Содержание

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | 1 Обзор литературы  и аналогов | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

[Содержание 1](#_Toc135857125)

[Реферат 4](#_Toc135857126)

[Abstract 5](#_Toc135857127)

[Введение 6](#_Toc135857128)

[1 Обзор литературы и аналогов 7](#_Toc135857129)

[1.1 Аналогичные программные решения 7](#_Toc135857130)

[1.1.1 Приложение Drebedengi 7](#_Toc135857131)

[1.1.2 Приложение Easyfinance 8](#_Toc135857132)

[1.1.3 Приложение Buddy 9](#_Toc135857133)

[1.1.4 Приложение Дзен-мани 10](#_Toc135857134)

[1.2 Патентный поиск 11](#_Toc135857135)

[1.3 Постановка задачи 12](#_Toc135857136)

[1.4 Выводы по разделу 13](#_Toc135857137)

[2 Проектирование программного обеспечения 14](#_Toc135857138)

[2.1 Архитектура приложения 15](#_Toc135857139)

[2.2 Проектирование клиентской части 15](#_Toc135857140)

[2.2.1 Использование *React* 15](#_Toc135857141)

[2.2.2 Redux 16](#_Toc135857142)

[2.2.3 Webpack 17](#_Toc135857143)

[2.2.4 Архитектура клиентской части 17](#_Toc135857144)

[2.3 Проектирование серверной части 18](#_Toc135857145)

[2.3.1 Процесс проверки целей пользователя 19](#_Toc135857146)

[2.4 Проектирование базы данных 19](#_Toc135857147)

[2.5 Выводы по разделу 24](#_Toc135857148)

[3 Разработка программного средства 25](#_Toc135857149)

[3.1 Серверная часть приложения 25](#_Toc135857150)

[3.1.1 Структура проекта 26](#_Toc135857151)

[3.1.2 Конфигурация и запуск проекта 28](#_Toc135857152)

[3.1.3 Уровень доступа к данным 30](#_Toc135857153)

[3.1.4 Уровень представления 32](#_Toc135857154)

[3.1.5 Уровень бизнес-логики 34](#_Toc135857155)

[3.1.6 Класс фоновых работ 35](#_Toc135857156)

[3.2 Клиентская часть приложения 36](#_Toc135857157)

[3.2.1 Структура проекта 36](#_Toc135857158)

[3.2.2 Настройка Webpack 39](#_Toc135857159)

[3.2.3 Разработка компонентов 39](#_Toc135857160)

[3.2.4 Менеджер состояний Redux 40](#_Toc135857161)

[3.2.5 Добавление стилей 41](#_Toc135857162)

[3.3 Выводы по разделу 42](#_Toc135857163)

[4 Тестирование приложения 43](#_Toc135857164)

[4.1 Тестирование серверной части приложения 43](#_Toc135857165)

[4.1.1 Unit тестирование приложения 43](#_Toc135857166)

[4.1.2 Мануальное тестирование Web API 46](#_Toc135857167)

[4.2 Тестирование клиентской части 50](#_Toc135857168)

[4.3 Выводы по разделу 53](#_Toc135857169)

[5 Руководство пользователя 54](#_Toc135857170)

[5.1 Выводы по разделу 63](#_Toc135857171)

[6 Технико-экономическое обоснование проекта 64](#_Toc135857172)

[6.1 Общая характеристика разрабатываемого программного средства 64](#_Toc135857173)

[6.2 Исходные данные для проведения расчётов и маркетинговый анализ 64](#_Toc135857174)

[6.3 Обоснование цены программного средства 64](#_Toc135857175)

[6.3.1 Расчёт затрат рабочего времени на разработку программного средства 65](#_Toc135857176)

[6.3.2 Расчет основной заработной платы 66](#_Toc135857177)

[6.3.3 Расчет дополнительной заработной платы 66](#_Toc135857178)

[6.3.4 Расчет отчислений в Фонд социальной защиты населения и по обязательному страхованию 66](#_Toc135857179)

[6.3.5 Расчет суммы прочих прямых затрат 67](#_Toc135857180)

[6.3.6 Расчет суммы накладных расходов 67](#_Toc135857181)

[6.3.7 Сумма расходов на разработку программного средства 68](#_Toc135857182)

[6.3.8 Расходы на реализацию 68](#_Toc135857183)

[6.3.9 Расчет полной себестоимости 68](#_Toc135857184)

[6.3.9 Определение цены, оценка эффективности 69](#_Toc135857185)

[6.4 Вывод по разделу 70](#_Toc135857186)

[Заключение 71](#_Toc135857187)

[Список использованных источников 72](#_Toc135857188)

[Приложение А Диаграмма вариантов использования 74](#_Toc135857189)

[Приложение Б Диаграмма архитектуры приложения 75](#_Toc135857190)

[Приложение В Блок-схема алгортима проверки цели 76](#_Toc135857191)

[Приложение Г Логическая схема базы данных 77](#_Toc135857192)

[Приложение Д Диаграмма развертывания 78](#_Toc135857193)

[Приложение Ж Диаграмма классов приложения 79](#_Toc135857194)

Реферат

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | Реферат | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
|  | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

Пояснительная записка дипломного проекта содержит Х страниц пояснительной записки, Х таблиц, Х формул, Х иллюстраций, Х источников литературы, Х приложений.

ВЕБ-ПРИЛОЖЕНИЕ, УЧЕТ РАСХОДОВ, УПРАВЛЕНИЕ ДОЛГАМИ, УПРАВЛЕНИЕ ФИНАНСАМИ, ФИНАНСОВЫЕ ЦЕЛИ, ФИНАНСОВЫЙ МЕНЕДЖЕР

Результатом проделанной работы является веб-приложение типа финансовый менеджер. Пользователю приложение доступно на всех платформах через веб-браузер.

Пояснительная записка состоит из введения, шести разделов, заключения и списка используемой литературы.

В введении сформулирована цель и задачи проекта.

В первом разделе производится обзор аналогичных решений, патентный поиск и постановка задачи.

Во втором разделе происходит обоснование технических приемов и производится проектирование веб-сервиса.

В третьем разделе описаны основные шаги в реализации серверной и клиентской частей приложения.

В четвертом разделе описаны основные шаги тестирования проекта.

В пятом разделе описано руководство пользователя, позволяющее понять способы взаимодействия с пользовательским интерфейсом клиентской части.

В шестом разделе приводится расчет экономических параметров и себестоимости программного средства, разработанного в рамках дипломного проекта.

В заключении представлены итоги дипломного проектирования и задачи, которые были решены в ходе разработки проекта.

Объем графической части дипломного проекта составляет 1.25 листа А1.

Abstract

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | Abstract | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
|  | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

The explanatory note of the diploma project consists of X pages of explanatory text, X tables, X formulas, X illustrations, X literature sources, and X appendices.

WEB APPLICATION, EXPENSE TRACKING, DEBT MANAGEMENT, FINANCIAL MANAGEMENT, FINANCIAL GOALS, FINANCIAL MANAGER

The outcome of the conducted work is a web-based financial manager application. The application is accessible to users on all platforms through a web browsers.

The explanatory note includes an introduction, six sections, a conclusion, and a list of references.

The introduction presents the project's goals and objectives.

The first section provides an overview of similar solutions, patent research, and problem formulation.

The second section justifies the technical approaches and conducts the design of the web service.

The third section describes the main steps in implementing the server-side and client-side components of the application.

The fourth section outlines the main steps of project testing.

The fifth section provides a user guide that explains the ways to interact with the user interface of the client-side.

The sixth section presents the calculation of economic parameters and the cost estimation of the software developed within the scope of the diploma project.

The conclusion summarizes the results of the diploma project and the tasks that were solved during the project development.

The graphical part of the diploma project has a volume of 1.25 sheets of A1 size.

Введение

В наше время экономическая стабильность является одним из ключевых приоритетов для многих людей. Эффективное управление финансами и правильное распределение денежных средств могут существенно повлиять на качество жизни и достижение финансовых целей. С развитием технологий возрос интерес к различным инструментам и приложениям, которые могут помочь в сохранении денег и управлении финансами.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | Введение | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
|  | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

Целью дипломного проекта является разработка веб-приложения «Better save», которое предоставляет пользователям инструменты для контроля над своими финансами, такие как календарь долгов, конструктор финансовых целей, аналитика о совершенных операциях.

Для достижения цели были поставлены следующие задачи:

* выбор и обоснование средств разработки;
* исследовать преимущества и недостатки аналогичных приложений;
* составить набор требований к разрабатываемому программному средству;
* спроектировать архитектуру приложения и базы данных;
* разработать серверную и клиентскую части приложения;
* провести тестирование работоспособности приложения;
* написать руководство пользователя.

Таким образом разрабатываемое приложение должно предоставляться пользователю в виде набора страниц, просматриваемых в браузере, между которыми присутствует возможность перехода. Веб-приложение должно взаимодействовать с централизованной базой данных, обеспечивающей хранение пользовательской информации. Само взаимодействие между клиентской и серверной частью должно базироваться на клиент-серверной архитектуре, пользователь должен иметь доступ к приложению через протокол HTTPS.

1. Обзор литературы и аналогов

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | 1 Обзор литературы  и аналогов | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

* 1. Аналогичные программные решения
     1. Приложение *Drebedengi*

*Drebedengi* – это приложение для учета личных финансов [1]. К плюсам приложения можно отнести то, что кроме веб-версии, существуют приложения для *iOS* и *Android*. Следует отметить, что приложение имеет устаревший интерфейс (рисунок 1.1).

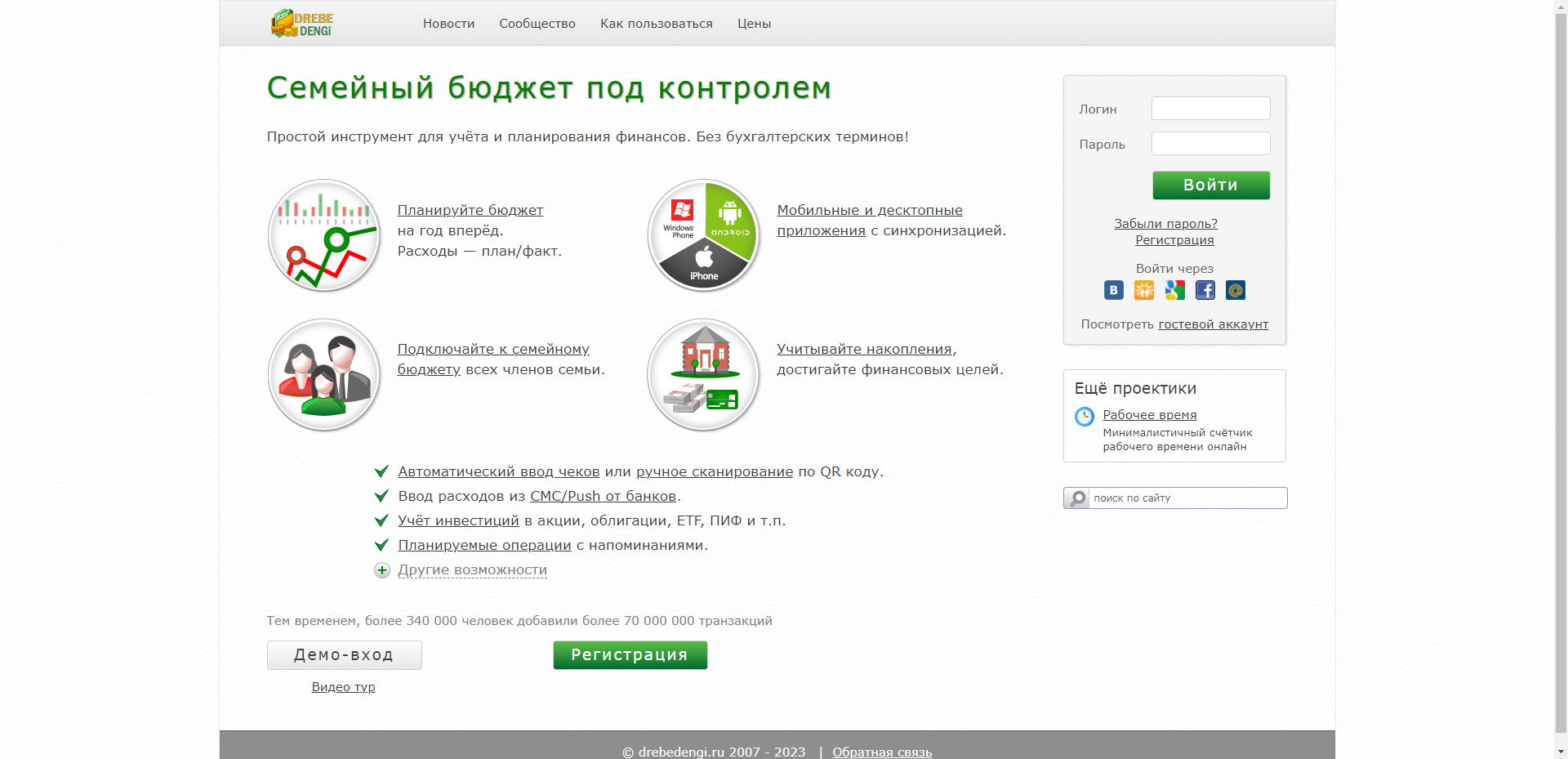


Рисунок 1.1 – Снимок главной страницы приложения *Drebedengi*

Программа позволяет производить различные операции, хранить историю покупок, просматривать графики, планировать бюджет. Снимок экрана приложения представлены на рисунке 1.2.

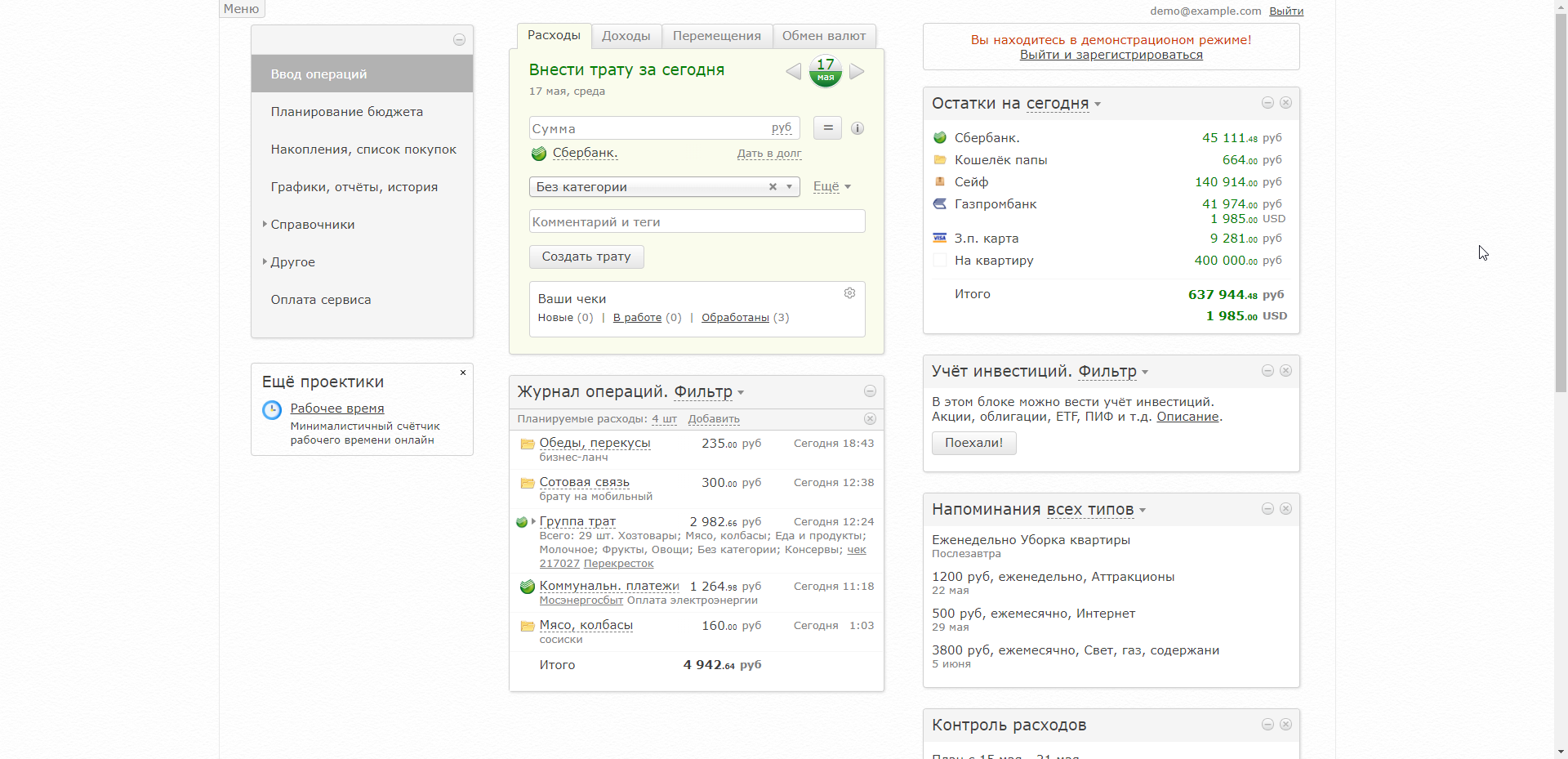


Рисунок 1.2 – Снимок экрана приложения *Drebedengi*

Достоинства приложения:

* есть функции, которые не требуют подписки;
* возможность создавать собственные категории;
* возможность отслеживать долги;
* возможность использовать на всех платформах.

Недостатки приложения:

* устаревший интерфейс;
* необходимость устанавливать напоминания самому при отслеживании долгов;
* графики не содержат категорий трат;
* неудобная форма регистрации.
  + 1. Приложение *Easyfinance*

*Easyfinance* – приложение для контроля расходов и доходов. Приложение позиционирует себя как лидер рынка систем управления личными финансами в России [2]. Мобильные приложения доступны по подписке. Интерфейс приложения изображен на рисунке 1.3

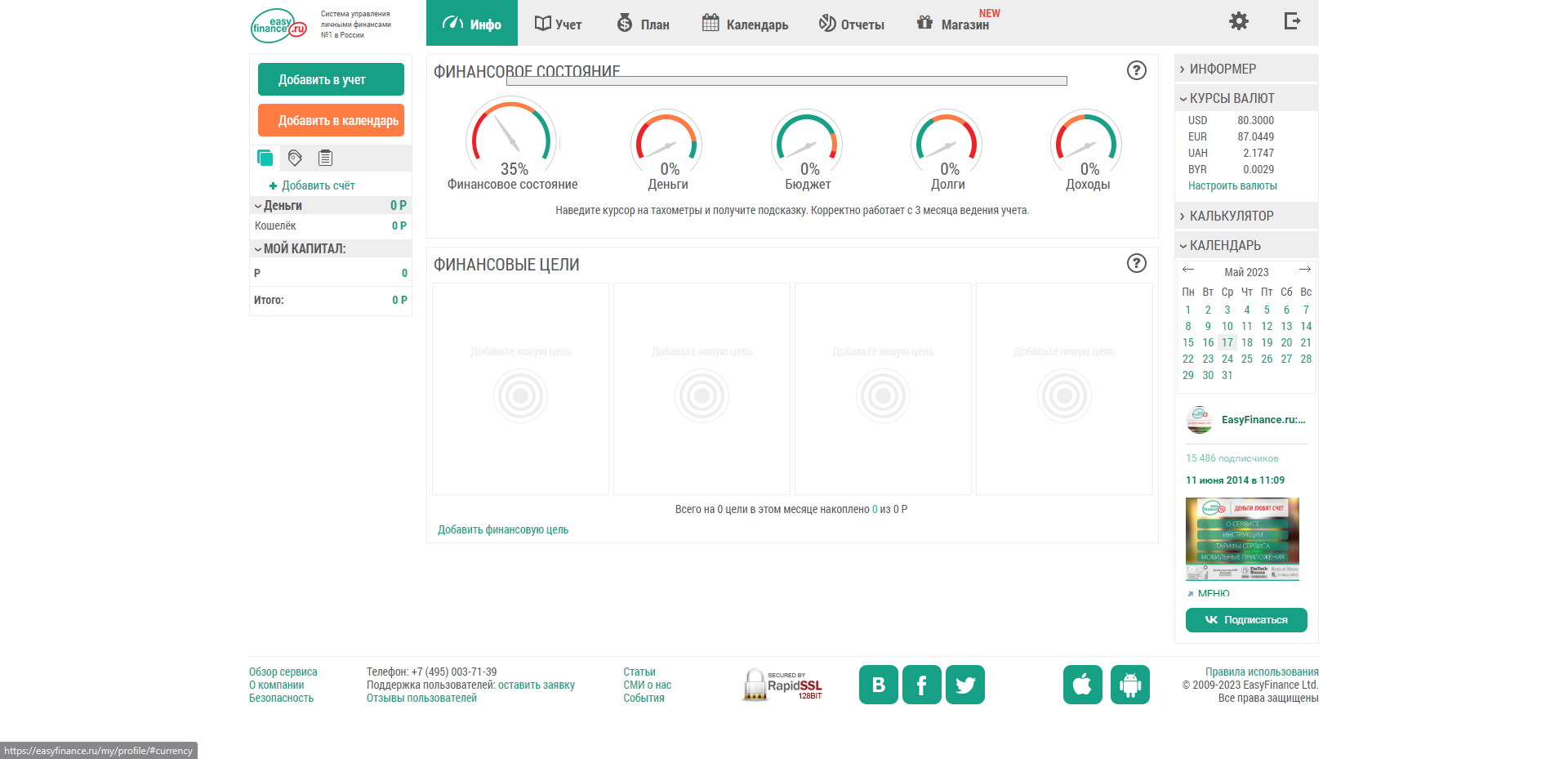


Рисунок 1.3 – Главная страница приложения *Easyfinance*

Когда пользователь приложения регистрируется в приложении его встречает опрос, по результатам которого будет настроена его учетная запись. Стоит отметить, что последовательность вопросов при первом входе в приложении содержит ошибки (рисунок 1.4). Кроме того, это усложняет процесс пользования приложением и отталкивает пользователя.

