Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Хакасский технический институт – филиал ФГАОУ ВО
«Сибирский федеральный университет»

Кафедра прикладной информатики, математики и естественно-научных дисциплин

УТВЕРЖДАЮ
Заведующий кафедрой
__________ Е. Н. Скуратенко
подпись
«_____» _________ 2021 г.

БАКАЛАВРСКАЯ РАБОТА

09.03.03 Прикладная информатика

Разработка системы отслеживания ошибок в программных продуктах

Руководитель _______________ доцент, канд. пед. наук И. В. Янченко
подпись, дата

Выпускник _______________ С. А. Яцутко
подпись, дата

Консультанты
по разделам:

Экономический ___________________ Е. Н. Скуратенко
подпись, дата

Нормоконтролер ___________________ В. И. Кокова
подпись, дата

Абакан 2021

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Хакасский технический институт – филиал ФГАОУ ВО «Сибирский федеральный университет»

Кафедра прикладной информатики, математики и естественно-научных дисциплин

УТВЕРЖДАЮ
Заведующий кафедрой
__________ Е.Н. Скуратенко
подпись
«___» _________ 2021 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
в форме бакалаврской работы

Студенту Яцутко Сергею Анатольевичу

Группа ХБ 17-03

Направление 09.03.03 Прикладная информатика

Тема выпускной квалификационной работы «Разработка системы отслеживания ошибок в программных продуктах»

Утверждена приказом директора № 222 от 08.04.2021 г.

Руководитель ВКР: И.В. Янченко, доцент кафедры ПИМиЕД, канд. пед. наук, ХТИ – филиал СФУ

Исходные данные для ВКР: общие требования заказчика к информационной системе отслеживания ошибок в программных продуктах.

Перечень разделов ВКР:

1. Аналитический раздел: «Анализ предметной области».
2. Проектный раздел: «Описание разработки системы отслеживания ошибок в программных продуктах».
3. Экономический раздел: «Оценка экономической эффективности разработки и внедрения системы отслеживания ошибок в программных продуктах».

Перечень графического материала: нет.

Руководитель ВКР ____________________ И. В. Янченко
подпись

Задание принял к исполнению ____________________ С. А. Яцутко
подпись

08 апреля 2021 г.

РЕФЕРАТ

Выпускная квалификационная работа по теме «Разработка системы отслеживания ошибок в программных продуктах» содержит 111 страниц текстового документа, 10 таблиц, 23 рисунка, 16 использованных источников.

JAVA, JAVASCRIPT, HTML, СЕРВЕР, ВЕБ-КЛИЕНТ, POSTGRESQL, SPRING FRAMEWORK, SPRING SECURITY, REACT, REDUX, АВТОРИЗАЦИЯ, МНОГОПОТОЧНОСТЬ, IDEF0, IDEF3, DFD, КЛИЕНТ-СЕРВЕР, КАПИТАЛЬНЫЕ ЗАТРАТЫ, ТСО, СТОИМОСТЬ, РИСКИ.

Объект выпускной квалификационной работы: процесс выполнения лабораторных работ по дисциплине «Тестирование и контроль качества информационных систем» в ХТИ – филиале СФУ.

Целью выпускной квалификационной работы является: разработка системы отслеживания ошибок в программных продуктах.

В первом разделе выпускной квалификационной работы описывается основная деятельность ХТИ – филиала СФУ. Проведен анализ процесса тестирования программных продуктов на наличие ошибок, построены диаграммы в нотации IDEF0. Принято решение о выборе клиент-серверной архитектуры. Описываются модели информационной системы с помощью диаграмм в нотациях IDEF0, IDEF3, DFD.

Во втором разделе описывается процесс создания системы отслеживания ошибок в программных продуктах, который сопровождается скриншотами кода программы и полученного интерфейса. В третьем разделе выполнена оценка экономической эффективности проекта, выполнен анализ рисков, которые могут возникнуть в процессе разработки и эксплуатации веб-приложения, а также представлены возможные мероприятия по их снижению. Проведен анализ рынка продуктов аналогов. Разработанная система может быть использована образовательных целях вузов на дисциплине тестирование и контроль качества информационных систем.

SUMMARY

The theme of the graduation thesis is «Bug-Tracking System Development for Software Products». It contains 111 pages of a text document, 10 tables, 23 figures, 16 reference items.

JAVA, JAVASCRIPT, HTML, SERVER, WEB-CLIENT, POSTGRESQL, SPRING FRAMEWORK, SPRING SECURITY, REACT, REDUX, AUTHORIZATION, MULTI-THREADING, IDEF0, IDEF3, DFD, CLIENT-SERVER, CAPITAL COSTS, TCO, COSTS, RISKS.

The object of the graduation paper is the process of performing laboratory classes in the course of "Testing and Quality Control of IT Systems" at KhTI - a branch of the SibFU.

The purpose of the graduation thesis: to develop a system for bug tracking in software products.

The first section of the thesis describes the workflow of KhTI - a branch of the SibFU. The analysis of the process of testing of software products for errors has been carried out, and the diagrams have been built in IDEF0 notation. It has been decided to choose client-server architecture. Models of the information system have been described using diagrams in the notations of IDEF0, IDEF3, DFD.

The second section describes the process of creating of a bug tracking system in software products accompanied by screenshots of the program code and the resulting interface. The third section deals with the assessment of economic efficiency of the project, analysis of risks that may arise during the development and operation of a web application, and possible measures to reduce them. The analysis of the market of analogue products has been carried out. The developed system can be used for educational purposes at universities in the course of "Testing and Quality Control of IT Systems".

English language supervisor: ________________ N.V. Chezybaeva

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 8
1 Анализ предметной области 9
1.1 Описание основной деятельности ХТИ – филиала СФУ 9
1.2 Предпосылки для разработки системы отслеживания ошибок в программных продуктах 10
1.3 Анализ бизнес-процесса тестирования информационных систем 11
1.4 Атрибуты ошибок, указываемые в баг-трекинговых системах 17
1.4.1 Локализации и серьезность 17
1.4.2 Атрибуты жизненного цикла бага: статус и резолюция 19
1.4.3 Отчет о дефекте (баг-репорт) 21
1.5 Обоснование выбора архитектуры и средств разработки системы 25
1.5.1 Выбор архитектуры приложения 25
1.5.2 Выбор языков программирования 27
1.5.3 Выбор библиотек и фреймворков 29
1.5.4 Выбор постоянного хранилища данных 31
1.5.5 Выбор системы контроля версий 31
1.5.6 Выбор сервера CI/CD 32
1.6 Схема взаимодействия пользователя с системой 33
1.7 Выводы по разделу «Анализ предметной области» 34
2Описание разработки системы отслеживания ошибок в программных продуктах 35
2.1 Жизненный цикл разработки системы отслеживания ошибок в программных продуктах 35
2.2 Проектирование базы данных 36
2.3 Настройка репозитория GitHub 37
2.4 Настройка сервера непрерывной интеграции 42
2.5 Разработка безопасного механизма авторизации пользователей 45
2.5.1 Описание процесса аутентификации 45
2.5.2 Реализация процесса аутентификации 46

2.5.3 Генерация и валидация токена 47
2.5.4 Валидация запросов 49
2.5.5 Настройка доступа к конечным точкам приложения 50
2.5.6 Разработка конечной точки аутентификации 52
2.6 Конфигурация сервера 55
2.7 Определение моделей данных 60
2.8 Разработка контроллеров 67
2.9 Разработка клиентской части ИС 70
2.10 Выводы по разделу «Описание разработки системы отслеживания ошибок в программных продуктах» 94
3Оценка экономической эффективности разработки и внедрения системы отслеживания ошибок в программных продуктах 95
3.1 Капитальные затраты 95
3.2 Затраты на проектирование ИС 96
3.3 Эксплуатационные затраты 100
3.2.1 Расчет затрат реализации проекта методом Total Cost of Ownership 103
3.4 Анализ рынка продуктов-аналогов. Установление стоимости программного продукта 104
3.5 Экономическая эффективность реализации системы отслеживания ошибок в программных продуктах 107
3.6 Выводы по разделу «Оценка экономической эффективности разработки и внедрения системы отслеживания ошибок в программных продуктах» 107
ЗАКЛЮЧЕНИЕ 109
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 110

ВВЕДЕНИЕ

Для получения определенных навыков и знаний в тестировании программного обеспечения, в учебную программу направления «Прикладная информатика» ХТИ – филиала СФУ включен такой предмет, как «Тестирование и контроль качества информационных систем». На данный момент, в процессе преподавания данной дисциплины используется бесплатная версия системы Mantis, у которой есть ограничения в виде размера базы данных и количества одновременных подключений. Проблему может решить система отслеживания ошибок в программных продуктах, которая будет разработана для проведения лабораторных работ по дисциплине «Тестирование и контроль качества информационных систем».

Объект выпускной квалификационной работы: процесс выполнения лабораторных работ по дисциплине тестирование и контроль качества в ХТИ – филиале СФУ.

Цель выпускной квалификационной работы: разработка информационной системы «Система отслеживания ошибок в программных продуктах» для выполнения лабораторных работ по дисциплине «Тестирование и контроль качества информационных систем».

В соответствии с целью проекта были поставлены следующие задачи:

1. проанализировать образовательную организацию, на базе которой будет внедряться ИС;
2. определить актуальность разработки системы отслеживания ошибок в программных продуктах;
3. выполнить анализ бизнес-процесса тестирования информационных систем;
4. произвести выбор и обосновать выбор архитектуры и средств разработки системы;
5. разработать серверную и клиентскую часть системы;
6. оценить экономическую эффективность реализации системы.

1 Анализ предметной области

1.1 Описание основной деятельности ХТИ – филиала СФУ

Хакасский технический институт – филиал ФГАОУ ВО «Сибирский федеральный университет» готовит и выпускает специалистов по весьма обширному списку направлений, среди которых строительство, электроэнергетика и электротехника, эксплуатация транспортно-технологических машин и комплексов, конструкторско-технологическое обеспечение машиностроительных производств, прикладная информатика, экономика.

Миссией ХТИ – филиала СФУ является создание передовой образовательной, научно-исследовательской и инновационной инфраструктуры, продвижение новых знаний и технологий для решения задач социально-экономического развития Сибирского федерального округа, а также формирование кадрового потенциала — конкурентоспособных специалистов по приоритетным направлениям развития Сибири и Российской Федерации, соответствующих современным интеллектуальным требованиям и отвечающих мировым стандартам [1].

Образовательные программы ХТИ – филиала СФУ, направлены на обучение технических специалистов. Всего в вузе реализуют 6 направлений

подготовки бакалавров и 2 магистерские программы:

- 08.03.01 Строительство (квалификация бакалавр);
- 08.05.01 Строительство уникальных зданий и сооружений (квалификация инженер-строитель);
- 09.03.03 Прикладная информатика (квалификация бакалавр);
- 13.03.02 Электроэнергетика и электротехника (квалификация бакалавр);
- 15.03.05 Конструкторско-технологическое обеспечение машиностроительных производств (квалификация бакалавр);
- 23.03.03 Эксплуатация транспортно-технологических машин и комплексов (квалификация бакалавр);
- 08.04.01 Строительство (квалификация магистр);
- 13.04.02 Электроэнергетика и электротехника (квалификация магистр).

Хакасский технический институт является филиалом Сибирского федерального университета, который применяет современные технологии обучения в электронной информационно-образовательной среде.

1.2 Предпосылки для разработки системы отслеживания ошибок в программных

продук

тах

Для получения определенных навыков и знаний в тестирования программного обеспечения, в учебный план направления 09.03.03 Прикладная информатика ХТИ – филиала СФУ включен такой предмет, как «Тестирование и контроль качества информационных систем». На данный момент, в процессе преподавания данной дисциплины используется бесплатная версия системы Mantis, у которой есть множество ограничений в виде размера базы данных и количество одновременных подключений.

Из-за данных ограничений невозможно нормально проводить занятия, потому что система регулярно зависает, вплоть до полного отказа, что мешает студентам получать новые знания. Из-за вышеперечисленных проблем возникла потребность в разработке собственной аналогичной системы, которая решала бы проблемы, возникающие при использовании существующей системы, и давала бы студентам возможность нормально выполнять все лабораторные работы и получать те практические навыки, которые были актуальны в современных реалиях рынка труда.

Все вышеперечисленное является основанием для разработки информационной системы, позволяющей проводить лабораторные работы в запланированном режиме.

В связи с этим возникла идея разработки системы отслеживания ошибок в программных продуктах, с помощью которой можно было бы произвести все действия, которые требуются в рамках выполнения лабораторных работ.

1.3 Анализ бизнес-процесса

тестир

ования

инфор

мацио

нных

систем

Прежде чем автоматизировать процесс тестирования информационных систем необходимо проанализировать бизнес-процессы, происходящие во время тестирования ИС.

На рисунке 1 показана диаграмма IDEF0, на которой изображен процесс тестирования ИС.

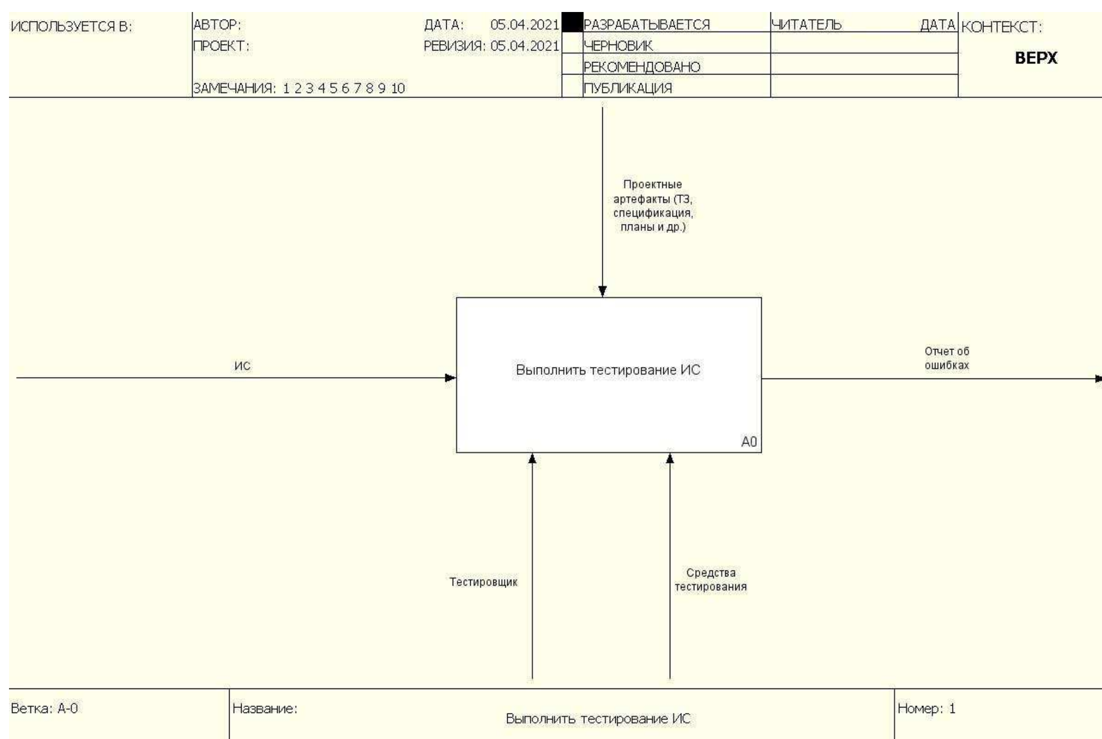


Рисунок 1 – Процесс тестирования ИС

Входными данными процесса является сама информационная система, которую необходимо протестировать.

Механизмами данного процесса являются тестировщик и средства тестирования.

Управляется процесс проектными артефактами, куда входят техническое задание, различные спецификации, планы и т.д.

Результатом выполнения процесса является отчет об ошибках.

Следующим этапом изучения предметной области является декомпозиция процесса тестирования ИС.

Диаграмма декомпозиции представлена на рисунке 2.

Рисунок 2 – Декомпозиция процесса тестирования ИС

Как видно, процесс тестирования ИС состоит из 4 функциональных блоков, а именно:

1. Составить план.
2. Проанализировать и спроектировать тесты.
3. Выполнить тестирование.
4. Проанализировать полученные результаты и свернуть тестовую среду.

Входными данными является тестируемая ИС.

Механизмами процесса являются тестировщик и средства тестирования.

Управляется процесс проектными артефактами, такими как техническое задание, спецификации, планы и т.д.

Результатом отработки всех функциональных блоков является отчет об ошибках.

Следующим этапом нужно выполнить декомпозицию процесса блока «Выполнить тестирование». Декомпозиция данного блока изображена на рисунке 3 в виде диаграммы IDEF0.

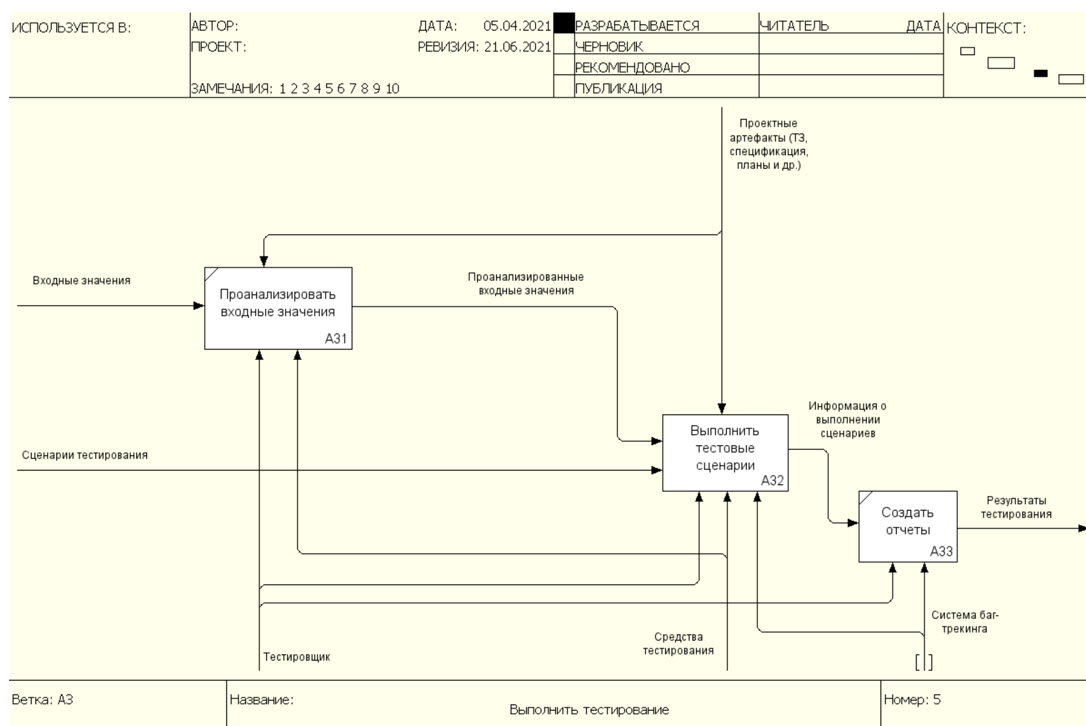


Рисунок 3 – Декомпозиция процесса «Выполнить тестирование»

Процесс «Выполнить тестовые сценарии» декомпозируется на следующие 3 блока:

1. Проанализировать входные значения.
2. Выполнить тестовые сценарии.
3. Создать отчеты.

Входными данными являются входные значения и сценарии тестирования.

Механизмами процесса являются тестировщик, средства тестирования и система отслеживания ошибок в программных продуктах.

Управляется процесс посредством проектных артефактов.

В результате работы выходной информацией являются результаты тестирования.

Теперь необходимо декомпозировать блок с названием «Выполнить тестовые сценарии». Результат декомпозиции данного блока изображен на рисунке 4 в нотации IDEF3.

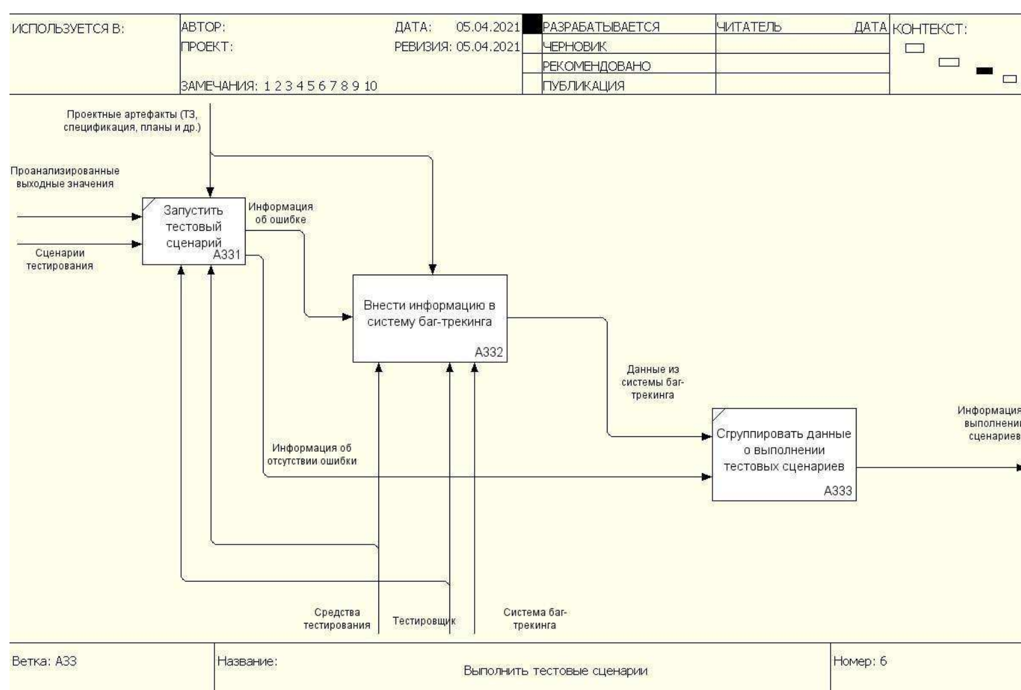


Рисунок 4 – Декомпозиция блока «Выполнить тестовые сценарии»

Блок «Выполнить тестовые сценарии» декомпозируется на 3 следующих функциональных блока:

1. Запустить тестовый сценарий.

2. Внести информацию в систему отслеживания ошибок в программных продуктах.

3. Сгруппировать данные о выполнении тестовых сценариев.

Входами являются проанализированные входные значения и сценарии тестирования.

