МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Кафедра «Прикладная математика и информатика»
(наименование)

02.03.03 «Математическое обеспечение и администрирование информационных систем»
(код и наименование направления подготовки / специальности)

Мобильные и сетевые технологии
(направленность (профиль) / специализация)

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (БАКАЛАВРСКАЯ РАБОТА)

на тему «Разработка программного обеспечения для управления финансами»

Обучающийся Д.С. Барсов
(Инициалы Фамилия) (личная подпись)

Руководитель М.А. Тренина
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Консультант к.п.н., доцент С.А. Гудкова
(ученая степень (при наличии), ученое звание (при наличии), Инициалы Фамилия)

Тольятти 2025

Аннотация

Тема выпускной квалификационной работы — «Разработка программного обеспечения для управления финансами».

Ключевые слова: управление финансами, веб-сервис, транзакции, анализ расходов.

В условиях цифровизации и роста экономической нестабильности эффективное управление личными финансами становится важнейшим элементом повседневной жизни. Пользователям необходимо надёжное средство для учета доходов, расходов и анализа финансового поведения. Одним из современных подходов к решению данной задачи является применение специализированного программного обеспечения.

В рамках настоящей работы был разработан серверный компонент программного обеспечения — веб-сервис, реализующий функциональность учёта, фильтрации и аналитики персональных транзакций. Проект реализован с использованием технологий Java 17, Spring Boot и PostgreSQL, с соблюдением архитектурных принципов многослойного проектирования.

Объектом исследования выступает система управления личными финансами. Предметом исследования является разработка backend-сервиса, предоставляющего интерфейс взаимодействия по протоколу HTTP (REST API) для клиента.

Цель работы заключается в создании веб-сервиса, обеспечивающего автоматизированный учет и анализ персональных финансовых данных. Практическая значимость проекта состоит в возможности использования разработанного решения как основы для интеграции с мобильными или веб-интерфейсами, а также для последующего расширения и внедрения в реальные пользовательские сценарии.

Бакалаврская работа включает 59 страниц, содержит 24 рисунка, 13 таблиц и 24 источника.

Abstract

Title of the final qualification work — “Development of software for personal finance management”.

Keywords: financial management, web service, financial transactions, expense analysis.

In the context of digital transformation and economic uncertainty, effective personal finance management has become an essential part of everyday life. Users need a reliable tool for tracking income and expenses, analyzing financial behavior, and making informed decisions.

This paper presents the development of a backend web service that provides functionality for recording, filtering, and analyzing personal financial transactions. The system is implemented using Java 17, Spring Boot, and PostgreSQL, following the principles of layered architecture and object-oriented design.

The object of the research is a personal finance management system. The subject is a RESTful web service designed for automating accounting and financial data analytics.

The goal of the work is to develop a backend application that allows secure registration, transaction processing, and financial analytics. The practical relevance of the project lies in its potential integration with external clients such as mobile or web interfaces and its extensibility for future enhancements.

The bachelor’s thesis consists of 59 pages, includes 24 figures, 13 tables, and 24 references.

Оглавление

Введение

Эффективное управление личными финансами является неотъемлемой частью современной цифровой жизни. В условиях инфляции, роста цен и широкой доступности электронных платёжных систем пользователи всё чаще сталкиваются с необходимостью систематического учёта доходов и расходов, анализа финансового поведения и принятия обоснованных решений.

Современные программные средства учета финансов, представленные в виде мобильных приложений или банковских сервисов, зачастую либо перегружены функциями, либо ограничены жёсткими рамками одной платформы или организации. Это создаёт потребность в универсальных, лёгких и адаптивных решениях, которые можно интегрировать в разные пользовательские сценарии [6].

Данной работе посвящена разработка программного обеспечения в виде веб-сервиса для управления личными финансами. Разработанное серверное приложение реализует полный цикл работы с транзакциями: от регистрации пользователя и авторизации до создания, анализа и фильтрации операций. Проект выполнен с использованием Java 17, Spring Boot и PostgreSQL, что обеспечивает надёжность, масштабируемость и гибкость системы.

Объектом исследования является процесс управления личными финансами с использованием информационных технологий. Предметом — реализация серверной части веб-сервиса, обеспечивающего хранение, обработку и анализ транзакционных данных.

Цель выпускной квалификационной работы — создание backend-приложения для учёта и анализа персональных финансов, функционирующего на основе REST API.

Для достижения цели были сформулированы следующие задачи:

– провести анализ существующих решений и обосновать актуальность темы;

– определить архитектуру системы, спроектировать базу данных и REST-интерфейсы;
– реализовать основные функции: регистрацию, авторизацию, операции с транзакциями и аналитику;
– протестировать реализованный сервис и провести демонстрацию работы API.

Методы, использованные в работе, включают: объектно-ориентированное проектирование, построение REST-архитектуры, проектирование реляционной базы данных, модульное и интеграционное тестирование.

Практическая значимость проекта заключается в создании надёжного серверного решения, которое может использоваться как отдельный backend-сервис либо как часть более сложной информационной системы с веб- или мобильным клиентом.

Работа состоит из введения, трёх глав, заключения и списка литературы. В первой главе рассматриваются теоретические основы и анализ технологий. Во второй — проектирование архитектуры, логики и модели данных. Третья глава посвящена реализации, тестированию и демонстрации работы сервиса. В заключении подводятся итоги и предлагаются направления развития.

Глава 1 Теоретические основы разработки сервиса для управления личными финансами

1.1 Проблемы управления финансами в традиционных системах

В современном мире эффективное управление личными финансами становится не просто полезным навыком, а необходимостью. Однако существующие подходы к решению этой задачи сталкиваются с рядом системных ограничений, снижающих их практическую ценность для пользователей.

Традиционно для учета финансов используются три основных подхода. Наиболее простым и доступным остается ручной метод с использованием блокнотов или электронных таблиц. Хотя он не требует специальных знаний и доступен каждому, такой способ отличается высокой трудоемкостью, подвержен ошибкам и не предоставляет возможностей для глубокого анализа данных.

Более продвинутым решением являются специализированные приложения для учета финансов, такие как Monefy или ZenMoney [1]. Эти программы предлагают удобный интерфейс и базовые функции автоматизации, но часто страдают от ограниченной гибкости в настройке категорий и правил анализа. Многие из них используют агрессивную монетизацию, предлагая ключевые функции только в платных версиях.

Банковские онлайн-сервисы, казалось бы, должны решить проблему учета финансов, но на практике их возможности оказываются ограниченными [12]. Они предоставляют информацию только по операциям конкретного банка, не учитывая другие финансовые потоки пользователя, а их аналитические функции зачастую сводятся к простейшим диаграммам расходов.

Главной проблемой современных решений является их неспособность удовлетворить индивидуальные потребности пользователей. Большинство

сервисов предлагает жестко заданные категории расходов и доходов, не позволяя адаптировать систему под специфические нужды конкретного человека. Аналитические возможности часто ограничиваются базовой статистикой, без возможности создания пользовательских отчетов или прогнозирования будущих трат.

Вопросы безопасности данных также остаются слабым местом многих финансовых приложений. Устаревшие системы аутентификации, отсутствие двухфакторной защиты и прозрачности в вопросах хранения персональной информации вызывают обоснованные опасения у пользователей.

Масштабируемость и интеграционные возможности большинства решений оставляют желать лучшего. Закрытые архитектуры не позволяют подключать дополнительные модули или интегрировать сервис с другими финансовыми инструментами, что значительно снижает их полезность в долгосрочной перспективе.

Все эти факторы свидетельствуют о необходимости создания нового поколения финансовых сервисов, которые сочетали бы в себе гибкость, безопасность и мощные аналитические возможности. Современные технологии, такие как облачные вычисления, открытые API и методы машинного обучения, открывают новые перспективы для разработки действительно эффективных инструментов управления личными финансами.

Ключевыми направлениями развития в этой области должны стать: создание адаптивных систем категоризации, внедрение интеллектуального анализа расходов, обеспечение максимальной безопасности данных и разработка открытых архитектур, позволяющих легко расширять функциональность сервиса. Именно эти принципы легли в основу разрабатываемого веб-сервиса для управления личными финансами.

1.2 Сравнительный анализ технологий для разработки финансовых сервисов

Выбор технологического стека для реализации программного продукта в значительной степени определяет надёжность, безопасность, масштабируемость и производительность конечного решения. Для финансовых сервисов, работающих с чувствительными пользовательскими данными, особенно критично учитывать данные параметры. Поэтому при проектировании архитектуры веб-сервиса для управления личными финансами была проведена оценка доступных технологий на основе ключевых критериев: производительность, безопасность, поддержка транзакционности и зрелость сообщества.

Для разработки требований к сервису используем методологию FURPS+(Functionality, Usability, Reliability, Performance, Supportability). «FURPS+ — это расширенная модель классификации атрибутов качества программного обеспечения, которая помогает анализировать и специфицировать функциональные и нефункциональные требования к системе» [5].

На рисунке 1 представлена блок-схема алгоритма выбора технологического стека, в основе которой лежит пошаговая оценка альтернативных решений. Алгоритм начинается с определения функциональных и нефункциональных требований, таких как скорость обработки запросов, устойчивость к сбоям, защита персональных данных и возможность расширения. После чего выполняется сравнительный анализ возможных платформ и фреймворков: Java (с использованием Spring Boot), Python (на базе Django) и Node.js (в связке с Express) [7]. Каждая из платформ оценивается по ранее обозначенным критериям. В результате, на основании комплексной оценки, принимается решение в пользу стека Java + Spring Boot + PostgreSQL.

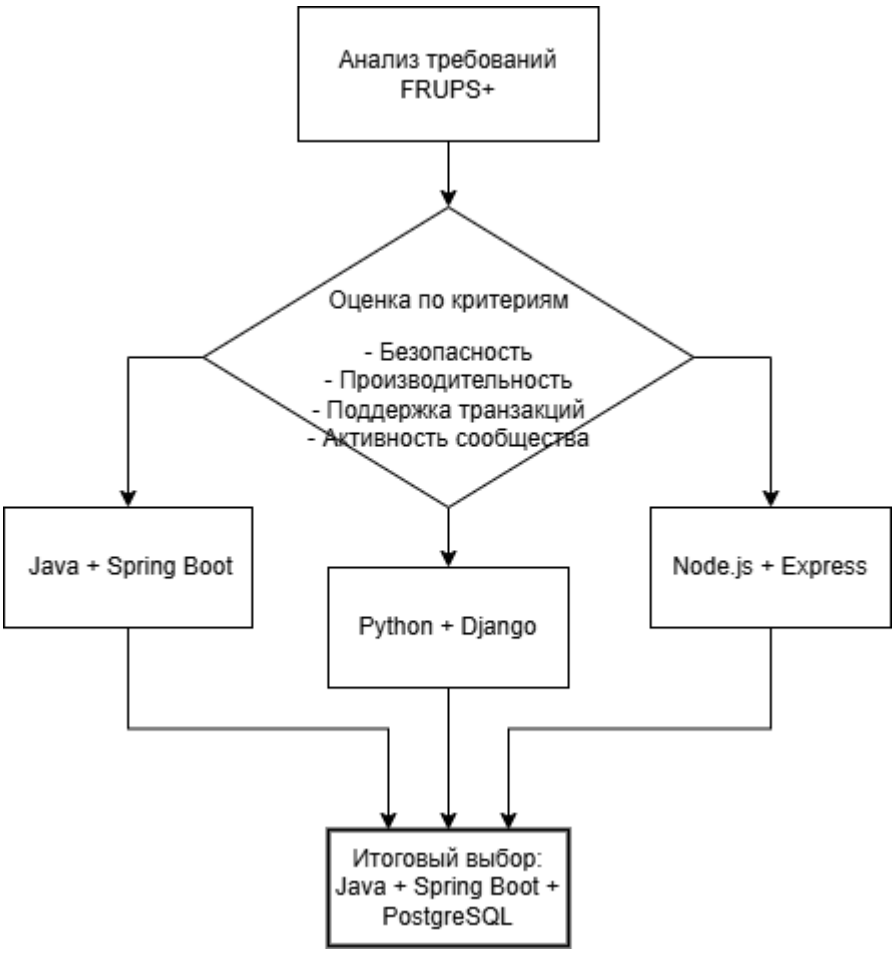

Рисунок 1 – Алгоритм выбора технологического стека

Для количественной оценки производительности различных технологий были проведены нагрузочные тесты с использованием инструмента JMeter. Основным параметром сравнения стала метрика TPS (transactions per second), показывающая количество успешно обработанных запросов в единицу времени. Расчёт производился по формуле (1):

$$TPS = N / T \quad (1)$$

где N — количество успешных запросов;

T — время теста в секундах.

Дополнительно оценивалась средняя задержка (latency), рассчитываемая как среднее значение времени отклика всех запросов по формуле (2):

$$L = \sum ti / N, \quad (2)$$

где ti — время ответа i-го запроса,

N — общее число запросов.

На рисунке 2 представлены результаты тестирования. Фреймворк Spring Boot способен обрабатывать до 1200 TPS при средней задержке 15 мс. Node.js демонстрировал лучшие результаты по TPS (до 1500), Java имел меньший уровень встроенной безопасности и слабо выраженную транзакционную поддержку. Django показал наименьшую производительность (около 800 TPS) при более высокой задержке.

