МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСИС»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОМПЬЮТЕРНЫХ НАУК
КАФЕДРА АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ И ДИЗАЙНА
НАПРАВЛЕНИЕ 09.04.02 ИФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

на тему: «Разработка и оптимизация пользовательских и серверных подсистем E-commerce платформы на Next.js с целью повышения производительности и удобства»

Студент _____

Руководитель работы _____ *Е.Г. Коржов*

Нормоконтроль проведен _____ *А.С. Оганесян*

Проверка на заимствования проведена _____ *А.А. Петрыкина*

Работа рассмотрена кафедрой и допущена к защите в ГЭК

Заведующий кафедрой _____*Е.Г. Коржов*

Директор института _____ *С.В. Солодов*

Москва 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСИС»

УТВЕРЖДАЮ

Институт Информационных технологий и компьютерных наук
Кафедра Автоматизированного проектирования и дизайна _____ **Зав. кафедрой** _____ Е.Г. Коржов
Направление 09.04.02 Информационные системы и технологии _____ «_____»__ _____2025г.

ЗАДАНИЕ НА ВЫПОЛНЕНИЕ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ МАГИСТРА

Студенту группы _____
_____

1. Тема работы Разработка и оптимизация пользовательских и серверных подсистем E-commerce платформы на Next.js с целью повышения производительности и удобства

2. Цели работы исследование разработки и оптимизации клиентской и серверной части веб-приложения электронной коммерции с использованием фреймворка Next.js и смежных технологий, обеспечивающих высокую производительность, удобство сопровождения и гибкость архитектуры.

3. Задачи исследования

1) Изучить современные подходы к разработке и архитектурному проектированию E-commerce приложений, а также определить ключевые требования к их производительности и масштабируемости.

2) Проанализировать современные технологии и инструменты, применяемые для реализации клиентской и серверной логики веб-приложений, включая фреймворки, библиотеки и среды выполнения.

3) Рассмотреть принципы оптимизации клиентской и серверной части веб-приложений, включая подходы к рендерингу, кешированию и доставке данных.

4) Исследовать роль API и микросервисной архитектуры в проектировании современных интернет-магазинов, а также их влияние на масштабируемость и интеграцию с внешними сервисами.

5) Сформулировать функциональные требования к разрабатываемому веб-приложению и обосновать выбор используемого технологического стека.

6) Спроектировать архитектуру клиентской части приложения на основе современных принципов и структурной организации кода.

7) Реализовать пользовательский интерфейс с использованием фреймворка Next.js и библиотеки Tailwind CSS, обеспечив адаптивность и соответствие современным требованиям к UI.

8) Разработать серверную часть приложения с реализацией взаимодействия с базой данных и бизнес-логикой, соблюдая принципы модульности и безопасности.

9) Реализовать механизмы оптимизации рендеринга и загрузки данных, обеспечив баланс между производительностью и пользовательским опытом.

10) Реализовать процесс развертывания приложения с использованием облачной платформы Vercel, обеспечивающей автоматизацию сборки и деплоя.

11) Провести ручное и инструментальное тестирование приложения, включая тестирование API с помощью Postman.

4. Основная литература:

4.1 Дудар С. А. Веб-разработка в сфере торговли: инновационные подходы к созданию эффективных веб-приложений // Теория и практика современной науки. 2023. №5 (95). – С. 73-75.

4.2 Data Insight. Маркетинговое исследование Интернет-торговля в России 2025 [Электронный ресурс]. Режим доступа – https://datainsight.ru/DI_eCommerce_2025 (дата обращения: 23.05.2025).

4.3 Городничев М. Г., Полонский Р. В. Оценка возможности использования микросервисной архитектуры при разработке пользовательских интерфейсов клиент-серверного программного обеспечения // Экономика и качество систем связи. 2020. №3 (17). – С. 33-43.

4.4 Волнейкина Е. С., Гек Д К., Кукарцев В. В. Как фреймворк Vue js упрощает работу с дизайном при создании интерйфеса // актуальные проблемы авиации и космонавтики. 2021. – с. 481-482.

4.5 Ильин А. Ю., Плотников С. Б. Использование предиктивного анализа пользовательского поведения для повышения скорости реактивной загрузки клиентской части веб-приложения // Международный журнал гуманитарных и естественных наук. 2025. №1-3 (100). – С. 140-143.

4.6 Холодков Д. В., Зинкин С. А. Влияния технологий контейнеризации приложений на скорость выполнения // Вестник ПензГУ. 2024. №4 (48). – С. 134-136.

4.7 Фаррахов И. Г., Якупов И. М. Автоматизированный инструментарий развертывания облачных сервисов // Мировая наука. 2021. №2 (47). – С. 115-118.

4.8 Жиренкин А. В., Старосельский А. К. Развитие элементов архитектуры современных веб-приложений и их влияние на разработку // Вестник науки. 2023. №6 (63). – С. 869-872.

4.9 Сергачева М. А., Михалевская К. А. Анализ фреймворков для разработки современных веб-приложений // Кронос: естественные и технические науки. 2020. №2 (30). – С. 35-39.

4.10 Гордеев С. Ю., Тимофеев А. В., Козлов В. В. Разработка веб-приложений с использованием angular. Js, node. Js, MongoDB на примере системы психологической поддержки студентов - участников программы «полет» // Наука и образование сегодня. 2017. №2 (13). – С. 1-6.

Костенко И. П., Ступина М. В. Повышение производительности WEB-приложений средствами СУБД Redis // Молодой исследователь Дона. 2022. №4 (37). – С. 29-32.

4.11 Хомоненко А. Д., Абу Хасан Р. О надежности и доступности объектных хранилищ данных // Интеллектуальные технологии на транспорте. 2023. №S1 (35-1). – С. 123-128.

4.12 Захаров С. В., Котов А. А. Инфраструктура приложений на основе Docker и Kubernetes // Информационные технологии и вычислительные системы. 2021. № 3. – С. 45–52.

4.13 Документация Next.js [Электронный ресурс]. Режим доступа – https://nextjs.org/docs

4.15 Документация Drizzle [Электронный ресурс]. Режим доступа – https://orm.drizzle.team/docs/overview

5. Перечень основных этапов исследования:

Этап Анализ теоретических основ и подходов к разработке веб-приложений электронной коммерции

Этап Обзор и обоснование выбора технологий и инструментов разработки

Этап Формализация требований к системе и определение архитектурной структуры проекта

Этап Разработка пользовательского интерфейса и реализация ключевых функциональных компонентов

Этап Разработка серверной логики и взаимодействие с базой данных

Этап Оптимизация производительности приложения

Этап Развертывание приложения

Этап Тестирование веб-приложения и проверка корректности работы функциональных модулей

Этап Обобщение результатов и формулировка выводов

6. Перечень (примерный) иллюстрированного материала:

1) графические материалы, отражающие статистические данные, представленные в виде графиков, иллюстрирующих развитие рынка электронной коммерции и обоснование актуальности исследования;

2) фрагменты исходного программного кода, включающие ключевые участки реализации клиентской и серверной логики веб-приложения;

3) изображения (скриншоты) пользовательского интерфейса веб-приложения, демонстрирующие внешний вид и работу основных функциональных модулей системы.

7. Руководитель диссертации к.т.н., доцент, заведующий кафедрой автоматизированного проектирования и дизайна Коржов Евгений Геннадьевич

8. Подготовка выпускной квалификационной работы: до 17.06.2025

Дата выдачи задания 03.03.2025

Задание принял к исполнению студент _____

(подпись)

Аннотация

Выпускная квалификационная работа (ВКР) содержит: 76 страниц, 3 раздела, 14 подразделов, 19 рисунков, 42 литературных источников, 5 приложений.

Работа направлена на анализ и оценку архитектурных, технологических и инженерных решений, применяемых при разработке веб-приложений в сфере электронной коммерции, выявление оптимальных подходов к проектированию клиентской и серверной логики, а также исследование современных средств повышения производительности, масштабируемости и удобства пользовательского взаимодействия.

Выпускная квалификационная работа состоит из пояснительной записки, графического приложения и презентации.

Пояснительная записка содержит введение, три раздела и заключение.

Первый раздел посвящен теоретическому обзору предметной области: рассмотрены актуальные подходы к проектированию систем, проведен анализ применяемых технологий, архитектурных решений и принципов оптимизации.

Во втором разделе представлен процесс разработки веб-приложения на базе фреймворка Next.js, включая проектирование архитектуры, реализацию пользовательского интерфейса, серверной логики, взаимодействие с базой данных.

Третий раздел включает описание процессов развертывания и тестирования приложения, использование облачных платформ и инструментов автоматизации, ручного и инструментального тестирования.

В заключении подведены итоги проделанной работы, сформулированы выводы и обозначены перспективы дальнейшего развития программного продукта.

Annotation

The graduate qualification work comprises 76 pages, 3 chapters, 14 subsections, 19 figures, 42 references, and 5 appendices. The study is aimed at analyzing and evaluating architectural, technological, and engineering solutions applied in the development of web applications in the field of e-commerce. It also focuses on identifying optimal approaches to designing client- and server-side logic and exploring modern tools for improving performance, scalability, and user experience.

The graduate qualification work includes an explanatory report, graphical appendix, and presentation.

The explanatory report contains an introduction, three chapters, and a conclusion.

The first chapter is devoted to a theoretical review of the subject area. It considers current approaches to system design, analyzes applied technologies, architectural solutions, and principles of optimization.

The second chapter presents the process of developing a web application based on the Next.js framework, including architectural design, implementation of the user interface, server-side logic, and interaction with the database.

The third chapter describes the deployment and testing processes of the application, including the use of cloud platforms and automation tools, as well as both manual and tool-based testing.

The conclusion summarizes the work performed, formulates the main findings, and outlines the prospects for further development of the software product.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 7
1. Теоретические аспекты проектирования и оптимизации веб-приложений E-commerce 11
1.1 Современные подходы к разработке E-commerce платформ 11
1.2 Обзор технологий и инструментов для разработки веб-приложений ... 16
1.3 Принципы оптимизации клиентской и серверной части веб-приложений 29
1.4 Роль API и микросервисов в современной архитектуре интернет-магазинов 32
Выводы по разделу 1 34
2. Разработка веб-приложения на Next.js: архитектура, реализация и оптимизация 36
2.1 Функциональные требования к системе и используемый стек технологий 36
2.2 Проектирование архитектуры клиентской части платформы 38
2.3 Реализация пользовательского интерфейса с использованием Next.js и Tailwind CSS 47
2.4. Разработка серверной части и взаимодействие с базой данных 52
2.5. Механизмы оптимизации рендеринга и загрузки 57
Выводы по разделу 2 62
3. Развертывание и тестирование веб-приложения 64
3.1 Развертывание приложения на платформе Vercel 64
3.2 Тестирование приложения: ручное тестирование и Postman 66
Выводы по разделу 3 69
ЗАКЛЮЧЕНИЕ 70
СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 73
ПРИЛОЖЕНИЕ А 77
ПРИЛОЖЕНИЕ Б 78
ПРИЛОЖЕНИЕ В 79
ПРИЛОЖЕНИЕ Г 80
ПРИЛОЖЕНИЕ Д 81

ВВЕДЕНИЕ

Актуальность темы исследования. В современных общественных условиях наблюдается процесс изменений, который связан с явлением цифровизации. Рост электронной коммерции (E-commerce) в России и за рубежом за последние десятилетия свидетельствует о значительном влиянии этой отрасли на экономическое развитие и потребительское поведение.

Недавние события, такие как пандемия COVID-19, еще более усилили значение электронной коммерции, привлекая все больше внимания со стороны потребителей, предпринимателей и IT-комьюнити.

Современная цифровая экономика, основанная на стремительном развитии информационных технологий и глобализации, определяет ключевые направления трансформации рынка товаров и услуг. Одной из наиболее значимых и динамично развивающихся сфер и является электронная коммерция, которая выступает важным инструментом взаимодействия между потребителями и предприятиями. В условиях стремительного роста онлайн-торговли исследование актуальности и перспектив развития E-commerce становится важной задачей, как с научной, так и с практической точки зрения.

Электронная коммерция позволяет оптимизировать бизнес-процессы, снижать затраты на ведение деятельности и предоставлять клиентам удобный и быстрый доступ к широкому ассортименту товаров и услуг.

Эффективная разработка E-commerce платформ требует сочетания передовых технологий и инженерных подходов, обеспечивающих высокую производительность, безопасность и удобство использования. Особое внимание уделяется пользовательскому опыту (UX/UI), автоматизации процессов логистики и персонализации контента. Эти аспекты не только

определяют конкурентоспособность платформы, но и влияют на доверие и лояльность потребителей.

Одной из ключевых задач разработки является создание адаптивных веб-приложений, которые обеспечивают доступ к платформе с различных устройств, включая настольные компьютеры, планшеты и смартфоны.

Целью настоящей работы является исследование разработки и оптимизации клиентской и серверной части веб-приложения электронной коммерции с использованием фреймворка Next.js и смежных технологий, обеспечивающих высокую производительность, удобство сопровождения и гибкость архитектуры.

Для достижения поставленной цели были сформулированы **следующие задачи:**

1. Изучить современные подходы к разработке и архитектурному проектированию E-commerce приложений, а также определить ключевые требования к их производительности и масштабируемости.
2. Проанализировать современные технологии и инструменты, применяемые для реализации клиентской и серверной логики веб-приложений, включая фреймворки, библиотеки и среды выполнения.
3. Рассмотреть принципы оптимизации клиентской и серверной части веб-приложений, включая подходы к рендерингу, кешированию и доставке данных.
4. Исследовать роль API и микросервисной архитектуры в проектировании современных интернет-магазинов, а также их влияние на масштабируемость и интеграцию с внешними сервисами.
5. Сформулировать функциональные требования к разрабатываемому веб-приложению и обосновать выбор используемого технологического стека.
6. Спроектировать архитектуру клиентской части приложения на основе современных принципов и структурной организации кода.
7. Реализовать пользовательский интерфейс с использованием

фреймворка Next.js и библиотеки Tailwind CSS, обеспечив адаптивность и соответствие современным требованиям к UI.

8. Разработать серверную часть приложения с реализацией взаимодействия с базой данных и бизнес-логикой, соблюдая принципы модульности и безопасности.

9. Реализовать механизмы оптимизации рендеринга и загрузки данных, обеспечив баланс между производительностью и пользовательским опытом.

10. Реализовать процесс развертывания приложения с использованием облачной платформы Vercel, обеспечивающей автоматизацию сборки и деплоя.

11. Провести ручное и инструментальное тестирование приложения, включая тестирование API с помощью Postman.

Объектом исследования является веб-приложение электронной коммерции – интернет-магазин.

Предметом исследования выступают методы проектирования, реализации и оптимизации архитектуры и функциональных компонентов таких приложений с применением фреймворка Next.js и сопутствующих технологий.

Методы и инструменты, используемые в работе

В работе применялись практики современного программирования, включая использование архитектурных шаблонов (FSD) и следование принципам чистого кода (DRY, KISS, SOLID).

Для реализации использовался следующий стек технологий:

1) Next.js;
2) TypeScript;
3) Tailwind CSS;
4) Drizzle ORM;
5) Zustand и React Query;
6) Vercel.

Практическая значимость работы заключается в создании адаптивного веб-приложения, которое отвечает современным требованиям к пользовательскому опыту, безопасности и масштабируемости. Это решение может стать прототипом или готовым продуктом для малого и среднего бизнеса в сфере электронной коммерции. Оно также может послужить основой для дальнейшего развития более сложных корпоративных систем.

Практическая значимость также заключается в проверке целостного подхода к созданию производительных клиент-серверных систем с использованием современных технологий. Предложенная архитектура готова к применению в реальных условиях.

Структура работы. Цель и задачи исследования определили следующую структуру работы: Введение; Раздел 1 «Теоретические аспекты проектирования и оптимизации веб-приложений E-commerce»: 1.1 Современные подходы к разработке E-commerce платформ; 1.2 Обзор технологий и инструментов для разработки веб-приложений; 1.3 Принципы оптимизации клиентской и серверной части веб-приложений; 1.4 Роль API и микросервисов в современной архитектуре интернет-магазинов; Выводы по разделу 1; Раздел 2: «Разработка веб-приложения на Next.js: архитектура, реализация и оптимизация»: 2.1 Функциональные требования к системе и используемый стек технологий; 2.2 Проектирование архитектуры клиентской части платформы; 2.3 Реализация пользовательского интерфейса с использованием Next.js и Tailwind CSS; 2.4 Разработка серверной части и взаимодействие с базой данных; 2.5 Механизмы оптимизации рендеринга и загрузки; Выводы по разделу 2; Раздел 3: «Развертывание и тестирование веб-приложения»: 3.1 Развертывание приложения на платформе Vercel; 3.2 Тестирование приложения: ручное тестирование и Postman; Выводы по разделу 3; Заключение; Список используемой литературы; Приложения.