Входными данными являются проанализированные входные значения и сценарии тестирования.

Механизмами процесса являются средства тестирования, тестироващик и система отслеживания ошибок в программных продуктах.

Управляется процесс проектными артефактами.

Результатом выполнения данного процесса является информация о выполнении сценариев.

Декомпозируем процесс «Внести информация в систему отслеживания ошибок в программных продуктах». Результат декомпозиции изображен на рисунке 5.

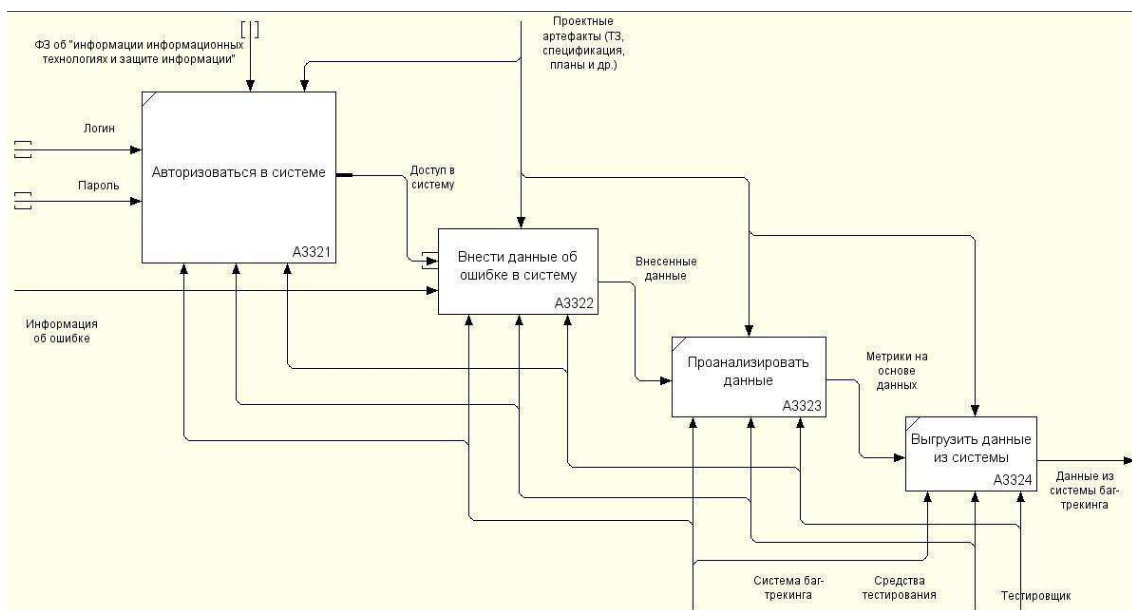


Рисунок 5 – Декомпозиции блока «Выполнить тестовые сценарии»

Процесс выполнения тестовых сценариев декомпозируется на 4 блока:

1. Авторизовать в системе.

2. Внести данные об ошибке в систему.

3. Проанализировать данные.

4. Выгрузить данные из системы.

Входными данными этого процесса являются логин, пароль и информация об ошибке.

Механизмами процесса являются система отслеживания ошибок в программных продуктах, средства тестирования и тестировщик.

Управляется данный процесс ФЗ «Об информации, информационных технологиях и защите информации».

Результатами работы процесса являются данные из системы отслеживания ошибок в программных продуктах.

Финальным этапом изучения предметной области будет декомпозиция процесса «Внести данные об ошибке в систему». Декомпозиция данного процесса изображена на рисунке 6.

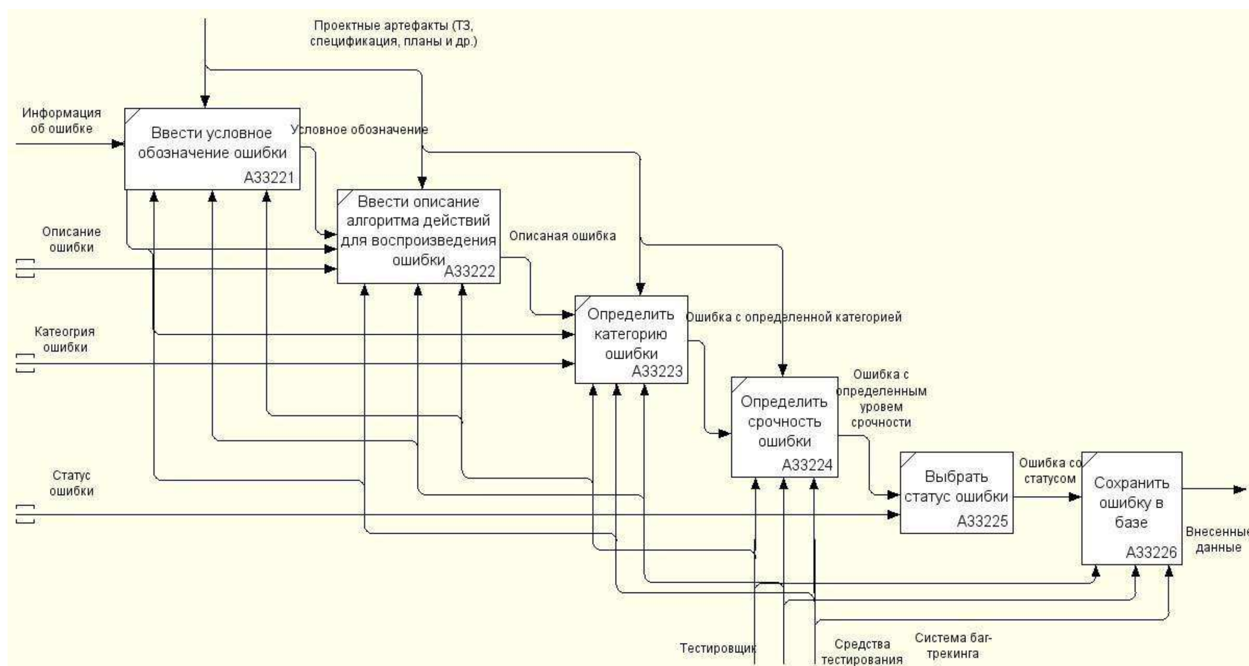


Рисунок 6 – Декомпозиция процесса внесения данных об ошибке

Процесс внесения данных об ошибке декомпозируется на 6 функциональных блоков, таких как:

1. Внести условное обозначение ошибки.

2. Ввести описание алгоритма действий для воспроизведения ошибки.

3. Определить категорию ошибки.

4. Определить приоритет ошибки.

5. Выбрать статус ошибки.

6. Сохранить ошибку в базу.

Входными данными для данного процесса являются информация об ошибке, описание ошибки, категория ошибки и статус ошибки.

Механизмами процесса являются тестировщик, средства тестирования и система отслеживания ошибок в программных продуктах.

Управляется данный процесс проектными артефактами.

Результатом работы данного процесса являются внесенные в базу данные.

1.4 Атрибуты ошибок, указываемые в баг-трекинговых системах

1.4.1 Локализации и серьезность

Обнаружение ошибки в программном продукте и ее фиксация в баг-трекинговой системе предполагают определенные действия, которые последуют после. Эти действия – действия программиста, который реагирует

на факт фиксации ошибки, анализирую ее влияние на программный продукт.

Действие программиста начинается с воспроизведения ошибки по ее описанию тестировщиком, в связи с чем первым и важным атрибутом в описании ошибки являются ее локализация и приоритет серьезности.

Локализация бага (localization bug) – процедура нахождения и описания условий, при котором дефект повторяется. При обнаружении ошибки (бага) тестировщиком, ему необходимо проанализировать ошибку, задокументировать и воспроизвести.

Серьезность (severity) – атрибут, характеризующий влияние дефекта на работоспособность программного продукта. Выставляя серьезность дефекта, тестировщик оценивает его влияние на работоспособность приложения. Чем выше серьезность, тем масштабнее негативные последствия данного дефекта.

Принята следующая градация серьезности ошибок:

– *Блокирующая* (Blocker). Блокирующая ошибка приводит приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна. Решение проблемы необходимо для дальнейшего функционирования системы.

– *Критическая* (Critical). Критическая ошибка – неправильно работающая ключевая бизнес-логика, дыра в системе безопасности; проблема, влекущая временное падение сервера или приводящая в нерабочее состояние некоторую часть системы без возможности обойти проблему, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системы.

– *Значительная* (Major). Значительная ошибка – ошибка, при которой часть основной бизнес-логики работает некорректно. Ошибка некритична или есть возможность для работы с тестируемой функцией, используя другие входные точки.

– *Незначительная* (Minor). Незначительная ошибка не нарушает бизнес-логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.

– *Тривиальная* (Trivial). Тривиальная ошибка не касается бизнес-логики приложения, это плохо воспроизводимая проблема пользовательского интерфейса; проблема сторонних библиотек или сервисов; проблема, не оказывающая никакого влияния на общее качество продукта.

Серьезность ошибки определяет приоритет ее исправления и указывается в баг-трекинговых системах.

Приоритет (Priority) – атрибут, указывающий на очередность выполнения задачи или устранения дефекта.

Градация приоритета дефекта может быть следующая:

– *Высокий приоритет* (High). Ошибка должна быть исправлена как можно быстрее, т. к. ее наличие является критической для ПО.

– *Средний приоритет* (Medium). Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.

– *Низкий приоритет* (Low). Ошибка должна быть исправлена, ее наличие не является критичной и не требует срочного решения.

Чем выше приоритет, тем быстрее нужно исправить дефект. Однако не всегда приоритет ошибки может быть связан с ее серьезностью, блокирующим или критическим действием.

Например, слово, напечатанное с ошибкой, имеет самый низкий уровень серьезности, но перед релизом эта ошибка может иметь наивысший приоритет и должна быть экстренно исправлена, например, как в случае фестиваля «ВолгаФест», где на плакате с изображением первого космонавта Земли было написано «Алексей Гагарин», вместо правильного «Юрий Гагарин». Организаторы назвали это «досадной ошибкой», которая была допущена из-за «больших объемов работ».

В пресс-службе выразили надежду, что "эта ошибка никого не расстроила", однако с этим согласны далеко не все, кто видел плакат [2].

В связи с тем, что высокий приоритет не всегда связан с серьезностью ошибки, в баг-трекинговых системах серьезность не устанавливается как обязательный атрибут, а может быть включена в описание, а обязательным

атрибутом является приоритет ошибки.

1.4.2 Атрибуты жизненного цикла бага: статус и резолюция

Простой жизненный цикл ошибки без применения инструментальных средств: тестировщик приходит к разработчику и просто сообщает об ошибке, но возникает риск, что разработчик может забыть про ошибку и не исправить ее. Поэтому обычно в IT-компаниях дефекты документируют в системах учета дефектов (баг-трекинговые системы, bug tracking system, BTS, bug-tracker) с возможностью общего доступа к ним.

В каждой *BTS* наименования важности (серьезности), приоритеты и статусы дефектов могут отличаться от приведенных и зависят от конкретной системы, но суть от этого не меняется. Распространенные атрибуты жизненного цикла:

1. статус;
2. резолюция.

Статус – основной атрибут, определяющий текущее состояние дефекта. Статус отражает меру активности бага от начального состояния, когда он еще не подтвержден как баг, до завершения, когда баг исправлен/решен. Набор атрибутов зависит от конкретной BTS, но чаще встречаются следующие:

- *Новый.* Дефект только что зарегистрирован, в зависимости от анализа дефекта его статус меняется на Отказ или Назначен.
- *Назначен* или *Открыт*. Дефект просмотрен и открыт (можно сказать, признан) для исправления и может быть назначен на другого сотрудника или переведен в состояние NEW.
- *Открыт заново*. Дефект был решен ранее, однако возникла необходимость вернуться к нему (решение было неверным либо неокончательным).
- *Закры*т. В результате определенного количества циклов баг все-таки окончательно устранен и больше не потребует внимания команды – он

объявляется закрытым.

Резолюция – важный атрибут, напрямую связанный со статусом, т. е. *резолюция* – детализация статуса. Финальная резолюция *Решено* – стандартная резолюция, означающая, что задание выполнено или баг исправлен.

После дефект переходит обратно в сферу ответственности тестировщика. Как правило, сопровождается резолюцией, например:

1. *Исправлено* (исправления включены в версию XXX).
2. *Дубль* (повторяет дефект, уже находящийся в работе).
3. *Не исправлено* (работает в соответствии со спецификацией, имеет слишком низкий приоритет, исправление отложено до следующей версии и т. п.).
4. *Невоспроизводимо* (запрос дополнительной информации об условиях, в которых дефект проявляется).
5. *Некорректно* – неправильная или некорректная постановка задачи.
6. *Проблема есть, но решаться не будет* – такая резолюция может быть вынесена в отношении request for enhancement (просьб об усовершенствовании), которые хоть и имеют смысл, являются слишком трудновыполнимыми и не являются обязательными.
7. *Дублирует* – описанная проблема уже зарегистрирована в другом баге.
8. *А у меня работает…* – не удалось воспроизвести описанную проблему ни эмуляцией сценария, ни анализом кода.
9. *Отказ* – сопровождается комментарием программиста причине отклонения: некачественное описание дефекта, дублирование эффекта, невозможность воспроизвести дефект. После этого тестер или закрывает дефект (Closed), или дополняет комментарии данного дефекта и переводит дефект заново в состояние Назначен (Open).
10. *Ожидает повторного тестирования* - баг-репорт ожидает повторного тестирования.
11. *Это не дефект.* Баг-репорт может иметь такой статус, например,

если клиент попросил внести небольшие изменения в продукт: изменить цвет, шрифт и т. д.

12. *Повторное тестирование.* На этом этапе тестировщики проверяют изменения, внесенные разработчиками, и повторно тестируют их.

1.4.3 Отчет о дефекте (баг-репорт)

Каждый дефект имеет ряд обязательных свойств, которые нужно внести в BTS при добавлении данного дефекта. Для этого необходимо написать отчет о дефекте (отчет об инциденте, баг-репорт).

Отчет о дефекте (баг-репорт, bug-report) – это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

В идеале описание дефекта должно иметь следующую структуру:

1. Уникальный номер: присваивается автоматически в BTS.
2. Краткое название: короткий текст, который помогает сразу понять, что это за дефект.
3. Описание: полное описание дефекта, включая шаги для воспроизведения.
4. Платформа: указывается, что используется для запуска программного обеспечения, в частности имя и версия операционной системы; в случае веб-приложения – имя и версия веб-браузера.
5. Шаги по воспроизведению (Steps to reproduce): шаги для воспроизведения дефекта разработчиком.
6. Ожидаемый результат. Что должно произойти, когда вы делаете какое-нибудь действие? Что вы ожидаете?
7. Фактический результат. Результат ошибки. Что случилось на самом деле?
8. Приложения: некоторые дефекты сложно описать, поэтому для наглядности к описанию ошибки прилагаются скриншоты, видео или лог-

файлы.

9. Серьезность - описывает влияние ошибки.

10. Приоритет - с помощью данного свойства определяется очередность исправления данного дефекта программистом.

11. Статус – состояние отчета об ошибке в любой системе отслеживания багов. Первоначальный статус отчета об ошибке – *Новы*й. После этого статус может измениться на резолюции *Назначен* или *Открыт, Открыт заново, Закры*т.

12. Комментарии – комментарии к багу.

Часто шаги воспроизведения (Steps to reproduce), фактический результат (Result) и ожидаемый результат (Expected Result) записывают в описание.

Если дефект описан согласно данной схеме, то он вызовет меньше всего вопросов, и разработчик, не теряя время на дополнительные разъяснения, приступит к его исправлению. Пример баг-репорта в системе Mantis:

0000159

Отсутствует контент на странице сайта «Кинополис» на вкладке «Сеансы».

Описание: Отсутствие нужной информации на странице сайта https://kino-polis.ru/ приводит к блокировке множества функций сайта, а именно: невозможно выбрать сеанс, забронировать кресло в зале и произвести оплату, т. к. все эти функции недоступны из-за ошибки. Также на странице не загружается рекламный видеоролик из-за ошибки: Error loading media: File could not be played!

Шаги по воспроизведению:

1. Открыть страницу сайта https://kino-polis.ru/.

2. Перейти на вкладку «Сеансы».

1. Дополнительные сведения: Фактический результат: отсутствие нужной информации на странице, необходимой для выполнения дальнейших действий.

2. Ожидаемый результат: открытие страницы с сеансами, где можно посмотреть информацию о фильме, описание и продолжительность,

характеристики зала; выбрать зал и места, произвести оплату выбранных мест.

3. Приоритет: Неотложный.

4. Влияние: Блокирующее.

5. Подкрепление: рисунок (рис. 7).

Грамотно написанный отчет об ошибке (баг-репорт) увеличивает шансы, что ошибка будет исправлена. Рекомендации для написания баг-репорта:

1. «Одна ошибка – один баг-репорт». Один баг в репорте может помочь избежать дублирования и путаницы.

2. Указывается сначала глагол. Все глаголы ставят в неопределенной форме: нажать, открыть, перейти.

3. Писать без лишних слов, использовать простые конструкции. Давать скриншотам «говорящие» заголовки, например «Сортировка.png».

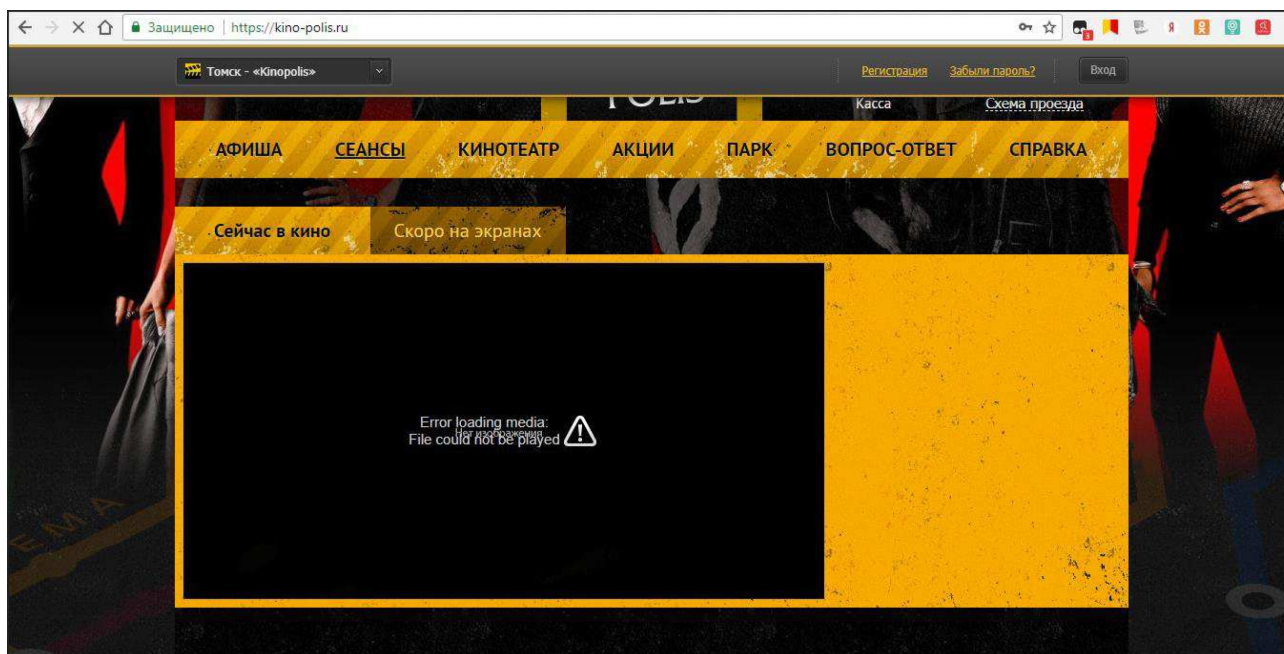


Рисунок 7 – Дефект на странице сайта «Кинополис» на вкладке «Сеансы»

4. Придерживаться принципа «Что-Где-Когда». Обязательно приводить ссылку, максимально подробную, на тот раздел, где баг. Но при этом не стоит полагаться только на ссылку, она может сломаться. Писать путь к

нужному месту по шагам. Если ошибка в мобильном приложении, то приводить максимум скриншотов на шаги.

5. Если баг только для авторизованных пользователей, то обязательно указывать данные для входа: логин и пароль.

6. Все скриншоты или упоминания о них приводить в нужных местах с названием, например см. скрин Сортировка.png.

7. Указывать серьезность и приоритет дефекта.

8. Обезличенность. Когда сообщают об ошибке, то сообщают о дефекте программного обеспечения, а не о дефекте разработчика.

9. Обязательно указывать фактический и ожидаемый результат. Это позволит уменьшить вероятность неправильных отчетов о дефектах, так как без ожидаемого результата трудно определить, прошел тест или нет.

1.5 Обоснование выбора архитектуры и средств разработки системы

1.5.1 Выбор архитектуры приложения

Для разработки клиентской части существует множество подходов,

например: десктоп-приложение, веб-приложение и мобильные приложения. Работать со столь большой системой с маленького экрана мобильного устройства было бы слишком неудобно, поэтому вариант с мобильными приложениями нецелесообразен с точки зрения использования. Десктоп-приложение также имеет ряд проблем, например совместимость с различными архитектурами процессоров и совместимость с различными операционными системами. Разрабатывать десктоп-клиенты под все операционные системы было бы слишком долго, поэтому в качестве клиентской части логичнее всего разработать веб-приложение. Благодаря тому, что веб-браузеры разрабатываются в соответствии с одними и теми же конвенциями и портированы практически под все платформы, разработанное веб-приложение получится кроссплатформенным.

Поскольку система подразумевает одновременное использование многими пользователями, то логичнее вынести бизнес-логику и оформить в виде отдельного приложения. Это приложение будет называться сервером, который будет взаимодействовать с базой данных и обрабатывать различные запросы, которые будет отправлять клиентское приложение.

В настоящее время на рынке актуальны два подхода к построению сервера: монолитная и микросервисная архитектура. Микросервисная архитектура обрела большую популярность относительно недавно.

Монолитная архитектура – это архитектура, в которой приложение представлено в виде единого приложения, которое содержит внутри себя всю бизнес-логику. Клиентские приложения, как правило, обращаются только к одному приложению, хотя возможна и стратегия, когда одно клиентское приложение взаимодействует с более чем одним сервисом.