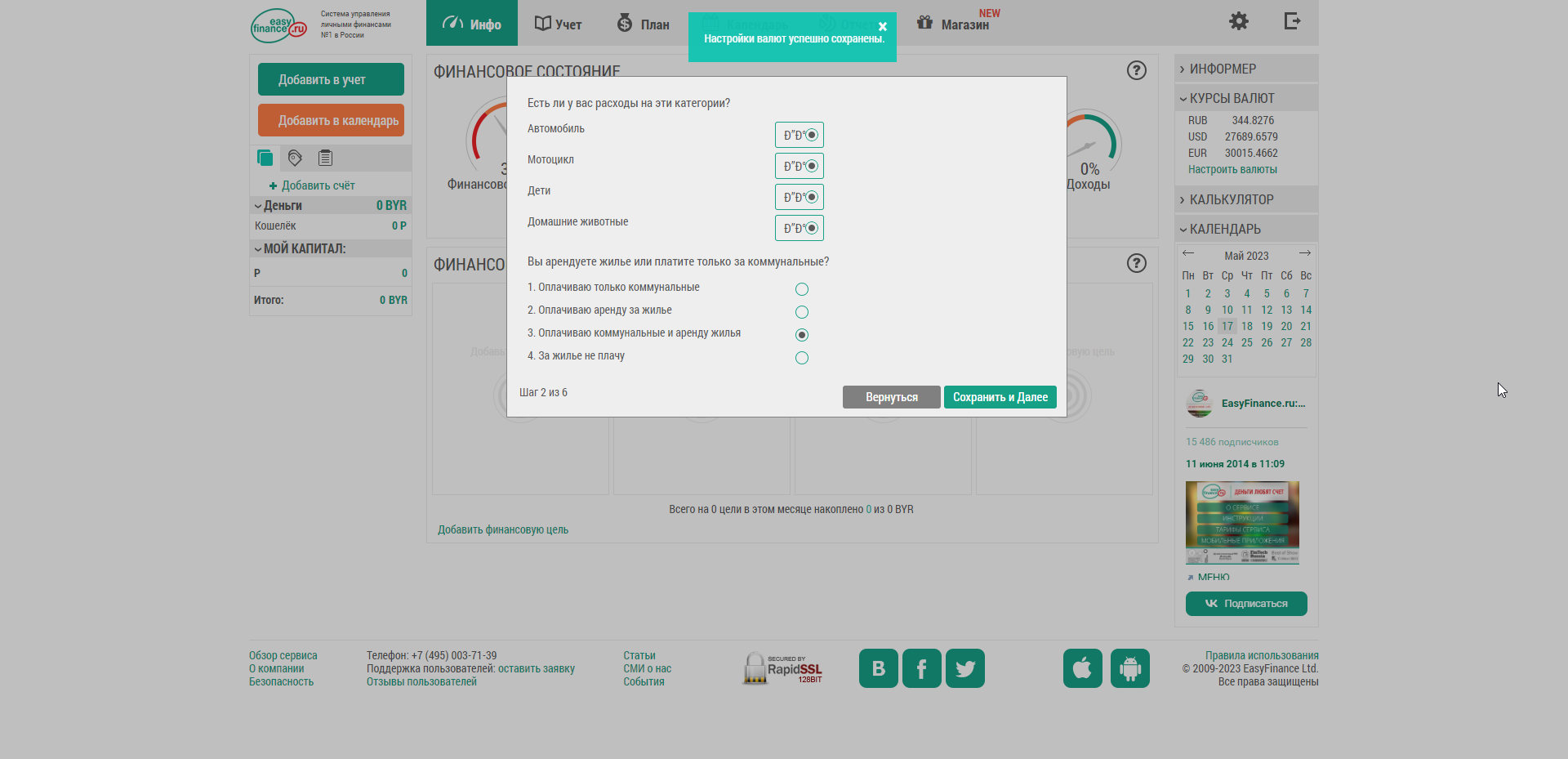


Рисунок 1.4 – Ошибки в приложении *Easyfinance*

Достоинства приложения:

* настройка приложения на старте;
* можно создавать отчеты;
* можно создавать несколько счетов.

Недостатки приложения:

* устаревший интерфейс;
* длинная начальная настройка;
* часть функционала доступна по подписке, включая мобильные приложения.
  + 1. Приложение Buddy

*Buddy* – это мобильное приложение, доступное только на платформе *iOS* [3]. Приложение предоставляет привлекательный и современный интерфейс, что выделяет его на фоне конкурентов. Часть функций приложения работают с подпиской, но некоторые доступны в бесплатной версии приложения. Например, планирование бюджета, хранение трат, разбивка суммы между людьми. Интерфейс приложения изображен на рисунке 1.5. Также следует отметить, что приложение позволяет сменить тему и локализацию, что также является плюсом.

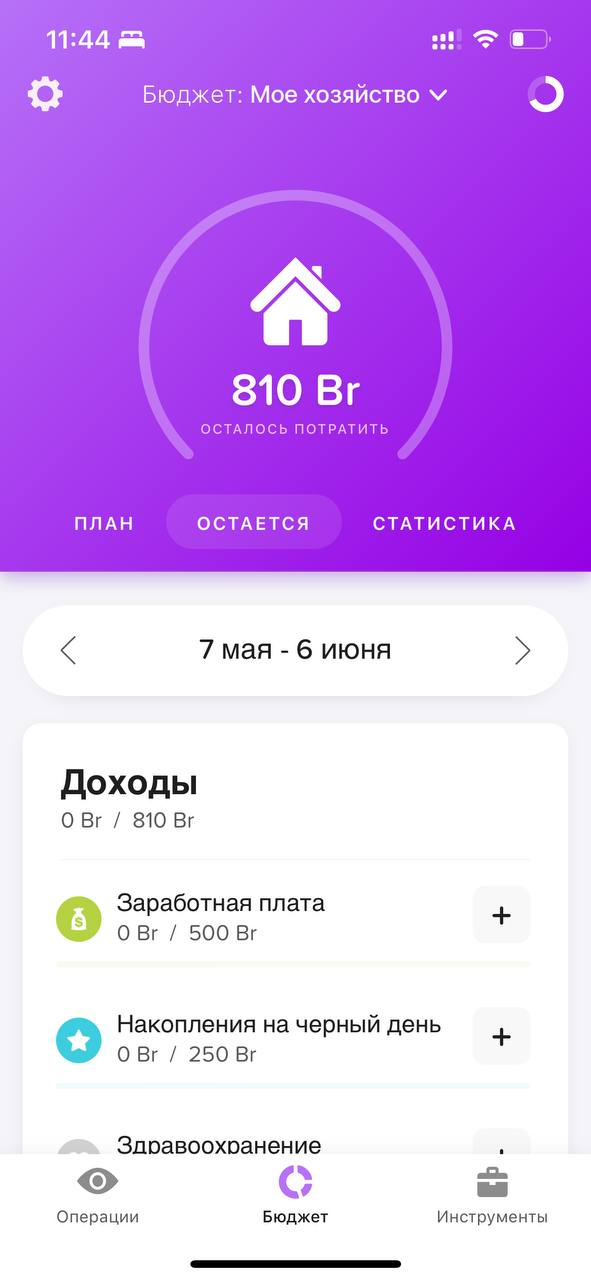


Рисунок 1.5 – Приложение *Buddy*

Достоинства приложения *Buddy*:

* приятный внешний вид;
* быстрота работы.

Недостатки приложения:

* часть функций доступна по подписке;
* доступ к приложению только для пользователей *iOS*;
* нет возможности добавлять долги и займы.
  + 1. Приложение Дзен-мани

Дзен-мани – это приложение для учета личных финансов, предназначенное для помощи пользователям в организации доходов, расходов, финансовых целей [4]. Приложение предоставляет широкий спектр функций. Также оно имеет мобильные аналоги. Стоит сказать, что интерфейс веб-версии немного старый, но к внешнему виду мобильных аналогов вопросов не возникает. Кроме перечисленных функций, приложение позволяет смотреть отчеты по счетам. Внешний вид приложения дзен-мани изображен на рисунке 1.6.

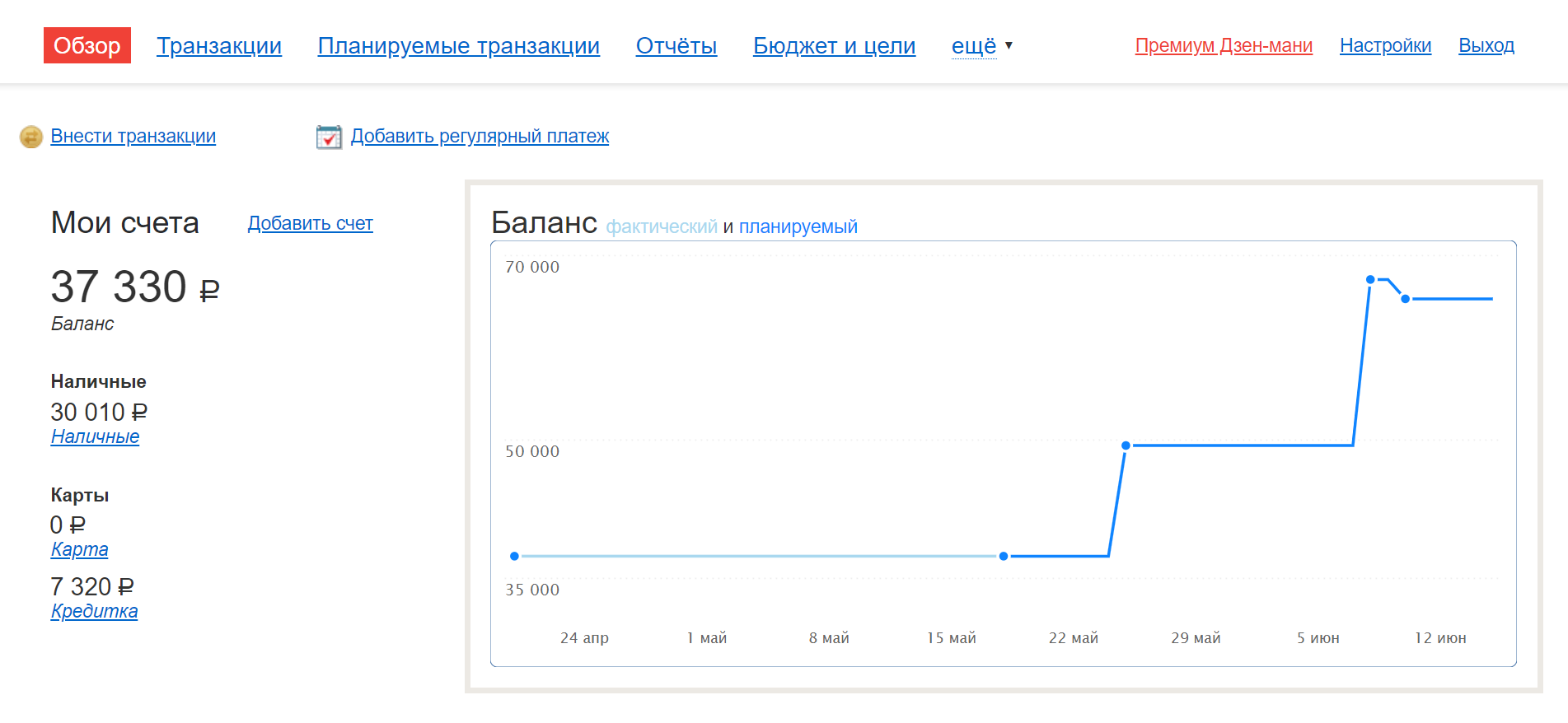


Рисунок 1.6 – Интерфейс приложения Дзен-мани

Достоинства приложения дзен-мани:

* учет доходов и расходов;
* категоризация трат;
* анализ финансовых трат.

Недостатки приложения:

* ограниченный набор функций;
* интеграция с банками накладывает региональные ограничения по доступности.
  1. Патентный поиск

Цель патентного поиска при выполнении дипломного проекта заключается в оценке технологического прогресса и оригинальности существующего технического решения.

Проведение поиска помогает установить, как задача была решена до появления патентов и авторских свидетельств, а также определить перспективы дальнейшего их развития. Обоснование технических приемов программирования.

В качестве результата патентного поиска были обнаружены патенты, представленные в таблицах 1.1, 1.2, 1.3.

Таблица 1.1

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | Номер патента | Опубликовано | Авторы |
| Интеллектуальная программа сбережения | *US8301530B2* | 30.10.2012 | Дэниел А. Карретта Томас М. Хэнкок Карен К. Льюис Чарльз Р. Лю Джек Мейерс Шерил У. Стотт |

В соответствии с описанием патента [5], представленная система позволяет автоматически накапливать средства на основе финансовых транзакций. Основная идея заключается в том, что при выполнении финансовой транзакции сумма покупки округляется до ближайшего доллара, и разница между округленной суммой и исходной суммой транзакции направляется в накопительный счет пользователя.

Таблица 1.2

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | Номер патента | Опубликовано | Авторы |
| Способ и система экономии денег с помощью группы мобильных устройств | *US8090656B2* | 03.01.2012 | Лия Соломон, Джейсон Халладей |

Основная идея данного патента заключается в создании системы, которая позволяет пользователям осуществлять финансовые операции и управлять своими финансами с помощью мобильных устройств, таких как смартфоны или планшеты [6]. Система предоставляет возможность осуществлять платежи, проверять баланс счетов, просматривать транзакции и выполнять другие операции, связанные с финансами, с помощью специального мобильного приложения.

Таблица 1.3

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование | Номер патента | Опубликовано | Авторы |
| Динамическая система, метод и устройство финансового управления | *US11100513B2* | 24.08.2021 | Келлар Шон Кроуфорд |

В соответствии с описанием патента, главной его целью является динамическая персонализируемая автоматизированная система управления финансами, которая предоставляет платформу для финансового управления, позволяющую пользователям легко создавать несколько настраиваемых правил или условий, связанных с одним или несколькими счетами, тем самым создавая планы счетов, которые интеллектуально и пассивно осуществляют перемещение средств между счетами. Правила с планом могут определять, когда, сколько и куда переводить деньги на основе пользовательских заданных критериев, на которых основаны правила/условия [7].

* 1. Постановка задачи

Для обеспечения конкурентоспособности разрабатываемое программное средство должно быть кроссплатформенным, используемым в сети интернет.

Проектирование программного средства будет включать в себя:

* проектирование архитектуры приложения;
* проектирование клиентской части приложения;
* проектирование серверной части приложения;
* проектирование базы данных.

Для выполнения поставленных целей, в рамках дипломного проектирования необходимо:

* реализовать механизм регистрации и авторизации;
* реализовать механизм изменения информации о пользователе;
* разработать способ хранения информации о тратах, категориях;
* реализовать механизм информирования пользователя;
* реализовать возможность добавления долгов и целей.
  1. Выводы по разделу

По результатам исследования реализаций аналогов приложения для учета финансов и патентного поиска было обнаружено, что на данном этапе не существует программного средства, которое реализовало бы все возможности разрабатываемого приложения, а именно:

* удобный и понятный интерфейс;
* отсутствие ошибок в работе программы;
* хранение информации о тратах;
* составление диаграмм с категориями трат;
* уведомление пользователя по электронной почте и в приложении;
* возможность ставить цели и отслеживать прогресс;
* возможность вести учет баланса пользователя;
* хранение информации о статусе займов.

Сводя всё вышесказанное воедино, можно утверждать, что проведенная аналитическая работа в обзоре аналогов является основополагающим фактором для создания качественного программного обеспечения. Этот этап работы дает нам возможность перейти к рассмотрению более конкретных технических приемов программирования, которые являются ключевыми для разработки программного продукта.

1. Проектирование программного обеспечения

Перед тем, как приступать к проектированию программного средства, необходимо провести анализ всех поставленных задач и возможных вариантов его использования. Для этой цели следует разработать диаграмму вариантов использования, которая станет основой дальнейшей работы над приложением. Пример такой диаграммы можно увидеть в приложении А научной работы.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
| Изм. | Лист | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | 2 Проектирование программного обеспечения | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

Разрабатываемое клиент-серверное приложение обладает несколькими преимуществами, такими как доступность через браузер на любом устройстве с интернет-соединением, кроссплатформенность и совместимость. Роль пользовательского интерфейса выполняется клиентской частью, которая отображает информацию и предоставляет пользователю возможность взаимодействия с ней.

Клиентская часть разработана с использованием библиотеки *React* [8]. Эта библиотека позволяет одностраничные приложения. Клиентская часть разрабатываемого приложения будет обмениваться данными с сервером незаметно для пользователя: без перезагрузки страницы или временной остановки возможности использования приложения. В качестве библиотеки, на которой будет основана вся остальная разработка клиентской части, была выбрана именно *React*, потому что это современная, быстро развивающаяся библиотека, использующая язык *TypeScript* [9], который в свою очередь является надмножеством языка *JavaScript* [10]. В отличии от других библиотек и фреймворков, в *React* используется технология *Virtual DOM* [11], использование которой позволяет оптимизировать производительность приложения и ускорить работу с интерфейсом, особенно в случае больших и сложных приложений.

Серверная часть представляет собой *ASP.NET Core Web API* приложение. *Web API* это фреймворк для создания веб-сервисов, которые могут общаться через *HTTP*-протокол [12]. Он позволяет создавать веб-сервисы, которые могут обмениваться данными с клиентскими приложениями или другими серверами программного обеспечения. Серверная часть приложения будет разделена на три слоя: *PL (Presentation Layer)*, *DAL (Data Access Layer)*, *BL (Business layer)*. Общая идея слоистой архитектуры заключается в том, чтобы разделить приложение на набор слоев, каждый из которых решает определенную задачу и имеет свои собственные обязанности. Это позволяет легче понимать и изменять код, уменьшает связность между разными частями приложения и упрощает тестирование и развертку приложения.

Для хранения информации пользователя используется система управления базами данных *MS SQL* [13]. Для взаимодействия *Web API* и базы данных используется фреймворк *Entity Framework Core* [14]. EF Core представляет собой средство объектно-реляционного отображения *(Object-Relational Mapping, ORM)*, которое позволяет отображать данные из базы данных в реальные объекты программы и обратно.

Использование *ORM* экономит время разработчиков и позволяет им сосредоточиться на бизнес-логике, а не на управлении базой данных.

* 1. Архитектура приложения

Для разрабатываемого проекта была разработана диаграмма архитектуры приложения, представленная на рисунке 2.1, а также в приложении Б. На этой диаграмме представлена взаимосвязь между компонентами серверной и клиентской частей приложения.

|  |
| --- |
|  |

Рисунок 2.1 – Диаграмма архитектуры приложения

Приложение будет работать по принципу взаимодействия клиентского и серверного приложений: клиент через *API* веб-браузера будет посылать асинхронные запросы на сервер и получать от него данные. Сервер будет работать на платформе *ASP.NET Core*, которая гарантирует кроссплатформенность полученного серверного решения. В качестве сервера будет использоваться *IIS* сервер [15].

* 1. Проектирование клиентской части
     1. Использование *React*

Клиентская часть приложения написана с использованием библиотеки *React*.

*React* – это *JavaScript* библиотека, которая используется для создания динамических пользовательских интерфейсов. Она позволяет разработчикам создавать компоненты, которые можно повторно использовать в других частях приложения. Это позволяет существенно ускорить процесс создания, тестирования и модификации приложений.

Каждый компонент в *React* может состоять из нескольких других компонентов, а также быть вложенным в другие компоненты. Они могут реагировать на действия пользователя, что делает сайт более интерактивным.

*React* использует виртуальный *DOM*, что увеличивает производительность приложения, так как обновления в реальном *DOM* происходят только при изменении данных. Это позволяет избежать его полной перерисовки, что делает работу с библиотекой более эффективной.

Одним из главных преимуществ *React* является то, что он позволяет разработчикам легко работать с данными. *React* позволяет получить данные из сервера при помощи *AJAX*-запросов, что делает приложение более динамичным.

В целом, *React* можно описать как мощный инструмент, который позволяет разработчикам создавать динамические, интерактивные и быстрые пользовательские интерфейсы. Использование этой библиотеки в разработке клиентской части может существенно ускорить процесс создания и модификации приложения.

* + 1. Использование *Redux*

*Redux* – это библиотека для управления состоянием приложения в *JavaScript* и *TypeScript* [16]. Она позволяет хранить все данные в едином объекте-хранилище. Это делает управление состоянием приложения более предсказуемым и удобным, а также улучшает производительность при работе с большим объёмом данных.

Redux позволяет отслеживать неконтролируемые изменения данных, хранимых в приложении, поскольку кроме хранения данных, после каждой операции делает снимок данных лежащих в памяти. Эти неконтролируемые изменения объектов в *TypeScript* могут быть опасными, так как они нарушают ограничения типизации и могут привести к непредсказуемому поведению в программе. В частности, они могут вызвать ошибки во время работы программы или привести к несогласованности данных. Но это не повод отказываться от *TypeScript* в пользу *JavaScript*, поскольку в *JavaScript* вообще нет типизации. Также *Redux* позволяет избежать дополнительных запросов на сторону сервера за одними и теми же данными, которые используются в разных местах приложения.

Для удобства разработки, повышения качества кода и удобства тестирования приложения, мною была выбрана библиотека *redux-toolkit [17].* Эта библиотека содержит внутри себя набор функций для работы с *redux.* Она позволяет уменьшить количества написанного кода путем инкапсуляции некоторых деталей реализации функций необходимых для работы с *redux*. С помощью этой библиотеки, разработчик способен манипулировать абстракциями, созданными за него и при желании изменять их.

Также, *redux-toolkit* содержит в себе функции, с помощью которых разработчик приложения может доставать объекты из памяти браузера, с помощью которых он может производить некоторые изменения.

Для взаимодействия с серверной частью приложения используется библиотека *redux-thunk*. Как можно понять из названия, эта библиотека тоже является частью экосистемы redux. *Redux-thunk* используется для того, чтобы с помощью асинхронных методов осуществлять запросы к серверу для получения данных. Плюсами этой библиотеки является то, что разработчик может контролировать стадии выполнения запроса, и, например, показывать пользователю приложения статус выполнения запроса. Также, использование этой библиотеки позволяет разделить приложение клиентской части на несколько логических слоев.

* + 1. Использование *Webpack*

*Webpack* является инструментом сборки *JavaScript*-приложений, которые используются веб-браузерами [18]. Это программное обеспечение позволяет разработчикам создавать оптимизированные пакеты кода, которые могут быть загружены в браузер для выполнения.

*Webpack* используется для компиляции и упаковки программных модулей, таких как *TypeScript*, *CSS*, изображения и др. в один или несколько файлов, что упрощает развёртывание приложения. Он также позволяет использовать новейшие функции *JavaScript*, которые не поддерживаются внешними браузерами, и преобразовать код в более старые версии языка, которые могут работать на более широком диапазоне браузеров.

*Webpack* также позволяет конфигурировать и настраивать сборку проекта в зависимости от конкретных потребностей разработчика, а также подключать плагины для оптимизации сборки, например для уменьшения и оптимизации размера файлов.

Благодаря возможностям *webpack*, разработчики имеют возможность создавать мощные и высокоэффективные веб-приложения для современных браузеров. Он позволяет бесконечно расширять возможности сборки приложения, путем добавления различных плагинов, библиотек при сборке приложения.

* + 1. Архитектура клиентской части

Архитектура клиентской части содержит в себе три слоя, каждый из которых отвечает за отдельную группу функций. Взаимодействие между слоями приложения показано на рисунке 2.2.

|  |
| --- |
|  |

Рисунок 2.2 – схема архитектуры клиентской части приложения

Когда пользователь нажимает на кнопку, уровень бизнес-логики обрабатывает это действие, и отправляет запрос к уровню доступа к данным, чтобы изменить или получить результат. Уровень доступа к данным, в свою очередь, инкапсулирует логику работы с сервером, и для уровня бизнес-логики создается впечатление, что сервера вовсе не существует, потому что о сервере знает только один уровень.

Когда запрос с сервера приходит на клиент, уровень доступа к данным обновляет значения, которые хранятся в памяти, и дает понять компонентам, которые подписаны на изменения состояния, что нужно перерисовать разметку страницы. После этого, функции бизнес-логики, которые подписаны на изменения, производят свои расчеты и возвращают данные в компоненты, которые видит пользователь.

* 1. Проектирование серверной части

Серверная часть приложения будет написана на языке *C*# и будет за выполнение функций бизнес-логики и хранения данных пользователей приложения.   
 Приложение, построенное на технологии *ASP.NET Core*, будет использовать конвейер обработки запроса. Это удобное решение для проектирования архитектуры, поскольку позволяет расположить несколько обработчиков, которые будут последовательно обрабатывать запрос и собирать ответ для клиента.

Серверная часть приложения будет разделена на контроллеры, которые хранят логику работы со своей доменной областью. Контроллеры расположены на уровне представления монолитной архитектуры.

Благодаря механизму внедрения зависимостей, существует возможность легко подключать различные объекты, которые реализуют интерфейс, определяемый при внедрении. С помощью этого механизма будут связаны все уровни приложения: на уровне контроллера будет внедрен сервис, который будет отвечать за бизнес-логику приложения. Когда происходит инициализация сервиса, мы внедряем в него репозиторий, который содержит в себе методы для работы с сущностями, которые предоставляет выбранная для разработки *ORM*. При разработке мною был выбран подход «тонкий контроллер», который подразумевает, что вся логика будет сосредоточена в сервисе, а контроллер будет использоваться только для приема запроса, валидации и отправки ответа.

