Министерство образования Республики Беларусь

Учреждение образования  
 БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
 ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Рефакторинг и оптимизация программного кода

Отчет

по результатам выполнения лабораторных работ  
и заданий к практическим занятиям

|  |  |  |
| --- | --- | --- |
| Проверила | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | А.В. Шелест |
| зачтено | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (дата защиты) |  |
| Выполнил | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | Д.В. Наврозов  гр. 214371 |

Минск, 2025

**СОДЕРЖАНИЕ**

[Ссылки на репозитории 3](#_heading=h.ss7nc96ksd0e)

[1 Архитектура программного средства 4](#_heading=h.gq2wtht0a6mx)

[1.1 Диаграмма вариантов использования 4](#_heading=h.ntmnea628747)

[1.2 Нотация моделирования C4-модель. 5](#_heading=h.zcoksvic3qog)

[1.3 Система дизайна пользовательского интерфейса 7](#_heading=h.l9a3ok6mja81)

[1.4 Описание спроектированной архитектуры по уровням   
Clean Architecture 9](#_heading=h.vc9kz66t28sl)

[2 Проектирование пользовательского интерфейса ПС 11](#_heading=h.c4e0hte29pf6)

[3 Реализация клиентской части ПС 14](#_heading=h.z3tt4lwqm9w3)

[4 Спроектировать схему бд и представить описание ее сущностей и их атрибутов 17](#_heading=h.58powoe6g6rc)

[5 Представить детали реализации пс через UML-диаграммы 19](#_heading=h.fs3j6ayypgwy)

[5.1 Описание статических аспектов программных объектов. 19](#_heading=h.y8d94a9grk4c)

[5.2 Описание динамических аспектов поведения программных объектов 22](#_heading=h.vbyqfthnpui3)

[6 Документация к ПС с open api 25](#_heading=h.l1ou6vxo70xq)

[7 Реализация системы аутентификации и авторизации пользователей ПС и механизмов обеспечения безопасности данных 28](#_heading=h.uff0yr3qb1fb)

[8 Unit- и интеграционные тесты 31](#_heading=h.qn6r1b6wonx)

[9 Описание процесса развертывания ПС 35](#_heading=h.u4sqgkmw6n6r)

[10 Разработка руководства пользователя 37](#_heading=h.x57ay96h08rs)

# **Ссылки на репозитории**

[https://github.com/](https://github.com/Ksenchez/OtyakovaK.A_214371_RIOPK_Server.git)

[https://github.com/](https://github.com/Ksenchez/OtyakovaK.A_214371_RIOPK_Front)

[https://github.com/](https://github.com/Ksenchez/OtyakovaK.A_214371_RIOPK_PZ.git)

# **1 АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА**

## **1.1 Диаграмма вариантов использования**

Диаграмма вариантов использования является ключевым инструментом визуализации взаимодействия различных пользовательских групп с программой. Она охватывает разнообразные потребности и интересы, предоставляя общий обзор функциональности программы. В данной диаграмме представлены основные функции программного средства оперативного управления поставками. Участниками системы выступают экономист и менеджер по продажам, каждый из которых обладает определённым набором действий в рамках своих обязанностей.

Сотрудники склада применяют программу для регистрации новых поступлений, их оперативного внесения в учетную систему через загрузку данных из файлов, а также для дальнейшего контроля остатков и подготовки товаров к распределению.

Менеджеры склада анализируют собранную статистику, формируют отчеты о состоянии запасов и эффективности складских процессов, на основании которых могут быть приняты обоснованные управленческие решения.



Рисунок 1.1 – Диаграмма вариантов использования

Диаграмма отражает логичную иерархию пользовательского взаимодействия с программной системой. Разделение ролей между сотрудником склада и менеджером позволяет чётко распределить ответственность: сотрудник управляет данными и приемкой товара, а менеджер использует эти данные для анализа и планирования. Использование механизмов «*include*» и «*extend*» помогает избежать дублирования функциональности и делает диаграмму компактной и понятной.

## **1.2 Нотация моделирования C4-модель.**

На рисунке 1.2 представлена диаграмма контекста системы, отражающая основные внешние взаимодействия программного средства с пользователями и базой данных. Пользователи (сотрудник склада и менеджер) получают доступ к функционалу системы через веб-интерфейс. Приложение обрабатывает их действия и взаимодействует с базой данных для выполнения операций.



Рисунок 1.2 – Контекстный уровень представления архитектуры

На рисунке 1.3 показана диаграмма контейнеров, в которой детализирована архитектура приложения на уровне логических блоков. Система состоит из трёх основных контейнеров: бэкенда на *NodeJS* и базы данных *MongoDB*. Также представлен контейнер для модульного тестирования бизнес-логики с использованием *XUnit*. Контейнеры взаимодействуют через *HTTP*-запросы и *ORM*-интерфейс *Entity Framework*.



Рисунок 1.3 – Контейнерный уровень представления архитектуры

На рисунке 1.4 приведена диаграмма компонентов, демонстрирующая структуру основных функциональных модулей системы. Для бэкенда выделены контроллеры, отвечающие за авторизацию, работу с поставками, поставщиками и товаром. Компоненты фронтенда реализованы как модули, каждый из которых обеспечивает доступ к определённой части бизнес-логики через *REST API*.

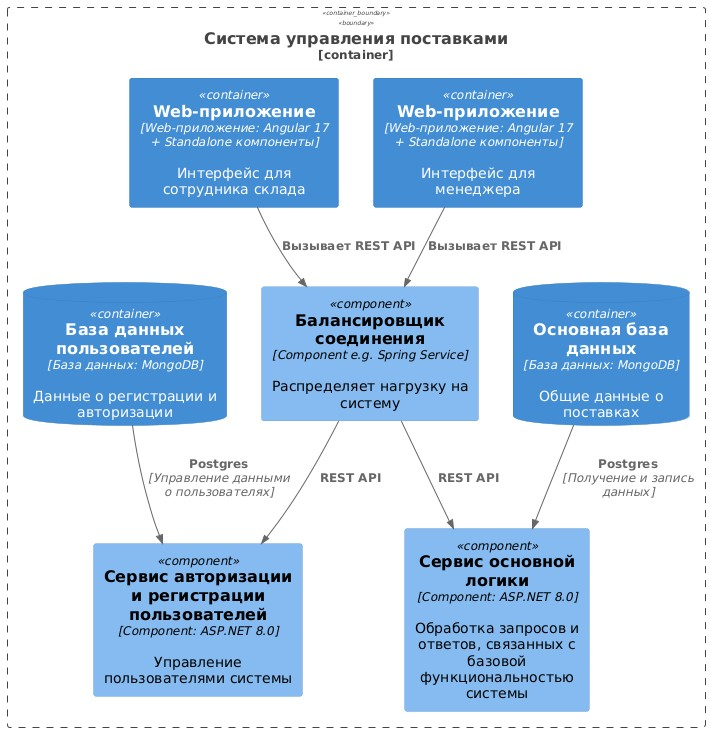


Рисунок 1.4 – Компонентный уровень представления архитектуры

На рисунке 1.5 представлена архитектура системы загрузки, иллюстрирующая взаимодействие различных компонентов. Данная диаграмма описывает кодовый уровень архитектуры *backend*-приложения, реализованного с использованием *NodeJs*. На этом уровне мы видим взаимодействие между ключевыми компонентами системы: контроллерами, сервисами, моделью данных и инфраструктурой доступа к данным.

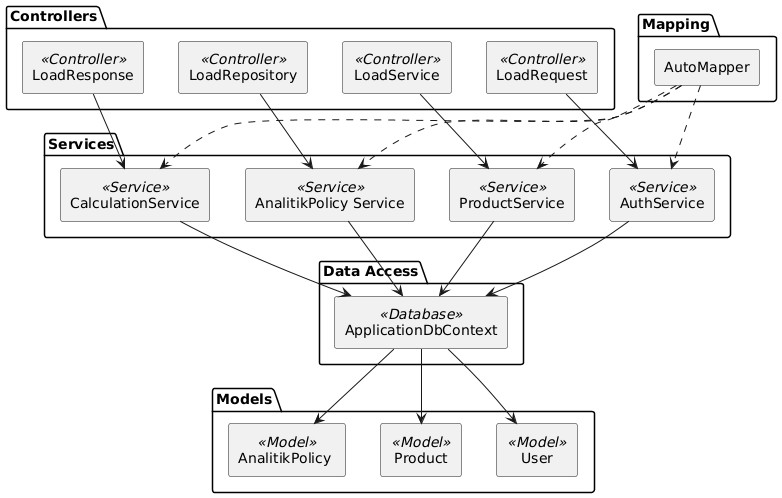


Рисунок 1.5 – Кодовый уровень представления архитектуры

Что отображает диаграмма:

1 *LoadRequest*: это компонент, который инициирует запрос на вход в программу. Он содержит различные параметры, необходимые для выполнения процесса загрузки.

2 *LoadService*: это сервис, который обрабатывает запрос на загрузку, полученный от *LoadRequest*. Он выполняет основную логику обработки данных и управляет процессом загрузки.

3 *SystemException*: это исключение, которое может быть сгенерировано в случае возникновения ошибки во время выполнения процесса загрузки. Это помогает в обработке ошибок и предоставляет информацию о проблеме, которая может быть использована для отладки и исправления.