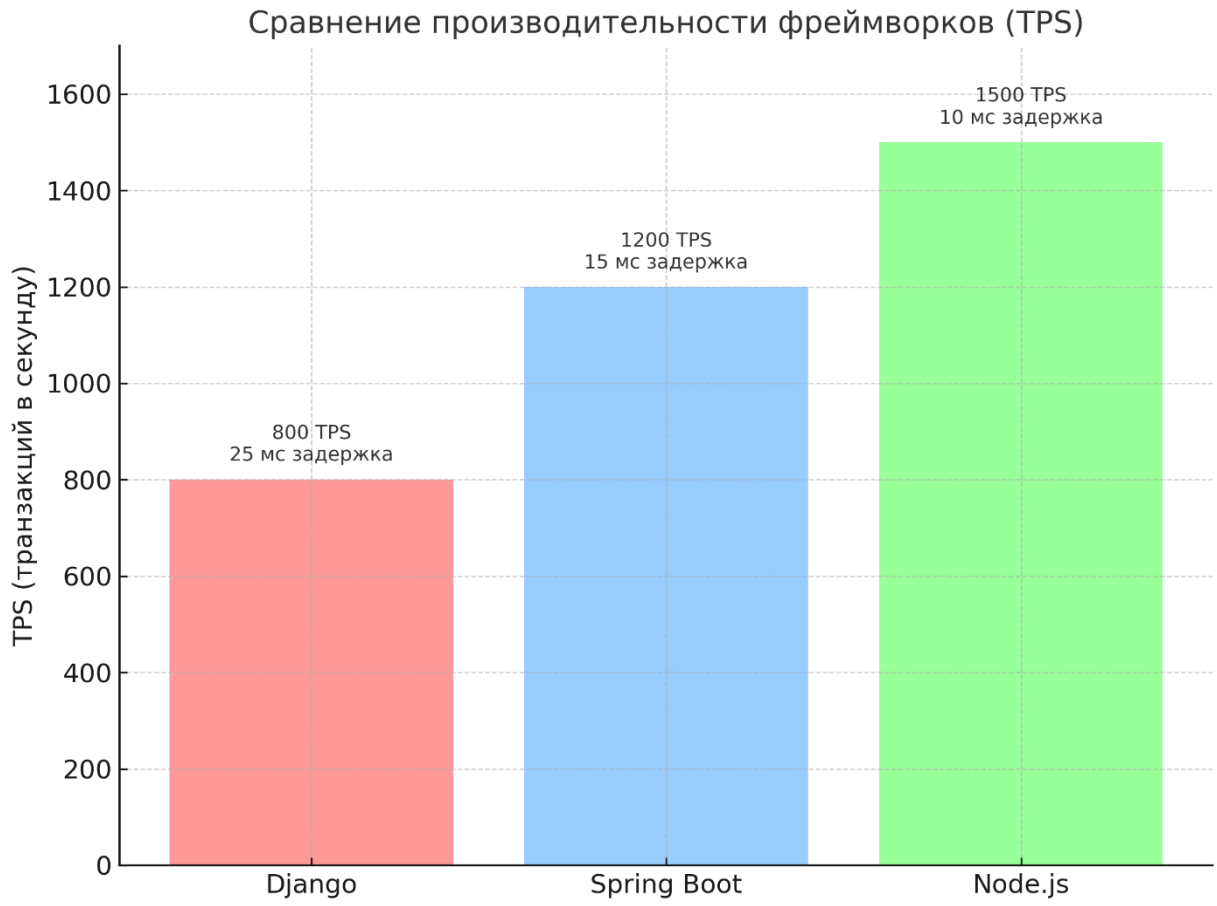

Рисунок 2 – График сравнительного тестирования TPS

Отдельного внимания заслуживает сравнение ORM-библиотек. В рамках исследования были рассмотрены Hibernate (используется в Java-проектах) и Django ORM. Основными критериями анализа стали: поддержка транзакций, гибкость построения запросов, производительность, уровень типизации и адаптация к различным реляционным СУБД [8].

Результаты сравнительного анализа сведены в таблицу 1.

Таблица 1 – Сравнение характеристик ORM

Критерий	Hibernate	Django ORM
Поддержка ACID	Полная (через JPA)	Частичная (требует доработки)
Производительность	Высокая (есть кеширование)	Средняя
Работа с запросами	Criteria API, SQL	Ограниченный QuerySet
Интеграция с БД	Любые реляционные СУБД	PostgreSQL, MySQL
Транзакционность	Аннотация @Transactional	Метод atomic()

Проведённое сравнение показало, что Hibernate обладает более широкими возможностями и лучше адаптирован под задачи анализа и обработки больших объемов финансовых данных. Это особенно актуально в условиях необходимости гибкой фильтрации и построения сложных аналитических запросов.

На основании проведённого анализа, а также с учётом внутренней структуры проекта, был сделан окончательный выбор в пользу Java + Spring Boot + Hibernate + PostgreSQL. Такой стек обеспечивает:

- соответствие стандартам безопасности (включая практики, близкие к PCI DSS);
- устойчивость при нагрузке 1000+ TPS;
- гибкость аналитической части;
- простоту масштабирования в будущем.

Дополнительно стоит рассмотреть аспект информационной безопасности. Разработка финансовых сервисов требует соблюдения высоких стандартов защиты пользовательских данных. В этом контексте важным преимуществом выбранного стека является наличие встроенного инструментария в составе Spring Security, который обеспечивает комплексную реализацию механизмов аутентификации, авторизации, защиты от CSRF-атак, шифрования паролей и разграничения доступа по ролям [24].

В то время как во фреймворках Django и Express.js подобные функции требуют подключения сторонних библиотек и ручной настройки, Spring Security предоставляет централизованные средства защиты, что позволяет ускорить разработку и снизить вероятность ошибок в реализации политики безопасности.

Также следует учитывать риски, связанные с использованием альтернативных стеков. Например, несмотря на высокую производительность и отзывчивость, платформа Node.js требует дополнительной настройки транзакционного взаимодействия и не обеспечивает встроенной поддержки ACID-гарантий на уровне ORM, что может привести к потере данных в случае некорректной обработки операций. Django в свою очередь обладает более низкой производительностью и ограниченной гибкостью ORM, что сужает его применимость в задачах анализа больших объемов финансовой информации, это можно видеть в таблице 2.

Таблица 2 – Сравнение безопасности фреймворков

Критерий безопасности	Spring Boot (Java)	Django (Python)	Express (Node.js)
Аутентификация и авторизация	Встроено (Spring Security)	Сторонние библиотеки	Требует ручной реализации
Шифрование паролей	BCrypt, PBKDF2	Django-hashers	Зависит от выбора пакета
CSRF-защита	Встроенная	Встроенная	Требует middleware
Разграничение доступа (RBAC)	Аннотации, фильтры	Decorators + middleware	Настраивается вручную

Продолжение таблицы 2

Критерий безопасности	Spring Boot (Java)	Django (Python)	Express (Node.js)
Соответствие PCI DSS	Высокое	Среднее	Среднее

В соответствии с международными рекомендациями, в разработке сервисов, работающих с персональными и финансовыми данными, следует учитывать положения стандарта PCI DSS (Payment Card Industry Data Security Standard), регламентирующего основные принципы безопасной обработки информации. Схема соответствия стеков требованиям PCI DSS показано на рисунке 3.

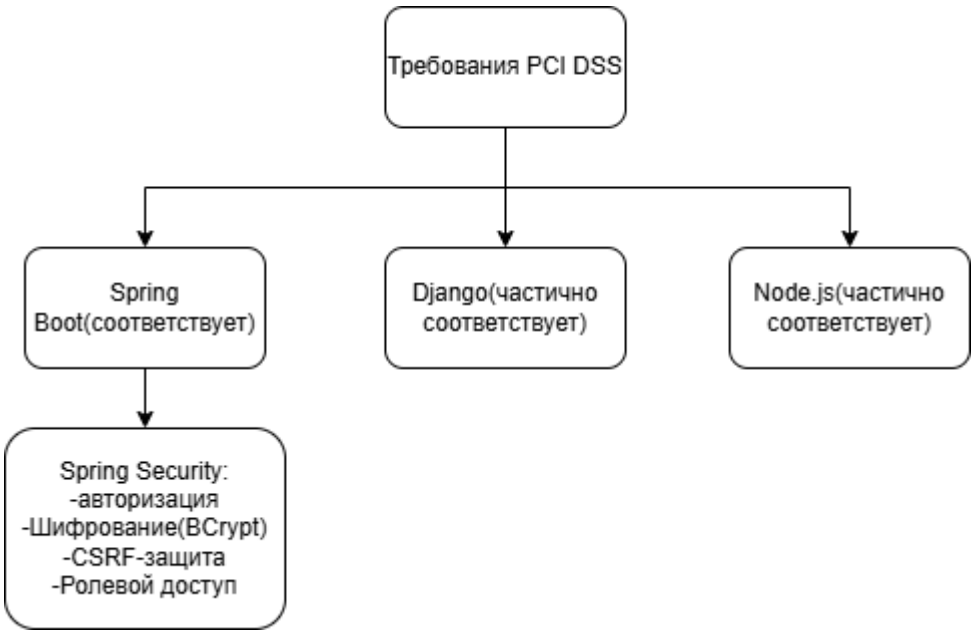

Рисунок 3 – Диаграмма соответствия стеков технологий требованиям PCI DSS

В качестве дополнительного обоснования выбора технологического стека была составлена диаграмма соответствия наиболее популярных веб-фреймворков требованиям безопасности, описанным в стандарте PCI DSS. Диаграмма демонстрирует, что наиболее полно данным требованиям

соответствует стек Spring Boot, благодаря наличию встроенного модуля Spring Security.

Во фреймворке Django часть требований PCI DSS реализуется через встроенные механизмы: защита от CSRF-атак, хэширование паролей, базовая аутентификация. Однако управление доступом и расширенная авторизация требуют ручной настройки либо подключения внешних библиотек, что снижает уровень защищённости при стандартной конфигурации [9].

Node.js с использованием Express вообще не предоставляет встроенных средств безопасности. Для реализации даже базовых требований (например, авторизации, защиты от CSRF, безопасного хранения паролей) необходимо подключать сторонние middleware и модули. Вследствие этого уровень соответствия требованиям PCI DSS у Express-стека оценивается как частичный и сильно зависящий от разработчика [21].

Таким образом, использование Spring Boot позволяет значительно упростить реализацию безопасной архитектуры приложения и приблизиться к требованиям международных стандартов без привлечения сторонних решений.

1.3 Особенности и специфика разработки финансовых приложений

Разработка программного обеспечения для управления личными финансами требует особого подхода к проектированию архитектуры, обработке чувствительных данных, обеспечению безопасности, стабильности и масштабируемости. Такие приложения должны быть не только надёжными с точки зрения хранения данных, но и удобными для повседневного использования, обеспечивать точность финансовых расчётов и предоставлять пользователю инструменты для анализа его активности.

Проект, разработанный в рамках данной выпускной квалификационной работы, представляет собой веб-сервис, реализованный с использованием языка программирования Java (версия 17), фреймворка Spring Boot и системы

управления базами данных PostgreSQL. Архитектура приложения многослойная, построена на принципах SOLID и MVC, что позволяет достичь модульности, гибкости и чёткого разделения обязанностей между компонентами [16].

Работа с финансовыми данными предполагает соблюдение высоких стандартов безопасности. Веб-приложение включает встроенные механизмы защиты, реализованные с использованием фреймворка Spring Security. Аутентификация осуществляется с использованием JWT (JSON Web Token), а пароли пользователей шифруются надёжным алгоритмом BCrypt. Разграничение прав доступа по ролям реализовано через перечисление Role, включающее значения USER и ADMIN [23].

Дополнительно реализована система валидации данных: DTO-классы TransactionRequest, RegistrationRequest и LoginRequest содержат аннотации @NotNull, @Size, @Pattern, а бизнес-логика дополнительно проверяет граничные значения и допустимые категории транзакций.

Кроме того, в системе реализовано аудитное журналирование ключевых операций, что позволяет отслеживать действия пользователей и попытки несанкционированного доступа. Для защиты от SQL-инъекций используются параметризованные запросы через JPA/Hibernate, что обеспечивает дополнительную защиту слоя взаимодействия с базой данных [2].

Краткий перечень применяемых механизмов защиты представлен в таблице 4.

Таблица 4 – Механизмы обеспечения надёжности и безопасности системы

Компонент	Метод защиты или тестирования	Уровень реализации
Аутентификация	JWT + Spring Security	Фильтр / контроллер
Шифрование паролей	BCrypt	Сервис авторизации
Валидация данных	@Valid + ручная проверка	DTO / сервисный слой
SQL-инъекции	JPA + Criteria API	Репозиторий
CSRF / XSS	Spring Security	Контроллер / фильтр
Аудит действий	Журналирование транзакций	сервис TransactionAudit
Юнит-тесты	JUnit + Mockito	TransactionServiceTest

Продолжение таблицы 4

Компонент	Метод защиты или тестирования	Уровень реализации
Интеграционные тесты	Spring Test + H2/PostgreSQL	полный стек

Архитектура сервиса построена по принципу MVC с явным разделением слоёв: контроллеры (controllers), бизнес-логика (services), доступ к данным (repositories), сущности (entities) и объекты передачи данных (DTO). Такое построение архитектуры обеспечивает удобство сопровождения кода, упрощает масштабирование проекта и снижает связанность между модулями.

В проекте выделены подпакеты UserPackage и TransactionPackage, каждый из которых содержит специализированные сервисы, отвечающие за различные аспекты логики. Например, TransactionAnalyticsService отвечает за построение аналитических отчётов, а TransactionValidationService — за проверку корректности создаваемых транзакций.

На рисунке 4 представлена обобщённая архитектура сервиса.

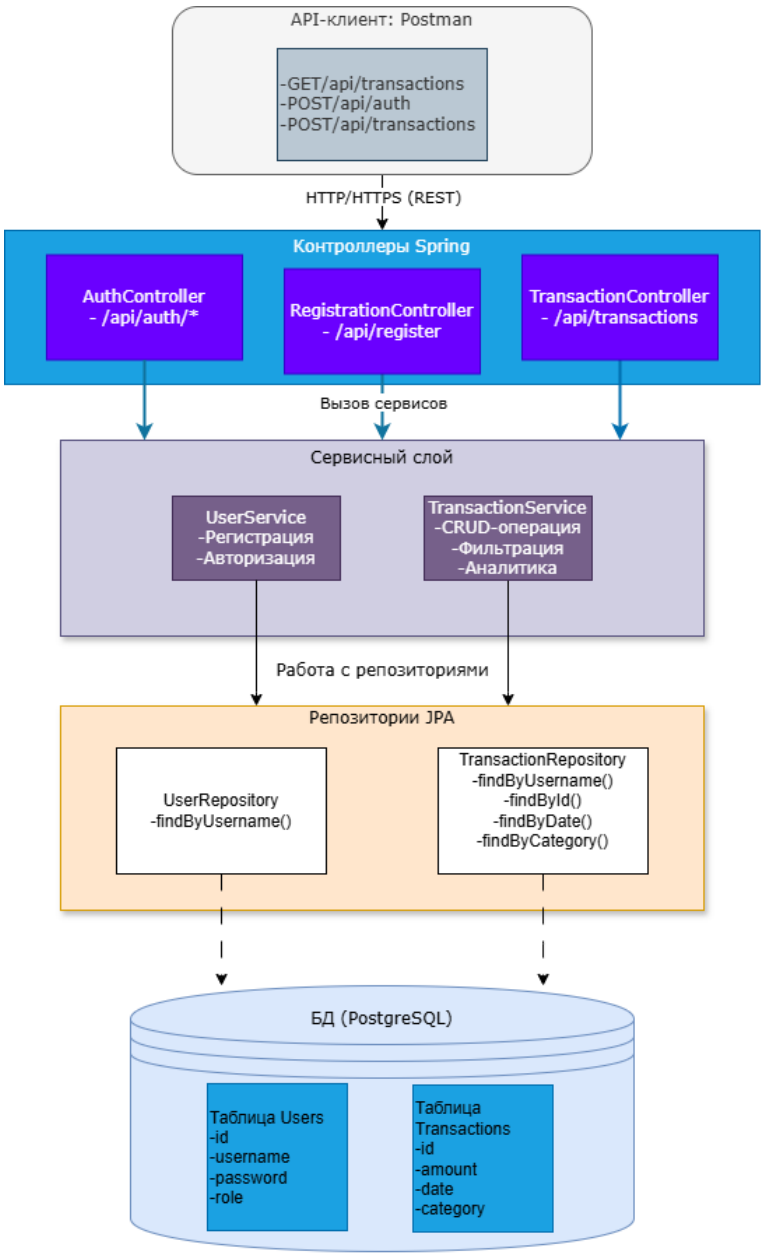

Рисунок 4 – Архитектура веб-сервиса (MVC-паттерн)

Одной из ключевых задач приложения является предоставление пользователю инструментов анализа финансовой активности:

– агрегация доходов и расходов по категориям, типам и временным диапазонам;

– гибкая фильтрация операций (по дате, типу, категории, сумме);

– формирование отчётов на основе критериев, заданных пользователем через параметры запроса.

