Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Высшая школа программной инженерии

Работа допущена к защите
Директор ВШПИ
_____П.Д.Дробинцев
«___»_____2024 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**магистерская диссертация**

# ФРЕЙМВОРК АВТОМАТИЗАЦИИ ТЕСТИРОВАНИЯ НА ПЛАТФОРМЕ DOTNET С ИСПОЛЬЗОВАНИЕМ СИСТЕМЫ УПРАВЛЕНИЯ ТЕСТИРОВАНИЕМ

по направлению подготовки (специальности)

**09.04.04 Программная инженерия**

Направленность (профиль)
**09.04.04_01 Технология разработки и сопровождения качественного программного продукта**

Выполнил студент гр. 5140904/20102 — К.Д.Савченко

Руководитель
Доцент — В.В.Амосов

Консультант
по нормоконтролю — Е.Г.Локшина

Санкт-Петербург
2024

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и кибербезопасности

Высшая школа программной инженерии

УТВЕРЖДАЮ

Директор высшей школы

П.Д. Дробинцев

« » 2024 г.

ЗАДАНИЕ

по выполнению выпускной квалификационной работы

студенту Савченко Константин Дмитриевич, 5140904/20102

1. Тема работы Фреймворк автоматизации тестирования на платформе dotnet с использованием системы управления тестированием
2. Срок сдачи студентом законченной работы: 30.05.2024
3. Исходные данные по работе: Литературный обзор по методикам автоматизации тестирования и инструментам разработки на платформе dotnet.
4. Содержание работы (перечень подлежащих разработке вопросов):
   - Анализ инструментов систем управления тестированием, совместимых с .NET.
   - Применение алгоритмов системы поддержки принятия решения для выбора системы управления тестированием.
   - Разработка собственного фреймворка автоматизации тестирования на платформе .NET.
   - Исследование эффективности и применимости разработанного фреймворка.
   - Проектирование и создание примеров использования фреймворка для автоматизации тестирования.
5. Перечень графического материала (с указанием обязательных чертежей):
   - Схемы архитектуры разработанного фреймворка.
   - Диаграммы, иллюстрирующие процессы автоматизации тестирования.
   - Снимки экрана с демонстрацией работы разработанного фреймворка.
   - Иллюстрации примеров использования системы управления тестированием в процессе разработки тестов.
6. Консультанты по работе:

_____

7. Дата выдачи задания 01.04.24

Руководитель ВКР _____ Амосов В. В.
(подпись) инициалы, фамилия

Задание принял к исполнению 01.04.2024
(дата)

Студент _____ Савченко К.Д.
(подпись) инициалы, фамилия

# Реферат

На 48 с., 28 рисунков, 4 таблицы, 4 приложения.

КЛЮЧЕВЫЕ СЛОВА: ТЕСТИРОВАНИЕ, АВТОМАТИЗАЦИЯ, МИКРОСЕРВИС, NUNIT, DOTNET.

Тема выпускной квалификационной работы: «Фреймворк автоматизации тестирования на платформе dotnet с использованием системы управления тестированием».

Данная работа посвящена автоматизации процессов тестирования программного обеспечения с помощью фреймворка автоматизации запуска тестов на платформе dotnet. Задачи, которые решались в ходе исследования:

- Анализ платформ реализации автоматизированного тестирования и систем управления тестированием;
- Выбор инструментария по разработке ПО на платформе dotnet;
- Построение архитектуры фреймворка автоматизации тестирования с учетом возможностей способов интеграций выбранной системы управления тестированием;
- Реализация собственного фреймворка автоматизации тестирования и его интеграция с системой управления тестированием;

Исходными данными к работе являлись публичные исследования в области автоматизации тестирования ПО. Выбор системы управления тестированием был произведен с помощью алгоритмов системы поддержки принятия решений.

В результате была спроектирована и разработана система запуска автоматизированных тестов, объединенная в общий фреймворк, реализующий запуск и публикацию результатов в системе управления тестированием. Гибкая микросервисная архитектура разработанного решения позволяет реализовать запуск тестов из системы управления “по требованию”, с минимальным участием пользователя.

# ABSTRACT

48 pages, 28 figures, 3 tables, 4 appendices.

keywords: TESTING, AUTOMATION, MICROSERVICE, NUNIT, DOTNET.

The subject of the graduate qualification work is "A testing automation framework on the dotnet platform using a testing management system."

This work is devoted to the automation of software testing processes using the test launch automation framework on the dotnet platform. Tasks that were solved during the research:

• Analysis of automated testing implementation platforms and testing management systems;

• Selection of software development tools on the dotnet platform;

• Building the architecture of the testing automation framework, taking into account the possibilities of integration methods of the selected test management system;

• Implementation of our own testing automation framework and its integration with the testing management system;

The initial data for the work were public research in the field of software testing automation. The choice of the test management system was made using the algorithms of the decision support system.

As a result, a system for running automated tests was designed and developed, combined into a common framework that implements the launch and publication of results in a test management system. The flexible microservice architecture of the developed solution allows you to run tests from the on-demand management system, with minimal user interaction.

# Содержание

# Введение

Обеспечение качества (Quality Assurance (QA)) — представляет собой совокупность мероприятий, охватывающих абсолютно все этапы разработки, выпуска и эксплуатации программного обеспечения. Это активности на всех этапах жизненного цикла ПО, которые предпринимаются для обеспечения заявленного требованиями уровня качества выпускаемого продукта. Контроль качества (Quality Control (QC)) — это часть комплекса QA в процессе разработки ПО, которая отвечает за анализ результатов тестирования, поиск ошибок и их устранение. Тестирование программного обеспечения (Software Testing) — одна из техник контроля качества, включающая в себя активности по планированию тестовых действий, дизайну тестов, их выполнению и анализу полученных данных т.е. уже непосредственно процесс проверки результатов работы на соответствие установленным требованиям. Непрерывное тестирование ускоряет поставку программного обеспечения, делая весь процесс более быстрым. А благодаря незамедлительной обратной связи, которая помогает уже на самых ранних этапах выявлять ошибки и другие проблемы в приложении, гарантирует, что команды разработки будут создавать высококачественные и надежные приложения. Кроме того, сама способность организовать и проводить эффективное тестирование может значительно снизить затраты в компании, как за счёт экономии времени разработчиков, так и вследствие создания непрерывного конвейера поставки, в котором они могут быстро вносить изменения в код с минимальными рисками нарушения работоспособности приложения в продуктивной среде, где продукт доступен пользователям.

Главным элементом непрерывного тестирования является его автоматизация. К преимуществам автоматизации относятся:

- Аккуратное и тщательное тестирование;

- Высокое покрытие тестами;
- Быстрое обнаружение ошибок;
- Повторное использование тестов;
- Более короткие сроки поставки;
- Экономия времени и денег;

В целях получения операционной отчетности о процессах тестирования, современные практики по построению тестовых активностей рекомендуют использования инструментов систем управления тестированием. Интеграция автоматизированного тестирования в такие системы кратно ускорят процессы проведения регрессионного и функционального тестирования.

Объектом исследования в данной работе является автоматизация бизнес-процессов тестирования программного обеспечения.

Целью работы является разработка фреймворка (бекенд приложения) запуска автоматизированных тестов на платформе dotnet и его интеграция с системой управления тестированием.

# 1. Обзор системы управления тестированием и способов интеграции

## 1.1 Тестирование как бизнес-процесс

Тестирование ПО – это, с одной стороны, этап разработки ПО, а с другой – специализированный бизнес-процесс в совокупности бизнес-процессов предприятия [1]. К функциям бизнес-процесса тестирования ПО относят:

- Определение, анализ и документирование производственных и организационных процессов отдела обеспечения качества;
- Стандартизация процессов, ролей, артефактов и шаблонов, используемых в отделе;
- Ускорение производственных процессов при сохранении высокого уровня качества;
- Оптимизация производственных и организационных активностей, направленных на повышение их эффективности и сокращение трудозатрат отдела обеспечения качества.

Процессная модель тестирования (рис. 1) состоит из групп процессов, в основе которых лежат процессы инструментальной поддержки тестирования.

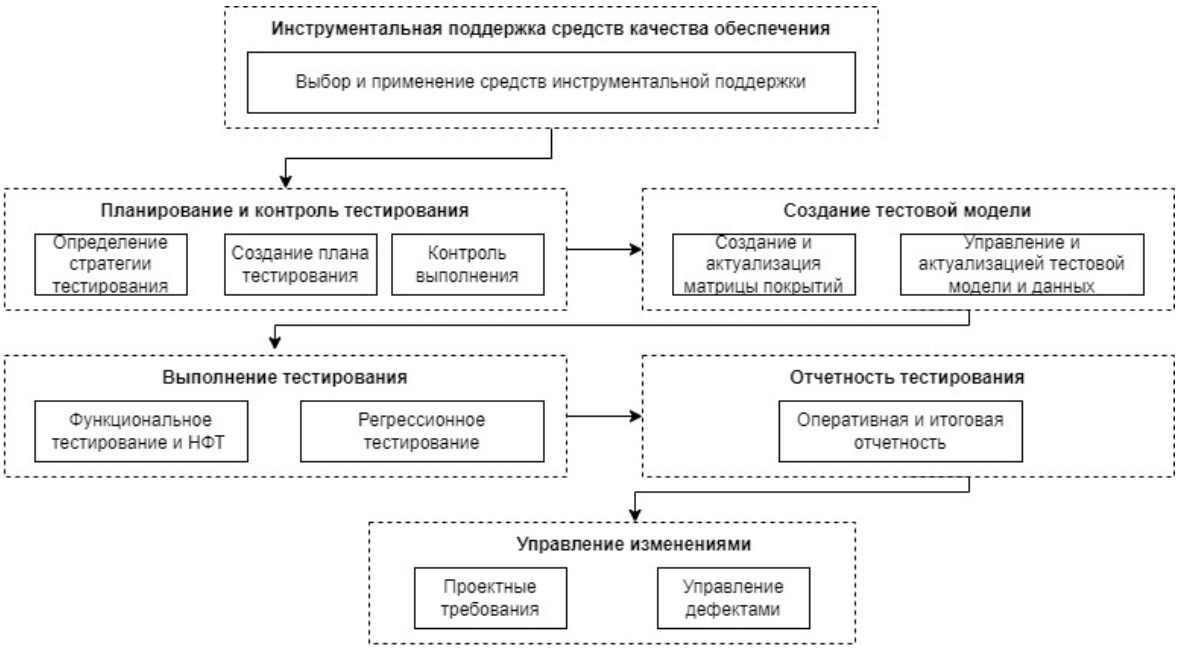



*Рисунок 1 – Процессная модель тестирования*

Существующие средства поддержки и автоматизации процессов разработки и тестирования упрощают реализацию отдельных прикладных задач [3]. Отдельный класс инструментов – системы управления тестированием (Test management system, TMS). Такие системы используются для хранения информации о том, как должным образом проводить тестирование, осуществление очередности проведения тестирования в соответствии с его планом, а также для получения информации в виде отчетов о стадии тестирования и качестве тестируемого продукта [5]. Для реализации этих целей TMS системы имеют функционал по учету численности команд отдела контроля качества, поддержке ролевой модели участников команд (лидер команды, тестировщик, администратор и т.д.); хранение и актуализация тестовых сценариев, формирование и хранение отчетов по прохождению тестовых прогонов в целом и выполнению тест-кейсов в частности.

## 1.2 Интеграция автоматизированного тестирования в системы управления тестированием

В рамках данной работы особый интерес представляет функционал TMS по поддержанию процессов автоматизированного тестирования, то есть возможность интегрироваться с фреймворками автоматизации тестов наиболее удобным образом. В ходе анализа существующих примеров по интеграциям систем управления тестированием на реальных предприятиях [6, 7], а также документаций по системам управления тестированием [8], были выявлены следующие способы интеграции:

- Использование API: Многие системы управления тестированием предоставляют API для интеграции с автоматизированными тестами. Публичный API может быть использован для отправки результатов выполнения автотестов в систему управления тестированием, а также для получения информации о текущем статусе тестов;
- Интеграция с CI/CD инструментами: Многие системы управления тестированием интегрируются с популярными инструментами непрерывной интеграции и доставки, такими как Jenkins, TeamCity, CircleCI и другими. Это позволяет автоматически запускать автотесты при каждом новом билде и отправлять результаты выполнения в систему управления тестированием;
- Использование специализированных библиотек: Некоторые автоматизированные тестовые фреймворки имеют специальные библиотеки для интеграции с системами управления тестированием. Например, Allure Framework предоставляет интеграцию с различными системами управления тестированием для отправки результатов выполнения тестов [9].
- Использование API/CLI для тестовых платформ: Многие системы управления тестированием предоставляют плагины или интерфейсы для загрузки отчетов из популярных платформ тестирования (Junit, NUnit и др.) [10]. Эти плагины облегчают интеграцию автотестов с

системой управления тестированием, позволяя загружать базовый отчет формата Junit в тестовый прогон.

- Webhooks – функционал проецирования действий в системе TMS на внешний узел. Такие действия как запуск тестов или остановка прогона могут быть спроецированы и обработаны внешней системой.

Для построения оптимального клиентского пути при работе с автоматизированными тестами из TMS необходима комбинация перечисленных выше методов. Так, наличие в системе перехватчика события о запуске тестов (webhook) позволяет реализовать запуск тестов "по требованию" (on demand). Таким образом, наличие различных способов интеграций в TMS является одним из ключевых факторов при выборе системы в данной работе.

### 1.3 Выбор системы управления тестированием

Выбор системы управления тестированием был произведен с помощью инструмента системы поддержки принятия решений (СППР).

Качественный подход к принятию решения подразумевает формализацию ситуации с использованием аппарата бинарных отношений или предпочтений, задание для каждого бинарного отношения весового коэффициента, проведение выбора решений с помощью аппарата функций выбора, поиск оптимальных вариантов решений для каждого бинарного отношения и сведение полученных результатов в обобщающую таблицу для наглядности и автоматизации выбора лицом, принимающим решение.

В системе применяются для оценки предпочтений бинарные отношения «больше» и «меньше» Аппарат функций выбора задаётся для каждого бинарного отношения механизмами доминирования, блокировки, турнирным механизмом и механизмом определения К - максимальных вариантов. Для каждого бинарного отношения задаются весовые

коэффициенты. Полученные по каждому механизму результаты ранжируются с учётом весовых коэффициентов бинарных отношений. [11].

Помимо наличия в системе перечисленных выше способов интеграции с TMS, параметрами для определения оптимального кандидата были:

1. Актуальная версия – год последнего обновления системы.
2. Стоимость лицензии, выраженная в рублях за месяц.
3. Количество дней пробного периода системы (для продуктов с открытым исходным кодом выставляется заведомо большое).
4. Ограничение по ресурсам RAM, ГБ.
5. Ограничение по ресурсам CPU.

На основе изученных источников на тему наиболее популярных систем управления тестированием за последние 5 лет [12, 13, 14] был выявлен ряд кандидатов среди коммерческих и open-source решений. Список кандидатов для сравнения, а также весовые коэффициенты и прочие входные данные для СППР по выбору TMS представлены в таблице 1.

*Таблица 1 - Матрица входных данных системы управления тестированием*

| TMS | Актуальная версия | Стоимость лицензии, руб. | Пробный период, дн | RAM | CPU | Webhooks | Allure Report | Адаптер NUnit |
|---|---|---|---|---|---|---|---|---|
| Test IT | 2023 | 3190 | 30 | 8 | 4 | 1 | 1 | 1 |
| Kiwi TMS | 2023 | 0 | $10^6$ | 2 | 1 | 0 | 0 | 0 |
| QuAsk | 2022 | 0 | $10^6$ | 1 | 1 | 0 | 0 | 0 |
| Allure TestOps | 2023 | 2800 | 14 | 8 | 4 | 0 | 1 | 1 |
| TestRail | 2023 | 3000 | 14 | 6 | 4 | 1 | 1 | 0 |
| TestLink | 2020 | 0 | $10^6$ | 4 | 2 | 0 | 0 | 0 |
| Параметры сравнения | | | | | | | | |
| *Весовые коэффициенты* | *0.05* | *0.05* | *0.1* | *0.1* | *0.1* | *0.35* | *0.15* | *0.1* |
| *Тип сравнения* | *1* | *0* | *1* | *0* | *0* | *1* | *1* | *1* |

В результате применения алгоритма СППР по бальной системе (Таблица 2) было определено, что среди коммерческих TMS наиболее оптимальный вариант для использования – система Test IT.