4 *LoadRepository*: это база данных, в которой хранятся все данные системы. Она необходима для выполнения различных команд запросов, отображения и хранения данных.

5 *LoadResponse*: это система отчетов, которая отображает всю необходимую информацию по складу.

## **1.3 Система дизайна пользовательского интерфейса**

В разработке программного средства мы уделяли особое внимание системе дизайна пользовательского интерфейса, чтобы обеспечить единый стиль, функциональность и привлекательность элементов интерфейса. Наша система дизайна была разработана для того, чтобы создать удобный и современный пользовательский опыт.

На рисунке 1.6 изображена разработанная система дизайна, которая включает основные элементы пользовательского интерфейса. Эта система определяет структуру и внешний вид кнопок, полей ввода, элементов навигации, цветовую палитру, шрифты и другие детали интерфейса. Она помогает обеспечить узнаваемость всех частей программного продукта.

Такой подход к дизайну позволяет пользователям легко ориентироваться в приложении, улучшает восприятие функциональности и делает использование программного средства более приятным и эффективным.

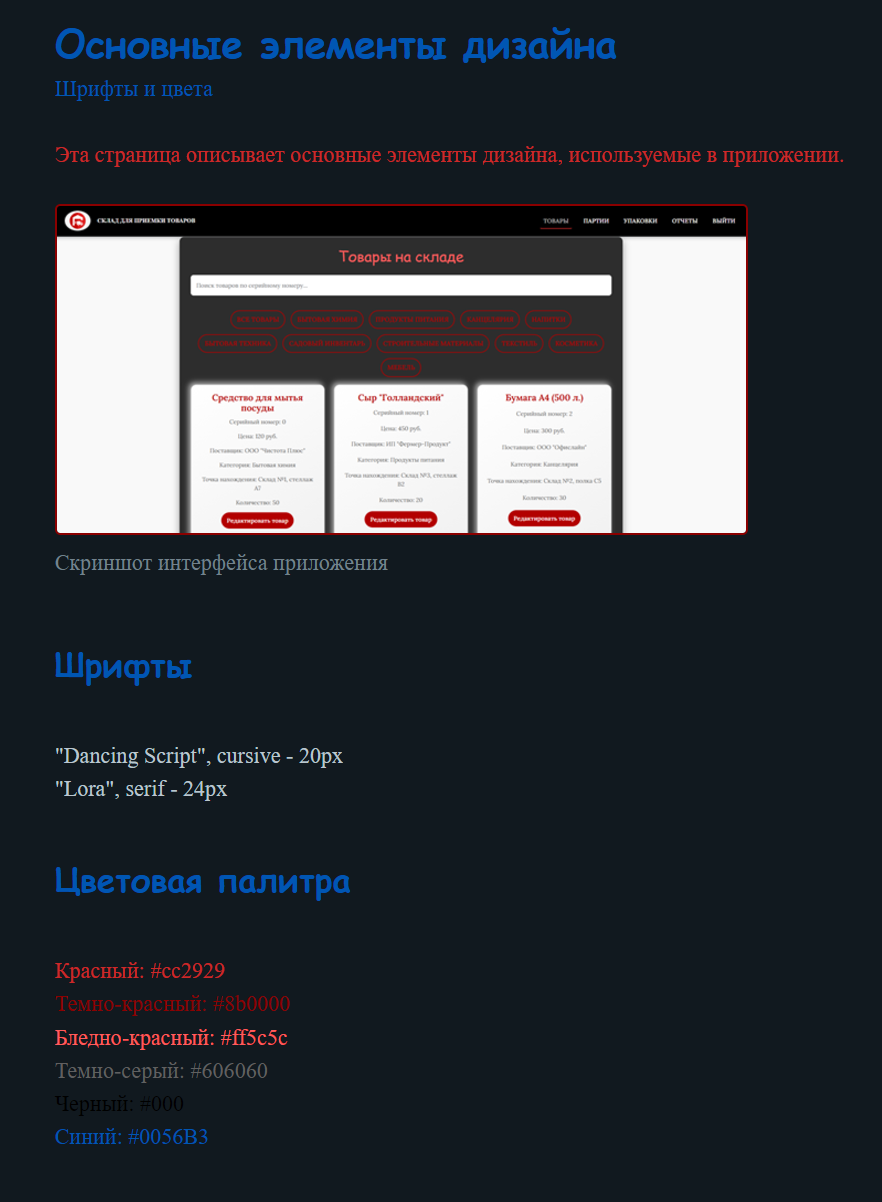
****

Рисунок 1.6 – Система дизайна пользовательского интерфейса программного средства

## **1.4 Описание спроектированной архитектуры по уровням Clean Architecture**

Уровень сущностей (*Entities*)

На данном уровне определены основные бизнес-сущности, отражающие предметную область:

* *Product*: Модель товара с атрибутами *name, category, baseCost, markup, finalPrice, createdAt*.
* *Supplier*: Поставщик с полями *name, contactInfo, products, createdAt*.
* User: Пользователь с *email, passwordHash, role*.
* Package: Партия товара с *product, quantity, location, supplier, receivedDate*.
* Movement: Перемещение товара с *from, to, date, package*.

Эти сущности реализованы с использованием *Mongoose* и не зависят от внешних фреймворков.

Уровень прикладной логики (*Use Cases*)

Этот уровень отвечает за реализацию бизнес-сценариев, управляющих взаимодействием между сущностями и внешними интерфейсами:

* Расчет итоговой стоимости товара: *finalPrice = baseCost \* (1 + markup)*.
* Аутентификация пользователей: Регистрация, вход, хеширование и проверка паролей.
* Управление партиями: Создание, перемещение и отслеживание партий товаров.
* Генерация отчетов: Формирование отчетов по складу и перемещениям товаров.

Логика реализована в контроллерах, таких как *productController.js, userController.js, reportController.js*.

Уровень интерфейсных адаптеров (*Interface Adapters*)

Этот уровень обеспечивает взаимодействие между прикладной логикой и внешними интерфейсами:

* Контроллеры *Express*: Обработка *HTTP*-запросов, валидация данных, вызов бизнес-логики.
* Маршруты: Определены в файлах *productRoute.js, userRoute.js, reportsRoute.js* и др.
* *Middleware: authMiddleware.js* для проверки *JWT* и авторизации пользователей.

Взаимодействие с клиентом осуществляется через *REST API*, обрабатывающее *JSON*-запросы и ответы.

Инфраструктурный уровень (*Frameworks & Drivers*)

На этом уровне используются внешние библиотеки и фреймворки:

*Backend*:

* + *Node.js* и *Express.js* для создания сервера и маршрутизации.
  + *MongoDB* с *Mongoose* для хранения и управления данными.
  + *JWT* для аутентификации и авторизации.

*Frontend*:

* + *React* для построения пользовательского интерфейса.
  + *React Context API* и кастомные хуки для управления состоянием.
  + *Webpack* и *Babel* для сборки и транспиляции кода.

Прочее:

* + *dotenv* для управления конфигурацией.
  + *cors, helmet* для обеспечения безопасности *HTTP*-запросов.
  + *bcryptjs, jsonwebtoken* для обеспечения безопасности данных.

Использование *Domain-Driven Design (DDD)*

Принципы *DDD* реализованы через четкое разделение предметной области и инфраструктурных компонентов:

* Бизнес-логика сосредоточена в сущностях и контроллерах.
* Контроллеры и маршруты не содержат бизнес-правил, а лишь делегируют их выполнение соответствующим модулям.

Реализация принципов *CQRS*

В проекте применены элементы *Command Query Responsibility Segregation (CQRS)*:

* Команды (*Commands*): Операции, изменяющие состояние системы, такие как создание, обновление и удаление данных.
* Запросы (*Queries*): Операции, извлекающие данные без изменения состояния системы.

Команды и запросы реализованы через отдельные маршруты и методы *HTTP*, обеспечивая четкое разделение операций чтения и записи.

# **2 ПРОЕКТИРОВАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА ПС**

Логика действий пользователя в программном средстве. Чтобы лучше понять, как пользователи будут взаимодействовать с программным средством, была создана диаграмма *User-flow*, которая отражает основные этапы действий пользователя. Эта диаграмма позволяет визуально представить последовательность шагов, которые пользователь выполняет при использовании программы.

Пользователь с ролью «Складской работник» после входа в аккаунт попадает на главную страницу системы. С этого экрана он может перейти в разделы, обеспечивающие выполнение ключевых бизнес-процессов:

Управление товарами – добавление, редактирование и удаление товарных позиций.

На рисунке 2.1 также графически выделены крупные процессы: вход в личный кабинет, управление товарами, а также аналитика и отчётность. Эти процессы отражают целевую логику бизнес-задач, решаемых экономистом при работе с системой.