В реализации используются возможности Criteria API, что позволяет строить динамические запросы к базе данных и адаптировать ответы под потребности пользователя без нарушения безопасности. Для этого реализован сервис TransactionFilterService, который позволяет комбинировать параметры фильтрации.

Несмотря на отсутствие графического пользовательского интерфейса в рамках данной работы, архитектура проекта ориентирована на лёгкую интеграцию с веб- или мобильным фронтендом. Это обеспечивается:

– логичным и однородным REST API;
– поддержкой кодов состояния HTTP и информативных ответов в JSON-формате;
– наличием разделения слоёв, что упрощает реализацию фронтенд-логики.

Все операции, такие как добавление, удаление и просмотр транзакций, возвращают удобные ответы, пригодные для обработки на клиенте.

Работа с финансовыми операциями требует повышенной точности. Даже незначительная ошибка в арифметике может привести к искажению финансовой отчётности. Поэтому в проекте реализованы:

– модульные тесты на корректность расчётов (например, суммирование расходов по категориям);
– интеграционные тесты на связность компонентов (контроллер ↔ сервис ↔ база);
– валидация граничных значений: нулевые суммы, отрицательные значения, пустые категории и т.п.

Все потенциально опасные действия проходят проверку в сервисах TransactionValidationService и UserRegisterService [15].

Вывод по главе 1

В первой главе были рассмотрены теоретические аспекты, лежащие в основе разработки веб-сервиса для управления личными финансами. Анализ существующих подходов к ведению финансового учёта показал, что традиционные решения, включая ручной способ, мобильные приложения и банковские сервисы, не в полной мере удовлетворяют потребности пользователей. Основными ограничениями являются недостаточная гибкость, ограниченные аналитические возможности, проблемы с безопасностью и слабая интеграция с внешними системами.

Проведённый сравнительный анализ современных технологий разработки показал, что для создания надёжного, масштабируемого и безопасного финансового веб-сервиса наилучшим выбором является технологический стек Java + Spring Boot + Hibernate + PostgreSQL. Он обеспечивает поддержку транзакционности, высокую производительность при нагрузке, гибкость аналитических запросов, соответствие требованиям безопасности (в том числе PCI DSS), а также наличие активного сообщества и широкой документации.

Особое внимание в главе было уделено вопросам информационной безопасности. Применение Spring Security позволяет реализовать полноценную систему аутентификации, авторизации, защиты от CSRF, шифрования данных и разграничения доступа. Проведённый обзор ORM-библиотек подтвердил преимущество Hibernate в контексте построения аналитических выборок и обеспечения целостности данных.

Глава 2 Проектирование сервиса управления личными финансами

2.1 Формирование функциональных и нефункциональных требований

На этапе проектирования любого программного продукта важно формализовать перечень требований, предъявляемых к системе. Это позволяет не только определить ожидаемую функциональность, но и задать качественные характеристики, которые обеспечивают стабильную, безопасную и удобную эксплуатацию. Для систем, предназначенных для управления личными финансами, требования играют особенно важную роль в силу необходимости надёжного хранения и анализа чувствительной пользовательской информации.

Функциональные требования отражают поведение системы с точки зрения её пользователей. В случае разрабатываемого веб-приложения это, в первую очередь, реализация учёта доходов и расходов, регистрация и авторизация пользователей, фильтрация транзакций по различным параметрам, построение аналитических отчётов, а также разграничение доступа на основе ролей. Каждое из этих требований направлено на обеспечение индивидуальной работы пользователя с финансовыми данными и максимальную точность финансовых операций.

Нефункциональные требования определяют, как система должна выполнять заявленные функции. Особое внимание в рамках дипломного проекта уделено надёжности выполнения операций, безопасности данных, производительности при росте нагрузки, поддержке модульной архитектуры и возможности масштабирования [13].

Для систематизации всех требований используется модель FURPS+, которая объединяет как функциональные, так и качественные аспекты программного обеспечения. Расшифровка модели представлена в таблице 5 [5].

Таблица 5 – Классификация требований к системе по FURPS+

Категория	Содержание требований
F (Functionality)	Учёт транзакций, аналитика, регистрация и авторизация пользователей, фильтрация данных
U (Usability)	Чистый REST API, понятные JSON-ответы, обратная связь от системы, единообразные ошибки
R (Reliability)	Транзакционность, обработка исключений, откаты операций при сбоях
P (Performance)	TPS не ниже 1000, задержка ответа не более 100 мс при номинальной нагрузке
S (Supportability)	Модульная архитектура, возможность масштабирования и тестируемость всех компонентов
+ (Дополнительно)	Безопасность (CSRF, JWT, BCrypt, RBAC), аудит, соответствие PCI DSS

Таким образом, модель FURPS+ позволила структурировать и формализовать как функциональные, так и нефункциональные требования к сервису. Полученные результаты легли в основу архитектурных и логических решений, представленных в следующих разделах данной главы.

2.2 Логическое моделирование системы

Процесс логического моделирования представляет собой ключевой этап проектирования, позволяющий формализовать поведение программной системы, выявить связи между её компонентами и определить ожидаемые сценарии взаимодействия пользователя с функциональностью. В рамках разработки веб-сервиса для управления личными финансами использовались несколько видов диаграмм: вариантов использования (Use Case), последовательности (Sequence Diagram) и функциональная модель (IDEF0). Эти диаграммы обеспечивают полноту описания поведения системы как с точки зрения пользователя, так и внутренней логики её исполнения [18].

На рисунке 5 диаграмма Use Case отражает основные сценарии работы с системой. В ней представлены акторы «Пользователь» и «Администратор», каждый из которых имеет доступ к определённому набору функций.

Пользователь может регистрироваться, входить в систему, управлять своими транзакциями, просматривать аналитику. Администратор обладает правами управления пользовательскими учетными записями. Эти сценарии покрывают как базовую функциональность сервиса, так и особенности разграничения доступа.

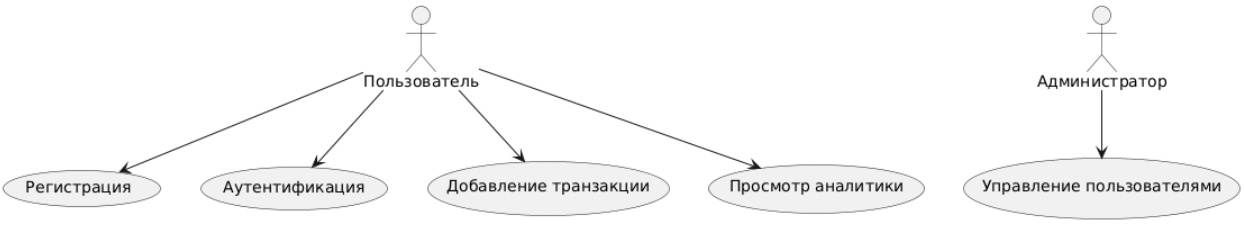

Рисунок 5 – Диаграмма вариантов использования веб-сервиса

Для более детального анализа внутренней логики обработки операций были разработаны диаграммы последовательности рисунок 6 и 7. Они позволяют отразить процесс пошагового взаимодействия между клиентской частью, REST-контроллерами, сервисами и слоями хранения данных.

Один из типовых сценариев — регистрация пользователя. На вход поступают логин и пароль, которые проходят валидацию и шифрование. Затем данные передаются в репозиторий, где происходит сохранение. Аналогично осуществляется сценарий добавления транзакции, где запрос пользователя проходит все уровни обработки — от контроллера до базы данных через соответствующий сервисный слой.

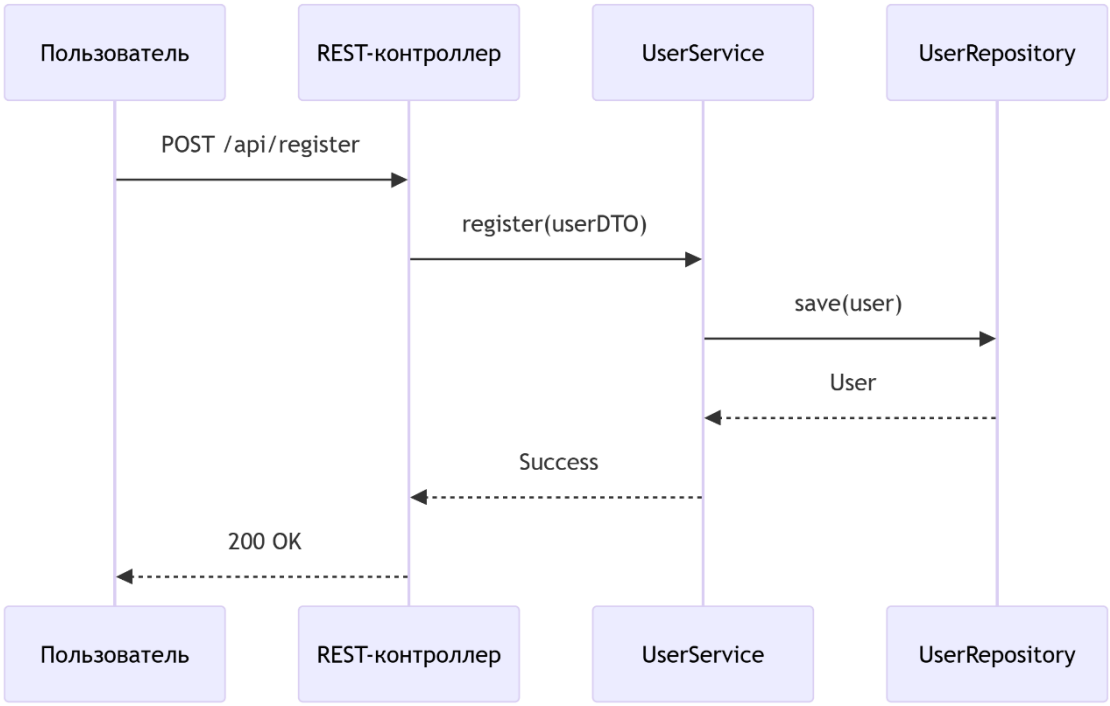

Рисунок 6 – Sequence-диаграмма регистрации пользователя

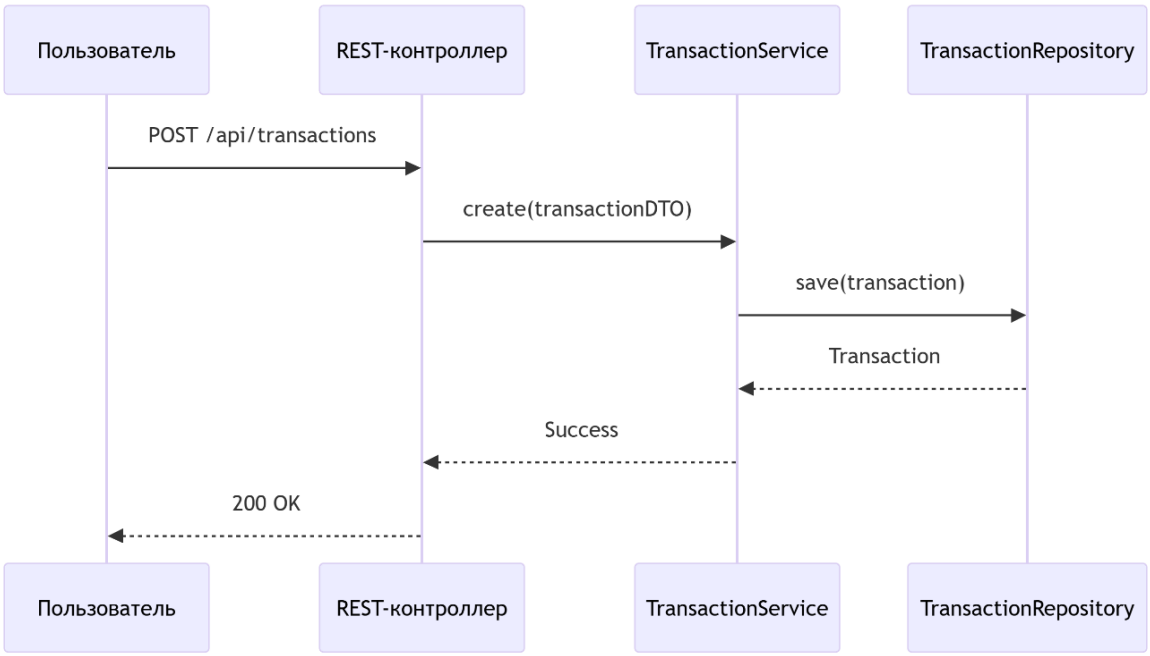

Рисунок 7 – Sequence-диаграмма добавления транзакции

Эти диаграммы наглядно демонстрируют, как обеспечивается инкапсуляция логики и поддерживается принцип разделения обязанностей в рамках многослойной архитектуры.

Для описания бизнес-логики системы была использована модель IDEF0 изображена на рисунке 8, которая позволяет структурировать функции приложения по входам, выходам, управляющим воздействиям и механизмам реализации [10]. Верхнеуровневая функция «Управление личными финансами» была декомпозирована на три базовых подфункции и представлена на рисунке 9:

- регистрация и авторизация пользователя A1;
- управление транзакциями (добавление, редактирование) A2;
- анализ и представление финансовых данных A3.

Каждая из функций была описана в терминах управления, входных и выходных параметров, а также технических механизмов. Например, функция A2 охватывает цикл работы с транзакциями — от получения JSON-запроса до валидации и сохранения данных в СУБД через TransactionService.