*Таблица 2 - Результат обработки данных СППР по выбору TMS*

| Вариант | Алгоритм работы СППР | | | | | Баллы |
|---|---|---|---|---|---|---|
| | Доминирования | Блокировки | Турнирный | Kmax(Sjp) | Kmax(Jjm) | |
| Test IT | 6 | 5 | 6 | 6 | 5 | 28 |
| Kiwi TMS | 3 | 5 | 4 | 4 | 5 | 21 |
| QuAsk | 4 | 6 | 3 | 3 | 6 | 22 |
| Allure TestOps | 3 | 5 | 2 | 2 | 5 | 17 |
| TestRail | 5 | 5 | 5 | 5 | 5 | 25 |
| TestLink | 2 | 5 | 1 | 1 | 5 | 14 |

### 1.4 Сценарий по запуску автоматизированных тестов из TMS

Учитывая тот факт, что выбранная система управления тестированием Test IT имеет все перечисленные выше способы интеграций с фреймворком автоматизации тестирования, запуск автоматизированных тестов может быть реализован по принципу on-demand непосредственно из TMS.

Сценарий по запуску автоматизированных тестов выглядит следующим образом:

1. Пользователь выбирает в системе автоматизированные тесты для запуска.
2. Система Test IT создает тестовый прогон (Test run), с пустыми карточками тестов.
3. С помощью настроенного webhook команда запуска транслируется на сервис обработки команды запуска.
4. Из тела запросов на запуск автоматизированных тестов извлекаются идентификаторы автотестов.
5. Производится последовательный запуск автотестов по указанному идентификатору теста;
6. Автотест генерирует Allure отчет.
7. С помощью дистрибутива Allure importer cli отчет вкладывается в созданный тестовый прогон.

8. После получения отчета по всем тестам система Test IT помечает прогон как пройденный. Тестировщик приступает к разбору результатов прогона.

На рисунке 2 данный процесс представлен в нотации бизнес процессов (BPMN) [15]. Стоит отметить, что в данном приближении поток управления “Сервис запуска” представлен как неделимый, но фактически – состоит из нескольких компонентов, что будет показано в архитектуре реализованного решения.

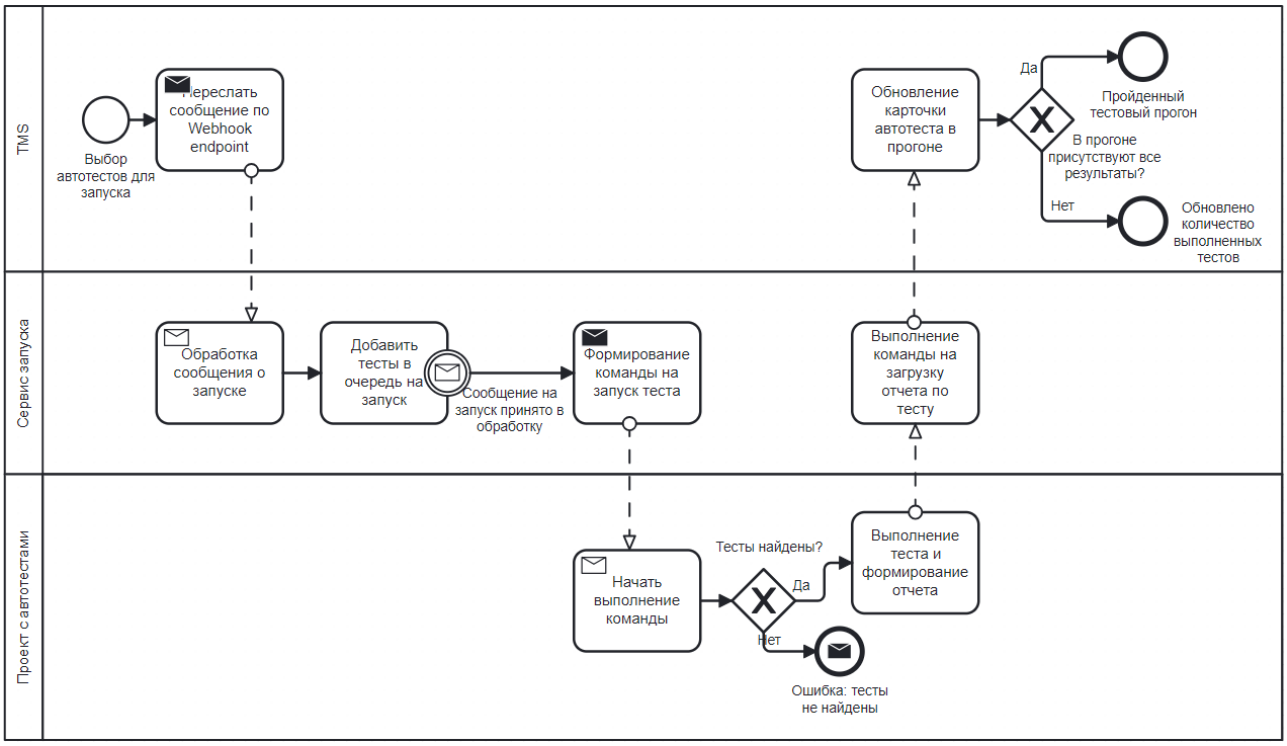



*Рисунок 2 – Нотация бизнес-процессов запуска автоматизированных тестов из TMS*

Таким образом, интеграция TMS с фреймворком автоматизации будет производится с учетом сценариев запуска тестов on-demand. В качестве программной системы для обработки команд запуска было принято решение реализовать собственное приложение на платформе dotnet. Данный вариант интеграции обладает большей гибкостью в сравнении с вариантами интеграции через CI системы и, следовательно, позволяет реализовать больше пользовательских сценариев.

### 1.5 Функциональные требования и ограничения

На основе обозначенного ранее базового сценария по запуску автоматизированных тестов из TMS и получения результатов выполнения можно составить расширенный набор сценариев по интеграции автоматизированных тестов с системой:

1. Запуск автоматизированных тестов с помощью прямого выбора карточек автотеста в системе;
2. Запуск автоматизированных тестов из составленного плана тестирования (коллекции ручных и автоматизированных тестов, объединенных в единую группу). Предполагается, что данный сценарий является расширением к обычному сценарию запуска, т.к. внутренние алгоритмы TMS Test IT автоматически выберут автоматизированные тесты в тест-плане и создадут советующий прогон;
3. Остановка выполнения тестового прогона. Сценарий изменения статуса тестового плана с запущенного на остановлен с одновременным удалением тестов из очереди на выполнения.

Диаграмма вариантов использования системы управления тестированием по интеграции с фреймворком автоматизации тестирования изображена на рисунке 3.

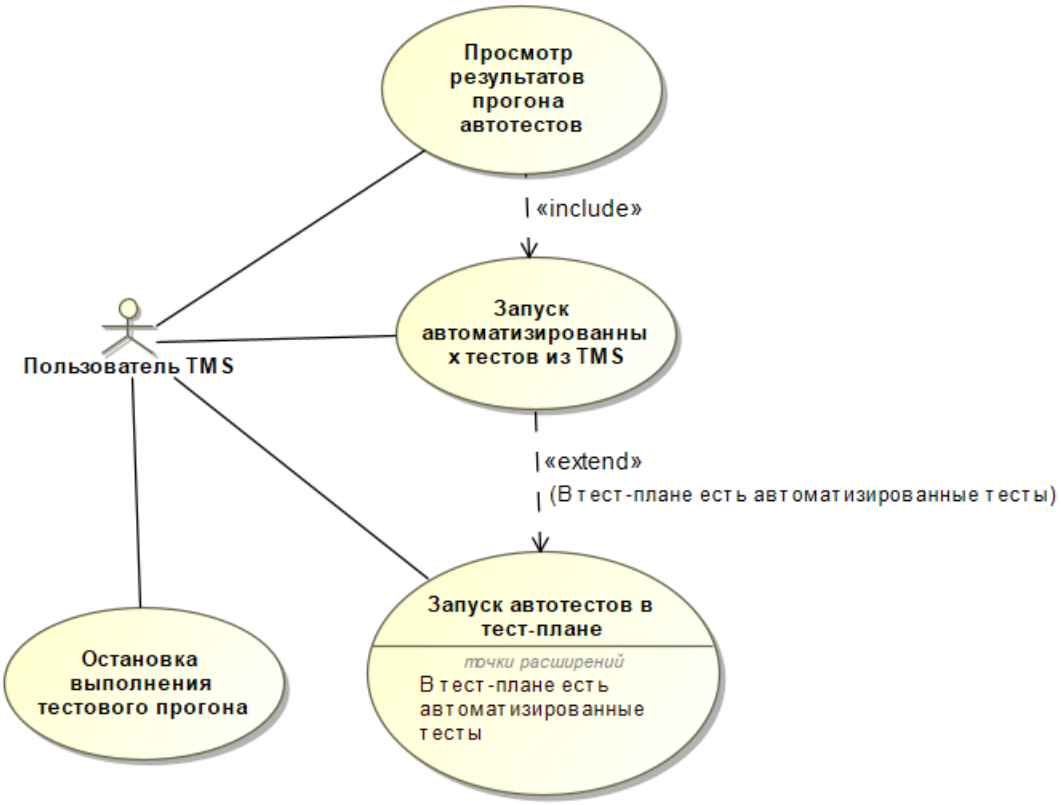



*Рисунок 3 – Диаграмма вариантов использования разрабатываемого решения*

Помимо требований к базовым сценариям по работе феймворка с TMS системой также были выделены требования к поддержке и расширению разрабатываемого решения, а именно:

1. Прием и обработка команд запуска тестового прогона из внешней системы (TMS);

2. Передача команды запуска на платформу NUnit;

После поступления запроса на запуск автотестов, система должна осуществить выполнить команду запуска указанных тестов.

3. Обработка и сохранение результата автоматизированного теста: в ходе выполнения тестового сценария, автотесты должен генерировать лог-файл с детализацией исполняемых действий Allure отчета. Вердикт прохождения автотреста должен быть одним из перечисленных: Passed (успешное прохождение теста), Failed (неуспешное прохождение теста, вызванное некорректной проверкой), Broken (непредвиденная ошибка в

ходе выполнение тестов). Фреймворк должен обеспечивать сохранение результатов в локальную директорию исполнения теста.

4. Отправка отчета выполненного теста в систему управления тестированием;

После выполнение теста, должна выпялятся операция отправки результата в TMS с использованием средства интеграции TMS (API, curl или Адаптер)

5. Возможность параллельного запуска;

Количество одновременно исполняемых тестов должно регулироваться вручную. Например, посредством инициализации дополнительных тестовых агентов (Job).

6. Логирование;

Система должна осуществлять логирование перечисленных операций: лог должен включать время выполнения команды, уровень информирование (Debug, Info, Warn, Error), а также дополнительную информацию, описывающее поведение системы.

7. Непрерывная доставка кода автоматизированных тестов;

Фреймворк должен поддерживать возможность загрузки актуальной сборки кода автотестов в указанную директорию, откуда производится запуск автоматизированных тестов.

Выполнение данных требований будет описано далее в разделе программной реализации.

В ходе настройки интеграции с системой управления тестированием Test IT был выявлен ряд особенностей в контракте работы системы с сущностью “Автотест”. Так, связь между карточкой ручного теста и автоматизированного сценария реализуется как “один ко многим”. В рамках разрабатываемой интеграции было принято решение принять ограничение на наличие единого автоматизированного теста, связанного с ручным сценарием. Также, карточки автоматизированных тестов, с которыми

предполагается наличие связи с ручным тестом, должны быть предварительно загружены в систему управления тестированием т.к. на начальном этапе разработки фреймворка не был учтен сценарий с автоматической генерацией карточки автотеста при первом запуске ручного теста из системы.

# 2. Обзор инструментов и технологий разработки

Выбор инструментов разработки любого программного обеспечения определяется, в первую очередь, решаемыми задачами разрабатываемой программы. От выбора оптимальных для поставленной задачи инструментов зависят качество, скорость и стоимость разработки.

На сегодняшний день можно найти наборы шаблонных программных решений для различных задач разработки, поэтому особое внимание при проектировании решений стоит уделять выбору технологий разработки. Качественный подбор инструментов избавляет разработчика от реализации излишних низкоуровневых операции, позволяя максимально сосредоточится на нюансах решаемой задачи.

В данном разделе будет произведен обзор инструментов и технологий для разработки фреймворка автоматизации тестирования на платформе dotnet.

## 2.1 Выбор среды разработки

Интегрированная среда разработки (IDE или единая среда разработки) — комплекс программных средств, используемый программистами для разработки программного обеспечения. Среда разработки включает в себя:

- текстовый редактор;
- компилятор;
- средства автоматизации сборки;
- отладчик.

В ходе аналитической работы, было проведено сравнение различных IDE, реализующих возможность работы с языком программирования C# (выбор языка описан в пункте 2.1). Результаты анализа сведены в таблице 3. В качестве источников для сравнения использовались материалы источников [22] и [23].

*Таблица 3 – Сравнение сред разработки*

| IDE | Достоинства | Недостатки |
|---|---|---|
| Visual Studio, разработчик: Microsoft. | • Официальная поддержка Microsoft;<br>• Бесплатная Community версия, где доступна возможность подключения сторонних и пользовательских расширений (в отличие от Express версии);<br>• Возможность расширение функционала с помощью дополнительных плагинов;<br>• Возможность непосредственного взаимодействия с облачными хранилищами;<br>• Поддержка технологий WPF (Windows Presentation Foundation) и Windows Froms;<br>• Амбассадоринг в образовательной среде технических специальностей; | • Сложная организация разделов меню;<br>• Нестабильная совместимость между платной и community версиями;<br>• Отсутствие кроссплатформенности (совместимость с Windows и macOS). |
| Project Reader, разработчик: JetBrains. | • Концепция инструмента основана на повышении производительности Visual Studio;<br>• Кросплатформенность; | • Относительно высокая стоимость и отсутствие полностью бесплатной версии (однако, имеются специальные предложения для студентов и непрофильных организаций). |
| Visual Studio Code, разработчик: Microsoft | • Кроссплатформенность;<br>• Легковестность, ресурсоэффективность;<br>• Бесплатность; | • Ограниченная функциональность, отсутствие средств полноценной IDE. |

При выборе среды разработки главными факторами были:

- Возможность объединения различных программных проектов (программных единиц) в единую структуру и предоставление зависимостей между проектами;
- Интеграция с системой контроля версий (GitHub);
- Возможность проведения отладки и профилирования;
- Система управления пакетами для платформ разработки;
- Наличие бесплатной (Community) версии в свободном доступе;

В качестве такой среды, описывающей все вышеописанные функции, была выбрана среда интегрированной разработки Microsoft Visual Studio версии 2022. Отличительным ее особенностей от аналогичных продуктов других компаний является ориентированность на язык программирования C#.

Ниже приведено описание выбранной среды разработки Visual Studio. В верхней части меню представлены средства запуска и отладки программы, выбора платформы запуска и переменных окружений. В обозревателе решений располагается иерархия проектов, над которыми ведется работа, таким образом имеется возможность проводить разработку над несколькими компонентами, например, проектом программной реализации алгоритмов автоматизированных тестов и проектом разработки среды запуска автотестов. В нижней части скриншота приведен вид обозревателя тестов, где представлен список находящихся в проекте автотестов, результаты выполнения конкретного теста и группировка информации о статусе тестирования (выполненных тестах).

В правом нижнем углу представляется удобная интеграция с системой контроля версий, имеется возможность выбора текущей ветки разработки, репозитория, возможность просмотра последних изменений в программном коде и его синхронизации с актуальной версией.

Стоит отметить, что вид отображения отдельных компонентов среды разработки гибко изменяется и масштабируется с помощью советующих разделов в настройках.