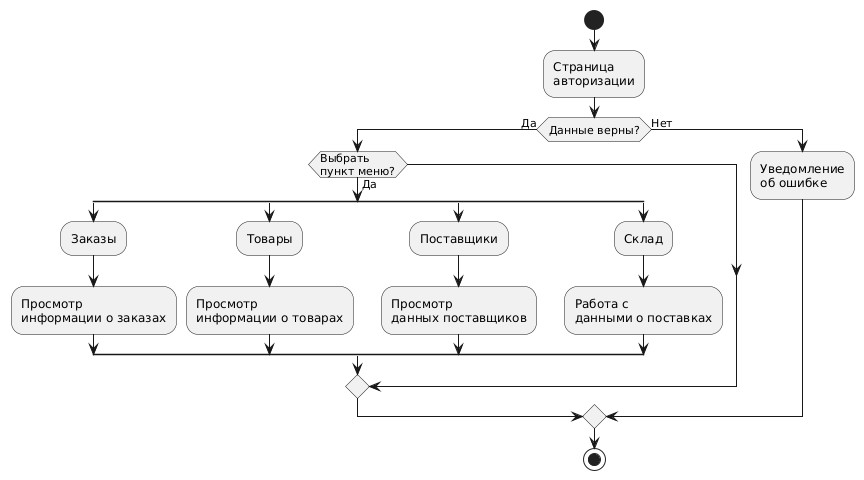


Рисунок 2.1 – *User-flow* диаграмма логики действий пользователя «Складской работник»

Пользователь с ролью «Менеджер» после успешного входа в систему попадает на главную страницу, с которой получает доступ к основным функциям, необходимым для выполнения его обязанностей:

Просмотр статистики – формирование и просмотр отчётов по поставкам, выполненным ранее.

Результат представлен на рисунке 2.2

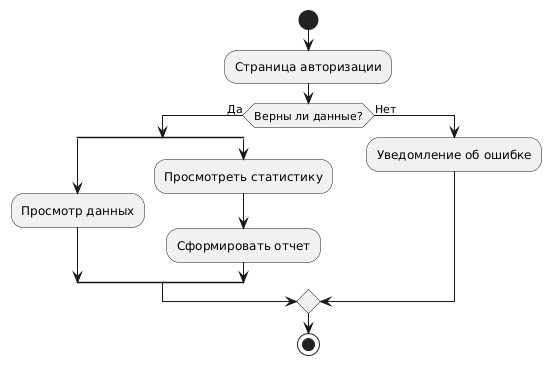


Рисунок 2.2 – *User-flow* диаграмма логики действий пользователя «Менеджер»

Интерфейс менеджера упрощён по сравнению с ролью сотрудника: отсутствует возможность редактирования продукции, поскольку эта функциональность не входит в его компетенцию. Это повышает удобство использования системы для данной роли.

На схеме визуально выделены основные бизнес-процессы: вход в систему, выполнение расчётов, просмотр отчётности, что позволяет отразить реальную логику взаимодействия менеджера с системой.

# **3 РЕАЛИЗАЦИЯ КЛИЕНТСКОЙ ЧАСТИ ПС**

На данном этапе была реализована визуальная часть программной системы «Программное средство реализации оперативного управления поставками на основе *BI*-решений» в соответствии с архитектурой и функциональными требованиями, определёнными в предыдущих разделах. Интерфейс обеспечивает доступ ко всем ключевым функциям: авторизация, регистрация, управление поставками, просмотр сводных данных и статистики.

Для разработки пользовательского интерфейса использовались современные инструменты и технологии, представленные в таблице 3.1.

Таблица 3.1 – Технологии и инструменты

|  |  |
| --- | --- |
| Инструмент / Технология | Назначение |
| *React.js* | Библиотека для построения пользовательского интерфейса |
| *React Router* | Организация маршрутизации и навигации по страницам |
| *Bootstrap 5* | Стилизация элементов интерфейса и адаптивная вёрстка |
| *SCSS* | Расширенный синтаксис *CSS* для стилизации компонентов |
| *Axios* | Выполнение *HTTP*-запросов к *REST API* |
| *Jest + React Testing Library* | Модульное тестирование компонентов интерфейса |

Пользовательский интерфейс системы включает следующие основные компоненты:

* *LoginComponent* – форма входа пользователя;
* *RegisterComponent* – форма регистрации нового пользователя;
* *DashboardComponent* – панель управления (главная страница после авторизации);
* *ProductsComponent* – управление товарами;
* *PricingPoliciesComponent* – настройка ценовых политик;
* *CalculationsComponent* – расчёты и расчётные алгоритмы;
* *StatisticsComponent* – отображение аналитики поставок;
* *LayoutComponent* – каркас приложения с навигацией и общей структурой отображения.

Маршруты конфигурируются с использованием библиотеки *React Router*. После входа пользователь перенаправляется в *LayoutComponent*, внутри которого загружаются дочерние страницы (рисунок 3.1).



Рисунок 3.1 – Реализация маршрутизации

Стилизация компонентов реализована с применением *SCSS* и фреймворка *Bootstrap 5*. Для повышения визуальной читаемости интерфейса использованы таблицы, модальные окна и выпадающие списки, адаптированные под различные разрешения экрана.

Для проверки корректности создания компонентов и начального рендеринга используются модульные тесты. На рисунке 3.2 показан пример базового теста, проверяющего успешное создание главного компонента приложения.

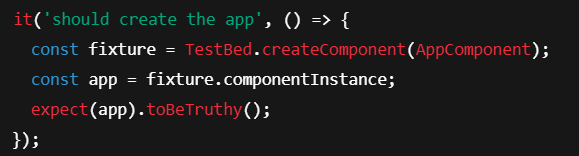


Рисунок 3.2 – Пример модульного теста

Таким образом, клиентская часть системы обеспечивает функциональную и адаптивную визуальную оболочку, соответствующую архитектурным требованиям проекта и современным подходам в веб-разработке.

Ра рисунке 3.3 продемонстрирован дизайн программы.

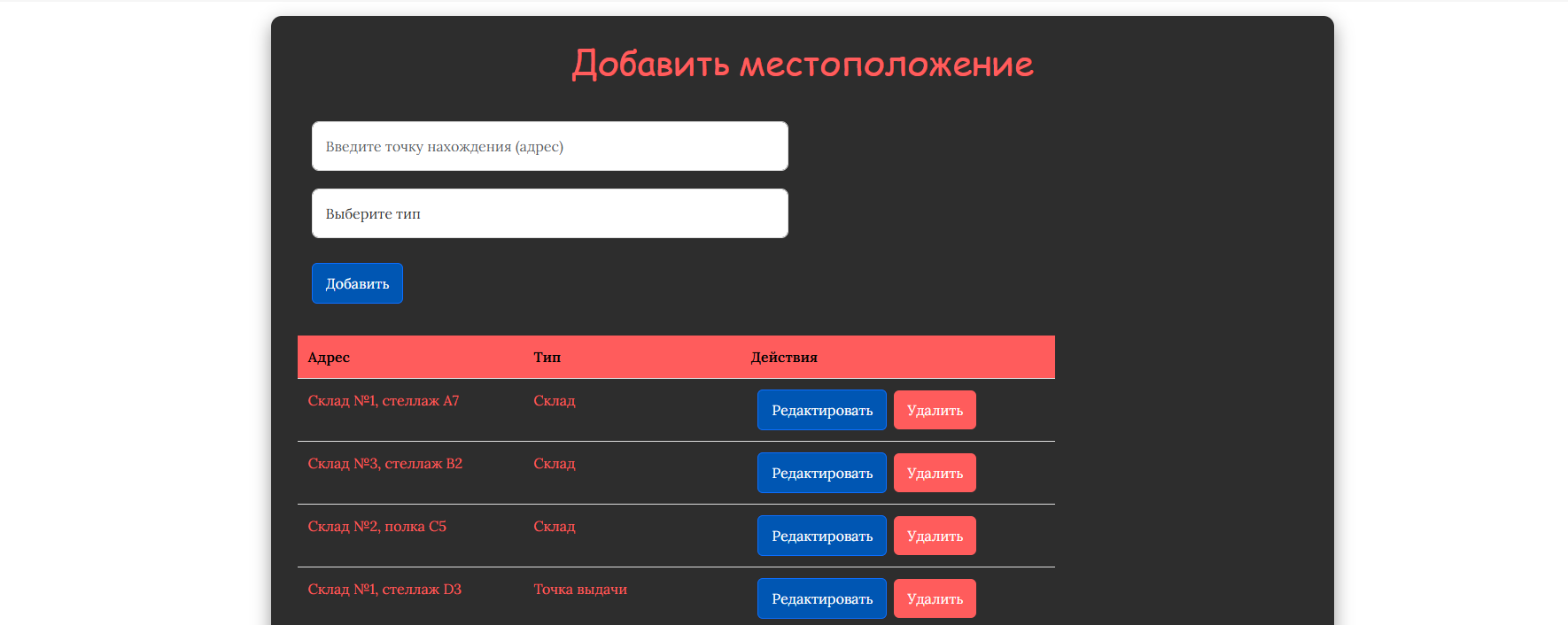


Рисунок 3.3 – Скриншот интерфейса

# **4 СПРОЕКТИРОВАТЬ СХЕМУ БД И ПРЕДСТАВИТЬ ОПИСАНИЕ ЕЕ СУЩНОСТЕЙ И ИХ АТРИБУТОВ**

На этапе физического проектирования была преобразована логическая модель данных в физическую структуру базы данных.

Схема представлена на рисунке 4.1.

