]['username'])
            mysql_client.delete_user(username=user_data[0]['username'])

class TestApiAddUserNoAuth(BaseApi):

    input_data = InputData()
    authorize = False

    @pytest.mark.API
    @allure.title("Negative test for creating a user without authorization.")
    def test_negative_add_user_no_auth(self, mysql_client):
        user_data = self.input_data.default_user()
        response = self.api_client_user_unlock.post_create_user(user_data[0],
401)
        check.equal(
            response.status_code, 401,
            f"Status code {response.status_code}, expected 401!"
        )
        response_db =
mysql_client.get_created_user(username=user_data[0]["username"])
        check.is_false(response_db, "User created without authorization!")
        if response_db:
            mysql_client.delete_user(username=user_data[0]['username'])

```

## Приложение Д

### Листинг Д.1 – Фикстура для генерации пути до файла с логами

```
@pytest.fixture(scope='function')
def temp_dir(request):
    test_dir = os.path.join(
        request.config.base_temp_dir,
        request._pyfuncitem.nodeid
    )
    test_dir = test_dir.replace(':', '--')
    os.makedirs(test_dir)
    return test_dir
```

### Листинг Д.2 – Фикстура для создания лог файла

```
@pytest.fixture(scope='function')
def logger(temp_dir, config):
    log_formatter = logging.Formatter(
        '%(asctime)s - %(filename)s - %(levelname)s - %(message)s'
    )
    log_file = os.path.join(temp_dir, 'test.log')
    log_level = logging.DEBUG if config['debug_log'] else logging.INFO
    file_handler = logging.FileHandler(log_file, 'w')
    file_handler.setFormatter(log_formatter)
    file_handler.setLevel(log_level)
    log = logging.getLogger('test')
    log.propagate = False
    log.setLevel(log_level)
    log.handlers.clear()
    log.addHandler(file_handler)
    yield log
    for handler in log.handlers:
        handler.close()
```

### Листинг Д.3 – Фикстура для создания конфигурации selenium

```
@allure.step("Creating a Selenium Assembly.")
@pytest.fixture(scope='session')
def config(request):
    browser = request.config.getoption('--browser')
    url = request.config.getoption('--url')
    debug_log = request.config.getoption('--debug_log')
    local_url = LOCAL_BASE_URL if request.config.getoption('--local') else None
    if local_url:
        url = local_url
    if request.config.getoption('--selenoid'):
        vnc = True if request.config.getoption('--vnc') else False
        selenoid = 'http://selenoid:4444/wd/hub'
    else:
        selenoid = None
        vnc = False
    return {
        'browser': browser,
        'url': url,
        'debug_log': debug_log,
        'selenoid': selenoid,
        'vnc': vnc,
        'local_url': local_url
    }
```



```
}
```

## Листинг Д.4 – Фикстуры инициализирующий Selenium Webdriver

```
@pytest.fixture()
def driver(config, temp_dir):
    browser = config['browser']
    url = config['url']
    selenoid = config['selenoid']
    vnc = config['vnc']
    local_url = config['local_url']
    options = Options()
    options.add_experimental_option(
        "prefs", {
            "download.default_directory": temp_dir
        }
    )
    if selenoid:
        capabilities = {
            "browserName": "chrome",
            'version': '99.0',
        }
        if vnc:
            capabilities['enableVNC'] = True
        driver = webdriver.Remote(
            'http://selenoid:4444/wd/hub',
            options=options,
            desired_capabilities=capabilities,
        )
    elif browser == 'chrome':
        driver =
webdriver.Chrome(executable_path=ChromeDriverManager().install(),
options=options)
    elif browser == 'firefox':
        driver =
webdriver.Firefox(executable_path=GeckoDriverManager().install())
    else:
        raise RuntimeError(f'Unsupported browser: "{browser}"')
    driver.get(url)
    driver.maximize_window()
    yield driver
    driver.quit()
```

## Листинг Д.5 – Фикстуры получения cookies для авторизации

```
@allure.step("Receiving cookies for authorization.")
@pytest.fixture(scope='session')
def cookies(config, credentials_user_unlock):
    cookies = []
    login, password = credentials_user_unlock
    if config['local_url']:
        api_client = ApiClient(LOCAL_BASE_URL, login, password)
    else:
        api_client = ApiClient(DOCKER_BASE_URL, login, password)
    api_client.authorize()
    cookies_dict = dict(api_client.session.cookies)
    for cookie_name in cookies_dict.keys():
        cookies.append(
```