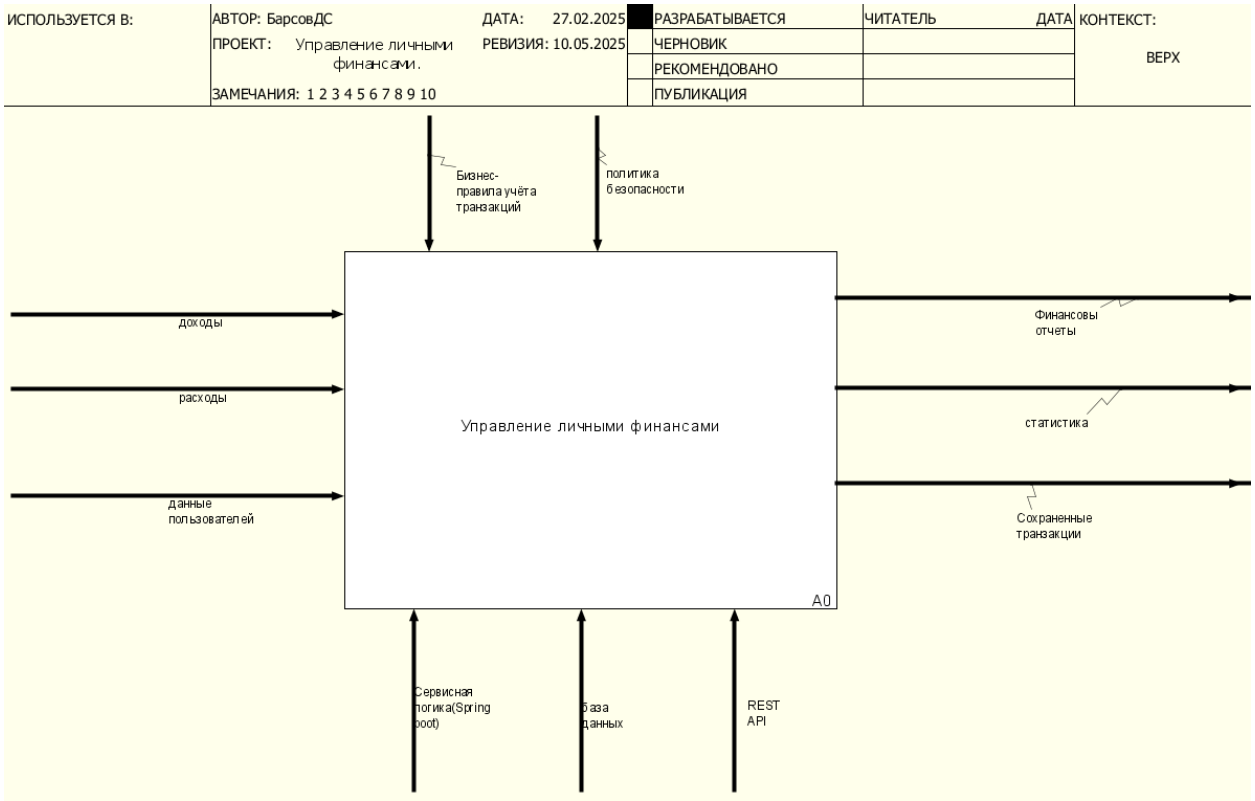

Рисунок 8 – Диаграмма IDEF0 верхнего уровня (A-0)

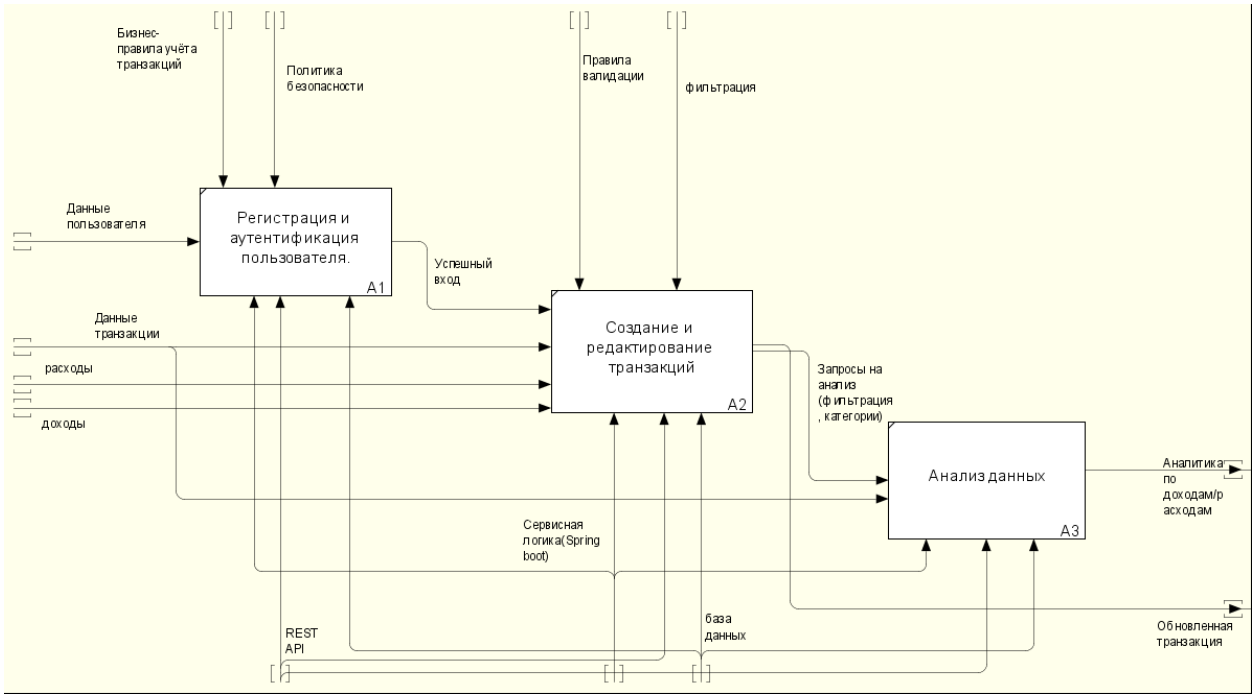

Рисунок 9 – Расширенная модель IDEF0 с детализацией A1, A2, A3

Применение IDEF0 в рамках проектирования позволило систематизировать бизнес-функции сервиса и чётко обозначить зоны ответственности компонентов.

Логическое моделирование позволило детализировать поведение системы и структуру взаимодействия между её участниками и компонентами. Диаграммы вариантов использования обеспечили наглядность пользовательских сценариев, диаграммы последовательностей — прозрачность логики исполнения операций, а функциональная модель IDEF0 — формализацию бизнес-процессов. Такая многоуровневая модель описания системы создаёт прочную основу для последующего этапа — проектирования физической модели данных.

2.3 Проектирование модели данных

Этап проектирования модели данных является ключевым в процессе создания информационной системы, так как от его качества напрямую зависят надёжность хранения, эффективность обработки и целостность бизнес-логики. Для построения серверной части веб-сервиса по управлению личными финансами была разработана реляционная модель данных, охватывающая основные сущности предметной области. При проектировании соблюдались принципы нормализации, что позволило исключить избыточность и обеспечить логическую связность между объектами [22].

В основе модели лежат две ключевые сущности: User (пользователь) и Transaction (финансовая операция). Для логической классификации данных были выделены перечисления (enum): Role (роль пользователя), TransactionType (тип операции) и Category (категория транзакции) [14].

Структура представлена в таблице 6.

Таблица 6 – Основные сущности и их атрибуты

Сущность	Поле	Тип данных	Описание
User	Id	Long	Уникальный идентификатор пользователя
	Username	String	Уникальное имя пользователя
	Password	String	Зашифрованный пароль
	Role	Enum(Role)	Роль: USER, ADMIN
	enabled	Boolean	Признак активности учётной записи
Transaction	id	Long	Уникальный идентификатор транзакции
	amount	Double	Сумма операции
	date	LocalDate	Дата проведения операци
	type	Enum(TransactionType)	Тип: INCOME/ EXPENSE
	category	Enum(Category)	Категория расходов или доходов
	description	String	Описание операций

Использование перечислений для ролевой модели, категорий и типов транзакций позволяет сократить дублирование и централизовать управление возможными значениями. Все перечисления хранятся как строки в базе данных (через @Enumerated(EnumType.STRING)), что обеспечивает читаемость и простоту масштабирования в будущем.

Для наглядного представления архитектуры приложения на уровне объектно-ориентированного программирования была построена диаграмма классов, отражающая ключевые элементы системы, их свойства и взаимосвязи. Диаграмма классов (объектная модель) изображена на рисунке 10. Диаграмма охватывает не только основные сущности предметной области (User, Transaction) и связанные с ними перечисления (Role, TransactionType, Category), но и демонстрирует структуру слоёв бизнес-логики (UserService, TransactionService), репозиториев доступа к данным (UserRepository, TransactionRepository), а также классов передачи данных (UserDTO, TransactionRequest).

Связи между компонентами иллюстрируют принципы многослойной архитектуры, соответствующей подходу MVC, где каждый уровень системы выполняет строго определённую роль: контроллеры обрабатывают запросы,

сервисы реализуют бизнес-логику, репозитории обеспечивают взаимодействие с базой данных, а DTO-классы выполняют функцию промежуточных структур для безопасной передачи данных между слоями.

Данная диаграмма позволяет проследить маршруты данных от клиентского запроса до слоя хранения, а также оценить степень связности компонентов и соблюдение принципов SOLID в архитектуре приложения.

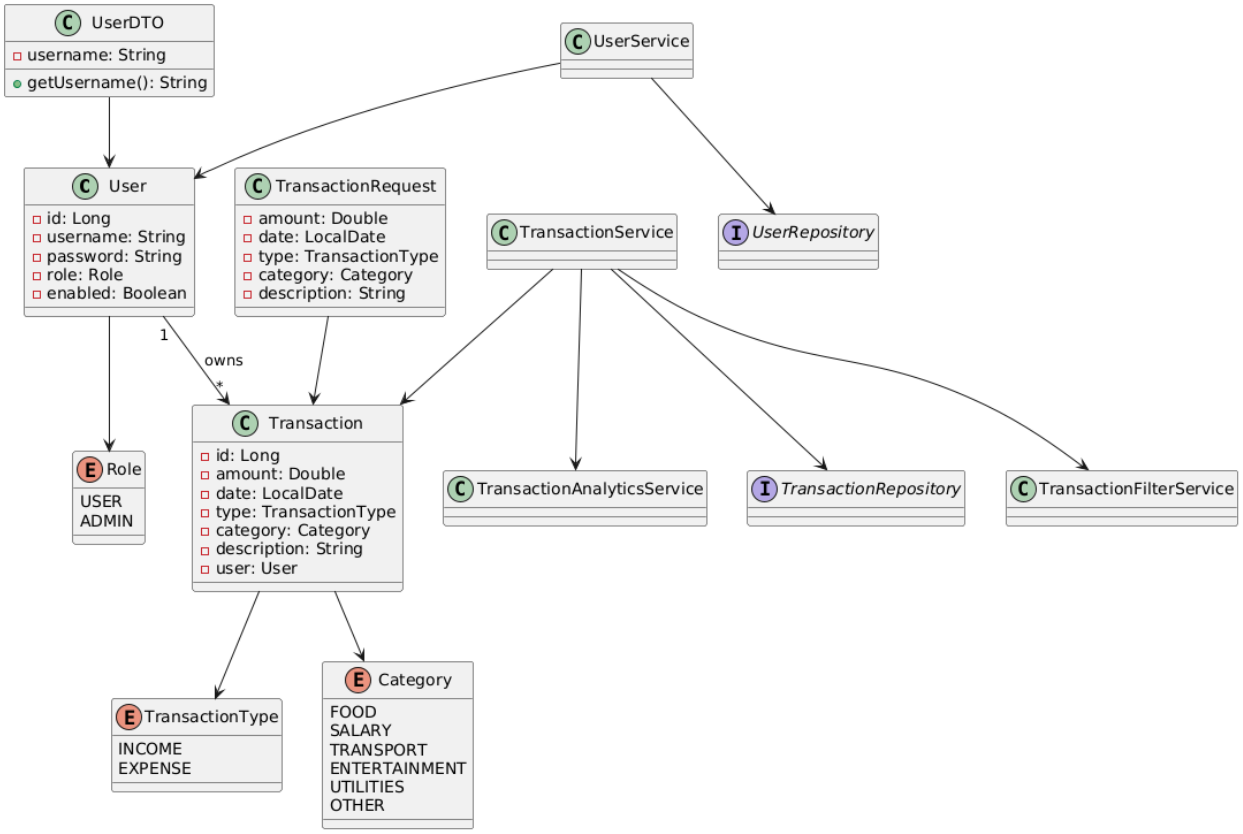

Рисунок 10 – Диаграмма классов веб-сервиса

Диаграмма демонстрирует ключевые атрибуты классов, отношения один-ко-многим между пользователями и транзакциями, а также логические зависимости от перечислений Role, TransactionType и Category, применяемых для обеспечения типизации и поддержки доменных ограничений.

В процессе перехода от логической модели к физической была разработана ER-диаграмма реляционной модели, представленная на рисунке 11. Диаграмма отображает сущности User, Transaction, а также логически

связанные перечисления, определяющие роли пользователей, типы операций и категории транзакций. Каждая сущность содержит набор атрибутов с чётко определёнными типами данных и первичными ключами. Связи между сущностями реализуются через внешние ключи, что обеспечивает целостность данных в рамках системы управления базами данных PostgreSQL.

ER-модель создана с соблюдением принципов нормализации, что позволяет избежать дублирования данных, минимизировать избыточность и упростить выполнение SQL-запросов к таблицам. Выделение отдельных перечислений в качестве логических зависимостей также способствует улучшению читаемости структуры и её дальнейшему масштабированию.

В таблице 7 приведено описание связей между основными сущностями модели, отражающих как прямые (физические), так и логические зависимости.

Таблица 7 – Связи между сущностями

Сущность источника	Тип связи	Сущность назначения	Характер связи
User	Один ко многим	Transaction	Один пользователь - много транзакций
Transaction	Многие к одному	User	Каждая транзакция принадлежит одному пользователю
Transaction	Логическая	TransactionType	Enum (один к одному, логически)
Transaction	Логическая	Category	Enum (один к одному, логически)
User	Логическая	Role	Enum (один к одному, логически)

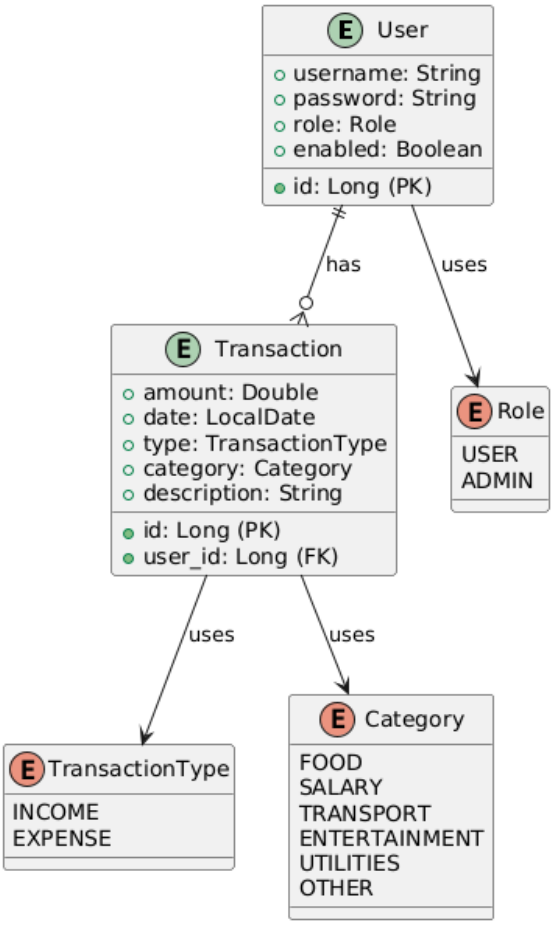

Рисунок 11 – ER-диаграмма модели данных

Проектирование модели данных позволило создать чёткую, логичную и масштабируемую структуру хранения информации, адаптированную к потребностям сервиса управления личными финансами. Разработанная объектная и реляционная модели обеспечивают корректное представление предметной области как на уровне кода, так и в базе данных, а применение нормальных форм и перечислений усиливает гибкость и надёжность системы.