1. Теоретические аспекты проектирования и оптимизации веб-приложений E-commerce

1.1 Современные подходы к разработке E-commerce платформ

В современном цифровом поле электронная коммерция (e-commerce) представляет собой одну из наиболее динамично развивающихся отраслей информационных технологий, что влечет за собой высокие требования к качеству, масштабируемости и надежности веб-приложений [6].

Электронная коммерция обусловлена быстрым технологическим прогрессом, изменением потребительских предпочтений и повышением уровня интернет-проникновения. По данным исследования Data Insight (2025), рынок электронной коммерции в России уже превысил 11,2 триллионов рублей и 6,8 миллиардов заказов и продолжает расти, что видно на рисунке 1[16].

Рисунок 1 – Динамика числа заказов 2013-2024

В условиях высокой конкуренции особое значение приобретают платформенные решения, которые обеспечивают гибкость в настройке,

масштабируемость и возможность интеграции современных сервисов, таких как системы персонализированных рекомендаций и аналитики.

Проектирование и оптимизация e-commerce платформ требует системного подхода, основанного на теоретических принципах программной инженерии, архитектурных шаблонах проектирования и современных подходов разработки, включая компонентно-ориентированное программирование, микросервисную архитектуру, DevOps-практики и принципы устойчивой масштабируемости [5].

Особое внимание уделяется обеспечению высокой производительности и отказоустойчивости систем при работе с большим объемом пользовательских запросов, сложной бизнес-логикой и необходимостью интеграции с внешними сервисами (платежными системами, провайдерами, CRM-системами и т.д.).

Проектирование эффективных E-commerce платформ предполагает глубокий анализ пользовательского поведения, моделирование бизнес-процессов и оптимизацию взаимодействия с базами данных. Системы электронной коммерции работают с высоко динамичным контентом, каталогами товаров, персонализированными рекомендациями и сложными системами фильтрации, что требует применения эффективных стратегий хранения и извлечения данных. В этой связи особую роль играет выбор архитектуры хранилища: от традиционных реляционных СУБД (например, PostgreSQL, MySQL) до гибридных решений с использованием NoSQL-хранилищ (MongoDB, Redis, Elasticsearch) для обеспечения высокой скорости доступа к часто изменяемым и масштабируемым данным.

Современные подходы к разработке E-commerce приложений включают в себя активное использование headless-архитектур, при которых фронтенд и бэкенд функционируют как слабо связанные компоненты, взаимодействующие через API-интерфейсы (чаще всего REST или GraphQL), что существенно повышает гибкость и ускоряет внедрение изменений. Также

наблюдается устойчивый тренд к применению JAMstack-решений, при которых клиентская часть предварительно генерируется и кэшируется на CDN, что значительно снижает задержки и улучшает пользовательский опыт. Параллельно с этим развивается практика использования PWA (Progressive Web Apps) – прогрессивных веб-приложений, обеспечивающих нативоподобный интерфейс, оффлайн-доступ и высокую отзывчивость.

При разработке современных E-commerce систем также важно учитывать вопросы безопасности и защиты пользовательских данных. В условиях строгих нормативных требований, таких как GDPR (General Data Protection Regulation), PCI DSS (Payment Card Industry Data Security Standard) и локальных законов о защите персональных данных, проектировщик обязан внедрять многоуровневые механизмы аутентификации и авторизации, обеспечивать шифрование данных при передаче и хранении, а также реализовывать системы мониторинга и реагирования на инциденты информационной безопасности.

Еще одним значимым аспектом оптимизации E-commerce платформ является обеспечение высокой производительности интерфейса пользователя. Это достигается за счет использования прогрессивных фронтенд-фреймворков (React, Vue.js, Next.js) [1], серверного рендеринга (SSR, SSG, ISR), динамической подгрузки компонентов (lazy loading) и агрессивной оптимизации ресурсов (минификация, сжатие, использование WebP-форматов изображений, внедрение Service Workers и т.д.) [9].

Следует также выделить значимость интеграционных возможностей, которые делают E-commerce платформу приоритетной среди конкурирующих развитых платформ. Интеграции с внешними API – маркетплейсами, платформами аналитики, рекламными кабинетами, партнерскими программами и службами доставки – требуют проектирования устойчивой API-инфраструктуры с соблюдением стандартов версионирования, документирования (например, OpenAPI/Swagger) и

управления нагрузкой. Здесь важную роль играет использование посреднических слоев – API-шлюзов (API Gateway), брокеров сообщений (Kafka, RabbitMQ) для асинхронной обработки событий и масштабирования взаимодействий между микросервисами.

Современные подходы к разработке и развертыванию программного обеспечения неввозможны без применения автоматизированных процессов непрерывной интеграции (Continuous Integration, CI) и непрерывной поставки (Continuous Delivery/Deployment, CD), которые обеспечивают сокращение времени выхода продукта на рынок, а также повышение его стабильности.

CI/CD-практики позволяют внедрять изменения в кодовую базу с минимальными затратами на верификацию и контроль качества. Каждое изменение, внесенное в основной репозиторий, автоматически инициирует цепочку сборки, тестирования и, при успешном прохождении всех этапов, развертывания новой версии приложения в продуктивной среде. Это исключает «человеческий фактор» и сводит к минимуму риск появления ошибок при публикации обновлений.

Контейнеризация с использованием технологии Docker обеспечивает изоляцию среды выполнения приложения, включая его зависимости, конфигурационные параметры и системные библиотеки, что гарантирует воспроизводимость результатов на всех этапах разработки, тестирования и продакшн-развертывания. Образ контейнера представляет собой самодостаточную, неизменяемую единицу поставки, что позволяет избежать проблем, связанных с различиями в окружении между локальной и серверной инстанциями [14].

Оркестрация контейнеров средствами Kubernetes позволяет масштабировать приложение горизонтально, управлять отказоустойчивостью, автоматически перераспределять нагрузку между узлами кластера и выполнять обновления с нулевым временем простоя.

Kubernetes предоставляет богатый инструментарий для описания инфраструктуры как кода, контроля над ресурсами, автоматического восстановления контейнеров при сбоях, а также стратегий развертывания, включая blue-green deployment и rolling updates. Это делает архитектуру приложения гибкой, масштабируемой и устойчивой к сбоям в условиях роста нагрузки или изменения внешней среды [13].

Надежность и корректность бизнес-логики обеспечивается путем внедрения многоуровневой системы автоматизированного тестирования, охватывающей ключевые аспекты функционирования приложения. На уровне модулей применяются unit-тесты, позволяющие изолированно проверять поведение отдельных функций и компонентов.

Интеграционные тесты предназначены для валидации взаимодействия между модулями и подсистемами, включая обращения к базе данных и сторонним API. End-to-end тесты, в свою очередь, симулируют реальные сценарии пользовательского взаимодействия, позволяя удостовериться в работоспособности бизнес-процессов в условиях, максимально приближенных к реальной эксплуатации пользователями.

Современные исследования подтверждают эффективность контейнеризации и оркестрации в повышении надежности и масштабируемости программных систем. В частности, подчеркивается роль Docker и Kubernetes в построении отказоустойчивых и гибких DevOps-процессов, а также в обеспечении согласованности окружений между стадиями CI/CD [8].

1.2 Обзор технологий и инструментов для разработки веб-приложений

Современная разработка веб-приложений представляет собой комплексный подход, требующий использования широкого спектра технологий, инструментов и подходов.

Сложность и масштабируемость современных веб-систем обуславливают необходимость выбора таких технологических решений, которые не только обеспечивают высокую производительность и безопасность, но и способствуют упрощению процессов разработки, тестирования, развертывания и поддержки программного продукта.

Технологический стек, применяемый при создании веб-приложений, охватывает как клиентскую (frontend), так и серверную (backend) части, включая базу данных, протоколы взаимодействия, системы управления состоянием, фреймворки, инструменты сборки и тестирования. Каждое из этих звеньев оказывает прямое влияние на итоговое качество программной системы.

В последние годы значительное развитие получили фреймворки и библиотеки, реализующие подходы компонентно-ориентированного программирования и декларативного описания интерфейсов. Появление таких инструментов, как Next.js, React, TypeScript, Drizzle ORM и Tailwind CSS, позволило разработчикам существенно повысить качество разрабатываемых решений.

В рамках данного исследования проведен анализ технологий и инструментов, используемых для реализации frontend и backend компонентов веб-приложений, а также применяемых баз данных.

Современные веб-приложения состоят из двух ключевых компонентов:

1. Клиентской части (frontend), обеспечивающей взаимодействие с пользователем.

2. Серверной части (backend), отвечающей за обработку данных, бизнес-логику и интеграцию с внешними сервисами [7].

Клиентская часть веб-приложений отвечает за пользовательский интерфейс и взаимодействие с конечным пользователем. Основным языком для создания интерактивных интерфейсов является JavaScript, который дополняется библиотеками и фреймворками для упрощения разработки сложных приложений. Среди наиболее распространенных инструментов для frontend-разработки выделяются следующие:

1. React: библиотека JavaScript, предназначеная для создания компонентно-ориентированных пользовательских интерфейсов. Благодаря своей компонентной архитектуре и виртуальному DOM, React обеспечивает высокую производительность при построении сложных интерактивных приложений, поддерживает декларативный подход к разработке, что упрощает управление состоянием и обновление интерфейса. Однако, являясь исключительно библиотекой для представления, React требует дополнительных решений для реализации маршрутизации, серверного рендеринга и взаимодействия с сервером [37].

2. Next.js: фреймворк на базе React, разработанный с целью устранения ограничений классических SPA-приложений. Он поддерживает серверный рендеринг (SSR), статическую генерацию (SSG), гибридные методы рендеринга и маршрутизацию на основе файловой структуры. Одним из его значительных преимуществ является возможность унифицированной разработки как клиентской, так и серверной логики в рамках единого проекта, без необходимости использования отдельных backend-фреймворков. Это достигается благодаря встроенной поддержке API-роутов и middleware, что существенно упрощает архитектуру и ускоряет разработку. Кроме того, Next.js обеспечивает высокую степень оптимизации производительности,

включая автоматическое разделение кода, адаптивную загрузку и поддержку edge-функций [34].

3. Angular: полноценный фреймворк, разработанный Google, основанный на TypeScript. Фреймворк предлагает двухстороннюю привязку данных, встроенные инструменты для управления зависимостями и модульную архитектуру. Angular предлагает все необходимое из коробки: систему модулей, инъекцию зависимостей, строгую типизацию, реактивные формы и собственный механизм шаблонов, а также подходит для разработки крупных корпоративных систем, однако его сложность, объем шаблонного кода и высокая входная планка делают его менее подходящим для гибких, быстроразвивающихся продуктов [17].

4. Vue.js: прогрессивный JavaScript-фреймворк, отличающийся простотой интеграции и гибкостью. Vue предоставляет встроенные инструменты для двухсторонней привязки данных, управления состоянием и маршрутизации, что делает его популярным среди малых команд и стартапов. Тем не менее, при масштабировании проекта могут возникать трудности, связанные с архитектурной изоляцией и тестированием. Фреймворк поддерживает реактивную модель данных и позволяет разрабатывать как небольшие компоненты, так и сложные одностраничные приложения (SPA) [42].

5. Svelte: современный фреймворк, который компилирует код в чистый JavaScript на этапе сборки, минимизируя объем клиентского кода и повышая производительность [41].

6. Python-библиотеки: хотя Python традиционно используется для серверной разработки, такие инструменты, как PyScript, позволяют интегрировать Python в браузер для создания интерактивных интерфейсов. Кроме того, фреймворки, такие как Dash (для визуализации данных) и Streamlit, применяются для разработки веб-интерфейсов с минимальными усилиями по написанию кода [36].

Дополнительно для стилизации интерфейсов используются технологии CSS и препроцессоры (Sass, Less), а также фреймворки, такие как Tailwind CSS и Bootstrap, которые обеспечивают адаптивный дизайн и упрощают верстку.

Серверная часть веб-приложений отвечает за обработку запросов, управление бизнес-логикой, интеграцию с базами данных и обеспечение безопасности. Для реализации backend применяются следующие различные языки программирования и их фреймворки:

1. JavaScript: изначально разработан как клиентский язык. Стал универсальным благодаря среде выполнения Node.js, которая позволяет использовать его для серверной разработки. TypeScript, надмножество JavaScript, добавляет строгую типизацию, что улучшает масштабируемость и поддержку кода в крупных проектах [11]. Для серверной разработки применяются следующие фреймворки:

а) Next.js: мощный фреймворк на основе JavaScript/TypeScript и React, который, хотя и изначально ассоциируется с клиентской разработкой, широко используется для серверной части благодаря поддержке серверного рендеринга (SSR), статической генерации (SSG) и API маршрутов. В контексте E-commerce Next.js позволяет разрабатывать серверные API-эндпоинты, обрабатывающие запросы для таких задач, как получение данных о товарах, управление корзиной или обработка платежей, без необходимости отдельного backend-фреймворка. Next.js предоставляет встроенные API Routes, которые позволяют создавать RESTful-эндпоинты непосредственно в рамках проекта, используя Node.js как среду выполнения. Фреймворк поддерживает интеграцию с базами данных через ORM (например, Drizzle, Prisma, TypeORM) или прямые запросы, а также взаимодействие с внешними сервисами, такими как платежные системы или системы управления контентом. Next.js также поддерживает асинхронное программирование и middleware для обработки запросов, что делает его подходящим для

реализации серверной логики в небольших и средних E-commerce проектах. Благодаря поддержке Vercel и оптимизации для SEO, Next.js особенно полезен для создания гибридных приложений, где часть функциональности выполняется на сервере, а часть – на клиенте [34].

b) Express.js: минималистичный и гибкий фреймворк для Node.js, который предоставляет базовые инструменты для создания веб-приложений и API. Express поддерживает маршрутизацию, middleware для обработки запросов, а также интеграцию с шаблонизаторами (например, EJS или Pug) для серверного рендеринга. Фреймворк широко используется для создания RESTful API в E-commerce системах благодаря своей простоте и большому количеству сторонних библиотек. Express легко интегрируется с базами данных, такими как MongoDB (через Mongoose) или PostgreSQL (через Sequelize). Однако из-за своей минималистичности Express требует от разработчиков самостоятельной настройки безопасности и масштабируемости [24].

c) NestJS: прогрессивный фреймворк, построенный на TypeScript и использующий архитектурные принципы Angular, такие как модульность и инверсия управления. NestJS поддерживает как Express, так и Fastify в качестве HTTP-платформ, что обеспечивает гибкость и производительность. Фреймворк предоставляет встроенные инструменты для создания RESTful и GraphQL API, обработки WebSocket-соединений и интеграции с ORM (TypeORM, Sequelize). NestJS подходит для крупных E-commerce приложений благодаря строгой типизации, поддержке микросервисов и возможностям тестирования [33].

2. Python: высокоуровневый интерпретируемый язык программирования, известный своей читаемостью, лаконичным синтаксисом и обширной экосистемой библиотек. Его универсальность и простота делают Python популярным выбором для разработки веб-приложений, включая E-commerce платформы. Python поддерживает парадигмы объектно-

ориентированного, функционального и процедурного программирования, что позволяет адаптировать его под различные архитектурные подходы [36].