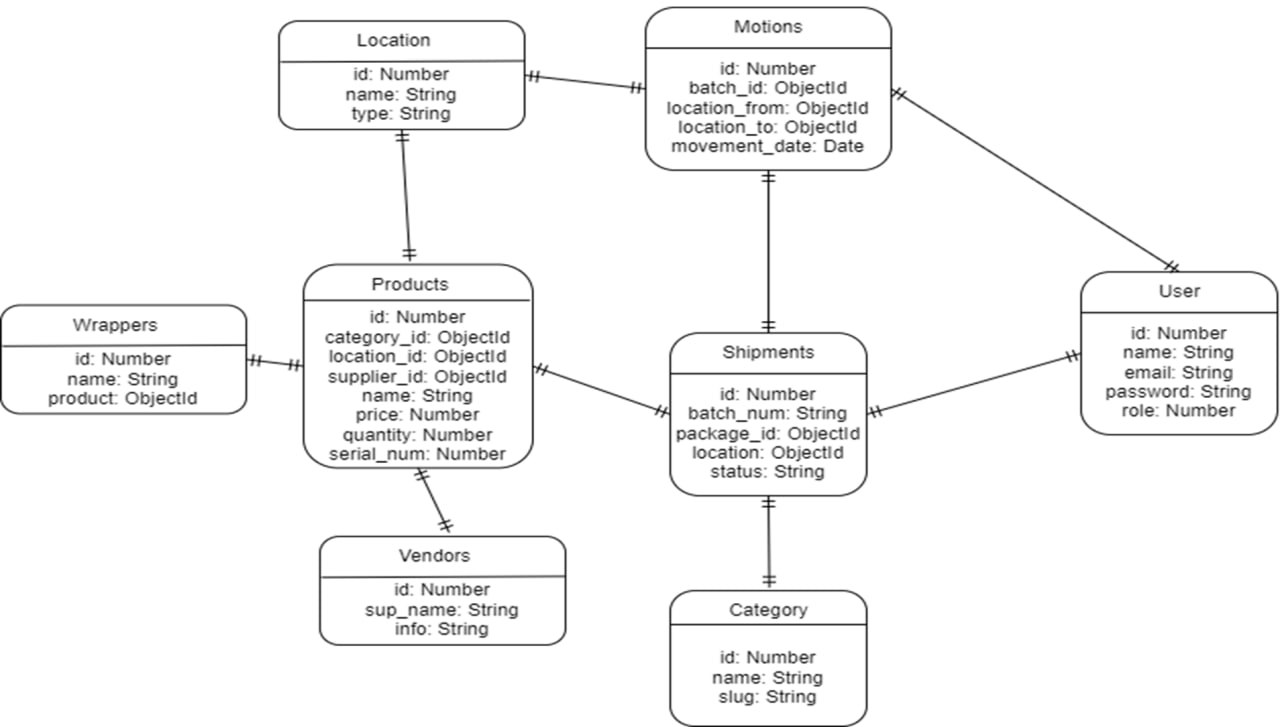


Рисунок 4.1 – Физическая схема БД

Текстовое описание сущностей базы данных представлено в виде таблицы 4.1.

Таблица 4.1 – Описание сущностей БД

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование поля | Назначение атрибута | Тип данных | Примечание |
| *User* (пользователь) | | | |
| *id* | Номер пользователя | Целое число типа *INT* | Первичный ключ |
| *name* | Имя пользователя | Строковое значение типа *String* |  |
| *email* | Адрес электронной почты | Строковое значение типа *String* | Уникальное |
| *password* | Пароль | Строковое значение типа *String* |  |
| *role* | Роль | Целое число типа *INT* | По умолчанию 0 |
| *createdAt* | Дата создания | *Date* |  |
| *updatedAt* | Дата обновления | *Date* |  |

Продолжение таблицы 4.1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *Location* (местоположение) | | | | | | |
| *id* | Номер места | | Целое число типа *INT* | | Первичный ключ | |
| *address* | Адрес местоположения | | Строковое значение типа *String* | |  | |
| *type* | Тип местоположения | | Строковое значение типа *String* | |  | |
| *Category*(категория) | | | | | | |
| *id* | Номер категории | | Целое число типа *INT* | | Первичный ключ | |
| *name* | Название категории | | Строковое значение типа *String* | |  | |
| *slug* | Слаг категории | | Строковое значение типа *String* | |  | |
| *Batches*(Партии) | | | | | | |
| *batch\_num* | Номер партии | | Целое число типа *INT* | | Первичный ключ | |
| *package\_id* | Ссылка на упаковку | | *ObjectId* | |  | |
| *location* | Ссылка на местоположение | | *ObjectId* | |  | |
| *status* | Статус партии | | Строковое значение типа *String* | |  | |
| *Packages*(Упаковки) | | | | | | |
| *id* | Номер упаковки | | Целое число типа *INT* | | Первичный ключ | |
| *name* | Наименование упаковки | | Строковое значение типа *String* | |  | |
| *Movement* (Перемещение) | | | | | | | |
| *id* | | Номер перемещения | | Целое число типа *INT* | | Первичный ключ | |
| *batch\_id* | | Ссылка на партию | | *ObjectId* | |  | |
| *location\_from* | | Откуда перемещается партия | | *ObjectId* | |  | |
| *location\_to* | | Куда перемещается партия | | *ObjectId* | |  | |
| *movement\_date* | | Дата перемещения | | *Date* | |  | |

База данных приведена к третьей нормальной форме, т.к.:

* 1НФ: Все атрибуты атомарные.
* 2НФ: В таблицах с составным ключом (если бы был) все неключевые атрибуты зависят от всего ключа.
* 3НФ: Нет транзитивных зависимостей между неключевыми атрибутами.

Это обеспечивает:

* Отсутствие дублирования данных.
* Минимизацию избыточности.
* Простоту поддержки и масштабируемости.

Скрипт генерации базы данных.Используемая база данных для хранения данных в данном проекте – это *NoSQL* база данных, которая работает на основе хранения данных в виде документов. В этой модели данных каждая сущность (например, объект или запись) представляется в виде отдельного документа. Каждый документ содержит информацию о соответствующей сущности, включая все ее атрибуты и значения.

Особенности автоматической генерации базы данных в используемом фреймворке заключаются в том, что он использует встроенные инструменты *ORM* (*Object-Relational Mapping*) для создания схемы базы данных на основе определенных моделей данных. Это означает, что вы определяете структуру данных в виде моделей или классов в вашем коде, а затем фреймворк автоматически создает или обновляет соответствующую структуру базы данных при запуске приложения.

Таким образом, при добавлении новых моделей или изменении существующих моделей в приложении, фреймворк автоматически адаптирует схему базы данных, чтобы отражать эти изменения. Это обеспечивает удобство и эффективность разработки, позволяя разработчикам сосредотачиваться на логике приложения, не беспокоясь о подробностях создания и поддержки структуры базы данных.

# **5 ПРЕДСТАВИТЬ ДЕТАЛИ РЕАЛИЗАЦИИ ПС ЧЕРЕЗ UML-ДИАГРАММЫ**

## **5.1 Описание статических аспектов программных объектов.**

Диаграмма классов представляет собой структуру сущностей и их взаимосвязей, использующихся в проекте.

Диаграмма иллюстрирует связи между этими сущностями, например, пользователь может иметь несколько расчётов, каждый продукт может быть связан с несколькими расчётами, а также могут быть привязаны различные ценовые политики для каждого продукта. Диаграмма представлена на рисунке 5.1.

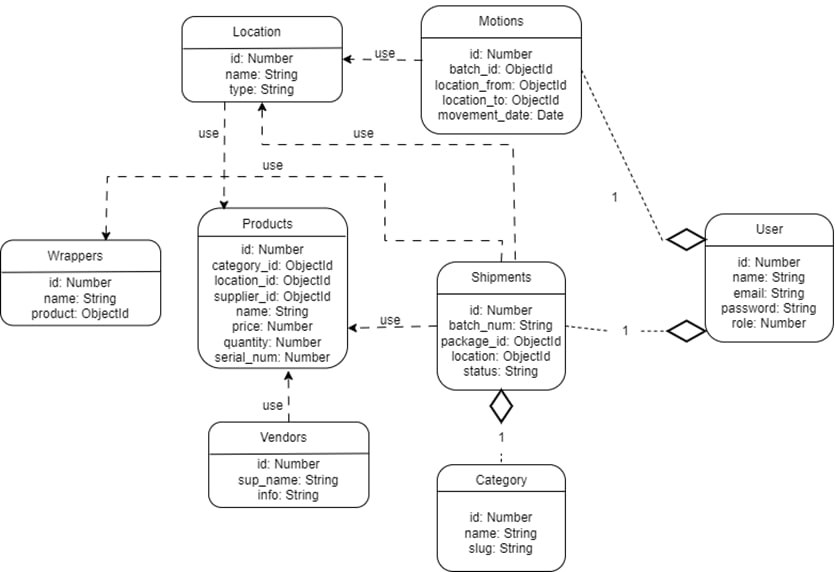


Рисунок 5.1 – Диаграмма классов

На изображении представлена диаграмма, моделирующая данные системы автоматизации ключевых операций складского учета. В системе шесть основных сущностей и связи между ними:

1. *Category* (категория). Одна категория может быть связана с несколькими партиями (*Batches*), что указывает на потенциальную связь «один ко многим».
2. *Packages* (упаковки). Упаковка может быть связана с несколькими партиями (*Batches*), что также указывает на связь «один ко многим».
3. *Batches* (партии). Партия связана с одной упаковкой (*Packages*) и одним местоположением (*Location*). Также связь с *Category* указывает на то, что партия может принадлежать одной категории товаров.
4. *Location* (местоположение). Местоположение используется в нескольких партиях (*Batches*) и в перемещениях (*Movement*), что позволяет отслеживать, где хранится или куда перемещается конкретная партия.
5. *Movement* (перемещение). Перемещение связано с одной партией (*Batches*) и двумя местоположениями (*Location*) для указания начального и конечного пунктов перемещения.
6. *User* (пользователь). Пользователи могут управлять данными других сущностей (например, партиями и перемещениями), хотя на диаграмме прямые связи не показаны. Роль указывает на возможность разграничения прав доступа.