Микросервисная архитектура – вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей – *микросервисов.* Как правило, сервисы взаимодействуют друг с другом посредством протокола HTTP, gRPC и SOUP напрямую, или же

осуществляют взаимодействие через различные брокеры сообщений по протоколу AMQP, например IBM MQ, RabbitMQ, Apache Kafka и другие.

У обоих подходов к построению архитектуры серверной части есть свои достоинства и недостатки. Достоинства и недостатки монолитной архитектуры представлены в таблице 1. Достоинства и недостатки микросервисной архитектуры представлены в таблице 2.

Таблица 1 – Достоинства и недостатки монолитной архитектуры

Плюсы	Минусы
Скорость разработки.	При большой кодовой базе сложно поддерживать.
Переиспользование кода.	Сложно вносить изменения в кодовую базу.
Низкие требования к автоматизации процесса сборки, тестирования и доставки.	Плохая отказоустойчивость.
Легко развертывается.	Отсутствие изоляции – проблема или ошибка в модуле может сломать все приложение.
Простое вертикальное масштабирование.	Плохое горизонтальное масштабирование.

Таблица 2 – Достоинства и недостатки микросервисной архитектуры

Плюсы	Минусы
Горизонтальное масштабирование.	Высокие требования к автоматизации сборки, тестирования и доставки.
Хорошая отказоустойчивость.	Дорого.
Переиспользование кода.	Сложно разрабатывать.
Может разрабатываться на разных языках программирования с использованием разных технологий.	Отсутствие согласованности данных.

На основании анализа двух подходов к построению серверных архитектур можно сформулировать следующий вывод: на старте разработки лучше всего использовать монолитную архитектуру, поскольку это поможет быстрее реализовать необходимый функционал системы и осуществить более быстрый ввод ее в эксплуатацию.

Микросервисная архитектура очень хорошо себя показывает при высоких нагрузках на систему за счет своей хорошей горизонтальной масштабируемости и высокой отказоустойчивости, ведь если один из

микросервисов «упадет», то у пользователей системы в худшем случае перестанет работать одна кнопка на клиентской части.

Вывод: информационная система отслеживания ошибок в программных продуктах будет разрабатываться с использованием клиент-сервисной архитектуры. Серверная часть будет реализована в виде монолитного приложения.

1.5.2 Выбор языков программирования

В настоящее время на рынке присутствует множество языков программирования. Не существует инструмента, который бы одинаково хорошо решал любую поставленную задачу, поэтому к выбору инструментов необходимо относится вдумчиво. Язык программирования – это, прежде всего инструмент, пригодный для решения конкретных задач программирования. По этой причине к выбору языков и технологий нужно подойти с осознанием задач, которые нам предстоит решить.

Для разработки веб-приложений вариантов достаточно не много, поскольку все браузеры поддерживают одни и те же спецификации, о чем уже было упомянуто ранее. Для отображения информации будет использован язык гипертекстовой разметки HTML, для придания HTML-коду визуальной составляющей будет использована таблица каскадных стилей – CSS.

Для придания динамичности сайту и отправки запросов на сервер будет использован язык JavaScript, который является реализацией спецификации ECMAScript.

Для разработки серверной части целесообразнее всего использовать компилируемые языки со строгой типизацией, это поможет определить достаточно большую часть ошибок еще на этапе компиляции и сборки программы. По этой причине скриптовые языки с динамической типизацией использоваться не будут. При выборе языка стоит обратить внимание на ряд следующих факторов:

1. Зрелость языка – если язык уже достаточно зрелый, и при этом не теряет своей популярность, то можно судить о его стабильности и обратной совместимости.

2. Сообщество программистов, использующих язык, иными словами – его популярность. Если язык популярен, есть много информации в виде книг, статей в интернете и видео. При возникновении проблем с использованием языка будет достаточно легко найти информацию по их устранению.

3. Богатство внутренней библиотеки. Богатая внутренняя библиотека позволит сэкономить время на разработке очевидных вещей, будет возможность сразу использовать готовые компоненты языка разрабатываемые и поддерживаемые высококвалифицированными специалистами.

4. Скорость работы.

Опираясь на вышеперечисленные пункты, можно определить 3 претендентов:

1. C++ (читается как Си Плюс Плюс) – компилируемый, статически типизированный язык программирования общего назначения.

2. Java (читается как Джава) – строго типизированный объектно-ориентированный язык программирования общего назначения.

3. C# (читается как Си Шарп) – объектно-ориентированный язык программирования.

Все три вышеперечисленных языка активно используются для разработки серверных приложений.

Язык программирования C++, несмотря на свою скорость работы, является достаточно сложным для разработки больших программ. У него очень большое сообщество программистов и, как следствие, по нему есть очень много материалов, но из-за своей сложности он с каждым годом теряет рынок в пользу более современных языков вроде Java и C#, по этой причине он выбран не будет.

Язык программирования Java в настоящий момент является лидером в разработке серверных приложений на рынке. Согласно различным рейтингам,

Java является самым популярным языком программирования в мире среди всех языков программирования. Все крупные высоконагруженные современные системы используют этот язык для разработки, включая банки и биржи. Данный язык достаточно быстро работает, по нему есть огромное количество материалов и ответы на все вопросы, которые могут возникнуть в работе. Также он имеет очень богатую библиотеку классов, что делает его еще более привлекательным.

Язык программирования C# – самый молодой из представленных. Его синтаксис очень похож на синтаксис Java, поскольку при проектировании этого языка разработчики во многом опирались на опыт команды, разрабатывающей Java. Он также является одним из самым популярных языков, но информации о нем в интернете на порядок меньше, чем по двум вышеперечисленным языка программирования.

Вывод, для разработки серверной части информационной системы выбран язык Java.

1.5.3 Выбор библиотек и фреймворков

Так как для разработки веб-приложения были выбраны такие языки как HTML, CSS и JavaScript, а для разработки серверного приложения был выбран язык Java, то выбирать библиотеки и фреймворки нужно исключительно в рамках перечисленных выше технологий.

Для разработки веб-приложения будут использованы следующие библиотеки:

1. ReactJS – JavaScript-библиотека с открытым исходным кодом для разработки пользовательских интерфейсов. Разрабатывается и поддерживается компанией Facebook [4].

2. Redux – JavaScript-библиотека с открытым исходным кодом, предназначена для управления состоянием приложения.

4. WebPack – сборщик модулей JavaScript с открытым исходным кодом.

5. NodeJS – программная платформа, позволяющая запускать JavaScript вне браузера.

6. Axios – JavaScript-библиотека с открытым исходным кодом, предназначена для отправки HTTP запросов. Будет использована для передачи запросов на сервер.

Для разработки серверной части будут использованы следующие библиотеки и фреймворки:

1. Spring Framework – универсальный фреймворк с открытым исходным кодом для Java-платформы [5].

2. Spring Data JPA – реализация спецификации JPA, созданной компанией Oracle для экосистемы Java. Служит для сохранения данных в базу данных. Под капотом используется ORM Hibernate.

3. Spring Security — это Java/Java EE фреймворк, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для промышленных приложений, созданных с помощью Spring Framework [6].

5. Apache Tomcat - веб-сервер для запуска Java-сервлетов, основан на веб-сервере Apache.

6. Hibernate – ORM для Java. Служит для преобразования реляционных таблиц в объекты Java и наоборот. Будет использована в виде Spring Data JPA [7].

Для написания веб-приложения была выбрана интегрированная среда разработки WebStorm, разработанная компанией JetBrains. Данная среда разработки имеет встроенный статический анализатор кода, который помогает исправлять типичные ошибки в коде, которые могут быть допущены по невнимательности. Также среда позволяет создавать различные конфигурации для запуска проекта, имеет встроенный отладчик и множество других полезных функций. Основана на движке IDEA.

1.5.4 Выбор постоянного хранилища данных

В качестве постоянного хранилища данных была выбрана реляционная система управления базами данных PostgreSQL. Данная СУБД является бесплатной, имеет очень быструю скорость работы, а также возможность масштабироваться посредством реплицирования, так же есть возможность использовать ACID-транзакции [8].

Для работы с базой данных был выбран плагин, встроенный в среду разработки Intellij IDEA, в котором использованы наработки другой IDE компании JetBrains – Data Grip. Данный плагин позволяет открывать сразу несколько подключений к разным базам данных, имеет удобную навигацию, умеет делать дампы баз данных, а также имеет встроенную консоль с функциями автодополнения с анализом схемы текущей базы данных. Также данный плагин поддерживает большинство современных SQL-диалектов.

1.5.5 Выбор системы контроля версий

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Де-факто стандартом стала система контроля версий Git, разработанная известным программистом Линусом Торвальдсом, автором ядра операционной системы Linux.

Также будет использована обертка поверх системы контроля версий Git – GitHub. GitHub – это крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки.

Также немалую роль данный сервис сыграет при работе с CI/CD

сервером.

1.5.6 Выбор сервера CI/CD

CI – практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки (до нескольких раз в день) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

CD – это подход к разработке программного обеспечения, при котором программное обеспечение производится короткими итерациями, гарантируя, что ПО является стабильным и может быть передано в эксплуатацию в любое время, а передача его происходит вручную.

Обе практики относятся к методологии DevOps, которая в последнее время стала очень популярна благодаря моде на микросервисную архитектуру.

Самым простым и очевидным вариантом в данном пункте является сервер непрерывной интеграции Jenkins.

Данный продукт является бесплатным, имеет открытый исходный код, очень просто настраивается и имеет множество плагинов, которые позволяют расширить функционал.

Данный класс ПО позволит автоматизировать процессы сборки и тестирования разрабатываемой ИС.

1.6 Схема взаимодействия пользователя с

систем

ой

Чтобы понять, как именно будут перетекать данные внутри разрабатываемой системы, построим диаграмму потоков данных DFD, которая показана на рисунке 8.

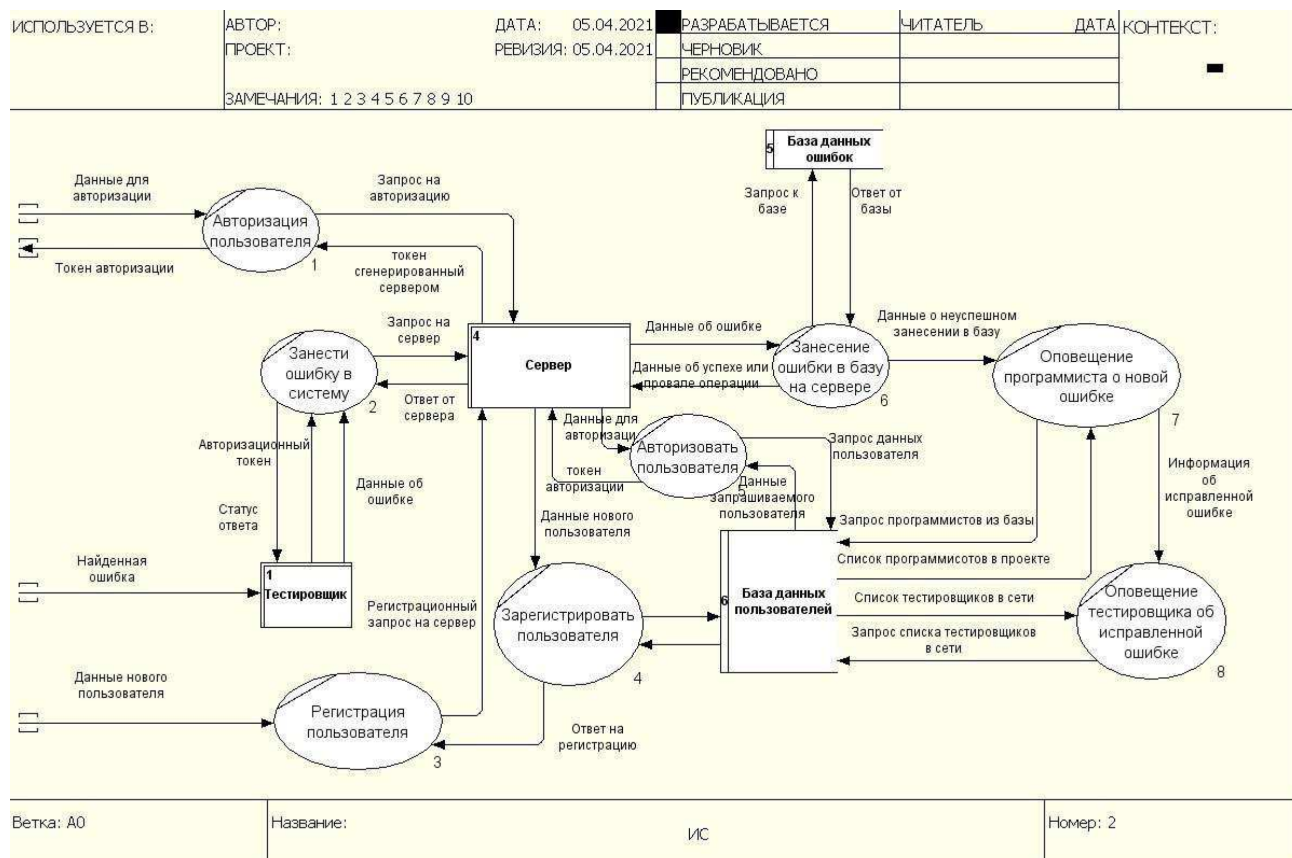


Рисунок 8 – Диаграмма потоков данных DFD

На диаграмме представлен процесс регистрации в системе, авторизации в системе и занесения ошибки в систему. На диаграмме изображены 8 процессов:

1. Авторизация пользователя.
2. Регистрация пользователя.
3. Занесение ошибки в систему.
4. Авторизовать пользователя.

5. Зарегистрировать пользователя.
6. Занести ошибку в систему.
7. Оповещение программиста о новой ошибке.
8. Оповещение тестировщика об исправленной ошибке.

Две внешние сущности:

1. Тестировщик.
2. Сервер.

И два хранилища данных:

1. База данных пользователей.
2. База данных ошибок.

Также, чтобы лучше понять, как пользователь сможет взаимодействовать с системой, построим диаграмму IDEF3, которая находится на рисунке А.1 в двух частях в приложении А.

1.7 Выводы по разделу «Анализ предметной области»

В первом разделе выпускной квалификационной работы описана основная деятельность ХТИ – филиал СФУ, для которого разрабатывается система отслеживания ошибок в программных продуктах.

Проведен анализ процесса тестирования программных продуктов на наличие ошибок, построены диаграммы в нотации IDEF0. Проанализирован

бизнес-процесс тестирования ИС с последующей декомпозицией в виде диаграмм IDEF0 вплоть до уровня, на котором будет использована данная система.

Принято решение о выборе клиент-серверной архитектуры. Описываются модели информационной системы с помощью диаграмм в нотациях IDEF0, IDEF3, DFD.

Выполнен выбор и обоснование средств разработки. Определены программные средства, которые использованы при разработке данной информационной системы, такие как языки программирования, библиотеки, фреймворки, базы данных и другие вспомогательные технологии.

2 Описание разработки системы отслеживания ошибок в программных продуктах

2.1 Жизненный цикл разработки системы отслеживания ошибок в программных продуктах

Для разработки информационной системы выбрана методология Agile, которая позволит оптимизировать временные затраты на разработку и системы.

Принципы Agile:

1. Задача высшего приоритета — регулярно и как можно раньше удовлетворять потребности заказчика, предоставляя ему программное обеспечение.

2. Учитывать, что требования могут измениться на любом этапе разработки. Если изменения быстро вносятся в проект, заказчик может получить конкурентные преимущества.

3. Выпускать версии готовой программы как можно чаще — с промежутком от двух недель до двух месяцев.

4. Ежедневно вместе работать над проектом — разработчикам и заказчикам.

5. Поручить работу мотивированным профессионалам. Обеспечить поддержку и условия, довериться им — и работа будет сделана.

6. Общаться напрямую — это самый эффективный способ взаимодействия внутри команды и вне ее.

7. Считать главным показателем прогресса работающий продукт.

8. Поддерживать постоянный ритм работы — касается и разработчиков, и заказчиков.

9. Уделять пристальное внимание техническому совершенству и качеству проектирования — это повышает гибкость проекта.

10. Минимизировать лишнюю работу.

11. Стремиться к самоорганизующейся команде — в ней рождаются наиболее эффективные и качественные решения.

12. Всем участникам команды — постоянно искать способы повышать эффективность работы.

2.2 Проек

тирова
ние
базы
данны
х

Для начала работы с базой данных необходимо разработать ER-диаграмму сущностей.

Схема «сущность-связь» (также ERD или ER-диаграмма) — это разновидность блок-схемы, где показано, как разные «сущности» (люди, объекты, концепции и так далее) связаны между собой внутри системы.

ER-диаграммы чаще всего применяются для проектирования и отладки реляционных баз данных в сфере образования, исследования и разработки программного обеспечения и информационных систем для бизнеса.

ER-диаграммы (или ER-модели) полагаются на стандартный набор символов, включая прямоугольники, ромбы, овалы и соединительные линии, для отображения сущностей, их атрибутов и связей.

Эти диаграммы устроены по тому же принципу, что и грамматические структуры: сущности выполняют роль существительных, а связи — глаголов. Диаграмма показана на рисунке 9.

Для лучшего понимания данной диаграммы следует подробно ее описать.

Таблица «project» хранит в себе записи о проектах, каждая запись имеет идентификатор и название проекта.

Поскольку на начальных этапах трудно спроектировать уже готовый вариант и он будет много раз дополняться по ходу разработки, то этого для начала хватит. У проекта есть свой собственный набор типов багов и приоритетов багов, записи об этих сущностях находятся в таблицах «bug_type» и «bug_priority» соответственно.

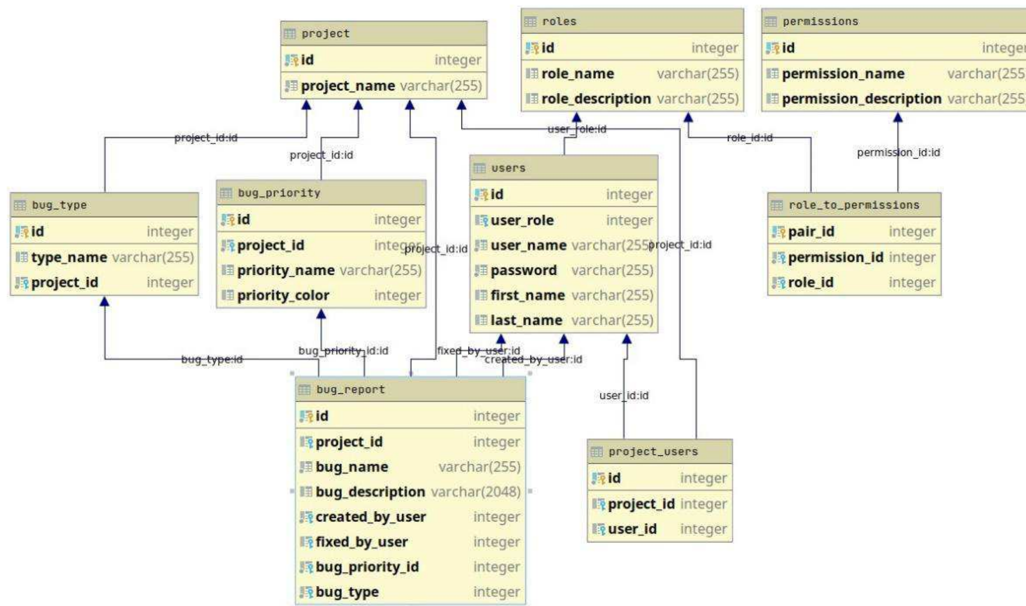


Рисунок 9 – ER-диаграмма базы данных

Самая главная сущность в данной системе – это отчет об ошибке, хранящийся в таблице «bug_report». В данной таблице есть поле, которое хранит идентификатор проекта, к которому относится баг, имя бага и описание для его воспроизведения, а также идентификаторы создавшего пользователя и пользователя, который взялся за исправление ошибки. Помимо этого, у отчета об ошибке есть приоритет и тип, описанные ранее.

Также у каждого проекта есть пользователи, у пользователей есть роли, а роли в свою очередь это логически сгруппированные разрешения, которые и наделяют пользователя различными возможностями.

2.3 Настройка репозитория GitHub

В качестве системы контроля версий был использован общепринятый Git, который позволит хранить все изменения, внесенные в репозиторий в виде коммитов. В случае, если при разработке была допущена ошибка и требуется «откатиться» назад – это можно будет сделать достаточно просто, и не придется вручную все переписывать обратно. Также можно параллельно работать над разным функционалом, переключаясь между ветками.

Для создания нового репозитория необходимо открыть терминал в той папке, где будет проект и ввести команду «git init». После выполнения данной команды будут созданы метафайлы, в которых будут регистрироваться изменения в файлах проекта.

Также был создан файл с названием «.gitignore» в корне проекта, в котором указаны файлы и директории, которые не должны индексироваться гитом. Содержимое данных папок и файлы не будут отправляться в удаленный репозиторий. В данном конфиг-файле следует прописать те файлы и директории, которые не относятся непосредственно к разработке, такие как сборки, результаты отработки тестов, но при этом могут быть воспроизведены заново их исходных кодов. Код данного файла приведен на рисунке 10.

Рассмотрим содержимое данного файла более подробно.

В строках 1-6 описаны файлы и папки связанные с системой сборки Gradle. В данных строках описаны все файлы и папки, генерируются во время сборки проекта.

Строки 8-18 относятся к файлам, которые будут созданы, если открыть код проекта в среде разработки Spring Tool Suite.

Строки 20-28 относятся к файлам, которые создает среда разработки Intellij IDEA. Они нужны для работы программы, но хранить их в репозитории не стоит, поскольку они не содержат в себе исходный код проекта.

Строки 30-35 исключают из индексации файлы и папки среды разработки NetBeans.

Строки 37-38 относятся к текстовому редактору Visual Studio Code, который был использован для написания кода клиентской части.