Для веб-разработки применяются следующие фреймворки Python:

a) Django: полнофункциональный фреймворк с открытым исходным кодом, который следует принципу «батарейки включены» (batteries-included), предоставляя разработчикам широкий набор встроенных инструментов. Django упрощает взаимодействие с реляционными базами данных, такими как PostgreSQL или MySQL, за счет абстрагирования SQL-запросов в Python-код. Фреймворк предлагает встроенные механизмы аутентификации и авторизации, включая управление пользователями, ролями и сессиями. Также Django предоставляет панель администрирования, автоматически генерируемую на основе моделей данных, что ускоряет разработку административных интерфейсов для управления контентом E-commerce платформ (например, каталогами товаров или заказами). Django поддерживает шаблонизатор для рендеринга HTML-страниц, а также интеграцию с REST API через библиотеку Django REST Framework. Благодаря строгой структуре и встроенным средствам безопасности (защита от XSS, CSRF, SQL-инъекций), Django подходит для создания сложных и масштабируемых веб-приложений [20].

b) FastAPI: современный асинхронный фреймворк, разработанный для создания высокопроизводительных API. Он основан на стандарте Python 3.6+ с использованием аннотаций типов и асинхронного программирования (asyncio). FastAPI использует библиотеку Starlette для обработки HTTP-запросов и Pydantic для валидации данных, что обеспечивает строгую типизацию и автоматическую генерацию документации API в формате OpenAPI (Swagger). Это делает FastAPI особенно подходящим для разработки RESTful и GraphQL API в E-commerce системах, где требуется быстрая обработка большого числа запросов, например, для поиска товаров или обработки заказов. Высокая производительность FastAPI достигается за

счет асинхронной обработки, что позволяет эффективно управлять параллельными соединениями. Фреймворк также поддерживает интеграцию с различными базами данных через ORM (например, SQLAlchemy) или прямые запросы [25].

c) Flask: легковесный и гибкий микрофреймворк, который предоставляет минимальный набор инструментов для веб-разработки, позволяя разработчикам самостоятельно выбирать дополнительные библиотеки и архитектурные решения. Flask не навязывает строгую структуру проекта, что делает его идеальным для небольших приложений, прототипов или стартапов в сфере E-commerce. Фреймворк поддерживает маршрутизацию запросов, шаблонизатор Jinja2 для рендеринга страниц и обработку HTTP-запросов. Для реализации API часто используется расширение Flask-RESTful. Flask легко интегрируется с базами данных через библиотеки, такие как SQLAlchemy или Flask-SQLAlchemy. Несмотря на свою простоту, Flask требует от разработчиков большего внимания к настройке безопасности и масштабируемости по сравнению с Django [26].

3. C#: мощный объектно-ориентированный язык программирования, разработанный Microsoft, который широко применяется в корпоративных системах благодаря своей строгой типизации, производительности и интеграции с экосистемой .NET. C# поддерживает разработку надежных и масштабируемых приложений, что делает его популярным выбором для крупных E-commerce платформ [19].

a) ASP.NET Core: кроссплатформенный фреймворк C# с открытым исходным кодом, предназначенный для создания современных веб-приложений и API. Он поддерживает как монолитные, так и микросервисные архитектуры, что делает его подходящим для E-commerce систем с высокой нагрузкой. ASP.NET Core включает встроенные механизмы маршрутизации, middleware для обработки запросов, а также поддержку dependency injection для управления зависимостями. Фреймворк предоставляет Entity Framework

Core – ORM для работы с реляционными базами данных, упрощающий выполнение сложных запросов. ASP.NET Core поддерживает серверный рендеринг (MVC, Razor Pages), а также разработку RESTful и GraphQL API. Благодаря интеграции с Azure и поддержке контейнеризации (Docker), ASP.NET Core обеспечивает высокую масштабируемость и отказоустойчивость. Фреймворк также включает инструменты для обеспечения безопасности, такие как защита от CSRF и управление аутентификацией через OAuth или OpenID Connect [18].

4. Go (Golang): язык программирования, разработанный Google, известный своей простотой, высокой производительностью и эффективной работой с конкурентными задачами. Go использует модель конкурентности на основе горутин и каналов, что делает его подходящим для высоконагруженных серверных приложений. Его компилируемая природа обеспечивает быстрое выполнение программ, что важно для E-commerce систем с большим числом транзакций [28].

a) Gin: легковесный фреймворк Go для создания RESTful API, отличающийся высокой производительностью и минималистичным дизайном. Gin предоставляет маршрутизацию запросов, middleware для обработки CORS, аутентификации и логирования, а также встроенные инструменты для валидации данных. Фреймворк оптимизирован для обработки большого числа HTTP-запросов, что делает его подходящим для реализации API в E-commerce, например, для обработки корзин, заказов или каталогов. Gin легко интегрируется с базами данных через сторонние библиотеки, такие как GORM (ORM для реляционных баз) или прямые драйверы для MongoDB [27].

b) Echo: высокопроизводительный фреймворк Go с минималистичным синтаксисом, предназначенный для создания API и веб-приложений. Echo поддерживает маршрутизацию, middleware, валидацию данных и рендеринг шаблонов. Фреймворк оптимизирован для скорости и масштабируемости, что

делает его подходящим для высоконагруженных систем. Echo предоставляет гибкость в выборе библиотек для работы с базами данных и другими сервисами, что позволяет разработчикам адаптировать его под конкретные требования E-commerce приложений [22].

5. Java: язык программирования, широко применяемый в корпоративных системах благодаря своей надежности, масштабируемости и обширной экосистеме. Java поддерживает объектно-ориентированное программирование и предоставляет мощные инструменты для разработки сложных E-commerce платформ [30]:

a) Spring Boot: фреймворк, упрощающий разработку приложений на Java за счет автоматической конфигурации и минимального количества boilerplate-кода. Spring Boot поддерживает создание RESTful API, микросервисов и монолитных приложений. Фреймворк включает Spring Data для работы с реляционными и NoSQL базами данных, Spring Security для реализации аутентификации и авторизации, а также Spring Cloud для управления распределенными системами. Spring Boot широко используется в E-commerce для реализации масштабируемых систем с высокой отказоустойчивостью [40].

b) Hibernate: библиотека ORM, которая упрощает взаимодействие с реляционными базами данных за счет отображения объектной модели на реляционную. Hibernate поддерживает сложные запросы, ленивую загрузку данных и кэширование, что повышает производительность. В E-commerce Hibernate используется для управления сложными данными, такими как каталоги товаров, заказы и профили пользователей [29].

6. Ruby: динамический язык программирования, известный своей выразительностью и ориентацией на удобство разработчика. Ruby популярен в веб-разработке благодаря фреймворку Ruby on Rails [38].

a) Ruby on Rails (Rails): полнофункциональный фреймворк, который следует принципу «конвенция важнее конфигурации» и «не повторяйся»

(DRY). Rails предоставляет встроенные инструменты для маршрутизации, ORM (Active Record), миграций базы данных, аутентификации и тестирования. Фреймворк поддерживает серверный рендеринг и создание RESTful API, что делает его подходящим для быстрой разработки E-commerce приложений. Rails широко используется в стартапах благодаря скорости прототипирования и богатой экосистеме gem-библиотек [39].

В дополнение к этому, важную роль в разработке веб-приложений играют системы управления базами данных (далее – СУБД), обеспечивающие хранение, обработку и извлечение информации.

Хранение и обработка данных являются критически важными для E-commerce систем, где требуется управление каталогами товаров, заказами, пользователями и транзакциями. Используются следующие типы СУБД:

1. Реляционные базы данных – это тип баз данных, в которых данные организованы в виде таблиц, состоящих из строк и столбцов, и между этими таблицами устанавливаются связи на основе ключей. Классификация реляционных баз данных включает в себя следующие основные типы:

a) PostgreSQL: мощная СУБД с открытым исходным кодом, поддерживающая сложные запросы, JSONB для хранения неструктурированных данных и расширения для геопространственных данных [35].

b) MySQL: популярная реляционная СУБД, известная своей производительностью и простотой интеграции [32].

c) MariaDB: форк MySQL, обеспечивающий совместимость и дополнительные оптимизации [31].

2. Нереляционные (NoSQL) базы данных – это класс систем управления базами данных, отличающихся от традиционных реляционных моделей тем, что данные в них организованы не в виде таблиц с фиксированной схемой, а в разнообразных форматах, таких как документы, ключ-значение, графы или колоночные хранилища. Такие базы данных ориентированы на

горизонтальное масштабирование, гибкость в моделировании данных и высокую производительность при работе с большими объемами разнородных и неструктурированных данных. Классификация нереляционных баз данных включает в себя следующие основные типы:

a) MongoDB: документоориентированная СУБД, подходящая для хранения гибких и неструктурированных данных, таких как каталоги товаров [4].

b) Redis: высокоскоростная база данных типа «ключ-значение», используемая для кэширования и обработки сессий [10].

c) Cassandra: распределенная СУБД, оптимизированная для работы с большими объемами данных и высокой доступности [15].

3. NewSQL представляет собой класс современных систем управления базами данных, которые сохраняют свойства традиционных реляционных баз данных (например, поддержку SQL, ACID-транзакции), но при этом обеспечивают улучшенную масштабируемость и производительность, характерные для NoSQL-систем. Таким образом, NewSQL объединяет преимущества реляционных моделей с возможностями масштабирования, характерными для распределенных и облачных сред. Классификация NewSQL включает в себя следующие основные типы:

a) CockroachDB: распределенная реляционная СУБД, обеспечивающая масштабируемость и отказоустойчивость.

b) Elasticsearch: поисковая система, используемая для реализации полнотекстового поиска и аналитики в E-commerce [23].

Современная разработка веб-приложений опирается на обширный спектр технологий и инструментов, каждый из которых ориентирован на решение специфических задач: построение пользовательского интерфейса, реализация серверной логики или организация хранения и обработки данных.

С точки зрения управления данными, PostgreSQL и MySQL остаются стандартами реляционного хранения. PostgreSQL предоставляет

расширенные возможности по работе со сложными структурами данных, транзакциями, индексами и пользовательскими типами. MySQL предлагает более простую модель и высокую производительность в типичных веб-нагрузках, однако ограничен в функциональности по сравнению с PostgreSQL. MongoDB, как документо-ориентированная СУБД, предоставляет гибкость в определении схем, горизонтальное масштабирование и быструю сериализацию данных в формате JSON, что делает ее удобной для высокодинамичных приложений. Однако отсутствие жестких схем может затруднять поддержку целостности данных в сложных бизнес-сценариях.

Современные подходы к взаимодействию с базами данных все чаще предполагают использование легких ORM-библиотек, минимизирующих прослойку между кодом и базой.

Одной из таких библиотек является Drizzle ORM, выделяющаяся полной типобезопасностью, реализуемой уже на стадии компиляции TypeScript-кода, что существенно снижает вероятность возникновения логических и структурных ошибок при работе с базой данных.

В отличие от традиционных решений, таких как Prisma или TypeORM, Drizzle не требует генерации промежуточного кода или метаданных, обеспечивая разработчику прямой и декларативный способ описания SQL-запросов, максимально приближенный к нативному синтаксису SQL. Такое проектное решение обеспечивает полную прозрачность бизнес-логики и контроль над слоем доступа к данным, исключая избыточную абстракцию и тем самым минимизируя накладные расходы как на этапе разработки, так и в процессе исполнения.

Особую ценность Drizzle представляет для TypeScript-ориентированных проектов, в которых приоритет отдается строгой типизации, предсказуемости структуры данных и статическому анализу кода. Благодаря тесной интеграции с системой типов TypeScript, Drizzle позволяет

разрабатывать согласованную, безопасную и легко масштабируемую схему данных, а также упрощает миграции и эволюцию модели данных в условиях активной фиче-разработки.

В связке с Drizzle все большее распространение получает современное решение для распределенного хранения данных – Turso, построенный на основе высокопроизводительной библиотеки libSQL, являющейся форком SQLite с расширенными возможностями. В отличие от централизованных архитектур традиционных реляционных СУБД (таких как PostgreSQL или MySQL), Turso ориентирован на edge-инфраструктуру, что позволяет географически распределять реплики данных вблизи конечных пользователей. Подобный подход обеспечивает минимизацию сетевых задержек, повышает скорость отклика приложения и улучшает общее восприятие производительности с точки зрения пользователя. Более того, архитектура Turso поддерживает оффлайн-доступ, локальные записи и последующую синхронизацию, что делает его особенно актуальным в сценариях с переменной доступностью сети и для построения отказоустойчивых геораспределенных систем.

Интеграция типобезопасной ORM-системы, обладающей декларативным и предсказуемым синтаксисом, с edge-ориентированной базой данных, адаптированной под открытые сценарии доступа. Она представляет собой современный архитектурный паттерн, обеспечивающий высокую адаптивность, отказоустойчивость и масштабируемость серверной инфраструктуры. Это позволяет разрабатывать приложения, обладающие как высокой внутренней согласованностью архитектуры, так и способностью к эффективной работе в условиях высокой географической дисперсии пользовательского трафика [21].

1.3 Принципы оптимизации клиентской и серверной части веб-приложений

Оптимизация клиентской и серверной части веб-приложений представляет собой комплексный процесс, включающий разнообразные инженерные, архитектурные и организационные методы, направленные на обеспечение высокой производительности, надежности, масштабируемости и удобства использования системы. В основе оптимизации лежит ключевая цель – снижение задержки взаимодействия, сокращение объема передаваемых данных и рациональное использование вычислительных ресурсов. Эти меры способствуют улучшению отклика приложения и повышению удовлетворенности пользователей.

Современные подходы к оптимизации базируются на принципах разделения ответственности, модульности и повторного использования компонентов, а также активном применении прогрессивных технологий, инструментов мониторинга и анализа производительности. Такой подход позволяет непрерывно совершенствовать качество веб-приложений и адаптировать их к быстро меняющимся требованиям цифровой среды.

Оптимизация становится неотъемлемой частью жизненного цикла разработки современных веб-приложений, обеспечивая их конкурентоспособность и стабильное функционирование в условиях динамичного роста пользовательской базы и усложнения бизнес-логики.

На стороне клиента (frontend) ключевыми задачами оптимизации являются:

1) снижение времени первой отрисовки (First Paint);
2) уменьшение общего времени загрузки;
3) обеспечение плавности интерфейсных переходов и отзывчивости.

Это достигается посредством использования:

1) техники ленивой загрузки модулей и ресурсов (lazy loading);

2) предварительной выборки критических данных (preloading, prefetching);

3) минимизации количества и объема HTTP-запросов;

4) внедрения современных форматов графики (например, WebP или AVIF);

5) агрессивного кэширования;

6) применения механизмов виртуализации DOM-дерева;

7) расщепления кода (code splitting);

8) сжатие и минификация скриптов и таблиц стилей;

9) асинхронная подгрузка компонентов;

10) оптимизация обработки событий (event delegation, throttling, debouncing) [12].

Серверная (backend) оптимизация охватывает широкий спектр задач, связанных как с внутренней логикой обработки данных, так и с инфраструктурной организацией всей системы. Одним из ключевых направлений является снижение задержки при выполнении бизнес-операций.

Для этого применяются:

1) асинхронные модели обработки запросов;

2) использование пулов соединений с базами данных;

3) эффективное индексирование и оптимизация SQL-запросов;

4) проектирование гибких и масштабируемых схем хранения данных.

Важную роль играют распределенные механизмы кэширования, такие как Redis или Memcached, которые позволяют минимизировать обращение к медленным слоям хранения данных.

Кроме того, для обеспечения масштабируемости и высокой доступности системы реализуются методы горизонтального масштабирования, включая контейнеризацию и оркестрацию с использованием Docker и Kubernetes, а также балансировка нагрузки [3].

Для оптимизации доставки статического контента применяется CDN (Content Delivery Network).

На уровне архитектуры особое внимание уделяется декомпозиции монолитных приложений в микросервисную структуру или внедрению серверлесс-подходов, что позволяет более эффективно использовать вычислительные ресурсы.

Для своевременного выявления узких мест и предотвращения деградации производительности применяются инструменты мониторинга и профилирования, такие как Prometheus, Grafana и системы APM.

В совокупности, перечисленные методы формируют основу для создания высокоэффективных, надежных и масштабируемых веб-приложений, способных стабильно функционировать при высокой пользовательской нагрузке и динамично изменяющихся условиях эксплуатации.

Важным аспектом оптимизации как клиентской, так и серверной части является обеспечение безопасности без ущерба для производительности.

Механизмы защиты от типовых уязвимостей, таких как XSS, CSRF и SQL-инъекции, должны быть реализованы так, чтобы не замедлять обработку запросов или загрузку страниц. Использование заголовков безопасности (Content Security Policy, HTTP Strict Transport Security), контроль целостности ресурсов (Subresource Integrity), а также внедрение политики ограниченного доступа на уровне API (RBAC, ABAC) позволяют поддерживать высокий уровень защиты при сохранении скорости отклика системы.

Не менее значима адаптация приложения к различным условиям эксплуатации, включая мобильные устройства и сети с низкой пропускной способностью. Здесь ключевую роль играют принципы адаптивной верстки, прогрессивного улучшения (progressive enhancement) и внедрение технологий Progressive Web Apps (PWA), обеспечивающих автономную работу, быструю загрузку и мгновенный отклик даже при нестабильном интернет-соединении.

Особое внимание в современной веб-разработке уделяется метрикам пользовательского опыта, таким как Core Web Vitals, рекомендованным Google. В число этих метрик входят LCP (Largest Contentful Paint), FID (First Input Delay) и CLS (Cumulative Layout Shift), которые позволяют объективно оценивать качество восприятия производительности веб-приложения конечным пользователем и служат важным ориентиром для последующей оптимизации.

1.4 Роль API и микросервисов в современной архитектуре интернет-магазинов

Современные интернет-магазины строятся на основе API и микросервисной архитектуры – такой подход упрощает масштабирование, ускоряет разработку и позволяет гибко управлять функциональностью отдельных компонентов системы.