Для физического представления системы была построена диаграмма компонентов. Данная диаграмма позволяет показать архитектуру системы в целом, а также зависимость между программными компонентами. Основные графические элементы данной диаграммы – это компоненты и интерфейсы, а также зависимости между ними. Диаграмма компонентов представлена на рисунке 5.2.

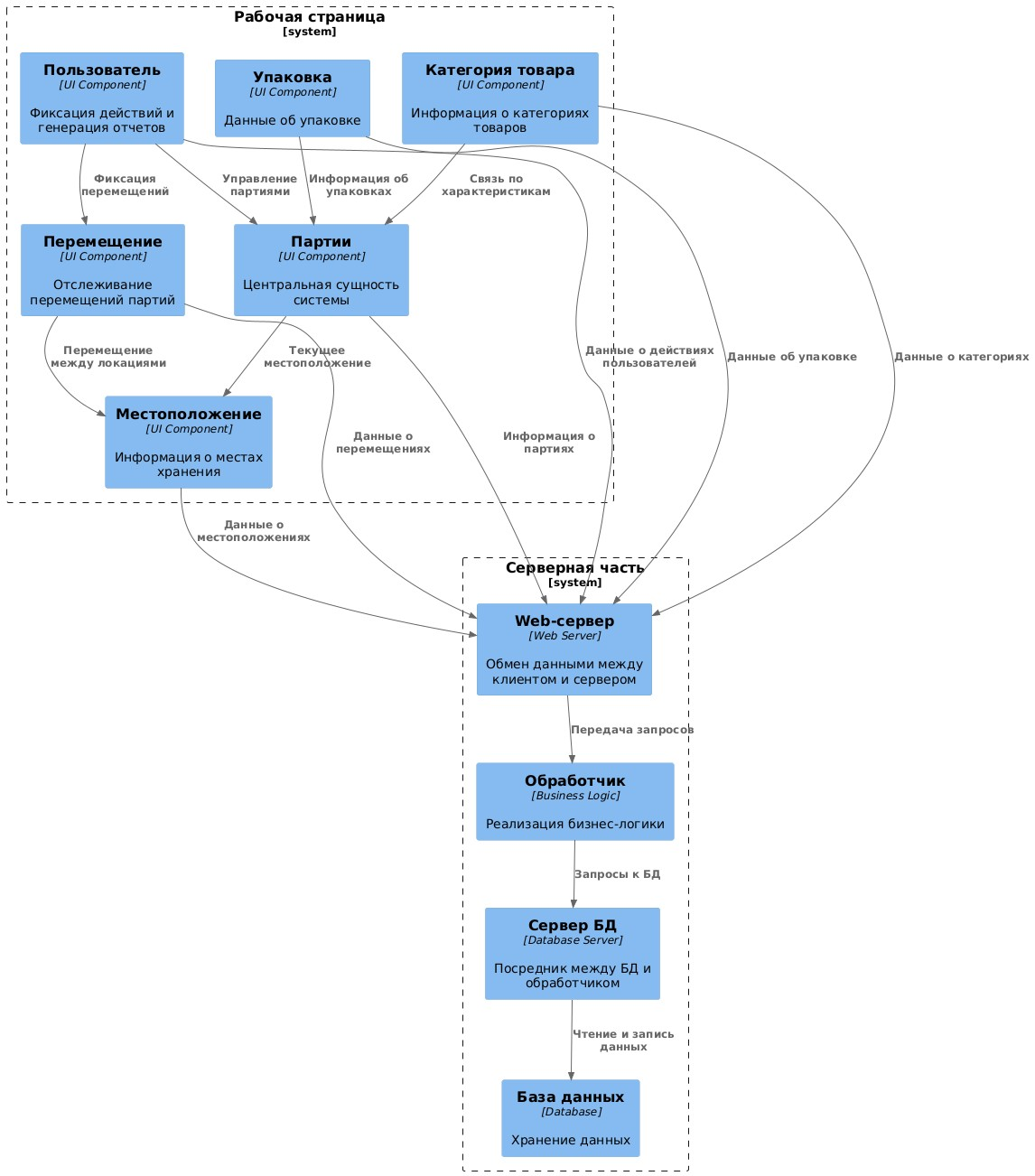


Рисунок 5.2 – Диаграмма компонентов

Эта диаграмма развертывания показывает взаимодействие между двумя устройствами: *Windows PC* и *DB Server*. *Windows PC* использует *Node.js* для выполнения различных *JS*-файлов, а *DB Server* на *MongoDB* хранит схемы этих данных. Это отражает структуру и связи между различными компонентами системы. Результаты представлены на рисунке 5.3.

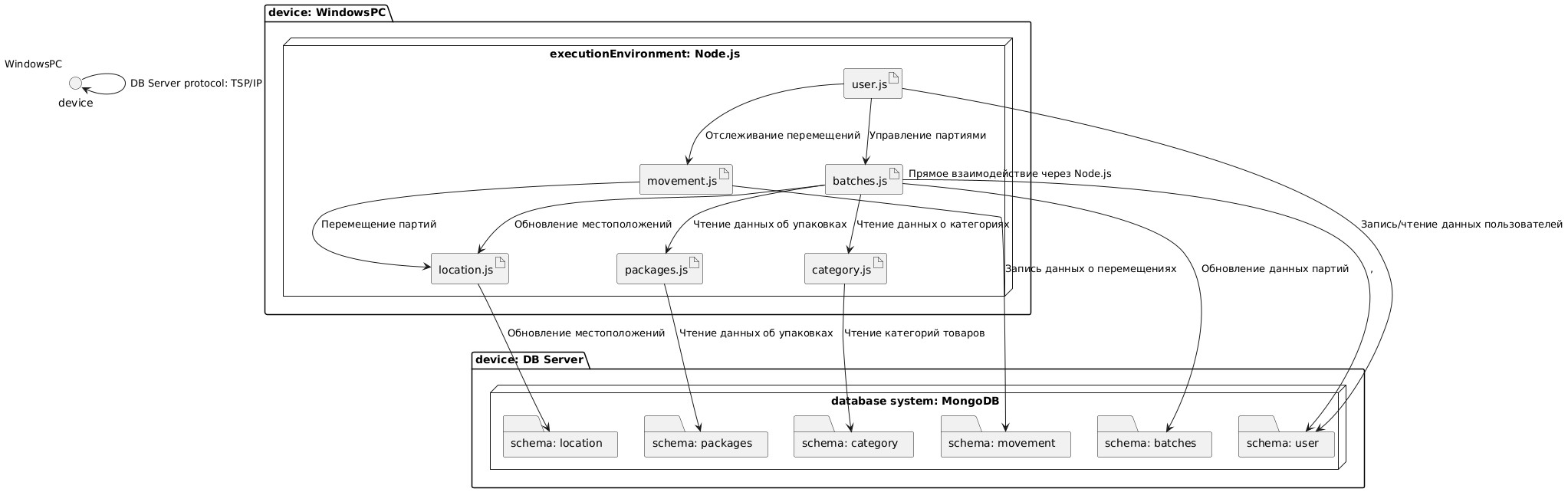


Рисунок 5.3 – Диаграмма размещения

## **5.2 Описание динамических аспектов поведения программных объектов**

На диаграмме деятельности, представленной на рисунке 5.4, отображён процесс взаимодействия пользователя с системой при выполнении варианта использования «Принимать и отгружать поставку». Диаграмма отражает последовательность шагов. Этот процесс иллюстрирует логику бизнес-функциональности системы и показывает, как система обеспечивает ведение базы поставок организации.

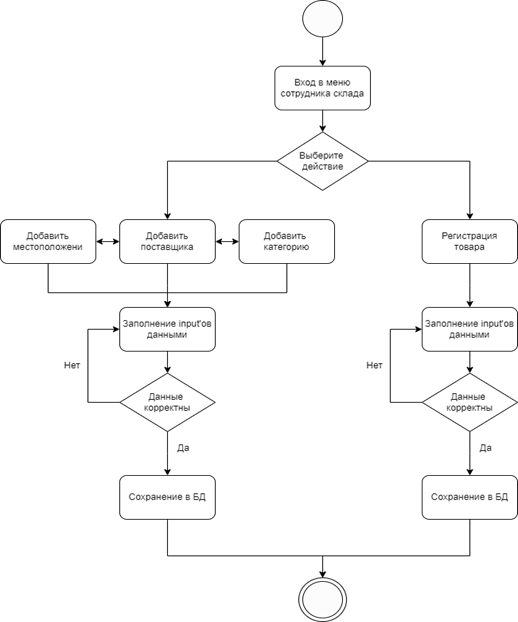


Рисунок 5.4 – Диаграмма деятельности Варианта использования «Принимать и отгружать поставку»

На Диаграмме последовательности, для класса *createInvoice*, был разработан следующий алгоритм:

1 Начинается выполнение класса *createInvoice*.