Строки 40-44 исключают индексирование библиотек, которые нужны для сборки проекта, а также файлы сборок, поскольку их всегда можно воссоздать из исходного кода.

```
1   HELP.md
2   .gradle
3   build/
4   !gradle/wrapper/gradle-wrapper.jar
5   !**/src/main/**/build/
6   !**/src/test/**/build/
7
8   ### STS ###
9   .apt_generated
10  .classpath
11  .factorypath
12  .project
13  .settings
14  .springBeans
15  .sts4-cache
16  bin/
17  !**/src/main/**/bin/
18  !**/src/test/**/bin/
19
20  ### IntelliJ IDEA ###
21  .idea
22  src/web-app/.idea
23  *.iws
24  *.iml
25  *.ipr
26  out/
27  !**/src/main/**/out/
28  !**/src/test/**/out/
29
30  ### NetBeans ###
31  /nbproject/private/
32  /nbbuild/
33  /dist/
34  /nbdist/
35  /.nb-gradle/
36
37  ### VS Code ###
38  .vscode/
39
40  ### webapp ###
41  src/main/resources/static/*
42  src/web-app/node_modules
43  src/web-app/package-lock.json
44  src/web-app/yarn.lock
```

Рисунок 10 – Содержимое файла gitignore

Для удаленного хранения репозитория был выбран крупнейший в мире веб-сервис для хостинга IT-проектов – GitHub. Страница репозитория показана на рисунке 11.

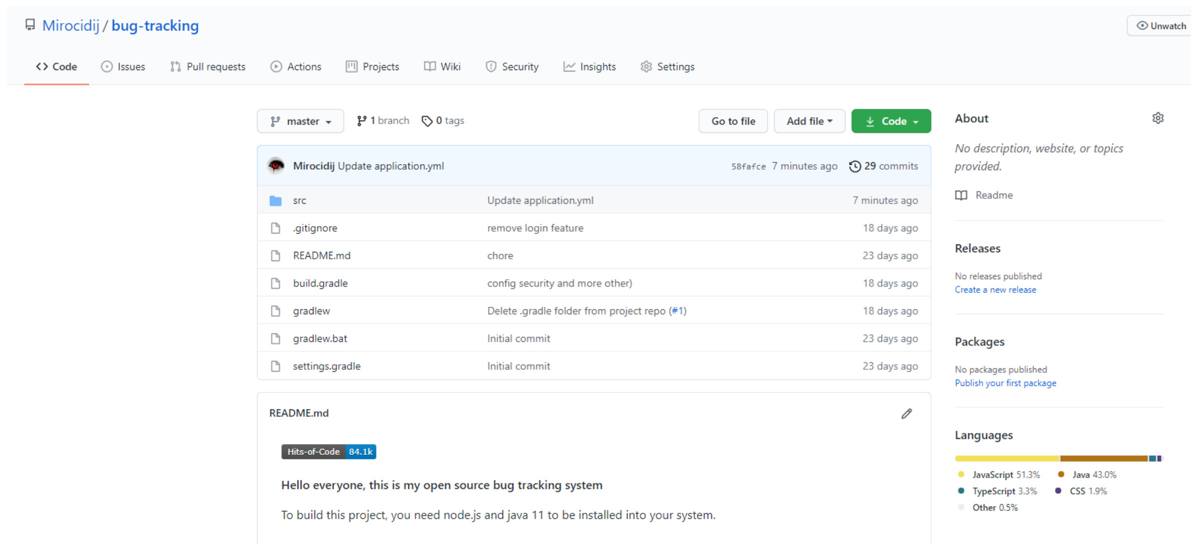


Рисунок 11 – Репозиторий проекта на GitHub

Вся работа над проектом ведется через удаленный репозиторий, что позволяет получить доступ к наработкам из любого места, где есть доступ в интернет.

Одной из особых полезных функций данного сервиса является удобный интерфейс для работы с Pull Request’s, что дает много возможностей при командной работе над проектом. Данный функционал позволяет создать отдельную ветку от основной, в которой будет реализован новый функционал, или исправлена какая-нибудь ошибка. Затем будет создан Pull Request, который означает намерение слиться своим изменения с основной веткой. Это позволяет команде работать над проектами параллельно, не мешая друг другу.

Также GitHub предоставляет удобный интерфейс для работы с Git. Через него можно посмотреть историю коммитов. Пример страницы с историей коммитов показан на рисунке 12. Также данный сервис позволяет посмотреть, какие именно были внесены изменения в том или ином коммите, что бывает полезно при разработке. Также видно, кто именно сделал коммит, посмотреть хэш коммита, который может быть использован для того, чтобы вставить один из коммитов в свою ветку или откатить его.

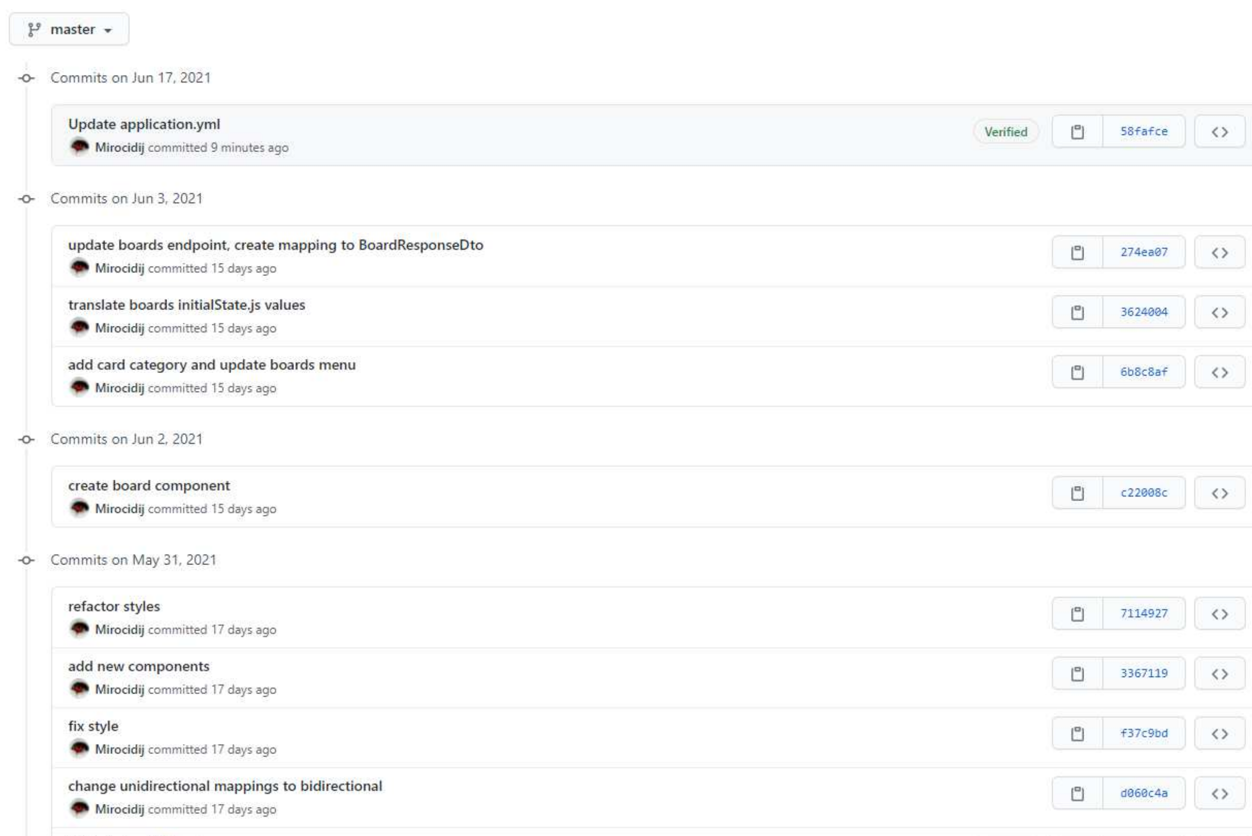


Рисунок 12 – История коммитов проекта

Пример изменений, внесенных в рамках одного из коммитов показан на рисунке 13. На данном рисунке можно увидеть, как строки были добавлены, удалены или же модифицированы.

Этот функционал очень часто используется при разработке, если нужно посмотреть, как выглядел код до внесенных изменений.

На данном рисунке показаны изменения файла application.yml, который отвечает за конфигурирование Spring Boot, подробнее его код будет разобран далее в соответствующем разделе.

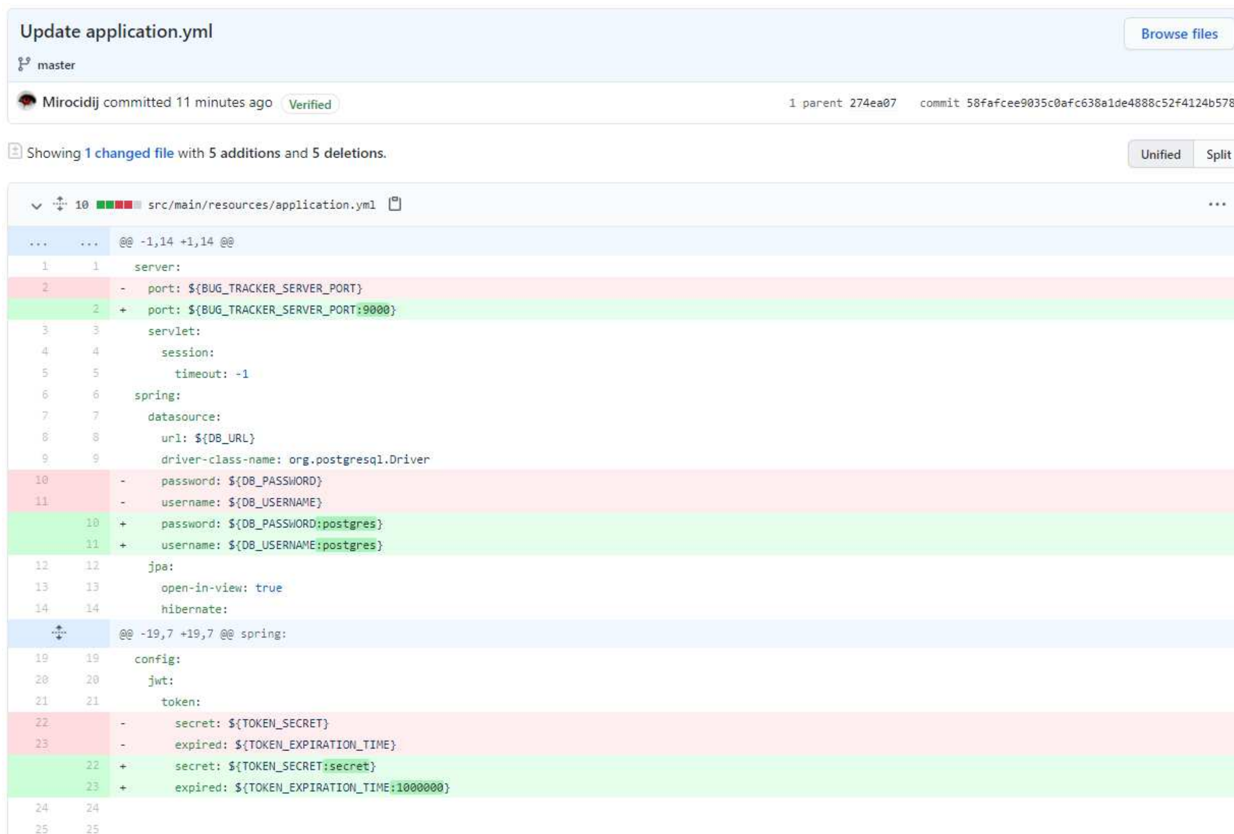


Рисунок 13 – Пример изменений из коммитов

2.4 Настройка сервера непрерывной интеграции

В качестве сервера непрерывной интеграции был выбран Jenkins. Jenkins – это популярный и простой инструмент для настройки CI/CD, также он является расширяемым, бесплатным и имеет открытый исходный код.

Для работы с Jenkins был написан скрипт, который запускается каждый раз, когда в репозиторий с проектом будет внесено изменение. Код данного скрипта показан на рисунке 14.

Данный код является конвейером, по которому будет идти Jenkins каждый раз, при внесенных изменениях. Конвейер состоит из шагов, каждый из которых означает какое-то полезное действие. Поскольку проект не большой, то и количество шагов тоже не большое.

В первом шаге Jenkins обращается к удаленному репозиторию проекта, забирает оттуда новую версию исходных файлов и проводит сборку проекта, чтобы убедиться, что в исходном коде, который попал в репозиторий, нет явных и грубых ошибок.

Второй шаг запускает тесты, результаты которых отдаются плагину junit для Jenkins, который обновляет статус сборки. В случае возникновения ошибок, сборка завершится неудачно и можно посмотреть, на каком именно шаге она произошла.

```groovy
#!groovy

properties([disableConcurrentBuilds()])

pipeline {
    agent any

    options {
        timestamps()
    }

    stages {
        stage('Build') {
            steps {
                git 'https://github.com/Mirocidij/Cerberus-server.git'
                sh './mvnw clean compile'
            }
        }

        stage('Test') {
            steps {
                sh './mvnw test'
            }

            post {
                always {
                    junit '**/target/surefire-reports/TEST-*.xml'
                }
            }
        }
    }
}
```

Рисунок 14 – Исходный код Jenkins конвейера

Для того, чтобы Jenkins мог понять, что в репозитории произошли изменения есть много стратегий.

Первая из них подразумевает, что Jenkins будет с определенных интервалом времени обращаться к репозиторию и проверять, нет ли там изменений. Данная стратегия не очень хорошо тем, что при редко вносимых изменениях, Jenkins будет совершать много бесполезных действий.

Вторая стратегия подразумевает, что GitHub будет сам посылать в Jenkins запрос о том, что в определенном репозитории произошли изменения.

Данный подход позволяет оптимизировать ресурсы сервера непрерывной интеграции, так как он будет обращаться к репозиторию только тогда, когда она сам об этом попросит.

В рамках данного проекта был выбран второй способ. Для его реализации необходимо настроить GitHub Webhooks для данного проекта.

Для этого необходимо перейти в настройки репозиторию во вкладку Webhooks и создать новый Webhook, в котором указать URL-адрес Jenkins сервера и формат, в котором необходимо отправлять данные.

Скриншот данной настройки показан на рисунке 15.

Рисунок 15 – Настройка Webhook'а для проекта

В качестве URL был введен адрес сервера, на котором развернут Jenkins сервер, в данном случае – http://jenkins.yacudza.ru/github-webhook/. В качестве формата данных был выбран json. После данной настройки при каждом коммите в репозиторий будет запускать конвейер Jenkins для данного проекта.

2.5 Разраб
отка

безопа
сного
механи
зма
автори
зации
пользо
вателе
й

2.5.1 Описание процесса аутентификации

При попытке отправки HTTP-запроса на одну из конечных точек нашего сервера, сервер будет проверять, присутствует ли в данном запросе заголовок «Authorization», который по логике нашего сервера должен содержать внутри себя токен авторизации. При отсутствии токена или при его недействительности будет происходить перенаправление на форму авторизации, на которой пользователю будет предложено ввести свои логин и пароль. Очевидно, что при такой модели никто и не сможет авторизоваться, ведь даже попытка получения токена будет заблокирована из-за его отсутствия в запросе, поэтому одна из конечных точек должна быть публичной.

При отправке логина и пароля на публичный адрес, сервер сначала произведет идентификацию пользователя при помощи логина. Если в базе действительно будет присутствовать пользователь с конкретным логином, то начнется процесс аутентификации, во время которого будут сравниваться отправленный пароль и действительный пароль пользователя. При успешном прохождении аутентификации, на основе личных данных пользователя, будет сгенерирован новый JWT-токен со сроком годности в одну неделю. Данный токен будет возвращен клиентской части, которая сохранит его в куки и будет активно использовать при отправке новых запросов, подставляя данный токен в

HTTP-заголовок «Authorization». Схема данного вида авторизации изображена на рисунке 16 [3].

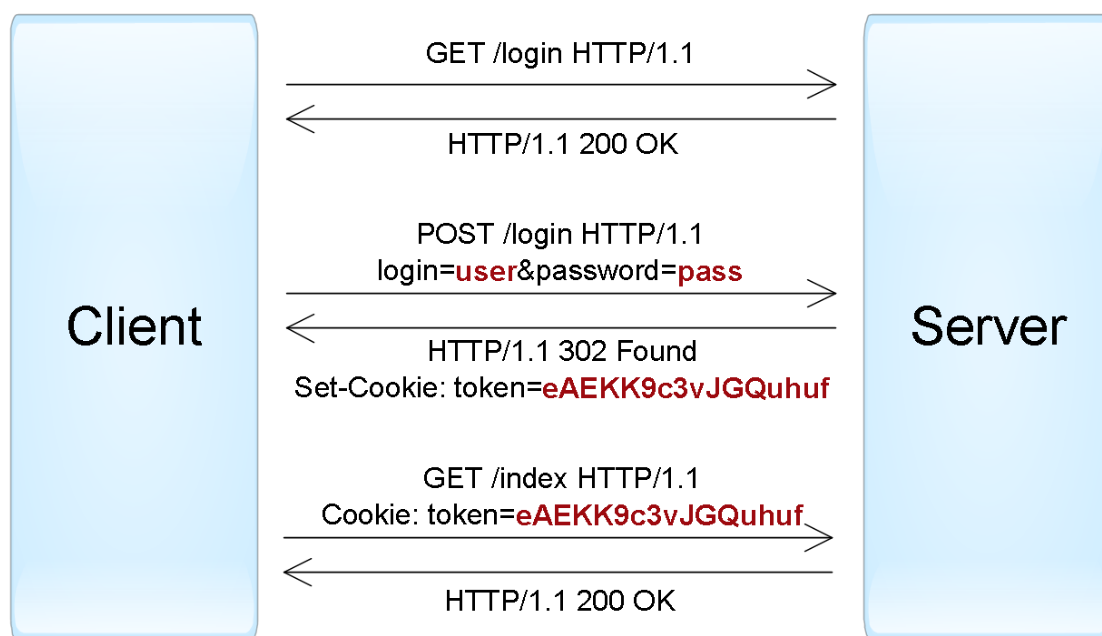


Рисунок 16 – Схема авторизации посредством токена

2.5.2 Реализация процесса аутентификации

Современные веб-клиенты разрабатываются на языке JavaScript. Все запросы на сервер отправляются также посредством языка JavaScript. Поскольку это не очень безопасно, в браузеры по умолчанию встроена защита, запрещающая кросс-доменные запросы при помощи данного языка.

Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент. Говорят, что агент пользователя делает запрос с другого источника (cross-origin HTTP request), если источник текущего документа отличается от запрашиваемого ресурса доменом, протоколом или портом.

По умолчанию CORS выключен, поэтому при отправке запроса на наш

сервер произойдет ошибка. Чтобы этого избежать включим поддержку кросс-доменных запросов и настроим ее. Пример подобной настройки в Spring Boot для адреса, на котором предполагаемо будет развернут сервер клиентской части приведен ниже. После данных настроек браузер пользователя, который зашел на клиентскую часть приложения сможет отправлять запрос на наш сервер и браузер его не заблокирует.

Исходный код класса WebConfig:

```
@Configuration
public class WebConfig implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:3000")
                .allowedMethods("*");
    }
}
```

2.5.3 Генерация и валидация токена

Следующим шагом будет разработка специального класса, который мог бы генерировать токены в нашем приложении используя для этого какой-нибудь секретный ключ и имя пользователя. Секретный ключ будет храниться в конфигурационных файлах и отделен от исходного кода, это позволит легко менять его без перекомпиляции приложения, если возникнет такая потребность. Код класса JwtTokenProvider показан ниже.

Исходный код класса JwtTokenProvider:

```
@Component
public class JwtTokenProvider {
    @Value("${jwt.token.secret}")
    private String secret;

    @Value("${jwt.token.expired}")
    private long validityInMilliseconds;

    @Autowired
    private UserDetailsService userDetailsService;
```

```java
    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @PostConstruct
    protected void init() {
        secret =
Base64.getEncoder().encodeToString(secret.getBytes());
    }

    public String createToken(
            String username,
            List<Role> roles) {

        Claims claims = Jwts.claims().setSubject(username);
        claims.put("roles", getRoleNames(roles));
        Date now = new Date();
        Date validity = new Date(now.getTime() +
validityInMilliseconds);

        return Jwts.builder()
                .setClaims(claims)
                .setIssuedAt(now)
                .setExpiration(validity)
                .signWith(SignatureAlgorithm.HS256, secret)
                .compact();
    }

    public Authentication getAuthentication(String token) {

        UserDetails userDetails = this
                .userDetailsService
                .loadUserByUsername(getUsername(token));

        return new UsernamePasswordAuthenticationToken(
                userDetails,
                "",
                userDetails.getAuthorities()
        );
    }

    public String getUsername(String token) {

        return Jwts.parser()
                .setSigningKey(secret)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
```

```java
    public String resolveToken(HttpServletRequest request) {

        var bearerToken = request.getHeader("Authorization");
        if (bearerToken != null &&
bearerToken.startsWith("Bearer_")) {

            return bearerToken.substring(7);
        }

        return null;
    }

    public boolean validateToken(String token) {

        try {
            var claims = Jwts.parser()
                    .setSigningKey(secret)
                    .parseClaimsJws(token);

            return !claims.getBody().getExpiration().before(new
Date());
        } catch (JwtException | IllegalArgumentException e) {
            var exception = new JwtAuthenticationException("JWT
token is expired or invalid");
            exception.addSuppressed(e);
            throw exception;
        }
    }

    private List<String> getRoleNames(List<Role> userRoles) {
        List<String> result = new ArrayList<>();
        userRoles.forEach(role -> result.add(role.getName()));

        return result;
    }
}
```

В приведенном выше коде в теле метода resolveToken делается проверка на присутствие в запросе заголовка Authorization. А в методе validateToken проверяется, не истек ли срок действия токена, который по умолчанию тоже берется из конфигурационных файлов.

2.5.4 Валидация запросов

После создания класса, который мог бы создавать и проверять токены,

нужно создать класс, который будет использовать JwtTokenProvider для проверки всех приходящих в приложение запросов, назовем его JwtTokenFilter. Код данного класса приведен ниже. Обратим внимание на то, что в нем используется ранее созданный нами класс JwtTokenProvider, а если конкретнее – вышеописанные методы resolveToken и validateToken.