API выступают в роли набора доступных интерфейсов, с помощью которых можно взаимодействовать с частью программы или внешним сервисом, не вникая в его внутреннюю логику. Они позволяют реализовать раздельную ответственность, ускорить интеграцию и обеспечить совместимость между сервисами. Особенно широко применяются REST и GraphQL API, которые служат универсальными механизмами передачи данных между клиентской частью (например, веб-интерфейсом или мобильным приложением) и серверной, а также между отдельными сервисами внутри инфраструктуры интернет-магазина.

Через API обеспечивается интеграция с платежными системами, CRM и ERP-решениями, службами доставки, аналитическими платформами и другими внешними сервисами. Это позволяет интернет-магазину

функционировать как часть более широкой цифровой экосистемы, обеспечивая обмен данными и бизнес-логикой с внешними партнерами.

Микросервисная архитектура предполагает разделение монолитного приложения на независимые, изолированные модули, каждый из которых отвечает за строго определенную бизнес-функцию – управление каталогом товаров, обработку заказов, авторизацию пользователей, аналитические задачи и прочее. Такой подход позволяет разрабатывать, разворачивать и масштабировать каждый микросервис автономно, что существенно повышает устойчивость системы к ошибкам, снижает операционные риски и облегчает сопровождение программного кода.

Кроме того, микросервисная архитектура способствует внедрению современных практик непрерывной интеграции и доставки (CI/CD), а также культуры DevOps. Это ускоряет релизные циклы и позволяет оперативно вносить изменения без необходимости модификации всего приложения. Разнообразие используемых технологий и языков программирования в разных микросервисах обеспечивает технологическую гибкость и возможность поэтапного внедрения инноваций без кардинальной перестройки системы.

Использование API и микросервисной архитектуры существенно упрощает реализацию многоканального взаимодействия с пользователями, при этом каждая технология вносит свой вклад.

API обеспечивает единый и стандартизированный способ обмена данными между различными клиентскими интерфейсами – веб-приложениями, мобильными приложениями, чат-ботами, терминалами самообслуживания и другими точками контакта.

Микросервисы, в свою очередь, позволяют гибко разрабатывать и масштабировать отдельные функциональные модули, такие как управление контентом, ценами, остатками на складе или пользовательскими сессиями.

В совокупности это обеспечивает централизованное управление бизнес-логикой и данными, а также формирует целостный пользовательский опыт вне зависимости от используемого канала доступа.

Особое внимание уделяется обеспечению отказоустойчивости. Микросервисная архитектура позволяет внедрять шаблоны устойчивости к сбоям, такие как circuit breaker, fallback-механизмы и репликацию сервисов. Это критически важно для интернет-магазинов, где даже кратковременный простой или ошибка в ключевых компонентах, например, в модуле оплаты или оформления заказов, могут привести к значительным финансовым потерям и ухудшению репутации.

Не менее важен аспект безопасности. Разделение функциональности между микросервисами дает возможность внедрять изолированные механизмы контроля доступа, снижая риск распространения последствий компрометации одного из сервисов на всю систему. Использование транспортного шифрования, аутентификации на уровне API (например, с помощью OAuth 2.0 или JWT), контроль частоты запросов (rate limiting) и централизованное логирование обеспечивают высокий уровень защиты всей архитектуры.

Выводы по разделу 1

Современное проектирование и оптимизация веб-приложений в сфере электронной коммерции представляют собой сложный, непрерывный процесс, охватывающий все этапы жизненного цикла программной системы – от формирования архитектурной модели и выбора технологий до тестирования, развертывания, масштабирования и постоянного мониторинга.

Совокупность архитектурных, инженерных и организационных решений позволяет создавать конкурентоспособные, масштабируемые и устойчивые к внешним воздействиям E-commerce платформы, способные эффективно функционировать в условиях высококонкурентной цифровой среды.

Современные инструменты разработки веб-приложений предоставляют широкий спектр возможностей для построения высокоэффективных, надежных и масштабируемых систем.

Интеграция клиентской и серверной логики в рамках единого фреймворка, применение типобезопасного ORM-слоя, использование edge-ориентированных хранилищ данных, а также построение взаимодействия на основе API и микросервисной архитектуры формируют устойчивую техническую основу для реализации масштабируемых и адаптивных решений. Такие подходы позволяют оперативно реагировать на изменения рыночной среды, интегрировать внешние сервисы и обеспечивать бесперебойную работу приложений в условиях высоких и нестабильных нагрузок.

Эффективность и конкурентоспособность E-commerce платформ определяется комплексным применением современных средств и методов разработки, включающих организацию архитектуры, стратегию хранения и доставки данных, стандарты взаимодействия между подсистемами и инструментальные средства производственного контроля. Оптимизация веб-приложения перестает быть локальной задачей, становясь системной деятельностью, требующей инженерной строгости, адаптивности и постоянного совершенствования технических решений.

2. Разработка веб-приложения на Next.js: архитектура, реализация и оптимизация

2.1 Функциональные требования к системе и используемый стек технологий

В рамках проекта реализована платформа электронной коммерции, включающая базовый набор функциональных модулей: авторизацию пользователей, управление каталогом товаров и функционал корзины покупок.

Пользовательский интерфейс построен по следующей структуре:

1. Главная страница с отображением баннеров, популярных товаров, категорий и поиском по сайту, а также элементами навигации к авторизации и регистрации.
2. Страницы авторизации и регистрации с соответствующими формами ввода и дополнительной возможностью использования социальных сетей для входа и регистрации.
3. Личный кабинет пользователя с отображением персональной информации, возможностью редактирования профиля, просмотра истории заказов и списка избранных товаров.
4. Страницы управления профилем, истории заказов и избранных товаров с функционалом просмотра, редактирования и навигации к карточкам продуктов.
5. Каталог товаров с фильтрацией.
6. Карточка отдельного товара с подробной информацией, медиа-материалами, пользовательскими рейтингами, а также возможностью добавления товара в корзину и избранное.

7. Корзина покупок с функционалом изменения количества товаров, удаления позиций и перехода к оформлению заказа.

8. Страница оформления заказа заполнением данных, вводом дополнительных данных, выбором способов оплаты и доставки.

9. Страница подтверждения заказа с отображением итоговой информации и возможностью перейти к истории заказов.

Дополнительный функционал, реализованный в рамках платформы и направленный на повышение удобства использования и конкурентоспособности системы, включает:

1) регистрацию и авторизацию через социальные сети (Яндекс);

2) расширенные возможности фильтрации товаров;

3) возможность выбора вида отображения каталога.

Разработка веб-приложения в рамках данного проекта осуществлялась на базе современного технологического стека, ориентированного на высокую производительность, масштабируемость, удобство сопровождения и быструю разработку в условиях ограниченных временных ресурсов.

Для реализации основных и дополнительных функций системы выбран современный технологический стек с использованием Next.js – фреймворка, объединяющего клиентскую и серверную логику, что позволило гибко реализовать стратегии рендеринга (SSR, CSR).

С помощью встроенных API-роутов реализованы серверные функции – работа с базой данных, авторизация и регистрация пользователей – без необходимости создания отдельного backend-сервера, что упростило архитектуру и ускорило разработку.

Весь код написан на TypeScript, что обеспечивает строгую типизацию, повышенную надежность и раннее выявление ошибок при реализации функционала, особенно важного для работы с формами, API.

Для стилизации интерфейсов применен Tailwind CSS, позволяющий быстро и эффективно создавать UI-компоненты, стандартизировать

визуальные паттерны и снижать сложность поддержки стилей, что способствует удобству взаимодействия пользователей с платформой.

Управление состоянием реализовано с помощью Zustand, что обеспечивает минималистичный и высокопроизводительный подход к работе с данными приложения без излишних архитектурных ограничений, позволяя оперативно адаптироваться под динамично изменяющиеся функциональные требования.

Для работы с асинхронными запросами к серверу и управления состоянием данных использована библиотека TanStack Query, обеспечивающая кэширование, инвалидацию, автоматическое обновление и обработку ошибок, что значительно улучшило взаимодействие пользователя с платформой и стабильность функционала.

Обработка и валидация форм реализованы через React Hook Form и Zod, что позволило минимизировать избыточные ререндеры, централизованно управлять правилами валидации и обеспечить корректность данных на клиентской и серверной сторонах – ключевое требование для обеспечения безопасности и удобства пользователей.

Приложение развернуто на платформе Vercel, что обеспечивает автоматическое масштабирование, CDN, preview-окружения и CI/CD, а хранение данных организовано через распределенную edge-базу Turso на базе SQLite, что гарантирует отказоустойчивость и высокую скорость отклика.

2.2 Проектирование архитектуры клиентской части платформы

Одним из ключевых аспектов при разработке программных продуктов является проектирование архитектуры приложения, поскольку от ее качества

напрямую зависит удобство сопровождения, масштабируемость и устойчивость системы.

Под архитектурой frontend-приложения принято понимать структурированное определение модульных границ, организацию каталогов и файлов, а также четкие правила взаимодействия между отдельными компонентами системы. Проще говоря, архитектура задает способ организации кода и его компонентов для обеспечения согласованности, повторного использования и удобства поддержки.

Одним из примеров является архитектурная методология FSD (Feature-Sliced Design) представляет собой набор правил и соглашений, предназначенных для систематизации структуры frontend-приложений. Методология FSD выделяет несколько логических слоев, каждый из которых выполняет определенную функцию [2]:

1) App – слой, отвечающий за инициализацию приложения, включающий роутинг, точки входа, подключение глобальных стилей и провайдеров;

2) Pages – отдельные страницы приложения или крупные секции, управляемые маршрутизацией;

3) Widgets – относительно крупные, автономные элементы функциональности или интерфейса, реализующие отдельные пользовательские сценарии;

4) Features – повторно используемые элементы, реализующие конкретные бизнес-функции, приносящие ценность конечному пользователю;

5) Entities – бизнес-сущности предметной области (например, пользователь, продукт), вокруг которых строится логика приложения;

6) Shared – общий, переиспользуемый код, не зависящий от бизнес-логики конкретного проекта.

Следует отметить, что практика разработки редко допускает «чистое» следование учебной модели. Даже такие методологии, как Feature-Sliced Design или Hexagonal Architecture, являются не более чем ориентирами: число слоев, их обязанности и взаимосвязи на практике подгоняются под конкретный продукт, размер команды и требования к развертыванию продукта.

Таким образом, каждое архитектурное решение – это компромисс между теоретическими принципами и практическими ограничениями. Поэтому выбранная архитектура опирается на ключевые идеи FSD, но упрощена под нужды проекта небольшой сложности.

Помимо организации структуры кода, при разработке программной системы рекомендуется опираться на принципы программирования, способствующие созданию масштабируемого, устойчивого и легко сопровождаемого кода. Эти принципы не зависят от используемых библиотек или фреймворков и представляют собой общепринятые подходы, лежащие в основе профессиональной разработки.

В профессиональной разработке можно выделить следующие принципы:

1) SOLID: для достижения более высокого уровня архитектурного качества и устойчивости проекта целесообразно применять принципы.

SOLID – совокупность методологических рекомендаций, направленных на повышение модульности, читаемости и расширяемости программного кода. Адаптация этих принципов под специфику разрабатываемой системы позволяет избежать распространенных архитектурных ошибок и закладывает основу для долгосрочной поддержки и развития проекта. Представляет собой набор из пяти следующих принципов.

S – единственная ответственность: у класса должна быть только одна причина для изменения;

O – открытость/закрытость: код открыт для расширения, но закрыт для изменения;

L – подстановка Лисков: подклассы должны полностью заменять родительские классы без нарушения логики;

I – разделение интерфейса: не заставляй класс реализовывать методы, которые ему не нужны;

D – инверсия зависимостей: зависимость должна быть от абстракций, а не от конкретных реализаций.

2) DRY (Don’t Repeat Yourself) – принцип, исключающий дублирование кода и способствующий повышению его переиспользуемости и легкости модификации;

3) KISS (Keep It Simple, Stupid) – призыв к простоте и ясности в реализации, избеганию излишней сложности и усложненных решений.

В контексте рассматриваемого проекта, обладающего относительно невысокой сложностью и ограниченным функциональным набором, применение строгих и комплексных архитектурных решений было признано излишним и потенциально усложняющим разработку и сопровождение.

Поэтому была реализована адаптированная архитектура, основанная на ключевых идеях FSD, но упрощенная и кастомизированная под конкретные нужды проекта. В частности, была выделена следующая структура слоев:

1) Pages – отвечающий за основные маршруты и отображение целых страниц приложения. Предназначен исключительно для композиции страниц приложения, не содержащих собственной бизнес-логики, а отвечающих за организацию структуры страниц посредством комбинирования компонентов и задания макета (layout).

2) Components – декомпозированные функциональные блоки интерфейса, которые благодаря Tailwind-стилям и изолированным хукам можно свободно переиспользовать между страницами внутри проекта.

3) Shared – модуль, содержащий общие утилиты, функции и стили, используемые во всех частях приложения. Слой Shared содержит в себе универсальные, переиспользуемые элементы приложения, такие как:

a) конфигурация API, включающая функции для выполнения сетевых запросов и обработки ответов;

b) описание констант, позволяющих централизованно управлять значениями, используемыми по всему приложению, что способствует удобству поддержки и снижению ошибок;

c) типы данных (TypeScript interfaces и типы), описывающие бизнес-сущности, например, User, Order, Product – что облегчает парсинг данных с сервера и типобезопасность кода;

d) кастомные React-хуки, обеспечивающие логику получения и обработки данных, в основе которых лежит библиотека React Query, позволяющая эффективно управлять кэшированием, повторными запросами и инвалидацией данных;

e) вспомогательные утилиты, реализующие часто используемые функции, например форматирование дат («06.02.2001» → «6 февраля 2001 года»);

f) конфигурация глобального состояния, реализованная посредством библиотеки Zustand, отвечающая за централизованное хранение и управление состоянием приложения.

В процессе разработки соблюдались перечисленные выше принципы: SOLID, DRY и KISS.

Пример реализации принципа DRY заключается в виде вынесения повторяющейся бизнес-логики в отдельные утилитарные функции или хуки, что позволило избежать дублирования кода в нескольких компонентах. Реализация принципа DRY представлена на рисунке 2.

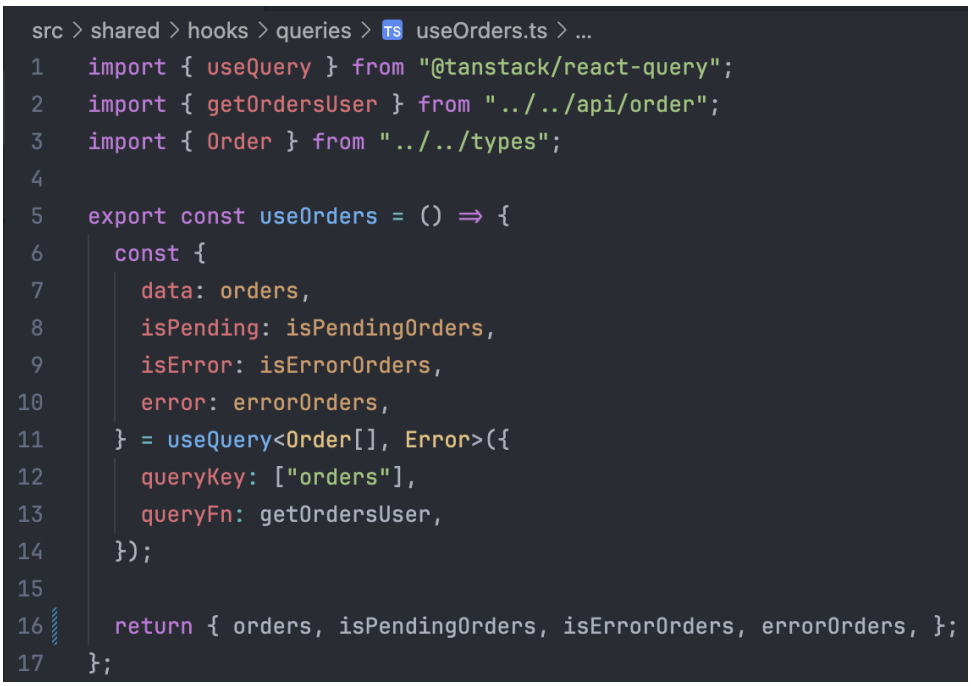


Рисунок 2 – Реализация принципа DRY

Принцип KISS реализуется путем выбора простых и понятных решений. Например, метод formattedDate, представленный на рисунке 3, отвечает за форматирование даты в нужный формат. Он использует стандартные возможности JavaScript без лишней логики или зависимостей. Функция выполняет одну конкретную задачу, легко читается и поддерживается, что полностью соответствует принципу KISS.