```

        {
            "name": cookie_name,
            "value": cookies_dict[cookie_name]
        }
    )
    return cookies

```

## Листинг Д.6 – Класс хранящий локаторы для страницы авторизации

```

from selenium.webdriver.common.by import By
from ui.locators.base_locators import BasePageLocators

class LoginPageLocators(BasePageLocators):
    LOGIN_FORM = (By.XPATH, '//div[contains(@class, "uk-width-large")]')
    LOGIN_FORM_HEADER = (By.XPATH, '//h3[contains(@class, "uk-card-title")]')
    LOGIN_INPUT_FIELD = (By.XPATH, '//input[@id="username"]')
    LOGIN_ICON_USERNAME = (By.XPATH, '//span[contains(@uk-icon,
"user")]//*[local-name() = "svg"]')
    PASSWORD_INPUT_FIELD = (By.XPATH, '//input[@id="password"]')
    LOGIN_ICON_PASSWORD = (By.XPATH, '//span[contains(@uk-icon,
"lock")]//*[local-name() = "svg"]')
    LOGIN_BUTTON_INPUT = (By.XPATH, '//input[@id="submit"]')
    TEXT_NO_REGISTERED = (By.XPATH, '//div[contains(@class, "uk-text-small")]')
    HYPERLINK_CREATE_ACCOUNT = (By.XPATH, '//a[@href="/reg"]')
    LOGIN_ALERT = (By.XPATH, '//div[@id="flash"]')

```

## Листинг Д.7 – Класс Page Object для страницы авторизации

```

from urllib.parse import urljoin
import allure
from ui.pages.base_page import BasePage
from ui.locators.login_locators import LoginPageLocators
import restrictions

class LoginPage(BasePage):

    URL = urljoin(BasePage.URL, "/login")
    locators = LoginPageLocators()
    LOGIN_FORM_HEADER = "Welcome to the TEST SERVER"
    PLACEHOLDER_LOGIN_INPUT = "Username"
    PLACEHOLDER_PASSWORD_INPUT = "Password"
    SUBMIT_LOGIN_VALUE = "Login"
    TEXT_ALERT_LOGIN_INCORRECT = "Invalid username or password"
    TEXT_NO_REGISTERED = "Not registered?"
    TEXT_HYPERLINK_CREATE_ACCOUNT = "Create an account"
    LIST_FIELD = [
        {
            "locator": locators.LOGIN_INPUT_FIELD,
            "minlength": restrictions.MIN_LENGTH_USERNAME,
            "maxlength": restrictions.MAX_LENGTH_USERNAME,
            "name_field": PLACEHOLDER_LOGIN_INPUT,
            "locator_icon_field": locators.LOGIN_ICON_USERNAME,
            "required": True
        },
        {
            "locator": locators.PASSWORD_INPUT_FIELD,
            "minlength": restrictions.MIN_LENGTH_PASSWORD,
            "maxlength": restrictions.MAX_LENGTH_PASSWORD,
            "name_field": PLACEHOLDER_PASSWORD_INPUT,

```

```

        "locator_icon_field": locators.LOGIN_ICON_PASSWORD,
        "required": True
    }]

    @allure.step("Authorization.")
    def authorization(self, username, password):
        self.visibility_element(self.locators.LOGIN_FORM)
        self.fill_field(username, self.locators.LOGIN_INPUT_FIELD)
        self.fill_field(password, self.locators.PASSWORD_INPUT_FIELD)
        self.visibility_element(self.locators.LOGIN_BUTTON_INPUT)
        self.click(self.locators.LOGIN_BUTTON_INPUT)

```

## Листинг Д.8 – Фикстура конфигурации тестового класса

```

@pytest.fixture(scope='function', autouse=True)
def setup(self, driver, config, logger, request: FixtureRequest,
mysql_client):
    self.driver = driver
    self.config = config
    self.logger = logger

    self.mysql: MysqlClient = mysql_client
    self.builder: MysqlBuilder = MysqlBuilder(self.mysql)

    self.login_page: LoginPage = (request.getfixturevalue('login_page'))
    self.registration_page: RegistrationPage =
(request.getfixturevalue('registration_page'))
    self.base_page: BasePage = (request.getfixturevalue('base_page'))
    self.welcome_page: WelcomePage =
(request.getfixturevalue('welcome_page'))

    if self.authorize:
        cookies = request.getfixturevalue('cookies')
        for cookie in cookies:
            self.driver.add_cookie(cookie)
        self.driver.refresh()

```

## Листинг Д.9 – Фикстура для получения тестовый артефактов из браузера