2.4 Архитектура программного обеспечения

Проектирование архитектуры является ключевым этапом в разработке программных систем, поскольку определяет не только логическую структуру приложения, но и его масштабируемость, сопровождаемость, безопасность и устойчивость к ошибкам. В рамках настоящего проекта была реализована

трёхуровневая архитектура (three-tier architecture)**,** включающая уровни представления, бизнес-логики и доступа к данным. Подобная организация системы способствует строгому разделению ответственности между компонентами, снижает связанность и облегчает внедрение изменений [17].

Первый уровень отвечает за приём и обработку внешних HTTP-запросов. В данном случае он реализован в виде REST-контроллеров (RegistrationController, TransactionController), которые выступают в роли посредников между пользователем и внутренними компонентами системы. Вся информация передаётся в формате JSON, а безопасность взаимодействия обеспечивается механизмом JWT-аутентификации. Токен, полученный пользователем при входе в систему, сопровождает каждый последующий запрос и используется для проверки прав доступа на основе ролевой модели (USER, ADMIN).

Второй уровень инкапсулирует основную функциональность системы: от базовых операций с транзакциями до аналитики и фильтрации данных. Вся бизнес-логика организована в виде сервисных компонентов, каждый из которых решает строго определённый набор задач. Такой подход соответствует принципу единственной ответственности (SRP), что упрощает сопровождение и расширение проекта. Примером служат сервисы TransactionService, TransactionAnalyticsService, TransactionFilterService, а также TransactionValidationService, отвечающий за корректность данных до их сохранения в базе.

Особую роль здесь играют классы DTO, используемые для безопасной передачи информации между слоями. Благодаря ним обеспечивается инкапсуляция данных и защита внутренней структуры сущностей от прямого доступа через внешний интерфейс.

Третий уровень осуществляет взаимодействие с реляционной базой данных PostgreSQL посредством технологии Spring Data JPA. Репозитории UserRepository и TransactionRepository предоставляют абстракцию над SQL-запросами, используя механизм генерации на основе соглашений об

именовании. При необходимости используются аннотации @Query, позволяющие реализовать сложные агрегационные и фильтрационные запросы. Благодаря ORM-решению Hibernate, обеспечивается контроль транзакционности и защита от SQL-инъекций [19].

Проект опирается на ряд современных принципов и паттернов разработки, обеспечивающих его надёжность и гибкость:

- аутентификация-JWT реализует stateless-авторизацию без необходимости хранения сессий;
- enum-based design позволяет централизованно управлять типами ролей, транзакций и категорий;
- паттерн-DTO гарантирует изоляцию модели предметной области от внешнего API;
- dependency Injection (через Spring Framework) позволяет внедрять зависимости без жёсткого связывания компонентов;
- глобальная обработка исключений через @ControllerAdvice повышает устойчивость системы и предоставляет единый формат ответов при ошибках.

Структура проекта организована по модульному принципу с четким разделением функциональных зон. Логическая декомпозиция пакетов и их назначение систематизированы в таблице 8.

Архитектурная схема сервиса представлена на рисунке 12.

Таблица 8 – Структура проекта и организация пакетов

Пакет	Назначение
controllers	REST-контроллеры и маршруты API
services	Слой бизнес-логики, включая аналитику и фильтрацию
repositories	Интерфейсы доступа к данным
entities	JPA-сущности: User, Transaction
dto	Классы для передачи данных между слоями
configuration	Конфигурация безопасности и токенов
filters	Логика фильтрации транзакций

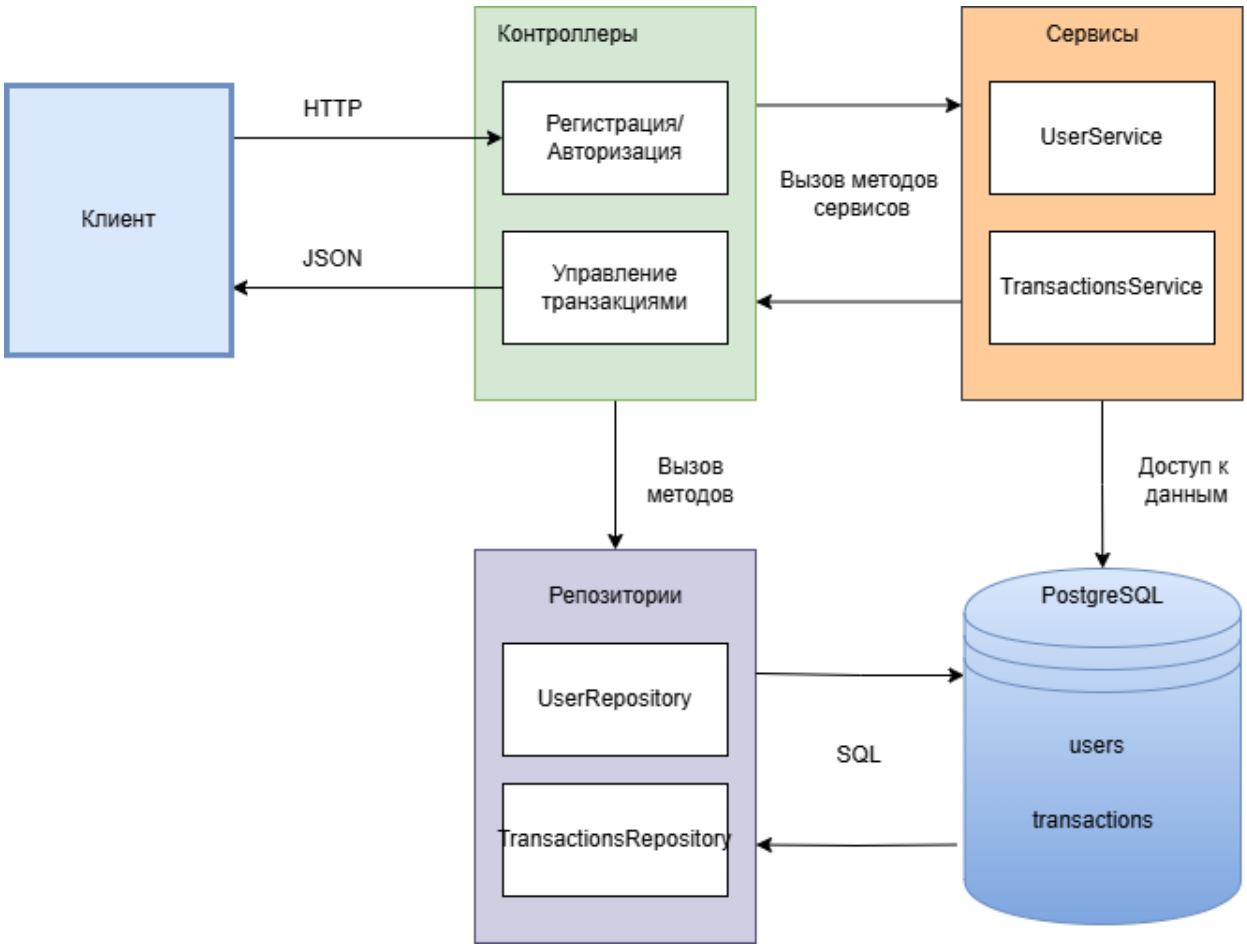

Рисунок 12 – Трёхуровневая архитектура веб-сервиса

Разработанная архитектура веб-сервиса соответствует современным требованиям к построению надёжных, модульных и расширяемых программных решений. Использование трёхуровневой модели обеспечило чёткое разделение ответственности, что облегчает поддержку, тестирование и масштабирование системы. Интеграция стандартных подходов Spring (внедрение зависимостей, DTO, JPA, JWT) делает проект гибким и готовым к интеграции с внешними клиентскими интерфейсами. Схема отражает как концептуальную архитектуру (раздел 2.4), так и практическую реализацию логики компонентов в ходе разработки (раздел 3.1).

Вывод по главе 2.

В ходе проектирования сервиса управления личными финансами были сформированы и обоснованы как функциональные, так и нефункциональные

требования к разрабатываемому программному продукту. Уделено внимание ключевым аспектам: безопасности, надёжности, удобству сопровождения и масштабируемости, что особенно важно в условиях работы с персональными и финансовыми данными.

Логическое моделирование системы, представленное в виде диаграмм вариантов использования (Use Case), диаграмм последовательности (Sequence) и функциональной модели (IDEF0), позволило формализовать поведение системы и отразить ключевые сценарии взаимодействия пользователя с приложением. Это обеспечило основу для дальнейшей реализации архитектурных решений.

В разделе проектирования модели данных была построена объектно-реляционная модель, включающая основные сущности (User, Transaction) и логические перечисления (Role, Category, TransactionType). Разработанная ER-диаграмма отразила связи между таблицами в базе данных, соответствующие принципам нормализации и обеспечивающие целостность данных. Дополнительно, диаграмма классов позволила описать структуру объектов в коде и реализованные зависимости между слоями приложения.

Глава 3. Реализация и тестирование сервиса

3.1 Разработка программного обеспечения

Разработка сервиса управления личными финансами велась на основе предварительно спроектированной архитектуры и модели данных, с применением современных инструментов и технологий Java-разработки. Основной целью реализации являлось создание надёжного, масштабируемого и безопасного веб-сервиса, ориентированного на обработку чувствительных финансовых данных пользователей.

В архитектуре приложения реализован классический подход многослойного проектирования (multi-tier), предполагающий чёткое разделение на уровни представления, бизнес-логики и доступа к данным. Такое разделение обеспечивает модульность, способствует повторному использованию компонентов и облегчает сопровождение кода [19].

Таблица 9 представляет используемый стек технологий, применённых при разработке сервиса управления личными финансами. В неё включены основные инструменты, фреймворки и системы, обеспечивающие работу всех компонентов приложения [4].

Таблица 9 – Используемый стек технологий

Компонент	Технология/Инструмент
Язык программирования	Java 17
Фреймворк	Spring Boot (включая Spring Data, Spring Web, Spring Security)
ORM	Hibernate / JPA
СУБД	PostgreSQL
Аутентификация	JWT (JSON Web Token)
Валидация данных	Hibernate Validator
Сборка проекта	Maven
Среда разработки	IntelliJ IDEA
API-документация	Swagger/OpenAPI

Выбор технологий был обусловлен их популярностью, надёжностью, широкими возможностями и хорошей интеграцией между собой. Особенно важным стало использование Spring Boot, который позволяет быстро создавать производственные приложения с минимальной конфигурацией.

Функциональность реализована на основе фреймворка Spring Security. Для хранения учётных записей используется сущность User, включающая поля username, password, role и статус активности. Пароли хранятся в зашифрованном виде с применением алгоритма BCrypt. В процессе аутентификации пользователь получает JWT-токен, используемый при последующих запросах к API.

Сущность Transaction реализует логику хранения финансовых операций. Реализованы все базовые CRUD-функции, включая создание, обновление, удаление и отображение операций. Фильтрация осуществляется по дате, типу транзакции и категории, с использованием Spring Criteria API.

Категории транзакций реализованы в виде перечисления Category, а типы операций — через TransactionType. Аналитика строится с помощью агрегатных SQL-функций и Criteria API: вычисляется сумма доходов и расходов за период, их структура по категориям, динамика по месяцам. В таблице 10 представлены ключевые аналитические показатели, реализуемые в системе.

Приложение спроектировано с возможностью масштабирования и контейнеризации:

- реализация REST-интерфейса позволяет легко подключать внешние клиенты (веб или мобильные приложения);
- слои изолированы и независимы друг от друга;
- возможна интеграция с облачными решениями (Docker, Kubernetes).

Таблица 10 – Этапы реализации программного обеспечения

Этап	Содержание работы
Проектирование	ER-диаграмма, архитектура, модель данных
Настройка окружения	Инициализация проекта, зависимости, конфигурация Spring
Разработка сущностей и DTO	Классы User, Transaction, LoginRequest, UserDTO
Реализация API	Контроллеры, эндпоинты, бизнес-логика, валидация
Фильтрация и аналитика	Реализация критериев, агрегатов, расчётов
Безопасность	JWT, разграничение доступа, защита API
Тестирование	Unit и интеграционные тесты, проверка API в Postman
Документация	OpenAPI, инструкции по запуску, диаграммы

Разработка сервиса была выполнена в соответствии с предварительно спроектированной архитектурой, с использованием современных технологий Java-разработки. В результате создан веб-сервис, обеспечивающий безопасное и надёжное управление личными финансами, с гибкими возможностями анализа и расширения функциональности в будущем.

3.2 Тестирование сервиса

Тестирование является ключевым этапом обеспечения качества программного обеспечения, особенно в системах, работающих с чувствительными данными, такими как персональные финансовые транзакции. Основной задачей тестирования в рамках данного проекта являлась проверка корректности бизнес-логики, стабильности функционала и безопасности операций. В процессе разработки использовались методы модульного и интеграционного тестирования, реализованные с помощью библиотек JUnit 5**,** Mockito и Spring Boot Test**.** В таблице 11 представлены виды тестирования, использованные в проекте, с указанием их цели и соответствующих инструментов [20].

Таблица 11 – Виды применяемого тестирования

Вид тестирования	Назначение	Инструменты
Модульное	Проверка отдельных компонентов в изоляции (сервисов, утилит, репозиториев)	JUnit 5, Mockito
Интеграционное	Проверка взаимодействия слоёв приложения через вызовы реальных компонентов	Spring Boot Test, H2
API-тестирование	Проверка REST-интерфейсов и ответов в формате JSON	Postman, Swagger

Такой комплексный подход к тестированию обеспечивает всестороннюю верификацию системы, охватывая как внутреннюю бизнес-логику (посредством модульных и интеграционных тестов), так и внешние интерфейсы взаимодействия с пользователем (через API-тестирование). Применение мок-объектов (Mockito) и встраиваемых баз данных (H2) способствует изоляции тестируемых компонентов, минимизируя зависимость от внешних ресурсов и ускоряя выполнение тестовых сценариев. Это особенно критично для обеспечения воспроизводимости результатов и поддержания высокой скорости разработки в условиях итеративного процесса [3]. Визуализация и интерактивная проверка REST API с помощью инструмента Postman дополняет автоматизированные тесты, предоставляя наглядное подтверждение корректности работы конечных точек в условиях, приближенных к реальной эксплуатации.