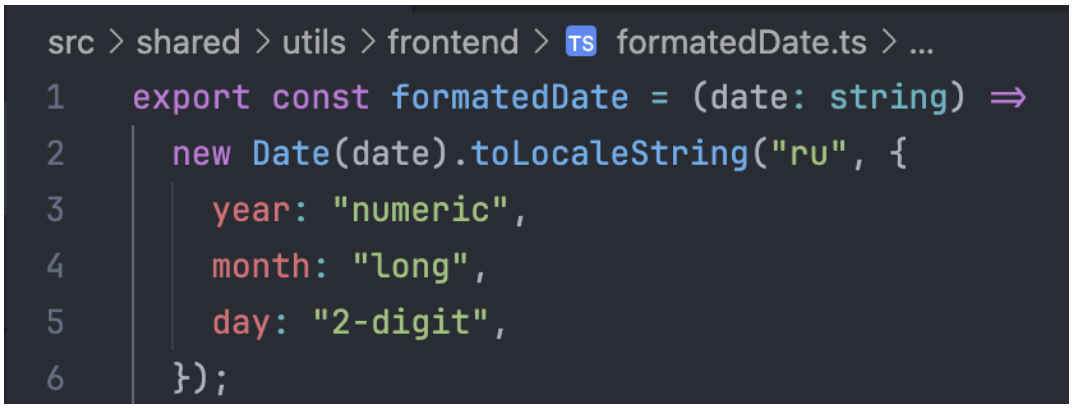


Рисунок 3 – Реализация принципа KISS

Принцип единственной ответственности S из SOLID можно показать на примере компонента OrderHistory, представленном на рисунке 4. Компонент

OrderHistory отвечает исключительно за визуализацию, а получение данных и бизнес-логику реализует специальный кастомный хук.

src > components > OrderHistory > ui > OrderHistory.tsx > ...

```tsx
import { roboto } from "@/styles/fonts";
import { useOrders } from "../../../shared/hooks/queries/useOrders";
import LoadingIcon from "../../LoadingIcon/LoadingIcon";
import { OrderCard } from "./OrderCard";

export function OrderHistory() {
  const { orders, isPendingOrders } = useOrders();

  return (
    <div className="flex flex-col pb-20 md:pb-0">
      <h1
        className={`${roboto.className} hidden pb-5 text-2xl font-bold md:block`}
      >
        История заказов
      </h1>
      {/*Список*/}
      <div className="flex flex-col gap-4">
        {/*Карточки*/}
        {isPendingOrders ? (
          <LoadingIcon />
        ) : orders && orders.length !== 0 ? (
          orders.map((order) => <OrderCard key={order.id} order={order} />)
        ) : (
          <div className="flex items-center justify-center ">
            <p className="text-base font-normal text-gray-500">
              Заказов пока нет
            </p>
          </div>
        )}
      </div>
    </div>
  );
}
```

Рисунок 4 – Реализация принципа S (единственной ответственности)

Принцип разделения интерфейса (Interface Segregation Principle) реализован через передачу в компоненты только тех данных, которые действительно необходимы для их работы.

В компоненте ProductCard вместо полного типа Product, содержащего множество полей, используется объект с минимальным набором свойств: title, photos, price.

Такой подход позволяет избежать избыточных зависимостей от интерфейсов, которые компонент не использует, и делает его более изолированным и устойчивым к изменениям в общем типе Product. Благодаря этому компонент проще переиспользовать, поддерживать и тестировать, так

как он четко описывает, что ему действительно нужно для корректной работы.

Дополнительно, использование возможностей TypeScript позволяет не создавать отдельные интерфейсы для каждого случая, а описывать необходимые поля непосредственно в типе пропсов. При этом объекты с более широким набором свойств могут передаваться без ошибок, если они содержат все требуемые поля. Это упрощает разработку, повышает читаемость кода и делает архитектуру более гибкой и устойчивой к изменениям. Реализация данного принципа представлена на рисунке 5.

```tsx
type Props = {
  item: {
    item: { title: string; photos?: { photoLink: string }[]; price: number };
    quantity: number;
  };
};

export const ProductCard = (props: Props) => {
  const { item } = props;

  return (
    <div className="flex gap-2">
      <Image
        className="rounded-lg object-contain"
        src={item.item.photos?.[0].photoLink || "/images/Product.jpg"}
        alt={item.item.title}
        width={80}
        height={80}
      />
      <div className="flex flex-col justify-between">
        <p
          className={`${inter.className} whitespace-nowrap text-base font-normal text-blue-600`}
        >
          {item.item.title}
        </p>
        <div className="flex items-end gap-1">
          <p
            className={`${roboto.className} whitespace-nowrap text-xl font-bold uppercase text-gray-800`}
          >
            {item.item.price} ₽
          </p>
          <p
            className={`${inter.className} whitespace-nowrap text-base font-normal text-gray-500`}
          >
            {item.quantity} шт.
          </p>
        </div>
      </div>
    </div>
  );
};
```

Рисунок 5 – Реализация принципа разделения интерфейса

Принцип инверсии зависимостей (Dependency Inversion Principle), представленный на рисунке, реализован через передачу зависимостей извне, а не создание их внутри компонента. В компоненте ProductList вместо жестко заданного способа отображения товаров используется абстракция – функция renderProduct, которая передается через пропсы. Это позволяет ProductList оставаться независимым от конкретной реализации визуализации продукта. Благодаря этому компонент можно легко переиспользовать в разных контекстах, подменять поведение в тестах или адаптировать под разные типы карточек без изменения самого компонента. Такой подход повышает гибкость архитектуры и облегчает поддержку и масштабирование интерфейса.

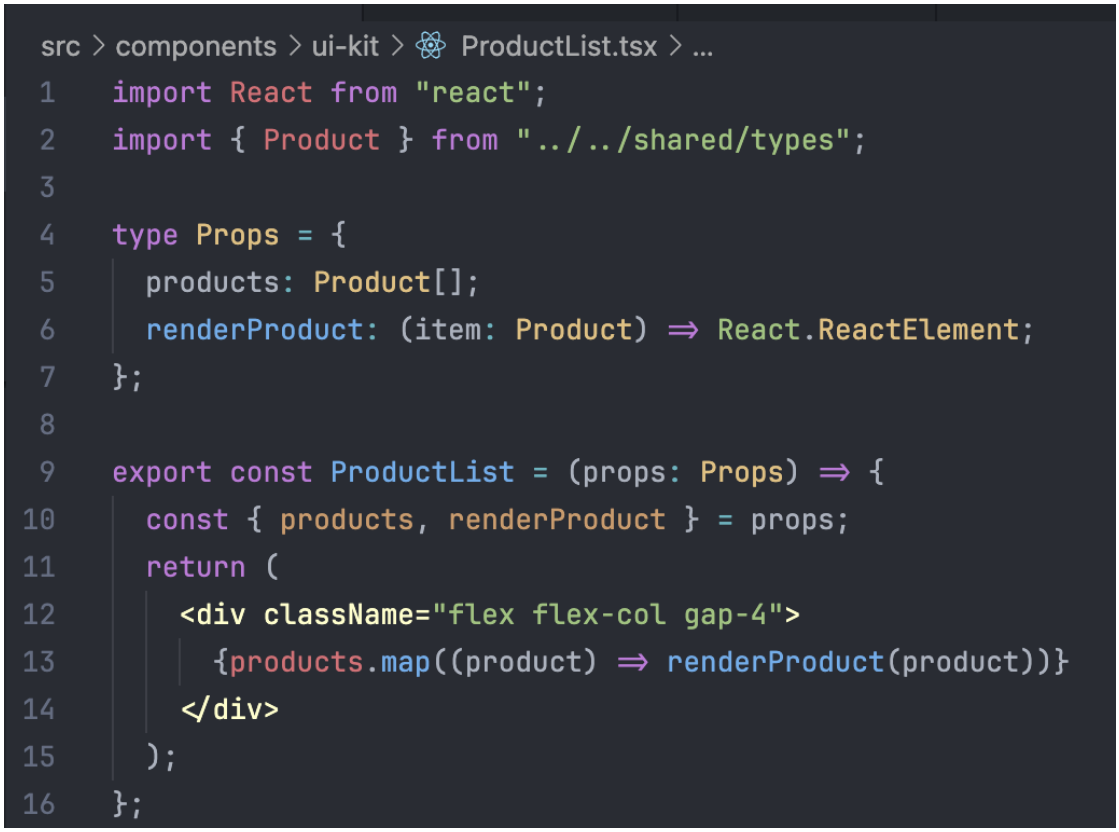


Рисунок 6 – Реализация принципа инверсии зависимостей

2.3 Реализация пользовательского интерфейса с использованием Next.js и Tailwind CSS

Для реализации пользовательского интерфейса в проекте был выбран фреймворк Next.js совместно с CSS-библиотекой Tailwind CSS. В отличие от классических CSS-библиотек, таких как Bootstrap или Foundation, которые предлагают набор готовых компонентов с фиксированными стилями, Tailwind CSS представляет собой утилитарный CSS-фреймворк с минималистичными, атомарными классами, контролирующими конкретные свойства CSS. Такой подход позволяет непосредственно в разметке HTML эффективно и гибко задавать стили элементов без необходимости создавать и поддерживать отдельные CSS-классы.

Использование заранее определенного, но при этом легко настраиваемого набора классов для управления отступами, цветами, размерами шрифтов и другими стилевыми параметрами, значительно ускоряет процесс разработки. Разработчики освобождаются от рутинной задачи придумывания имен, классов и написания соответствующих CSS-правил, что особенно важно при сжатых сроках и необходимости быстрого прототипирования. Пример реализации в коде представлено на рисунке 7.

```jsx
{/*Статус*/}
<div className="flex gap-2">
  <p
    className={`${inter.className} text-base font-bold text-emerald-500`}
  >
    {calculateStatusOrder(order.createOrderDate)}
  </p>
  <p
    className={`${inter.className} text-small font-normal text-gray-500`}
  >
    {order.isCourier ? "доставка" : "в пункте выдачи"}
  </p>
</div>
```

Рисунок 7 – Пример JSX-разметки компонента

Для динамичного формирования стилей в компонентах пользовательского интерфейса широко применяются утилиты clsx и tailwind-merge. Clsx позволяет декларативно управлять списками классов на основе логических условий, упрощая реализацию вариативного внешнего вида компонентов.

В свою очередь, tailwind-merge обеспечивает корректное объединение и приоритетное разрешение конфликтующих классов в рамках системы Tailwind CSS.

Совместное использование этих инструментов особенно эффективно при построении параметризуемых и переиспользуемых компонентов, внешний вид и поведение которых изменяются в зависимости от состояния приложения или переданных параметров, что представлено на рисунке 8.

```
<Input
  {...field}{...inputProps}
  onChange={fieldOnChange}
  ref={props.mask}
  className={cn(
    `${fieldState.error ? "border-red-500" : "border-gray-400"} w-full rounded px-3 py-2 text-sm sm:text-base`,
    "hover:border-blue-600"
  )}
  autoComplete="on"
/>
{fieldState.error?.message && (
  <ErrorMessage text={fieldState.error?.message} font={errorFont} />
)}
```

Рисунок 8 – Передача условия зависимости стилей

Такой подход способствует повышению читаемости кода, снижению количества дублируемой логики и упрощает масштабирование интерфейса в условиях растущей функциональной сложности.

Применение этих библиотек вынесено в отдельную функцию, представленную на рисунке 9.

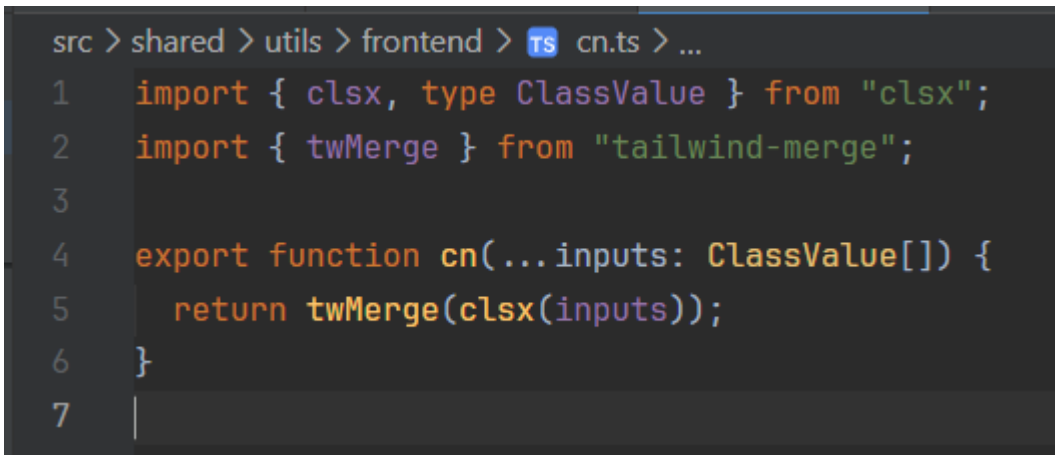


Рисунок 9 – Совместное применение библиотек clsx и tailwind-merge

В основе Tailwind CSS лежит использование адаптивных единиц измерения «rem» вместо фиксированных пикселей (px). Единица «rem» ссылается на системный размер шрифта браузера, что обеспечивает естественное и масштабируемое поведение интерфейса при изменении пользовательских настроек браузера и адаптации к различным устройствам с разной плотностью пикселей и разрешением экрана. Такой подход повышает доступность веб-приложений, поскольку элементы интерфейса корректно масштабируются при увеличении базового размера шрифта – это особенно важно для пользователей с нарушениями зрения, а также при использовании мобильных устройств с различными настройками масштабирования. Кроме того, применение «rem»-единиц способствует однообразности верстки и упрощает управление типографикой и межэлементными отступами.

В разработанном веб-приложении при изменении размера шрифта в параметрах браузера визуальное представление интерфейса корректируется соответствующим образом, сохраняя читаемость и структурную целостность элементов. Данный эффект проиллюстрирован в Приложении А: в первом случае размер шрифта составляет 16 пикселей, во втором – увеличен до 22 пикселей, при этом компоненты интерфейса автоматически масштабируются, обеспечивая корректное отображение и удобство взаимодействия с приложением.

Следует подчеркнуть, что при использовании фиксированных значений размеров шрифта в пикселях, в отличие от применения адаптивных классов фреймворка Tailwind CSS (как продемонстрировано в Приложении Б), интерфейс веб-приложения теряет способность к динамическому масштабированию в ответ на изменения пользовательских настроек браузера. В данном случае элементы отображались бы с жестко заданным размером шрифта, как представлено в Приложении В, что препятствовало бы корректной адаптации интерфейса и могло бы негативно сказаться на доступности и удобстве восприятия контента.

Дополнительно, Tailwind CSS предоставляет широкий набор встроенных классов для реализации адаптивной верстки. Это существенно упрощает и ускоряет создание отзывчивых интерфейсов, позволяя настраивать поведение и отображение элементов на различных точках перелома (breakpoints) без необходимости написания кастомных CSS-медиа-запросов.

Например, такой класс, как md, представленный на рисунке 10, уже содержит предопределенные параметры ширины экрана, соответствующие распространенным устройствам (мобильные телефоны, планшеты, десктопы). Это позволяет разработчикам декларативно управлять стилями, изменяя отступы, размеры, цвета и выравнивание, исходя из ширины экрана, без необходимости самостоятельно рассчитывать и указывать значения ширины. Данный подход снижает вероятность ошибок, связанных с несогласованностью адаптивных стилей, и повышает скорость прототипирования. В результате был реализован адаптивный пользовательский интерфейс, представленный в Приложении Г.

Рисунок 10 – Применение адаптивного класса md

Помимо этого, Tailwind CSS обладает гибкой системой конфигурации, реализуемой через файл tailwind.config.js. С помощью этой системы разработчики могут переопределять значения по умолчанию, создавать собственные темы и дизайн-системы, включая индивидуальные цветовые палитры, параметры типографики, сетки и пользовательские точки перелома. Такая расширяемость особенно актуальна для крупных проектов, где требуется строгое соблюдение корпоративных стандартов и поддержание визуальной целостности интерфейса на всех уровнях приложения.

Важно отметить, что выбор Tailwind CSS в качестве основного инструментария для стилизации позволяет избежать распространенных проблем с каскадом и конфликтами имен, свойственных традиционным подходам к CSS, благодаря строго утилитарной природе классов и отсутствию глобальных стилей. Это способствует улучшению модульности кода и его переиспользованию в рамках масштабных и долгосрочных проектов.