Для корректного отображения данных на стороне клиента необходима информация о пользователе, которая также используется в дальнейших запросах. Поэтому для решения этой проблемы была выбрана аутентификация на основе *JWT* токена, потому что токен может хранить дополнительную информацию, такую как идентификатор пользователя и аккаунта, которые участвуют во многих запросах. Благодаря этому, нет необходимости делать дополнительные запросы к базе данных для получения этой информации.

Тело в *JWT* токене содержит информацию, которая передается между двумя сторонами. Оно состоит из набора пар ключ-значение, обычно содержащих информацию о пользователе, разрешениях, сроке действия и других метаданных.

Тело *JWT* токена не является зашифрованным, но может быть защищено цифровой подписью для проверки подлинности и целостности данных. При получении *JWT* токена, получатель сможет раскодировать тело, прочитать данные и использовать их для проверки прав доступа или других целей.

Кроме общей привычной структуры сервера был класс-обработчик фоновых работ. Фоновые работы запускаются вместе с сервером и выполняются в зависимости от *CRON* выражения, описанного при запуске. За функционал фоновых работ отвечает библиотека *Hangfire [19]*.

Архитектура серверной части приложения изображена на рисунке 2.3.

|  |
| --- |
|  |

Рисунок 2.3 – Архитектура серверной части приложения

* + 1. Процесс проверки целей пользователя

Во время пользования приложением, пользователь должен иметь возможность ставить цели, которые будут учитывать сумму операций за определенный период.

Алгоритм проверки цели изображен в приложении В. На основе этого алгоритма работает фоновая работа, которая срабатывает ежедневно и которая завершает неактуальные задачи пользователей. Также, этот алгоритм отвечает за создание записей для ежедневных целей. По этим записям, пользователь сможет посмотреть свой прогресс в выполнении своих целей.

* 1. Проектирование базы данных

Для проектирования базы данных необходимо изначально проанализировать предметную область и сущности в ней. В итоге будет получена модель данных, показывающая как разработчик должен взаимодействовать с сущностями и как они взаимодействуют между собой.

Главными целями проектирования баз данных являются:

* гарантирование сохранения всей необходимой информации в базе данных;
* обеспечение возможности получения данных при помощи различных запросов;
* сокращение избыточности и дублирования информации в базе данных;
* обеспечение быстрого доступа к данным при поиске по записям;
* установление связей между таблицами в базе данных;
* обеспечение уникальности определенных столбцов в базе данных;
* сокращение необходимости вычисления данных в базе данных;
* гарантия доступности данных для пользователей;
* обеспечение целостности всей базы данных в целом.

Правильное проектирование базы данных обеспечивает эффективную работу с ней и гарантирует, что необходимые данные могут быть получены быстро и точно. Более того, правильно спроектированная база данных может предотвратить ошибки, связанные с неправильным использованием данных, и защитить целостность базы данных.

Модели базы данных для программного средства представлены на рисунке 2.1 и в приложении Г.

|  |
| --- |
|  |

Рисунок 2.1 – Логическая схема базы данных

База данных состоит из основных таблиц и вспомогательных, которые в основном используются для осуществления связи многие-ко-многим. Список всех основных таблиц базы данных:

* *Account;*
* *Aim;*
* *AimRecording;*
* *AspNetUsers;*
* *Category;*
* *Loans;*
* *Operation;*
* *Subcategory.*

Также в базе данных хранится одна вспомогательная таблица *UserAims.*

В базе данных все уникальные значения первичных ключей, которые используются для идентификации записей, представлены и сохранены в виде 128-битных глобальных идентификаторов GUID, которые состоят из тридцати двух шестнадцатеричных цифр, разделенных на группы дефисами. Этот формат идентификаторов обеспечивает максимальную уникальность каждого ключа и предотвращает коллизии, что позволяет точно идентифицировать каждую запись в базе данных. В результате, надежность и точность работы системы баз данных повышаются, что является критически важным для успешной работы любой организации. Структура всех таблиц представлена в приложении Б.

Таблица *Account* хранит информацию о счетах пользователей. Все поля таблицы *Account* перечислены в таблице 2.1

Таблица 2.1 ­­­– Свойства таблицы *Account*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор записи |
| *Name* | *nvarchar(max)* | Название счета |
| *IconColor* | *nvarchar(max)* | Название цвета иконки |
| *IconName* | *nvarchar(max)* | Название иконки |
| *Balance* | *real* | Баланс счета |
| *UserId* | *uniqueidentifier* | Уникальный идентификатор пользователя, которому принадлежит счет |
| *IsDeleted* | *bit* | Флаг, который обозначает, удалена ли запись |

Таблица *Aim* хранит информацию о цели пользователя. Все поля таблицы перечислены в таблице 2.2.

Таблица 2.2 ­­­– Свойства таблицы *Aim*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор записи |
| *Name* | *nvarchar(max)* | Название цели |
| *FinishDate* | *datetime2* | Дата окончания цели |
| *Amount* | *real* | Цель пользователя в денежном эквиваленте |
| *Type* | *int* | Тип цели |
| *DateType* | *int* | Тип цели по дате |
| *IsMastered* | *bit* | Флаг, указывающий завершена ли цель |
| *CreatingDate* | *datetime2* | Дата создания цели |
| *UserId* | *uniqueidentifier* | Уникальный идентификатор пользователя, которому принадлежит цель |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |

Таблица *AimRecording* содержит информацию о прогрессе пользователя в выполнении цели. Поля таблицы содержатся в таблице 2.3.

Таблица 2.3 ­­­– Свойства таблицы *AimRecording*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор записи |
| *Date* | *datetime2* | Дата, когда была выполнена цель |
| *AimId* | *uniqueidentifier* | Уникальный идентификатор цели, которой принадлежит метка о выполнении |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |

Таблица *AspNetUsers* содержит информацию о пользователях. Поля таблицы содержатся в таблице 2.4.

Таблица 2.4 ­­­– Свойства таблицы *AspNetUsers*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор пользователя |
| *FirstName* | *nvarchar(max)* | Имя пользователя |
| *LastName* | *nvarchar(max)* | Фамилия пользователя |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |
| *UserName* | *nvarchar(max)* | Имя аккаунта пользователя |
| *EmailConfirmed* | *bit* | Флаг, который обозначает подтверждена ли почта пользователя |
| *PasswordHash* | *nvarchar(max)* | Хэш пароля пользователя |
| *Birthday* | *datetime2* | Дата рождения пользователя |

Таблица *Category* необходима для хранения информации о главной категории, которая хранит подкатегории. Поля таблицы *Category* содержатся в таблице 2.5.

Таблица 2.5 ­­­– Свойства таблицы *Category*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор записи |
| *Name* | *nvarchar(max)* | Название категории |
| *Icon* | *nvarchar(max)* | Название иконки |
| *Color* | *nvarchar(max)* | Цвет иконки |
| *UserId* | *uniqueidentifier* | Уникальный идентификатор пользователя, которому принадлежит категория |
| *IsDeleted* | *bit* | Флаг, укзаывающий, удалена ли запись |

Таблица *Loans* содержит информацию о долгах пользователя. Таблица 2.6 содержит все поля таблицы *Loans*.

Таблица 2.6 ­­­– Свойства таблицы *Loans*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Флаг, который обозначает удалена ли запись |
| *Name* | *nvarchar(max)* | Имя, которое пользователь дает долгу |
| *Description* | *nvarchar(max)* | Описание долга |
| *IsMine* | *bit* | Флаг, который указывает, пользователь взял взаймы или дал в долг |
| *ReturnDate* | *datetime2* | Дата, когда долг должен быть возвращен |
| *UserId* | *uniqueidentifier* | Уникальный идентификатор пользователя |
| *Amount* | *real* | Сумма долга |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |

Таблица *Operation* содержит информацию о операциях пользователя по определенному счету. Поля таблицы содержатся в таблице 2.7.

Таблица 2.7 ­­­– Свойства таблицы *Operation*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор записи |
| *Type* | *int* | Тип операции |
| *Value* | *real* | Сумма операции |
| *Description* | *nvarchar(max)* | Описание операции |
| *CreatedDate* | *datetime2* | Дата создания |
| *SubCategoryId* | *uniqueidentifier* | Уникальный идентификатор категории |
| *AccountId* | *uniqueidentifier* | Уникальный идентификатор счета |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |

Таблица *Subcategory* используется для хранения информации о подкатегориях для операций. Таблица 2.8 содержит все поля таблицы *Subcategory*.

Таблица 2.8 ­­­– Свойства таблицы *Subcategory*

|  |  |  |
| --- | --- | --- |
| Название поля | Тип данных | Описание |
| *Id* | *uniqueidentifier* | Уникальный идентификатор подкатегории |
| *Name* | *nvarchar(max)* | Название подкатегории |
| *Icon* | *nvarchar(max)* | Название иконки |
| *Color* | *nvarchar(max)* | Название цвета иконки |
| *CategoryId* | *uniqueidentifier* | Уникальный идентификатор категории |
| *IsDeleted* | *bit* | Флаг, который обозначает удалена ли запись |

Структура таблицы *UserAims* представляет собой три поля, каждое из которых является уникальным идентификатором. Идентификатором записи, пользователя и цели.

* 1. Выводы по разделу

На данном этапе была спроектирована архитектура приложения. Была спроектирована серверная часть, ее архитектура и связь между классами и компонентами. Также было определено, какие сторонние библиотеки будут использованы в работе серверной части приложения.

Была спроектирована архитектура клиентской части приложения, решены вопросы создания хранилища данных приложения клиентской части и выделения функций библиотеки *redux-thunk* в отдельный слой приложения.

Была разработана диаграмма вариантов использования и определили все основные сценарии, как может быть использовано приложение.

Кроме того, было спроектировано хранилище данных и структурная диаграмма приложения, а также подробно описаны все спроектированные таблицы базы данных, их предназначение, предназначение всех полей каждой из таблиц и тип данных каждого из полей.

Был описан основной алгоритм проверки цели, который будет использоваться в приложении.

После проектирования можно приступать к разработке программного средства.

1. Разработка программного средства

В процессе разработки программного средства были использованы следующие технические средства.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
| Изм. | Лист | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | 3 Разработка  программного средства | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

Технические средства:

* процессор *AMD Ryzen 5 5600X*;
* графический адаптер *Asus Dual RTX 3060 V2 OC12GB*;
* материнская плата *Gigabyte b550 Aorus Elite V2 rev. 1.1*;
* оперативная память 16 ГБ, 2 модуля *DDR4* по 8 ГБ, частота 3200 МГц.

В процессе разработки для быстрой развертки различных компонентов программного средства использовалось средство контейнеризации *Docker*. Для развертывания в условиях хостингового сервера и для тестирования приложения использовалась утилита *docker-compose*, с помощью которой можно запустить и настроить сразу несколько *docker*-контейнеров.

Диаграмма развертывания приложения изображена на рисунке 3.1 и в приложении Д.

|  |
| --- |
|  |

Рисунок 3.1 – Диаграмма развертывания приложения

* 1. Серверная часть приложения
     1. Структура проекта

Для разработки серверной части был использован язык *C#* и платформа *ASP.NET Core*. Структура проекта изображена на рисунке 3.1

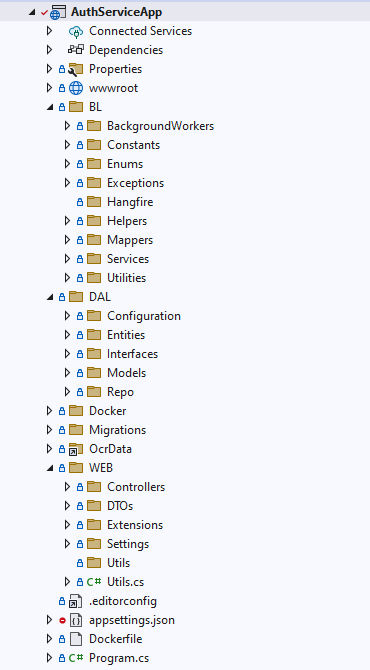


Рисунок 3.1 ­­­– Строение проекта серверной части приложения

Файлы различных слоев приложения расположены в соответствующих папках. Рассмотрим их содержимое подробнее. Внутри папки *Settings*, хранятся файлы настроек, которые импортируются из файла конфигурации проекта *appsettings.json*.

Файлы настроек инициализируются на старте приложения в файле *program.cs* – главном файле приложения. Он содержит в себе главную точку входа в приложения, выполняет важную функцию, так как запускает приложение и настраивает его основные компоненты, такие как веб-сервер и конфигурацию.

Рассмотрим содержимое папки *WEB*. Эта папка отвечает за объекты слоя представления. В папке *Controllers* содержатся классы контроллеры, которые принимают все запросы и адресуют выполнения сервисам.

В папке *DTOs* содержатся специальные классы *dto* *(data transfer object),* которые нужны для передачи данных между сервером и клиентом. Они используются для скрытия информации от клиента.

В папке *Extensions* хранятся классы, методы которых расширяют объекты платформы *ASP.NET*. Это делается для удобства разработки, поскольку улучшает читаемость файла program.cs и позволяет лучше в нем ориентироваться.

После обработки запроса приложение вызывает методы папки *BL*. Папка *Background workers* отвечает за методы для работы с *Hangfire*.

Папка *Constants* содержит в себе файлы с константами, которые используются в целях отображения сообщений об ошибках, статус-кодах.

Папка *Enums* хранит информацию о статус кодах, типах операций клиента и типе окружающих сред для развертки, например *Docker*.

Папка *Exceptions* хранит в себе определение уникальной ошибки, которая используются для отлова исключительных ситуаций в приложении и отображении этих ошибок для клиентской стороны.

Папка *Helpers* содержит классы, которые по-всякому помогают в разработке, например, помогают достать полезные данные из *JWT* токена.

Папка *Mappers* содержит в себе профили для библиотеки *AutoMapper* [20].

Папка *Services* содержит в себе классы, которые исполняют всю бизнес-логику приложения.

Папка *Utilities* содержит файлы, которые предоставляют функции для работы с отправкой почтовых сообщений и созданием токенов

Рассмотрим содержимое папки *DAL*. В папке *Configuration* хранятся данные для конфигурации таблиц с помощью *Fluent* *API* [21]. С помощью этой технологии можно более гибко конфигурировать поведение таблицы базы данных, например, настроить внешние ключи, настроить поведение связей, при удалении данных. Пример такого файла содержится в листинге 3.1.

|  |
| --- |
| public class CardConfiguration : IEntityTypeConfiguration<CardEntity>  {  public void Configure(EntityTypeBuilder<CardEntity> builder)  {  builder.HasKey(key => key.Id);  builder.Property(prop => prop.Id).IsRequired();  builder.HasMany(key => key.Spendings).WithOne(one =>  one.Card).OnDelete(DeleteBehavior.ClientCascade);  builder.HasOne(key => key.User).WithMany(one => one.Cards).OnDelete(DeleteBehavior.NoAction);  }  } |

Листинг 3.1 – пример конфигурации с помощью *Fluent* *API*.

В папке *Entities* содержатся сущности базы данных в виде классов *C#*.

В папке *Interfaces* хранятся интерфейсы для репозиториев, которые нужны для инъекции репозиториев в сервисы.

В папке *Models* хранится файл, с помощью которого происходит конфигурация базы данных и идет подключение конфигурации *Fluent* *API* из папки *Configuration*.

В папке *Repo* содержатся файлы репозиториев для работы с сущностями базы данных.

* + 1. Конфигурация и запуск проекта

Этот файл содержит в себе пары ключ-значение, которые отвечают за настройку приложения, например, хранят пароли от почтового аккаунта. Далее все настройки собираются в один файл для удобства импорта всех данных.

Пример файла настроек представлен в листинге 3.1.

|  |
| --- |
| namespace AuthServiceApp.WEB.Settings;  public class DatabaseSettings  {  public string Server { get; set; }  public string Port { get; set; }  public string Database { get; set; }  public string UserId { get; set; }  public string Password { get; set; }  public int ConnectTimeout { get; set; }  public bool Encrypt { get; set; }  public bool TrustedServerCertificate { get; set; }  public string ApplicationIntent { get; set; }  public bool MultiSubnetFailover { get; set; }  public string ConnectionString =>  $"Server={Server},{Port};" +  $"Database={Database};" +  $"User ID={UserId};" +  $"Password={Password};" +  $"Connect Timeout={ConnectTimeout};";  } |

Листинг 3.1 – Файл настроек базы данных

Файл *program.cs* является ключевым файлом для запуска и настройки *ASP.NET Core* приложения, и содержит множество возможностей для настройки веб-сервера и компонентов приложения. С его помощью можно добавлять различные компоненты для конвейера запросов, cодержимое файла представлено в приложении Г.

Внутри файла *program.cs* необходимо получить доступ к объекту *ConfigurationManager*, который реализует интерфейс *IConfiguration* и находится внутри объекта *WebApplicationBuilder*. Для удобства разработки все функции работы с коллекцией сервисов приложения, были объединены в файлы, которые используют механизм расширения методов. То есть были написаны методы, расширяющие функционал классов, которые используются в файле *program.cs*. Процесс добавления настроек приложения последовательно изображен на листингах ниже.

|  |
| --- |
| var appSettings = builder.Configuration.RegisterSettings(); |

Листинг 3.2 – Получение объекта настроек

Содержимое метода *RegisterSettings* представлено в листинге 3.3.

|  |
| --- |
| public static AppSettings RegisterSettings(this IConfiguration configuration)  {  Console.WriteLine(Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT"));  var conf = new ConfigurationBuilder()  .AddJsonFile("appsettings.json")  .AddJsonFile(  $"appsettings.{Environment.GetEnvironmentVariable("ASPNETCORE\_ENVIRONMENT") ?? "Development"}.json",  optional: true)  .AddEnvironmentVariables()  .Build();  return new()  {  Database = conf.GetSection(nameof(AppSettings.Database))  .Get<DatabaseSettings>(),  Token = conf.GetSection(nameof(AppSettings.Token))  .Get<TokenSettings>(),  SmtpClientSettings = conf.GetSection(nameof(AppSettings.SmtpClientSettings))  .Get<SmtpClientSettings>(),  GoogleAuthSettings = conf.GetSection(nameof(AppSettings.GoogleAuthSettings))  .Get<GoogleAuthSettings>()  };  } |

Листинг 3.3 – получение настроек в зависимости от переменных среды.

Файл *appsettings.json* меняется в зависимости от среды разработки. Так, этот файл будет содержать другие значения для процесса разработки и для развертки в *Docker*. Поэтому для правильной работы приложения необходимо иметь актуальный файл.

Далее содержимое этого файла в формате ключ-значение передается в метод расширения, который отвечает за внедрение всех зависимостей, которые были созданы. После того, как был создан объект-*singleton*, он может быть через конструктора при инициализации других объектов. На листинге 3.4 представлен код для внедрения написанных зависимостей.

|  |
| --- |
| public static void RegisterServices(this IServiceCollection services, AppSettings appSettings)  {  //settings  services.AddSingleton(appSettings);  //Services  services.AddScoped(typeof(IGenericService<>), typeof(GenericService<>));  services.AddTransient<IServiceManagement, ServiceManagement>();  services.AddTransient<IAccountService, AccountService>();  services.AddTransient<IOperationService, OperationService>();  services.AddTransient<ICategoryService, CategoryService>();  services.AddTransient<IAimService, AimService>();  services.AddTransient<IPictureService, PicturesService>();  services.AddTransient<ILoanService, LoanService>();  services.AddTransient<ICardService, CardService>();  services.AddTransient<IAuthService, AuthService>();  services.AddTransient<IRoleService, RoleService>();  services.AddTransient<IUserService, UserService>();  services.AddTransient<ISpendingService, SpendingService>();  services.AddTransient<IShopService, ShopService>();  //Repositories  services.AddScoped(typeof(IBaseRepository<>), typeof(BaseRepository<>));  services.AddTransient<IAccountRepository, AccountRepository>();  services.AddTransient<ICategoryRepository, CategoryRepository>();  services.AddTransient<ICardRepository, CardRepository>();  services.AddTransient<ISpendingRepository, SpendingRepository>();  services.AddTransient<IShopRepository, ShopRepository>();  services.AddTransient<IUserRepository, UserRepository>();  services.AddTransient<IEmailSender, EmailSender>();  } |

Листинг 3.4 – Внедрение зависимостей в приложении.

После того, как настроен файл *program.cs* и все сопутствующие компоненты, можно переходить к другим аспектам серверной части приложения.

* + 1. Уровень доступа к данным

Когда приложение запускается, срабатывает метод, который открывает соединение с базой данных, после этого происходит проверка моделей, которые описаны в приложении. При создании базы данных был использован подход *code-first*, при котором сначала проектируются сущности, с которыми будет работать инженер, а на основе этих моделей уже создается база данных. В файле *ApplicationContext.cs*, содержится метод, который применяет конфигурацию таблиц базы данных, написанную с помощью технологии *Fluent API*. Содержимое метода *OnModelCreating* представлено в листинге 3.5

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder builder)  {  base.OnModelCreating(builder);  builder.ApplyConfiguration(new OperationConfiguration());  builder.ApplyConfiguration(new UserConfiguration());  builder.ApplyConfiguration(new ShopPositionConfiguration());  builder.ApplyConfiguration(new SpendingConfiguration());  builder.ApplyConfiguration(new CardConfiguration());  builder.ApplyConfigurationsFromAssembly(Assembly.GetExecutingAssembly());  } |

Листинг 3.5 – Метод *OnModelCreating*, который добавляет конфигурацию таблиц.

Репозитории представляют классы, содержащие набор методов, которые реализуют определенный интерфейс. Реализация интерфейсов особенно важна в данном контексте поскольку это основной механизм внедрения зависимостей в приложении.

Поскольку работу с некоторыми сущностями приложения можно объединить и обобщить в один репозиторий, сначала был создан обобщенный интерфейс, который будет содержать набор основных методов для работы с сущностью, на основе которой будет создан репозиторий. Далее создадим базовый класс-репозиторий, который реализует этот интерфейс. Содержимое этого репозитория представлен в приложении Г.

После создания базового репозитория, уже можно внедрять его в сервисы и использовать для манипуляции над данными. Но могут возникнуть случаи, когда функционала, который содержится в базовом классе, недостаточно, а другим потомкам базового класса эти методы не нужны. В таком случае, разработчик создает новый интерфейс, описывает список методов, реализует их в новом классе, и наследуется от базового класса. Пример такого репозитория показан на листинге 3.6

|  |
| --- |
| using AuthServiceApp.DAL.Entities.Account;  using AuthServiceApp.DAL.Interfaces;  using AuthServiceApp.DAL.Models;  using Microsoft.EntityFrameworkCore;  namespace AuthServiceApp.DAL.Repo.Account;  public class AccountRepository : BaseRepository<AccountEntity>, IAccountRepository  {  public AccountRepository(ApplicationDbContext databaseContext) : base(databaseContext)  {  }  public Task<List<AccountEntity>> GetAllUserAccounts(Guid userId)  {  return Entity.Where(account => account.UserId == userId).Include(x => x.Operations).ToListAsync();  }  }  public interface IAccountRepository : IBaseRepository<AccountEntity>  {  Task<List<AccountEntity>> GetAllUserAccounts(Guid userId);  } |

Листинг 3.6 – репозиторий приложения серверной части

После создания репозиториев можно приступать к работе над бизнес-логикой приложения.