Исходный код класса JwtTokenProvider:

```
public class JwtTokenFilter extends GenericFilterBean {

    private JwtTokenProvider jwtTokenProvider;

    public JwtTokenFilter(JwtTokenProvider jwtTokenProvider) {

        this.jwtTokenProvider = jwtTokenProvider;
    }

    @Override
    public void doFilter(ServletRequest servletRequest,
                         ServletResponse servletResponse,
                         FilterChain filterChain)
    throws IOException, ServletException {
        var token =
jwtTokenProvider.resolveToken((HttpServletRequest)
servletRequest);

        if (token != null &&
jwtTokenProvider.validateToken(token)) {
            Authentication authentication =
jwtTokenProvider.getAuthentication(token);

            if (authentication != null) {

SecurityContextHolder.getContext().setAuthentication(authenticatio
n);
            }
        }

        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

2.5.5 Настройка доступа к конечным точкам приложения

После написания инфраструктурной части приложения, необходимо

«сказать» Spring Security, чтобы он использовал для своей работы именно созданные нами классы. Для этого создадим еще один конфигурационный файл, на этот раз назовем его SecurityConfig. Доступ ко всем конечным точкам, адрес которых начинается на строку, которая сохранена в константе PUBLIC_ENDPOINT, является публичным и не требует авторизации. Доступ к конечным точкам администратора позволен только пользователю, у которого есть роль ADMIN, для обработки всех остальных запросов пользователь просто должен быть авторизован.

Исходный код класса SecurityConfig:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    private final JwtTokenProvider jwtTokenProvider;

    public static final String PUBLIC_ENDPOINT =
"/api/v1/auth/login";

    public static final String ADMIN_ENDPOINT =
"/api/v1/admin/**";

    @Autowired
    public SecurityConfig(JwtTokenProvider jwtTokenProvider) {
        this.jwtTokenProvider = jwtTokenProvider;
    }

    @Bean
    @Override
    protected AuthenticationManager authenticationManager() throws
Exception {

        return super.authenticationManager();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
                .httpBasic().disable()
                .csrf().disable()

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.S
TATELESS)
                .and()
                .authorizeRequests()
```

```
                .antMatchers(PUBLIC_ENDPOINT).permitAll()
                .antMatchers(ADMIN_ENDPOINT).hasRole("ADMIN")
                .anyRequest().authenticated()
                .and()
                .apply(new JwtConfigurer(jwtTokenProvider));
```

2.5.6 Разработка конечной точки аутентификации

После настройки безопасности внутри приложения, можно приступать непосредственно к разработке конечных точек, к которым будут обращаться клиентские приложения, такие как веб-приложения и мобильные-приложения. Конечная точка для получения данных пользователя будет доступна только администратору.

Прежде чем создать конечную точку для логина пользователя, создадим макет данных, которые хотим получать в эту конечную точку. В роли макета выступит Data Transfer Object класс, который будет содержать в себе только имя пользователя и пароль. Аннотация @Data нужна для того, чтобы библиотека Lombok сгенерировала методы-аксессоры для полей username и password. Такой подход позволяет нам не писать их вручную и не загромождать классы простым шаблонным кодом.

Модель данных, которая будет приходить в запросе на аутентификацию:

```
@Data
public class AuthenticationRequestDto {
    private String username;
    private String password;
}
```

Также нужно создать класс, объект которого будет возвращаться в ответ на запрос.

Исходный код класса UserDto:

```
@Data
@JsonIgnoreProperties
public class UserDto {
    private Long id;
    private String username;
```

```java
    private String firstname;
    private String lastname;
    private String email;
    private String token;

    public User toUser() {
        var user = new User();
        user.setId(id);
        user.setUsername(username);
        user.setFirstName(firstname);
        user.setLastName(lastname);
        user.setEmail(email);

        return user;
    }

    public static UserDto fromUser(User user) {
        UserDto userDto = new UserDto();
        userDto.setId(user.getId());
        userDto.setUsername(user.getUsername());
        userDto.setFirstname(user.getFirstName());
        userDto.setLastname(user.getLastName());
        userDto.setEmail(user.getEmail());

        return userDto;
    }
}
```

Следующим шагом создадим конечную точку по адресу /api/v1/auth/login, в которую и будет приходить объект данного класса, после чего мы будем производить аутентификацию пользователя и, возможно, даже вернем ему токен.

В терминологии серверной разработки, классы, которые содержат в себе конечные точки, принято называть контроллерами. В них и располагается основная бизнес-логика приложения. Создадим класс AuthenticationRestController и наполним его логикой аутентификации. Логика будет состоять всего из одной конечной точки, на которую клиентское приложение и будет отправлять запрос с данными. Поскольку запрос принимает в себе данные, и ожидается, что при обращении к данному запросу будет создаваться новая сущность – токен, логично оформить его в виде Post-запроса по протоколу HTTP.

Исходный код контроллера аутентификации:

```
@RestController
@RequestMapping("/api/v1/auth/")
public class AuthenticationRestControllerV1 {

    private AuthenticationManager authenticationManager;

    private JwtTokenProvider jwtTokenProvider;

    private UserService userService;

    @Autowired
    public AuthenticationRestControllerV1(
            AuthenticationManager authenticationManager,
            JwtTokenProvider jwtTokenProvider,
            UserService userService) {

        this.authenticationManager = authenticationManager;
        this.jwtTokenProvider = jwtTokenProvider;
        this.userService = userService;
    }

    @PostMapping("/login")
    public ResponseEntity<UserDto> login(@RequestBody
AuthenticationRequestDto requestDto) {
        try {
            String username = requestDto.getUsername();
            var namePassAuthToken = new
UsernamePasswordAuthenticationToken(
                    username,
                    requestDto.getPassword()
            );

            authenticationManager.authenticate(namePassAuthToken);

            var user = userService.findByUserName(username);
            if (user == null) {
                throw new UsernameNotFoundException(
                        "User with username: " + username + " +
not found");
            }

            String token  = jwtTokenProvider.createToken(username,
user.getRoles());

            var response = UserDto.fromUser(user);
            response.setToken(token);

            return ResponseEntity.ok(response);
        } catch (AuthenticationException e) {
```

```
                throw new BadCredentialsException("Invalid username or
password");
            }
        }
}
```

После запуска приложения, появится возможность обратиться по адресу /api/v1/auth/login.

Для того, чтобы получить доступ к системе, необходимо осуществить авторизацию, эту обязанность на себя целиком и полностью берет библиотека Spring Security, форма, которую она генерирует, показана на рисунке 17.

Рисунок 17 – Форма входа в систему

2.6 Конфигурация сервера

В первую очередь необходимо настроить Spring Boot. Для этого в проекте есть файл application.yml, который уже упоминался выше. Код данного файла показан на рисунке 18.

Данный файл написан в формате yaml, который достаточно удобен для написания различных конфигураций. В данном файле указан порт, на котором будет работать сервер, время жизни сессии пользователя, данные для подключения к базе данных, такие как адрес базы данных, имя класса-драйвера, пароль и имя пользователя в базе данных.

application.yml

```
server:
  port: ${BUG_TRACKER_SERVER_PORT:9000}
  servlet:
    session:
      timeout: -1
spring:
  datasource:
    url: ${DB_URL}
    driver-class-name: org.postgresql.Driver
    password: ${DB_PASSWORD:postgres}
    username: ${DB_USERNAME:postgres}
  jpa:
    open-in-view: true
    hibernate:
      ddl-auto: update
    show-sql: true

# My config variables
config:
  jwt:
    token:
      secret: ${TOKEN_SECRET:secret}
      expired: ${TOKEN_EXPIRATION_TIME:1000000}
```

Рисунок 18 – Код файла application.yml

Также в данном файле указываются данные для создания JWT-токенов, такие как секрет и время жизни токена. Можно заметить, что многие параметры заданы не явно. Они берутся из переменных среды операционной системы. Для того, чтобы данный проект запустился без ошибок, необходимо их предварительно задать.

Помимо всех надстроек, что были выполнены выше, необходимо сконфигурировать библиотеку Spring Security, внутри которой будет происходить вся «магия» авторизации пользователей, контроль времени жизни сессий приложения и прочие полезные вещи, необходимые при разработке веб приложений. Благодаря конфигурации приложение станет безопасным и не пустит незваных гостей к своим ресурсам.

Код конфигурации библиотеки:

```
package com.github.mirocidij.bugtracking.config;

import
com.github.mirocidij.bugtracking.security.jwt.JwtTokenProvider;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import
org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.W
ebSecurityConfigurerAdapter;
import
org.springframework.web.servlet.config.annotation.CorsRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer
@Configuration
@RequiredArgsConstructor
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter
```

```
implements WebMvcConfigurer {

    private static final String[] ADMIN_ENDPOINTS = {
            "/admin/**",
            "/api/v1/admin/**"
    };

//    private final JwtTokenProvider jwtTokenProvider;

    @Bean
    @Override
    protected AuthenticationManager authenticationManager() throws
Exception {
        return super.authenticationManager();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()

.antMatchers(ADMIN_ENDPOINTS).hasAuthority("ADMIN")
                .anyRequest().authenticated()
                .and()
                .formLogin();
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("http://localhost:9000/**")
                .allowedMethods("*");
    }

}
```

Для того, чтобы библиотека Spring Security смогла взаимодействовать с данными пользователей из базы данных, необходимо «научить» ее правильно извлекать оттуда данные и преобразовывать в понятный для Spring Security формат. Для этого был реализован специальный интерфейс UserDetailService, который содержит внутри себя всего один метод – loadUserByUsername, внутри которого и будет производится доступ к данным пользователей из базы данных и их преобразование согласно стратегии нашего приложения. Также класс, который реализует данный интерфейс, необходимо сделать бином в нашем приложении, для того чтобы при запуске Spring Security смог неявно получить

к нему доступ из контекста Spring Framework.

Код реализации данного интерфейса:

```
package com.github.mirocidij.bugtracking.security;

import com.github.mirocidij.bugtracking.domain.model.user.User;
import com.github.mirocidij.bugtracking.security.jwt.JwtUser;
import
com.github.mirocidij.bugtracking.security.jwt.JwtUserFactory;
import com.github.mirocidij.bugtracking.service.UserService;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExce
ption;
import org.springframework.stereotype.Service;

@Service
@Slf4j
@RequiredArgsConstructor
public class JwtUserDetailService implements UserDetailsService {

    private final UserService userService;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        User user = userService.findByUserNameWithRoles(username);

        if (user == null) {
            throw new UsernameNotFoundException("User wit
username: " + username + " not found");
        }

        JwtUser jwtUser = JwtUserFactory.create(user);

        log.info("IN loadUserByUsername - user with username: {}
successfully loaded", username);

        return jwtUser;
    }
```

Для того, чтобы упростить развертывание системы, было принято запаковывать серверную часть вместе с клиентской в один .jar-файл. Это

позволит буквально одной командой запускать данное приложение на компьютере, который будет выступать в роли сервера в закрытых локальных сетях. Поскольку доступ к статическим файлам приложения будет осуществляться через веб сервер Apache Tomcat, который встроен в Spring Boot, то нужно произвести некоторую настройку, которая позволит переадресовывать все неизвестные серверу запросы к клиентской части. Иными словами, необходимо помочь приложению построить маршруты.

Исходный код класса-маршрутизатора:

```
package com.github.mirocidij.bugtracking.controller;

import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
@Slf4j
public class RouteController {

    @RequestMapping(value = "/{path:[^\\.]*}")
    public String redirect() {
        log.info("In " + this.getClass().getSimpleName());

        return "forward:/";
    }

}
```

В данном контроллере есть всего один метод redirect, который будет вызываться каждый раз, когда запрос пользователя в браузере будет совпадать с некоторым паттерном, который указан в аннотации RequestMapping при помощи регулярного выражения. Данный метод делает forward перенаправления запроса к корню нашего приложения, на котором и доступна клиентская часть, причем клиентская часть получит полный адрес запроса и сможет отрисовать правильную страницу.

2.7 Опред

еление

моделе
й
данны
х

Из-за различий в модели хранения данных между Java и реляционными базами данных была применена библиотека, реализующая паттерн ORM, которая позволяет получать табличные данные и на их основе создавать привычные для ООП-парадигмы объекты, отражающие эти данные. Данные классы называются модели, их единственная задача — это правильная репрезентация табличных данных в контексте программы на Java.

Поскольку все сущности в приложении должны иметь некоторые служебные поля, вроде статуса, даты создания и даты последнего обновления, а также уникальные идентификаторы, имеет смысл вынести определении этих полей в отдельный класс, от которого будут наследоваться все остальные сущности в приложении. Данный класс будет параметризированным, чтобы можно было определить тип данных для поля id для каждого класса индивидуально. Данный класс был назван BaseEntity, или базовая сущность на русском.

Исходный код класса BaseEntity.

```
package com.github.mirocidij.bugtracking.domain.model;

import lombok.Data;
import lombok.EqualsAndHashCode;
import org.springframework.data.annotation.CreatedDate;
import org.springframework.data.annotation.LastModifiedDate;

import javax.persistence.*;
import java.io.Serializable;
import java.util.Date;

@Data
@MappedSuperclass
@EqualsAndHashCode(of = "id")
public class BaseEntity<T extends Serializable> {
```

```java
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private T id;

    @CreatedDate
    @Column(name = "created_datetime")
    private Date createdDateTime;

    @LastModifiedDate
    @Column(name = "updated_datetime")
    private Date updateDateTime;

    @Enumerated(EnumType.STRING)
    @Column(name = "status")
    private Status status;

}
```

При рассмотрении данного кода можно увидеть, что классы, которые можно передавать в него в качестве параметра, должны реализовывать интерфейс Serializable. Также можно заметить, что данный класс имеет поле status типа Status. Status – это перечисление, которое определено в отдельно файле.

Исходный код перечисления Status:

```java
package com.github.mirocidij.bugtracking.domain.model;

public enum Status {
    ACTIVE, BANNED, DELETED
}
```

Самая главная сущность в приложении — это пользователь, для него был создан класс User. Для того, чтобы ORM могла понять, к какой именно таблице в базе относится данная модель, необходимо сделать ряд настроек. В первую очередь указывается аннотация Table, в которую в качестве параметра передается название таблицы в базе данных, далее над классом идут аннотации, не относящиеся к базе данных. Также специальные аннотации должны быть расставлены над полями класса, чтобы ORM могла сопоставить поля класса с колонками в таблице данных, это делается при помощи аннотации Column, в которую в качестве параметра передается название колонки.

Исходный код модели пользователя:

```
@Data
@Entity
@Table(name = "users")
@Accessors(chain = true)
@EqualsAndHashCode(callSuper = true)
@JsonIgnoreProperties
public class User extends BaseEntity<Long> {

    @Column(name = "username")
    private String username;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email")
    private String email;

    @Column(name = "password")
    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "user_roles",
            joinColumns = {@JoinColumn(name = "user_id",
referencedColumnName = "id")},
            inverseJoinColumns = {@JoinColumn(name = "role_id",
referencedColumnName = "id")})
    private List<Role> roles;

}
```

Рассмотрим поле roles, типа List<Role>. Это поле представляет собой список ролей пользователя, поскольку у каждого пользователя может быть много ролей и роли могут повторяться, то данное поле помечено аннотацией ManyToMany, чтобы указать ORM-библиотеке, что данное поле является ссылкой на другую таблицу со связью многие-ко-многим. А аннотация JoinTable указывается для того, чтобы указать название связующей таблицы и определить названия колонок, через которые происходит связь.

Также можно заметить, что класс данной модели наследуется от абстрактного класса BaseEntity, который принимает в себя тип Long,

обозначающий тип идентификатора. BaseEntity это стандартный служебный класс, который был создать во избежание повторения большого количества однотипного кода. Все дело в том, что каждая модель базы данных должна иметь поле id, обозначающее идентификатор сущности. Также в данном классе присутствует ряд других служебных полей, необходимых для корректной работы приложения, таких как дата создания, дата последнего обновления и статус записи.

Исходный код модели роли:

```
package com.github.mirocidij.bugtracking.domain.model.role;

import com.github.mirocidij.bugtracking.domain.model.BaseEntity;
import lombok.Data;
import lombok.EqualsAndHashCode;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;

@Data
@Entity
@Table(name = "roles")
@EqualsAndHashCode(callSuper = true)
public class Role extends BaseEntity<Long> {

    @Column(name = "name")
    private String name;

    @Override
    public String toString() {
        return "Role {" +
                "id: " + super.getId() + ", " +
                "name: " + name + " }";
    }

}
```

В данном фрагменте кода видно, что модель роли содержит только одно значимое поле – название.

Исходный код модели доски:

```
package com.github.mirocidij.bugtracking.domain.model.boards;
```

```java
import com.github.mirocidij.bugtracking.domain.model.BaseEntity;
import com.github.mirocidij.bugtracking.domain.model.user.User;
import lombok.Data;
import javax.persistence.*;
import java.util.List;

@Data
@Entity
@Table(name = "boards")
public class Board extends BaseEntity<Long> {

    private String boardTitle;

    private String boardDescription;

    private Integer backgroundColor;

    private String backgroundImageUrl;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "user_id")
    private User user;

    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
            name = "boards_users",
            joinColumns = { @JoinColumn(name = "board_id",
referencedColumnName = "id") },
            inverseJoinColumns = { @JoinColumn(name = "user_id",
referencedColumnName = "id") }
    )
    private List<User> users;

    @OneToMany(mappedBy = "board", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<CardList> cardLists;

}
```

В данном классе присутствует поле user типа User, которое хранит в себе ссылку на объект пользователя, который является создателем доски. Логика системы построена таким образом, что один пользователь может создать множество досок и логично было бы предположить, что это список досок должен быть внутри пользователя, а не наоборот, но все не так просто. Дело в том, что ORM сгенерировала и выполнила бы целых два запроса в случае, если бы мы сделали связь один-ко-многим от пользователя к доске, хотя можно было бы обойтись и одним при помощи связывания таблиц. Данная проблема

называется «Проблема N + 1 запроса». Одной из техник решения данной проблемы является создание связи ManyToOne, а также написание специального метода в интерфейсе репозитория, в котором будет произведено связывание, в результате при попытке получения всех досок для конкретного пользователя будет выполнен всего один запрос вместо двух.

Доска содержит список объектов типа CardList. CardList является абстракцией над колонкой доски.

Исходный код модели колонки:

```java
package com.github.mirocidij.bugtracking.domain.model.boards;

import com.github.mirocidij.bugtracking.domain.model.BaseEntity;
import lombok.Data;
import javax.persistence.*;
import java.util.List;

@Data
@Entity
@Table(name = "card_lists")
public class CardList extends BaseEntity<Long> {

    private String title;

    @OneToMany(mappedBy = "cardList", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Card> cards;

    @ManyToOne(fetch = FetchType.LAZY)
    private Board board;

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer serialNumber;

}
```

В данном фрагменте кода видно, что данный класс содержит обратную ссылку на объект доски, к которому он принадлежит. Также видно, что колонка содержит список объектов типа Card, который является моделью карточки.

Исходный код модели карточки:

```java
package com.github.mirocidij.bugtracking.domain.model.boards;
```

```java
import com.github.mirocidij.bugtracking.domain.model.BaseEntity;
import lombok.Data;

import javax.persistence.*;

@Data
@Entity
@Table(name = "cards")
public class Card extends BaseEntity<Long> {

    private String cardTitle;

    private String content;

    private BugStatus bugStatus;

    private BugSeverity bugSeverity;

    private Sgtring cardDescription;

    private User cardCreator;

    private List<User> userList;

    @ManyToOne(fetch = FetchType.LAZY)
    private CardList cardList;

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer serialNumber;

}
```

В коде данного класса видно, что карточка имеет такие поля как заголовок, содержимое, статус, приоритет, описание. Также данная карточка ссылается на пользователя создателя и имеет поле, хранящее список пользователей – людей, который работают над проблемой, которую описывает карточка. Карточка также имеет обратную ссылку на список колонок и порядковый номер в данном списке.

2.8 Разработка контроллеров

После того, как пользователь авторизуется в системе, он сразу же сможет получить доступ к странице со своими досками, которые он создал или в которые его пригласили другие пользователи.

Исходный код контроллера, который содержит внутри себя логику получения досок:

```
package com.github.mirocidij.bugtracking.controller;

import com.github.mirocidij.bugtracking.domain.dto.UserDto;
import
com.github.mirocidij.bugtracking.domain.dto.boards.BoardResponseDt
o;
import com.github.mirocidij.bugtracking.domain.model.boards.Board;
import
com.github.mirocidij.bugtracking.repository.boards.BoardRepo;
import com.github.mirocidij.bugtracking.security.jwt.JwtUser;
import lombok.RequiredArgsConstructor;
import org.modelmapper.ModelMapper;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import
org.springframework.security.core.annotation.AuthenticationPrincip
al;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;
import java.util.stream.Collectors;

@RestController
@RequestMapping("/api/v1/boards/")
@RequiredArgsConstructor
public class BoardController {

    private final ModelMapper modelMapper;

    private final BoardRepo boardRepo;

    @GetMapping("/all")
    public ResponseEntity<List<BoardResponseDto>> getAllBoards(
            @AuthenticationPrincipal JwtUser use
    ) {
        var boards = boardRepo

.findAllByUserUsernameOrderById(user.getUsername());
```

```
        if (boards.size() == 0) {
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }

        var ret = boards.stream()
                .map(board -> {
                    var boardResponseDto = modelMapper.map(
                            board,
                            BoardResponseDto.class
                    );
                    var userDto = modelMapper
                            .map(
                                    board.getUser(),
                                    UserDto.clas
                            );
                    boardResponseDto.setUserDto(userDto);

                    return boardResponseDto;
                })
                .collect(Collectors.toList());

        return new ResponseEntity<>(ret, HttpStatus.OK);
    }

}
```

Данный контроллер содержит в себе такие поля, как ModelMapper и BoardRepo. ModelMapper это класс одноименно библиотеки ModelMapper, которая служит для преобразования объектов одного класса в объекты другого класса. BoardRepo это имплементация паттерна репозиторий, который облегчает доступ к базе данных.

В данном классе присутствует метод getAllBoards, который принимает в себя объект авторизованного пользователя, использую имя которого мы можем получить доски из репозитория, которые доступны только конкретному пользователю. После получения досок происходит некоторое преобразование посредством библиотеки ModelMapper, а затем результат преобразования возвращается в виде HTTP-ответа со статусом OK.

Преобразование посредством библиотеки ModelMapper необходимо для того, чтобы избежать рекурсии при сериализации объектов в формат JSON, так как они могут содержать циклические ссылки друг на друга.

Исходный код интерфейса BoardRepo:

```java
package com.github.mirocidij.bugtracking.repository.boards;

import com.github.mirocidij.bugtracking.domain.model.boards.Board;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;
import java.util.Optional;

public interface BoardRepo extends JpaRepository<Board, Long> {

    List<Board> findAllByUserId(Long userId);

    @Query("select b from Board b join fetch b.user where b.id =
:id")
    Optional<Board> findWithJoinFetch(Long id);

    List<Board> findAllByUserUsernameOrderById(String username);

}
```

В данном классе присутствует объявление необходимых методов, на основании наименования которых ORM будет генерировать запросы к базе данных.