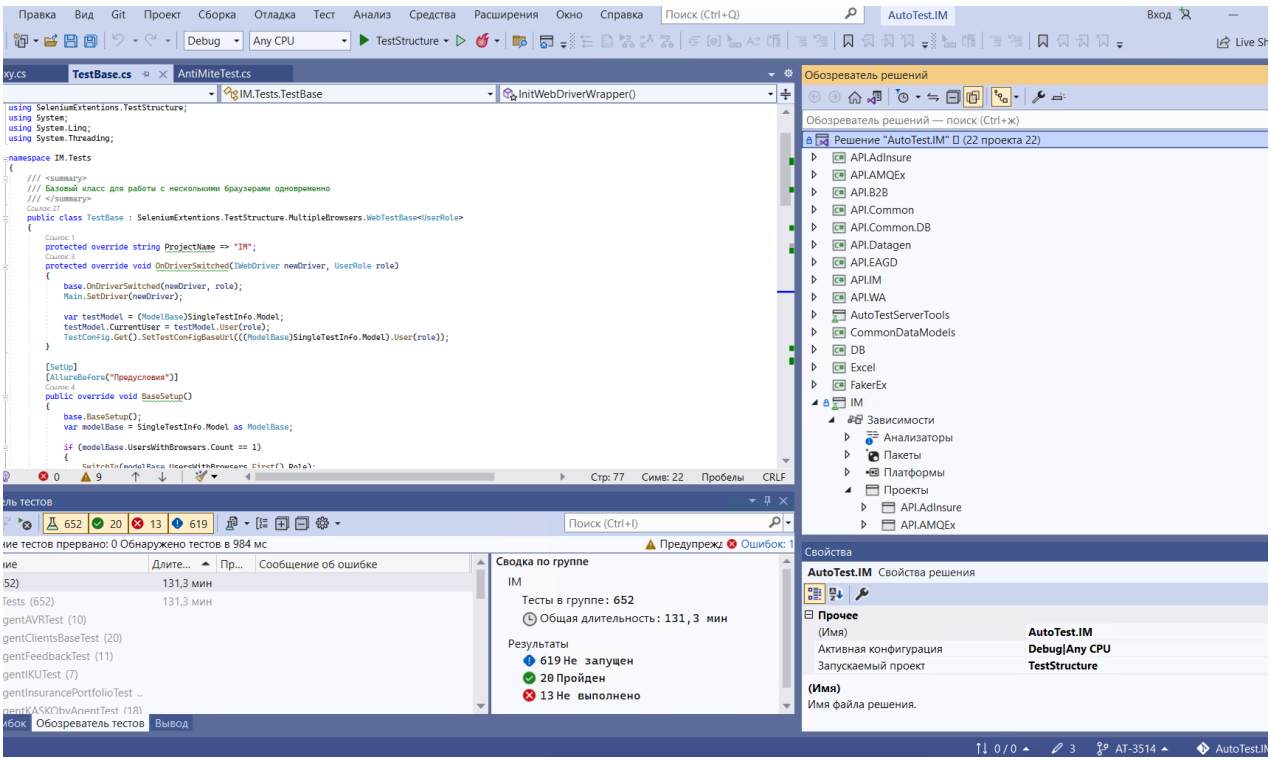



*Рисунок 4 – Главное окно среды разработки Visual Studio*

На скриншоте ниже приводится пример возможности одновременного запуска нескольких проектов, что в сильной степени облегчает и ускоряет работу над несколькими проектами.

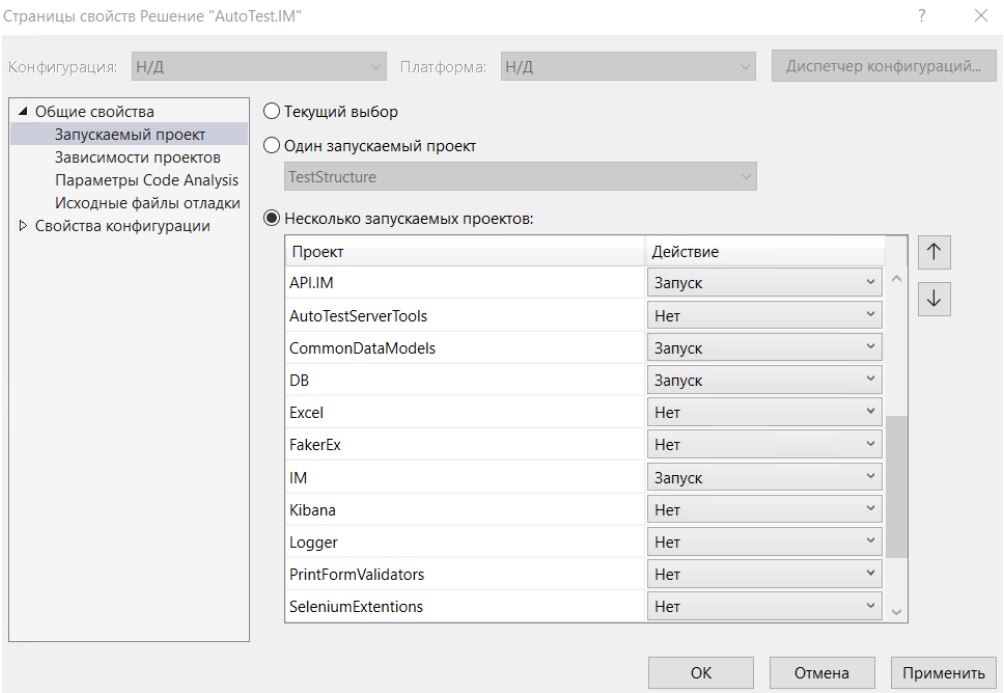



*Рисунок 5 – Настройка запуска нескольких проектов*

## 2.2 Языки разработки, библиотеки и фреймворки

В качестве языка платформы для разработки системы запуска тестов и реализации автоматизированных тестов на разработанном фреймворке выбрана платформа Microsoft .Net 6 (язык программирования C#). Программы, разработанные посредством Microsoft .Net, являются кроссплатформенными, что положительно выделяет данную платформу на фоне ее предшественника – Net Framework.

К преимуществам использования языка программирования C# и платформы .Net относят [24]:

- Автоматическое управление памятью и сборка мусора;
- Наличие встроенных синтаксических конструкции для работы с перечислениями, структурами и свойствами классов;
- Возможность перегружать операторы;
- Поддержка аспектно-ориентированных программных технологий (таких как атрибуты).

В качестве системы управления пакетами для выбранной платформы разработки используется система NuGet [25]. Основным ее преимуществом является высокая степень интеграции в среду разработки Visual Studio, поэтому отсутствует необходимость явно прописывать зависимости используемых библиотек для проектов, т.к. IDE делает это автоматически.

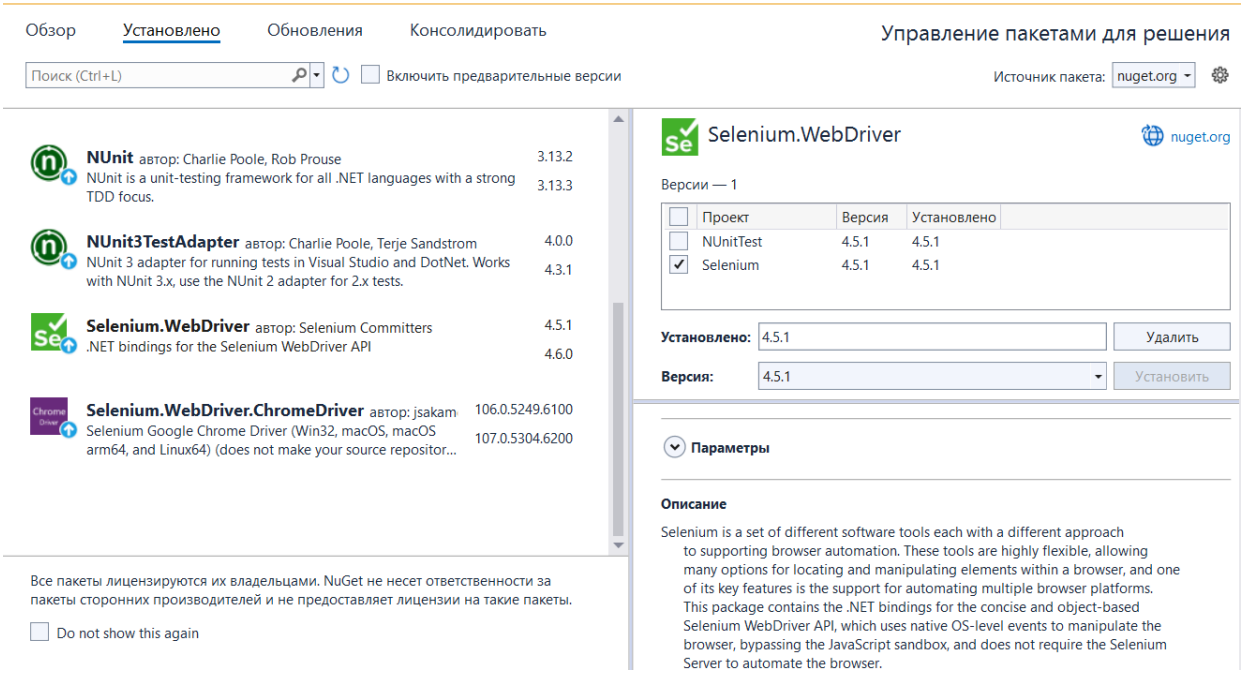



*Рисунок 6 – Использование системы NuGet в IDE Visual Studio*

### 2.3 Платформенный фреймворк тестирования

Для написания автоматизированных тестов как правило применяются специальные платформенные фреймворке тестирования. Платформа dotnet предоставляет следующие фреймворки:

- xUnit.net: фреймворк тестирования для платформы .NET. Наиболее популярный фреймворк для работы именно с .NET Core и ASP.NET Core;
- MS Test: фреймворк юнит-тестирования от компании Microsoft, который по умолчанию включен в IDE Visual Studio и который также можно использовать с .NET Core;
- NUnit: фреймворк тестирования с открытым исходным кодом, перенесенная из JUnit [21].

Данные фреймворки предоставляют несложный API, который позволяет быстро написать и автоматически проверить тесты.

В качестве платформенного фреймворка тестирования в среде dotnet был выбран NUnit (версия 3.12.0). Данный инструмент интегрируется в среду разработки Visual Studio. Проект модульного тестирования может быть

добавлен как через шаблонный проект в конструкторе проектов Visual Studio, так и с помощью соответствующего NuGet пакета. В среде разработки Visual Studio инструмент модульного тестирования отображает имеющиеся тесты в соответствующем окне “Обозреватель тестов”.

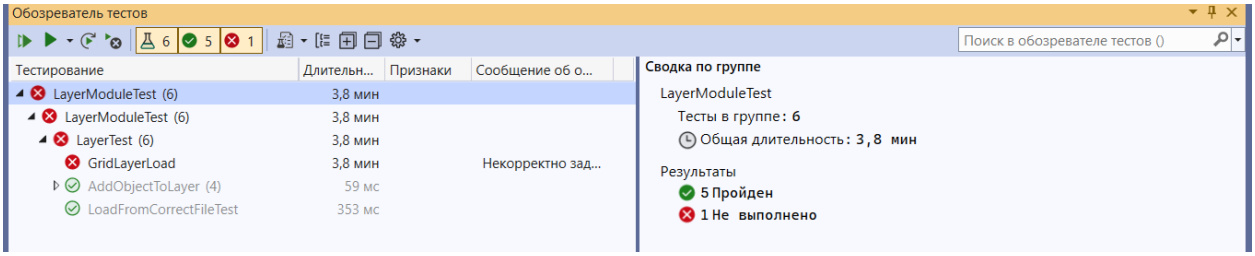



*Рисунок 7 - Окно "Обозреватель тестов" в IDE Visual Stuio*

Агрегация тестов осуществляется в соответствии с разделением на тестовые классы и методы. Для указания принадлежности определенного класса к тестовому используется атрибут TestFixture, а для отметки метода как теста – атрибут Test.

```
8   {
9       [TestFixture]
        Ссылок: 0
10      public class LayerTest
11      {
12          [Test]
            Ссылок: 0
13          public void LoadFromCorrectFileTest()
14          {
15              string layerMifFile = "mifLayer.mif";
16              string filePath = Path.Combine(Path.GetDirectory
17              VectorLayer layer = new VectorLayer();
18              layer.LoadFromFile(filePath);
19
20              Assert.Multiple(() =>
21              {
22                  Assert.AreEqual(1, layer.Objects.Where(o =>
23                  Assert.AreEqual(1, layer.Objects.Where(o =>
24                  Assert.AreEqual(1, layer.Objects.Where(o =>
25                  Assert.AreEqual(1, layer.Objects.Where(o =>
26              });
27          }
```

*Рисунок 8 - Использование базовых атрибутов NUnit*

Запуск теста или группы тестов может осуществляться через графический интерфейс среды разработки:

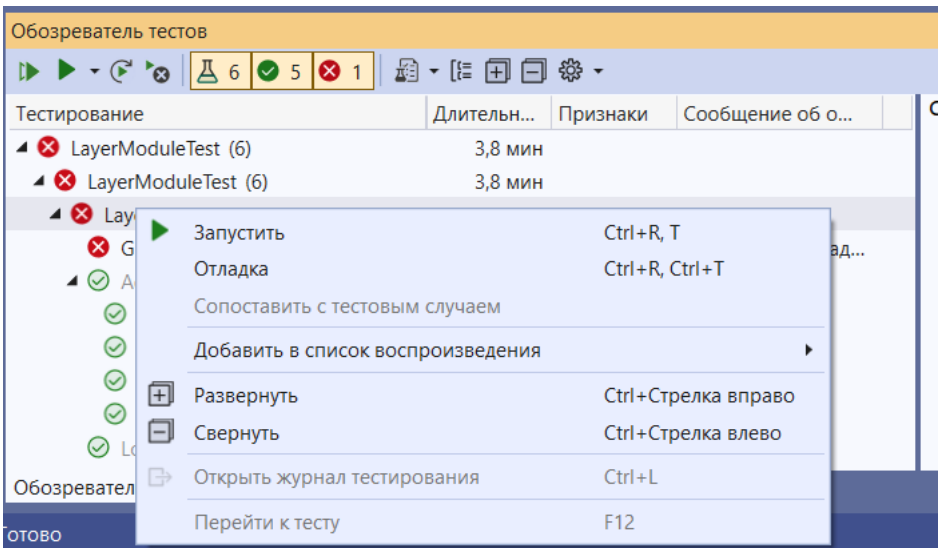



*Рисунок 9 - Запуск тестов из среды разработки*

Также запуск тестов доступен и через консоль

```
D:\TPUProjects\Geoinformatika(29.03.2022)\LayerModuleTest\bin\Debug\net48>dotnet test LayerModuleTest.dll
Программа Microsoft (R) Test Execution Command Line Tool версии 17.1.0
(c) Корпорация Майкрософт (Microsoft Corporation). Все права защищены.

Запуск выполнения тестов; подождите...
Общее количество тестовых файлов (1), соответствующих указанному шаблону.
  Не пройден GridLayerLoad [124 ms]
  Сообщение об ошибке:
     Некорректно заданы границы у слоя GridLayer после интерполяции из векторного слоя
  Expected: True
  But was:  False

  Трассировка стека:
     в LayerModuleTest.LayerTest.GridLayerLoad() в D:\TPUProjects\Geoinformatika(29.03.2022)\LayerModuleTest\LayerTest.c
s:строка 65


Не пройден!: не пройдено     1, пройдено     6, пропущено     0, всего     7, длительность . - LayerModuleTest.dll (net4
8)

D:\TPUProjects\Geoinformatika(29.03.2022)\LayerModuleTest\bin\Debug\net48>_
```

*Рисунок 10 – Запуск тестов в среде NUnit через консольную команду*

Для непосредственных проверок и вывода ошибки (логирования) в случае неуспешной проверки в NUnit применяются функции-утверждения – Assert.

### 2.4 Брокеры сообщений

Брокер сообщения является типом архитектурного построения, которое позволяет приложениям, системам и службам обмениваться информацией путем перевода сообщений между формальными протоколами обмена сообщениями. Это позволяет взаимозависимым

сервисам взаимодействовать друг с другом напрямую, даже если они были написаны на разных языках или реализованы на разных платформах [26].

Современная экосистема программного обеспечения построена на распределенных системах, при этом брокеры обмена сообщениями являются важнейшим компонентом управления связью между службами. RabbitMQ, Kafka и ActiveMQ — три популярных брокера обмена сообщениями, каждый из которых обладает уникальными функциями и вариантами использования.

В разрабатываемой системе запуска автоматизированных тестов для обмена сообщений между программными сервисами будут применятся брокер сообщений Apache Kafka.

Применение брокера сообщений Apache Kafka позволит масштабировать функционал системы и повысить ее отказоустойчивость. Принцип гарантированной доставки сообщений обеспечивает получение сообщений потребителем (consumer) при его работоспособности в конкретный момент времени, а в случае недоступности потребителя – сообщение будет сохранено в хранилище kafka и будет вычитано в момент восстановления соединения с брокером.

Утилитой для тестирования взаимодействия с кластером Kafka является Offset Explorer. Основным ее преимуществом является удобный пользовательский интерфейс (рис. 11). В структуре директорий кластера в отдельных папках собраны топики кластера и потребители сообщений. В инструменте визуализации сообщений из топика можно получить информацию о значении сообщения, времени отправки, ключа сообщения, а также номера патриции (Partition) и порядковому номеру сообщения в топике (Offset).