* + 1. Уровень представления

После того, как при запуске приложения был получен набор абстракций для работы над данными, можно начинать разработку бизнес-логики. Она может быть расположена в контроллерах, но в таком случае они получатся громоздкими, трудными для тестирования и поддержки. Плюсы использования отдельного слоя для таких задач неоспорима, поэтому оставим разработку бизнес-логики на потом и займемся уровнем представления.

Для мануального тестирования написанного *API* был добавлен сервис *Swagger* [22]. Он позволяет проектировать документацию для сервера в удобном формате, а также предоставляет визуальный инструмент для тестирования написанных точек.

Когда сервер принимает запрос, объект запроса проходит через конвейер обработки, модифицируется и попадает в контроллер в виде объекта. Для того, чтобы ограничить количество информации получаемой и отправляемой клиентской частью, используются *dto*. Например, клиент не сможет увидеть идентификатор сущности, если ее скрывает *dto.*

Для быстрой конвертации между сущностями базы данных и *dto* используется библиотека *AutoMapper*. Для ее использования нужно создать профиль маппинга – замена значений сущностей одного объекта на поля другого. Затем вызвать функцию *CreateMap* и указать, класс-источник и класс-назначение, на который накладываются новые значения.

Пример класса-профиля *AutoMapper* содержится на листинге 3.7

|  |
| --- |
| public class MapperProfile: Profile  {  public MapperProfile()  {  AccountMappings();  OperationMappings();  }  private void AccountMappings()  {  CreateMap<AccountEntity, AccountModel>().ReverseMap();  CreateMap<CreateAccountModel, AccountEntity>();  CreateMap<UpdateAccountModel, AccountEntity>();  }  private void OperationMappings()  {  CreateMap<OperationModel, OperationEntity>().ReverseMap();  CreateMap<CreateOperationModel, OperationEntity>();  }  } |

Листинг 3.7 – класс профиль, который настраивает библиотеку AutoMapper

Когда метод контроллера получает обработанный объект запроса, выполняется проверка получаемых параметров, и в случае их отсутствия, вызывается исключение. Затем вызывается метод, который отвечает за выполнение бизнес-логики. Для того чтобы вызвать этот метод, в контроллер при инициализации через конструктор внедряется сервис, который содержит внутри себя все методы, необходимые для работы с сущностью.

Когда метод сервиса возвращает результат выполнения, контроллер формирует ответ, определяет статус код и возвращает ответ клиенту.

Но также может возникнуть ситуация, когда внутри сервиса возникает исключительная ситуация. Например, сервис будет ожидать, что вернется из базы одна сущность, но при этом вернулось несколько. В таком случае, будет поднята уникальная ошибка, назначение которой быть отловленной конвейером обработки запросов. Код промежуточного ПО цикла обработки запроса, которое отлавливает ошибки представлен на листинге 3.8.

|  |
| --- |
| public static class ExceptionHandlerExtension  {  public static void RegisterExceptionHandler(this IApplicationBuilder app, Serilog.ILogger Logger)  {  #if DEBUG  app.UseDeveloperExceptionPage();  #endif  app.Use(async (context, next) =>  {  try  {  await next();  }  catch (ApplicationHelperException ex)  {  context.Response.StatusCode = ex.ErrorStatus switch  {  ServiceResultType.NotFound => StatusCodes.Status404NotFound,  ServiceResultType.InvalidData => StatusCodes.Status400BadRequest,  ServiceResultType.NotAuthorized => StatusCodes.Status401Unauthorized,  ServiceResultType.Forbidden=> StatusCodes.Status403Forbidden,  ServiceResultType.ServerError => StatusCodes.Status500InternalServerError,  \_ => throw ex  };  Logger.Information($"Error caught in ApplicationHelperException: {ex.Message}");  var result = new { errorMessage = ex.Message };  await context.Response.WriteAsJsonAsync(result);  }  catch (System.Exception ex)  {  Logger.Information($"{ex.Message}");  //#if DEBUG  throw ex;  //#else  context.Response.StatusCode = (int)System.Net.HttpStatusCode.InternalServerError;  var result = new { errorMessage = "Something went wrong..." };  await context.Response.WriteAsJsonAsync(result);  //#endif  }  });  }  } |

Листинг 3.8 – класс обработчика уникальных ошибок.

Этот класс расширяет функционал интерфейса *IApplicationBuilder*. Разработчик может удобно вызвать этот метод из класса *Program*.*cs*, чтобы добавить ПО в конвейер обработки запросов. Также здесь добавлен логгер, который записывает информацию об ошибке в отдельный файл, чтобы позднее разработчик мог отследить, что могло произойти не так. В случае ошибки, вызов которой был спланирован, ПО само формирует ответ и отсылает его, не передавая управление методу контроллера.

* + 1. Уровень бизнес-логики

После того, как контроллер отдает на обработку объект запроса, он попадает на уровень бизнес-логики. Там он с помощью объекта *mapper*, который реализует интерфейс *IMapper* и внедряется с помощью конструктора, приводится к объекту типа сущности, далее над ним проводятся некоторые действия, и результат обработки отправляется обратно в контроллер.

Для того, чтобы успешно выполнить все манипуляции необходимы зависимости, такие как репозиторий из которого будут браться и куда будут сохраняться сущности, другие сервисы. Для их внедрения используются конструктор, который ожидает, что в него придут объекты, которые реализуют интерфейсы определенных типов. Диаграмма классов для сервиса *AimService* изображена в приложении Ж.

Перед тем как вернуть результат выполнения метода, этот результат приводится к другому типу, чтобы ограничить доступ уровня представления к данным. Это делается с помощью того же объекта *mapper*.

Все сервисы, в целом, выполняют схожие задачи, но есть некоторые, возможности которых выделяется. Примерами таких сервисов могут служить сервис пользователей. Это обусловлено тем, что он использует внутри себя функционал, который предоставляет *Identity Server framework*, а именно классы для работы с пользователями, авторизацией. Эти классы позволяют пользователям входить в систему, выходить, менять пароли.

Для прохождения авторизации пользователь отправляет *POST* запрос в контроллер пользователей, где результат обрабатывается, и клиент получает *JWT* токен. Этот токен содержит внутри закодированный идентификатор пользователя, который вошел в систему. Далее клиент с этим идентификатором может получать информацию от сервера, посылая этот идентификатор в запросе и получая персонализированные данные. Когда пользователь выходит из приложения, его токен стирается, и пользователь не может получить доступ к *API* сервера.

Также нужно отметить использование библиотеки *Mailkit*. С ее помощью происходит уведомление пользователей о событиях системы: приближение выплаты долга, успешная регистрация, сообщение с ссылкой на подтверждение почты. Данные для работы библиотеки, такие как пароль и имя аккаунта, с которого будет идти отправка почтовых писем, берутся из класса *AppSettings*, который был внедрен в приложение и который хранит значения файла appsettings.*json*. На рисунке 3.2 можно увидеть пример электронного сообщения, которое уведомляет пользователя об успешной регистрации.

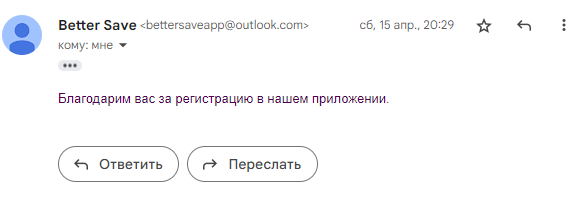


Рисунок 3.2 – Оповещение пользователя об успешной регистрации

* + 1. Класс фоновых работ

Для выполнения специфичных работ, таких как ежедневный мониторинг выполнения целей, актуальности целей пользователей и их нотификация, мною используется библиотека *Hangfire*. Она предоставляет по для работы с фоновыми работами. Эта библиотека также предоставляет интерфейс для мониторинга и анализа выполнения фоновых работ. Изображение интерфейса сервиса фоновых работ содержится на рисунке 3.3.

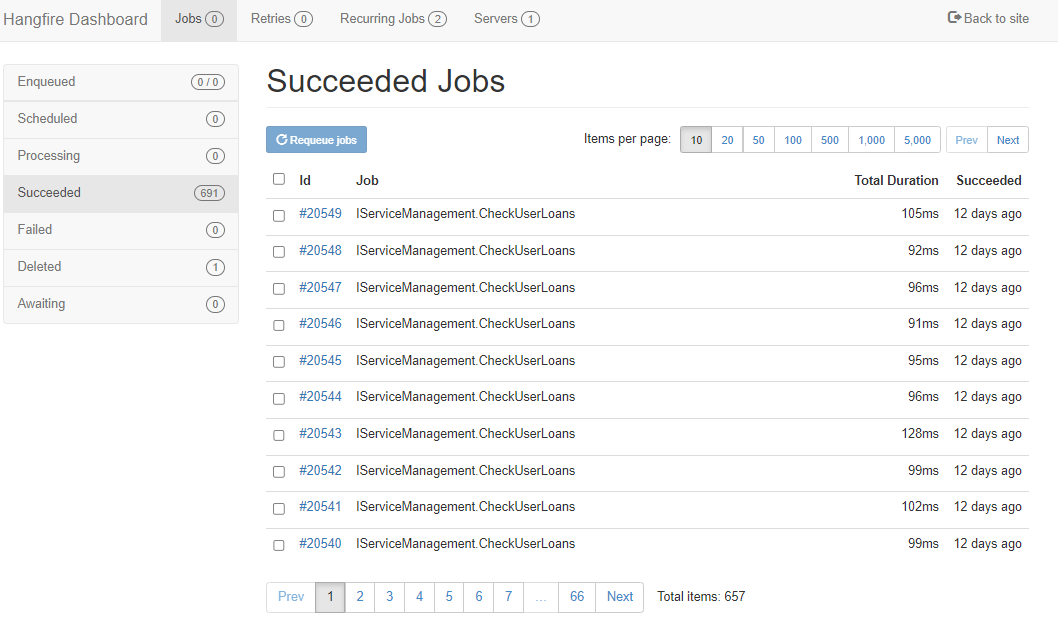


Рисунок 3.3 – интерфейс управления фоновыми работами

В приложении есть три операции, которыми занимается *Hangfire*. Во-первых, это оповещение пользователя после успешной регистрации, во-вторых, это оповещение пользователей, если приближается срок выплаты долга, в-третьих, анализ выполнения целей и создание объектов прогресса в дни, когда цели пользователей выполняются. Последние две операции выполняются с частотой в один день. Пример запуска функции через *Hangfire* содержится на листинге 3.9

|  |
| --- |
| [HttpGet("email-confirmation")]  [ProducesResponseType(StatusCodes.Status204NoContent)]  [ProducesResponseType(StatusCodes.Status400BadRequest)]  public async Task<IActionResult> ConfirmEmail(string id, string token)  {  var confirmResult = await \_authService.ConfirmAsync(id, token);  var jobId = BackgroundJob.Enqueue<IServiceManagement>(x => x.SendSuccessMessage(confirmResult));  return Ok("Confirmed");  } |

Листинг 3.9 – Запуск функции через *Hangfire*.

* 1. Клиентская часть приложения
     1. Структура проекта

Структура проекта является ключевым элементом для успешного и эффективного развития и поддержки проекта на *React*. Это важно по таким причинам, как улучшение читаемости кода, легкость поддержки, облегчение масштабирования. Структура проекта клиентской части изображена на рисунке 3.4

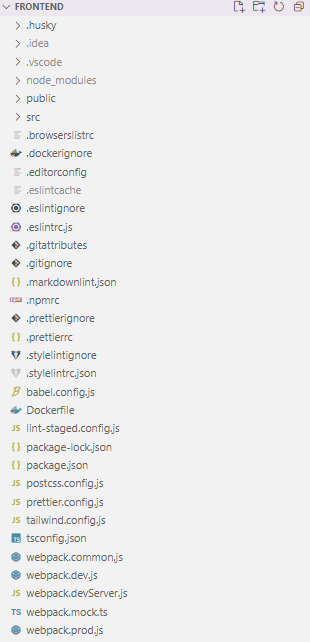


Рисунок 3.4 – Структура проекта клиентской части

По рисунку можно увидеть обилие файлов конфигурации, которые необходимы для правильной работы проекта. Разберем поочередно для чего нужны папки и файлы проекта.

Папка .*husky* содержит конфигурационную информацию для *Husky*, определяющую, какие скрипты должны быть запущены при определенных операциях *Git* [23]. Например, при фиксации изменений *(commit)* или при загрузке *(push)* изменений в удаленный репозиторий. В случае нашей программы этот файл занимается запускает программу-линтер, которая проверяет все файлы проекта и запрещает фиксировать изменения, если в программе найдены ошибки. Это заставляет ответственнее подходить к написанию кода и приучает отправлять в рабочий репозиторий только работающий код.

Папка .*vscode* содержит настройки для среды разработки *Visual Studio Code*, такие как действие на нажатие клавиши сохранения и т.д.. В данном случае, эти действия запускают библиотеку *prettier*, которая приводит код на странице к определенному виду, который описан правилами приложения.

Папка *node\_modules* содержит код библиотек зависимостей проекта, которые подключаются в приложении.

В папке *public* хранятся различные источники медиаконтента, такие как логотипы, картинки, фоны к приложению, шрифты.

В папке *src* хранится весь код к приложению. Структура папки *src* представлена на рисунке 3.5.

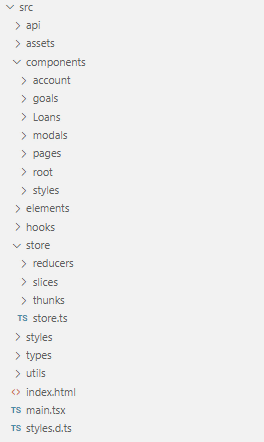


Рисунок 3.5 – структура папки src

В папке *api* хранится набор модулей, которые посылают запросы к серверу.

Папка *assets* содержит изображения, которые используются в приложении.

Папка компонент содержит в себе все структурные блоки приложения.

Папка *elements* хранит в себе элементы страницы, которые могут быть многократно использованы повторно в приложении в разных компонентах.

В папке *hooks* расположены функции для работы с механизмом *react*, называемым хуками. Хуки являются функциями, которые обращаются к *API React* и могут использовать другие функции, но при этом не могут находиться нигде, кроме функциональных компонентов. Поэтому нельзя просто написать модуль, который экспортирует функцию, но нужно написать хук.

Далее в папке *store*, хранится функционал для работы с уровнем доступа к данным. Здесь расположены файлы для работы с *redux*, *redux-toolkit*, *redux-thunk*, а именно для создания хранилища данных, функций-обработчиков *(reducers),* которые манипулируют данными, и функций, которые отправляют запросы на сервер, используя модули папки *api*.

Папка *styles* хранит набор стилей приложения, различные *scss* переменные, такие как цветовая палитра приложения. Также здесь подключаются стили библиотеки *tailwind* [24].

В папке *types* находятся файлы, содержимое которых определяет все классы приложения, интерфейсы и типы. Это преимущество языка *TypeScript* над *JavaScript*, поскольку разработчик во время написания кода может видеть, какие поля содержит тот или иной объект, а также с помощью инструмента *ESLint* проверить написал ли он код правильно еще до запуска приложения.

* + 1. Настройка *Webpack*

Для запуска приложения используется команда описанная в файле *package.json*. Эта команда запускает пакет *Webpack*. Пакет считывает данные из файлов настроек проекта и понимает, каким образом ему нужно собрать все файлы проекта в одну сборку. Существуют файлы настроек для каждого типа среды: среда разработки и среда развертки приложения.

Для среды разработки существует пакет, который запускает приложение *Webpack* в режиме быстрой перезагрузки, что позволяет моментально видеть изменения на странице, что важно в разработке клиентской части.

Для среды публикации существует специальный файл настроек, с помощью которого *Webpack* максимально оптимизирует получающуюся сборку, что уменьшает размер конечного пакета. В ходе этого процесса убираются пакеты, которые использовались в процессе разработке, уменьшается размер файлов стилей, сжимаются файлы, которые используются в проекте.

Настройки для среды развертки приложения позволяют уменьшить время загрузки приложения, что существенно влияет на пользовательский опыт, ведь никто из пользователей не хочет, чтобы доступ к их приложению занимал много времени. Быстродействие – это ключ к победе над конкурентами.

* + 1. Разработка компонентов

Главным плюсом приложений на *react* является возможность повторного использования программных компонентов. Грамотное использование компонентов может сыграть большую роль во времени и ресурсах, затраченных на разработку пользовательского интерфейса.

В приложении есть главный компонент *App*, который содержит в себе обертку приложения компонентом *redux*, который предоставляет в дочерние элементы дерева хранилище данных и методы для работы с ним, также в этом компоненте содержится другой компонент, который отвечает за маршрутизацию в приложении, путем сопоставления маршрута в браузере и отрисовываемого компонента. Также тут содержится компонент библиотеки *react*-*toastify*, который встраивает в разметку страницы элементы, которые сигнализирует о статусе выполнения запросов к серверному *API*, когда они выполняются, путем показа уведомлений в правой части экрана [25]. Содержимое компонента *App* находится в листинге 3.9.

|  |
| --- |
| import store from "@/store/store";  import { ToastContainer } from "react-toastify";  import { Provider } from "react-redux";  import AppRouter from "./appRouter";  import "rsuite/dist/rsuite.css";  import "slick-carousel/slick/slick.css";  import "slick-carousel/slick/slick-theme.css";  import "react-toastify/dist/ReactToastify.css";  function AppRoot() {    return (      <Provider store={store}>        <AppRouter />        <ToastContainer />      </Provider>    );  }  export default AppRoot; |

Листинг 3.9 – Содержимое компонента *App*.

Также здесь происходит добавление глобальных стилей для используемых библиотек приложения. Это одна из отличительных функций react. Разработчик может писать стили, которые в зависимости от компонента и способа написания могут быть как глобальными, так и инкапсулированными внутри своего компонента. Это заключается в том, что модульные стили при сборке приложения сокращаются и изменяются сборщиком, после чего даже два одинаковых класса определенные в разных компонентах не будут перекрывать друг друга.

Маршрутизация приложения настроена таким образом, что, если у пользователя нет *JWT* токена для доступа к определенному маршруту в приложении, то его перебрасывает на страницу с авторизацией в приложении. После чего устанавливается токен, и пользователь может пользоваться приложением.

* + 1. Менеджер состояний *Redux*

Отличием от обычного порядка разработки хранилища на *redux*, является использование пакета *redux-toolkit*. С его помощью многие объекты можно создать, вызвав определенную функцию. Так, чтобы создать *reducer*, которые будет определять, какие действия происходят с данными при вызове той или иной функции, необходимо вызвать функцию *createSlice*, передав в нее состояние функции пакета *redux-thunk*, при котором будет срабатывать необходимая функция, а также функцию, которая будет производить манипуляции над данными.

На листинге 3.11 содержится процесс создания функции-обработчика для хранения информации о пользователе и способах авторизации.

|  |
| --- |
| const authSlice = createSlice({    name: "auth",    initialState: initState,    reducers: {      logout(state) {        state.user = initState.user;        state.authStatus = "notauthenticated";      },    },    extraReducers: (builder) => {      builder.addCase(loginThunk.fulfilled, (state, action) => {        if (action.payload !== null) {          state.user = action.payload;          state.authStatus = "authenticated";        }      });      builder.addCase(getUserInfoThunk.fulfilled, (state, action) => {        if (action.payload !== null) {          state.user = action.payload;          state.authStatus = "authenticated";        }      });    },  });  export default authSlice.reducer; |

Листинг 3.11 – Функция для изменения информации о пользователе.

* + 1. Добавление стилей

Стили приложения в основном написаны на библиотеке *tailwind*. Также для сложных случаев, которые не предусмотрены библиотекой, были написаны собственные модульные стили, а также переписаны оригинальные стили библиотеки. Например, для анимации на странице входа в приложении.

Механизм разработки стилей в приложении представляет собой сборку интерфейса по проектированному дизайну, путем итеративного подхода, когда разработчик постепенно добавляет классы в атрибут *className*.

Библиотека *tailwind* предоставляет набор классов, который покрывает практически все доступные стили *css*, исключением являются стили, которые могут быть недоступны в различных браузерах. Названия классов происходят от стилей, которые они реализуют. Пример класса *tailwind* и свойства, которые он реализует изображены на рисунке 3.6.



Рисунок 3.6 –Классы *tailwind* и их реализация в библиотеке

С помощью этой библиотеки были настроены такие аспекты приложения как отступы между компонентами верстки, размер шрифтов, их вес, способ расположения элементов, их порядок и т.д.

Также библиотека *tailwind* предоставляет набор классов для создания простой анимации в приложении, например, пульсации некоторых компонентов, что может обратить внимание пользователей к компоненту, например, показать, что у поля ввода есть описание, которое он может прочитать, чтобы лучше понимать, за что отвечает вводимое им значение.

* 1. Выводы по разделу

Руководствуясь раннее спроектированной архитектурой, были разработаны приложения серверной и клиентской частей. В качестве источника данных серверная часть использует раннее спроектированную базу данных. Было достигнуто взаимодействие между различными частями приложения.

Серверная часть отвечает за обработку запросов клиентского приложения, выполняет всю бизнес-логику приложения и отправляет данные для просмотра клиенту. В качестве протокола передачи данных между клиентом и сервером используется *https*, а в качестве формата передачи используется *json*, ввиду простоты чтения его клиентской частью.

Клиентская часть предоставляет интерфейс пользователю для чтения, изменения, создания и удаления контента. Она состоит из повторно используемых компонентов, что позволяет составить унифицированный интерфейс и ускорить разработку, а также улучшает структуру проекта и читаемость кода. Также клиентское приложение содержит в своей структуре компонент, который отвечает за маршрутизацию запросов и отрисовку соответствующих страниц.

После того как пройден этап разработки, можно заняться тестированием приложения.

1. Тестирование приложения

В ходе проверки соответствия разработанного программного средства применяется несколько способов тестирования: *unit*-тестирование и мануальное тестирование.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | Abstract | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

*Unit*-тестирование представляется в качестве процесса тестирования отдельных блоков кода (или «юнитов») для проверки их корректности и соответствия спецификации.

Такой способ тестирования выбран поскольку помогает увеличить надежность приложения, так как юнит тестирование имеет четкий критерий качества приложения. Этот критерий называется процент покрытия тестами. Поэтому высокий процент покрытия приложения тестами может являться знаком того, что приложение не сломается в неподходящий. Но нужно иметь в виду, что даже 100% покрытый код может сломаться.

Мануальное тестирование рассматривается как процесс проверки функциональности и качества программного продукта вручную, без использования автоматизированных инструментов или скриптов.

Гибкость и адаптивность мануального тестирования является ключевым фактором в выборе способа проверки программного средства на предмет соответствия норме. Мануальное тестирование может быть очень гибким и адаптивным к требованиям проекта, поскольку тестировщик может реагировать на изменения в продукте и изменять тест-кейсы по мере необходимости.

Недостатком мануального тестирования является то, что оно требует больших затрат времени и усилий, поскольку тестирование выполняется вручную. Также мануальное тестирование может быть не эффективным в случае больших и сложных проектов.

* 1. Тестирование серверной части приложения

Над серверной частью было произведено unit-тестирование и мануальное тестирование через утилиту *Postman*.

* + 1. Unit тестирование приложения

Unit-тестирование серверной части приложения проводилось с помощью библиотек *xUnit*, *FakeItEase*, *AutoFixture*.

Библиотека *xUnit* предоставляет инструменты для создания и выполнения unit-тестов [26].

Библиотека *FakeItEasy* используется для создания автоматических заглушек, которые используются в unit-тестировании для имитации поведения внедренных компонентов [27].

Библиотека *AutoFixture* позволяет автоматически создавать объекты с заполненными свойствами и значениями, уменьшая затраты на написание тестовых данных вручную [28].