2 Далее происходит проверка наличия всех обязательных полей. Если какое-либо поле отсутствует, функция возвращает ошибку клиенту.

3 Создание новой записи *invoiceModel*:

После успешной проверки данных, создается новый объект с переданными данными (*numberTTN, date, shipper, nameOfShipper, nameOfConsignee, productref*).

4 Сохранение записи в базе данных. Созданный объект *newInvoice* сохраняется в базе данных с помощью метода *save*().

5 Завершение функции. Функция успешно завершается, и клиент получает подтверждение о создании новой записи.

Диаграмма представлена на рисунке 5.5

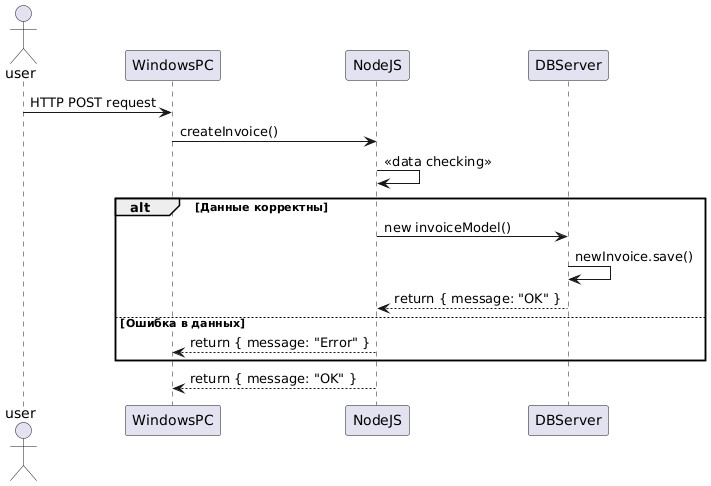


Рисунок 5.5 – Диаграмма последовательности

Диаграмма состояния. Выполним расчёт графика поставок. Пользователь выбирает поставщика, указывает период. Диаграмма представлена на рисунке 5.6.

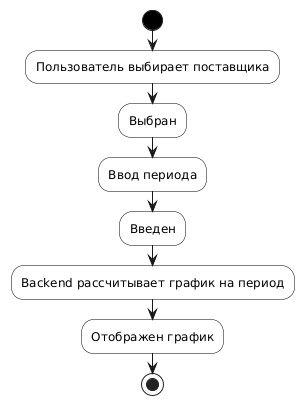


Рисунок 5.6 – Диаграмма состояния

* Начальное состояние — выбор пользователем поставщика.
* Затем он вводит период.
* После отправки данных сервер рассчитывает диаграмму и отображает его пользователю.

# **6 ДОКУМЕНТАЦИЯ К ПС С OPEN API**

**6.1 Реализация серверной части программной системы**

Серверная часть программной системы реализована на платформе *NodeJS* с использованием шаблона *MVC*. Проект следует принципам раздельной ответственности: контроллеры обрабатывают *HTTP*-запросы, сервисы инкапсулируют бизнес-логику, а модели описывают структуру данных. Архитектура построена на *REST*-подходе, данные передаются в формате *JSON*.

Для хранения данных используется база *MongoDB*, взаимодействие с которой осуществляется через драйвер *MongoDB.Driver*. Использована репозиторная архитектура с асинхронными методами (*async/await*) для операций с коллекциями базы данных.

Реализованные контроллеры:

* *AuthController* - регистрация и аутентификация пользователей. Пароли хранятся в хешированном виде. Используется *cookie*-аутентификация на основе *ClaimsPrincipal*.
* *SuppliesController* - операции с поставками (создание, получение, удаление, обновление).
* *StatisticsController* - сбор и возврат агрегированных статистических данных.
* *ProductsController* - работа с товарами (поиск, добавление, удаление).
* *SuppliersController* - управление справочником поставщиков.

Все маршруты построены с использованием *REST*-методов (*GET*, *POST, PUT, DELETE*), а ответы и запросы оформлены в формате *JSON*. Доступ к *API* возможен только для авторизованных пользователей. Каждый пользователь получает доступ к данным в соответствии со своей ролью.

Пример запроса на создание новой поставки:

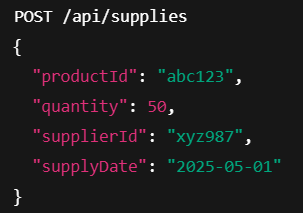


Рисунок 6.1 – Cоздание новой поставки

Пример ответа:

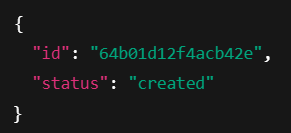


Рисунок 6.2 – Ответ

Таким образом, взаимодействие между frontend и backend частями системы осуществляется полностью через API-интерфейс, работающий по HTTP с JSON-сообщениями.

**6.2 Документация к API (в формате JSON)**

В проекте не используется *OpenAPI*-спецификация, однако структура *API* документирована в исходных файлах, а все взаимодействия реализованы по *REST*-принципам. Ниже представлены примеры ключевых эндпоинтов:

* *POST /api/auth/register* - регистрация нового пользователя
* *POST /api/auth/login* - вход в систему
* *GET /api/products* - получить список всех товаров
* *POST /api/suppliers* - добавить поставщика
* *POST /api/supplies* - зарегистрировать новую поставку
* *GET /api/statistics* - получить аналитические данные
* *DELETE /api/supplies/{id}* - удалить поставку по *ID*

*API* использует *cookie*-аутентификацию. Каждый ответ содержит информацию в формате *JSON*. Для ручного тестирования использовались инструменты *Postman* и *curl*.

**6.3 Метрики качества кода**

Код серверной части проекта был проанализирован с помощью встроенных средств *Visual Studio* и плагинов *JetBrains Rider*. Основные метрики представлены в таблице 6.1.

Таблица 6.1 – Метрики качества кода

|  |  |
| --- | --- |
| Метрика | Описание |
| *Cyclomatic Complexity* | Оценка логической сложности методов |
| *Maintainability Index* | Индекс удобства сопровождения |
| *Lines of Code (LOC)* | Количество строк кода |
| *Code Coverage* | Покрытие кода модульными тестами |
| *Number of Code Smells* | Потенциально проблемные участки кода |

**6.4 Оценка качества кода ПС**

Анализ качества серверного кода дал следующие результаты:

Таблица 6.2 – Оценка качества кода ПС

|  |  |
| --- | --- |
| Метрика | Значение |
| *Cyclomatic Complexity* | 1–4 (низкая сложность) |
| *Maintainability Index* | 85–100 (высокая поддерживаемость) |
| *Lines of Code* | ~850 |
| *Code Coverage* | 75% (*unit* + интеграционные) |
| *Code Smells* | 1 предупреждение, 0 критических |

Эти показатели подтверждают, что код проекта является структурированным, легко поддерживаемым и пригодным к расширению. Модули разделены по ответственности, соблюдены принципы *SOLID*. Покрытие тестами выше среднего: протестированы основные контроллеры, сервисы и авторизация.

.

# **7 РЕАЛИЗАЦИЯ СИСТЕМЫ АУТЕНТИФИКАЦИИ И АВТОРИЗАЦИИ ПОЛЬЗОВАТЕЛЕЙ ПС И МЕХАНИЗМОВ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ДАННЫХ**

Для реализации механизма аутентификации и авторизации в программной системе использованы технологии, представленные в таблице 7.1.

Таблица 7.1 – Используемые технологии

|  |  |
| --- | --- |
| Технология / Библиотека | Назначение |
| *Node.js (Express)* | Обработка *HTTP*-запросов, реализация *REST API* |
| *MongoDB + mongoose* | Хранение и управление данными пользователей |
| *Crypto (SHA256)* | Хеширование паролей |
| *React.js* | Пользовательский интерфейс |
| *Axios* | Обращение к *REST API* с клиента |

Аутентификация и авторизация реализованы через следующие ключевые компоненты:

* Контроллер *auth.routes.js* - обрабатывает регистрацию и вход пользователя. Используется модуль crypto для хеширования пароля при создании и проверке учётной записи.
* Контроллеры *supplies.routes.js, statistics.routes.js, products.routes.js* - защищены авторизацией: доступ разрешён только при наличии действительных пользовательских данных.
* React-компоненты *Login.jsx, Register.jsx, Dashboard.jsx, Layout.jsx* - реализуют логику входа, регистрации и защиты маршрутов в интерфейсе.
* Маршруты защищаются на *frontend'е* путём проверки авторизации: при отсутствии авторизационных данных (*userId, role*) пользователь перенаправляется на страницу входа.