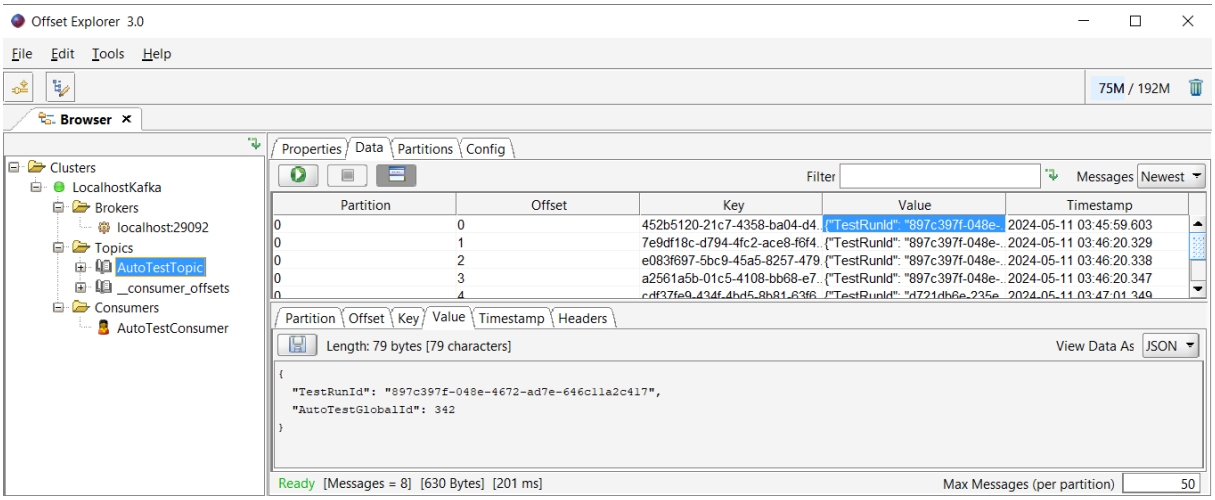



*Рисунок 11 – Пользовательский интерфейс утилиты Offset Explorer*

### 2.5 Вывод по разделу

Таким образом, в ходе работы были рассмотрены технологии и фреймворки для разработки фреймворка автоматизации тестированием. Программную основу фреймворка составляет платформа dotnet (версия 6) и язык программирования C#. Система запуска автоматизированных тестов взаимодействует с средой тестирования NUnit, с помощью которой должны быть реализованы автоматизированные тесты для запуска. Выбранная среда разработки – Visual Studio имеет функционал по взаимодействию с менеджером пакетов NuGet и средой тестированием NUnit. Для межсервисного взаимодействия между компонентами системы запуска автоматизированных тестов предполагается использование брокера сообщений Apache Kafka.

## 3. Программная реализация

### 3.1 Архитектура фреймворк

Микросервис – это независимый, автономный ресурс, спроектированный как отдельный выполняемый файл или процесс и взаимодействующий с другими микро сервисами через стандартные, но легковесные межпроцессные связи, такие как протокол передачи

гипертекста (HTTP), веб-службы RESTful (построенные на архитектуре репрезентативной передачи состояния – Representational State Transfer, REST), очереди сообщений и т. п. Уникальность микросервисов обусловлена тем, что каждый из них разрабатывается, тестируется, развертывается и масмасштабируется независимо от других микросервисов. Идея использования микросервисов основана на лучших принципах разработки программного обеспечения, в том числе таких, как слабая взаимозависимость, высокая масштабируемость и ориентированность на службы [20]. Выполняемые фреймворком функциональные задачи были логически выделены в отдельные сервисы, которые и составляют основу архитектуры разрабатываемого фреймворка, а именно:

- Сервис обработки сообщения о запуске тестов из системы управления тестированием. Программное название данного сервиса – HookKafkaProducer;
- Сервис обработки сообщения о запуске теста и оркестрации тестовыми агентами – программное название: KafkaConsumer;
- Приложение запуска теста из указанной директории и отправки результатов в TMS – тестовый агент, RunnerClient.

Отдельным компонентом, но не микросервисом, в архитектуре фреймворка является сборка тестов для запуска.

Диаграмма размещения компонентов системы запуска автотестов представлена на рисунке 12.

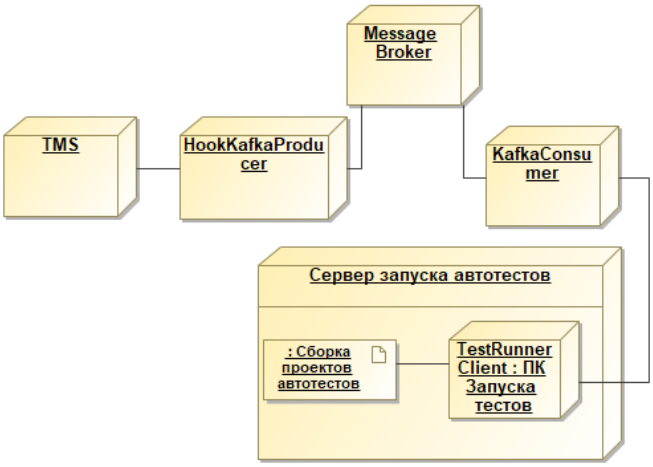



*Рисунок 12 – Диаграмма развертывания системы запуска тестов*

Далее будет описано программное устройство каждого компонента, принципов их разработки и контракты межсервисного взаимодействия.

### 3.2 Взаимодействие среды тестирования с фреймворком

Основным артефактом выполнения автоматизированного теста является отчет выполнения. Разработанный фреймворк тестирования взаимодействует с библиотекой Allure Report, которая создает, обогащает и сохраняет отчет по ходу выполнения автоматизированного теста. Адаптер сохраняет данные в форматах файлов Allure в отдельный каталог.

Сгенерированные файлы включают в себя:

- файлы результатов тестирования для описания выполнения тестов,
- файлы-контейнеры для описания вложений,
- вложения файлов (скриншоты, pdf и т.д.).

Таким образом, отчет о выполнении одного теста содержит набор файлов. Для удобной загрузки allure отчета (группы файлов) в систему управления тестированием – необходимо, чтобы тест сохранял отчет в уникальной директории. Для этой цели были разработаны методы вспомогательного класса PrepareOutputDirectoryFixture. Метод данного класса, помеченный атрибутом OneTimeSetUp выполняется перед запуском теста и генерирует директорию с отчетом allure, устанавливая при этом

переменную окружения ALLURE_CONFIG_ENV_VARIABLE в значение пути созданной директории. Имя директории задается из параметров команды запуска теста. Полный листинг вспомогательного класса представлен в приложении 3 .

Фильтрация тестов для запуска будет осуществляется по его уникальному идентификатору – Id теста из системы Test IT. Для обеспечения выполнения данного требования каждый тестовый метод является параметризованным. В качестве параметра выступает модель теста. Среда NUnit реализует атрибут TestCaseSource, принимающий в качестве параметра название статического метода, генерирующего список входных моделей. Функционал такого генератора был реализован в классе FileInputParamsBase (Приложение 4– Листинг класса генератора тестовых моделей FileInputParamsBase. Класс является типизированным (Generic) с типом входного параметра – класс модели теста. Данный класс используется для десериализации модели из json файла с данными для модели. Конструктор класса принимает путь до файла с данными. Тестовый метод используемы для запуска в фреймворке должен содержать ряд обязательных атрибутов:

```
private static FileInputParamsBase<ModelBase> NewTestSource => new
FileInputParamsBase<ModelBase>("TestData\\" +
"NotImplemendetBuildTest.json");
[Test, TestCaseSource(nameof(NewTestSource))]
public void NewTest(ModelBase data){/*Test method*/}
```

Файл с данными для тестового метода может содержать массив моделей, например:

```
{
```

```
  "ModelBase": [
    {
      "ManualTestId": 18
    },
    {
      "ManualTestId": 25
    }
  ]
}
```

Среда NUnit преобразует данный файл в список тестов. В обозревателе тестов метод NewTest будет отображаться следующим образом:

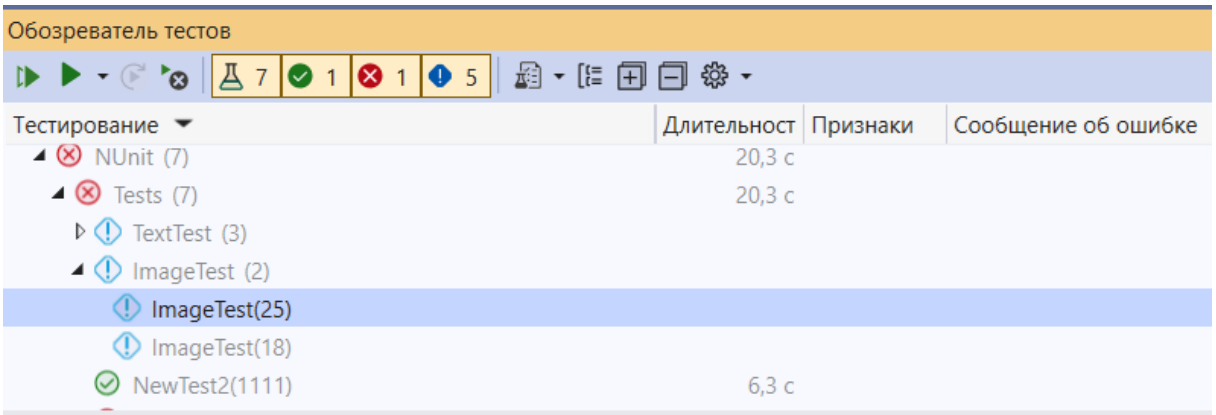



*Рисунок 13 – Отображение параметризированного теста в обозревателе тестов Visual Studio*

Таким образом, функциональность по генерации отчётности прохождения теста реализована в функциональном компоненте сборки автотестов. Фреймворк накладывает ряд ограничений на принцип написания автотестов в числе которых: обязательное наличие файла с содержанием тестовой модели, пометка тестов атрибутами TestCaseSource. Также функциональность фреймворка имеет зависимость от NuGet библиотек Allure. Однако, реализованный подход позволяет избавиться от функциональных зависимостей по интеграции с системой управления тестированием – логика по непосредственной интеграции с TMS реализована в микросервисах системы запуска.

### 3.3 Агент запуска тестов

Программной службой, непосредственно взаимодействующей со сборкой автотестов, является микросервис RunnerClient, осуществляющий выполнение команды на запуск теста из указанной директории, а также загрузку результата в систему управления тестированием. Тестовый агент производит запуск тестов по их идентификатору в системе Test IT. В ходе работы по реализации фреймворка была выявлена особенность в модели данных системы Test IT, а именно – различия в сущностях автоматизированного и ручного теста. Поэтому, при получении команды на запуск теста, компонент тестового агента использует методы адаптера Test IT для получения идентификатора ручного теста, связанного с автоматизированным. Команда для запуска автоматизированного теста из компонента “Тестовый агент” выглядит следующим образом:

$$CmdHelper.ExecuteCommand(\$"dotnet\ test\ \{Path.GetFileName(RunnerConfig.DllPath)\} - v\ n \\ --filter\ "Name \sim \{manualTestId\}" \\ --TestRunParameters.Parameter(name = "TestRunId",\ value = \{testRunReportFolder\})", \\ workingDirectory : Path.GetDirectoryName(RunnerConfig.DllPath));$$

(1) где,

RunnerConfig.DllPath – путь до динамической библиотеки в сборке проекта автотестов;

manualTestId – идентификатор теста в фреймворке, совпадающий со значением идентификатора ручного теста в TMS;

testRunReportFolder – идентификатор директории для сохранения Allure отчета теста.

Сервис является веб – сокет клиентом и получает сообщения от сервера. Для взаимодействия по протоколу WebSocket используется стандартная библиотека платформы dotnet – System.Net.WebSockets.

Взаимодействие сервиса с системой управление тестированием реализуется посредством утилиты TestITAllureImporter – загрузчика Allure отчета в систему [27]. Утилита представляет собой набор Python скриптов и

исполняемы файл. Для удобного взаимодействия с утилитой был реализован скрипт bat файл, реализующий операцию загрузки отчета в 2 действия:

1. Инициализация окружения утилиты: установка url системы и указания private token переменной;
2. Вызов утилиты с передачей параметров --testRunId и --resultDir.

Процесс взаимодействия сервиса с зависимыми компонентами изображен на рисунке 14.

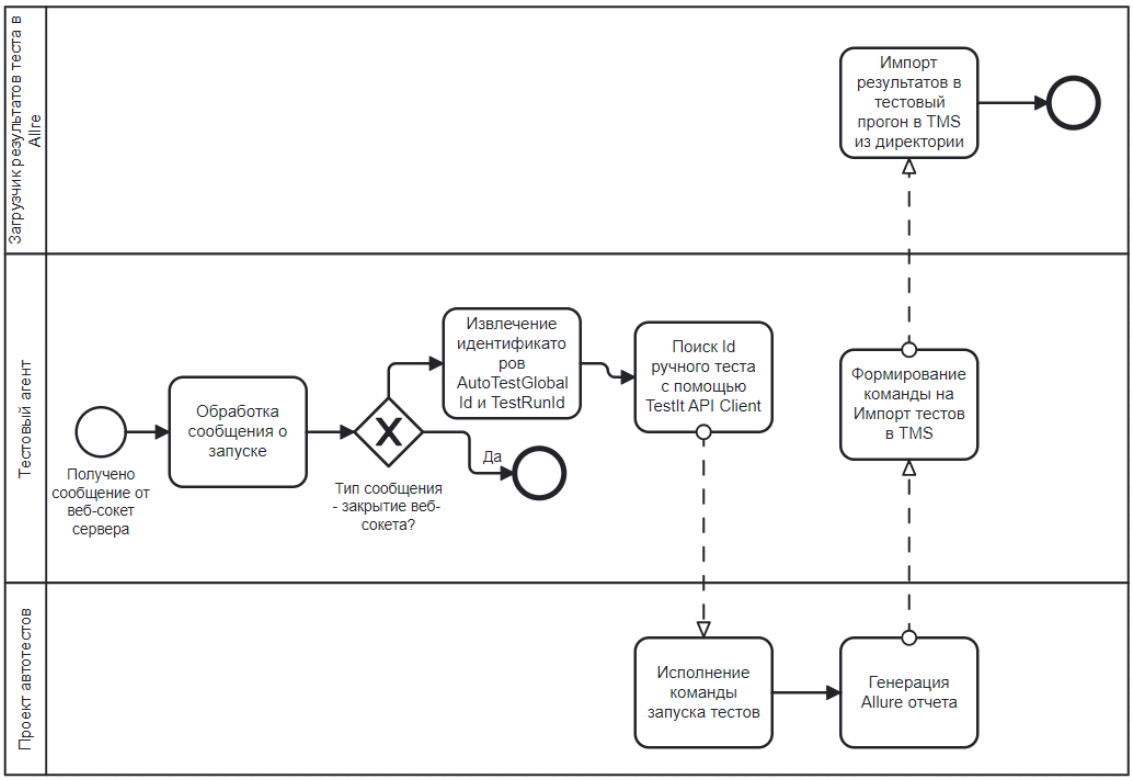



*Рисунок 14 – Диаграмма процессов работы тестового агента по запуску теста.*

### 3.4 Сервер тестовых агентов

Сервером для тестовых агентов является сервис с программным названием KafkaConsumer. К функциям данного сервиса относится

1. Прослушивание и прием сообщений о запуске теста из топика брокера Apache Kafka;
2. Актуализация списка активных тестовых агентов;

3. Оркестрация набором подлеченных тестовых агентов и асинхронная передача сообщений на запуск тестов свободным агентам.

Для взаимодействия с брокером сообщений Apache Kafka используется библиотека Confluent.Kafka. Подключение потребителя сообщений к топику выглядит следующим образом:

```
using var consumer = new ConsumerBuilder<Ignore, string>(new ConsumerConfig
            {
                BootstrapServers = Configuration["Kafka:BootstrapServers"],
                GroupId = "AutoTestConsumer",
                AutoOffsetReset = AutoOffsetReset.Earliest
}).Build();
```

Формат сообщений для команды запуска тестов представляет собой JSON документ, содержащий поля TestRunId и AutoTestGlobalId. При получении такого сообщения через топик “AutoTestTopic” сервис запускает процесс по поиску свободного тестового агента и передаче сообщения на исполнение теста.

На рисунке 15 изображена диаграмма работы сервиса в нотации BPMN. Взаимодействие между сервером и клиентом по подписке обеспечивает выполнение требования к масштабируемости фреймворка – возможность параллельного запуска тестов на нескольких агентах.