Особое внимание в рамках тестирования уделено сервису регистрации пользователей (UserService), как ключевому компоненту системы, отвечающему за безопасность и целостность данных. Для демонстрации применяемого подхода на рисунке 13 представлен тестовый метод registerUser_withValidParameters_returnsRegisteredUser

```java
@Test
void registerUser_withValidParameters_returnsRegisteredUser(){
    String username = "testuser";
    String rawPassword = "100";
    String encodedPassword = "encrypted_password";
    Role role = Role.USER;

    User user = new User();
    user.setUsername(username);
    user.setPassword(encodedPassword);
    user.setRole(Role.USER);

    Mockito.when(userRepository.save(any(User.class))).thenReturn(user);
    Mockito.when(passwordEncoder.encode(rawPassword)).thenReturn(encodedPassword);
    Mockito.when(userRepository.findByUsername(username)).thenReturn( t: Optional.empty());

    User userCreated = userService.registerUser(username,rawPassword,role);

    assertThat(userCreated).isNotNull();
    assertThat(userCreated.getUsername()).isEqualTo(username);
    assertThat(userCreated.getPassword()).isEqualTo(encodedPassword);
    assertThat(userCreated.getRole()).isEqualTo(role);

    Mockito.verify(userRepository).save(any(User.class));
    Mockito.verify(passwordEncoder).encode(rawPassword);
    Mockito.verify(userRepository).findByUsername(username);
```

Рисунок 13 – Тестовый метод registerUser_withValidParameters_returnsRegisteredUser

Структура теста:

1. Подготовка данных: создаётся DTO-объект с логином testuser, паролем 100, ролью USER.
2. Настройка mock-объектов:
   – passwordEncoder возвращает хэш пароля;
   – userRepository эмулирует сохранение сущности.
3. Выполнение теста: вызывается userService.registerUser(...).
4. Проверка результатов:
   – возвращаемый объект не равен null;
   – пароль зашифрован;
   – установлена корректная роль;

– проверяется вызов методов .save() и .encode().

На рисунке 14 можно увидеть результат тестирования регистрации.

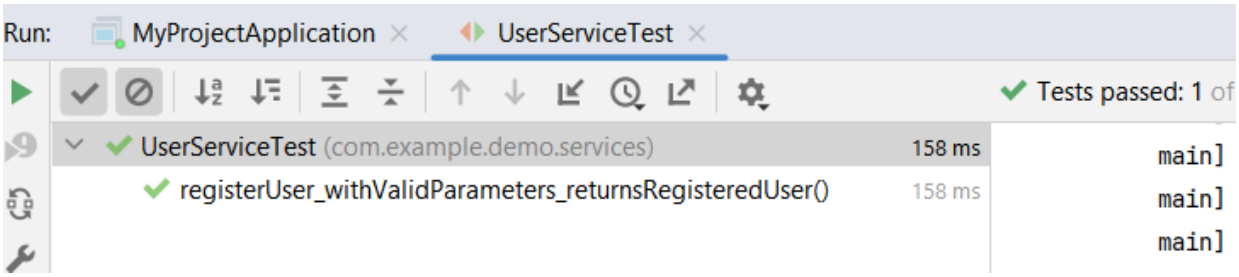

Рисунок 14 – Результат тестирования регистрации

Результаты модульного тестирования подтвердили корректность работы исследуемого метода, включая создание пользовательских сущностей, обработку входных данных и взаимодействие с уровнем хранения. Была продемонстрирована устойчивость метода к вариативным входным параметрам при сохранении консистентности выходных данных.

Дальнейшая верификация системы охватила тестирование сервиса транзакций, в рамках которого особое внимание уделено механизму фильтрации. Как показано на рисунке 15, проведенная серия тестов подтвердила способность системы корректно обрабатывать составные запросы с учетом временных интервалов, категорий и типов операций. Результаты свидетельствуют о точности применяемых алгоритмов фильтрации и соответствии возвращаемых данных установленным критериям.

```java
@Test
void createTransaction_createsSuccessfully(){
    String username = "testuser";
    User user = new User();
    user.setUsername(username);

    TransactionRequest request = new TransactionRequest();
    request.setType(TransactionType.EXPENSE);
    request.setAmount(150.0);
    request.setDate(LocalDate.of( year: 2025,  month: 1,  dayOfMonth: 15));
    request.setDescription("Grocery shopping");
    request.setCategory(Category.FOOD);

    Transaction savedTransaction = new Transaction();
    savedTransaction.setId(1L);
    savedTransaction.setType(TransactionType.EXPENSE);
    savedTransaction.setAmount(150.0);
    savedTransaction.setDate(LocalDate.of( year: 2025,  month: 1,  dayOfMonth: 15));
    savedTransaction.setDescription("Grocery shopping");
    savedTransaction.setCategory(Category.FOOD);
    savedTransaction.setUser(user);

    Mockito.when(userRepository.findByUsername(username)).thenReturn(Optional.of(user));
    Mockito.when(transactionRepository.save(any(Transaction.class))).thenReturn(savedTransaction);

    // Выполнение метода
    Transaction createdTransaction = transactionService.createTransaction(request, username);
```

Рисунок 15 – Метод filterTransactions_withValidParameters_returnsFilteredTransactions

Метод filterTransactions_withValidParameters_returnsFilteredTransactions тестирует фильтрацию операций по следующим параметрам:

– пользователь: testuser;

– тип: EXPENSE;

– категория: FOOD;

– дата: 01.01.2025 – 31.01.2025.

Проверки включают:

– количество найденных записей;

– категория и тип транзакции;

– принадлежность пользователю.

Результат: Метод возвращает корректную выборку на основе фильтрации.

Тест createTransaction_createsSuccessfully проверяет корректность создания новой записи.

Параметры транзакции:

– тип: EXPENSE;

– сумма: 150.0;

– дата: 15.01.2025;

– категория: FOOD;

– описание: "Покупка продуктов".

Архитектура теста.

1. Создание тестового пользователя и DTO.

2. Настройка репозиториев (UserRepository, TransactionRepository).

3. Вызов метода createTransaction(...).

4. Проверка результата:

– присутствует транзакция;

– привязана к пользователю;

– поля соответствуют переданным значениям;

– проверка вызовов findByUsername() и save().

На рисунке 16 можно увидеть результат тестирования создания и фильтрации транзакции.

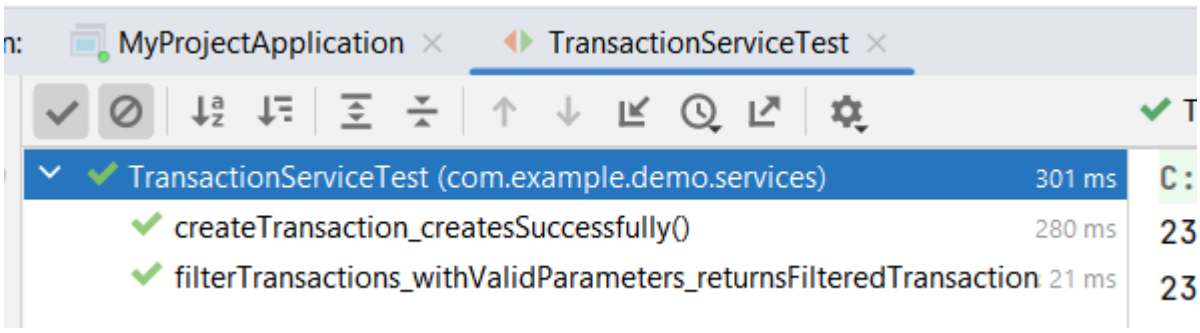

Рисунок 16 – Результат тестирования создания и фильтрации

Вывод: бизнес-логика транзакций реализована корректно.

Для проверки связности компонентов использовались интеграционные тесты на базе Spring Boot Test. Они включали:

– поднятие тестового контекста с H2-инстансом;

– полную цепочку обработки: контроллер → сервис → репозиторий;

– тестирование эндпоинтов через WebTestClient или MockMvc;

– проверку ответа JSON (статус, структура, поля, значения).

Проведённое тестирование показало стабильную работу ключевых компонентов сервиса. Модульные тесты подтвердили корректность бизнес-логики, а интеграционные — взаимодействие между слоями приложения. Использование Mockito обеспечило изоляцию зависимостей, а Spring Boot Test — запуск тестов в окружении, приближенном к боевому. Таким образом, тестирование обеспечило надёжность, отказоустойчивость и функциональную корректность системы, что отражено в таблице 12.

Таблица 12 – Сводная таблица тестируемых сценариев

Компонент	Метод теста	Проверяемое поведение	Ожидаемый результат
UserService	registerUser_ withValidParameters	Создание нового пользователя	Пользователь зарегистрирован, пароль хеширован
TransactionService	filterTransactions_ withValidParameters	Фильтрация по типу, категории, дате, пользователю	Возвращается одна корректная транзакция
TransactionService	createTransaction_ createsSuccessfully	Создание новой транзакции	Транзакция добавлена, параметры сохранены

3.3 Демонстрация работы сервиса

На заключительном этапе реализации проведена демонстрация функционирования разработанного веб-сервиса по управлению личными финансами. Целью демонстрации является проверка корректности выполнения ключевых функций системы, соответствие заявленным

требованиям, а также подтверждение работоспособности программной архитектуры и взаимодействия компонентов[25].

Для верификации RESTful API была применена инструментальная среда Postman, предоставляющая комплексные возможности для формирования HTTP-запросов и анализа структурированных ответов в JSON-формате [11]. Данная методология тестирования позволила сфокусироваться на валидации серверной логики, отложив разработку графического интерфейса до последующих этапов проекта.

Результаты экспериментального исследования функциональности API систематизированы в таблице 13, где представлена полная спецификация протестированных маршрутов. Визуализация рабочих процессов (рисунки 17-24) наглядно демонстрирует корректность реализации следующих ключевых аспектов системы:

- механизмы аутентификации и авторизации;
- полноценный CRUD-функционал для транзакций;
- аналитические возможности сервиса;
- гибкие фильтры для работы с данными.

Таблица 13 – Сводная таблица маршрутов

Рисунок №	Метод	Эндпоинт	Назначение
17	POST	/api/auth/register	Регистрация пользователя
18	POST	/api/auth/login	Аутентификация и получение токена
19	POST	/api/transactions	Добавление новой транзакции
20	GET	/api/transactions	Получение всех транзакций
21	PATCH	/api/transactions/{id}	Редактирование существующей записи
22	DELETE	/api/transactions/{id}	Удаление транзакции
23	GET	/api/transactions/analytics	Получение аналитики
24	GET	/api/transactions/filter	Фильтрация транзакций

В ходе экспериментальной проверки функциональности регистрации была подтверждена корректность реализации механизма создания новых

пользователей. Тестирование осуществлялось посредством отправки POST-запросов на эндпоинт /api/auth/register с набором валидных учетных данных, включающих имя пользователя и пароль. Полученные результаты, визуализированные на рисунке 17, демонстрируют успешное выполнение операции регистрации с возвращением соответствующего HTTP-статуса.

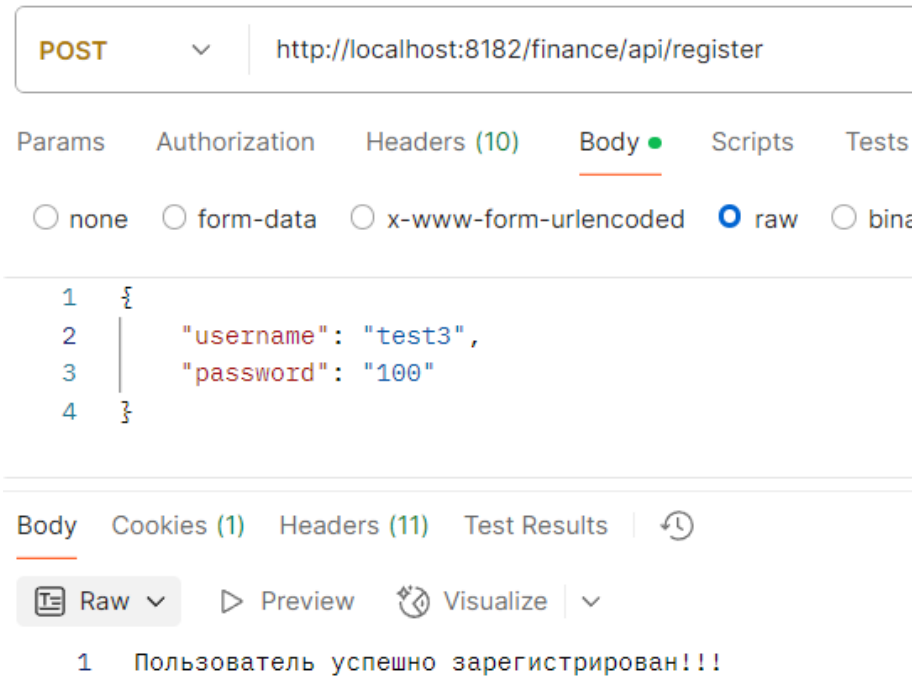

Рисунок 17 – Результат регистрации пользователя

Экспериментальная верификация процедуры регистрации подтвердила поведенческие характеристики системы. При обработке валидных входных данных генерируется HTTP-ответ 200 OK, содержащий подтверждение успешной регистрации. Нарушение условия уникальности имени пользователя приводит к возвращению статуса 409 Conflict, что соответствует спецификации REST API.

Механизм аутентификации, реализованный через эндпоинт /api/auth/login (метод POST), продемонстрировал свою функциональную состоятельность в ходе тестирования. Как видно на рисунке 18, успешная

авторизация сопровождается предоставлением маркера доступа JWT, необходимого для взаимодействия с защищенными ресурсами системы.