В процессе проектирования интерфейса были также рассмотрены альтернативные методы стилизации, включая CSS Modules, CSS-in-JS (например, styled-components, emotion) и классические SCSS/LESS решения.

CSS Modules предоставляют ограниченную область видимости классов и позволяют избежать конфликтов в глобальном пространстве имен. Однако при активной работе с адаптивностью и сложными условиями отображения компонентов такой подход требует генерации множества именованных классов и дополнительной логики, усложняя читаемость кода и увеличивая его объем. Кроме того, подобная структура препятствует визуальной локализации — разработчику необходимо переходить между JSX и CSS-файлами для понимания, как формируется интерфейсный элемент.

CSS-in-JS решения демонстрируют высокую гибкость и интеграцию логики с визуальным оформлением, особенно в динамически изменяемых интерфейсах. Однако их использование часто приводит к увеличению итогового JavaScript-бандла, негативно сказывается на производительности при рендеринге, а также усложняет отладку стилей, которые компилируются в рантайме и не всегда отображаются в DevTools в привычном виде. В условиях, где приоритет отдается стабильности, предсказуемости и производительности, эти недостатки становятся критичными.

SCSS и другие препроцессоры, несмотря на свою выразительность и богатство возможностей (миксины, вложенность, переменные), плохо сочетаются с компонентной моделью разработки. Глобальные и слабо изолированные стили требуют строгого контроля над каскадом, что приводит к значительным накладным расходам на поддержку и отладку крупных кодовых баз.

2.4. Разработка серверной части и взаимодействие с базой данных

Серверная часть исследуемого веб-приложения была реализована с использованием фреймворка Next.js, что обусловлено его встроенной поддержкой серверных функций и бесшовной интеграции с клиентским

кодом. Благодаря гибкой архитектуре и возможности совмещения клиентской и серверной логики в едином проекте, Next.js позволяет существенно упростить организацию архитектуры приложения и ускорить разработку функциональных модулей.

Архитектура серверной части была структурирована по принципу трехуровневой модели, включающей контроллеры, сервисы и модели (рисунок 11).

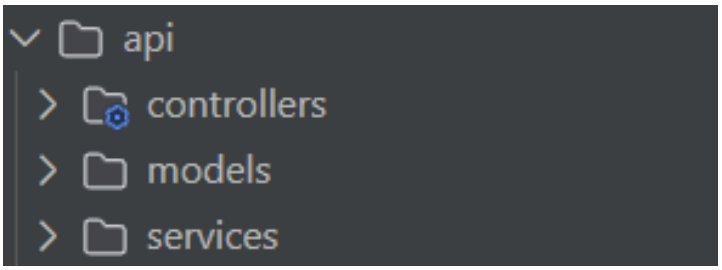


Рисунок 11 – Архитектура серверной части

Подобное разделение ответственности между слоями направлено на обеспечение модульности, тестируемости и масштабируемости кода, а также на соблюдение принципов чистой архитектуры и инверсии зависимостей. Каждый из слоев выполняет строго определенную функцию, способствуя логической декомпозиции системы и упрощая сопровождение проекта.

Контроллеры отвечают за прием, интерпретацию и валидацию входящих HTTP-запросов, а также за формирование соответствующих ответов. Именно в контроллерах осуществляется первичная обработка ошибок, включая конструкцию try/catch, перехватывающую исключения, выброшенные на уровне бизнес-логики.

В случае ошибок формируется корректный ответ с соответствующим HTTP-статусом и сообщением, обеспечивая надежное взаимодействие с клиентской частью приложения, представленное на рисунке 12.

```
export async function getOrderByIdHandler(
  userId: number,
  orderId: number,
  res: NextApiResponse,
) {
  try {
    const orderData = await getOrderById(userId, orderId);
    if (!orderData) {
      return res.status(404).json({ error: "Заказ не найден" });
    }
    return res.status(200).json(orderData);
  } catch (error) {
    console.error("[LOG] Ошибка получения заказа:", error);
    return res.status(500).json({ error: "Ошибка сервера" });
  }
}
```

Рисунок 12 – Обработка ошибок в контроллере

Дополнительно, на уровне контроллеров реализована валидация входных данных при помощи библиотеки Zod (пример реализации представлен на рисунке 13), обеспечивающей строгость типов и структурную целостность запроса. При несоответствии входных данных ожидаемой схеме возвращается ошибка с описанием причины, не доходя до выполнения бизнес-логики.

```
export async function createOrderHandler(
  userId: number,
  data: {
    comment?: string;
    address: string;
    phone: string;
    isCourier: boolean;
    payment?: number;
  },
  res: NextApiResponse,
) {
  try {
    const parsedData = orderSchema.parse(data);

    const orderId = await createOrder(userId, parsedData);
    return res.status(201).json({ orderId });
  } catch (error) {
    if (error instanceof z.ZodError) {
      return res.status(400).json({
        error: "Некорректные данные заказа",
        details: error.errors,
      });
    }
    console.error("[LOG] Ошибка создания заказа:", error);
    return res.status(500).json({ error: "Ошибка при создании заказа" });
  }
}
```

Рисунок 13 – Валидация данных для создания заказа

Пример реализации схемы валидации Zod представлен на рисунке 14.

```
const orderSchema = z.object({
  comment: z.string().optional(),
  address: z.string().min(1, "Адрес обязателен"),
  phone: z.string().min(5, "Телефон обязателен"),
  isCourier: z.boolean(),
  payment: z.number().nullable().optional(),
});
```

Рисунок 14 – Реализация схемы валидации

Сервисный слой инкапсулирует бизнес-логику приложения, выступая связующим звеном между контроллерами и моделью данных. Здесь реализованы основные операции с сущностями системы – такие как

создание, модификация, проверка наличия, удаление и агрегирование данных.

В рамках сервисов используется механизм транзакций для обеспечения атомарности операций, критически важных, например, при создании связанных сущностей или обновлении корзины пользователя. Пример подобного использования приведен в Приложении Д, где демонстрируется процесс создания товара с ассоциированными сущностями.

Также реализованы проверки уникальности данных – например, перед регистрацией пользователя осуществляется проверка уникальности email-адреса, а перед добавлением товара в корзину – проверка его наличия в текущем списке пользователя, как представлено на рисунке 15.

```
// Добавление товара в корзину
export async function addItemToCart(
  userId: number,
  itemId: number,
  quantity: number,
): Promise<void> {
  const existingItem = await db.query.basket.findFirst({
    where: (basket, { eq }) ⇒
      and(eq(basket.userId, userId), eq(basket.itemId, itemId)),
  });

  if (existingItem) {
    await db
      .update(basket)
      .set({ quantity: existingItem.quantity + quantity })
      .where(eq(basket.id, existingItem.id));
  } else {
    await db.insert(basket).values({ userId, itemId, quantity });
  }
}
```

Рисунок 15 – Реализация проверки уникальности данных в корзине

Модельный слой служит для описания структуры и взаимосвязей данных на уровне базы данных. В проекте использовалась библиотека Drizzle ORM, предоставляющая декларативный синтаксис для описания таблиц, связей и ограничений, а также для выполнения типобезопасных SQL-

запросов. Использование Drizzle позволило сократить вероятность ошибок на этапе взаимодействия с базой данных, обеспечить строгую типизацию и повысить читаемость запросов. Модели отражают структуру основных бизнес-сущностей, таких как товары, пользователи, заказы и корзины, и являются единым источником правды для всех операций, связанных с данными. Пример создания модели пользователя представлен на рисунке 16.

```
4   // Пользователи
5   export const users = sqliteTable("users", {
6     id: integer("userId").notNull().primaryKey({ autoIncrement: true }),
7     name: text("name").notNull(),
8     surname: text("surname").notNull(),
9     avatar: text("avatar").notNull().default("images/avatar.png"),
10    email: text("email").notNull().unique(),
11    password: text("password"),
12  });
13
14  export type User = InferSelectModel<typeof users>;
```

Рисунок 16 – Создание модели пользователя

2.5. Механизмы оптимизации рендеринга и загрузки

В современных веб-приложениях оптимизация процессов рендеринга и загрузки контента является одним из ключевых факторов, определяющих как воспринимаемую производительность интерфейса, так и общее качество пользовательского опыта. Архитектура рассматриваемого приложения, разработанного с использованием Next.js, ориентирована на реализацию гибкого и сбалансированного подхода к распределению вычислительной нагрузки между сервером и клиентом с целью максимального повышения эффективности и скорости отклика.

В качестве основного механизма формирования пользовательского интерфейса на стороне сервера применен динамический серверный рендеринг (Server-Side Rendering, SSR), реализуемый средствами Next.js.

Данный подход предполагает генерацию HTML-разметки непосредственно на сервере в момент поступления клиентского запроса, что позволяет существенно сократить время до отображения первого контента (First Contentful Paint, FCP) и повысить доступность ресурса для поисковых систем. SSR также предоставляет возможность учитывать контекст запроса, включая параметры аутентификации, сессионные данные и иные динамические характеристики, что делает его особенно эффективным при построении персонализированных и интерактивных интерфейсов. Применение функции getServerSideProps, реализованное в коде, представлено на рисунке 17.

Рисунок 17 – Применение функции getServerSideProps

На клиентской стороне для организации эффективной работы с асинхронными данными применяется библиотека TanStack Query (ранее – React Query). Она инкапсулирует сложную логику загрузки, кеширования, повторного запроса и оптимистичного обновления данных, позволяя изолировать асинхронные процессы от бизнес-логики компонентов. Кеширование данных, реализуемое на уровне клиента, снижает частоту сетевых запросов и способствует уменьшению задержек при повторных обращениях, а механизмы автоматического обновления (background refetching) и обработки состояний ошибок повышают устойчивость и

предсказуемость поведения интерфейса. Базовое использование useQuery для получения данных и кеширования представлено на рисунке 18.

```
const {
  data: orders,
  isPending: isPendingOrders,
  isError: isErrorOrders,
  error: errorOrders,
} = useQuery<Order[], Error>({
  queryKey: ["orders"],
  queryFn: getOrdersUser,
});
```

Рисунок 18 – Базовое использование useQuery

Для оптимизации загрузки мультимедийного контента в архитектуре приложения применяется компонент <Image> из библиотеки next/image, специально разработанный для эффективной работы с изображениями в среде Next.js. Данный компонент автоматически реализует отложенную загрузку (lazy loading) с использованием внутреннего механизма отслеживания появления элемента в зоне видимости, что позволяет загружать изображения только в момент их фактической необходимости.

Среди ключевых преимуществ компонента <Image> можно выделить: автоматическую генерацию адаптивных изображений для различных разрешений экранов, поддержку современных форматов (таких как WebP и AVIF), оптимизацию размеров на этапе сборки и возможность интеграции с CDN для ускоренной доставки контента. Кроме того, он позволяет явно управлять размерами и приоритетами загрузки с помощью соответствующих атрибутов (width, height, priority и др.), тем самым обеспечивая контроль над визуальной стабильностью макета и производительностью интерфейса.

Использование <Image> способствует снижению времени загрузки страниц, уменьшению трафика и повышению визуальной отзывчивости, что

особенно важно для мобильных пользователей и в условиях ограниченной пропускной способности сети.

Важным элементом комплексной оптимизации веб-приложений выступает рациональное управление загрузкой шрифтов и прочих веб-ресурсов, поскольку от этого напрямую зависят задержки рендеринга и визуальная стабильность интерфейса. Современные практики включают использование актуальных форматов шрифтов, поддерживающих переменные начертания, что позволяет существенно уменьшить объем передаваемых данных за счет динамического изменения веса и стиля шрифта в рамках одного файла. Дополнительно значимый эффект достигается за счет реализации механизма предзагрузки (preload) критически важных ресурсов, который позволяет браузеру приоритетно загружать ключевые элементы, необходимые для первоначального рендеринга страницы. Инлайнинг критических CSS-стилей, включающий стили, непосредственно влияющие на внешний вид контента первого экрана, способствует снижению времени до отображения визуально полной страницы и предотвращает эффект «мигания» незагруженного стиля.

В контексте фреймворка Next.js оптимизация загрузки шрифтов реализуется посредством встроенного API next/font/google, что представляет собой современную и эффективную практику управления веб-шрифтами. Этот механизм автоматически выполняет предварительную загрузку, ограничение набора символов и инкапсуляцию шрифтов на этапе сборки, обеспечивая минимизацию задержек и оптимальный размер загружаемых ресурсов.

Рассмотрим ключевые аспекты оптимизации, реализованные посредством данного API:

1. Использование встроенного API next/font/google обеспечивает интеграцию шрифтов с системой сборки Next.js, благодаря чему происходит автоматическая загрузка и оптимизация файлов шрифтов на серверном этапе,

что повышает производительность и сокращает время до первого отображения.

2. Определение подмножеств символов (например, cyrillic) позволяет загружать только необходимые наборы глифов, исключая избыточные данные, не востребованные в конкретном приложении. Такой подход снижает общий объем файлов и ускоряет время загрузки.

3. Выбор ограниченного количества весов шрифтов (например, 400 и 700) направлен на минимизацию количества загружаемых вариантов начертания, что дополнительно снижает нагрузку на сеть и ускоряет рендеринг. Отсутствие ненужных весов исключает избыточную загрузку и способствует эффективному использованию ресурсов.

4. Установка параметра display: «swap» в конфигурации шрифтов задает стратегию отображения font-display: swap, которая позволяет браузеру использовать запасной системный шрифт до тех пор, пока основной шрифт не будет загружен. Это существенно улучшает восприятие пользователем интерфейса, снижая визуальные артефакты, известные как Flash of Unstyled Text (FOUT).

Реализация загрузки шрифтов, посредством встроенного API next/font/google представлена на рисунке 19.

```
import { Inter, Roboto } from "next/font/google";

//Основной текст, подписи и мелкий текст, текст в кнопках
export const inter = Inter({
  subsets: ["cyrillic"],
  weight: ["400", "700"], // 400 - Regular, 700 - Bold
  display: "swap",
});

//Заголовки
export const roboto = Roboto({
  subsets: ["cyrillic"],
  weight: ["700"], // 700 - Bold
  display: "swap",
});
```

Рисунок 19 – Реализация загрузки шрифтов

Выводы по разделу 2

Разработанная архитектура веб-приложения на основе фреймворка Next.js в полной мере отвечает функциональным требованиям, предъявляемым к современной E-commerce платформе. В рамках реализации обеспечена стабильная работа ключевых пользовательских модулей, включая механизмы аутентификации, управление каталогом, взаимодействие с корзиной и оформление заказов. Использование унифицированного архитектурного подхода позволило достичь согласованности между компонентами клиентской и серверной части приложения.

Выбор архитектурной структуры, ориентированной на простоту и модульность, в сочетании с соблюдением принципов чистого кода (DRY, KISS, SOLID) обеспечил оптимальный баланс между скоростью разработки,

читаемостью кода и возможностью дальнейшего масштабирования. Такая организация проекта способствует повышению устойчивости к техническому долгу и снижает издержки на сопровождение системы.

Особое значение в проекте приобрело применение утилитарной CSS-библиотеки Tailwind, обеспечившей высокую производительность, гибкость в управлении визуальной частью и соответствие современным требованиям к адаптивным интерфейсам. Интеграция Tailwind CSS с Next.js позволила сформировать строгий и единообразный UI без избыточной сложности, что особенно важно при реализации приложений, рассчитанных на широкий круг устройств и сценариев использования.

Серверная логика, реализованная с применением Next.js и Drizzle ORM, демонстрирует современный подход к построению отказоустойчивых и масштабируемых решений. Такая реализация обеспечивает безопасность, расширяемость и удобство поддержки серверной части приложения, закладывая основу для возможной интеграции с микросервисной архитектурой и CI/CD-инфраструктурой.

Совокупность принятых архитектурных и технологических решений формирует надежную, производительную и масштабируемую платформу, адаптированную к потребностям электронной коммерции и способную обеспечить высокий уровень пользовательского опыта в условиях динамически изменяющейся цифровой среды.

3. Развертывание и тестирование веб-приложения

3.1 Развертывание приложения на платформе Vercel

Развертывание (деплоймент) веб-приложения представляет собой завершающий этап жизненного цикла разработки, в рамках которого реализованное программное решение переносится в эксплуатационную среду, доступную конечному пользователю. На данном этапе особое значение приобретает выбор соответствующей инфраструктурной платформы, способной обеспечить высокую доступность, масштабируемость, производительность и отказоустойчивость. В рамках выполнения данной выпускной квалификационной работы в качестве среды для развертывания пользовательского интерфейса было выбрано облачное решение Vercel, зарекомендовавшее себя как надежный и высокоуровневый сервис для хостинга фронтенд-приложений и серверного рендеринга.