```

@pytest.fixture(scope='function', autouse=True)
def ui_report(self, driver, request, temp_dir):
    failed_test_count = request.session.testsfailed
    yield
    if request.session.testsfailed > failed_test_count:
        browser_logs = os.path.join(temp_dir, 'browser.log')
        with open(browser_logs, 'w') as f:
            for i in driver.get_log('browser'):
                f.write(f"{i['level']} - {i['source']} - {i['message']}\n")
        screenshot_path = os.path.join(temp_dir, 'failed.png')
        driver.get_screenshot_as_file(screenshot_path)
        allure.attach.file(
            screenshot_path, 'failed.png', allure.attachment_type.PNG
        )
        with open(browser_logs, 'r') as f:
            allure.attach.file(
                f.read(), 'test.log', allure.attachment_type.TEXT
            )

```

## Листинг Д.10 – Часть тестового класса страницы авторизации

```

class TestLoginPage(base.BaseCase):
    """AUTH"""

    authorize = False

    @pytest.mark.UI
    @allure.title("Successful authorization.")
    def test_successful_authorization(self):
        """AUTH_001. Successful authorization"""
        response_data = self.builder.create_new_user(1, 0)

        response_data = response_data['user_data']
        login = response_data['username']
        password = response_data['password']

        self.login_page.authorization(login, password)
        self.driver.refresh()
        assert self.welcome_page.is_opened(), \
            "Authorization failed"

    @pytest.mark.UI
    # @pytest.mark.xfail(resolve="БАГ: Уведомление не пропадает")
    @allure.title("Failed authorization.")
    def test_negative_authorization(self):
        """AUTH_002. Failed authorization"""
        user_data = BuilderRegData().build()
        self.login_page.authorization(
            user_data["username"],
            user_data["password"]
        )
        assert self.login_page.is_opened(), \
            "Login page not open"
        assert self.login_page.visibility_element(
            self.login_page.locators.LOGIN_ALERT
        ), "Alert not visible"
        assert self.login_page.compare_text_elements(
            self.login_page.locators.LOGIN_ALERT,
            self.login_page.TEXT_ALERT_LOGIN_INCORRECT
        ), "Text is alert incorrect"
        self.driver.refresh()
        check.is_false(self.login_page.visibility_element(
            self.login_page.locators.LOGIN_ALERT,
            "Alert not gone"
        ))

```

## Приложение Е

### Листинг Е.1 – Хранимые шаги Allure

```
import allure
import pytest
from .steps import imported_step

@allure.step
def passing_step():
    pass

@allure.step
def step_with_nested_steps():
    nested_step()

@allure.step
def nested_step():
    nested_step_with_arguments(1, 'abc')

@allure.step
def nested_step_with_arguments(arg1, arg2):
    pass

def test_with_imported_step():
    passing_step()
    imported_step()

def test_with_nested_steps():
    passing_step()
    step_with_nested_steps()
```

## Приложение Ж

### Листинг Ж.1 – Dockerfile для Mock

```
FROM python:3.8

ADD requirements.txt /requirements.txt
RUN pip3 install -r /requirements.txt && rm -f /requirements.txt

ADD ./app mock
WORKDIR /mock
ENTRYPOINT ["python3", "vk_api.py"]
EXPOSE 8085
```

### Листинг Ж.2 – Dockerfile для БД MySQL

```
FROM mysql:latest
COPY init_db.sql /docker-entrypoint-initdb.d/
```

### Листинг Ж.3 – Dockerfile для проекта с тестами

```
FROM python:3.9
ADD requirements.txt /requirements.txt
RUN pip3 install -r /requirements.txt && rm -f /requirements.txt
WORKDIR /src
```

### Листинг Ж.4 – Bash скрипт для запуска тестов

```
#!/bin/bash

chmod -x tests
cd tests
ls -a
pytest -s -l -v "${PATH_TESTS}" -n "${THREADS}" --vnc --selenoid --alluredir
/tmp/allure
```

### Листинг Ж.5 – Docker - Compose