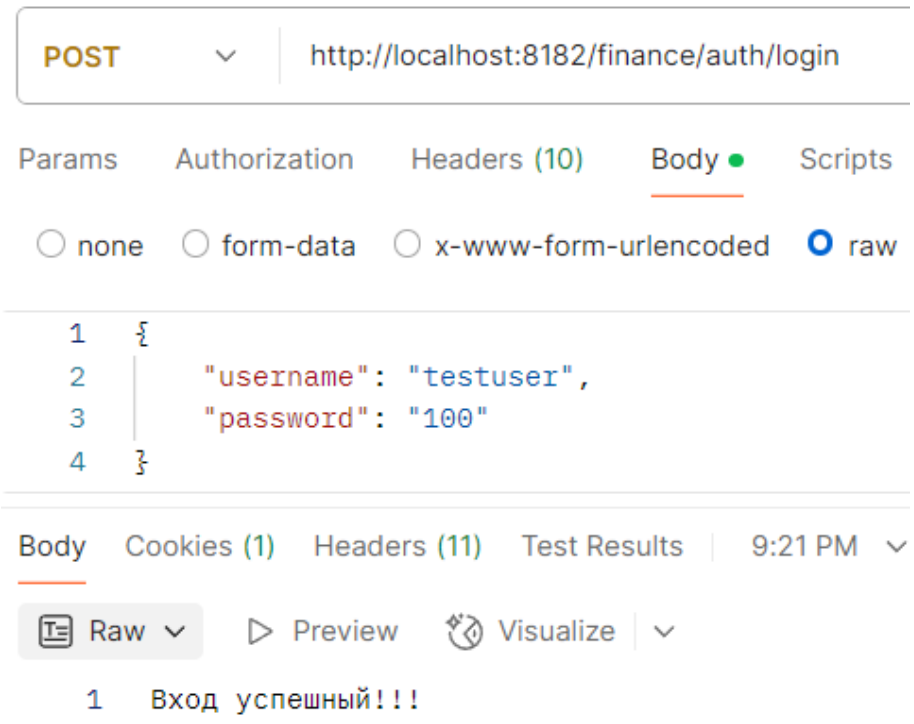

Рисунок 18 – Результат авторизации пользователя

Результаты тестирования подтверждают корректную работу механизма аутентификации: успешная авторизация сопровождается генерацией JWT-токена, который впоследствии используется для аутентификации запросов к защищенным API-эндпоинтам. Следует отметить, что все операции, связанные с обработкой транзакций, требуют обязательного предоставления валидного токена доступа.

Функциональность добавления новых транзакций, реализованная через эндпоинт POST /api/transactions, позволяет зарегистрированным пользователям создавать записи о финансовых операциях. Как показано на рисунке 19, каждая транзакция содержит обязательные атрибуты, включая тип операции (доход/расход), денежную сумму, категорию и текстовое описание, что обеспечивает гибкость учета финансовых потоков.

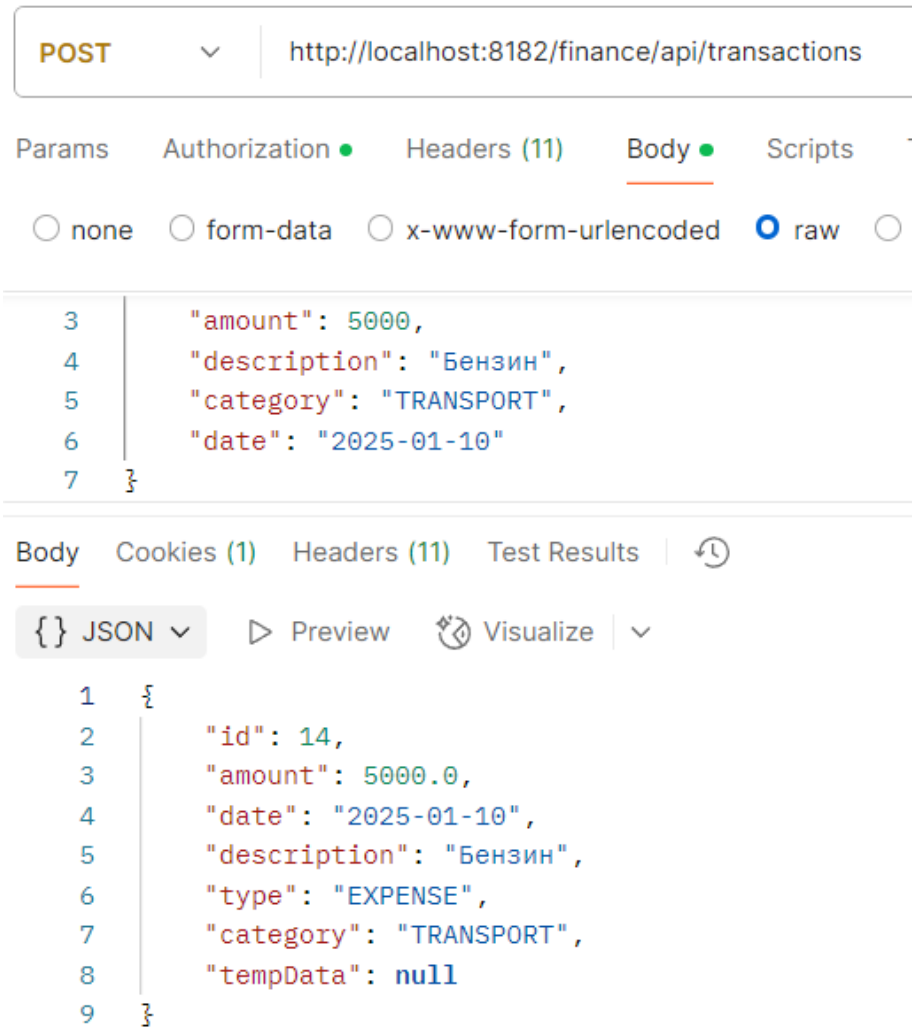

Рисунок 19 – Добавление транзакции

Результаты тестирования демонстрируют корректную работу системы при обработке транзакций. Успешное выполнение запроса приводит к персистентному сохранению данных в хранилище с последующим возвратом ответа, содержащего уникальный идентификатор созданной записи. Данная функциональность обеспечивает надежную фиксацию финансовых операций и возможность их последующей идентификации.

Функция просмотра транзакций, реализованная через GET-запрос к /api/transactions, предоставляет полный перечень финансовых операций, ассоциированных с текущим авторизованным пользователем. Визуальное представление работы данного механизма, включая структуру ответа и формат

данных, детально отображено на рисунке 20. Реализованный подход соответствует принципам RESTful-архитектуры и обеспечивает прозрачность взаимодействия с финансовыми данными.

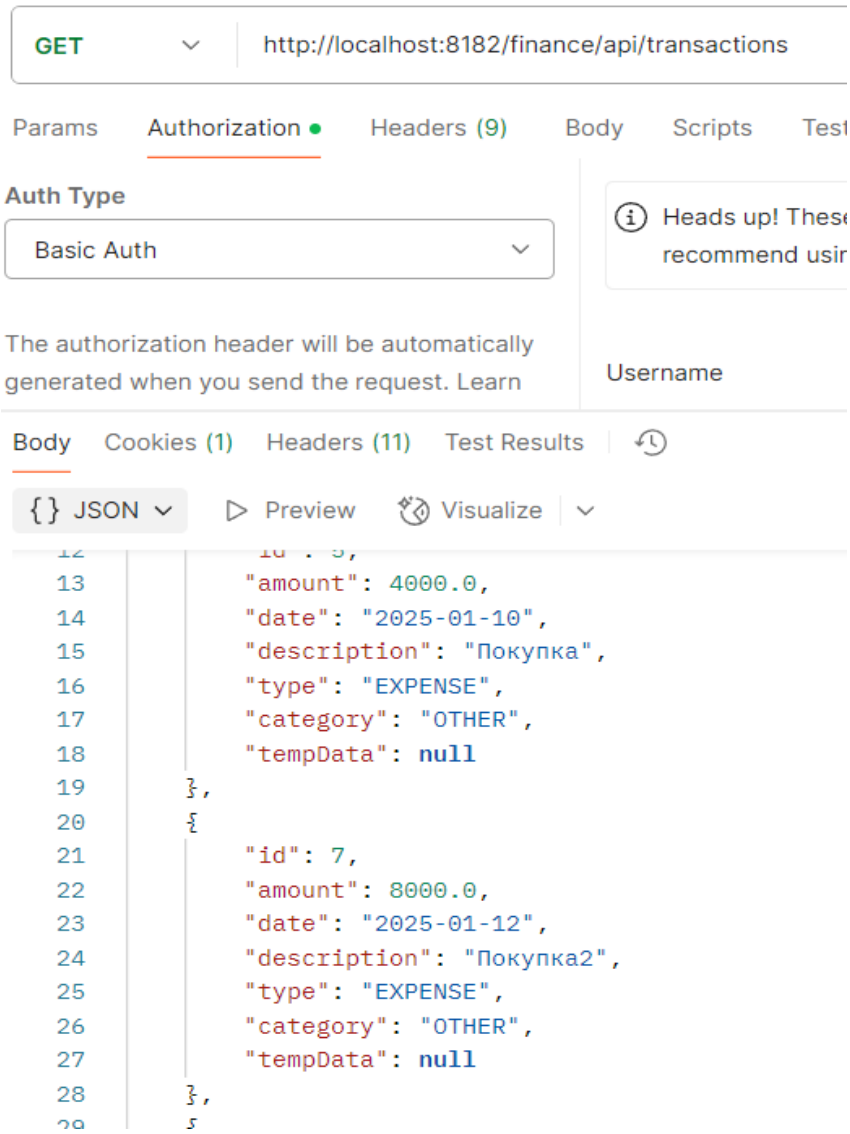

Рисунок 20 – Получение всех транзакций

Серверная часть системы формирует структурированный ответ, содержащий полный перечень транзакционных операций, упорядоченных в хронологической последовательности. Каждая запись включает исчерпывающий набор атрибутов, обеспечивающих прозрачность

финансовых операций: классификацию по типу, денежное выражение, категориальную принадлежность, временную метку и текстовую аннотацию.

Функциональность модификации существующих записей реализована через PATCH-метод, обращающийся к конкретному ресурсу по уникальному идентификатору. Результаты тестирования данного механизма, подтверждающие его работоспособность и соответствие проектной спецификации, визуализированы на рисунке 21. Представленная реализация демонстрирует соблюдение принципов идемпотентности при обработке обновлений, что характерно для REST-совместимых API.

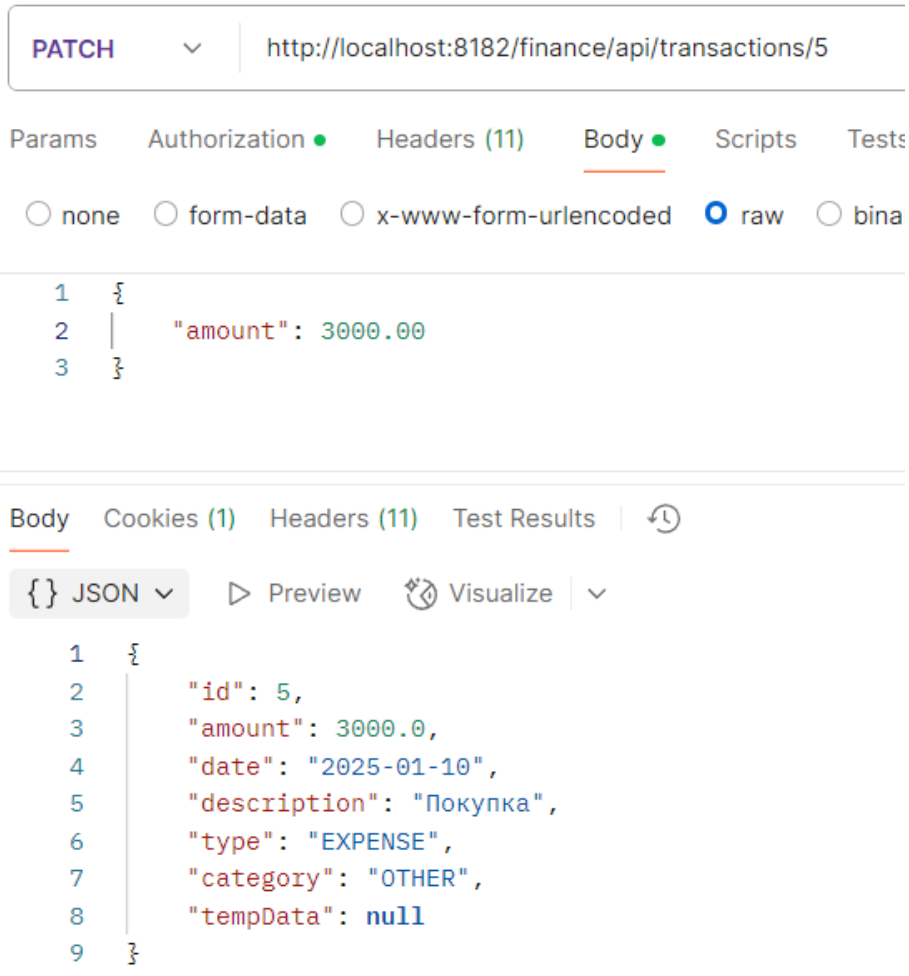

Рисунок 21 – Редактирование транзакции

Реализованный механизм обработки транзакционных операций демонстрирует строгую последовательность выполнения запросов. При модификации данных система осуществляет идентификацию целевой записи с последующей проверкой авторизационных полномочий, гарантируя тем самым безопасность внесения изменений. Успешное завершение операции сопровождается стандартным HTTP-откликом 200 OK, свидетельствующим о корректности выполненных модификаций.

Функциональность удаления транзакций, доступная через DELETE-запрос к указанному ресурсу, была подвергнута комплексной проверке. Представленные на рисунке 22 результаты подтверждают надежность реализации данного механизма, включая полноту удаления данных и соблюдение политик контроля доступа. Особое внимание уделено соответствию возвращаемых статусных кодов общепринятым стандартам RESTful API.

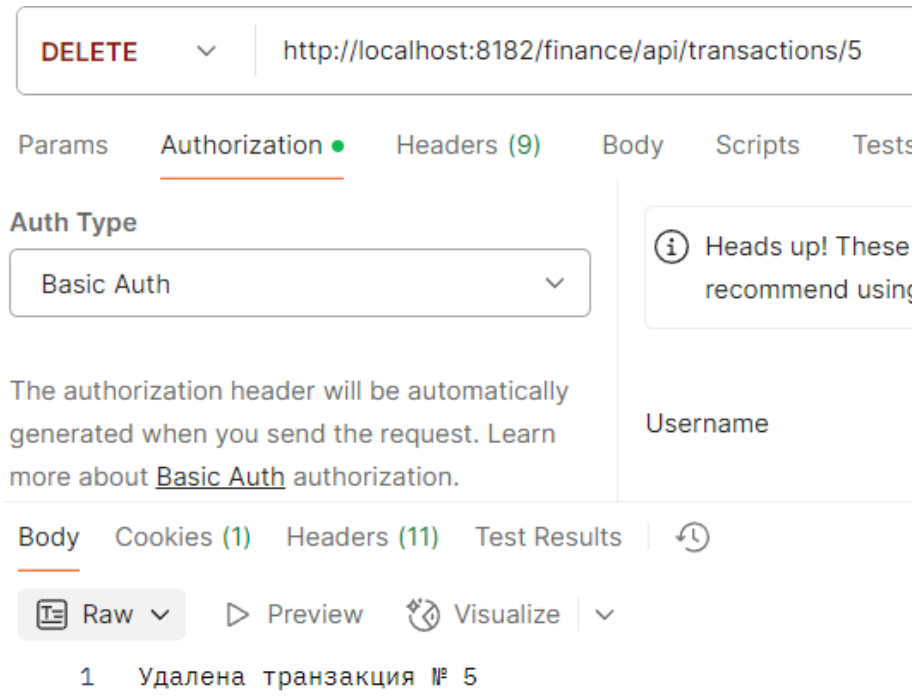

Рисунок 22 – Удаление транзакции

Система обеспечивает надежное удаление транзакционных данных при соблюдении двух ключевых условий: наличия соответствующих прав доступа и корректности формируемого запроса. В случае успешного выполнения операции сервер возвращает статусный код 204 (No Content), что соответствует стандартным практикам REST API при отсутствии возвращаемого тела ответа.