Vercel является платформой непрерывной доставки и автоматического развертывания, ориентированная на разработчиков, использующих современные JavaScript-фреймворки, в частности, Next.js, с которым платформа имеет нативную интеграцию. Одним из главных преимуществ Vercel является полная автоматизация процесса CI/CD (Continuous Integration / Continuous Deployment): каждый коммит в основной или функциональной ветке репозитория GitHub, GitLab или Bitbucket инициирует автоматическую сборку проекта, его тестирование, оптимизацию и размещение на производственной или временной (preview) среде. Такой подход позволяет ускорить цикл разработки, повысить стабильность поставки изменений и минимизировать человеческий фактор при развертывании.

На архитектурном уровне Vercel предоставляет гибридную модель рендеринга, поддерживая как статическую генерацию (SSG), так и серверный рендеринг (SSR), а также инкрементальную генерацию (ISR), что особенно важно для достижения высокой производительности и масштабируемости без потери гибкости. Интеграция с глобальной сетью CDN позволяет кэшировать сгенерированные страницы на узлах, максимально приближенных к пользователю, что значительно снижает метрики задержек (latency) и улучшает пользовательский опыт. Кроме того, Vercel самостоятельно управляет балансировкой нагрузки, масштабированием функций и страниц, а также перераспределением трафика при высокой посещаемости.

Важным функциональным компонентом является система предпросмотровых развертываний (Preview Deployments), которая автоматически генерируется для каждого запроса на внесение изменений (pull request). Данная система обеспечивает создание уникального изолированного пространства, позволяющего проводить визуальное тестирование, приемку и ревью функциональных возможностей без необходимости развертывания на основной ветке разработки.

Такой подход значительно ускоряет процесс командной разработки, способствует повышению прозрачности валидации новых функциональностей и способствует снижению технического долга.

Кроме того, данная платформа предоставляет встроенные средства мониторинга и анализа производительности – Vercel Analytics.

Одним из значимых факторов, обусловивших выбор Vercel, является отсутствие необходимости в ручной конфигурации серверной инфраструктуры. Все компоненты, включая маршрутизацию, HTTPS, кастомные домены, переменные окружения, обработку ошибок и логирование, управляются через единый web-интерфейс или CLI-

инструментарий. Это позволяет сосредоточиться на разработке бизнес-логики, не отвлекаясь на иные дополнительные аспекты.

3.2 Тестирование приложения: ручное тестирование и Postman

Тестирование веб-приложения представляет собой неотъемлемую часть жизненного цикла разработки программного обеспечения и служит основным инструментом обеспечения его функциональности, стабильности, безопасности и соответствия требованиям разработки.

В рамках выпускной квалификационной работы было проведено исследование эффективности пользовательского интерфейса и API-интерфейсов. Для оценки функциональности системы использовались методы ручного тестирования и специализированные инструменты, включая Postman, который предназначен для автоматической отправки HTTP-запросов. Эти методы позволили провести анализ корректности работы системы и обеспечить высокую точность результатов тестирования.

Ручное тестирование является одним из наиболее универсальных методов проверки программной системы, при котором специалист по тестированию вручную выполняет действия, симулирующие поведение конечного пользователя поле чего фиксирует наблюдаемые результаты.

Несмотря на ограниченную масштабируемость данного подхода, он остается незаменимым на этапах начальной валидации пользовательского интерфейса (UI), логики взаимодействия компонентов и проверки нестандартных сценариев, трудно поддающихся автоматизации.

В процессе проведения тестирования производилась поэтапная проверка пользовательских сценариев: авторизация, просмотр карточек

товаров, управление корзиной, оформление заказа, проверка реакций на ошибки.

Ручной подход позволил оперативно выявить ошибки отображения, неочевидные недоработки в UX, а также критические состояния.

Для того, чтобы убедиться, что API-интерфейсы работают правильно и надежно, был использован инструмент Postman.

Postman является одной из самых популярных программ для создания, тестирования и описания RESTful API. С помощью Postman можно отправлять запросы к основным частям приложения, таким как создание, чтение, обновление и удаление данных о пользователях, корзинах и товарах.

Были проверены запросы для того, чтобы убедиться, что они соответствует ожидаемому результату: правильно ли сформированы запрос и ответ, какие должны быть коды состояния HTTP, соблюдаются ли правила доступа и аутентификации, как приложение реагирует на неправильные данные, пустые параметры и неверные идентификаторы.

Одним из главных преимуществ использования Postman для тестирования является его способность проверять, как разные части приложения работают вместе.

Например, он может смоделировать взаимодействие между веб-сайтом (frontend) и сервером (backend), а также базой данных (БД). Это позволяет не только проверять, как отдельные запросы работают сами по себе, но и убедиться, что все части приложения правильно взаимодействуют друг с другом.

Кроме того, Postman позволяет создавать цепочки запросов, где данные передаются от одного запроса к другому. Это помогает проверить, как информация проходит через все этапы бизнес-процесса.

В Postman можно использовать простые скрипты на языке JavaScript для автоматической проверки ответов. Это значит, что можно не только убедиться, что сервер вернул правильный ответ, но и проверить, что данные

в этом ответе соответствуют ожиданиям. Например, можно проверить, что в ответе есть нужные поля и что они заполнены правильно.

С помощью Postman можно записывать и сохранять результаты тестирования, что помогает отслеживать, какие тесты прошли успешно, а какие нет.

В процессе тестирования была выполнена проверка на то, чтобы операции не создавали дублирующих записей при повторных запросах. Также проверялось, чтобы API корректно обрабатывал частично валидные данные.

Например, если пользователь пытался добавить один и тот же товар в корзину несколько раз, API должен был увеличивать количество товара, а не создавать дублирующие записи.

Этот подход позволил проверить логику агрегирования и согласованности данных в условиях, приближенных к реальной нагрузке.

Оценка производительности и стабильности API проводилась в рамках базового нагрузочного тестирования вручную при помощи Postman Runner – инструмента для пакетного запуска коллекций запросов. Были смоделированы сценарии с высокой частотой последовательных запросов к серверу, в том числе – к операциям, связанным с корзиной, регистрацией, авторизацией и оформлением заказа. Целью данных тестов было выявление потенциальных проблемных мест в логике обработки, чрезмерной зависимости от состояния базы данных, а также тестирование корректности асинхронных операций при росте нагрузки.

Выводы по разделу 3

В рамках данного этапа было проведено тестирование и развертывание веб-приложения, направленное на обеспечение стабильной и предсказуемой работы функциональных модулей в условиях ограниченных ресурсов и сроков.

Проведенное ручное тестирование, а также использование инструмента Postman для проверки API-интерфейсов позволило выявить функциональные несоответствия веб-приложения.

Примененный метод тестирования сочетает преимущества ручной верификации с возможностью масштабирования процессов контроля качества в будущем. Полученные результаты формируют прочную основу для последующей автоматизации тестов.

Процесс развертывания веб-приложения реализован с использованием платформы Vercel, что позволило обеспечить высокую скорость публикации, автоматическую синхронизацию с системой контроля версий и стабильную доставку кода до конечного пользователя.

Интеграция с CI/CD-инфраструктурой позволила упростить переход от разработки к эксплуатации, минимизируя количество ошибок в процессе публикации и повышая общее качество пользовательского опыта.

Совокупность выбранных подходов к тестированию и развертыванию обеспечила надежность, воспроизводимость и масштабируемость жизненного цикла разработки веб-приложения, соответствующую современным стандартам DevOps-культуры.

ЗАКЛЮЧЕНИЕ

Современные тенденции в области электронной коммерции требуют не только наличия функциональных решений, но и соответствия высоким стандартам производительности, масштабируемости и адаптивности веб-приложений. Разработка конкурентоспособных решений в сфере e-commerce невозможна без системного подхода к проектированию, реализации и оптимизации приложений. В рамках выполненной выпускной квалификационной работы были всесторонне исследованы и применены теоретико-практические аспекты, обеспечивающие соответствие современным требованиям разработки веб-приложений.

В первом разделе работы был проведен обзор ключевых подходов к разработке и оптимизации веб-приложений электронной коммерции, включая современные архитектурные принципы, механизмы интеграции клиентской и серверной логики, применение микросервисной архитектуры и API. Особое внимание уделено вопросам оптимизации клиентской и серверной частей, эффективной доставки ресурсов, кэширования и обработки данных в условиях высокой нагрузки. Теоретический анализ подтвердил необходимость гибких и адаптивных решений, способных устойчиво функционировать в высококонкурентной и быстро меняющейся цифровой среде.

Во втором разделе была спроектирована и реализована технологическая архитектура веб-приложения с использованием современного фреймворка Next.js, TypeScript, Tailwind CSS, Zustand и TanStack Query. Разработанная архитектура ориентирована на обеспечение простоты поддержки, гибкости масштабирования и удобства сопровождения.

Клиентская и серверная логика были интегрированы в рамках единого приложения, что позволило сократить избыточность, повысить производительность и обеспечить целостность обработки данных.

Использование строгой типизации с помощью TypeScript обеспечило своевременное выявление ошибок и снижение вероятности логических нарушений в коде. Tailwind CSS позволил стандартизировать визуальные паттерны и ускорить разработку пользовательского интерфейса, а Zustand и TanStack Query обеспечили эффективное управление состоянием и обработку асинхронных запросов. Реализованный стек технологий был признан оптимальным по совокупности факторов: скорости разработки, удобству поддержки, качеству UI/UX и надежности взаимодействия с backend-инфраструктурой.

Третий раздел был посвящен развертыванию и тестированию приложения. Размещение проекта на облачной платформе Vercel обеспечило автоматизацию всех этапов CI/CD, сокращение времени отклика, внедрение preview-окружений и полную интеграцию с системой контроля версий.

Проведенное ручное тестирование в сочетании с инструментами Postman позволило подтвердить правильность реализации пользовательских сценариев.

В результате проведенной работы:

– разработана технологическая архитектура, включающая основные функциональные модули электронной коммерции: каталог, карточки товаров, корзина, оформление и история заказов, личный кабинет, аутентификация и авторизация;

– реализовано полнофункциональное веб-приложение, соответствующее современным требованиям к производительности и адаптивности;

– внедрены актуальные инженерные практики, включая принципы DRY, KISS, SOLID;

– обеспечено соответствие DevOps-стандартам: CI/CD, Preview Deployments, автоматическая сборка и деплой;

– закладываются возможности масштабирования и перехода к микросервисной архитектуре, что позволяет применять данную платформу как базу для дальнейшего расширения и интеграции с корпоративными решениями.

Таким образом, работа обладает не только теоретической, но и практической значимостью: сформированное веб-приложение представляет собой прототип современного e-commerce решения, соответствующего требованиям отрасли как с точки зрения архитектуры, так и с точки зрения методов разработки.

Все поставленные в данном исследовании задачи были выполнены.

Работа подтверждает, что совокупность современных подходов, правильный выбор стека и инженерных практик, а также внимание к архитектурному проектированию – являются определяющими факторами в успешной реализации проектов электронной коммерции в современную цифровую эпоху.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1. Волнейкина Е. С., Гек Д К., Кукарцев В. В. Как фреймворк Vue js упрощает работу с дизайном при создании интерйфеса // актуальные проблемы авиации и космонавтики. 2021. – с. 481-482.
2. Герасимов А.В. Использование модульных монолитов для разработки масштабируемых веб-приложений // Вестник науки. 2023. № 6 (63). – С. 869–872.
3. Горбунов, А. А., Петров, И. В. Оптимизация серверной инфраструктуры с использованием Redis, Memcached, Docker и Kubernetes. Вестник информационных технологий. 2021. 12(4). – С. 45–58.
4. Гордеев С. Ю., Тимофеев А. В., Козлов В. В. Разработка веб-приложений с использованием angular. Js, node. Js, MongoDB на примере системы психологической поддержки студентов - участников программы «полет» // Наука и образование сегодня. 2017. №2 (13). – С. 1-6.
5. Городничев М. Г., Полонский Р. В. Оценка возможности использования микросервисной архитектуры при разработке пользовательских интерфейсов клиент-серверного программного обеспечения // Экономика и качество систем связи. 2020. №3 (17). – С. 33-43.
6. Дудар С. А. Веб-разработка в сфере торговли: инновационные подходы к созданию эффективных веб-приложений // Теория и практика современной науки. 2023. №5 (95). – С. 73-75.
7. Жиренкин А. В., Старосельский А. К. Развитие элементов архитектуры современных веб-приложений и их влияние на разработку // Вестник науки. 2023. №6 (63). – С. 869-872.
8. Захаров С. В., Котов А. А. Инфраструктура приложений на основе Docker и Kubernetes // Информационные технологии и вычислительные системы. 2021. № 3. – С. 45–52.

9. Ильин А. Ю., Плотников С. Б. Использование предиктивного анализа пользовательского поведения для повышения скорости реактивной загрузки клиентской части веб-приложения // Международный журнал гуманитарных и естественных наук. 2025. №1-3 (100). – С. 140-143.

10. Костенко И. П., Ступина М. В. Повышение производительности WEB-приложений средствами СУБД Redis // Молодой исследователь Дона. 2022. №4 (37). – С. 29-32.

11. Сергачева М. А., Михалевская К. А. Анализ фреймворков для разработки современных веб-приложений // Кронос: естественные и технические науки. 2020. №2 (30). – С. 35-39.

12. Томилов И. О., Трифанов А. В. Современные принципы и подходы к frontend архитектуре веб-приложений // Наука, техника и образование. 2019. № 10(63). – С. 54–57.

13. Фаррахов И. Г., Якупов И. М. Автоматизированный инструментарий развертывания облачных сервисов // Мировая наука. 2021. №2 (47). – С. 115-118.

14. Холодков Д. В., Зинкин С. А. Влияния технологий контейнеризации приложений на скорость выполнения // Вестник ПензГУ. 2024. №4 (48). – С. 134-136.

15. Хомоненко А. Д., Абу Хасан Р. О надежности и доступности объектных хранилищ данных // Интеллектуальные технологии на транспорте. 2023. №S1 (35-1). – С. 123-128.

16. Data Insight. Маркетинговое исследование Интернет-торговля в России 2025 [Электронный ресурс]. Режим доступа – https://datainsight.ru/DI_eCommerce_2025 (дата обращения: 23.05.2025).

17. Documentation Angular [Электронный ресурс]. Режим доступа – https://angular.dev/overview (дата обращения 09.05.2025).

18. Documentation ASP.NET Core [Электронный ресурс]. Режим доступа – https://dotnet.microsoft.com/ru-ru/apps/aspnet (дата обращения 09.05.2025).

19. Documentation C# [Электронный ресурс]. Режим доступа – https://dotnet.microsoft.com/ru-ru/languages/csharp (дата обращения 09.05.2025).

20. Documentation Django [Электронный ресурс]. Режим доступа – https://www.djangoproject.com/ (дата обращения 09.05.2025).

21. Documentation Drizzle [Электронный ресурс]. Режим доступа – https://orm.drizzle.team/docs/overview (дата обращения 09.05.2025).

22. Documentation Echo [Электронный ресурс]. Режим доступа – https://echo.labstack.com/docs (дата обращения 09.05.2025).

23. Documentation Elasticsearch [Электронный ресурс]. Режим доступа – https://www.elastic.co/docs (дата обращения 09.05.2025).

24. Documentation Express.js [Электронный ресурс]. Режим доступа – https://expressjs.com/ (дата обращения 09.05.2025).

25. Documentation FastAPI [Электронный ресурс]. Режим доступа – https://fastapi.tiangolo.com/ (дата обращения 09.05.2025).

26. Documentation Flask [Электронный ресурс]. Режим доступа – https://flask.palletsprojects.com/en/stable/ (дата обращения 09.05.2025).

27. Documentation Gin [Электронный ресурс]. Режим доступа – https://gin-gonic.com/ru/ (дата обращения 09.05.2025).

28. Documentation Go [Электронный ресурс]. Режим доступа – https://go.dev/doc/ (дата обращения 09.05.2025).

29. Documentation Hibernate [Электронный ресурс]. Режим доступа – https://hibernate.org/tools/ (дата обращения 09.05.2025).

30. Documentation Java [Электронный ресурс]. Режим доступа – https://docs.oracle.com/en/java/ (дата обращения 09.05.2025).

31. Documentation MariaDB [Электронный ресурс]. Режим доступа – https://mariadb.com/ (дата обращения 09.05.2025).

32. Documentation MySQL [Электронный ресурс]. Режим доступа – https://dev.mysql.com/doc/ (дата обращения 09.05.2025).

33. Documentation NestJS [Электронный ресурс]. Режим доступа – https://nestjs.com/ (дата обращения 09.05.2025).

34. Documentation Next.js [Электронный ресурс]. Режим доступа – https://nextjs.org/docs (дата обращения 09.05.2025).

35. Documentation PostgreSQL [Электронный ресурс]. Режим доступа – https://www.postgresql.org/ (дата обращения 09.05.2025).