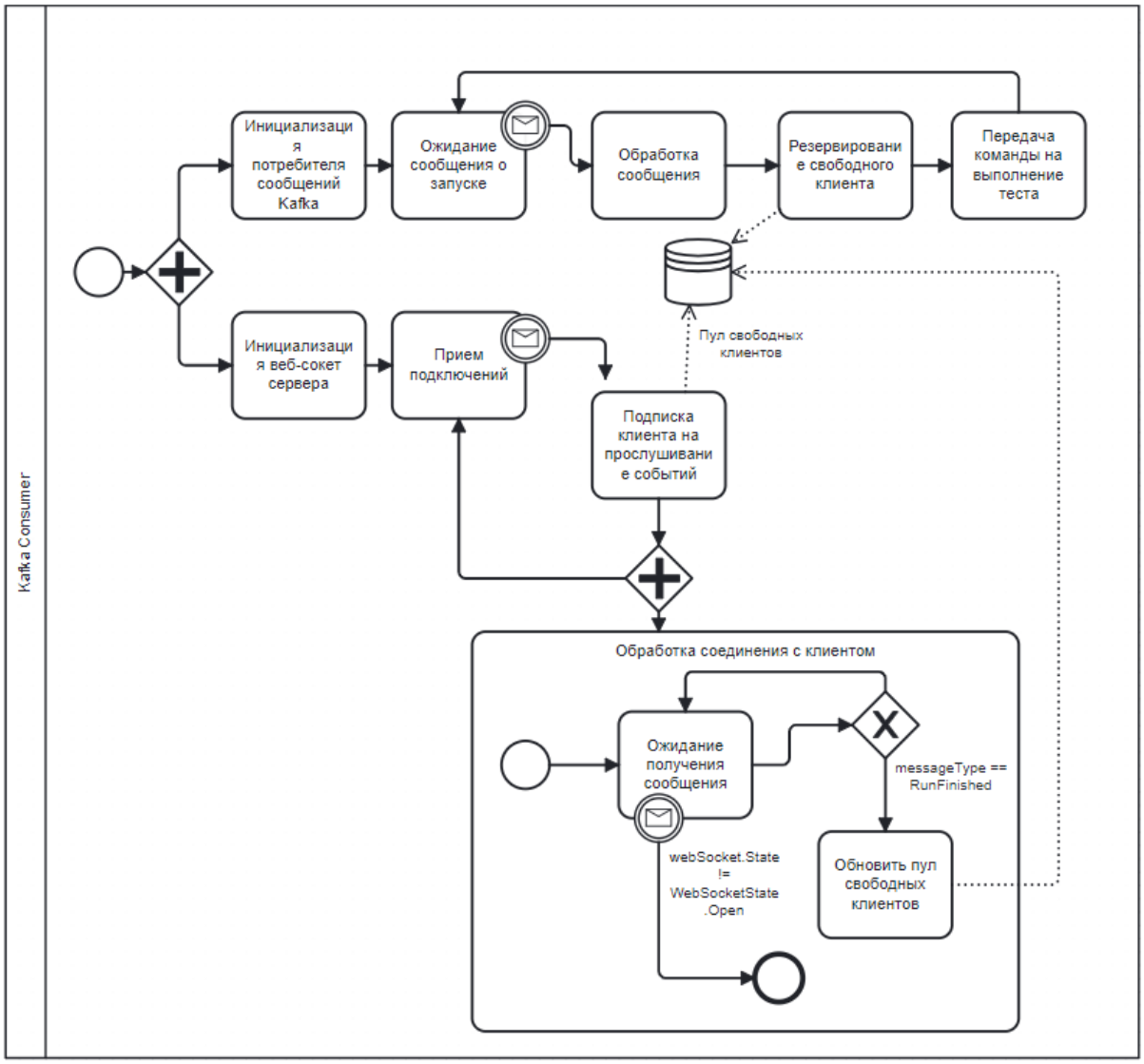



*Рисунок 15 – Диаграмма процессов работы сервиса Kafka Consumer*

### 3.5 Сервис Webhook API

WebHook API перекладчик представляет собой WebApi приложение, реализующее единственный метод “/api/hook” типа POST, на вход которого передается Json сообщение о запуске тестового прогона. В данном сообщении полезной информацией для фреймворка являются поля TestRunId и список AutoTestGlobalId. Для каждого автотеста перекладчик транслирует сообщение в топик брокера Kafka. Таким образом формируется очередь на запуск автотестов.

Для удобства проверки доступности и работоспособности предоставляемых сервисом API методов в конфигурации приложения был добавлен инструмент Swagger.

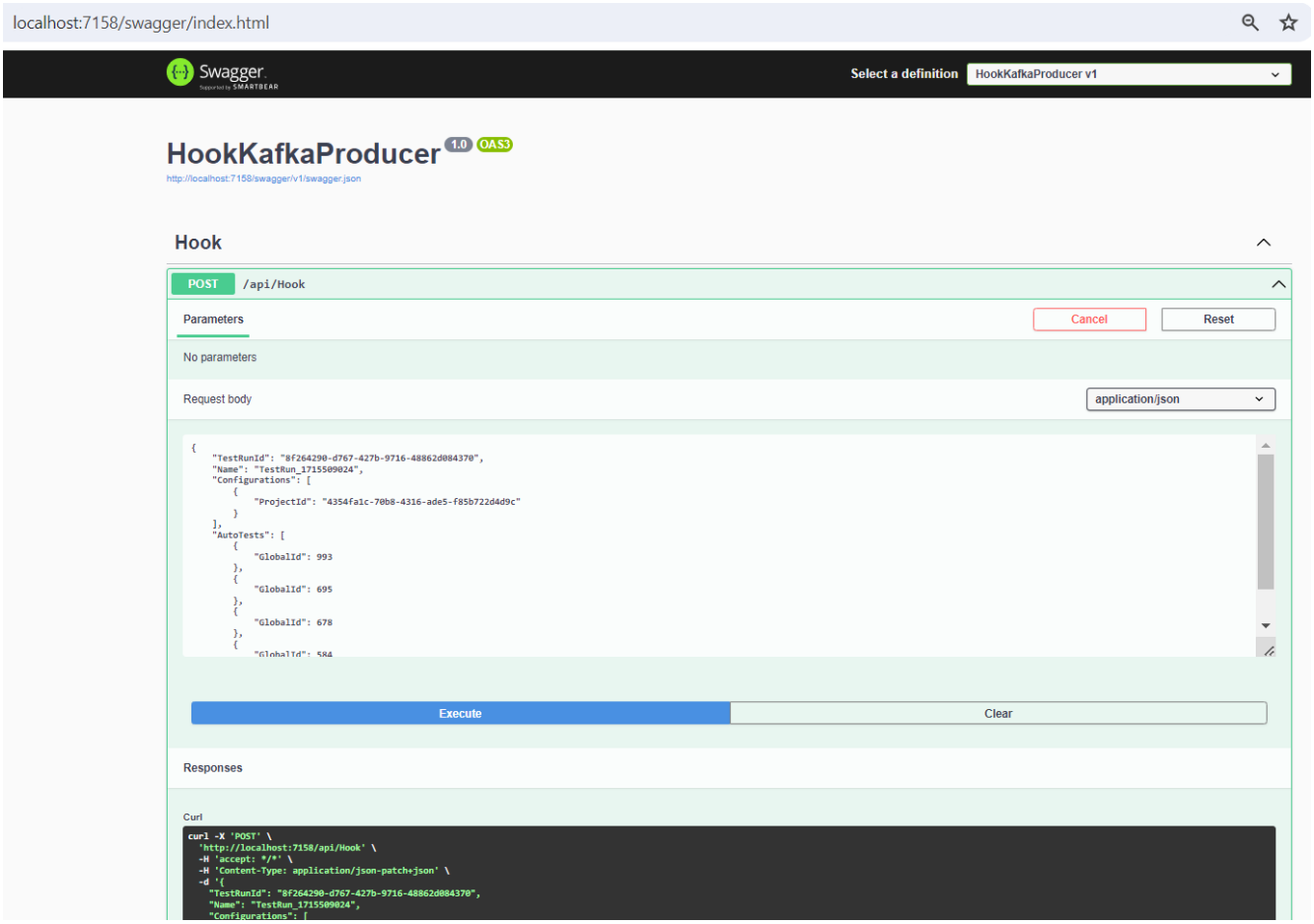



*Рисунок 16 – Применение инструмента Swagger UI для разработки и тестирования API сервиса HookKafkaProducer*

### 3.6 Общая схема компонентов разработанного фреймворка

Функциональность разработанного фреймворка состоит из следующих компонентов:

- TMS – в качестве системы управления тестирования используется Test IT;
- WebHook API перекладчик – микросервис, реализующий прием синхронных API сообщения о запуске тестов из TMS и перенаправляющее их в команду запуска в брокер сообщений;

- Apache Kafka – брокер сообщений реализующий асинхронную передачу данных между сервисом WebHook API перекладчиком и оркестратором тестовых агентов. В брокере используется единственный топик AutoTestTopic. При дальнейшем расширении функционала планируется релазовать работу с топиком StopLaunchCommand – для реализации сценария остановки тестов до их завершения;
- Сервис KafkaConsuner – микросервис приема сообщений из Kafka и организации асинхронной работы с группой тестовых агентов;
- Тестовый агент (микросервис RunnerClient) – веб-сокет клиент для запуска тестов и отправки результатов в TMS;
- Test IT Allure CLI – утилита, предоставляемая TMS TestIT для загрузки Allure отчета в систему управления тестированием;
- Сборка автотестов – целевой потребитель разработанной системы запуска автоматизированных тестов. Благодаря абстракции уровня системы запуска тестов от конечных автоматизированных тестов – на структуру проекта тестирования накладываются минимальные ограничения. Таким образом, на фреймворке можно реализовывать тестирования разных уровней. Например, с помощью подключения библиотеки Selenium можно реализовать тесты уровня GUI.
- Allure Report – основная составляющая компонента в сборке автоматизированных тестов, обеспечивающая связь между системой запуска и тестовым проектом;
- Test IT NUnit Adapted – Nuget пакет в составе тестового агента, предоставляющий вспомогательные методы по взаимодействию с TMS, например, такие как получение привязанного ручного теста к карточке автотеста.

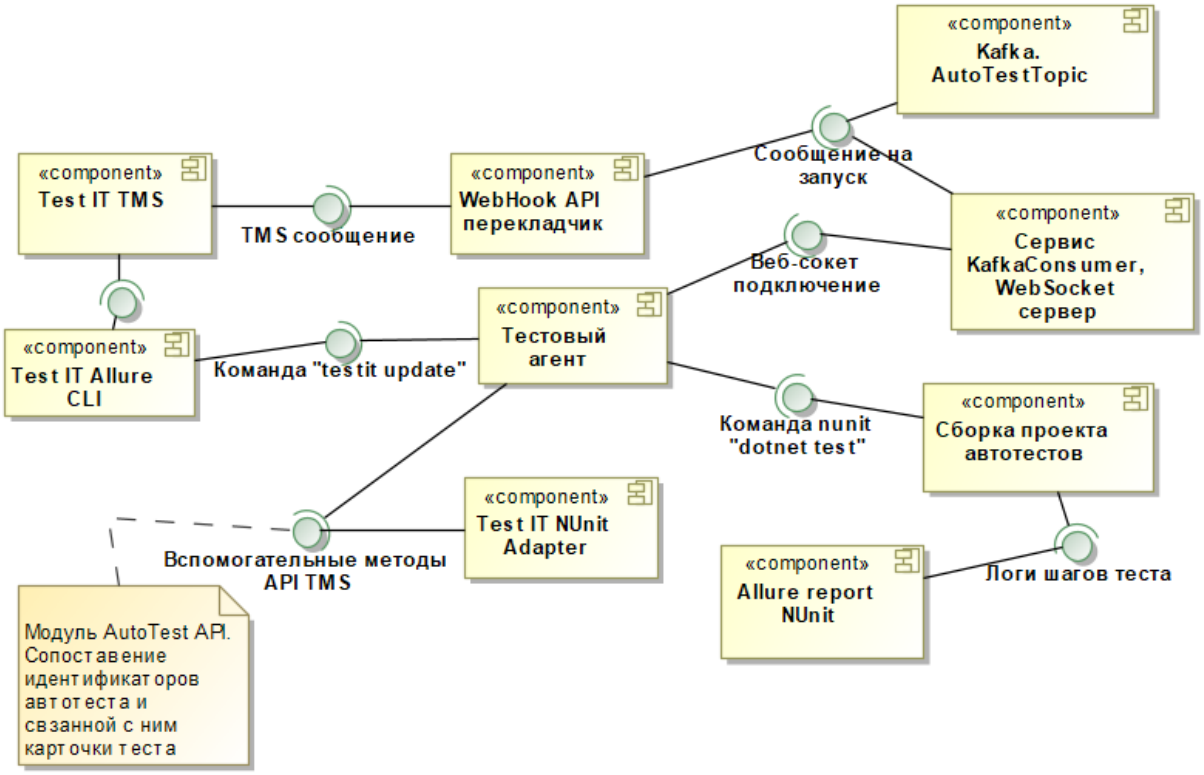



*Рисунок 17 - Диаграмма компонентов разработанного фреймворка*

### 3.7 Использование подходов DevOps при разработке

DevOps – это методология совместной работы разработчиков (Development) и операционных специалистов (Operations), нацеленная на ускорение процесса разработки и улучшения качества программного обеспечения. Она основывается на принципах автоматизации, самообслуживания, непрерывного тестирования и непрерывного развертывания (CI/CD), а также на использовании инструментов для управления ресурсами и мониторинга состояния системы. Чаще всего DevOps специалисты нужны компаниям и организациям, чья деятельность тесно связана с разработкой приложений или сервисов, а также компаниям, в ведении которых много серверов, и они, естественно, должны работать как единая система. И хотя небольшим организациям, стартапам, где работает всего 5-10-20 человек, применение DevOps может являться

нецелесообразным [16], современная разработка ПО не обходится без внедрения каких-либо практик данной методологии.

Далее будет представлено описание практик DevOps инженерии, применяемых при разработке фреймворка автоматизации тестирования.

### 3.8 Контейнеризация и оркестрация

Контейнеризация – это процесс развертывания программного обеспечения, который объединяет код приложения со всеми файлами и библиотеками, необходимыми для запуска в любой инфраструктуре. К преимуществам контейнеризации относят:

- Портативность;
- Возможность масштабирования;
- Отказоустойчивость;
- Гибкость;

Одним из самых популярных наборов инструментов, позволяющих создавать, публиковать, запускать и управлять контейнерами приложений является Docker [17].

Опишем Dockerfile для создания образа docker, содержащего публикацию приложения webhook перехватчика (HookKafkaProducer) в сборке Release. Основным компонентом образа укажем платформу dotnet:

$$FROM\ mcr.microsoft.com/dotnet/sdk:6.0\ AS\ build$$

Далее зададим директории, откуда будет произведена сборка и публикация приложения dotnet. Сборочные данные скопируем из директории проекта, в которой находится Dockerfile:

$$WORKDIR\ /app$$
$$COPY\ *.csproj\ ./$$
$$COPY\ ../$$
$$RUN\ dotnet\ publish\ -c\ Release\ -o\ out$$

Установим переменную окружающей среды dotnet [18] в значение Prod и установим точку запуска приложения:

$$COPY\ --from=build\ /app/out\ .$$
$$ENV\ ASPNETCORE_ENVIRONMENT = Prod$$
$$ENTRYPOINT\ ["dotnet","HookKafkaProducer.dll"]$$

С помощью переменных окружения регулируется актуальный конфигурационный файл приложения. В проектах разрабатываемых сервисов заданы конфигурационные файлы для Development и Prod окружений – appsettings.development.json и appsettings.Prod.json соответственно. В файлах описаны разные переменные, в частности, для подключения серверу kafka. Таким образом, при отладке приложения в среде разработки приложение будет обращаться к кластеру kafka, запущенному на локальном хосте, а при запуске приложения в контейнере – к кластеру в сети docker. Dockerfile приложений Kafka consumer и Runner Client состоят из аналогичных команд. Однако в связи с наличием зависимости проекта kafka_consumer от общей библиотеки классов CommonLibrary – данный образ был опубликован в репозитории docker hub (рис 18.).