```
version: '2.1'

networks:
  selenoid:
    external: False
    name: selenoid

services:
  tests:
    build: config_tests
    tty: true
    networks:
      - selenoid
    volumes:
      - "./tests_project:/src"
```

```

        - "./alluredir:/tmp/allure"
        - /var/run/docker.sock:/var/run/docker.sock
working_dir: /src
entrypoint: /bin/bash /src/start_tests.sh
environment:
    - PATH_TESTS= #При необходимости запустить какой нибудь отдельный тест
    - THREADS=4
depends_on:
    myapp:
        condition: service_healthy

myapp:
    image: "myapp"
    networks:
        - selenoid
    volumes:
        - "./myapp_config/app_config.txt:/app/config/app_config.txt"
    command: /app/myapp --config=/app/config/app_config.txt
    ports:
        - "8083:8083"
    depends_on:
        db:
            condition: service_healthy
    healthcheck:
        test: [ "CMD", "curl", "-f", "127.0.0.1:8083" ]
        timeout: 10s
        retries: 15

myapp_proxy:
    image: "nginx:stable"
    networks:
        - selenoid
    volumes:
        - "./myapp_config/nginx/default.conf:/etc/nginx/conf.d/default.conf"
    depends_on:
        myapp:
            condition: service_started

db:
    image: "mysql:latest"
    networks:
        - selenoid
    volumes:
        - "./myapp_config/database/init_db.sql:/docker-entrypoint-
initdb.d/init_db.sql"
    environment:
        MYSQL_DATABASE: vkeducation
        MYSQL_ROOT_PASSWORD: toor
        MYSQL_USER: test_qa
        MYSQL_PASSWORD: qa_test
    ports:
        - "3306:3306"
    healthcheck:
        test: [ "CMD", "mysqladmin", "-uroot", "-ptoor", "ping", "-h", "db" ]
        timeout: 5s
        retries: 15

phpmyadmin:
    image: "phpmyadmin:latest"
    networks:
        - selenoid
    ports:
        - "8086:80"

```

```

environment:
  - PMA_ARBITRARY=1
  - PMA_HOST=db
  - PMA_USER=root
  - PMA_PASSWORD=toor
depends_on:
  - db

vk_api:
  build: tests_project/mock
  networks:
    - selenium
  ports:
    - "8085:8085"

selenium_chrome:
  image: "selenium/chrome:99.0"
  depends_on:
    - selenium

selenium:
  image: "aerokube/selenium:latest"
  networks:
    - selenium
  ports:
    - "4444:4444"
  volumes:
    - "./selenium:/etc/selenium"
    - "/var/run/docker.sock:/var/run/docker.sock"
  command: [ "-conf", "/etc/selenium/browsers.json", "-container-network",
"selenium", "-timeout", "5m" ]

selenium-ui:
  image: "aerokube/selenium-ui:latest"
  networks:
    - selenium
  ports:
    - "8888:8080"
  command: [ "--selenium-uri", "http://selenium:4444" ]
  depends_on:
    - selenium

```



## Приложение 3

### Листинг 3.1 – Pipeline – скрипт .groovy

```
properties([disableConcurrentBuilds()])

pipeline {
    agent {
        label 'master'
    }

    options {
        buildDiscarder(logRotator(numToKeepStr: '3'))
        timestamps()
    }

    environment {
        DOCKER_COMPOSE = "/usr/bin/compose"
    }

    stages {
        stage('Build') {
            steps {
                echo 'Building..'
            }
        }
        stage("Testing myapp") {
            steps {
                withEnv(["PATH+EXTRA=$DOCKER_COMPOSE"]) {
                    sh "cd $WORKSPACE/final_project"
                    dir("$WORKSPACE/final_project") {
                        sh "ls -a"
                        sh "docker compose up --abort-on-container-exit"
                    }
                }
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying....'
            }
        }
    }

    post {
        always {
            allure([
                reportBuildPolicy: 'ALWAYS',
                results: [[path: 'alluredir']]
            ])
            script {
                withEnv(["PATH+EXTRA=$DOCKER_COMPOSE"]) {
                    sh "cd $WORKSPACE/final_project"
                    dir("$WORKSPACE/final_project") {
                        sh 'docker compose down'
                    }
                }
            }
            cleanWs()
        }
    }
}
```