36. Documentation Python [Электронный ресурс]. Режим доступа – https://www.python.org/ (дата обращения 09.05.2025).

37. Documentation React [Электронный ресурс]. Режим доступа – https://react.dev/ (дата обращения 09.05.2025).

38. Documentation Ruby [Электронный ресурс]. Режим доступа – https://www.ruby-lang.org/ru/ (дата обращения 09.05.2025).

39. Documentation Ruby on Rails (Rails) [Электронный ресурс]. Режим доступа – https://rubyonrails.org/docs (дата обращения 09.05.2025).

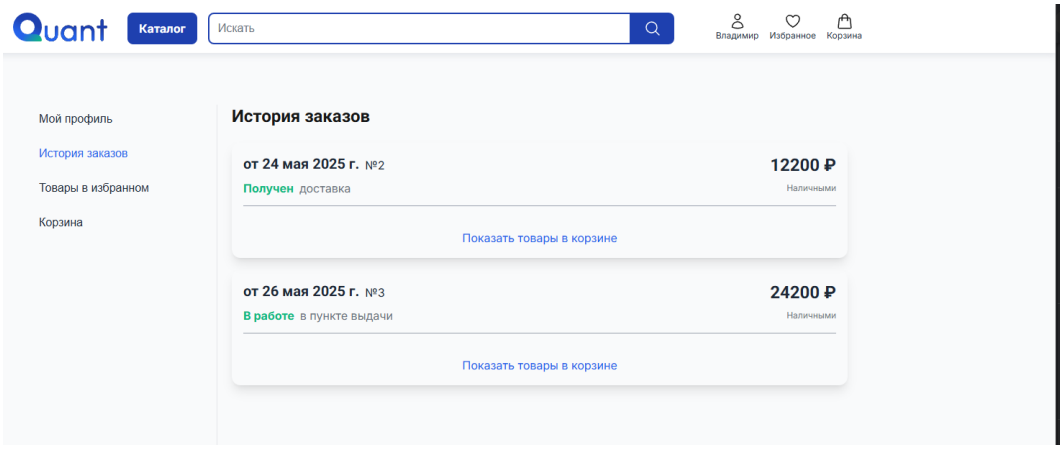
40. Documentation Spring Boot [Электронный ресурс]. Режим доступа – https://spring.io/projects/spring-boot (дата обращения 09.05.2025).

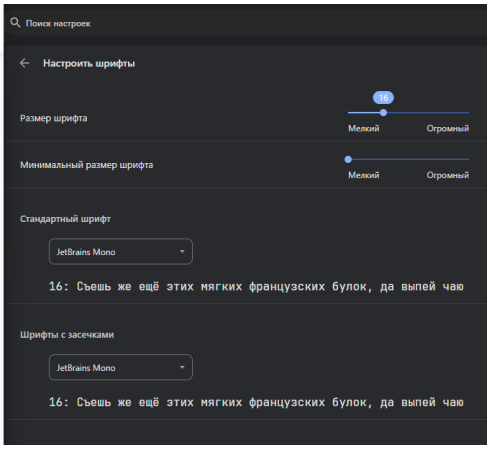
41. Documentation Svelte [Электронный ресурс]. Режим доступа – https://svelte.dev/ (дата обращения 09.05.2025).

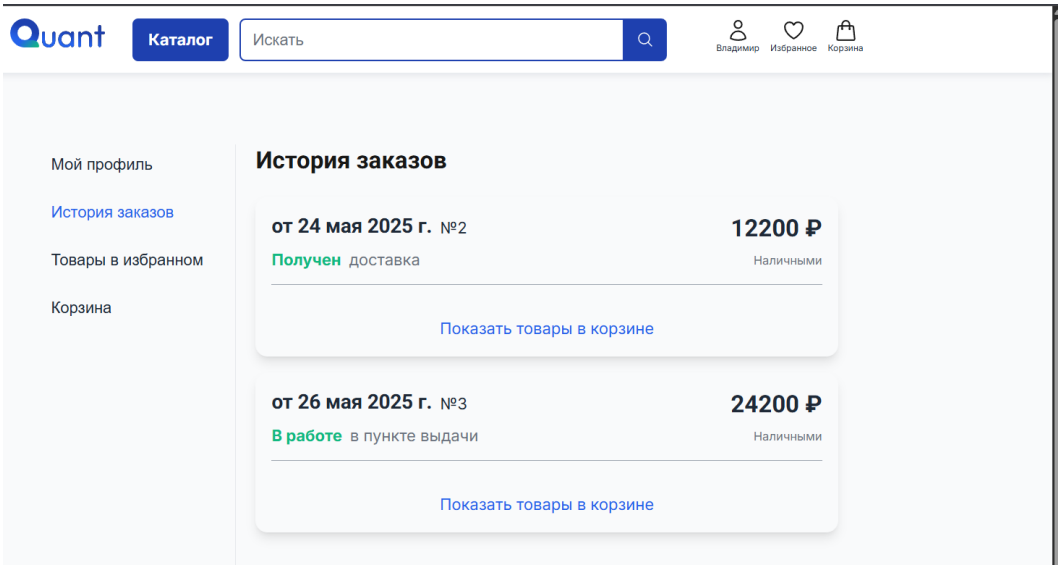
42. Documentation Vue.js [Электронный ресурс]. Режим доступа – https://vuejs.org/ (дата обращения 09.05.2025).

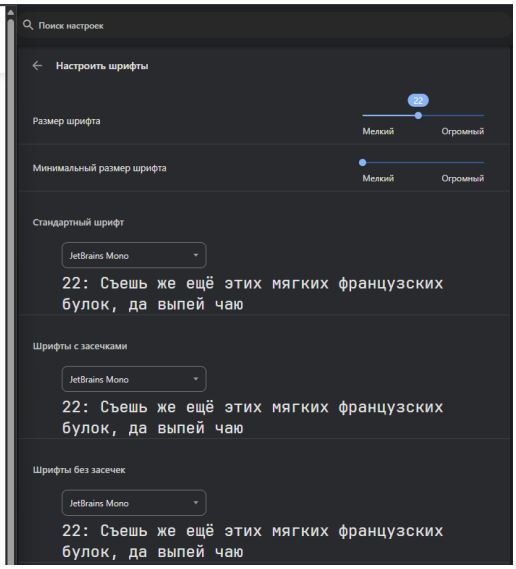
ПРИЛОЖЕНИЕ А

Смена размера шрифта в настройках браузера









ПРИЛОЖЕНИЕ Б

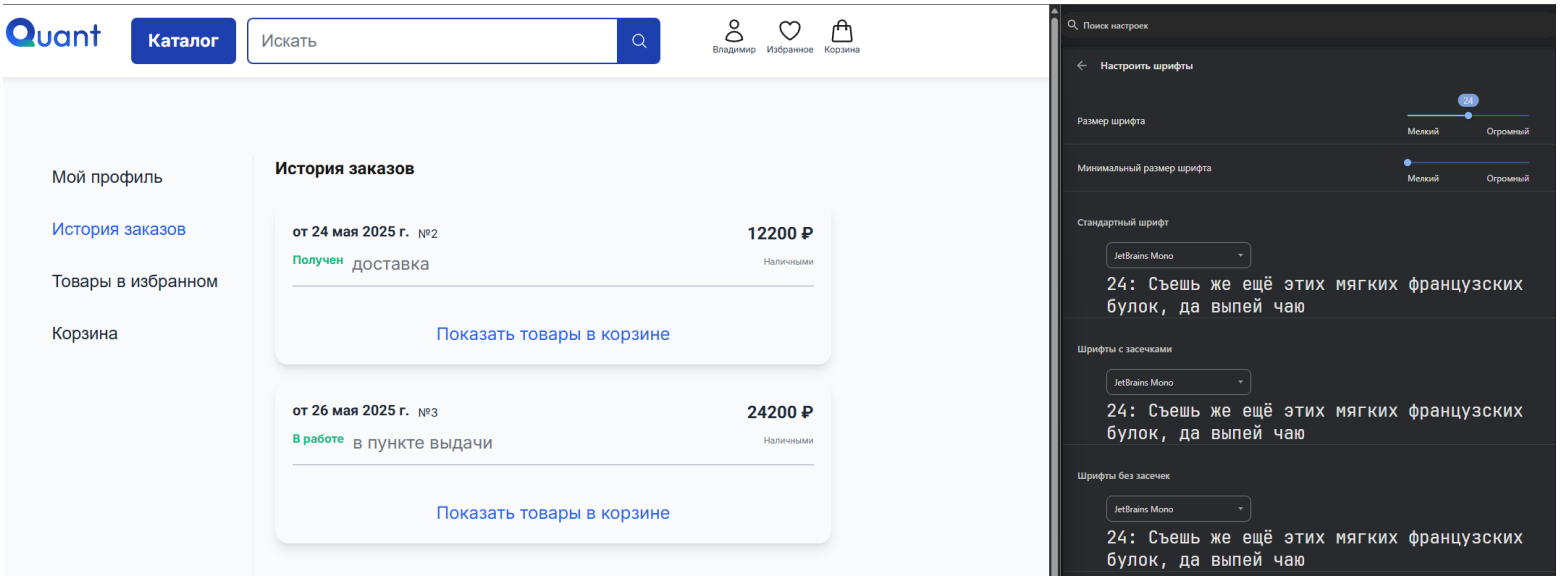
Фиксированные значения размеров шрифта в пикселях

```
return (
  <div className="flex flex-col gap-4 rounded-xl p-4 shadow-lg
  md:min-w-[580px]">
    <div className="flex flex-col border-b border-gray-400 md:flex-row
    md:justify-between">
      {/*Секция с информацией*/}
      <div className="flex flex-col gap-2">
        {/*Время*/}
        <div className="flex items-end justify-between gap-2
        md:justify-start">
          <p
            className={`${roboto.className} text-xl font-bold
            text-gray-800`}
          >
            {`от ${formatedDate(order.createOrderDate)}`}
          </p>
          <p
            className={`${inter.className} text-base font-normal
            text-gray-800`}
          >
            {`№${order.id}`}
          </p>
        </div>
        {/*Статус*/}
        <div className="flex gap-2">
          <p
            className={`${inter.className} text-base font-bold
            text-emerald-500`}
          >
            {calculateStatusOrder(order.createOrderDate)}
          </p>
          <p
```

```
return (
  <div className="flex flex-col gap-4 rounded-xl p-4 shadow-lg md:min-w-
  [580px]">
    <div className="flex flex-col border-b border-gray-400 md:flex-row
    md:justify-between">
      {/*Секция с информацией*/}
      <div className="flex flex-col gap-2">
        {/*Время*/}
        <div className="flex items-end justify-between gap-2
        md:justify-start">
          <p
            className={`${roboto.className} text-[20px] font-bold
            text-gray-800`}
          >
            {`от ${formatedDate(order.createOrderDate)}`}
          </p>
          <p
            className={`${inter.className} text-[16px] font-normal
            text-gray-800`}
          >
            {`№${order.id}`}
          </p>
        </div>
        {/*Статус*/}
        <div className="flex gap-2">
          <p
            className={`${inter.className} text-[16px] font-bold
            text-emerald-500`}
          >
            {calculateStatusOrder(order.createOrderDate)}
          </p>
          <p
```

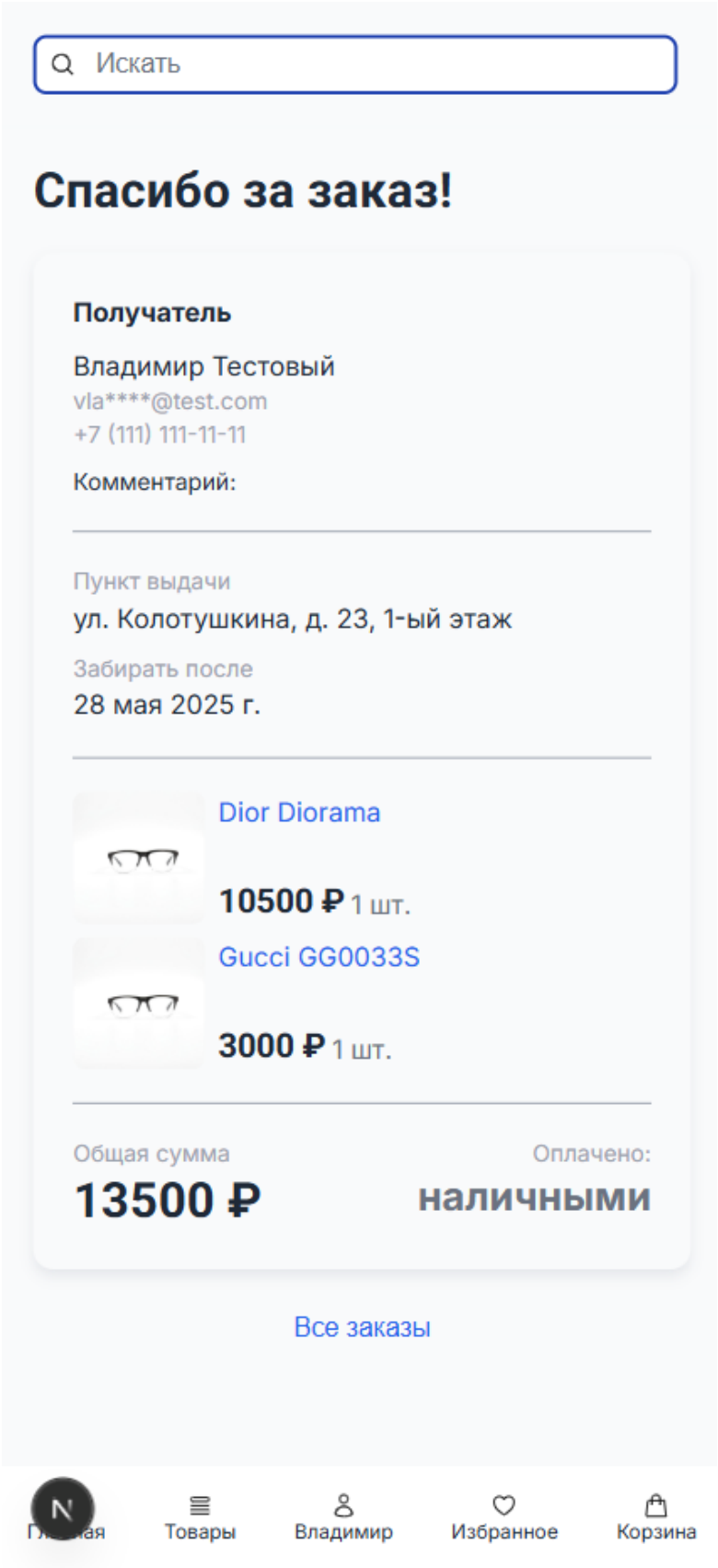
ПРИЛОЖЕНИЕ В

Отображение пользовательского интерфейса с жестко заданным размером шрифта



ПРИЛОЖЕНИЕ Г

Адаптивный пользовательский интерфейс



ПРИЛОЖЕНИЕ Д

Процесс создания товара с ассоциированными сущностями

```
// Создание товара
export async function createItem(itemData: {
  title: string;
  price: number;
  description: string;
  availability: boolean;
  photos?: { photoLink: string; isMainPhoto: boolean }[];
  characteristics?: {
    color: string;
    frameMatherials: string;
    linzeMatherials: string;
    linzeTypes: string;
    linzeUVDefences: string;
    linzeEffects: string;
  };
}): Promise<number> {
  return db.transaction(async (tx) => {
    const [newItem] = await tx
      .insert(items)
      .values({
        title: itemData.title,
        price: itemData.price,
        description: itemData.description,
        availability: itemData.availability,
      })
      .returning({ id: items.id });

    if (itemData.photos && itemData.photos.length > 0) {
      await tx.insert(photos).values(
        itemData.photos.map((photo) => ({
          itemId: newItem.id,
          photoLink: photo.photoLink,
          isMainPhoto: photo.isMainPhoto,
        })),
      );
    }

    if (itemData.characteristics) {
      await tx.insert(characteristics).values({
        itemId: newItem.id,
        color: itemData.characteristics.color,
        frameMatherials: itemData.characteristics.frameMatherials,
        linzeMatherials: itemData.characteristics.linzeMatherials,
        linzeTypes: itemData.characteristics.linzeTypes,
        linzeUVDefences: itemData.characteristics.linzeUVDefences,
        linzeEffects: itemData.characteristics.linzeEffects,
      });
    }

    return newItem.id;
  });
}
```