При оркестрации запуска группы контейнеров достаточно указать собранный образ в репозитории docker hub, без привязки к зависимым проектам.

```
D:\SPBPU\dipl\TestRunnerSolution>docker tag kafka_producer kosts0/kafka_producer

D:\SPBPU\dipl\TestRunnerSolution>docker push kosts0/kafka_producer
Using default tag: latest
The push refers to repository [docker.io/kosts0/kafka_producer]
4cec1e6b8222: Pushed
7be572f42c3a: Pushed
8dafbc47cfcf: Pushed
d72a6f2a3bb9: Pushing [========================>                          ]  204.2MB/416.2MB
6408b47932e0: Pushed
9ab11312ee1a: Pushed
1de75ad3b618: Pushed
ee6f2f4e1db2: Pushing [====>                                              ]  5.975MB/70.73MB
5b4020ebdae3: Pushing [==========>                                        ]  7.783MB/36.24MB
a28e71738952: Pushing [===>                                               ]   5.38MB/80.66MB
```

*Рисунок 18 - Публикация образа kafka_producer в docker hub*

Разработанный фреймворк состоит из пяти docker образов. Docker Compose — это инструмент для определения и запуска многоконтейнерных приложений. Compose упрощает управление всем стеком приложений,

упрощая управление службами, сетями и томами в одном понятном файле конфигурации YAML [19]. Итоговый compose файл представлен в приложении 2. В данном файле, для каждого разворачиваемого сервиса описаны исходный docker образ, зависимости от других запущенных сервисов, а также необходимые публичные порты, с помощью которых можно получить доступ к контейнеру непосредственно из хоста запуска. Отдельно стоит отметить возможность масштабирования разработанной системы путем увеличения реплик сервиса RunnerClient. Так, в блоке deploy задано 3 экземпляра для запуска. После отправки тестовой команды на запуск можно видеть равномерное распределение запусков на каждый из тестовых агентов (рис. 19).

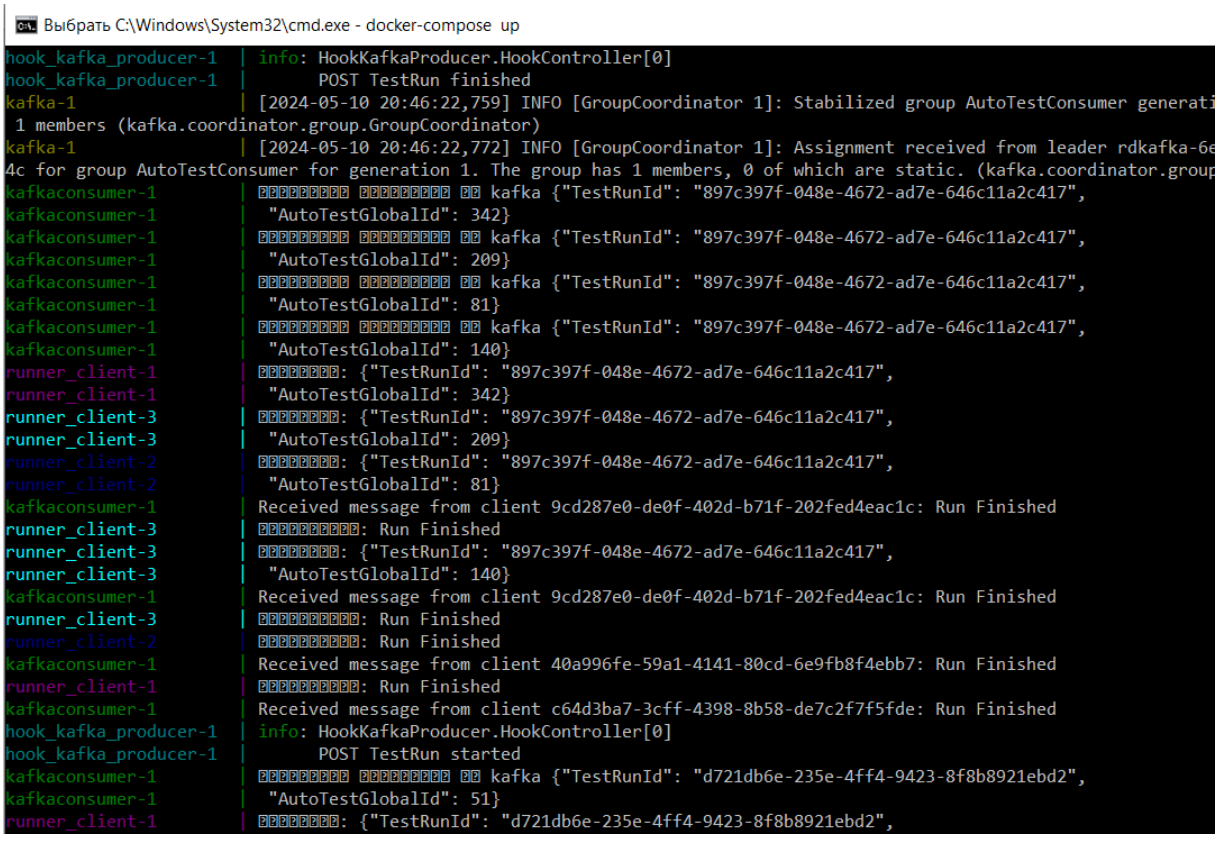



*Рисунок 19 - Работа нескольких экземпляров Runner Client*

### 3.10 Применение инструментов CI/CD

Внедрение механизмов непрерывного развертывания (CD, Continuous Deployment) позволяет автоматизированным способом отправлять код из

репозитория на удаленный сервер. В данной работе был настроен конвейер непрерывной поставки, реализующий шаги по загрузке master ветки кода репозитория автоматизированных тестов, его сборку и загрузку сборки в директорию запуска автоматизированных тестов. В качестве системы для реализации непрерывной интеграции был выбран Jenkins как система с открытым исходным кодом и масштабным сообществом поддержки.

Система Jenkins была установлена на сборочный ПК. Реализованный скрипт конвейера поставки представлен в приложении  автотестов. Код автоматизированных тестов опубликован в публичном репозитории на платформе github – https://github.com/kosts0/TestProject.

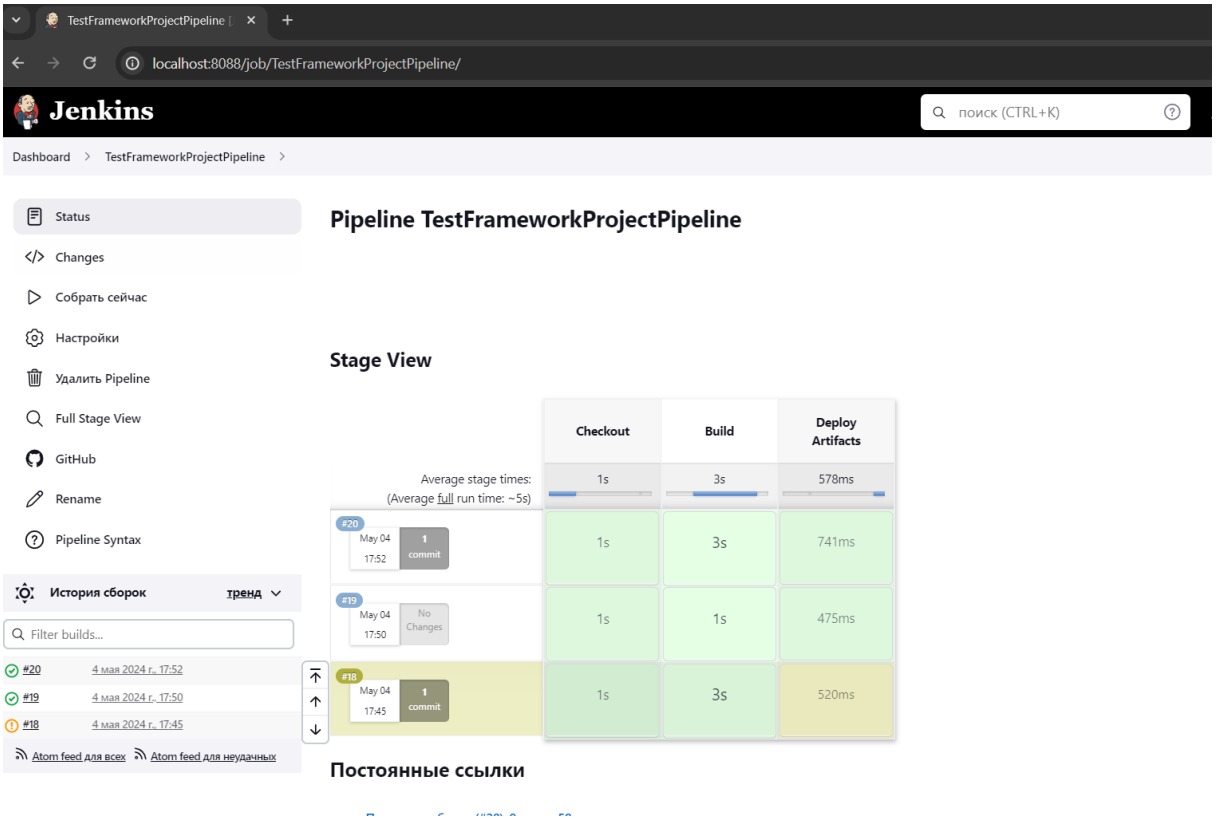



*Рисунок 20 – Страница проекта непрерывной интеграции в системе Jenkins*

Настроенный конвейер непрерывной поставки сокращает время на доставку актуальной версии автоматизированных тестов, что актуально при частых обновлениях в тестируемой системе и, как следствие, падения автотестов в связи с устареванием. Время работы скрипта непрерывной

поставки зависит от размера проекта (шаг build) и характеристик ПК сервера запуска автотестов (шаг Deploy Artifacts) – скорость записи диска, пропускная способность сети и т.д. На момент ранней разработки и отладки проекта, общее время на сборку и проекта и ее публикацию в папку запуска автоматизированных тестов составляет примерно 5 с.

# 4. Тестирование решения

## 4.1 Интеграционное тестирование, реализация полного клиентского пути

Опишем прохождения полного пути по запуску автоматизированных тестов из системы Test IT. Для проверки основной функциональности разработанного фреймворка создадим демо-проект в системе Test IT, содержащий набор ручных тестов (рис. 21).

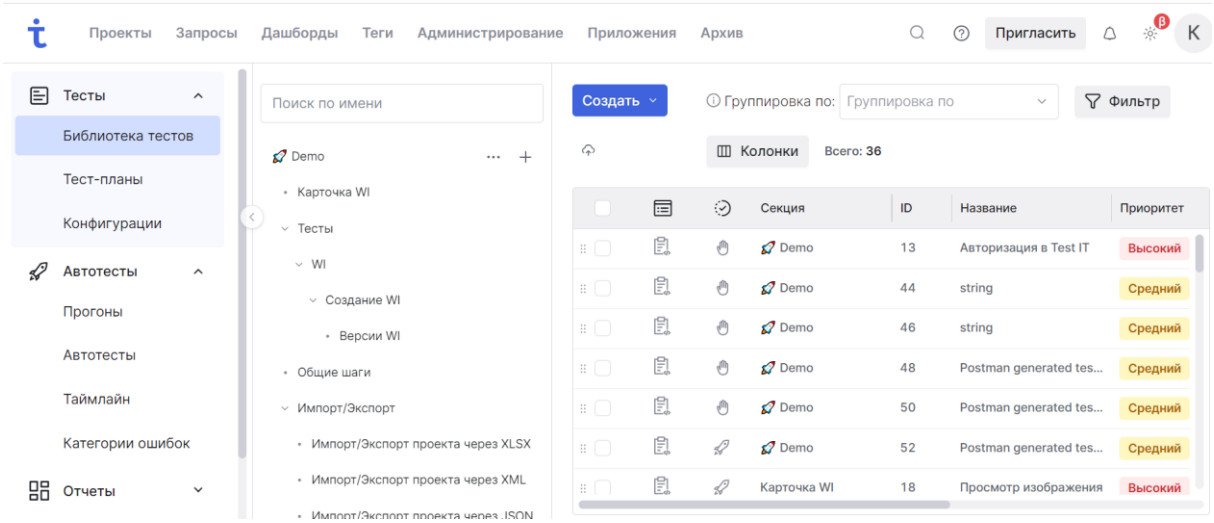
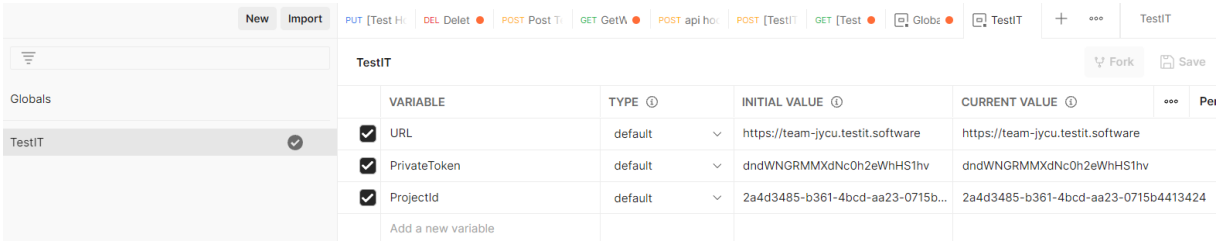


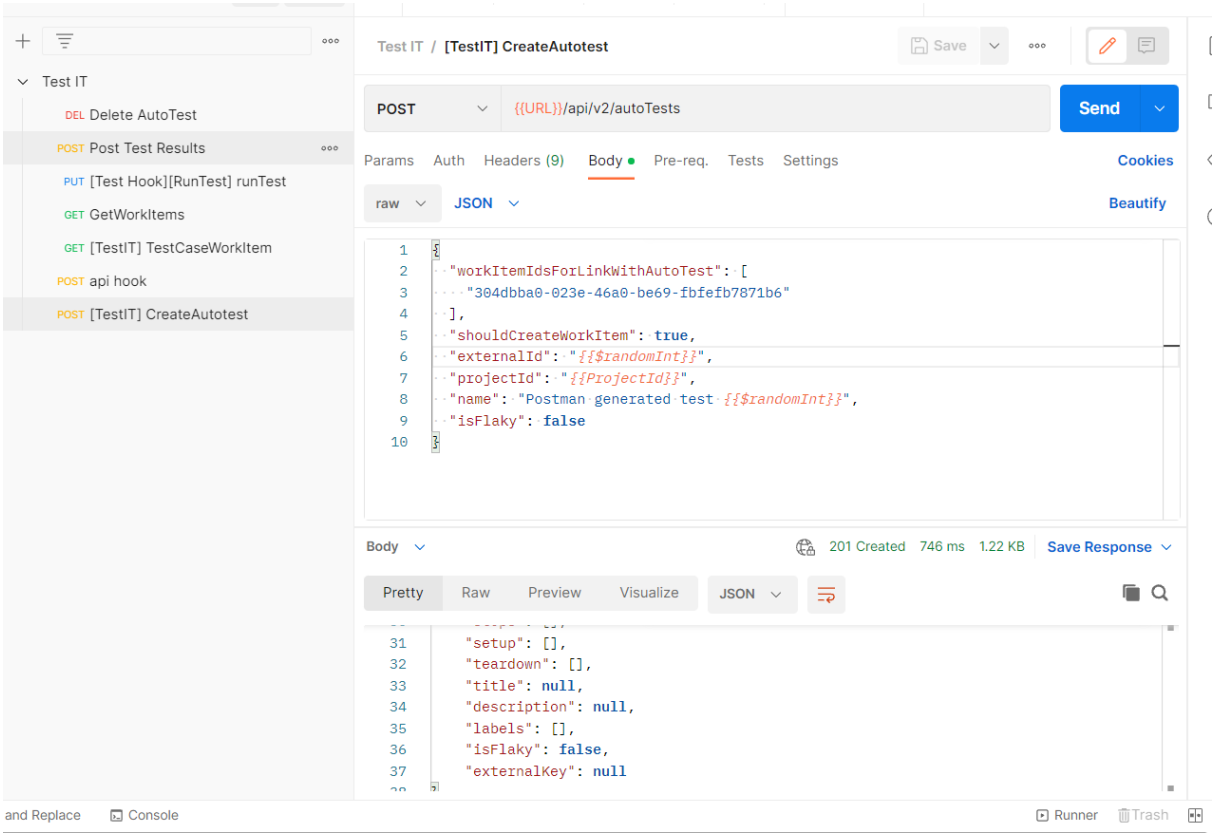

*Рисунок 21 – Раздел "Библиотека тестов" в демо-проекте Test IT*

В демонстрационном проекте отсутствуют карточки Автотестов – их необходимо создать посредством API запросов к системе. TMS Test IT предоставляет документацию по публичным API методам в системе Swagger UI. Однако ввиду большого объема справочной информации в данной системе, а также необходимости вручную подставлять

авторизационные данные при отправке запросов – основные запросы были вынесены в коллекцию утилиты Postman.