Использование последних двух библиотек предоставляет необходимые данные для имитации работы методов, которые вызывает тестируемый сервис. Это необходимо, чтобы сосредоточиться на уровне логики и чтобы рассмотреть различное поведение метода, при разных входных данных.

При написании тестов использовался популярный подход *AAA* *(Arrange-Act-Assert)*, который заключается в упорядочивании и структурировании содержимого теста, что делает его более понятным и легко читаемым [29].

На этапе *Arrange* происходит подготовка необходимых данных и настройка тестируемой среды для выполнения тестового сценария. На этом этапе создаются необходимые объекты, инициализируются переменные и выполняются другие необходимые действия.

Этап *Act* – это этап, на котором запускается код, который должен быть протестирован.

На этапе *Assert* происходит проверка ожидаемого результата, выполняемого на прошлом этапе, с фактическим.

Такой подход к unit-тестированию помогает разделить процесс написания тестов на части, упорядочивая их и делая код более понятным и читаемым.

В среде программистов хорошим значением для процента покрытия тестами является значение в 80%.

Был протестирован главный уровень серверной части приложения, а именно уровень бизнес-логики. Его покрытие тестами играет наибольшую роль в тестировании серверной части, поскольку именно этот уровень содержит методы для реализации бизнес-процессов.

Пример unit-теста содержится в листинге 4.1

|  |
| --- |
| public async Task ShouldReturnNotFoundOnGetEntity()  {  //Arrange  var repository = A.Fake<IBaseRepository<LoanEntity>>();  A.CallTo(() =>  repository.SearchForSingleItemAsync(  A<Expression<Func<LoanEntity, bool>>>.Ignored))!  Returns(Task.FromResult<LoanEntity>(null));  var mapper = A.Fake<IMapper>();    var loanService = new LoanService(repository, mapper);  var fixture = new Fixture(){  Behaviors = { new NullRecursionBehavior() }  };  var loan = fixture.Create<LoanEntity>();  //Act  async Task Action()  {  await loanService.GetLoanById(loan.UserId);  }  //Assert  await Assert.ThrowsAsync<ApplicationHelperException>(Action);  } |

Листинг 4.1 – unit-тест сервиса долгов

В приведенном примере ожидается, что метод выбросит ошибку об исключительной ситуации, которую в будущем подхватит обработчик и отправит ее пользователю. С помощью библиотек было указано, что вызов функции внутри тестируемого фрагмента вернет специальное значение типа *null*. Что послужит сигналом для отправки исключения.

Тестируемый метод содержится в листинге 4.2.

|  |
| --- |
| public async Task<LoanDto> GetLoanById(Guid id)  {  var result = await \_loanRepository.SearchForSingleItemAsync(loan =>  loan.Id == id);  if (result is null)  {  throw new ApplicationHelperException(ServiceResultType.NotFound,  ExceptionMessageConstants.NotFound);  }  var mapped = mapper.Map<LoanDto>(result);  return mapped;  } |

Листинг 4.2 – Содержимое тестируемого метода.

В ходе тестирования процент покрытия слоя бизнес-логики достиг Х процентов, это можно увидеть на рисунке 4.1.

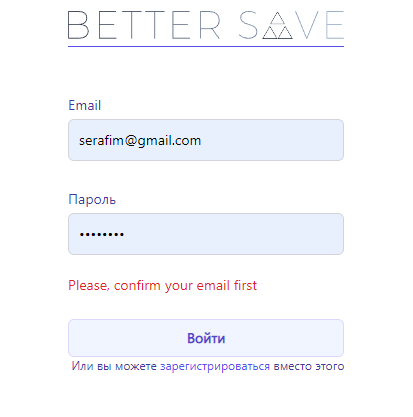


Рисунок 4.1 – Процент покрытия слоя бизнес-логики *unit*-тестами

* + 1. Мануальное тестирование Web API

Для мануального тестирования серверной части была выбрана утилита *Postman*. Была создана коллекция *api*, которая содержит в себе папки для тестирования каждого контроллера серверного приложения. Структуру коллекции можно увидеть на рисунке 4.2.

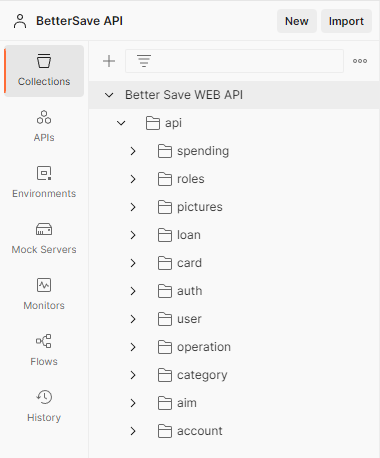


Рисунок 4.2 – Структура коллекции *api*

Каждая папка содержит в себе набор запросов, для каждого метода класса контроллера. Интерфейс *Postman* позволяет протестировать весь контроллер целиком, отправив набор запросов, нажав на кнопку *Run folder.*

Пример выполнения коллекции запросов изображен на рисунке 4.3. По результатам запросов видно, что часть запросов была выполнена успешно. Также видно, что часть, где в параметре запроса допущена ошибка, выполнена с кодом 404, что означает, что сущность, поиск которой зависит от передаваемого параметра, была не найдена.

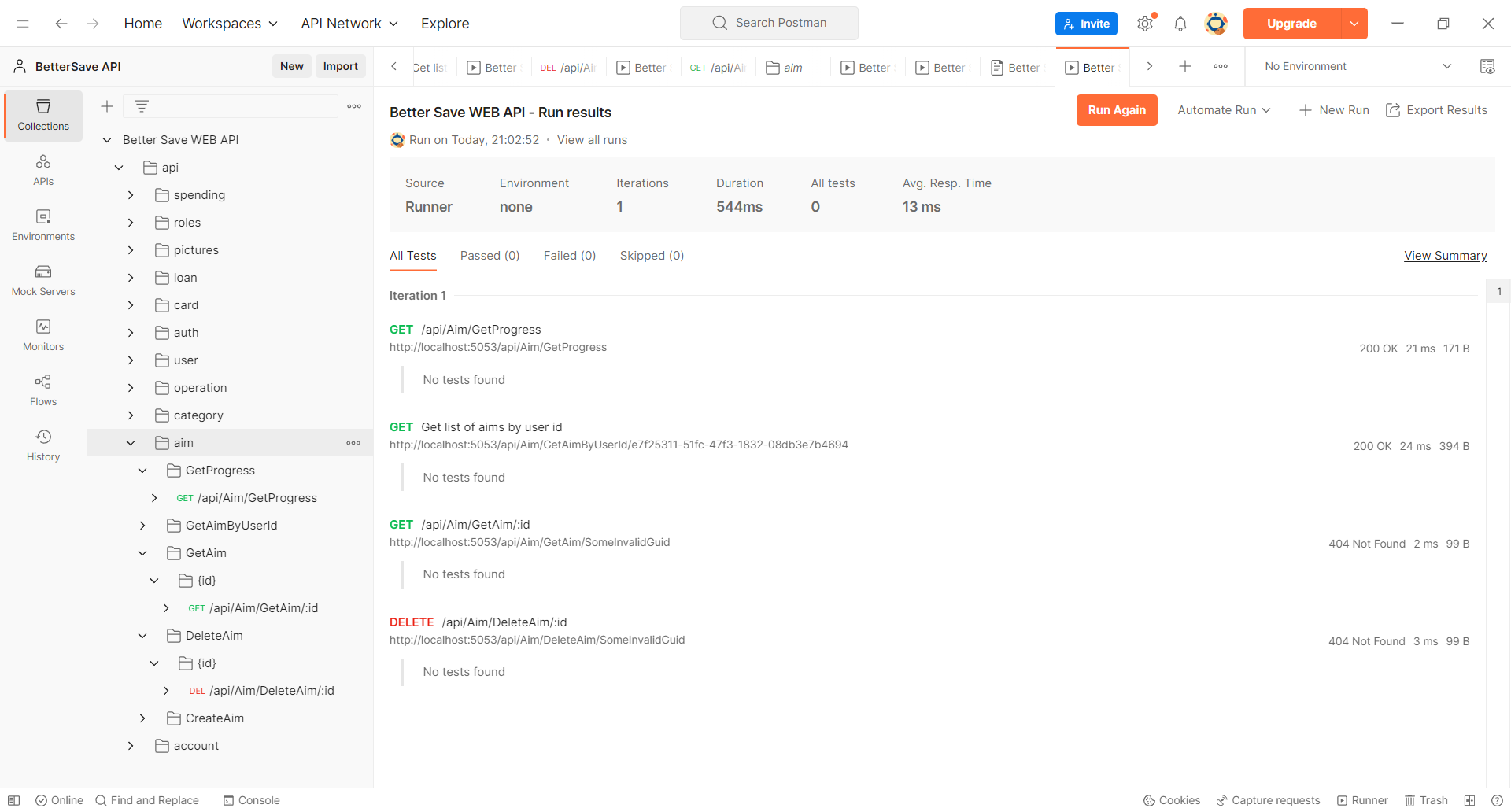


Рисунок 4.3 – Результат выполнения коллекции запросов

Если исправить ошибку в идентификаторе записей, результат исполнения коллекции запросов исправится. В таком случае ответы от сервера будут содержать только статус коды, которые говорят о положительном результате операций.

Пример исполнения коллекции запросов с правильными значениями изображен на рисунке 4.4.

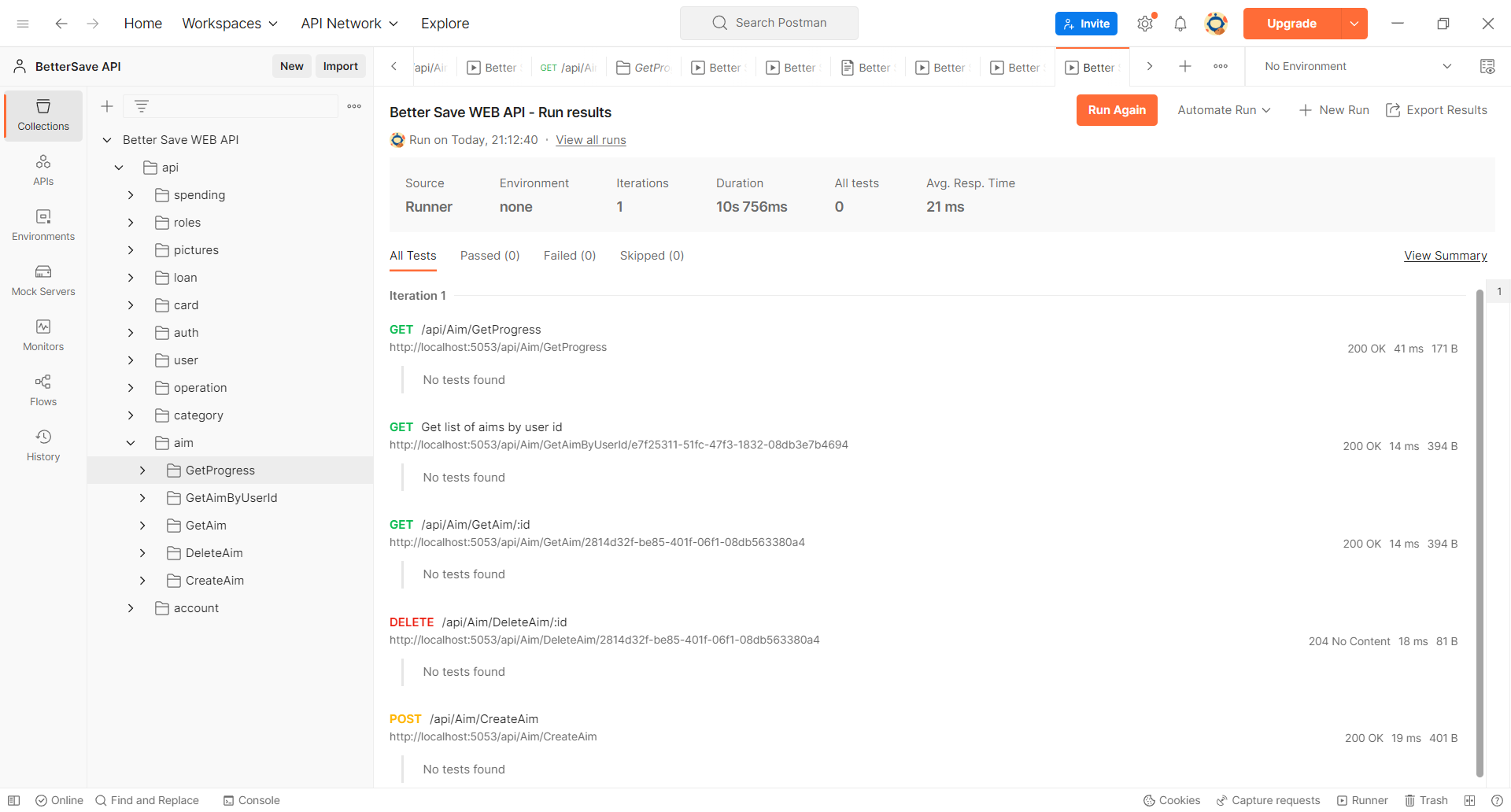


Рисунок 4.4 – Положительный результат выполнения коллекции запросов

Также одним из сценариев тестирования *API* может быть отправка запросов к защищенным маршрутам серверного приложения. Так на рисунке 4.5 изображен результат запроса без указания токена аутентификации.

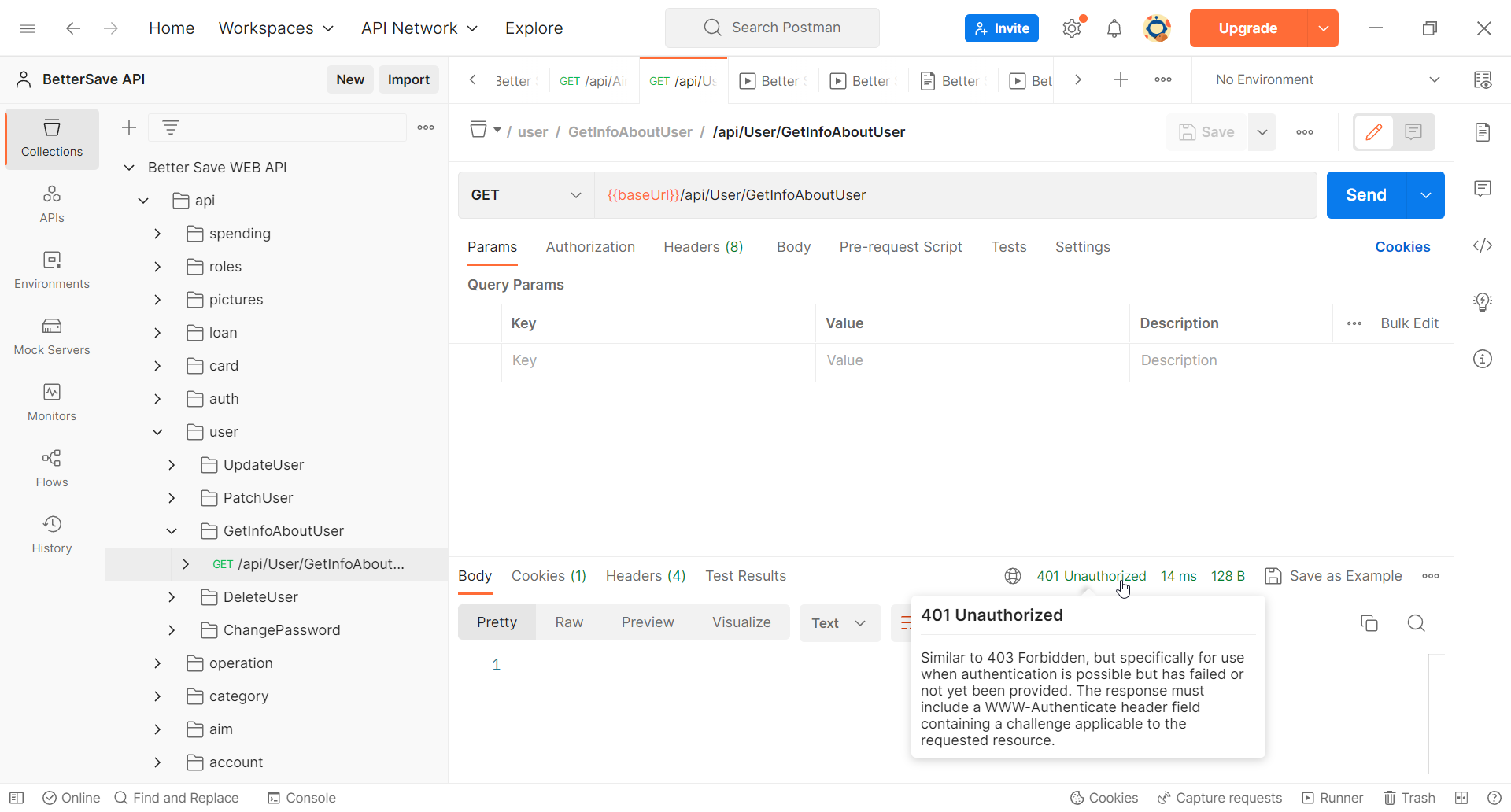


Рисунок 4.5 – Результат запроса без токена аутентификации

После добавления токена результат выполнения запроса меняется. Пример ответа серверного приложения изображен на рисунке 4.6.

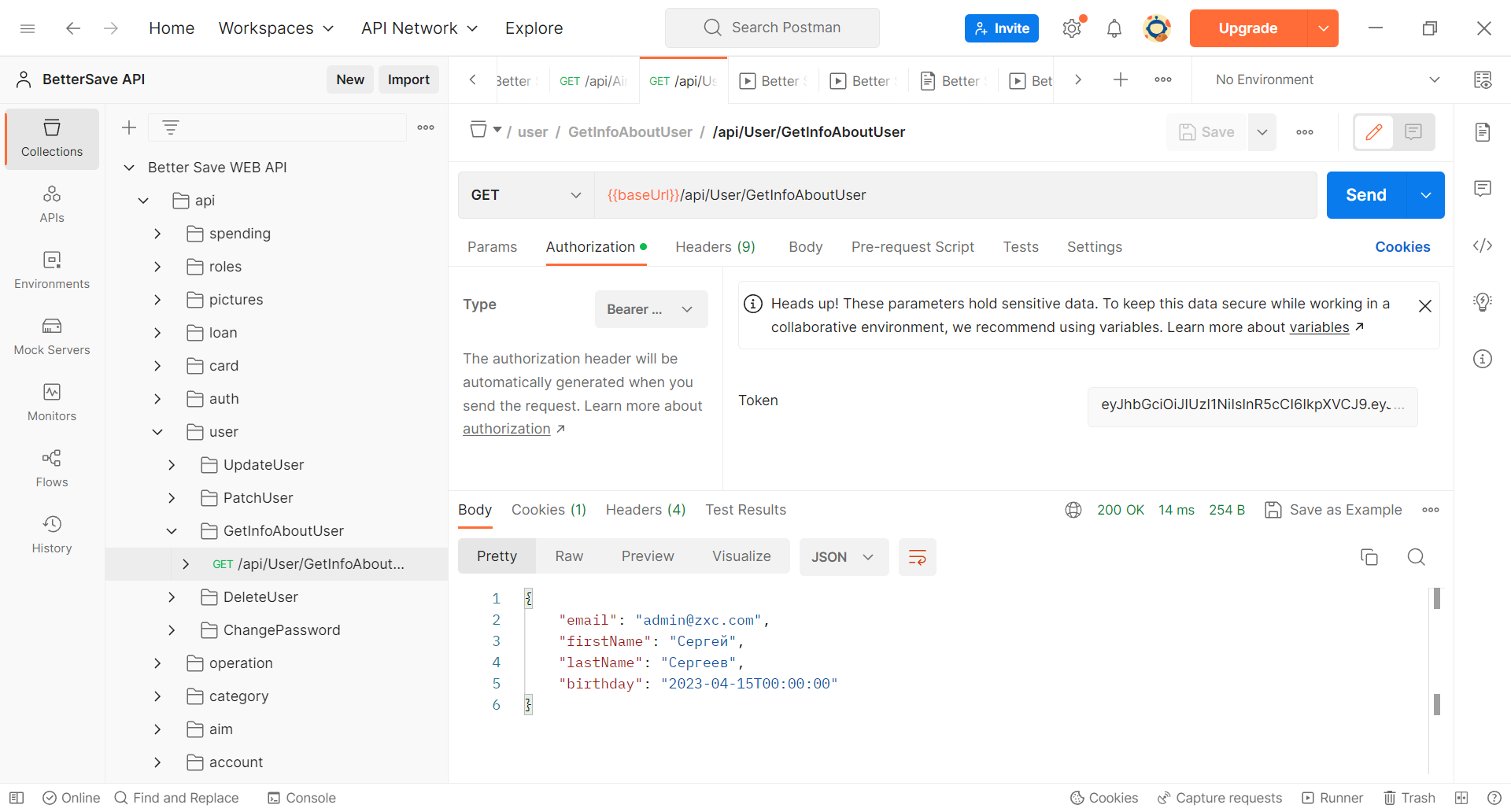


Рисунок 4.6 – Положительный результат запроса о пользователе

* 1. Тестирование клиентской части

Над клиентской частью было проведено мануальное тестирование.

Для проверки поведения программного средства при вводе данных, не соответствующих формату, в заполняемые формы были проведены тесты на контрольных примерах. Для выявления ошибок необходимо было выполнить определенные действия, сравнить полученные результаты с ожидаемыми и сделать вывод о наличии ошибок. Для этого были разработаны тестовые случаи. Список тестовых случаев содержится в таблице 4.1.

Таблица 4.1 – Тестовые случаи

|  |  |
| --- | --- |
| Описание тестового случая | Ожидаемый результат |
| Попытка войти в приложение, не введя необходимые данные | Показ информационного сообщения рядом с незаполненными полями |
| Попытка войти в приложение под незарегистрированным пользователем | Вывод сообщения, что пользователь с такой почтой не найден |
| Попытка ввести адрес электронной почты неправильного вида во время входа в приложение | Вывод сообщения об ошибке в формате адреса |
| Попытка ввести слишком длинный адрес электронной почты при входе в приложение | Вывод сообщения об ограничении длинны адреса |
| Попытка войти, указав неправильный пароль | Вывод сообщения о неправильном пароле |
| Попытка войти в приложение, не активировав учетную запись на почте | Вывод сообщения о необходимости активировать учетную запись на почте |
| Попытка зарегистрироваться в приложении, не указав обязательные поля | Показ информационного сообщения рядом с незаполненными полями |
| Попытка зарегистрироваться, указав адрес электронной почты в неправильном формате | Вывод сообщения об ошибке в формате адреса |
| Попытка зарегистрироваться, указав пароль неправильного формата | Вывод сообщения об ошибке в формате пароля |
| Попытка зарегистрироваться с электронным адресом уже зарегистрированного пользователя | Вывод сообщения о существовании пользователя с таким адресом электронной почты |

После создания и описания тестовых сценариев можно начинать мануальное тестирование клиентской части по тестовым случаям и сравнивать полученный результат и ожидаемый.

Протестируем формы входа и регистрации в приложении.