2.9 Разработка клиентской части ИС

Чтобы создать JS проект, необходимо открыть терминал в папке с проектом и ввести команду «npm init». Данная команда создаст пару файлов и папок, которые необходимы для проекта, в частности файл package.json. В данном файле будут хранить скрипты для запуска и сборки проекта, а также в данном файле будут указаны все зависимости проекта.

Исходный код файла package.json.

```json
{
  "name": "bug-tracking",
  "version": "1.0.0",
  "private": true,
  "repository": "https://github.com/Mirocidij/bug-tracking.git",
  "author": "Yatsutko Sergey <s3rzh@inbox.ru>",
  "license": "MIT",
  "scripts": {
    "dev": "cross-env NODE_ENV=development webpack --mode
development",
    "build": "cross-env NODE_ENV=production webpack --mode
production",
    "watch": "cross-env NODE_ENV=development webpack --mode
development --watch",
    "start": "cross-env NODE_ENV=development webpack serve --mode
development --open"
  },
  "dependencies": {
    "@atlaskit/css-reset": "^6.0.5",
    "@babel/polyfill": "^7.12.1",
    "axios": "^0.21.1",
    "bootstrap": "4.6.0",
    "lodash": "^4.17.21",
    "normalize.css": "^8.0.1",
    "react": "^16.8.0",
    "react-beautiful-dnd": "^13.1.0",
    "react-bootstrap": "^1.6.0",
    "react-dom": "^16.8.5",
    "react-redux": "^7.2.4",
    "react-router": "^5.2.0",
    "react-router-dom": "^5.2.0",
    "react-scripts": "4.0.3",
    "redux": "^4.1.0",
    "redux-thunk": "^2.3.0",
    "styled-components": "^5.3.0"
  },
  "devDependencies": {
    "@babel/core": "^7.14.3",
    "@babel/plugin-proposal-class-properties": "^7.13.0",
    "@babel/preset-env": "^7.14.2",
    "@babel/preset-react": "^7.13.13",
    "@babel/preset-typescript": "^7.13.0",
    "babel-loader": "^8.2.2",
    "clean-webpack-plugin": "^4.0.0-alpha.0",
    "copy-webpack-plugin": "^9.0.0",
    "cross-env": "^7.0.3",
    "css-loader": "^5.2.6",
    "css-minimizer-webpack-plugin": "^3.0.0",
    "eslint": "^7.27.0",
```

```
    "file-loader": "^6.2.0",
    "html-webpack-plugin": "^5.3.1",
    "less": "^4.1.1",
    "less-loader": "^9.0.0",
    "mini-css-extract-plugin": "^1.6.0",
    "style-loader": "^2.0.0",
    "terser-webpack-plugin": "^5.1.2",
    "typescript": "^4.3.2",
    "webpack": "^5.37.1",
    "webpack-cli": "^4.7.0",
    "webpack-dev-server": "^3.11.2"
  },
  "browserslist": "> 0.25%, not dead"
}
```

Данный файл написан в формате JSON, как следует из расширения файла. Особое внимание следует уделить таким полям, как dependencies и devDependencies. Данные поля являются объектами, который содержат библиотеки, используемые в проекте, то есть – зависимости проекта. В поле dependencies содержатся те зависимости, которые попадут в финальную сборку, в поле devDependencies только те зависимости, которые нужны на этапе разработки, в финальную сборку они включены не будут.

Можно заметить, что среди зависимостей для разработки есть webpack. WebPack – это сборщик модулей JavaScript с открытым исходным кодом. Он создан в первую очередь для JavaScript, но может преобразовывать внешние ресурсы, такие как HTML, CSS и изображения, если включены соответствующие загрузчики. webpack принимает модули с зависимостями и генерирует статические ресурсы, представляющие эти модули.

Для того, чтобы добавить поддержку различных форматов при разработке необходимо настроить webpack.

Исходный код конфигурационного файла webpack.

```
const path = require('path')
const HTMLWebpackPlugin = require('html-webpack-plugin')
const {CleanWebpackPlugin} = require('clean-webpack-plugin')
const CopyWebpackPlugin = require('copy-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-
plugin')
```

```
const TerserWebpackPlugin = require('terser-webpack-plugin')

const isDev = process.env.NODE_ENV === 'development'
const isProd = !isDev

const optimization = () => {
    const config = {
        splitChunks: {
            chunks: "all"
        }
    }

    if (isProd) {
        config.minimizer = [
            new CssMinimizerWebpackPlugin(),
            new TerserWebpackPlugin()
        ]
    }

    return config
}

const filename = ext => isDev ? `[name].${ext}` :
`${ext}/[name].[fullhash].${ext}`

const cssLoaders = ext => {
    const loaders = [
        {
            loader: MiniCssExtractPlugin.loader
        },
        'css-loader'
    ]

    if (ext) {
        loaders.push(ext)
    }

    return loaders;
}

const fileLoader = () => ['file-loader']

const babelOptions = preset => {
    const options = {
        presets: [
            '@babel/preset-env'
        ],
        plugins: [
            '@babel/plugin-proposal-class-properties'
        ]
    }
```

```
    if (preset) {
        options.presets.push(preset)
    }

    return options
}

module.exports = {
    context: path.resolve(__dirname, 'src'),
    mode: 'development',
    devtool: isDev ? 'source-map' : 'nosources-source-map',
    entry: {
        main: ['@babel/polyfill', './index.jsx']
    },
    output: {
        filename: filename('js'),
        path: path.resolve(__dirname, '..', 'main', 'resources',
'static')
    },
    resolve: {
        extensions: ['.js', '.ts', '.png', '.css', '.less',
'.jsx', '.tsx']
    },
    optimization: optimization(),
    devServer: {},
    plugins: [
        new HTMLWebpackPlugin({
            template: './index.html',
            minify: {
                collapseWhitespace: isProd
            }
        }),
        new CleanWebpackPlugin(),
        new CopyWebpackPlugin({
            patterns: [
                {
                    from: path.resolve(__dirname,
'src/favicon.ico'),
                    to: path.resolve(__dirname, '..', 'main',
'resources', 'static')
                }
            ]
        }),
        new MiniCssExtractPlugin({
            filename: filename('css')
        })
    ],
    module: {
        rules: [
            {
                test: /\.css$/,
                use: cssLoaders()
```

```
            },
            {
                test: /\.less$/,
                use: cssLoaders('less-loader')
            },
            {
                test: /\.(png|jpg|svg|gif)$/,
                use: fileLoader()
            },
            {
                test: /\.(ttf|woff|woff2|eot)$/,
                use: fileLoader()
            },
            {
                test: /\.js$/,
                exclude: /node_modules/,
                use: {
                    loader: 'babel-loader',
                    options: babelOptions()
                }
            },
            {
                test: /\.ts$/,
                exclude: /node_modules/,
                use: {
                    loader: 'babel-loader',
                    options: babelOptions(['@babel/preset-
typescript'])
                }
            },
            {
                test: /\.jsx$/,
                exclude: /node_modules/,
                use: {
                    loader: 'babel-loader',
                    options: babelOptions([
                        '@babel/preset-react'
                    ])
                }
            }
        ]
    }
}
```

В данном коде указана директория с исходными кодами веб-приложения, указан путь до директории, в которую должна попасть сборка проекта. Также подключен ряд плагинов и модулей, которые позволяют добиться различного поведения, такого как понимание специальных форматов данных, оптимизация

кода для более быстрой загрузки приложения и компиляцию кода из специфичных языках программирования, которые не поддерживаются браузерами, но которые удобны при разработке.

После реализации базового сервера был разработан клиент, который во время финальной сборки упаковывается вместе с сервером. Код клиента написан на языке программирования JavaScript, с использованием библиотеки React и сопутствующих технологий. Библиотека React использует компонентный подход, каждый из компонентов можно встраивать внутрь других компонентов.

Для того, чтобы клиентская часть могла посылать HTTP-запросы на сервер была подключена библиотека axios. Также была написана обертка над этой библиотекой для более удобного взаимодействия с api. Обертка была названа HttpClient.

Исходный код класса HttpClient:

```
import axios from 'axios';

function createSession() {
    return axios.create({
        baseURL: "http://localhost:9000/api/v1/",
    });
}

class _HttpClient {
    private session: any

    constructor() {
        this.session = createSession();
    }

    get = (...params: any) =>
        this.session
            .get(...params)
            .catch((exception: any) => {
                throw exception;
            });

    post = (...params: any) => this.session.post(...params);

    put = (...params: any) => this.session.put(...params);
```

```
    delete = (...params: any) => this.session.delete(...params);
}

const HttpClient = new _HttpClient();

export default HttpClient;
```

После того, как данные будут получены, их нужно куда-то сохранять. В качестве хранилища данных была подключена библиотека Redux.

Исходный код конфигурации библиотеки Redux:

```
import { applyMiddleware, compose, createStore } from "redux";
import thunk from "redux-thunk";
import rootReducer from './rootReducer'
import { loadState, saveState } from "./loadState";

let persistedReduxStore = loadState();

const store = createStore(
  rootReducer,
  persistedReduxStore ? persistedReduxStore : {},
  compose(
    applyMiddleware(thunk)
  ));

store.subscribe(() => {
  saveState(store.getState())
})

export default store;
```

В данном фрагменте кода видно, что перед созданием хранилища вызывается метод loadState,

Данный метод нужен, чтобы проверить, нет ли локальном хранилище браузера данных, которые уже могли быть загружены ранее. Также можно заметить строку, в которой приложение подписывается на store и при каждом его обновлении вызывает метод saveState, который выполняет сохранение данных в localStorage браузера. Это нужно для того, чтобы некоторые данные могли храниться в браузере пользователя

Исходный код методов loadState и saveState:

```
export const loadState = () => {
  try {
    const serializedState =
localStorage.getItem('persistedReduxStore');
    if (serializedState === null) {
      return undefined;
    }

    return JSON.parse(serializedState);
  } catch (err) {
    return undefined;
  }
}

export const saveState = (state) => {
  try {
    const stateToSave = {
      users: state.users,
      boards: state.boards,
      board: state.board
    }

    const serializedState = JSON.stringify(stateToSave);
    localStorage.setItem('persistedReduxStore', serializedState);
  } catch (err) {
    // Ignore writes errors
  }
}
```

Код компонента главной страницы со всеми главными маршрутами:

```
const MainContainer = styled.div`
  height: 100vh;
  width: 100%;
  display: flex;
  flex-direction: column;
  background-color: #FAFBFC;

  ${props => props.backgroundIsPicture
          ? `background-image: url(${props.backgroundUrl});`
          : 'background-image: none'}

  background-size: 100% 100%;
`;

const ContentContainer = styled.div`
  padding-top: 8px;
  flex-grow: 1;
  position: relative;
  outline: none;
```

```
`;

const App = ({ backgroundIsPicture, backgroundUrl }) => {
  return (
    <MainContainer
      backgroundIsPicture={backgroundIsPicture}
      backgroundUrl={backgroundUrl}
      className="main-container"
    >
      <Header backgroundIsPicture={backgroundIsPicture}/>

      <ContentContainer>
        <Switch>
          <Route exact path="/" component={Board}/>
          <Route exact path="/users" component={Users}/>
          <Route exact path="/companies" component={Companies}/>
          <Route exact path="/boards" component={Boards}/>
          <Route exact path="/profile" component={Profile}/>
          <Route exact component={NoMatchPage}/>
        </Switch>
      </ContentContainer>
    </MainContainer>
  );
}

const mapStateToProps = (state) => ({
  ...state.app
})

const mapDispatchToProps = (dispatch) => ({
  actions: bindActionCreators({}, dispatch)
})

export default connect(mapStateToProps, mapDispatchToProps)(App);
```

В данном примере кода видно, что все приложение имеет один родительский компонент, которым является MainContainer, который является обычным блочным элементом со стандартными стилями, задающими ему размер на всю ширину и высоту экрана. Далее в этот блок вложен другой компонент – Header, этот компонент отвечает за навигацию по приложению, в нем расположены пункты меню и ссылки на различные маршруты.

Следующим блоком является ContentContainer, в нем располагается весь контент сайта. Компонент каждой страницы отрисовывается именно внутри данного блока, в то время как полоса навигации статична и не меняется от страницы к странице. Данный подход создания приложений называется SPA –

Single Page Application, или одностраничное приложение, если переводить на русский. Основная концепция данной идеи заключается в том, что в приложении нет отдельных html страниц под каждый маршрут – вместо этого есть одна страницы с блоком, наподобие ContentContainer, у которого просто меняется содержимое. Это удобно, поскольку при переходе на новый маршрут не происходит перезагрузки страницы, как при стандартном подходе, что дает ряд преимуществ, например возможность хранить состояние приложение в виде одного большого хранилища данных, и можно не бояться, что оно обнулится при перезагрузке страницы, поскольку страницы не перезагружаются во время работы с системой.

При переходе в раздел меню «Мои доски» пользователь попадает на страницу, которая запрашивает с сервера список всех досок для авторизованного пользователя и показывает их ему в виде плиток. Скриншот данного раздела приложения показан на рисунке 19.

При нажатии на плитку одной из досок приложение переносит пользователя к содержимому конкретной доски. Поскольку сущность доски содержит сущность колонки, а сущность колонки содержит сущность задачи, то можно сказать, что доски имеют иерархическую структуру данных.

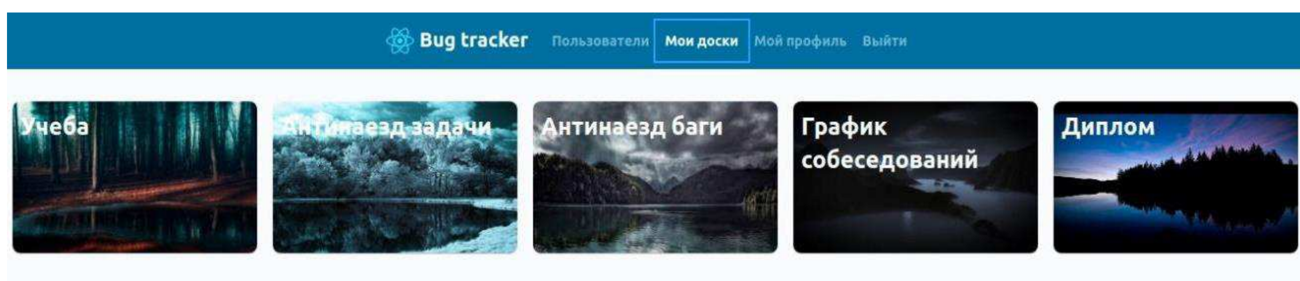


Рисунок 19 – Раздел приложения «Мои доски»

Для отрисовки содержимого досок был разработан специальный компонент, который при открытии в приложении посылает запрос на сервер, загружает всю информацию касательно конкретной доски показывает ее пользователю в виде интерактивных колонок.

Исходный код компонента, который выводится при нажатии на кнопку «Мои доски»:

```
const Container = styled.div`
  background-color: #fafbfc;
  display: flex;
  flex-direction: row;
  justify-content: flex-start;
  flex-wrap: wrap;
  padding: 8px;
  margin: 8px;
`;

const BoardTitle = styled.div`
  color: white;
  font-weight: bold;
  font-size: 170%;
`;

const BoardInfo = styled.div`
  margin: 8px;
  padding: 8px;
  width: 250px;
  height: 150px;
  border-radius: 8px;
  box-shadow: 0 1px 0 rgb(9 30 66 / 25%);
  display: block;
  background-color: rgba(0,0,0,.15);

  background-image: ${props => props.backgroundImage
          ? `url(${props.backgroundImage})`
          : 'none'
  };
  background-size: 100% 100%;

  &:hover {
    background-color: rgba(0, 0, 0, 0.25);
    cursor: pointer;
  }
`;

class UserBoards extends React.Component {
  componentDidMount() {
    this.props.actions.getAllBoards();
  }

  render() {
    const {
      boards,
      actions: {
```

```
          getAllBoards
        }
      } = this.props;

      return (
        <Container>
          {
            boards && boards.map(board => (
              <BoardInfo backgroundImage={board.backgroundImageUrl}
key={board.id}>
                <BoardTitle>{board.boardTitle}</BoardTitle>
              </BoardInfo>
            ))
          }
        </Container>
      );
    }
}

const mapStateToProps = (state) => ({
  boards: state.boards.boards
})

const mapDispatchToProps = (dispatch) => {
  return {
    actions: bindActionCreators({ ...actions }, dispatch)
  }
}

export default connect(mapStateToProps,
mapDispatchToProps)(UserBoards);
```

В коде класса данного компонента переопределен метод componentDidMount, который вызывается библиотекой React в тот момент, когда компонент вывелся на странице. В данном методе идет обращение к серверу, который отдает информацию о досках, которые доступны конкретному пользователю.

Концепция лямбда выражений пришла из мира функционального программирования. По своей сути лямбда выражения являются анонимными методами, которые мы можем легко передавать в качестве параметра в различные методы. В контексте стандарта языка JavaScript – ECMAScript6 данные выражения называются стрелочными функциями.

Исходный код компонента, ответственного за отрисовку доски:

```
const BoardWrapper = styled.div`
  margin: 4px;
  position: absolute;
  left: 0;
  right: 0;
  top: 0;
  bottom: 0;

  overflow-x: auto;
  overflow-y: hidden;
`;
const BoardMainContent = styled.div`
  height: 100%;
  display: flex;
  flex-direction: column;
  margin-right: 0;
  position: relative;
  transition: margin .1s ease-in;
`;
const BoardCanvas = styled.div`
  background: linear-gradient(180deg, rgba(0, 0, 0, .24) 0,
rgba(0, 0, 0, .24) 48px, transparent 80px, transparent);
`;
const Container = styled.div`
  user-select: none;
  white-space: nowrap;
  padding-bottom: 8px;
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
`;

class Board extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      active: false,
      currentTaskInModal: {},
      currentListIdInModal: 'column-0'
    }
  }

  componentDidMount() {
    this.props.actions.changeBackgroundState()
  }

  componentWillUnmount() {
    this.props.actions.changeBackgroundState()
```

```
  }

  render() {
    const {
      boardState: {
        boardDataIsLoading,
        tasks,
        columns,
        columnOrder,
        lastColumnId
      },
      actions: {
        loadBoardData,
        onDragEnd,
        saveNewCard,
        addNewColumn
      }
    } = this.props;

    return (
      <BoardWrapper>
        <DragDropContext
          onDragEnd={onDragEnd}
        >
          <Droppable
            droppableId="all-columns"
            direction="horizontal"
            type="column"
          >
            {(provided) => (
              <BoardMainContent>
                <BoardCanvas>
                  <Container
                    {...provided.droppableProps}
                    ref={provided.innerRef}
                  >
                    {columnOrder.map((columnId, index) => {
                      const columnProps = columns[columnId];
                      const tasksProps =
columnProps.tasksIds.map((taskId) => tasks[taskId]);
                      return (
                        <TaskListColumn
                          key={columnProps.id}
                          column={columnProps}
                          tasks={tasksProps}
                          index={index}
                          addNewCard={(active, task) => {
                            this.setState({
                              ...this.state,
                              active: active,
                              currentTaskInModal: {
                                ...task
```

```
                              },
                              currentListIdInModal: columnProps.id
                            });
                          }}
                        />
                      )
                    })}
                    {provided.placeholder}
                    <ColumnAdder
                      addNewColumn={addNewColumn}
                    />
                  </Container>
                </BoardCanvas>
              </BoardMainContent>
            )}
          </Droppable>
        </DragDropContext>
        <Modal
          show={this.state.active}
        >
          <Modal.Header closeButton onHide={() => this.setState({
active: false })}>
            <Modal.Title>
              {
                this.state.currentTaskInModal.id
                  ? "Редактирование карточки"
                  : "Создание карточки"
              }
            </Modal.Title>
          </Modal.Header>
          <Modal.Body>
            <CardForm
              handleSubmit={(newCardData) => {
                saveNewCard(
                  this.state.currentListIdInModal,
                  newCardData
                )

                this.setState({ active: false })
              }}
              cardProps={this.state.currentTaskInModal}
              handleClose={() => this.setState({ active: false })}
            />
          </Modal.Body>
        </Modal>
      </BoardWrapper>

    )
  }
}

const mapStateToProps = (state) => ({
```

```
  boardState: state.board
});

const mapDispatchToProps = (dispatch) => ({
  actions: bindActionCreators({ ...actions, changeBackgroundState
}, dispatch)
})

export default connect(mapStateToProps,
mapDispatchToProps)(Board);
```

В данном коде видно, что массив колонок преобразуется в компонент TaskListColumn посредством функции map.

Исходный код компонента TaskListColumn.

```
import React from "react";
import styled from "styled-components";
import Task from './Task'
import { Draggable, Droppable } from "react-beautiful-dnd";
import CardAdder from "./CardAdder";
import { Form } from "react-bootstrap";

const Container = styled.div`
  width: 272px;
  margin: 0 4px;
  height: 100%;
  box-sizing: border-box;
  display: inline-block;
  vertical-align: top;
  white-space: nowrap;
`;
const ColumnContent = styled.div`
  background-color: #ebecf0;;
  border-radius: 3px;
  box-sizing: border-box;
  display: flex;
  flex-direction: column;
  max-height: 100%;
  position: relative;
  white-space: normal;
`;
const Title = styled.h3`
  flex: 0 0 auto;
  padding: 10px 8px;
  position: relative;
  min-height: 20px;
`;
const TaskList = styled.div`
  flex: 1 1 auto;
```

```
  margin: 0 4px;
  padding: 0 4px;
  min-height: 20px;
  box-sizing: content-box;
`;

export default class TaskListColumn extends React.Component {
  render() {
    function getStyle(style, snapshot) {
      if (!snapshot.isDropAnimating) {
        return style;
      }
      return {
        ...style,
        // cannot be 0, but make it super tiny
        transitionDuration: `0.001s`,
      };
    }

    return (
      <Draggable
        draggableId={this.props.column.id}
        index={this.props.index}
      >
        {
          (provided, snapshot) => (
            <Container
              ref={provided.innerRef}
              {...provided.draggableProps}
              style={getStyle(provided.draggableProps.style,
snapshot)}
            >
              <ColumnContent>
                <Title {...provided.dragHandleProps}>
                  {this.props.column.title}
                </Title>
                <Droppable
                  droppableId={this.props.column.id}
                  type="task"
                >
                  {(provided, snapshot) => (
                    <TaskList
                      ref={provided.innerRef}
                      {...provided.droppableProps}
                      isDraggingOver={snapshot.isDraggingOver}
                    >
                      {
                        this.props.tasks.map((task, index) => {
                          return (
                            <Task
                              openModal={() =>
this.props.addNewCard(true, task)}
```

```
                              key={task.id}
                              task={task}
                              index={index}/>
                        )
                      })}
                    {provided.placeholder}
                  </TaskList>
                )}
              </Droppable>
              <div style={{
                padding: 8
              }}>
                <CardAdder
                  addNewCard={(newCard) => {
                    this.props.addNewCard(true, newCard);
                  }}
                />
              </div>
            </ColumnContent>
          </Container>
        )
      }
    </Draggable>
  )
```

В данном компоненте задаются стили для колонки, а также массив карточек, который передается в данный компонент в качестве свойств, преобразуется в массив компонентов типа Task посредством уже описанной выше функции map.