а)




б)

*Рисунок 22 – Проект тестирования взаимодействие с Test IT и фреймворком в инструменте Postman. а) Переменные окружения, используемые в запросах. б) Коллекция запросов.*

В теле запроса метода CreateAutotest был оставлен минимальный набор данных: externalId – внешний идентификатор карточки автотеста в системе, pojectId – идентификатор проекта в системе (значение для демо проекта задано в переменных окружения Postman) и отображаемое имя в карточке автотеста.

```
{
  "externalId": "{{$randomInt}}",
  "projectId": "{{ProjectId}}",
  "name": "Postman generated test {{$randomInt}}",
}
```

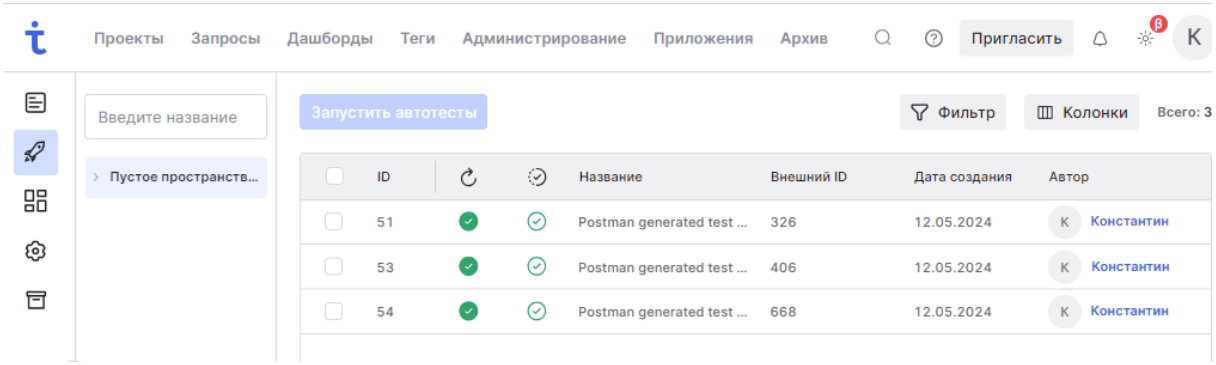



*Рисунок 23 – Карточки автотестов, созданные с помощью Postman запросов*

Привяжем карточку автоматизированного теста к карточке ручного теста.

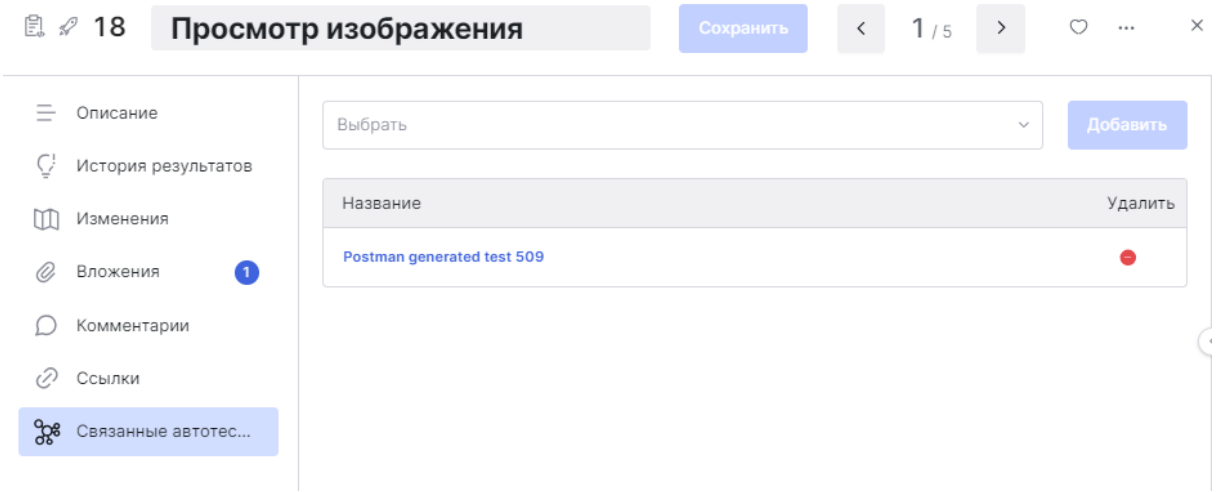



*Рисунок 24 – Привязка автотеста к ручному тесту в системе Test IT*

В настройках вебхук укажем адрес для проецирования команды запуска автотестов на разработанный фреймворк.

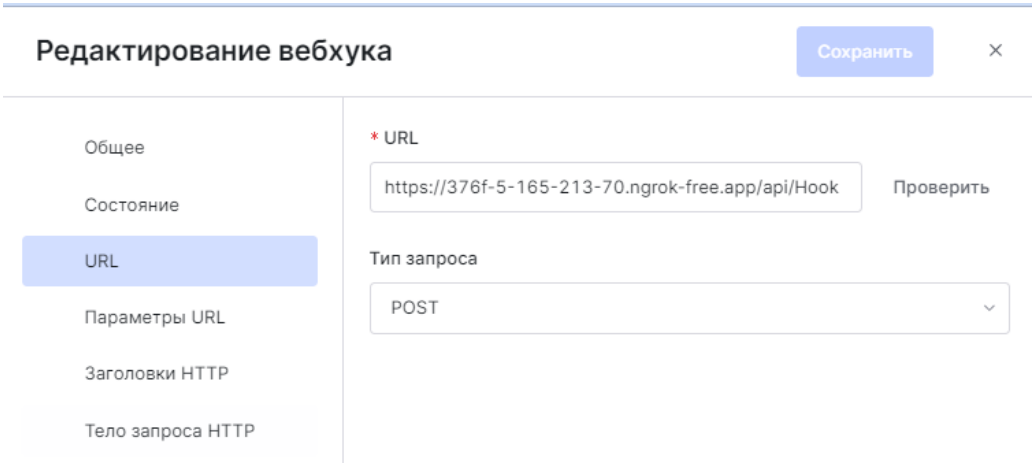



*Рисунок 25 – Вебхук команды запуска автотестов в системе TestIT*

В рамках локальной разработки и отладки для доступа внешней системы TestIT к локальному хосту разработки на хосте была запущена утилита ngrok, публикующая локальный порт в общедоступной сети.

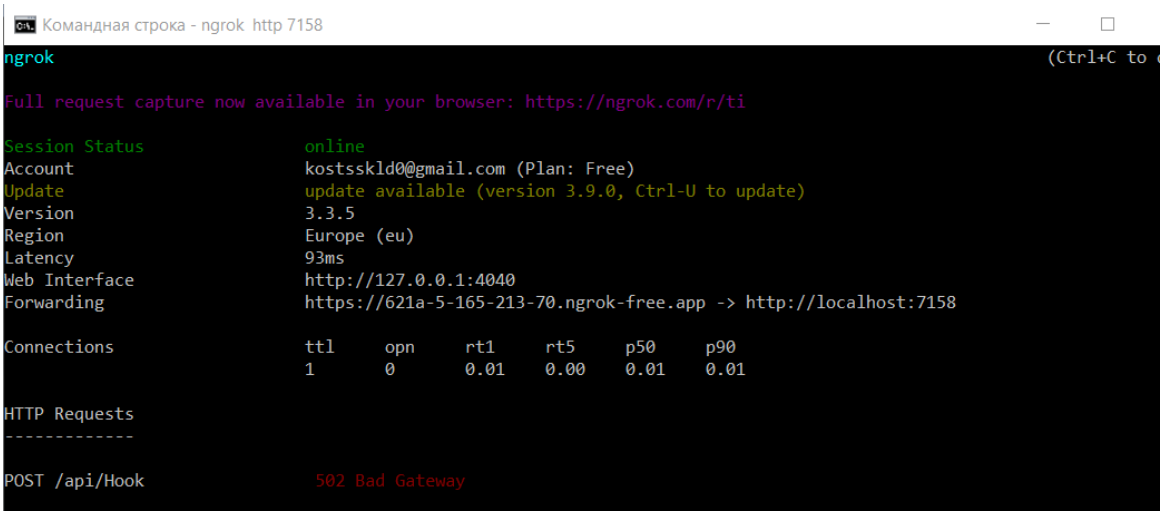
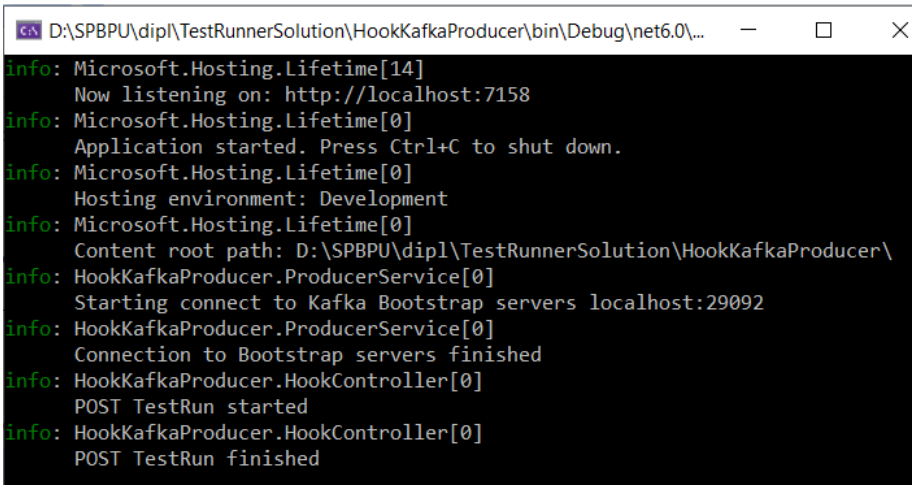


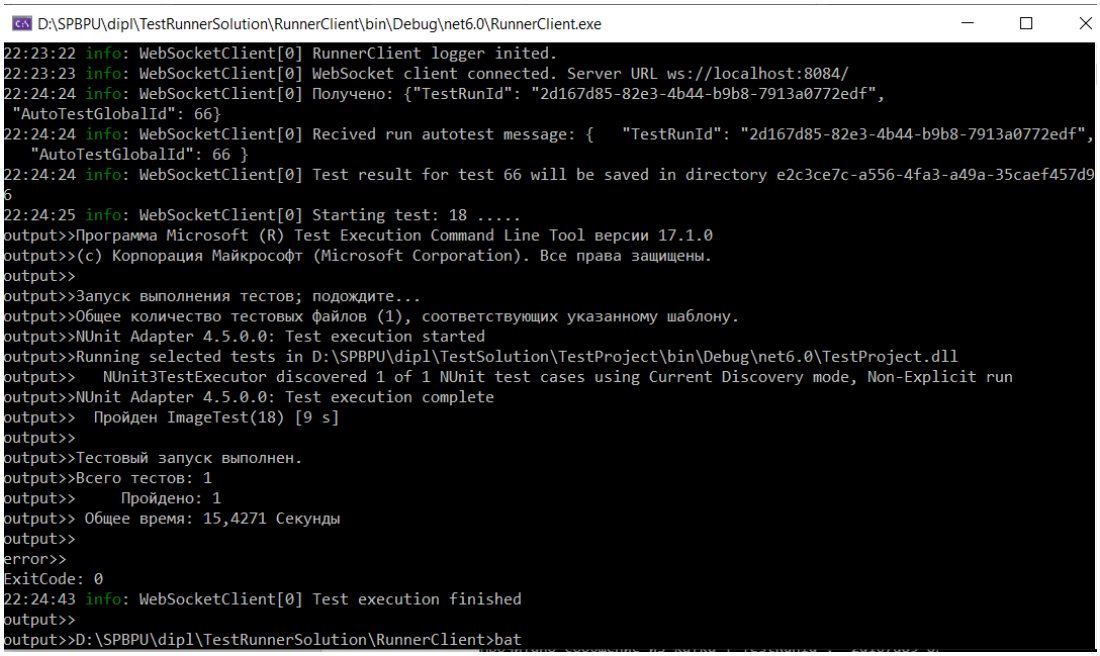

*Рисунок 26 – Утилита ngrok*

Выберем привязанные к ручным тестам автотесты и запустим их. По логам системы можем увидеть происходящие в сервисах активности. Так, в логе сервиса HookKafkaProducer (рис. 27, а) можно увидеть, что 2 команды на запуск были отправлены в очередь. Сервисом KafkaConsumer было получено 2 сообщения из брокера, а после исполнения тестов от агента были получены сообщения о завершении исполнения (рис. 27, б).




а)


б)




в)

*Рисунок 27 – Логи сервисов при запуске прогона. а) Сервиса HookKafkaProducer. б) Сервиса KafkaConusmer. в) Сервиса RunnerClient.*

Логи тестового агента (рис. 27, в) показывают, что клиент был подключен к серверу, получил команду на запуск теста под идентификатором “18” и передал серверу команду о завершении исполнения. В рамка отладки в сервисе тестового агента логируется вывод среды NUnit, в которой содержится общая информация о результатах выполнения теста.

На рисунке 28 представлена страница с прогонами, запущенными непосредственно из системы. Прогон автоматически переходит в статус “Завершен”, когда в него будут загружены результаты по всем тестам.

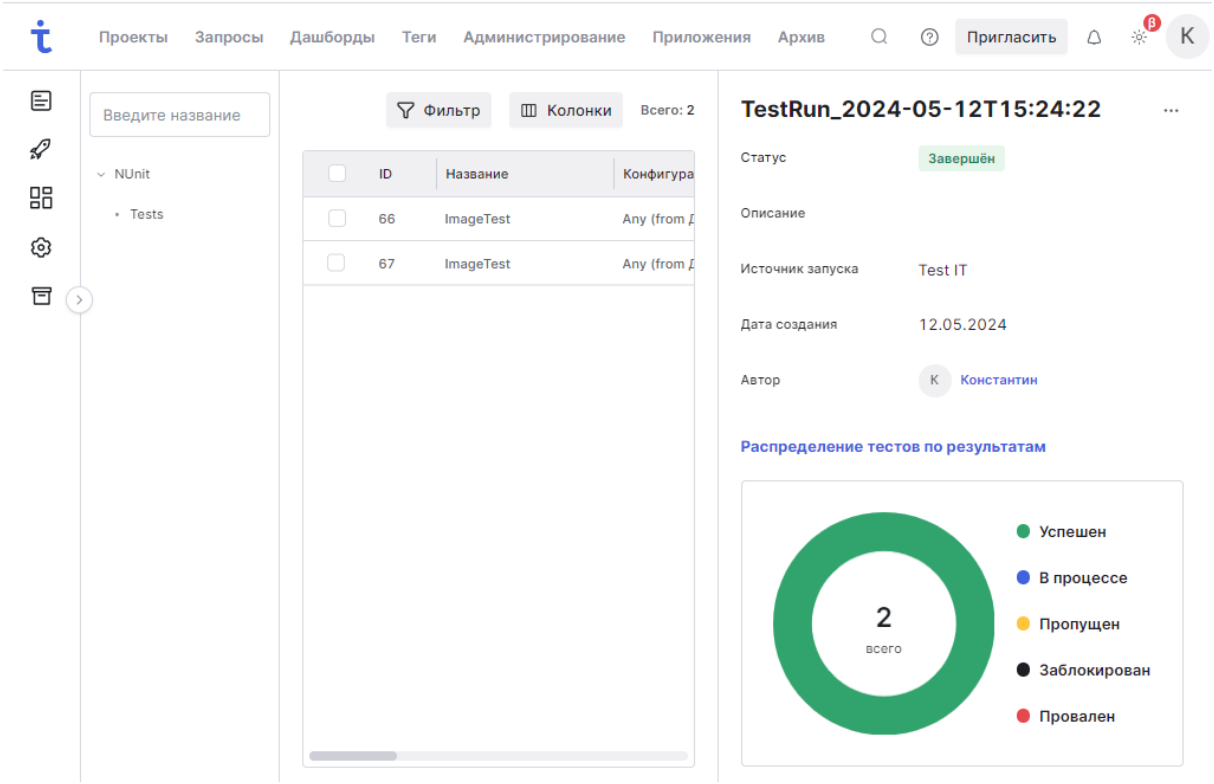



*Рисунок 28 – Страница прогона тестов*

На рисунке 29 представлен пример отчёта автотеста, завершенного с ошибкой. Отчет с формирован в соответствии с содержимым файла Allure report container, за исключением загрузки вложения.