Пример *backend*-кода (*Node.js*), реализующего вход пользователя, представлен на рисунке 7.1:

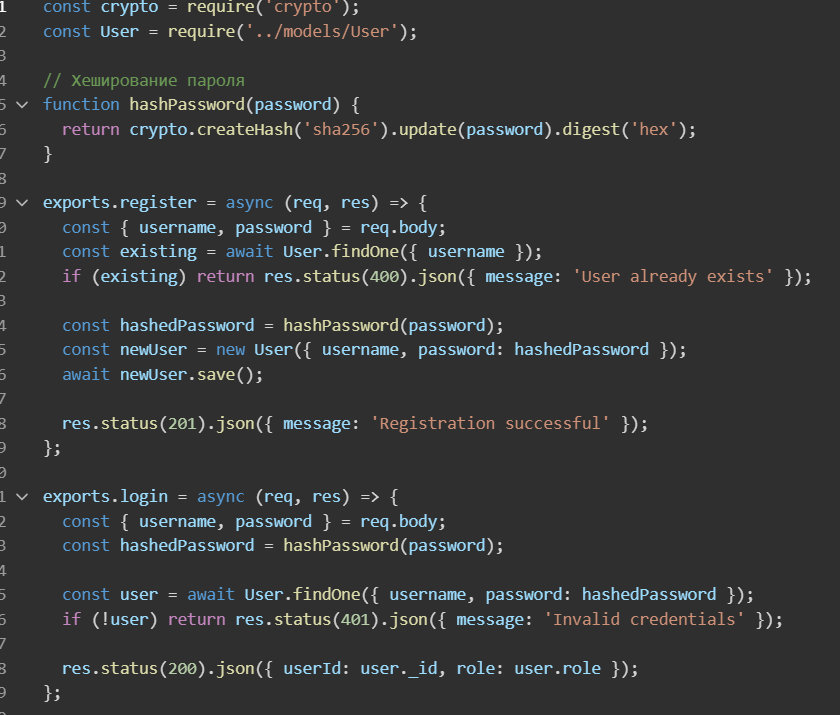


Рисунок 7.1 – Контроллер *AuthController (Node.js)*

Метод хеширования пароля, реализованный через *crypto.createHash('sha256')*, приведён на рисунке 7.2:

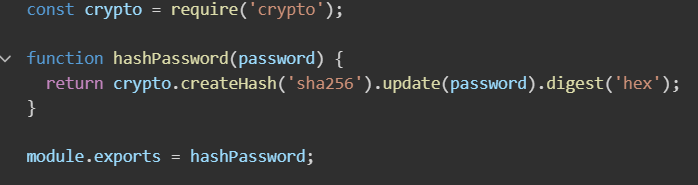


Рисунок 7.2 – Метод хеширования пароля

Пример *frontend*-кода (*React*), реализующего форму входа и сохранение данных сессии, приведён на рисунке 7.3:

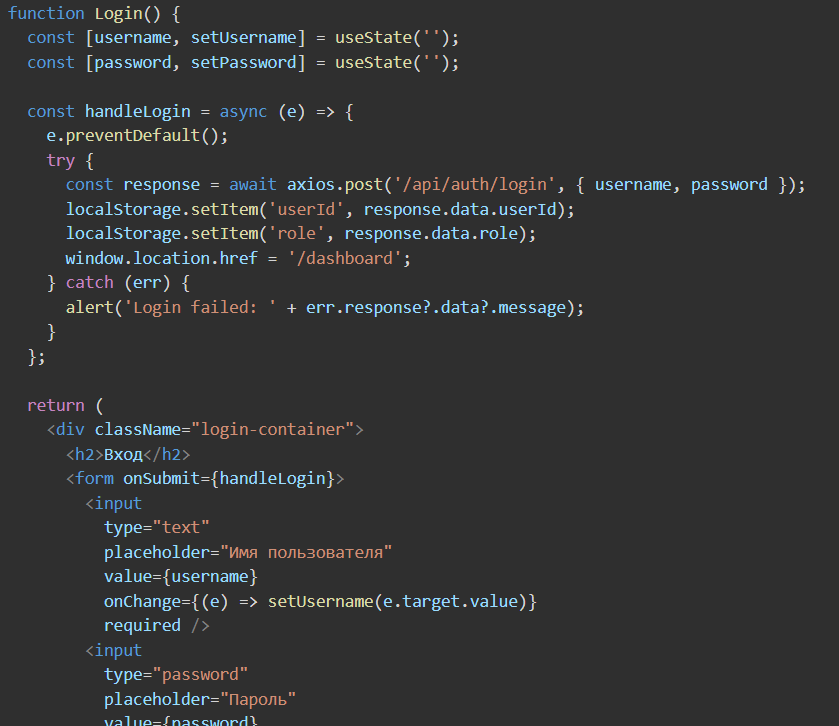


Рисунок 7.3 – Форма логина (*login.jsx*)

В результате внедрения системы авторизации были реализованы следующие изменения:

* Созданы маршруты *POST /api/auth/register и POST /api/auth/login* с сохранением пользователей в *MongoDB*.
* Введены модели *User, LoginRequest, RegisterRequest* для обработки и хранения данных.
* Все защищённые маршруты *backend*-а проверяют наличие валидного userId.
* На клиенте авторизация проверяется в *localStorage* и используется для ограничения доступа к интерфейсу.
* Предусмотрена возможность в будущем внедрить токены (например, *JWT*) и перейти на *httpOnly*-куки для повышения уровня безопасности.

Таблица 7.2 – Механизмы безопасности данных

|  |  |
| --- | --- |
| Механизм | Описание |
| Хеширование паролей (*SHA256*) | Пароли хранятся в базе в виде безопасных хешей |
| Разграничение доступа | Только авторизованные пользователи имеют доступ к защищённым маршрутам |
| *Client-side* защита | *React*-приложение проверяет авторизацию через *localStorage*, реализует переадресацию |
| Валидация данных | Проверка на стороне сервера и частично на клиенте |
| Возможность масштабирования | В будущем возможно расширение на *JWT* или *OAuth* |

Таким образом, система авторизации реализована на базе современных и проверенных решений: *Node.js + MongoDB + React*. Это обеспечивает необходимую гибкость и безопасность для дальнейшего развития проекта.

# **8 UNIT- И ИНТЕГРАЦИОННЫЕ ТЕСТЫ**

*Unit*-тестирование (модульное тестирование) - это методика, при которой тестируется отдельная, изолированная часть программы (обычно один метод, функция или компонент). Цель - убедиться, что логика работы конкретного элемента корректна вне зависимости от других компонентов системы.

Основные характеристики *unit*-тестов:

* Тестируется один метод или модуль;
* Внешние зависимости (например, базы данных или *API*) мокаются;
* Выполняются быстро и независимо;
* Используются фреймворки: *Jest* (для *Node.js* и *React*), *React Testing Library* (для тестирования интерфейса компонентов).

Пример *backend*-функции для расчёта итоговой стоимости поставки представлен на рисунке 8.1.

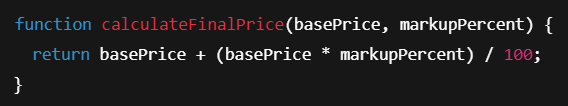


Рисунок 8.1 – Пример тестируемой функции *calculateFinalPrice*

Эта функция реализует бизнес-логику: расчёт финальной цены товара с учётом наценки. Например, при *basePrice* = 100 и *markupPercent* = 20, результат будет 120.

*Unit*-тест к этой функции, написанный на *Jest*, показан на рисунке 8.2.

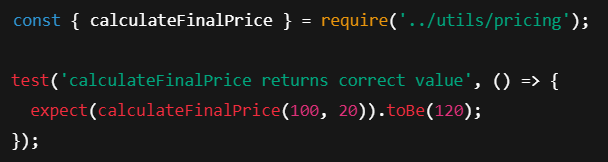


Рисунок 8.2 – Unit-тест функции *calculateFinalPrice*

Файл *unit*-теста главного компонента *React* показан на рисунке 8.3.

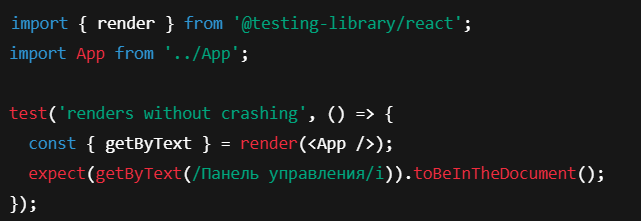


Рисунок 8.3 – Базовый unit-тест компонента *App.jsx*

Интеграционные тесты используются для проверки взаимодействия нескольких компонентов одновременно (например, маршрутов, сервисов и базы данных).

На рисунке 8.4 приведён пример интеграционного теста контроллера */api/products*.



Рисунок 8.4 – Интеграционный тест контроллера *products.routes.js*

Такой тест запускает сервер *Express*, отправляет *HTTP*-запрос и проверяет, что контроллер возвращает ожидаемый статус и данные.

Тест-кейсы для проверки уровня базовых пользовательских требований приведены в таблице 8.1.