Исходный код компонента Task:

```
import React from 'react'
import styled from "styled-components";
import { Draggable } from "react-beautiful-dnd";

const Container = styled.div`
  width: auto;
  padding: 4px;
  background-color: #fff;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgb(9 30 66 / 25%);
  display: block;
  margin-bottom: 8px;
  min-height: 100px;
  position: relative;
  text-decoration: none;
```

```
  &:hover {
    background-color: rgba(0, 0, 0, 0.1);
    cursor: pointer;
  }
`;

const CardContent = styled.div`
  display: flex;
  flex-direction: column;
  justify-content: flex-start;
`;

const CardTitle = styled.div`

`;

function getColorByUrgency(urgency) {
  switch (urgency) {
    case 'Бессрочная': return 'green';
    case 'Желательно выполнить': return 'lightgreen';
    case 'Нужно выполнить': return 'yellow';
    case 'Срочная': return 'red';
    default:
      return 'transparent'
  }
}

const CardUrgency = styled.div`
  margin: 4px;
  width: 35px;
  height: 10px;
  border-radius: 5px;
  background-color: ${props => getColorByUrgency(props.urgency)};
`;

export default class Task extends React.Component {
  render() {
    function getStyle(style, snapshot) {
      if (!snapshot.isDropAnimating) {
        return style;
      }
      return {
        ...style,
        // cannot be 0, but make it super tiny
        transitionDuration: `0.001s`,
      };
    }

    return (
      <Draggable
        draggableId={this.props.task.id}
        index={this.props.index}
```

```
        >
          {(provided, snapshot) => (
            <Container
              onClick={this.props.openModal}
              ref={provided.innerRef}
              isDragging={snapshot.isDragging}
              {...provided.draggableProps}
              {...provided.dragHandleProps}
              style={getStyle(provided.draggableProps.style,
snapshot)}
            >
              <CardContent>
                {this.props.task.urgency && <CardUrgency
urgency={this.props.task.urgency} />}
                <CardTitle>{this.props.task.title}</CardTitle>
              </CardContent>
            </Container>
          )}
        </Draggable>
      )
    }
  }
```

Результатом корректной отработки данного кода станет страница с колонками и задачами в них. Скриншот данной страницы продемонстрирован на рисунке 20.

На рисунке 20 примечательно то, что для каждой доски пользователь может выбрать фоновое изображение, которое будет видеть каждый участник команды, работающей на конкретной доске, что не несет особых функциональных возможностей, но позволяет продемонстрировать дружелюбность системы к пользователю.

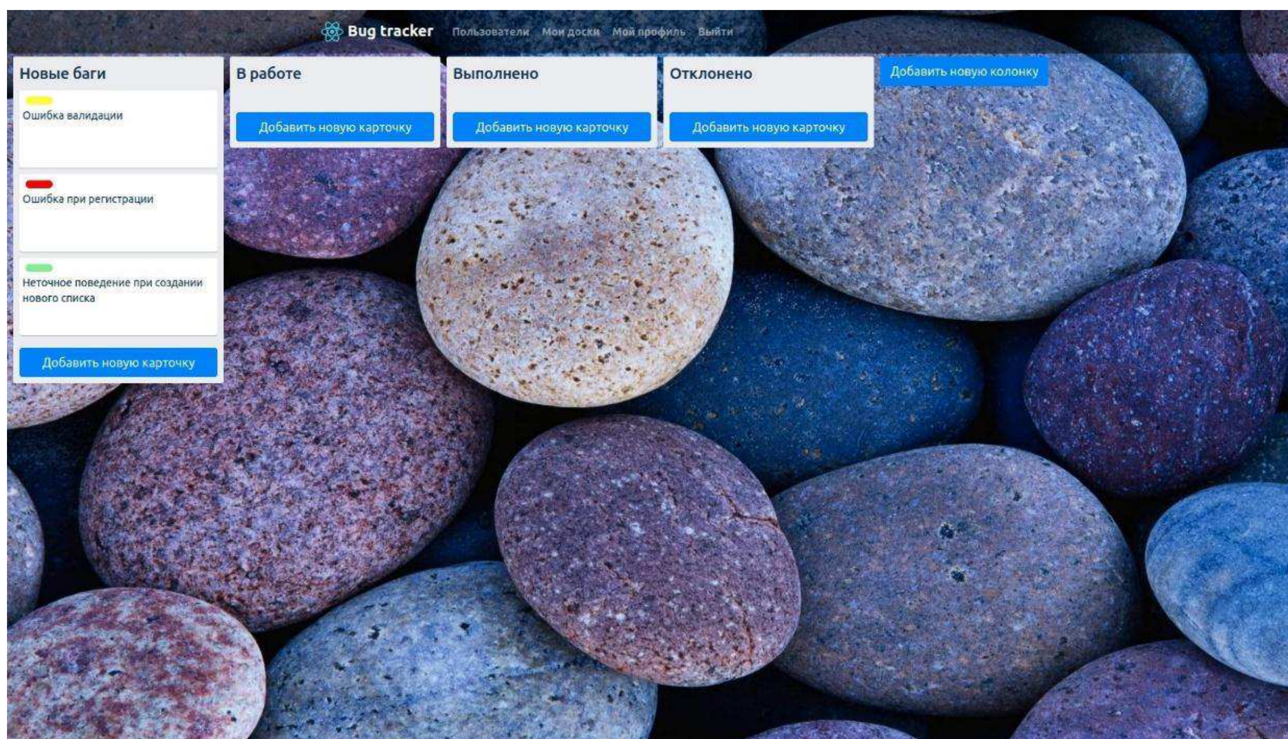


Рисунок 20 – Внешний вид одной из досок

Колонки по своей сути обозначают текущий статус, на котором находится ошибка. Также у ошибок могут быть различные категории срочности, в зависимости от которых на карточке будет отображен маркер специального цвета для лучшего визуального различия. Если нажать на одну из таких карточек, то откроется модальное окно с формой редактирования, в которой мы сможем изменить текущее состояние карточки. Скриншот данной формы изображен на рисунке 21.

При работе с данной формой мы имеем возможность сменить заголовок, описание, категорию и приоритет карточки, что повлияет на цветовой маркер.

Редактирование карточки

Заголовок:

Неточное поведение при создании нового списка

Описание:

Если на доске много колонок, что они не помещаются на один экран, и мы создаем новую колонку, то нижний скролл должен перемещаться в правый конец экрана, чтобы новосозданная колонка и кнопка создания новой колонки была видна.

Категория:

Без категории

Срочность:

Желательно выполнить

Закрыть Сохранить

Рисунок 21 – Форма редактирования карточки

Исходный код формы редактирования карточки:

```
import React, { useState } from "react";
import { Button, Form, Modal } from "react-bootstrap";

const CardForm = ({ cardProps, handleClose, handleSubmit }) => {

  const [title, setTitle] = useState(cardProps.title);
  const [description, setDescription] =
useState(cardProps.description);
  const [category, setCategory] = useState(cardProps.category);
  const [urgency, setUrgency] = useState(cardProps.urgency);

  let innerHandleSubmit = (e) => {
    e.preventDefault()
```

```
    handleSubmit({
      ...cardProps,
      title,
      description,
      category,
      urgency
    })
  }

  return (
    <Form onSubmit={innerHandleSubmit}>
      <Form.Group>
        <Form.Label>Заголовок:</Form.Label>
        <Form.Control
          value={title}
          onChange={(e) => setTitle(e.target.value)}
        />
      </Form.Group>

      <Form.Group>
        <Form.Label>Описание:</Form.Label>
        <Form.Control
          value={description}
          onChange={e => setDescription(e.target.value)}
          as="textarea"
          rows={6}
        />
      </Form.Group>

      <Form.Group>
        <Form.Label>Категория:</Form.Label>
        <Form.Control
          value={category}
          onChange={e => setCategory(e.target.value)}
          as="select"
        >
          <option>Без категории</option>
          <option>Баг на сервере</option>
          <option>Баг на клиенте</option>
          <option>Улучшение</option>
        </Form.Control>
      </Form.Group>

      <Form.Group>
        <Form.Label>Приоритет:</Form.Label>
        <Form.Control
          value={urgency}
          onChange={e => setUrgency(e.target.value)}
          as="select"
        >
          <option> </option>
```

```
          <option>Бессрочная</option>
          <option>Желательно выполнить</option>
          <option>Нужно выполнить</option>
          <option>Срочная</option>
        </Form.Control>
      </Form.Group>

      <Modal.Footer>
        <Button variant="secondary" onClick={handleClose}>
          Закрыть
        </Button>
        <Button type="submit" variant="primary">
          Сохранить
        </Button>
      </Modal.Footer>
    </Form>
  )
}

export default CardForm;
```

Код формы примечателен тем, что это не классовый компонент, а функциональный. Раньше в функциональных компонентах React нельзя было хранить состояние, но с появлением хуков – это стало возможно и именно они используются для того, чтобы сохранять состояние полей формы.

Как уже было сказано выше, данная доска является интерактивной. Это значит, что взаимодействие с ее элементами должно производиться при помощи мыши, посредством такие движений, как перетаскивание. Можно перетаскивать как сами колонки, так и карточки между колонками. Результаты данных манипуляций продемонстрированы на рисунках 22 и 23 соответственно.

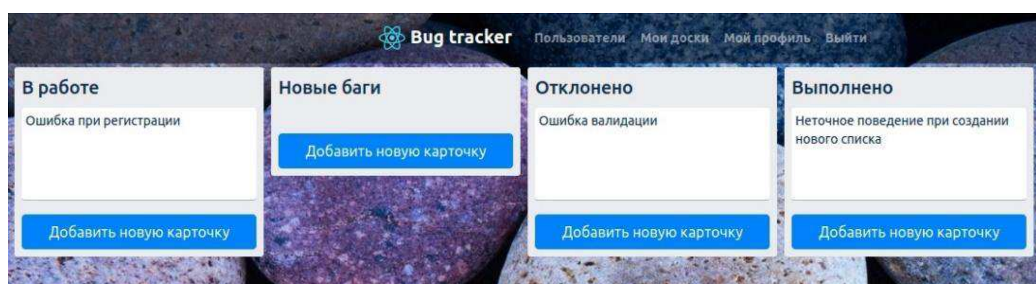


Рисунок 22 – Состояние доски после смены порядка следования колонок

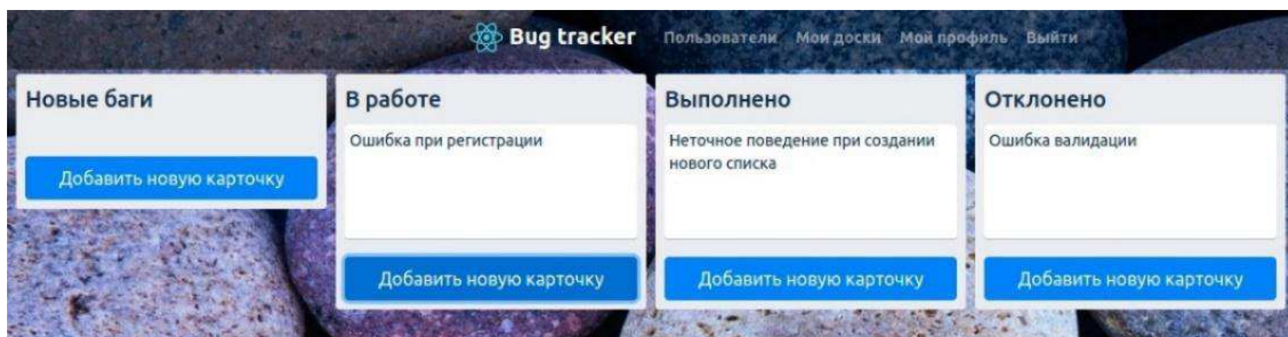


Рисунок 23 – Доска после перемещения карточек между колонками

Данный способ взаимодействия нелегко построить, но это того стоит, поскольку он крайне сильно повысит дружелюбность системы по отношению к новых пользователям, что позволит им тратить меньше времени на то, чтобы научиться ей пользоваться. Особенно это касается студентов, для которых данная система разрабатывается, ведь количество занятий ограниченно, и хотелось бы как можно скорее пройти процесс обучения работы в системе и приступить непосредственно к выполнению лабораторных работ.

2.10 Выводы по разделу «Описание разработки системы отслеживания

ошибо к в програ ммных продук тах»

Во втором разделе выпускной квалификационной работы выбран жизненный цикл разработки проекта, выполнено проектирование базы данных, приняты самые важные инфраструктурные решения, которые являются основой разрабатываемой системы. Также описаны основные процессы, связанные с разработкой системы.

В разработанной системе реализован процесс авторизации, аутентификации и генерации JWT-токенов по всем правилам безопасности. Также реализован процесс введения новых пользователей в систему, создание досок, колонок и карточек. Реализован процесс шифрования паролей при создании для них учетных записей. Эта мера предосторожности позволит сохранить конфиденциальные данные пользователей в случае, если злоумышленник сможет получить доступ к актуальной базе данных.

Полностью автоматизирован процесс сборки клиентского приложения, а также упаковка его в один файл с серверным приложением.

3 Оценка экономической эффективности разработки и внедрения системы отслеживания ошибок в программных продуктах

Совокупная стоимость владения или стоимость жизненного цикла — это общая величина целевых затрат, которые вынужден нести владелец с момента начала реализации вступления в состояние владения до момента выхода из состояния владения и исполнения владельцем полного объёма обязательств, связанных с владением.

3.1 Капитальные затраты

Капитальные затраты — это затраты на информационную систему, носящие разовый характер, приносят прибыль. Данные затраты не утрачиваются, а воспроизводятся. Затраты на разработку информационной системы вычисляются по формуле 1.

$$К=К_{пр} + К_{тс} + К_{лс} + К_{ло} + К_{ио} + К_{об} + К_{оэ}, \qquad (1)$$

где $К_{пр}$ – затраты на проектирование ИС;

$К_{тс}$ – затраты на технические средства управления;

$К_{лс}$ – затраты на создание линий связи локальных сетей;

$К_{ло}$ – затраты на программные средства;

$К_{ио}$ – затраты на формирование информационной базы;

$К_{об}$ – затраты на обучение персонала;

$К_{оэ}$ – затраты на опытную эксплуатацию.

3.2 Затраты на проектирование ИС

Затраты на проектирование ИС рассчитываются по формуле 2:

$$К_{пр}=К_{зп}+К_{кипс}+К_{свт}+К_{проч}, \qquad (2)$$

где $К_{зп}$ – затраты на заработную плату специалистов;

$К_{кипс}$ – затраты на инструментальные программные средства проектирования;

$К_{свт}$ – затраты на средства вычислительной техники для проектирования;

$К_{проч}$ – прочие затраты на проектирование.

Расчет заработной платы. Для разработки данной системы требуется 1 человек: программист, со знаниями языков программирования JavaScript, Java, SQL, фреймворка Spring Framework, библиотек ReactJS и Redux, СУБД PostgreSQL, а также со знаниями такого программного обеспечения как Jenkins и Git.

ЗП программиста – 12 792 руб., что соответствует МРОТ с 1 Января 2021 года. Работать программист будет 1 месяц. Срок работы разработчика – 22 полных дня. Расчет заработной платы представлен в таблице 3.

Таблица 3 – Расчет заработной платы

	Начислено, руб.	Удержано, руб.
По окладу	12 792	
Премия	1000	
СК	3 840	
РК	3 840	
НДФЛ		2 791
Итого	21 472	
На руки	18 681	

Общая заработная плата всех специалистов составляет:

$К_{зп}$ = 21 472 * 1,302 = 27 956 рублей.

Затраты на программное обеспечение. Затраты на ПО представлены в таблице 4

Таблица 4 – Затраты на ПО

№	Наименование ПО	Количество	Стоимость, руб	Срок службы
1	JetBrains All Products Pack	1	5 000	1 месяц

2	Ubuntu Linux	1	бесплатно	бессрочно
5	Google Chrome	1	бесплатно	3 года

Стоимость затрат на программное обеспечение будет включать в себя только стоимость пакета лицензионных сред разработки JetBrains All Products Pack.

$К_{ипс}$ = 5 000 рублей.

Расчет затрат на оборудование. При правильно эксплуатации техника прослужит 4 года. Расчет затрат на оборудование представлен в таблице 5.

Таблица 5 – Расчет затрат на оборудование

№	Наименование	Количество	Стоимость, руб.	Срок службы, лет.
1	Рабочее место пользователя			
2	Процессор INTEL Core i5 10400, LGA 1200, OEM [cm8070104290715s rh3c]	1	16 890	4
3	SSD накопитель KINGSTON A2000 SA2000M8/250G 250ГБ, M.2 2280, PCI-E x4, NVMe	1	3 690	4
4	Материнская плата MSI H410M-A PRO, LGA 1200, Intel H410, mATX, Ret	1	6 290	4
5	Модуль памяти PATRIOT PSD48G2400K DDR4 - 2x 4ГБ 2400, DIMM, Ret	1	4 096	4
6	Блок питания AEROCOOL AERO BRONZE, 550Вт, 120мм, черный, retail [aero bronze 550]	1	4 860	4
7	Монитор PHILIPS 273V7QDSB (00/01) 27", черный	1	10 990	4
	Итого:		46 816	

Поскольку данный компьютер будет использоваться только один месяц, то необходимо рассчитать амортизацию оборудования. Расчет амортизации компьютера за 1 месяц использования:

$К_{свт}$ = (46 816 / 4) / 12 = 975.

$К_{свт}$ = 975 рублей в месяц.

Расчет затрат на прочие расходы. На прочие затраты выделено 3% от затрат на проектирование. Это стандартный процент для прочих расходов, которого хватит на покрытие мелких проблем.

Расчет затрат на проектирование. Сумма затрат на проектирование представлена в таблице 6.

Таблица 6 – Сумма затрат на проектирование

Затраты	Сумма затрат, руб.
Заработная плата специалистов	27 956
Затраты на ПО	5 000
Затраты на средства вычислительной техники	975
Затраты на прочие расходы	1 018
Итого:	34 949

$К_{пр}$ = 34 949 рублей.

Расчет капитальных затрат. Капитальные затраты рассчитываются по формуле 3:

$$К = К_{пр}+К_{тс}+К_{лс}+К_{по}+К_{ио}+К_{об}+К_{оэ}, \quad (3)$$

где $К_{пр}$ – затраты на проектирование ИС;

$К_{тс}$ – затраты на технические средства управления;

$К_{лс}$ – затраты на создание линий связи локальных сетей;

$К_{по}$ – затраты на программные средства;

$К_{ио}$ – затраты на формирование информационной базы;

$К_{об}$ – затраты на обучение персонала;

$К_{оэ}$– затраты на опытную эксплуатацию;

$К_{пр}$ посчитано ранее и составляет 34 949 рублей;

$К_{тс}$ равны 0 руб., так как приложение будет работать на компьютерах института;

$К_{лс}$ равны 0 руб., так как при проектировании и разработке приложения используется локальный компьютер;

$К_{по}$ равны 0 руб., так как для работы приложения потребуется только бесплатный браузер;

$К_{ио}$ равны 0 руб., так как информационная база приложения создана на этапе проектирования;

$К_{об}$ – затраты на обучение персонала. Чтобы приложением можно было пользоваться так, как и задумано, требуется обучить преподавателя, который будет его использовать в рамках своих занятий. Обучением займется программист. Обучение займет 2 дня по 3 часа в день или один полный рабочий день. Поэтому $К_{об}$ это зарплата программиста за полный рабочий день или 20 472 / 22 * 1,302 = 1 211 рублей.

$К_{оэ}$– затраты на опытную эксплуатацию. Для того, чтобы все нормально работало, систему необходимо протестировать. Заниматься этим будет программист 1 рабочую неделю или 5 рабочих дней. Тестировать приложение программист будет неполный рабочий день – 3 часа в день, поэтому зарплата рассчитывается за 5 * 3 = 15 часов или 2 полных рабочих дня. ЗП программиста составляет 20 472 рублей в месяц (без надбавок).

ЗП программиста за период опытной эксплуатации равна:

20 472 / 22 * 2 * 1,302 = 2 423 рублей.

Расчет всех капитальных затрат представлен в таблице 7.

Таблица 7 – Расчет капитальных затрат

Вид затрат	Затраты	Сумма затрат, руб.
Затраты на проектирование	Заработная плата специалистов	29 956
	Затраты на программное обеспечение	5 000
	Затраты на оборудование	975
	Затраты на прочие расходы	1 018

Капитальные затраты	Затраты на технические средства управления	0
	Затраты на создание линий связи	0
	Затраты на программные средства	0
	Затраты на формирование информационной базы	0
	Затраты на обучение персонала	1 211
	Затраты на опытную эксплуатацию	2 423
Итого:		40 583

3.3 Эксплуатационные затраты

Расчет эксплуатационных затрат происходит по формуле 4:

$$C = C_{зп} + C_{ао} + C_{то} + C_{гс} + C_{ни} + C_{проч}, \quad (4)$$

где $C_{зп}$ – зарплата персонала, работающего с информационной системой;

$C_{ао}$ - амортизационные отчисления;

$C_{то}$ - затрата на техническое обслуживание;

$C_{лс}$ - затраты на использование глобальных сетей;

$C_{ни}$ - затраты на носитель информации.

Для поддержания приложения в рабочем состоянии необходимы затраты на заработную плату администратора, который периодически будет удалять из базы неактивные учетные записи пользователей, поскольку эта операция требуется всего раз в год, то можно выделить на это 500 рублей.