Введем пустые значения в форму для входа. Результат теста соответствует ожидаемому результату и представлен на рисунке 4.7

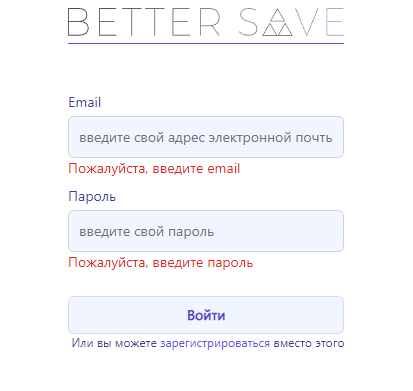


Рисунок 4.7 – Результат тестирования ввода пустых полей

Далее протестируем возможность входа пользователем, у которого не активирована учетная запись. Результат теста соответствует ожидаемому результату и представлен на рисунке 4.8

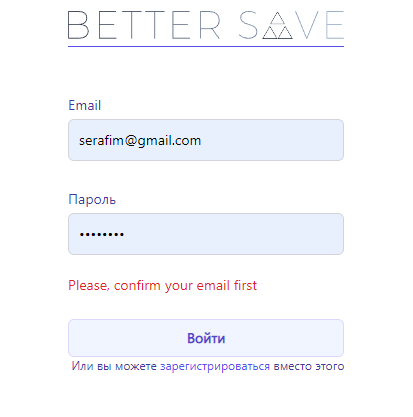


Рисунок 4.8 – Результат теста входа пользователем, который не активировал учетную запись

Следом протестируем возможность зарегистрироваться, указав пароль неправильного формата, в ожидании получить сообщение об ошибке о неправильности введенных данных. Результат соответствует ожидаемому результату и представлен на рисунке 4.9

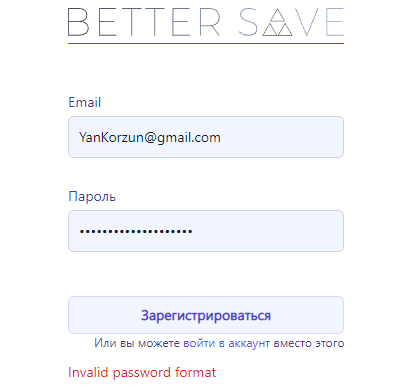


Рисунок 4.9 – Результат теста регистрации пользователя, указав пароль неправильного формата.

* 1. Выводы по разделу

В процессе разработки проекта тестирование является важной составляющей, которая необходима для обеспечения правильной работы приложения. Тестирование выполняется с целью проверки того, что функции приложения работают ровно так, как планировалось. Продукт допускается к использованию, только в том случае, если все тестовые сценарии пройдены успешно

В данном разделе было проведено unit-тестирование уровня бизнес-логики, составлены тестовые случаи, проведено мануальное тестирование клиентской и серверной части. Тестовые случаи описывают шаги для тестирования конкретного функционала, а также ожидаемый результат.

При проведении тестирования приложения были рассмотрены самые распространенные исключительные ситуации. Согласно всем проведенным тестам, приложение работает исправно.

1. Руководство пользователя

Когда пользователь первый раз открывает веб-приложение он видит окно приветствия (рисунок 5.1). На данной странице пользователь видит список возможностей приложения. Пользователь имеет возможность перейти на страницы входа и регистрации.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | БГТУ 00.00 ПЗ | | | | | |
|  |  |  |  |  |
|  |  | ФИО | Подпись | Дата |
| Разраб. | | Сергеев С. А. |  |  | Abstract | Лит. | | | Лист | Листов |
| Пров. | | Пахолко А. С. |  |  |  |  |  | 1 | 1 |
| Консульт. | |  |  |  | *74417005, 2023* | | | | |
| Н. контр. | | Грудо С. К. |  |  |
| Утв. | | Пацей Н. В. |  |  |

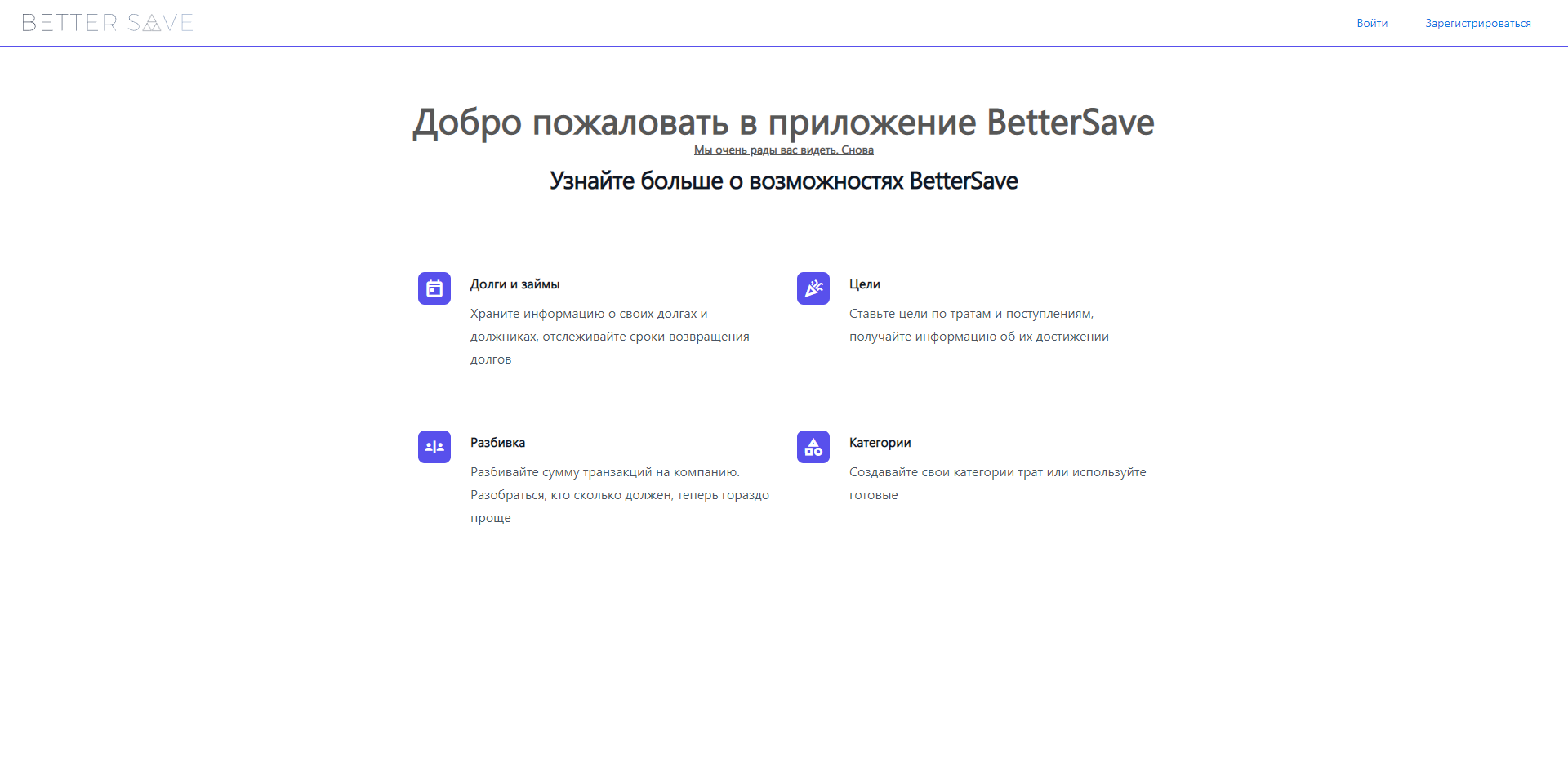


Рисунок 5.1 – Экран приветствия

Перейдя на страницу регистрации, пользователь видит страницу, изображенную на рисунке 5.2. На ней новый пользователь может создать учетную запись в приложении.

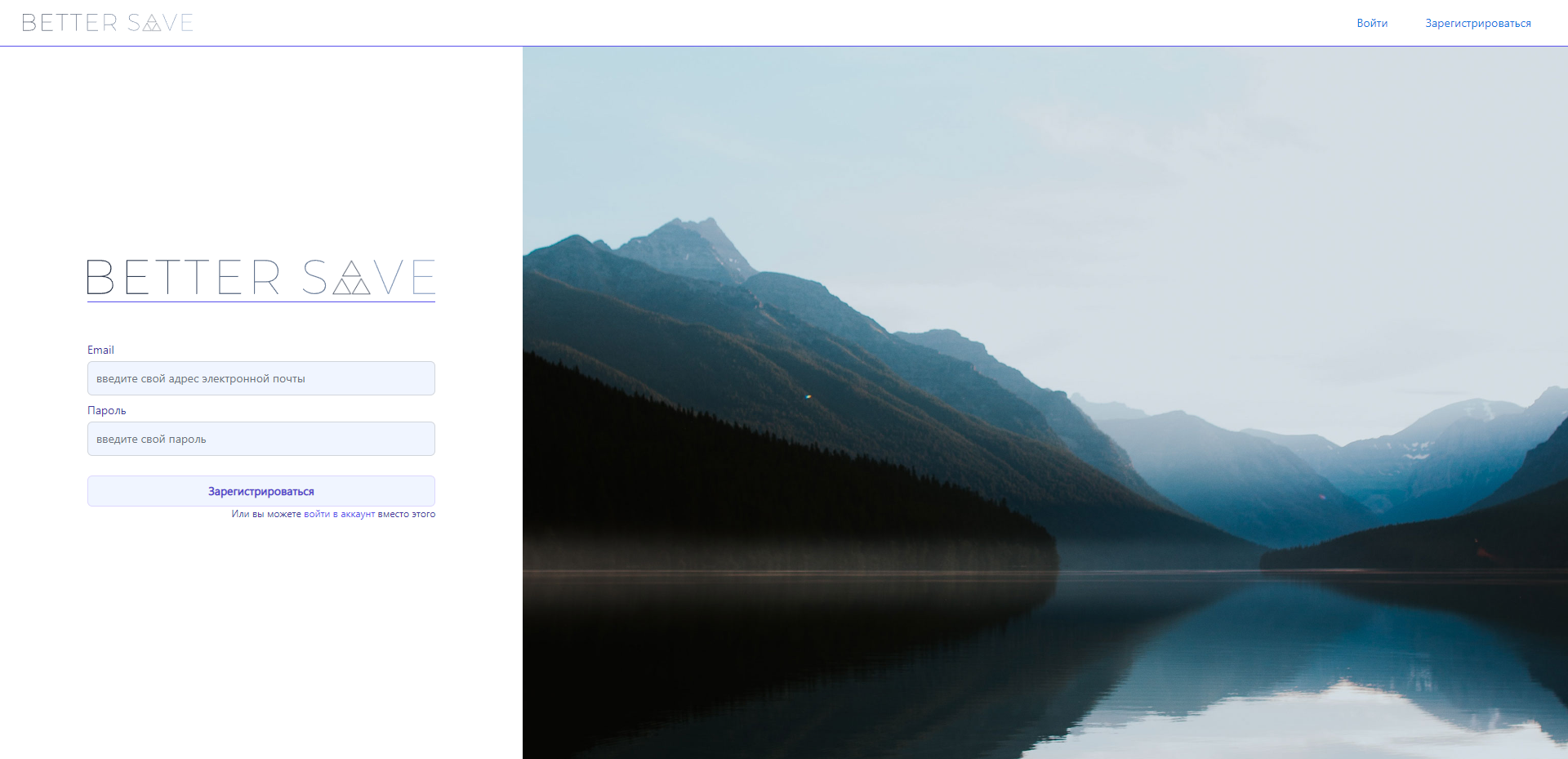


Рисунок 5.2 – Окно регистрации

После регистрации пользователь оказывается переадресован на страницу, где видит сообщение с просьбой активировать свою учетную запись. Эта страница изображена на рисунке 5.3.

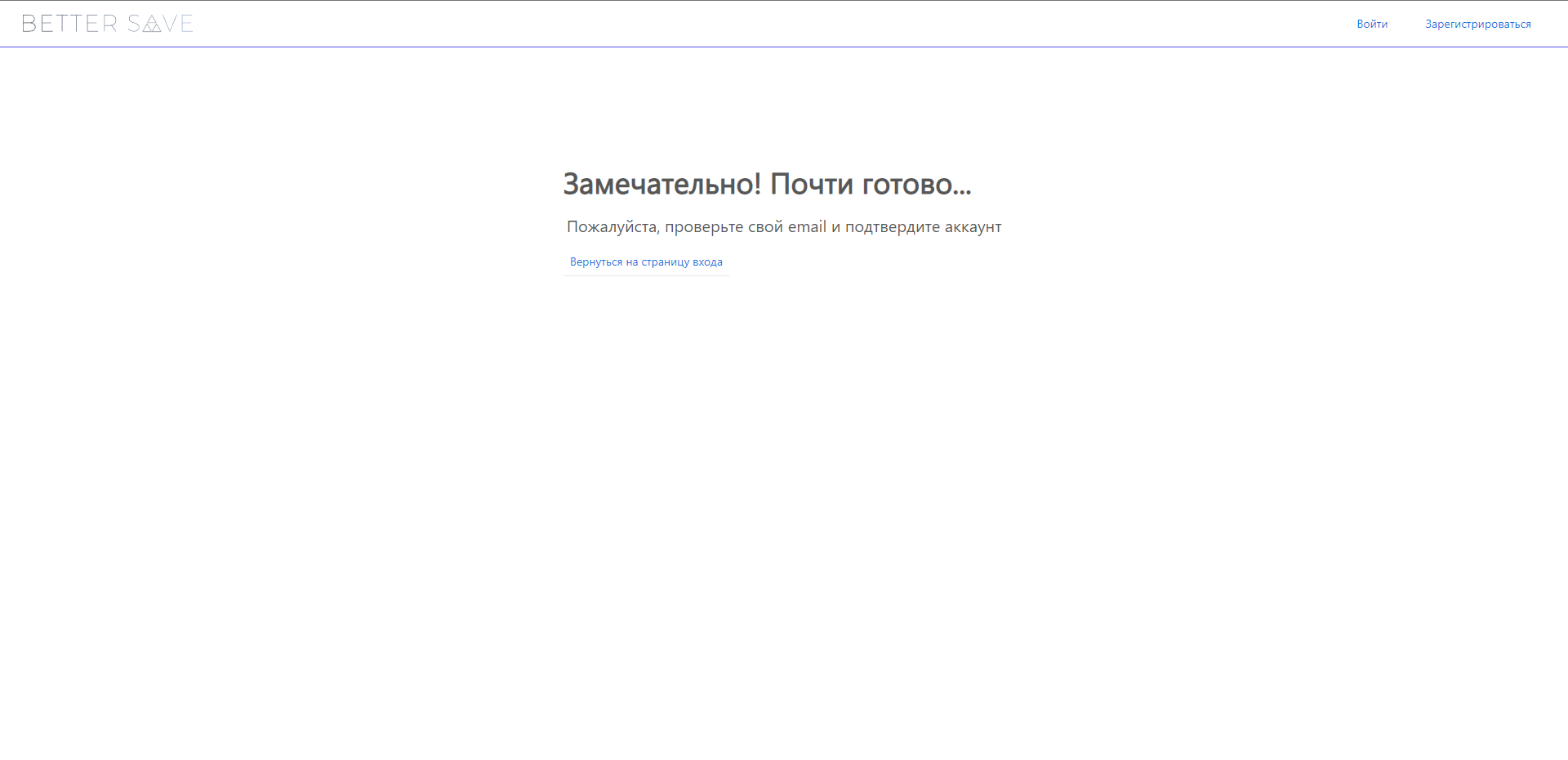


Рисунок 5.3 – Страница с инструкциями после регистрации

После того, как пользователь перешел на почту и перешел по ссылке для подтверждения учетной записи. Он может перейти на страницу со входом в приложение. Она изображена на рисунке 5.3.

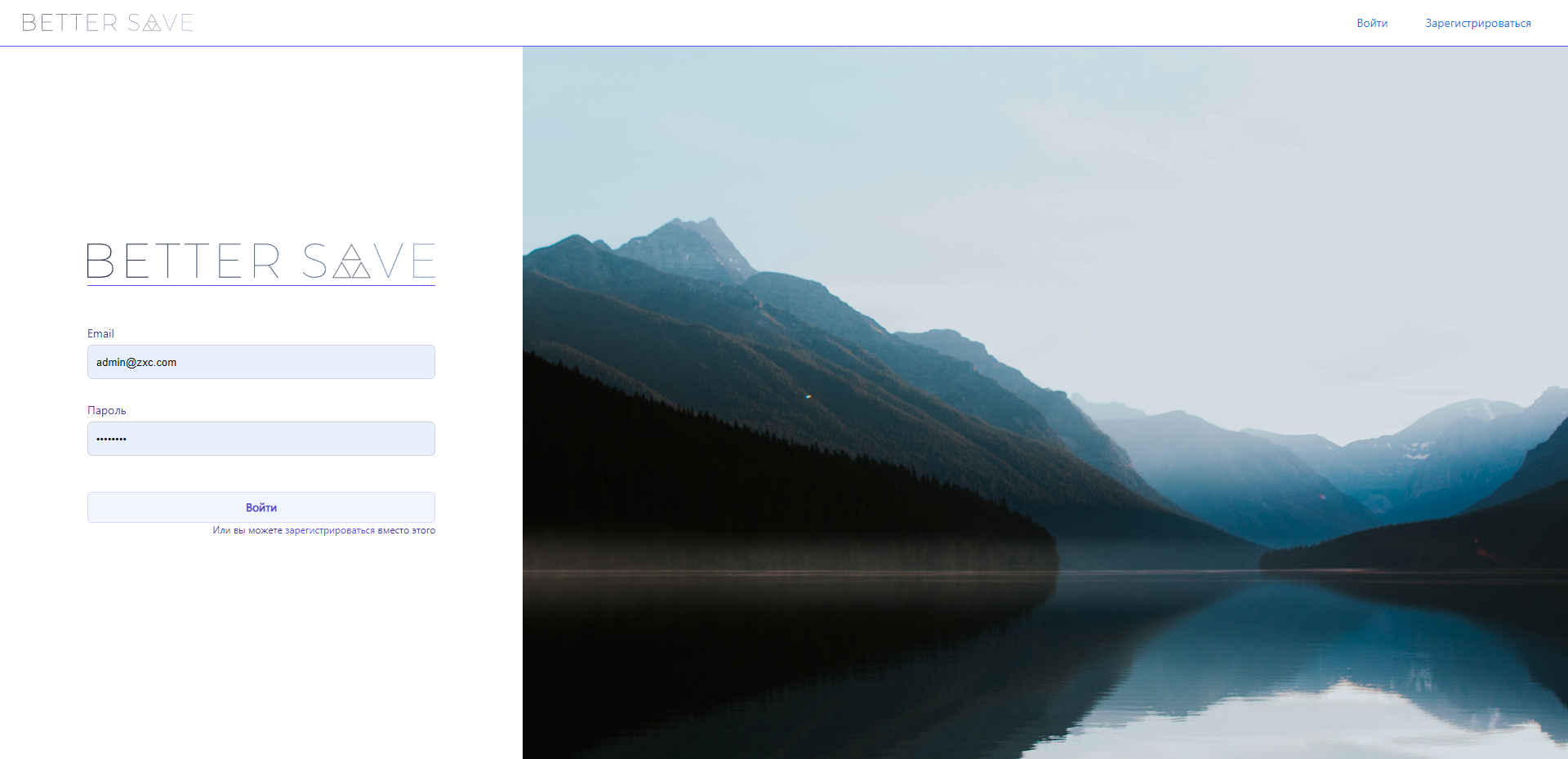


Рисунок 5.3 – Страница входа в приложение

После входа в приложение пользователь видит часть страницы, изображенной на рисунке 5.1, но без верней части, поскольку, когда пользователь авторизован, он уже не может ни войти, не зарегистрироваться. Чтобы открыть меню приложения, пользователь может нажать клавишу *«s»* на клавиатуре, на значок меню внизу страницы

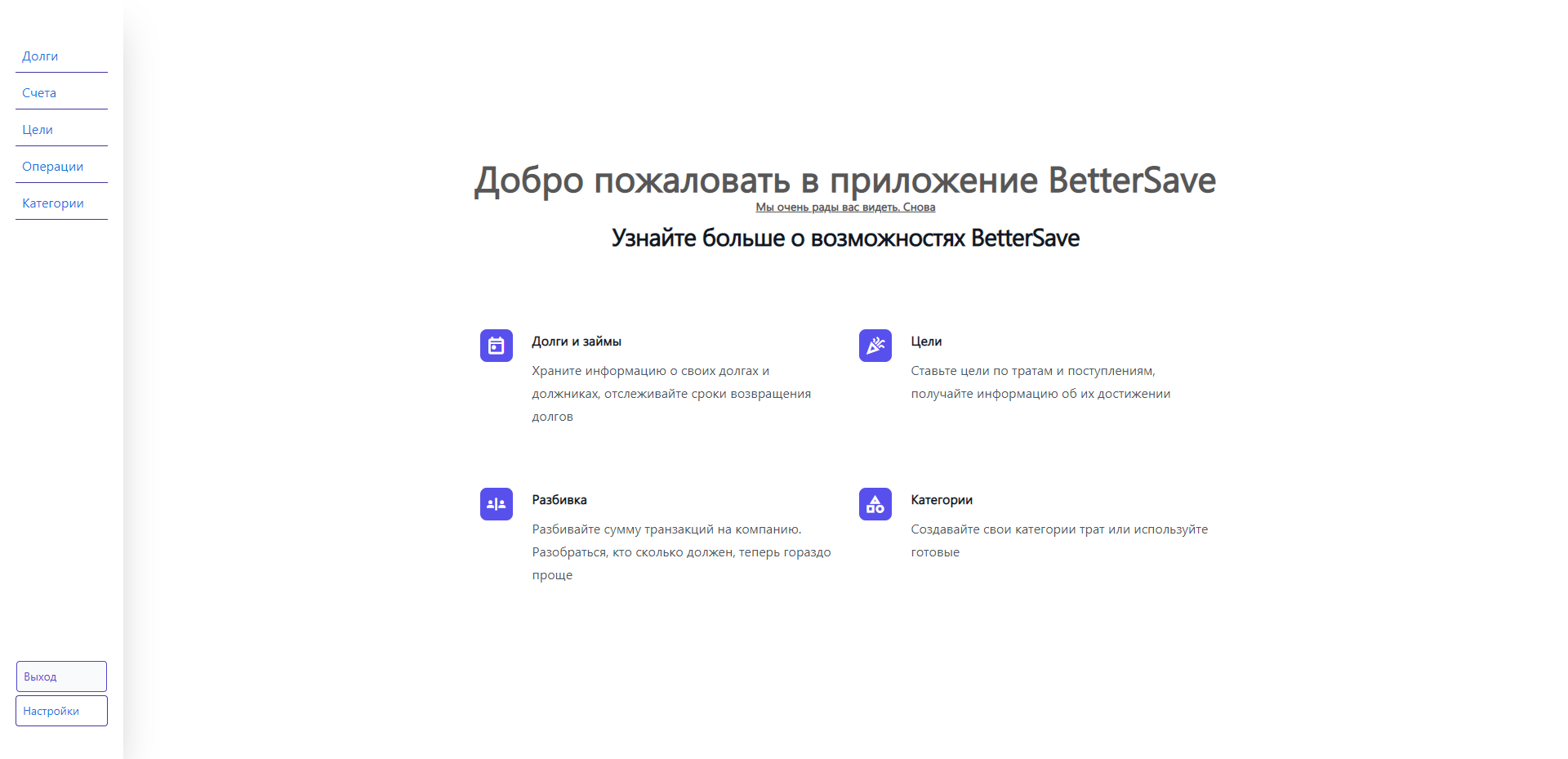


Рисунок 5.4 – Открытое меню приложения.

На мобильных устройствах это меню доступно в шапке веб-приложения. Открытое на мобильном устройстве меню изображено на рисунке 5.5.