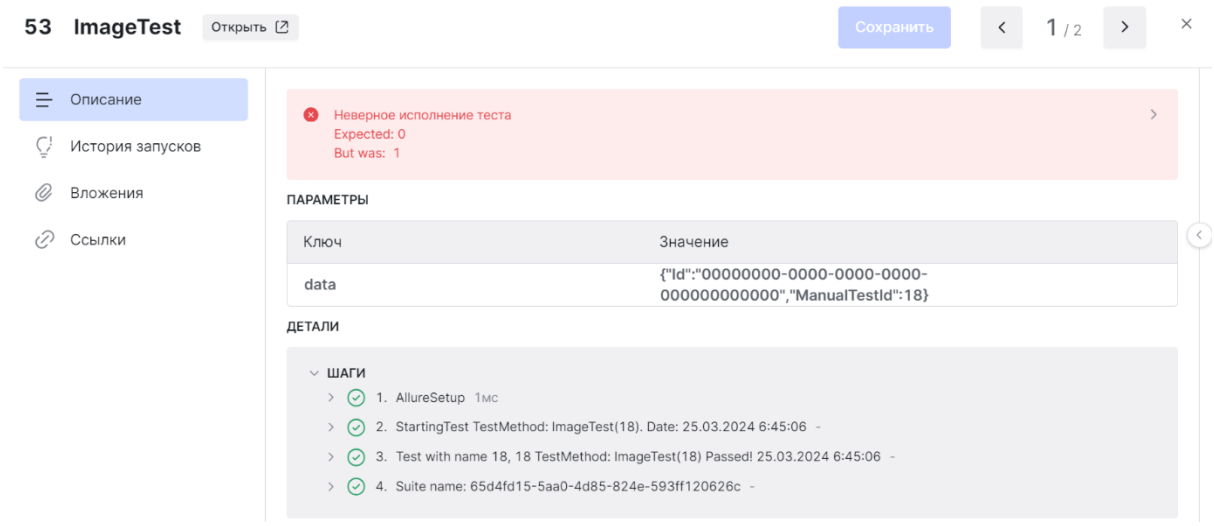



*Рисунок 29 – Результат автотеста в TMS*

Таким образом, успешно был пройден полный путь запуска автотестов “по требованию” – непосредственно из TMS.

# 5. Результаты работы

## 5.1 Сравнение процессов тестирования

Принцип работы фреймворка был продемонстрирован в вышестоящих разделах. Целью работы фреймворка является автоматизация части процессов в процессной модели тестирования (рис. 1). Проведем сравнительный анализ процессов тестирования с использованием фреймворка, без него, и без автоматизированного тестирования в принципе (табл.4).

*Таблица 4 – Сравнение подходов к тестированию*

| **Критерии сравнения** | **Автоматизация тестирование с использованием TMS** | **Автоматизация тестирования локальными инструментами** | **Ручное тестирование** |
|---|---|---|---|
| Человеческий фактор | Невозможно полностью исключить. Минимизируется за счет автоматизации доставки кода автотестов и четкой структурой | Зависит от компетенций тестировщика и качества тестовых скриптов. | Наиболее подвержено влиянию человеческого фактора. |

| | | | |
|---|---|---|---|
| | организации данных в фреймворке | | |
| Затраты на поддержку | Необходима поддержка как самих тестов, так и контрактов интеграции с TMS. | Зависит от обновлений системы. | Не нуждается в поддержке. |
| Прозрачность отчетности | Процессы тестирования – публичные. Любой член команды разработки может видеть оперативную отчетность в TMS по автоматизированному тестированию и документации тестирования в целом. | При отсутствии специализированных инструментов – запрос отчетности по требованию. Необходимо время на сбор отчетности, процесс не автоматизирован. | Процесс не автоматизирован, необходимо время на составление отчетности и предоставления тестовых артефактов. |
| Компетенция сотрудников | Для работы на фреймворке требуются навыки программирования. Отдельно требуется команда для поддержки работоспособности интеграции TMS с системой запусков. | Требуются навыки программирования. | Специализированных навыков не требуется. |
| Универсальность | Функциональность TMS рассчитана на комплексное обеспечение процессов тестирования. В работе с TMS учувствуют разные члены команд: менеджеры, администраторы, разработчики, тестировщики. | Решает задачи регрессионного тестирования конкретной команды разработки. | Отдельные задачи для тестировщика, предоставляемые менеджером команды. |

В сравнении видно, что использование TMS и автоматизации тестирования требуют больших вложений для обеспечения должного результата. Поддержкой процессов автоматизации, как правило, занимаются отдельные команды в компании. Эффект от автоматизации кратно увеличивается в совокупности с использованием TMS. Когда тестировщики пишут код автотестов в едином пространстве и по единым технологиям – повышается уровень стандартизации данных процессов в компании. Фреймворк автоматизации тестирования помимо прочего

является важным инструментом для достижения стандартизации процессов тестирования.

## Заключение

Результатом проделанной работы является программный продукт – фреймворк автоматизации тестирования на платформе dotnet. Интеграция автоматизированного тестирования в TMS систему позволяет предоставлять компаниям тестирования как услугу по обеспечению бизнес-процессов разработки. В условиях современной конвейерной разработки такой вид автоматизации позволит кратно сократить время на проведение регрессионного и функционального тестирования в компании, а следовательно, увеличить долю исследовательского тестирования у специалистов обеспечения качества.

Повторно рассмотрев процессную модель тестирования (рис. 1)

Дистрибутивом для поставки разработанного решения является его Docker Compose спецификация – средство для определения и запуска приложений Docker с несколькими контейнерами. Следовательно, зависимости на размещение системы запуска сведены к минимуму.

Фреймворк имеет ряд ограничений, выраженных в правилах по написанию автоматизированных тестов – обязательная реализация в среде NUnit и атрибутивная нотация данных у тестовых методов, что позволяет запускать тест как процесс из внешних агентов.

Дальнейшее развитие фреймворка может быть выражено в реализации сценариев остановки теста из TMS, что позволит повысить удобность его использования. Также основным направлением развития является выстраивание цепочек заданных тестов в последовательность уникальных запусков, что может быть полезно, например, когда в сценарии используется общие пользовательские данные.

## Список использованных источников


1. В. Ф. Корнюшко, А. В. Костров, П. А. Породникова. Подход к развитию системы управления тестированием программных средств // Программные продукты и системы / Software & Systems – выпуск № 4 (112) – 2015 г. – С. 126 – 132.
2. Как отчёт по тестированию помогает лучше понять качество ИТ-решения // Точка качества, блок компании: сайт. – 2023 г. – URL: https://tquality.ru/blog/Kak-otchyot-po-testirovaniyu-pomogaet-luchshe-ponyat-kachestvo-IT-resheniya.
3. А.А. Тютюных, И.С. Полевщиков. Разработка автоматизированной системы управления процессом тестирования программного обеспечения // АВТОМАТИЗИРОВАННЫЕ СИСТЕМЫ УПРАВЛЕНИЯ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ. Материалы всероссийской научно-технической конферен ции: в 2х томах. 2018. С. 104-109.
4. Даниил Замешаев. Обзорная статья о видах тестирования IT-продуктов. // Хабр. – 2023. – URL: https://habr.com/ru/companies/otus/articles/720664/ (дата обращения: 30.03.2024).
5. Вячеслав Зимин. СИСТЕМЫ УПРАВЛЕНИЯ ТЕСТИРОВАНИЕМ. // Школа седого тестировщика. – 2019. URL: https://sedtest-school.ru/testovaya-dokumentacziya/sistemy-upravleniya-testirovaniem/ (дата обращения: 30.03.2024).
6. Дмитрий Аверьянов. Все яйца — в одну корзину. Как мы интегрировали автотесты с TestIt // Хабр. – 2023. – URL: https://habr.com/ru/companies/T1Holding/articles/749324/ (дата обращения: 30.03.2024).
7. Даниил Мельников. Автотесты и TMS: как мы реализовали интеграцию АФТ с Test IT // Хабр. – 2023. – URL:

https://habr.com/ru/companies/domrf/articles/720598/ (дата обращения: 30.03.2024).

8. Работа с системами автоматизированного тестирования. // Test IT. Портал документации: сайт. – 2023 – URL: https://docs.testit.software/user-guide/integrations/.
9. Импорт результатов из Allure Adapters. // Test IT. Портал документации: сайт. – 2023 – URL: https://docs.testit.software/user-guide/autotests/import-results-from-allure.
10. Test IT CLI. // Test IT. Портал документации: сайт. – 2023 – URL: https://docs.testit.software/user-guide/integrations/cli.
11. Амосов В. В., Петров А. В. Система поддержки принятия решения на основе качественного подхода. // Перспективы науки – выпуск № 1(172) – 2024 г. – С. 90 - 94.
12. Qa_meister. Топ-12 лучших систем управления тестированием 2021. // Software Testing. – 2021 г. URL: https://software-testing.ru/library/testing/test-management/3696-top-10-best-test-management-systems-2021 (дата обращения: 30.03.2024).
13. Allure TestOPS, TestIT, TestRail: сравнение систем в разрезе автоматизации. // Статьи ВКонтакте: сайт. – 2023. – URL: https://m.vk.com/@simbirsoft_team-allure-testops-testit-testrail-sravnenie-sistem-v-razreze-av
14. Jess Charlton. 10 Best Open Source Test Management Tools in 2023. // QA Lead – 2023 г. URL: https://theqalead.com/tools/best-open-source-test-management-tools/ (дата обращения: 30.03.2024).
15. Анна Вичугова. Как начать моделировать бизнес-процессы в BPMN. // Школа системного анализа и проектирования System Education – 2023 г. URL: https://systems.education/bpmn-start (дата обращения: 30.03.2024).

16. Арина Петрова. Охота на DevOps-инженеров: в чем их сила и почему за ними будущее ИТ. // Хайтек+: сайт – 2021. – URL: https://hightech.plus/2021/10/08/ohota-na-devops-inzhenerov-v-chem-ih-sila-i-pochemu-za-nimi-budushee-it (дата обращения: 09.05.2024)
17. Docker и Kubernetes — чем отличаются технологии контейнеризации //Etherhost – Хостинг с системой защиты от DDoS-атак – 2021. – URL: https://eternalhost.net/blog/razrabotka/docker-kubernetes#p1 (дата обращения: 09.05.2024).
18. Tom Dykstra, Wade Pickett. Use multiple environments in ASP.NET Core // Microsoft Learn – 2024. – URL: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/environments?view=aspnetcore-6.0 (дата обращения 09.05.2024).
19. Docker Compose overview. – docker docs – 2024. – URL: https://docs.docker.com/compose/ (дата обращения 11.05.2024).
20. Кочер П. С. Микросервисы и контейнеры Docker / пер. с англ. А. Н. Киселева. - М.: ДМК Пресс, 2019. - 240 с.: ил.
21. Тестирование. Введение в юнит тесты – Metanit.com, сайт о программировании – 2019. – URL: https://metanit.com/sharp/aspnet5/22.1.php (дата обращения: 11.05.2024).
22. Илья Бубнов. Лучшие IDE для разработки на C#. //GeeckBrains ИТ–образование. – 2018. – URL: https://gb.ru/posts/c_sharp_ides (дата обращения: 11.05.2024);
23. 10 ЛУЧШИХ IDE И РЕДАКТОРОВ КОДА. – Бортовой журнал Space Web. – 2022. – URL: https://journal.sweb.ru/article/10-luchshih-ide-i-redaktorov-koda (дата обращения: 11.05.2024).
24. Билл Вагнер. Краткий обзор языка C#. //Microsoft Learn .NET. – 2022. – URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/tour-of-csharp/ (дата обращения 11.05.2024);

25. James Newton-King. NuGet. //Microsoft Learn .NET – 2022. – URL: https://learn.microsoft.com/ru-ru/dotnet/standard/library-guidance/nuget (дата обращения 02.01.2023).
26. Якунович, А. В. Брокеры сообщений / А. В. Якунович, А. С. Север, Ю. А. Чистякова // Информационные технологии. Физика и математика: материалы 87-й научно-технической конференции профессорско-преподавательского состава, научных сотрудников и аспирантов, Минск, 31 января - 17 февраля 2023 г. - Минск : БГТУ, 2023. – С. 79-82.
27. Импорт результатов из Allure Adapters – TestIT портал документации. – 2024. URL: https://docs.testit.software/user-guide/autotests/import-results-from-allure.html (дата обращения: 12.05.2024).
28. Manan Patadiya. Creating Microservices with .NET Core and Kafka: A Step-by-Step Approach. – Текст: электронный // Medium. – 2023. – № 1. URL: https://medium.com/simform-engineering/creating-microservices-with-net-core-and-kafka-a-step-by-step-approach-1737410ba76a (дата обращения 24.03.2024).

## Приложение 1 – Скрипт Jenkins pipeline для непрерывной доставки сборки автотестов

```
pipeline {
    agent any
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/kosts0/TestProject.git'
            }
        }
        stage('Build') {
            steps {
                bat 'dotnet build -c Release'
            }
```

```
        }
        stage('Deploy Artifacts') {
            steps {
                catchError(buildResult: 'UNSTABLE', stageResult: 'UNSTABLE') {
                    script {
                        def buildFolder = "D:\\SPBPU\\dipl\\JenkinsBuildFolder"
                        def sourceFolder = "bin\\Release"
                        bat "xcopy /E /Y ${sourceFolder} ${buildFolder}"
                    }
                }
            }
        }
    }
}
```

## Приложение 2 – docker-compose файл

```
version: "3.0"
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.4
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 22181:2181
    networks:
      - my-network

  kafka:
    image: confluentinc/cp-kafka:7.4.4
    depends_on:
      - zookeeper
    ports:
      - 29092:29092
    networks:
      - my-network
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
  hook_kafka_producer:
```

```
    build: ./HookKafkaProducer
    image: hook_kafka_producer
    depends_on:
      - kafka
    ports:
      - 7158:7158
    networks:
      - my-network
  kafkaconsumer:
    image: kosts0/kafka_consumer
    depends_on:
      - kafka
    ports:
      - 8084:8084
    networks:
      - my-network
  runner_client:
    build: ./RunnerClient
    image: runner_client
    depends_on:
      - kafkaconsumer
    networks:
      - my-network
    deploy:
      mode: replicated
      replicas: 3
networks:
  my-network:
```

## Приложение 3 – Листинг класса PrepareOutputDirectory

```
[SetUpFixture]
class PrepareOutputDirectoryFixture
{
    string? path;

    [OneTimeSetUp]
    public void SetOutputDirectory()
    {
        Trace.Listeners.Add(new ConsoleTraceListener());
        this.path = Path.GetTempFileName();
        var directory = Directory.CreateDirectory(
            Path.Combine(GetCurrentAllureDirectory(), CreateSubfolderName()));
        File.WriteAllText(this.path,
            JsonSerializer.Serialize( new { allure = new { directory =
directory.FullName } }),
            Encoding.UTF8);

Environment.SetEnvironmentVariable(AllureConstants.ALLURE_CONFIG_ENV_VARIABLE,
this.path);
    }
    [OneTimeTearDown]
```

```
    public void DeleteTempConfigFile()
    {

Environment.SetEnvironmentVariable(AllureConstants.ALLURE_CONFIG_ENV_VARIABLE,
null);
        if (this.path is not null && File.Exists(this.path))
        {
            File.Delete(this.path);
        }
        Trace.Flush();
    }
    static string GetCurrentAllureDirectory() => "allure-results";
    static string CreateSubfolderName() =>
string.Concat(TestContext.Parameters.Get("TestRunId") ?? "DebugTestRun");
      }
```

# Приложение 4 – Листинг класса генератора тестовых моделей FileInputParamsBase

```
public class FileInputParamsBase<TTestCaseParams> : IEnumerable
    {
        public FileInputParamsBase(string fileName)
        {
            FileName = fileName;
        }
        protected virtual string DataFilesSubFolder =>
Directory.GetCurrentDirectory();
        protected virtual string FileName { get; set; }
        private string GetFilePath()
        {
            var directoryName =
Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
            if (directoryName == null)
                throw new Exception("Couldn't get assembly directory");

            return Path.Combine(directoryName, DataFilesSubFolder,
$"{FileName}");
        }
        public IEnumerator GetEnumerator()
        {
            var testFixtureParams =
JsonConvert.DeserializeObject<JObject>(File.ReadAllText(GetFilePath()));
            var genericItems =
testFixtureParams[$"{typeof(TTestCaseParams).Name}"].ToObject<IEnumerable<TTestCa
seParams>>();

            foreach (var item in genericItems)
            {
                yield return new object[] { item };
            }
        }
}
```