Также необходимы затраты на веб-сервер, на котором будет располагаться приложение. Хостер Fornex предоставляет очень выделенные тарифы на VPS сервера. На рисунке 24 показан список тарифов с ценой за месяц использования.

Для данного проекта подойдет самый дешевый тариф, с учетом того, что

администратор раз в год будет чистить базу данных. Поэтому стоимость веб-сервера составит 369 * 12 = 4 428 рублей.

Интернет входит в стоимость хостинга, поэтому затраты на интернет составят 0 рублей в год.

Рисунок 24 – Тарифы хостинга Fornex на выделенный сервер в месяц

Итого, эксплуатационные затраты составят 7 655 рублей за 1 год.

Список эксплуатационных затрат показан в таблице 8.

Таблица 8 – Список эксплуатационных затрат

Состав затрат	Планируемая сумма расходов в год, руб.
Затраты на заработную плату персонала	500
Затраты на амортизационные отчисления	0
Затраты на техническое оборудование	4 428
Затраты на использование глобальных сетей	0
Итого	4 928

Расчет рисков реализации. Невозможно знать все, что случится в

процессе реализации проекта, поэтому любые инвестиции будут сопряжены с риском. Для расчета рисков необходимо проанализировать создание системы поддержки принятия решений по трем основным группам рисков: риски инвестирования в разработку проекта, риски внедрения проекта, эксплуатационные риски.

Риски проекта представлены в таблице 9.

Мероприятия по снижению вероятности рисков, имеющих наивысшую оценку. При анализе рисков реализации проекта было найдено два важных риска, которые требуют детального разъяснения.

Риск несвоевременной сдачи системы обусловлен ее сложностью, трудно предусмотреть все необходимые функции на этапе описания системы.

Риск увеличенных расходов на разработку системы также обусловлен высокой сложностью разработки данной системы. Снизить данный риск можно при помощи написания и регулярного запуска юнит-тестов, а также при помощи тестирования работы данной системы на тестовом стенде.

Чтобы снизить риск несоответствия требованиям заказчика необходимо более подробно обсудить устройство разрабатываемой системы с заказчиком, обсудить все непонятные детали и подробно описать их в техническом задании.

Для снижения данных риском следует производить тестирование системы в процессе разработки после каждого цикла реализации нового функционала.

Таблица 9 – Риски проекта

№	Группы рисков	Перечень рисков проекта	Уровень влияния риска на проект	Вероятность риска	Возможность предотвращения или снижения риска
1.	Реализационные риски	Несвоевременная сдача проекта	Средний уровень	Низкая	Проработка технического задания, установка сроков сдачи базовых модулей разработки.
		Увеличенные расходы на разработку	Средний уровень	Низкая	Тщательное планирование разработки, тщательное изучение

					ТЗ.

Окончание таблицы 9

2.	Риск соответствия	Несоответствие требованиям заказчика	Низкий уровень	Низкая	Составление четкого и подробного технического задания.

3.2.1 Расчет затрат реализации проекта методом Total Cost of Ownership

Согласно методике TCO совокупная стоимость владения системой рассчитывается по следующей формуле:

$$TCO = DE + IC_1 + IC_2, \quad (5)$$

где DE (direct expenses) – прямые расходы;

IC_1 (indirect costs) – косвенные расходы первой группы;

IC_2 (indirect costs) – косвенный расходы второй группы;

Прямые расходы рассчитываются по формуле

$$DE=DE_1+DE_2+ DE_3+DE_4+ DE_5+DE_6+ DE_7+DE_8, \quad (6)$$

где DE_1 – капитальные затраты;

DE_2 – расходы на управление информационными технологиями;

DE_3 – расходы на техническую поддержку автоматизированного обеспечения и программного обеспечения;

DE_4 – расходы на разработку прикладного программного обеспечения внутренними силами;

DE_5 – расходы на аутсорсинг;

DE_6 – командировочные расходы;

DE_7 – расходы на услуги связи;

DE_8 – другие группы расходов.

Капитальные затраты были посчитаны ранее: DE_1 = 40 583 рублей.

Расходы на управление также были посчитаны ранее: $DE_2 = C_{зп}$ = 500 рублей.

Для данной системы DE3 = $C_{ао} + C_{то}$ = 0 + 4 428 = 4 428 рублей.

DE_4 = 0 рублей, так как приложение будет запускаться через браузер, который бесплатен для любого пользователя.

DE_5 = 0 рублей, так как при разработке и внедрении системы не использовалась помощь сторонних специалистов.

DE_6 = 0 рублей, так как для разработки системы не требуются командировочных расходов.

DE_7 = 0 рублей, так как расходы на интернет включены в тарифный план аренды сервера.

$DE_8 = C_{проч}$. $C_{проч}$ составляют 3% от всех прямых расходов. DE_8 = 1 401 руб.

TCO = 40 583 + 500 + 4 428 + 1 401 = 46 912 рублей.

3.4 Анализ рынка продуктов-аналогов. Установление стоимости программного

продук

та

Разрабатываемое в рамках данного проекта программное обеспечение является уникальным, поскольку на рынке не существует решений именно для изучения процесса тестирования информационных систем.

По этой причине будет проведен сравнительный анализ разрабатываемой ИС с программным обеспечением, которое используется в коммерческой разработке.

Сравнительный анализ показан в таблице 10.

Таблица 10 – Сравнительная таблица продуктов-аналогов

Критерии оценки	Система Jira Standart	Система YouTrack	Система Pivotal Tracker Standart	Приложение «Система отслеживания ошибок в программных продуктах»
Функционал	Система управления проектами, которое включает в себя доски, отслеживание проектов и заданий, шаблоны бизнес-проектов, отчеты и информационные панели, интеграции и журналирование.	Программное обеспечение для управления проектами и совместной работы, включает в себя баг-трекинг, agile-доски, базу знаний, составление отчетов, планирование, интеграции с другими системами и контроль времени за выполняемыми задачи.	Система баг-трекинга с функциями совместной работы, интеграции с различными системами.	Создаваемое приложение предоставляет функции системы баг-трекинга, совместной работы над проектами, доски, контроль времени, потраченного на исправление ошибки.

Оценка функционала	4/5	5/5	2/5	3/4
Дополнительное ПО	Веб-клиент работает через браузер	Веб-клиент работает через браузер	Веб-клиент работает через браузер	Веб-клиент работает через браузер
Минимальные системные требования	1 ГБ ОЗУ, Процессор 2-х ядерный с тактовой частотой 1,4 ГГЦ	1 ГБ ОЗУ, Процессор 2-х ядерный с тактовой частотой 1,4 ГГЦ	1 ГБ ОЗУ, Процессор 2-х ядерный с тактовой частотой 1,4 ГГЦ	1 ГБ ОЗУ, Процессор 2-х ядерный с тактовой частотой 1,4 ГГЦ
Стоимость за 25 человек	9 625 рублей в месяц	8 470 рублей в месяц	12 512 рублей в месяц	Стоимость на основе сравнения 3 500 рублей в месяц

Из таблицы выше видно, что системы-аналоги ничем не уступают разрабатываемой системе, а то и превосходят ее по функционалу. Но проблема в том, что весь функционал этих систем не нужен для выполнения лабораторных работ. Также большим недостатком систем-аналогов будет их цена, поскольку лицензионные версии этих программ стоят очень дорого.

Установление стоимости программного продукта.

1. Затратный метод.

Стоимость программного продукта рассчитывается на основании расчёта капитальных затрат, определяется себестоимость продукта, прибавляется прибыль равная 10% от себестоимости, прибавляется стоимость рекламы и продвижения продукта.

Капитальные затраты на создание системы отслеживания ошибок в программных продуктах

К = 40 583 рублей.

Прибыль в размере 10% от себестоимости:

П = 40 583 * 0,1 = 4 058 рублей.

Для данного приложения нет необходимости в рекламных акциях, так как оно разрабатывается исключительно для внутренних нужд ХТИ – филиала СФУ. Поэтому затраты на рекламу 0 рублей.

Итоговая стоимость программного продукта:

40 583 + 4 058 + 0 = 44 641 рубля.

2. Доходный метод.

Для того, чтобы определить стоимость данного программного продукта доходным методом, необходимо посчитать доходы, которые будет приносить данное веб-приложение. Система отслеживания ошибок в программных продуктах создается для удовлетворения внутренних нужд ХТИ – филиала СФУ, поэтому она будет бесплатной и не будет приносить дохода.

3.5 Экономическая эффективность реализации системы отслеживания ошибо

к в программных продуктах

Система отслеживания ошибок в программных продуктах разрабатывается для учебного заведения с целью дать студентам возможность выполнять лабораторные работы по дисциплине «Тестирование и контроль качества информационных систем», поэтому данный продукт не будет приносить прибыль, сокращать расходы на персонал или повышать эффективность работы ХТИ – филиала СФУ. Проект предоставит возможность студентам выполнять лабораторные работы в комфортной информационной среде параллельно со своими одногруппниками. Институт сможет заменить ограниченные тестовые версии сторонних информационных систем на альтернативную, специально приспособленную для занятий. Наличие такой системы, разработанной в институте, дает возможность эффективнее привлекать абитуриентов во время сезона поступлений.

3.6 Выводы по разделу «Оценка экономической эффективнос

ти разраб отки и внедре ния систем ы отслеж ивани я ошибо к в програ ммных продук тах»

В данном разделе были посчитаны капитальные и эксплуатационные затраты на создание проекта. Произведен расчет рисков реализации проекта, а также описаны мероприятия по снижению вероятности рисков, имеющих наивысшую оценку, после чего были посчитаны косвенные расходы и наконец подведена итоговая цена проекта.

Определена экономическая эффективность реализации проекта путем сравнительного анализа продуктов аналогов, а также по расчетам затрат на разработку веб-приложения.

По итогу этого этапа было установлено, что реализация проекта полностью оправдана, так как стоимость создания подобных проектов у конкурентов стоит заметно дороже, и не всегда обладает тем наборов характеристик, которые нужны проекту.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы был выполнен анализ предметной области, описана разработка системы отслеживания ошибок в программных продуктах и выполнена оценка экономической эффективности данной разработки.

Определены предпосылки для разработки системы, определена сфера автоматизации и актуальности разработки. Проанализирован бизнес-процесс тестирования ИС с последующей декомпозицией в виде диаграмм IDEF0 вплоть до уровня, на котором будет использована данная система.

Выполнен выбор и обоснование средств разработки. Определены программные средства, которые использованы при разработке данной информационной системы, такие как языки программирования, библиотеки, фреймворки, базы данных и другие вспомогательные технологии.

Описаны основные процессы, связанные с разработкой системы. Выбран жизненный цикл разработки, выполнено проектирование базы данных в виде диаграммы ER. Создан GitHub репозиторий, а также настроен сервер непрерывной интеграции. Разработаны клиентская и серверная часть информационной системы.

Проведен расчет рисков реализации проекта, а также описаны мероприятия по снижению вероятности рисков, имеющих наивысшую оценку, после чего были посчитаны косвенные расходы и наконец подведена итоговая цена проекта. Определены доходы и экономическая эффективность реализации информационной системе, произведен анализ рынка, выявлены сильные стороны проекта и проведен анализ конкурентов.

Разработанная система отслеживания ошибок в программных продуктах может быть использована ХТИ – филиалом СФУ в образовательных целях вместо аналогичных коммерческих систем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ХТИ – филиал СФУ [Электронный ресурс]. – Режим доступа: http://khti.sfukras.ru/

2. Вести.ru: Фестиваль в Самаре приветствовал первый космонавт "Алексей" Гагарин [Электронный ресурс]. – Режим доступа: https://www.vesti.ru/article/1542569

3. Json Web Token Specification [Электронный ресурс]. – Режим доступа: https://tools.ietf.org/html/rfc7519

4. Документация к библиотеке React [Электронный ресурс]. – Режим доступа: https://ru.reactjs.org/docs/getting-started.html

5. Spring Framework Reference Documentation [Электронный ресурс]. – Режим доступа: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/htmlsingle/

6. Spring Security Reference [Электронный ресурс]. – Режим доступа: https://docs.spring.io/spring-security/site/docs/current/reference/html5/

7. Hibernate Documentation [Электронный ресурс]. – Режим доступа: https://docs.jboss.org/hibernate/orm/5.4/userguide/html_single/Hibernate_User_Guide.html

8. PostgreSQL Documentation [Электронный ресурс]. – Режим доступа: https://www.postgresql.org/docs/current/

9. Head First. Паттерны проектирования. Обновленное юбилейное издание: книга /, Э. Фримен, Э. Робсон, К. Сьерра, Б. Бейтс. – СПб.: Питер, 2018. — 656 с.: ил. — (Серия «Head First O’Reilly»).

10. Приемы объектно-ориентированного проектирования. Паттерны проектирования: книга / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб: Питер, 2001. — 368 с.: ил. (Серия «Библиотека программиста»)

11. Козмина, Ю., Харроп, Р. Spring 5 для профессионалов: книга / Ю. Козмина, Р. Харроп, Пер. с англ. – СПб.: ООО "Диалектика", 2019. – 1120 с.

12. Шилдт, Герберт Java. Полное руководство, 10-е изд.: книга /

Герберт Шилдт, Пер. с англ. – СПБ.: ООО «Альфа-книга», 2018. – 1488 с.

13. Эккель, Б., Философия Java. 4-е полное изд.: книга / Б. Эккель. – СПб.: Питер, 2015. — 1168 с.: ил. — (Серия «Классика computer science»).

14. Spring in Action. Craig Walls [Электронный ресурс] – https://vk.com/doc26879026_479528892?hash=262228264ba26b1344&dl=5b8c181f57617bb10b

15. Spring Boot in Action. Craig Walls [Электронный ресурс] – https://doc.lagout.org/programmation/Spring%20Boot%20in%20Action.pdf

16. Spring Security in Action. Laurentiu Spilca [Электронный ресурс] – https://mirlib.ru/knigi/programming/450797-spring-security-in-action.html

ПРИЛОЖЕНИЕ А

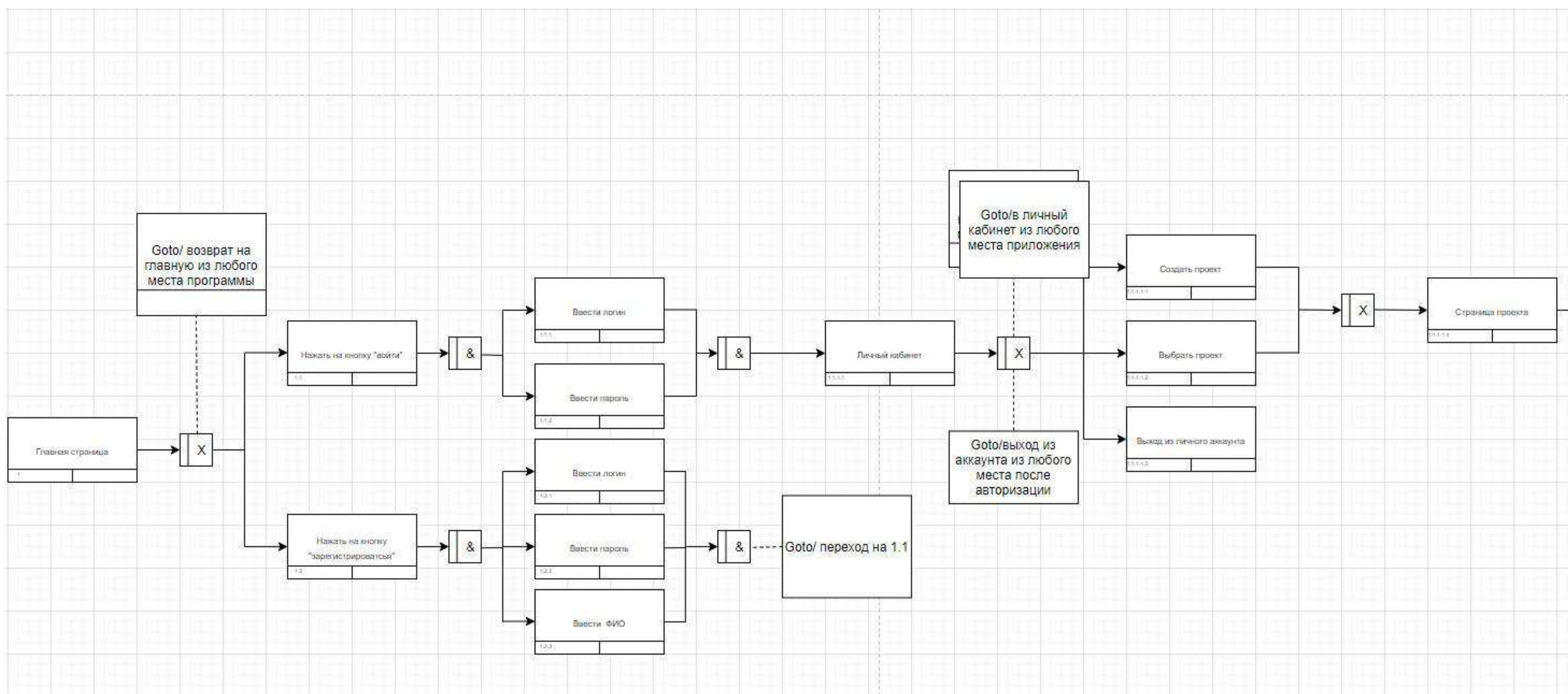


Рисунок А.1 – IDEF3 информационной системы, лист 1

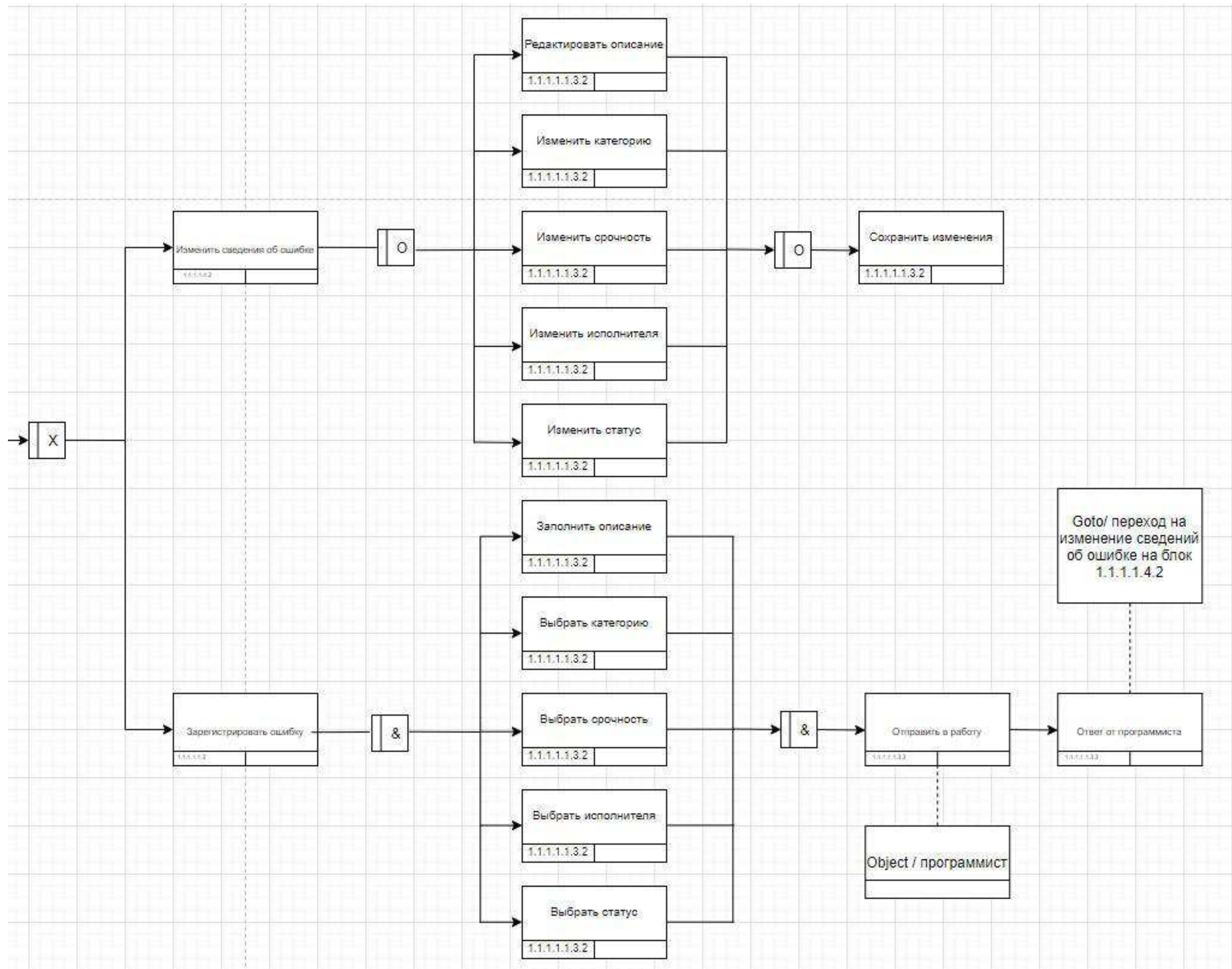


Рисунок А.1, лист 2

Выпускная квалификационная работа выполнена мной самостоятельно. Использованные в работе материалы и концепции из опубликованной научной литературы и других источников имеют ссылки на них.

Отпечатано в одном экземпляре.

Библиография 16 наименований.

Один экземпляр сдан на кафедру.

«____» ____________ 2021 г.

____________________ Яцутко Сергей Анатольевич

подпись

Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Хакасский технический институт – филиал ФГАОУ ВО
«Сибирский федеральный университет»

Кафедра прикладной информатики, математики и естественно-научных дисциплин

УТВЕРЖДАЮ
Заведующий кафедрой

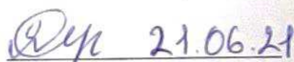
____________ Е. Н. Скуратенко
подпись
« 21 » 06 2021 г.

БАКАЛАВРСКАЯ РАБОТА

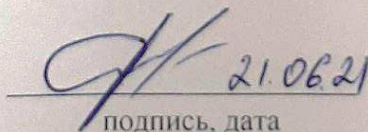
09.03.03 Прикладная информатика

Разработка системы отслеживания ошибок в программных продуктах

Руководитель доцент, канд. пед. наук И. В. Янченко
подпись, дата

Выпускник  С. А. Яцутко
подпись, дата

Консультанты
по разделам:

Экономический  Е. Н. Скуратенко
подпись, дата

Нормоконтролер В. И. Кокова
подпись, дата

Абакан 2021