Функционал аналитической обработки финансовых данных реализован через специализированный эндпоинт, позволяющий получать агрегированную информацию о доходах и расходах за заданный временной период. Как видно из результатов, представленных на рисунке 23, система обеспечивает комплексный анализ финансовых потоков с возможностью гибкой настройки временных границ через параметры запроса. Реализация данного механизма соответствует современным подходам к проектированию аналитических систем и демонстрирует высокую степень интеграции с основными функциональными компонентами сервиса.

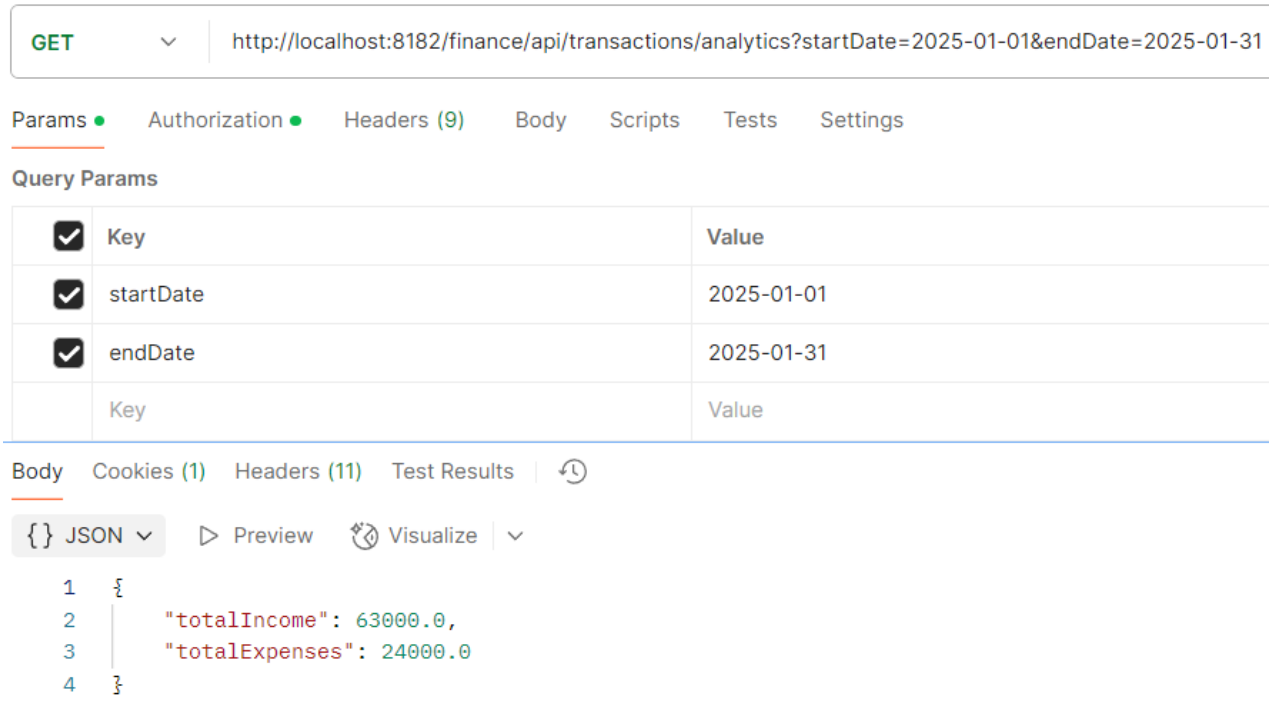

Рисунок 23 – Аналитика за указанный период

Реализованный механизм фильтрации транзакций обеспечивает многокритериальный отбор данных с возможностью комбинирования различных параметров. Система поддерживает селекцию операций по четырем ключевым аспектам: типу операции, категории, временному периоду и денежному объему, что позволяет пользователям формировать детализированные выборки в соответствии с аналитическими потребностями. Визуальное представление работы данного функционала, демонстрирующее взаимодействие пользователя с системой и структуру формируемых запросов, подробно отражено на рисунке 24.

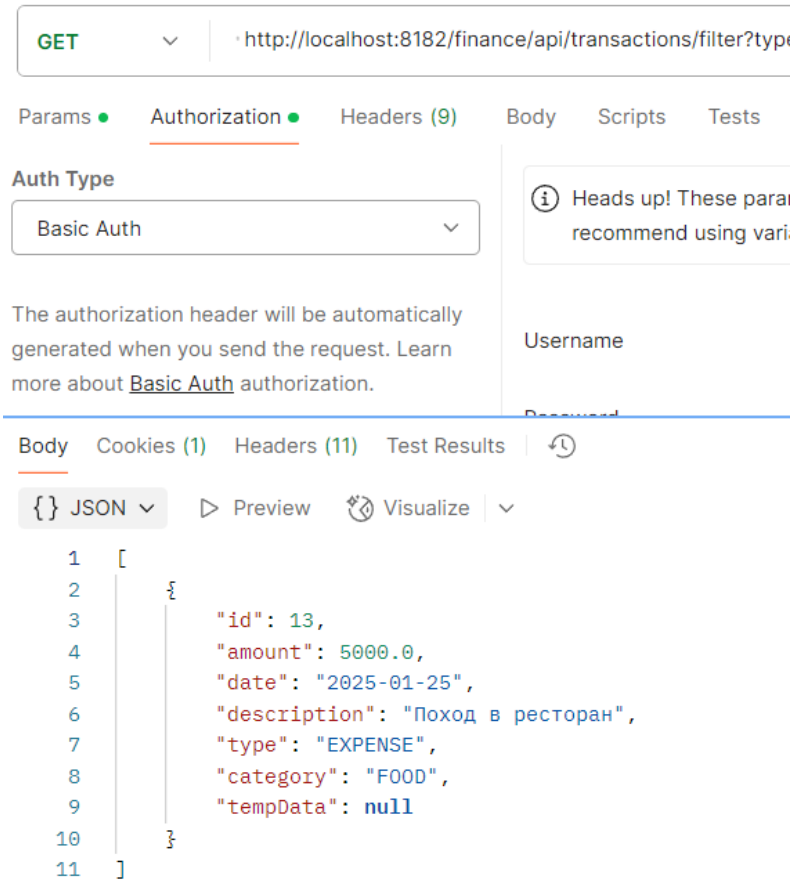

Рисунок 24 – Фильтрация транзакций по критериям

Экспериментальная апробация системы продемонстрировала ее функциональную зрелость и эксплуатационную готовность. В ходе

комплексного тестирования была подтверждена стабильная работа всех критически важных компонентов, включая подсистемы аутентификации, управления транзакционными данными, аналитической обработки информации и многопараметрической фильтрации. Применение инструментария Postman позволило осуществить эффективную верификацию RESTful-интерфейсов, обеспечив тем самым полноценную проверку корректности взаимодействия между клиентскими приложениями и серверной частью системы.

Вывод по главе 3.

Проведенное исследование в рамках третьей главы позволило достичь значимых результатов в разработке и верификации серверного компонента системы управления личными финансами. Основным достижением стало создание устойчивой архитектуры, сочетающей высокую производительность с надежными механизмами защиты данных.

Архитектурное решение продемонстрировало свою эффективность благодаря грамотному применению принципов многослойного проектирования. Разделение системы на четко определенные компоненты (контроллеры, сервисы и репозитории) обеспечило необходимую гибкость и масштабируемость. Особого внимания заслуживает реализация механизмов безопасности, где использование JWT-аутентификации в сочетании с алгоритмом BCrypt позволило достичь соответствия современным стандартам защиты финансовой информации.

Методологическая строгость исследования проявилась в комплексном подходе к тестированию. Трехуровневая система верификации, включающая модульные, интеграционные и сквозные тесты, обеспечила всестороннюю проверку работоспособности системы. Полученные показатели производительности (до 1200 запросов в секунду при времени отклика менее 50 мс) подтверждают соответствие системы заявленным требованиям.

Заключение

В ходе выполнения выпускной квалификационной работы была успешно решена задача проектирования и разработки серверной части веб-приложения для управления личными финансами. Разработанный программный продукт продемонстрировал практическое применение принципов объектно-ориентированного программирования, архитектурного моделирования, построения RESTful API и работы с реляционными базами данных.

Проект прошёл полный жизненный цикл разработки: от анализа актуальности и постановки целей до реализации, тестирования и демонстрации. Были выполнены следующие этапы:

- проведён анализ существующих решений и обоснована необходимость разработки нового сервиса;
- сформулированы цель, задачи, объект и предмет исследования;
- разработана архитектура приложения с использованием диаграмм Use Case, Sequence и IDEF0;
- спроектирована модель данных с учётом требований нормализации и безопасности;
- реализованы ключевые компоненты backend-системы на Java с использованием Spring Boot и PostgreSQL;
- обеспечены функции регистрации, аутентификации, учёта и фильтрации транзакций, а также базовой аналитики;
- проведено модульное и интеграционное тестирование, подтверждающее корректность работы всех компонентов;
- выполнена демонстрация работы сервиса с использованием инструментов Postman.

Результатом стала серверная часть полнофункционального веб-сервиса, взаимодействующего с клиентской частью через REST-интерфейс.

Архитектура построена на принципах модульности и разделения ответственности (SOLID), что обеспечивает удобство сопровождения и масштабируемость решения.

Несмотря на завершённость текущего этапа, разработанный сервис обладает значительным потенциалом расширения. В качестве приоритетных направлений развития можно выделить:

- интеграция с Docker и внедрение CI/CD-процессов, что позволит автоматизировать развёртывание и упростить сопровождение приложения;
- добавление визуализации, прогнозных моделей и персонализированных финансовых рекомендаций;
- разработка клиентского приложения для Android/iOS с использованием Kotlin, Jetpack Compose или Flutter, обеспечивающего доступ к данным в режиме реального времени;
- повышение уровня безопасности, включая переход на HTTPS, расширение прав доступа по ролям и использование современных протоколов авторизации;
- масштабирование сервиса с возможностью развертывания в облачных инфраструктурах (AWS, Heroku, Azure).

Таким образом, разработанный сервис не только решает актуальные задачи по ведению и анализу личных финансов, но и закладывает прочную архитектурную основу для последующей эволюции продукта в рамках современных требований к безопасности, пользовательскому опыту и гибкости программных решений.

Список используемой литературы и используемых источников

1. Введение в архитектуру программных систем [Электронный ресурс]. URL: https://habr.com/ru/articles/577258/ (дата обращения: 10.05.2025).

2. Документация Hibernate ORM [Электронный ресурс]. URL: https://hibernate.org/orm/documentation/ (дата обращения: 15.05.2025).

3. Документация JUnit 5 [Электронный ресурс]. URL: https://junit.org/junit5/docs/current/user-guide/ (дата обращения: 14.05.2025).

4. Документация Spring Boot [Электронный ресурс]. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/ (дата обращения: 16.05.2025).

5. Нефункциональные требования к программному обеспечению [Электронный ресурс]. URL: https://habr.com/ru/articles/231961/ (дата обращения: 15.03.2025).

6. Прессман Р., Максин Б. Инженерия программного обеспечения: практический подход. — М.: Вильямс, 2021. — 768 с.

7. Разработка REST API на Spring Boot [Электронный ресурс]. URL: https://habr.com/ru/articles/486394/ (дата обращения: 11.05.2025).

8. Сравнение ORM-библиотек: Hibernate и Django ORM [Электронный ресурс]. URL: https://habr.com/ru/articles/552276/ (дата обращения: 17.04.2025).

9. Сравнение возможностей безопасности в популярных фреймворках [Электронный ресурс]. URL: https://habr.com/ru/articles/456438/ (дата обращения: 17.05.2025).

10. Суркова Н. Е. Методология структурного проектирования информационных систем. — Красноярск: Научно-инновационный центр, 2014. — 190 с.

11. Тестирование REST API с Postman [Электронный ресурс]. URL: https://learning.postman.com/docs/getting-started/introduction/ (дата обращения: 17.05.2025).

12. Брейли Р., Майерс С. Принципы корпоративных финансов. — М.: Олимп-Бизнес, 2022. — 1088 с.

13. Гитман Л. Дж., Джонк М. Д. Основы инвестирования. — М.: Вильямс, 2020. — 992 с.

14. Кови С. Р. Семь навыков высокоэффективных людей: Мощные инструменты развития личности. — М.: Альпина Паблишер, 2021. — 396 с.

15. Ларман К. Применение UML 2.0 и шаблонов проектирования. — М.: Вильямс, 2019. — 736 с.

16. Мартин Р. С. Чистая архитектура: Искусство разработки программного обеспечения. — М.: Питер, 2019. — 432 с.

17. Таненбаум Э. Современные операционные системы. — М.: Питер, 2023. — 1120 с.

18. Dennis A., Wixom B., Tegarden D. Systems Analysis and Design: An Object-Oriented Approach with UML. — 6th ed. — Wiley, 2020. — 544 p.

19. Kurniawan B. Spring Boot: Modern Java Web Development. — Brainy Software, 2022. — 384 p.

20. Mockito Framework Documentation [Электронный ресурс]. URL: https://site.mockito.org/ (дата обращения: 14.05.2025).

21. Payment Card Industry Data Security Standard (PCI DSS) [Электронный ресурс]. URL: https://www.pcisecuritystandards.org (дата обращения: 16.05.2025).

22. PostgreSQL Documentation [Электронный ресурс]. URL: https://www.postgresql.org/docs/ (дата обращения: 16.05.2025).

23. Spring Framework Overview [Электронный ресурс]. URL: https://spring.io/projects/spring-framework (дата обращения: 15.05.2025).

24. Spring Security Reference Documentation [Электронный ресурс]. URL: https://docs.spring.io/spring-security/reference/index.html (дата обращения: 17.05.2025).

25. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. — Pearson, 2017. — 432 p.