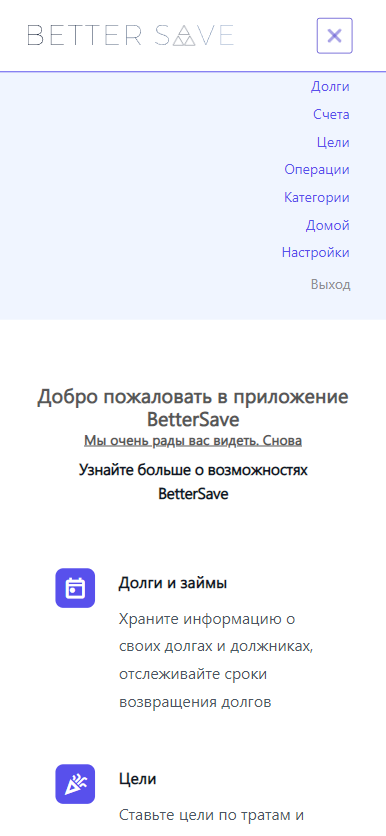


Рисунок 5.5 – Меню на мобильном устройстве.

Имея доступ к меню, пользователь может перейти по любой из ссылок приложения. Если пользователь переходит на страницу долгов, он видит страницу, изображенную на рисунке 5.6.



Рисунок 5.6 – Страница долгов и займов

На этой странице пользователь может добавить в календарь напоминание, кому он дал в долг или у кого он взял взаймы. Чтобы посмотреть этот календарь, пользователь должен нажать кнопку «Просмотреть все долги (займы)». Пример календаря долгов и займов изображен на рисунке 5.7.

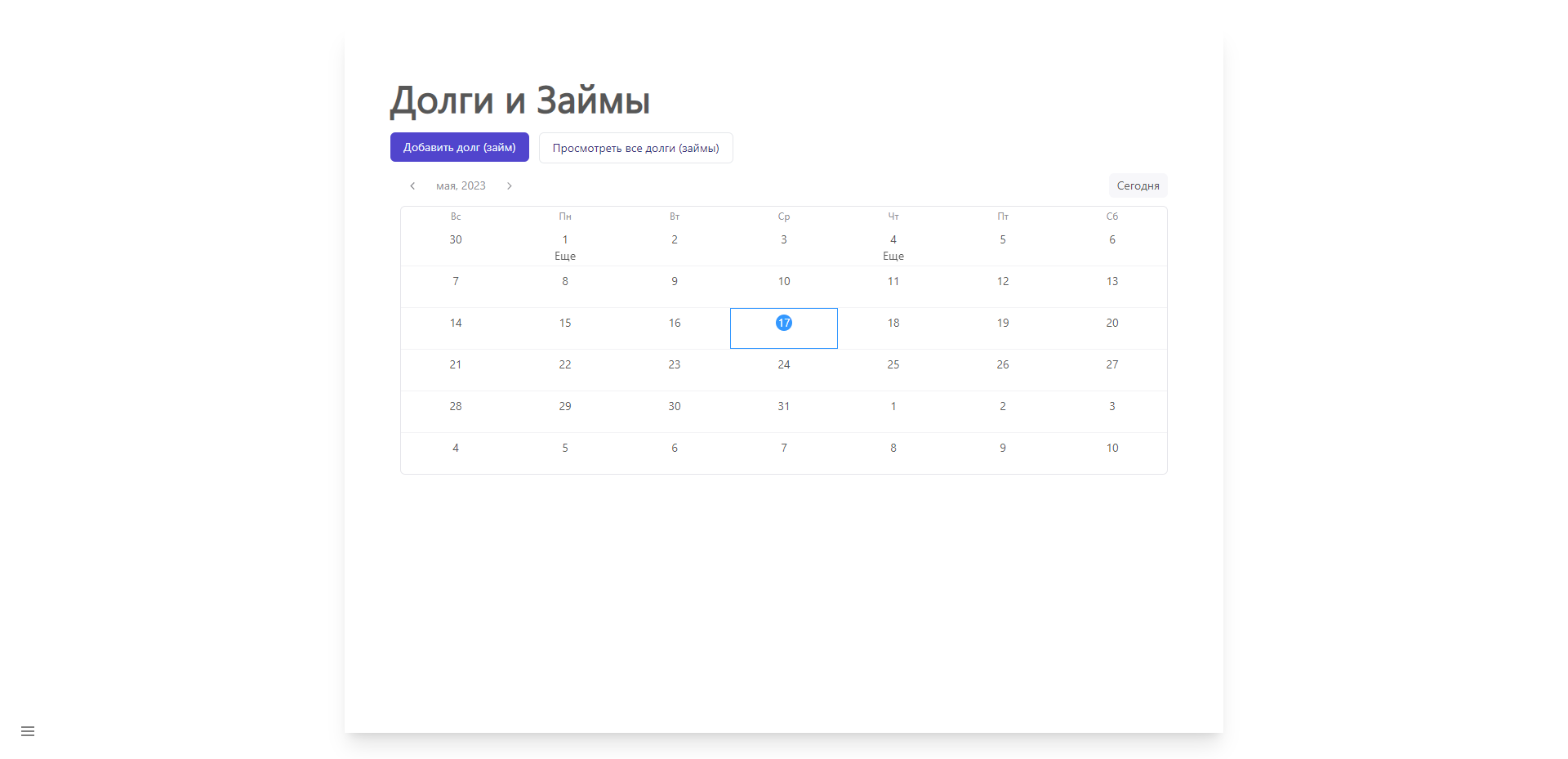


Рисунок 5.7 – Календарь долгов и займов.

Если в определенный день у пользователя существует запись, то он увидит под соответствующим числом календаря надпись «*Еще».* Если нажать на эту кнопку, то откроется окно, содержащее дополнительную информацию о записи. Пример такого окна изображен на рисунке 5.8

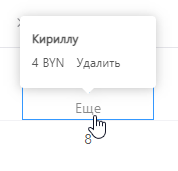


Рисунок 5.8 – Окно с дополнительной информацией о записи

Далее пользователь может перейти на страницу счетов. Где сперва получит сообщение, о том, что у него их пока нет, а также нет трат. Начальная страница счетов изображена на рисунке 5.9.

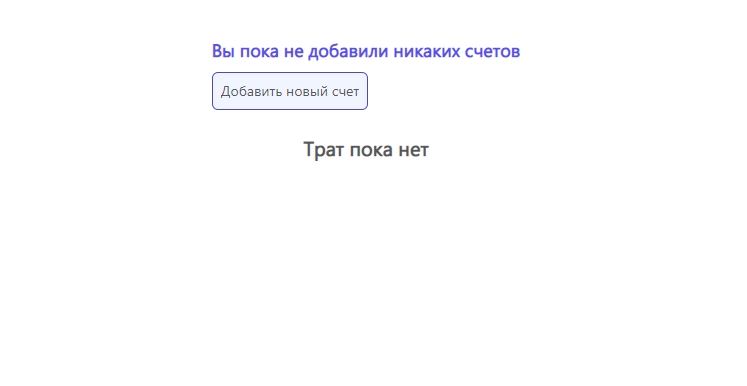


Рисунок 5.9 –Начальное состояние страницы счетов

Пользователь должен нажать кнопку «Добавить новый счет», после чего откроется окно, которое содержит настройки для добавления нового счета. Пример окна с инструментами для добавления счета изображен на рисунке 5.10.

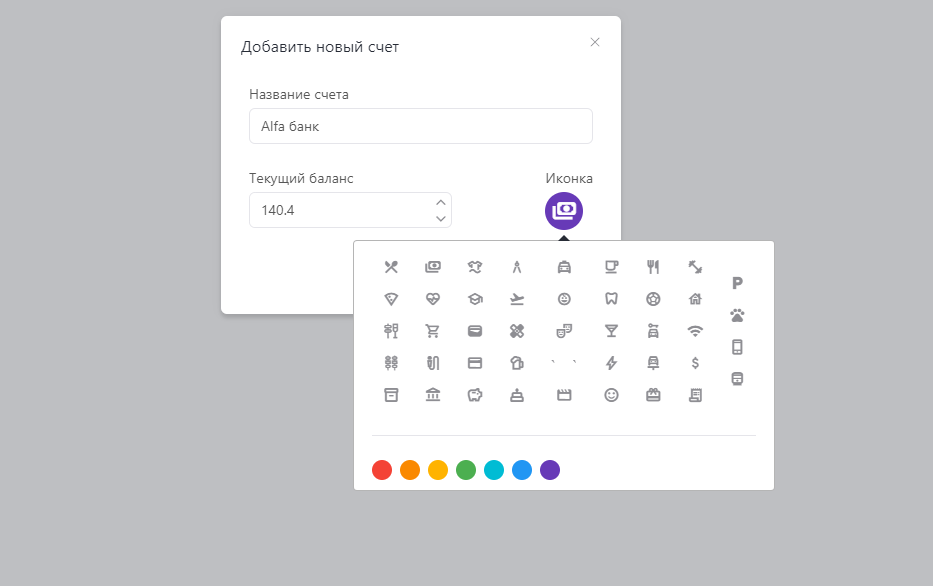


Рисунок 6.0 – Инструменты для создания нового счета

После чего пользователь может начинать следить за своими тратами и пользоваться функциями операций и целей. Чтобы добавить цели и посмотреть историю своих трат, пользователю необходимо перейти на страницу операций. Пример страницы операций изображен на рисунке 6.1. Тут пользователь может добавить операции к определенному счету, посмотреть статистику по определенным категориям, как таким, которые он создал, так и по таким, как расходы и доходы.

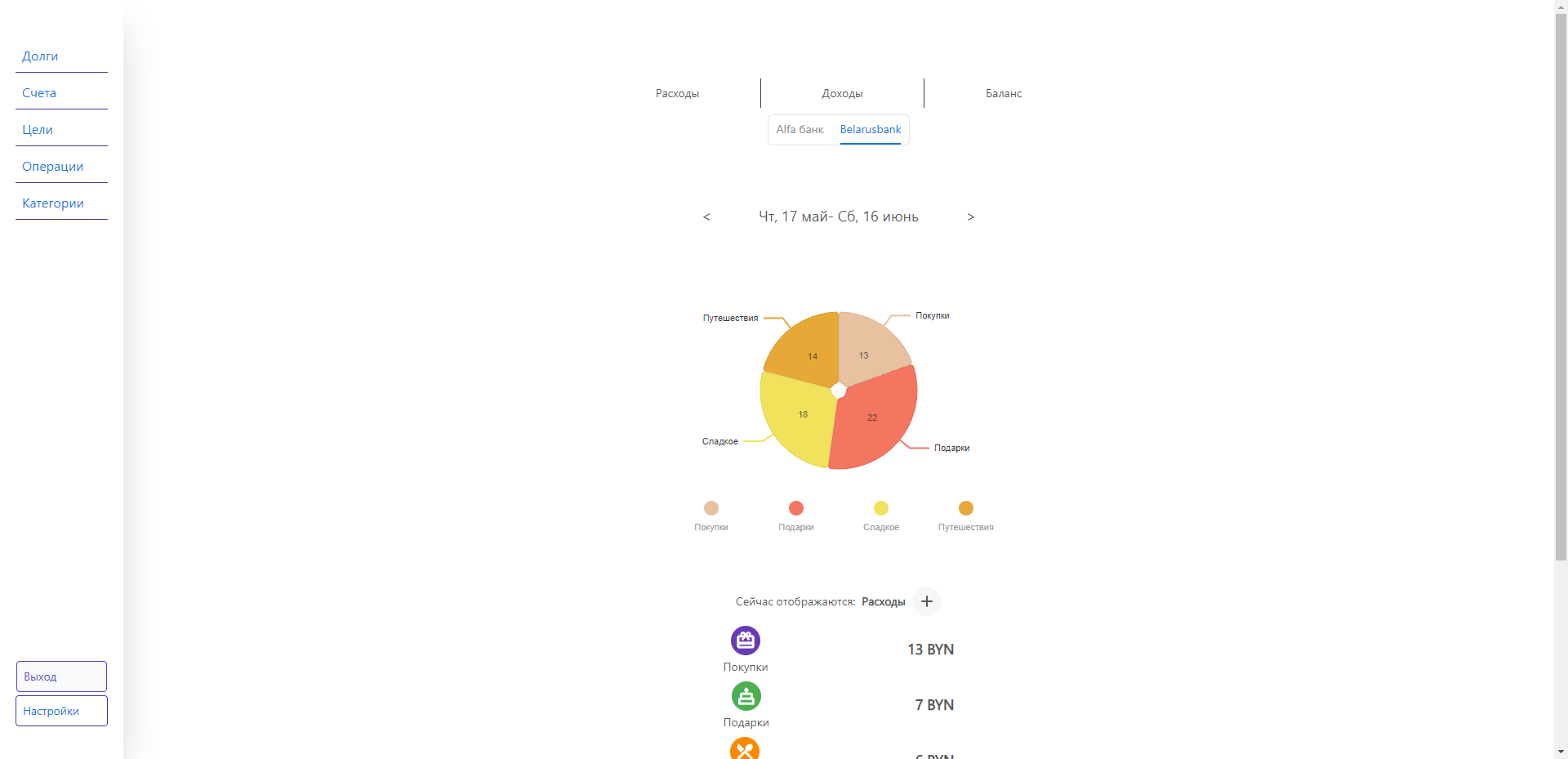


Рисунок 5.10 – Страница операций

Чтобы добавить новую операцию по счету необходимо сначала выбрать счет, затем нажать на кнопку над историей трат. После чего откроется окно для добавления новых операций. Оно изображено на рисунке 5.11

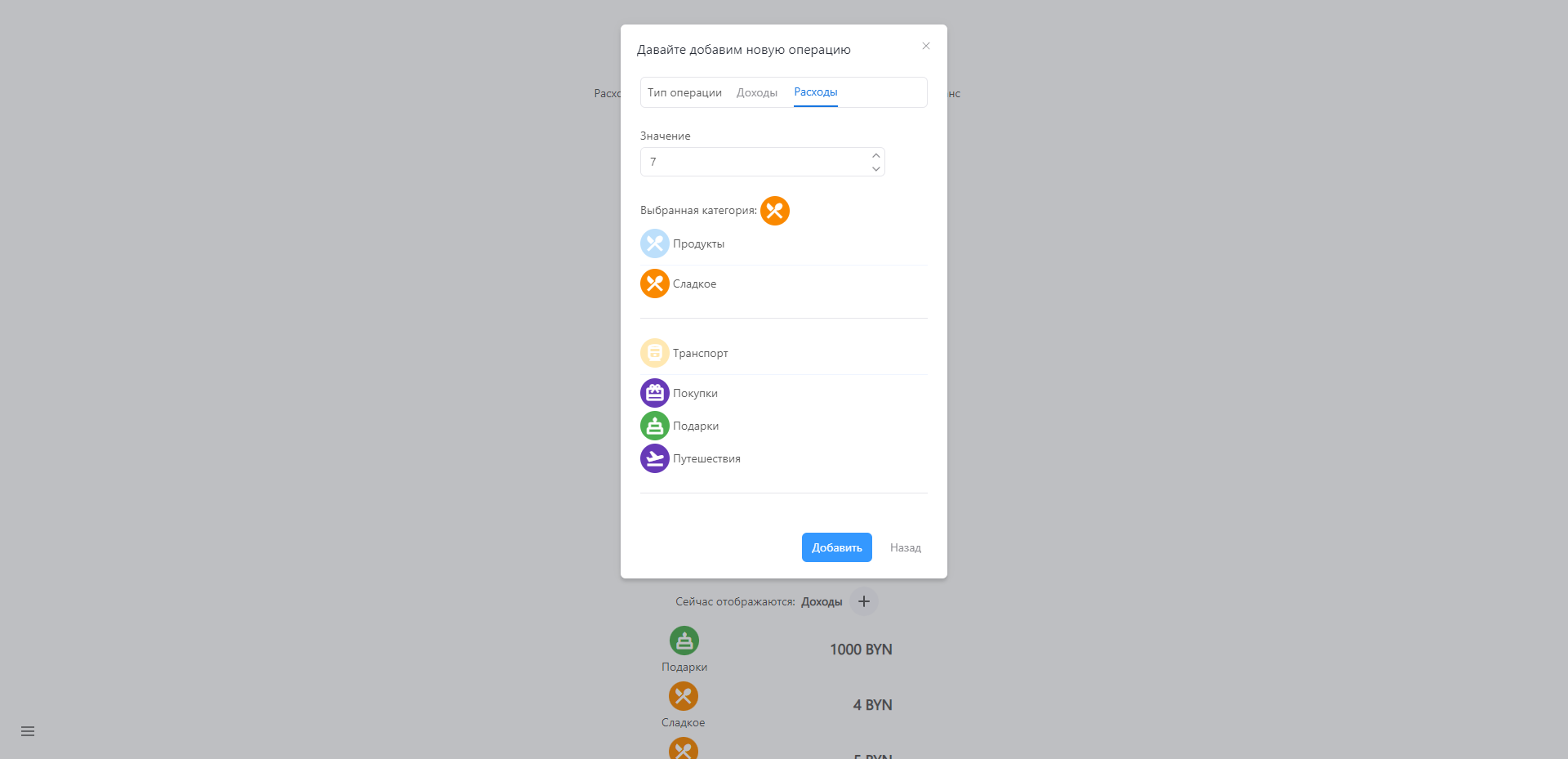


Рисунок 5.11 – Окно добавления новых операций

Категории для операций пользователь должен добавить на странице категорий, нажав на соответствующую кнопку. После добавления категории, пользователь может добавить подкатегории, которые он сможет выбрать на странице операций. Пример страницы категорий изображен на рисунке 5.12.

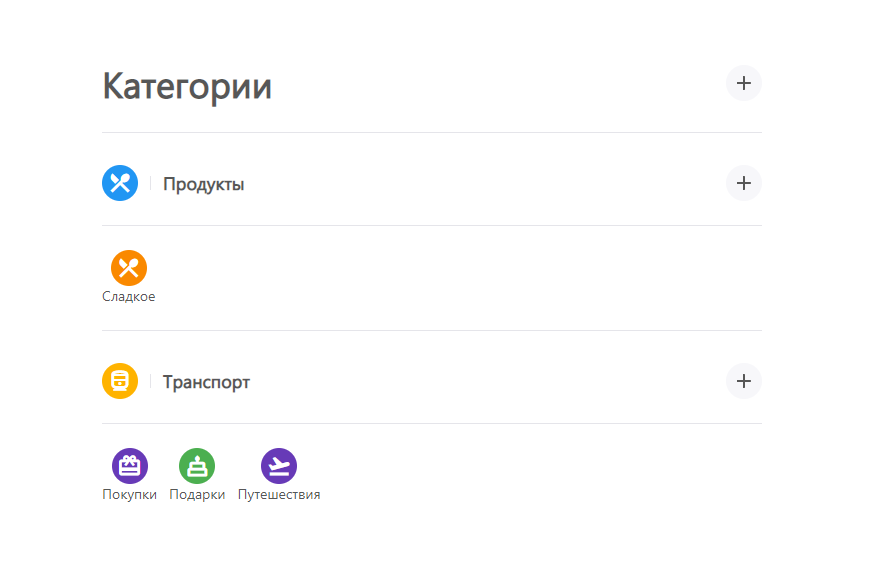


Рисунок 5.12 – Страница добавления категорий

Окно добавления категорий представляет собой поле для ввода названия, а также инструмент для выбора иконки для категории. Пример окна для выбора категорий изображен на рисунке 5.13

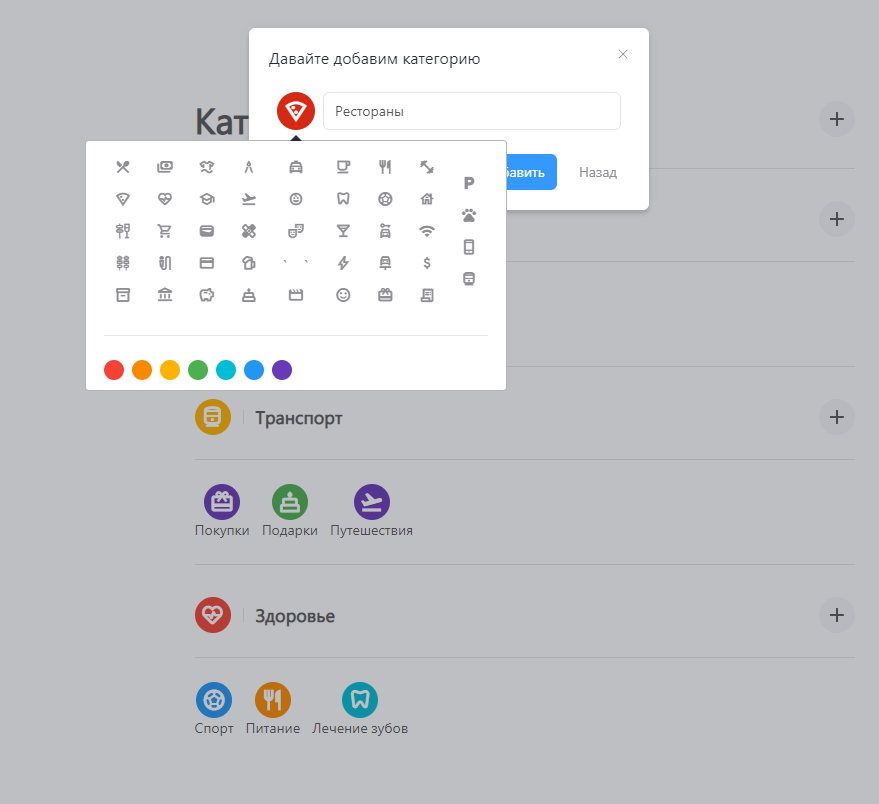


Рисунок 5.13 Окно добавления категорий

После добавления категории или подкатегории, происходит уведомление пользователя о статусе выполнения его запроса. Пример уведомления изображен на рисунке 5.14

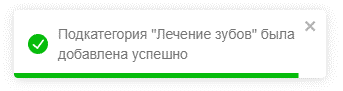


Рисунок 5.14 – Уведомление пользователя

Чтобы добавить цели, которые пользователь хочет отслеживать, он должен перейти на соответствующую страницу. На странице целей пользователь должен описать цель, которую он хочет поставить, выбрать до какого времени он хочет ее выполнять. Также, если у него уже существуют цели, он может перейти и посмотреть свой прогресс в их выполнении. Пример страницы добавления цели изображен на рисунке 5.15

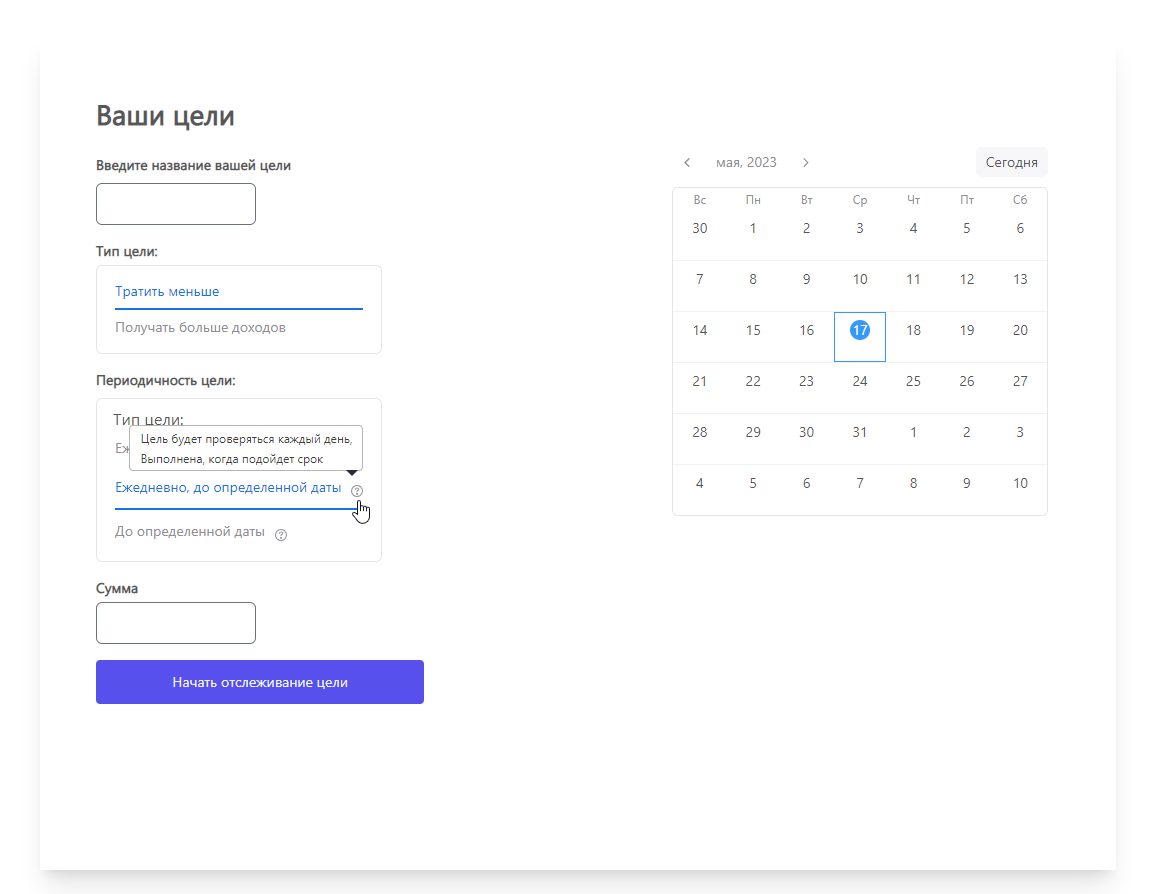


Рисунок 5.15 – Страница добавления цели

Чтобы изменить данные своего профиля, пользователю необходимо перейти на страницу настроек приложения. После чего он может изменить значения интересующих полей и сохранить результат. Страница настроек изображена на рисунке 5.16.

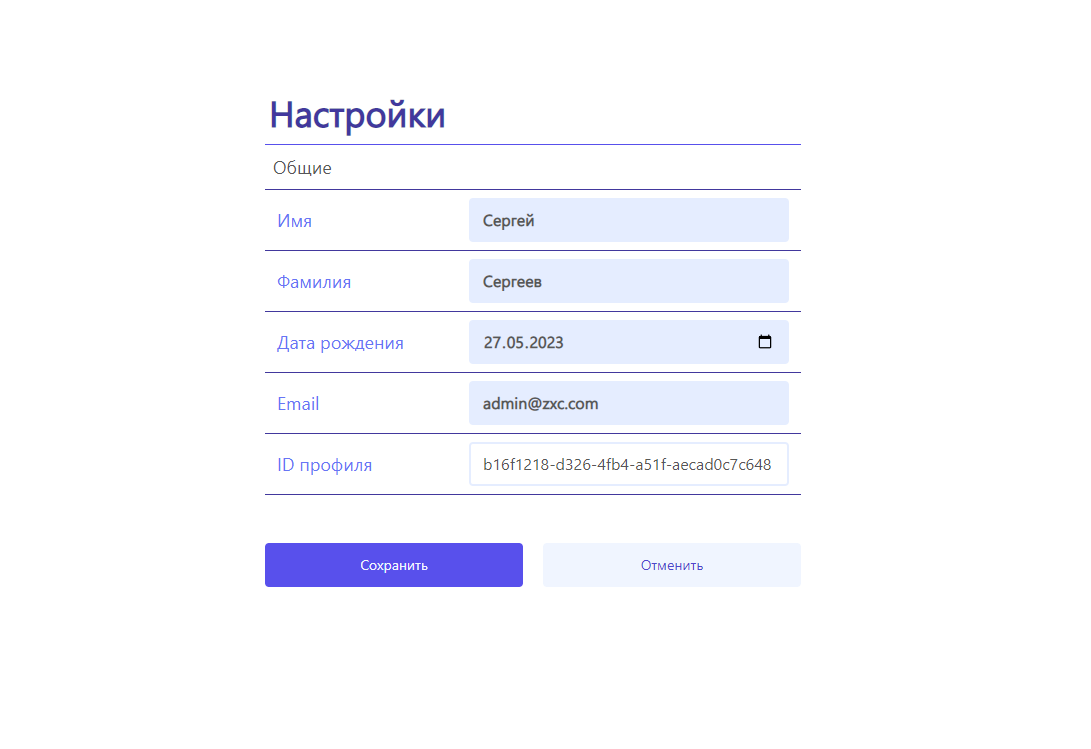


Рисунок 5.16 – Страница настроек

* 1. Выводы по разделу

В данном разделе было предоставлено руководство пользователя веб-приложением. Был описан весь основной функционал приложения, для лучшего понимая были вставлены снимки экрана.

Приложение адаптировано под экраны мобильных устройств. Также интерфейс веб-приложения довольно несложный, поэтому у пользователя не должно возникнуть затруднений при пользовании.

Пользователь веб-приложения способен проходить регистрацию и авторизацию, управлять счетами, операциями, категориями, целями и долгами, получать уведомления от системы при выполнении запросов.

Для пользователей веб-приложения предоставлен доступ ко всем функциям, которые описаны в диаграмме вариантов использования, представленной в приложении Б.

1. Технико-экономическое обоснование проекта

ФИО

Подпись

Дата

Лист

1

БГТУ 06.00.ПЗ

Разраб.

*Сергеев С. А.*

Пров.

Пахолко А. С.

.

Консульт.

Семенова Л. С.

Н. контр.

Грудо С. К.

.

Утв.

Пацей Н. В.

*6 Технико-экономическое*

*обоснование проекта*

Лит.

Листов

7

*74417005, 2023*

У

* 1. Общая характеристика разрабатываемого программного средства

Целью работы было написание программного средства, которое могло бы помочь пользователю начать делать первые правильные шаги в сторону планирования своего бюджета. В программном средстве делается упор на следующие аспекты планирования своих трат: отслеживание долговых обязательств пользователя, хранение информации о тратах и их категориях, постановка целей к определенному периоду или ежедневно. Календарь цели поможет пользователю быстро понять, как он распоряжается своими финансами, насколько близок он к достижению своих целей и какие дни могут потребовать большего внимания в будущем.

Программное средство предназначено для бесплатного использования. Стратегия монетизации предполагает размещение баннерной рекламы в приложении.

Разработка программного средства обусловлена необходимостью оказания помощи людям, которые хотят грамотно планировать использование своих финансов.

* 1. Исходные данные для проведения расчётов и маркетинговый анализ

Источниками исходных данных для данных расчетов выступают действующие нормативные правовые акты. Исходные данные для расчета приведены в таблице 6.1.

Таблица 6.1 – Исходные данные для расчета

|  |  |  |
| --- | --- | --- |
| Наименование показателя | Условные обозначения | Норматив |
| Численность разработчиков, чел. | Чр | 1 |
| Норматив дополнительной заработной платы, % | Ндз | 15 |
| Ставка отчислений в Фонд социальной защиты населения, % | Нфсзн | 34 |
| Ставка отчислений по обязательному страхованию в БРУСП «Белгосстрах», % | Нбгс | 0,6 |
| Норматив накладных расходов, % | Нобп, обх | 40 |
| Норматив расходов на реализацию, % | Нрр | 10 |
| Ставка дисконтирования, % | Сд | 16 |
| Ставка НДС, % | ННДС | 0 |
| Налог на прибыль, % | Нп | 0 |

* 1. Обоснование цены программного средства

Широкое применение вычислительных технологий требует постоянного обновления и совершенствования ПС. Выбор эффективных проектов ПС связан с их

Изм.

Лист

экономической оценкой и расчетом экономического эффекта, который может определяться как для разработчика, так и для пользователя.

У разработчика экономический эффект выступает в виде чистой прибыли от реализации ПС, остающейся в распоряжении организации.

Стоимостная оценка программных средств у разработчиков предполагает определение затрат, что включает следующие статьи:

* заработная плата исполнителей – основная и дополнительная;
* отчисления в фонд социальной защиты населения;
* отчисления по обязательному страхованию от несчастных случаев на производстве и профессиональных заболеваний;
* прочие прямые затраты;
* накладные расходы.

На основании затрат рассчитывается себестоимость и отпускная цена конечного программного средства.

* + 1. Расчёт затрат рабочего времени на разработку программного средства

В таблице 6.2 в укрупнённом виде указаны все работы, реально выполненные для создания, указанного в дипломной работе программного средства и количество рабочих дней, реально потраченных для выполнения этих работ.

Таблица 6.2 – Затраты рабочего времени на разработку ПС

|  |  |
| --- | --- |
| Содержание работ | Затраты рабочего времени, часов |
| Проектирование архитектуры приложения | 18 |
| Проектирование базы данных | 8 |
| Реализация архитектуры приложения | 68 |
| Реализация авторизации | 16 |
| Реализация фоновых работ | 16 |
| Отладка серверной части приложения | 18 |
| Написание клиентской части приложения | 120 |
| Написание методов взаимодействия c серверной частью | 56 |
| Отладка клиентской части приложения | 24 |
| Добавление уведомлений пользователя | 12 |
| Тестирование | 7 |
| Документирование | 7 |
| Всего | 370 |

В результате получилось 370 часов, что будет использовано далее для расчётов. Исходя из этого, и учитывая, что в среднем месяце, количество рабочих часов равняется 168, полная реализация программного средства заняла 2,20 месяца.

* + 1. Расчет основной заработной платы

Для определения величины основной заработной платы, было проведено исследование величин заработных плат разработчиков на .*NET* и *React*. В итоге было установлено, что заработная плата специалиста уровня *middle* составляет около 2 553,60 рублей, что эквивалентно 15,20 руб./час.

Согласно таблице 6.2, проект разрабатывался одним специалистом на протяжении 370 часов. Таким образом, основная заработная плата будет рассчитываться по формуле 6.1

|  |  |  |
| --- | --- | --- |
|  | , | (6.1) |

где Соз– основная заработная плата, руб.;

Траз – время раработки (часов);

Сзп – средняя часовая ставка руб./час;

Краз – количество разработчиков, человек.

|  |  |  |
| --- | --- | --- |
|  | руб*.* |  |

В дальнейшем для других расчётов используется основная заработная плата, рассчитанная по указанной выше методике.

* + 1. Расчет дополнительной заработной платы

Дополнительная заработная плата представляет собой выплаты, предусмотренные законодательством о труде, включает компенсирующие выплаты (например, доплаты за работу в сверхурочное время, в государственные праздники, праздничные и выходные дни) и определяется по нормативу в процентах к основной заработной плате по формуле 6.2.

|  |  |  |  |
| --- | --- | --- | --- |
|  | , | (6.2) | (6.4) |

где Соз – основная заработная плата, руб.;

Ндз – норматив дополнительной заработной платы, %.

 руб.

* + 1. Расчет отчислений в Фонд социальной защиты населения и по обязательному страхованию

Отчисления в Фонд социальной защиты населения (ФСЗН) и по обязательному страхованию от несчастных случаев на производстве и профессиональных заболеваний в БРУСП «Белгосстрах» определяются в соответствии с действующими законодательными актами по нормативу в процентном отношении к фонду основной и дополнительной зарплаты исполнителей.

Отчисления в Фонд социальной защиты населения вычисляются по формуле 6.3.

|  |  |  |
| --- | --- | --- |
|  |  | (6.3) |

где – основная заработная плата, руб.;

– дополнительная заработная плата на конкретное ПС, руб.;

– норматив отчислений в Фонд социальной защиты населения, %.

Отчисления в БРУСП «Белгосстрах» вычисляются по формуле 6.4.

|  |  |  |
| --- | --- | --- |
|  |  | (6.4) |

Таким образом, общие отчисления в БРУСП «Белгосстрах» составили 38,81 руб., а в фонд социальной защиты населения – 2 198,98 руб. Суммарные отчисления составили 2 237,79 руб.

* + 1. Расчет суммы прочих прямых затрат

Расходы на конкретное программное средство Спз включают затраты на стоимость приобретения лицензий на средства разработки.

Средства разработки использовались следующие: *WebStorm*, *Rider*, *DataGrip*. Данные программные средства поставляются компанией *JetBrains* и продаются по системе ежемесячной подписки. Согласно официальному сайту [30], подписка на данные утилиты в месяц в сумме обходится в 80,70 долларов, что эквивалентно 236,45 руб. на момент 7 мая 2023 г. Как было выяснено ранее, разработка длится 2,2 месяца, поскольку тарифы на неполный месяц не рассчитаны, то для завершения процесса разработки, длительность подписки должна составлять 3 месяца. Таким образом, сумма прочих прямых затрат рассчитывается как произведение затрат в месяц Сзт/м на количество месяцев разработки N и высчитывается по формуле

Сумма прочих прямых затрат составит

= 236,45 ⋅ 3 = 709,35 руб.

* + 1. Расчет суммы накладных расходов

Сумма накладных расходов Собп,обх – произведение основной заработной платы исполнителей на конкретное программное средство Соз на норматив накладных расходов в целом по организации Нобп,обх, по формуле 6.5.

|  |  |  |
| --- | --- | --- |
|  |  | (6.5) |

Сумма накладных расходов составит:

= 5 624 ⋅ 40 / 100 = 2 249,60 руб*.*

* + 1. Сумма расходов на разработку программного средства

Сумма расходов на разработку программного средства Ср определяется как сумма основной и дополнительной заработных плат исполнителей на конкретное программное средство, отчислений на социальные нужды, суммы прочих затрат и суммы накладных расходов, по формуле 6.6.

|  |  |
| --- | --- |
| . | (6.6) |

Все данные необходимые для вычисления есть, поэтому можно определить сумму расходов на разработку программного средства.

Ср = 5 624 + 843,6 + 2 198,98 + 38,81 +709,35 + 2 249,60 = 11 664,34 руб.

Сумма расходов на разработку программного средства была вычислена на основе данных, рассчитанных ранее в данном разделе, и составила 11 664,34 рублей.

* + 1. Расходы на реализацию

Сумма расходов на реализацию программного средства Срр определяется как произведение суммы расходов на разработку на норматив расходов на реализацию Нрр, и находится по формуле 6.7.

|  |  |  |
| --- | --- | --- |
|  |  | (6.7) |

= 11 664,34 ⋅ 10 / 100 = 1 166,43 руб.

Все проведенные выше расчеты необходимы для вычисления полной себестоимости проекта.

* + 1. Расчет полной себестоимости

Полная себестоимость Сп определяется как сумма двух элементов: суммы расходов на разработку Ср и суммы расходов на реализацию программного средства Срр.

Полная себестоимость Сп вычисляется по формуле 6.8

|  |  |  |
| --- | --- | --- |
|  |  | (6.8) |

Сп = 11 664,34 + 1 166,43= 12 830,77 руб.

Полная себестоимость программного средства была вычислена на основе данных, рассчитанных ранее в данном разделе.

* + 1. Определение цены, оценка эффективности

Монетизация программного средства будет осуществляться через показ рекламных баннеров различных банков, ориентированных на привлечение новых клиентов.

Предполагается, что количество рекламных баннеров на странице будет равно двум. Размещение рекламы для одного спонсора будет стоить 280,00 рублей в месяц.

Таким образом, доход от размещения рекламы в год

= 280 2 12 = 6 720,00 руб.

Срок окупаемости приложения Tок вычисляется по формуле 6.9.

|  |  |  |
| --- | --- | --- |
|  |  | (6.9) |

где – полная себестоимость, руб.;

Пост.в год – денежные поступления от показа рекламы в приложении за год, руб.

Ток = 12 830,77 / 6 720 = 1,91 года.

Срок окупаемости веб-приложения составит 1,91 года. Чистый дисконтированный доход рассчитывается по формуле 6.10.

|  |  |
| --- | --- |
|  | (6.10) |
| где СД– ставка дисконтирования, %,  И – инвестиционные затраты, руб,  СFn– дисконтированный доход, руб; |  |

Дисконтированный доход за первые три года составит:

= 6 720,00 / 1,16 = 5 793,10 руб.

= 6 720,00 / 1,162= 4 994,05 руб.

= 6 720,00 / 1,163 = 4 305,22 руб.

Расчёты будут необходимы для расчётов дисконтируемого срока окупаемости. Результаты расчётов, представленных выше, описаны в таблице 6.3.

Таблица 6.3 – Результаты расчетов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Показатель, год | 0 | 1 | 2 | 3 |
| Инвестиционные затраты, руб. | 12 830,77 | − | ­− | − |

Окончание таблицы 6.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Доход, руб. | − | 6 720 | 6 720 | 6 720 |
| ДД, руб. | − | 5 793,10 | 4 994,05 | 4 305,22 |
| ДД нараст.итогом, руб. | − | 5 793,10 | 10 787,16 | 15 092,38 |

Дисконтированный срок окупаемости составит:



Таким образом, дисконтированный срок окупаемости составит 2,47 г.

* 1. Вывод по разделу

В таблице 6.4 представлены результаты расчётов для основных показателей данной главы в краткой форме.

Таблица 6.4 – Результаты расчетов

|  |  |
| --- | --- |
| Наименование показателя | Значение |
| Время разработки, ч. | 370 |
| Количество разработчиков, чел. | 1 |
| Основная заработная плата, руб. | 5 624,00 |
| Дополнительная заработная плата, руб. | 843,60 |
| Отчисления в Фонд социальной защиты населения и БРУСП «Белгосстрах», %, руб. | 2 237,79 |
| Накладные расходы, руб. | 2 249,60 |
| Себестоимость разработки программного средства, руб. | 11 664,34 |
| Расходы на реализацию, руб. | 1 166,43 |
| Полная себестоимость, руб. | 12 830,77 |
| Срок окупаемости, лет | 1,91 |
| Дисконтированный срок окупаемости, лет | 2,47 |

Разработка программного средства обусловлена необходимостью создания многофункционального приложения для помощи людям, которые хотят грамотно планировать использование своих финансов.

Разработка программного средства, осуществляемая одним программистом в течение 2,2 месяцев. Полная себестоимость составила 12 830,77 руб. Денежные поступления от размещения рекламы в приложении за год составили 6 720 рублей. Дисконтированный итоговый доход за три года составит 15 092,38 рублей. Срок окупаемости приложения составит 1,91 года. Дисконтированный срок окупаемости приложения составит 2,47 года.

Заключение

В ходе выполнения дипломного проекта были рассмотрены основные аналоги, произведен патентный поиск. Были проанализированы и выбраны основные средства для разработки дипломного проекта. На основе выбранных технологий была спроектирована и реализована архитектура веб-приложения.

ФИО

Подпись

Дата

Лист

1

БГТУ 06.00.ПЗ

Разраб.

*Сергеев С. А.*

Пров.

Пахолко А. С.

.

Н. контр.

Грудо С. К.

.

Утв.

Пацей Н. В.

*Заключение*

Лит.

Листов

1

*74417005, 2023*

У

В ходе выполнения дипломного проекта были выполнены все поставленные задачи:

Для разработки серверной части были использованы язык *С#*, платформа *ASP.NET Core*, фреймворки *EF Core*, *Hangfire*, пакеты *Mailkit*, Swagger. В качестве инструментов разработки клиентской части были использованы библиотеки *ReactJS*, *Redux*, *Redux-toolkit*, *Webpack*, *Tailwind*.

Для хранения информации была спроектирована и реализована *MS* *SQL* база данных. Для работы с ней была выбрана *ORM*-библиотека *EF* *Core*. База данных содержит 9 таблиц, хранящих все информацию о пользователях, долгах, целях, операциях.

Для удобства пользования системой был разработан удобный и красивый пользовательский интерфейс клиентской части приложения. К нему было написано руководство пользователя, позволяющее подробно рассмотреть варианты использования приложения.

Для проверки правильности работы программного средства было проведено мануальное и *unit*-тестирование. Процент покрытия тестами уровня бизнес-логики равен 85%.

Было выполнено технико-экономическое обоснование проекта, рассчитана стоимость приложения, срок его окупаемости.

Таким образом, при создании программного средства были использованы все указанные технологии, выполнены все функциональные требования.

Список использованных источников

1. Веб-приложение *Drebedengi* [Электронный ресурс] / <https://www.drebedengi.ru/>
2. Веб-приложение *EasyFinance* [Электронный ресурс] <https://vk.com/easyfinance_fans>
3. Приложение *Buddy* <https://buddy.download/> [Электронный ресурс]
4. Веб-приложение Дзен-мани <https://zenmoney.ru/> [Электронный ресурс]
5. <https://patentimages.storage.googleapis.com/ed/56/0d/c744d08d6fe73e/US8301530.pdf> [Электронный ресурс]
6. <https://patentimages.storage.googleapis.com/40/53/e5/8a063dc57739e4/US8090656.pdf> [Электронный ресурс]
7. <https://patentimages.storage.googleapis.com/3e/44/7c/cc7c28c96ca8eb/US11100513.pdf> [Электронный ресурс]
8. Библиотека *ReactJS* <https://react.dev/> [Электронный ресурс]
9. Язык программирования *TypeScript* <https://www.typescriptlang.org/> [Электронный ресурс]
10. Язык программирования *JavaScript* <https://www.javascript.com/> [Электронный ресурс]
11. Статья: виртуальный *DOM*<https://legacy.reactjs.org/docs/faq-internals.html> [Электронный ресурс]
12. Платформа *ASP.NET Core*<https://learn.microsoft.com/ru-ru/aspnet/core/?view=aspnetcore-7.0> [Электронный ресурс]
13. База данных: *MS SQL*<https://www.microsoft.com/ru-ru/sql-server> [Электронный ресурс]
14. Фреймворк *EF Core* <https://learn.microsoft.com/ru-ru/ef/core/> [Электронный ресурс]
15. Веб-сервер *IIS*: <https://www.iis.net/overview> [Электронный ресурс]
16. Библиотека *Redux* <https://redux.js.org/> [Электронный ресурс]
17. Библиотека *Redux-toolkit* <https://redux-toolkit.js.org/> [Электронный ресурс]
18. Библиотека *Webpack* <https://webpack.js.org/> [Электронный ресурс]
19. Фреймворк *Hangfire* <https://www.hangfire.io/> [Электронный ресурс]
20. Библиотека *Automapper* <https://automapper.org/> [Электронный ресурс]
21. Статья: *Fluent API*<https://www.entityframeworktutorial.net/efcore/fluent-api-in-entity-framework-core.aspx> [Электронный ресурс]
22. Веб-сервис *Swagger* <https://swagger.io/> [Электронный ресурс]
23. Библиотека *Husky* <https://github.com/typicode/husky> [Электронный ресурс]
24. Библиотека стилей *Tailwind* <https://tailwindcss.com> [Электронный ресурс]
25. Библиотека *React-toastify* <https://fkhadra.github.io/react-toastify/introduction/> [Электронный ресурс]
26. Библиотека *xUnit* <https://xunit.net/> [Электронный ресурс]
27. Библиотека *FakeItEasy* <https://fakeiteasy.github.io/> [Электронный ресурс]
28. Библиотека *AutoFixture* <https://github.com/AutoFixture/AutoFixture> [Электронный ресурс]
29. *AAA* <https://www.manning.com/books/unit-testing?utm_source=freecontentcenter&utm_medium=website&utm_campaign=book_unittesing&utm_content=article_03> [Электронный ресурс]
30. Официальный сайт компании *JetBrains* <https://www.jetbrains.com/store/#personal?billing=monthly> [Электронный ресурс]

Приложение А  
Диаграмма вариантов использования

Приложение Б  
Диаграмма архитектуры приложения

Приложение В  
Блок-схема алгортима проверки цели

Приложение Г  
Логическая схема базы данных

Приложение Д  
Диаграмма развертывания

Приложение Ж  
Диаграмма классов приложения