Таблица 8.1 – Тест-кейсы для проверки уровня базовых пользовательских   
 требований

|  |  |  |  |
| --- | --- | --- | --- |
| Иденти-фикатор тест-кейса | Заглавие  тест-кейса | Шаги тест-кейса | Ожидаемый результат |
| *UC*-1 | Мониторинг данных товаров | 1. Войти в систему под учетной записью Контролера качества. 2. Перейти в раздел «Аналитика». 3. Просмотреть данные по номеру партии. | 1. Вход в систему с ролью «Менеджер». 2. Переход на форму с аналитикой 3. Пользователь видит все необходимые данные. |
| *UC*-2 | Регистрация поставок | 1. Войти в систему под учетной записью сотрудника склада. 2. Открыть раздел добавления новой поставки. 3. Ввести товар с накладной. 4. Оформить приемку товара в системе. | 1. Вход в систему по аккаунтом с ролью «Сотрудник склада». 2. Переход на страницу товаров. 3. Добавление или взятие из базы товаров. 4. Сохранение накладной в базу |
| *UC*-3 | Подготовка товаров к отгрузке | 1. Войти в систему под учетной записью логиста. 2. Перейти в раздел заказов и поставок. 3. Проанализировать существующие заказы и поставки. 4. Внести корректировки по необходимости. | 1. Вход в систему с ролью «Логист». 2. Переход на страницу с отгрузкой товара. 3. Просмотр текущих отгрузок 4. Изменение отгрузок и сохранение результатов |

Продолжение таблицы 8.1

|  |  |  |  |
| --- | --- | --- | --- |
| Иденти-фикатор тест-кейса | Заглавие  тест-кейса | Шаги тест-кейса | Ожидаемый результат |
| *UC*-4 | Работа с товарами | 1. Перейти в раздел «Товары»  2. Убедиться, что отображается список добавленных товаров | Список товаров корректно отображается |
| *UC*-5 | Добавление товаров | 1. Нажать «Добавить товар»  2. Заполнить обязательные поля  3. Нажать «Сохранить» | Товар успешно добавлен, появляется в списке |
| *UC*-6 | Редактирование товаров | 1. Перейти к списку товаров  2. Нажать «Редактировать» у нужного товара  3. Внести изменения  4. Сохранить | Изменения сохранены, данные обновлены |
| *UC*-7 | Удаление товраов | 1. Перейти к списку товаров 2. Нажать «Удалить» у нужного товара 3. Подтвердить действие | Товар удалён из системы, список обновлён |

# **9 ОПИСАНИЕ ПРОЦЕССА РАЗВЕРТЫВАНИЯ ПС**

Для корректного запуска программной системы, реализованной на базе *Node.js (Express.js)* и *React.js*, необходимо выполнить пошаговую настройку как клиентской, так и серверной части.

1 Подготовка среды для запуска проекта

Установка *Visual Studio Code*

Скачать последнюю версию редактора *Visual Studio Code* с официального сайта: [*https://code.visualstudio.com/*](https://code.visualstudio.com/)

Установка *Node.js*

Скачать *Node.js* версии 20.11.0 (или новее) с официального сайта: [*https://nodejs.org/*](https://nodejs.org/)

Убедиться, что команды *node -v* и *npm -v* работают в терминале.

Установка *MongoDB*

Установить *MongoDB Community Edition* с сайта: [*https://www.mongodb.com/try/download/community*](https://www.mongodb.com/try/download/community)

После установки убедиться, что *MongoDB* успешно запущена (например, через *mongosh* или *MongoDB Compass*).

2 Запуск клиентской части (*frontend*)

Открыть директорию проекта *frontend* во *Visual Studio Code*.

Открыть встроенный терминал сочетанием клавиш *Ctrl + Shift +* Ё (или *Ctrl + `* ).

Выполнить следующие команды:

*cd frontend*

*npm install*

*npm start*

После сборки *React*-приложение будет доступно по адресу: [*http://localhost:3000*](http://localhost:3000)

3 Запуск серверной части (*backend*)

Открыть директорию *backend* во *Visual Studio Code*.

Создать файл .*env* с переменными подключения к *MongoDB* и порту сервера. Пример:

*ini*

*PORT=5000*

*MONGO\_URI=mongodb://localhost:27017/supply\_db*

Установить зависимости и запустить сервер:

*cd backend*

*npm install*

*npm run dev*

Сервер будет доступен по адресу: [*http://localhost:5000*](http://localhost:5000)

4 Авторизация в системе

* Администратор может быть добавлен вручную в базу данных *MongoDB*.
* Обычные пользователи регистрируются через интерфейс клиентской части (форма /*register*).
* После входа данные пользователя сохраняются в *localStorage* браузера.

5 Запуск *Unit*-тестов

Тесты написаны с использованием фреймворка *Jest* (как для *frontend*, так и для *backend*).

Запуск *backend*-тестов

*cd backend*

*npm run test*

Запуск *frontend*-тестов

*cd frontend*

*npm run test*

5.3 Результаты тестов

* Успешный тест отображается с зелёной галочкой.
* Неуспешный - с красным крестиком и описанием ошибки.
* Все тесты выполняются изолированно, без необходимости подключения к реальной БД (при помощи моков или in-memory *data*).

# **10 РАЗРАБОТКА РУКОВОДСТВА ПОЛЬЗОВАТЕЛЯ**

В рамках проектирования и разработки программной системы были определены ключевые функции и задачи, обеспечивающие оперативное управление поставками, автоматизацию расчётов и формирование аналитической отчётности. Интерфейс и логика системы разработаны с учётом удобства работы как для обычных пользователей, так и для администраторов.

Таблица 10.1 – Описание функций и задач

|  |  |  |
| --- | --- | --- |
| Функции | Задачи | Описание |
| Управление пользователями | Регистрация, вход в систему | Обеспечивает доступ к системе. Поддерживается разграничение ролей (пользователь/администратор), идентификация по логину и паролю. |
| Управление поставками | Добавление, редактирование, удаление | Администратор управляет поставками: выбирает товар, указывает поставщика, дату, объём и цену поставки. |
| Управление товарами | Обновление данных о товарах | Поддерживается добавление новых товаров, изменение их характеристик, а также удаление устаревших позиций. |
| Статистика | Просмотр графиков и аналитики | Пользователь может просматривать сводные данные по поставкам за выбранный период: графики, количество и объёмы. |
| Расчёт себестоимости | Автоматизированный расчет | Вычисление себестоимости с учётом количества, стоимости, поставщика и других параметров. |
| История операций | Просмотр прошлых поставок и расчетов | Отображает ранее внесённые данные, поддерживает повторный расчёт по сохранённым данным. |
| Генерация отчётов | Формирование отчёта по поставкам | Автоматическое создание отчёта по стоимости и объёмам поставок за указанный период с возможностью экспорта. |

В таблице 10.2 описаны операции обработки данных для задач.

Таблица 10.2 – Описание реализуемых операций

|  |  |
| --- | --- |
| Операция | Описание |
| Регистрация пользователя | Создание новой учётной записи с базовой ролью «пользователь». |
| Авторизация | Проверка логина и пароля, доступ к защищённым разделам при успешном входе. |
| Добавление поставки | Заполнение формы с указанием товара, даты, количества и поставщика. |
| Редактирование поставки | Внесение изменений в данные о ранее созданной поставке. |
| Удаление поставки | Удаление записи о поставке из системы (доступно только администратору). |
| Добавление товара | Заполнение характеристик нового товара (наименование, упаковка, единица измерения). |
| Обновление информации о товаре | Изменение характеристик существующего товара. |
| Просмотр аналитики | Отображение диаграмм по количеству и объёму поставок за выбранный период. |
| Автоматический расчёт себестоимости | Расчёт стоимости на основании введённых данных и параметров поставок. |
| Генерация отчёта | Формирование структурированного документа с итоговыми значениями (может быть выгружен в PDF). |

Разработанная программная система включает в себя все необходимые функции для эффективного управления поставками и аналитического контроля. Пользователи могут регистрироваться, выполнять авторизацию и работать с системой в соответствии с назначенными ролями. Визуальные компоненты, аналитические диаграммы и отчёты помогают принимать обоснованные управленческие решения.

Функциональность проекта охватывает ключевые процессы: от ввода данных до их анализа и отчётности. Благодаря автоматизации расчётов и истории операций, система снижает риск ошибок и экономит рабочее время пользователей. Она легко масштабируется и может быть расширена для применения на различных уровнях управления поставками.

**Вывод**

В рамках выполненной лабараторной работы была разработана комплексная программная система, предназначенная для оперативного управления поставками на основе BI-решений. Проект охватывает весь жизненный цикл разработки программного обеспечения — от анализа требований и проектирования архитектуры до реализации, тестирования и развертывания.

Ключевым достижением стало построение модульной архитектуры, соответствующей принципам *Clean Architecture* и *Domain-Driven Design*. Архитектура включает чёткое разделение на слои сущностей, бизнес-логики, интерфейсных адаптеров и инфраструктуры. Это позволило повысить гибкость, сопровождаемость и масштабируемость системы. Технологический стек проекта включает *Node.js, Express, MongoDB, React* и сопутствующие инструменты для стилизации, маршрутизации и тестирования.

Проект реализует полноценную систему авторизации и аутентификации, обеспечивая разграничение прав доступа между ролями пользователя. Это повышает уровень безопасности и гарантирует защиту персональных и бизнес-данных. Также внедрены механизмы безопасного хранения паролей, валидации данных и клиентской защиты интерфейса.

Интерфейс разработан с учётом современных требований *UX/UI*: система дизайна обеспечивает единый стиль и высокую удобочитаемость, а User Flow диаграммы позволяют легко понять взаимодействие различных ролей с программой. Реализация клиентской части на React с использованием маршрутизации, адаптивной верстки и модульного тестирования обеспечивает высокое качество визуального и функционального взаимодействия.

Для хранения и обработки информации была спроектирована и реализована нормализованная структура базы данных, приведённая к третьей нормальной форме. Применение *MongoDB* в сочетании с *Mongoose* обеспечило гибкость в представлении и масштабировании данных.

Особое внимание уделено качеству кода: проведены модульные и интеграционные тесты, выполнен анализ метрик (*Cyclomatic Complexity, Maintainability Index, Code Coverage*), которые показали высокую степень качества и сопровождаемости. Развёртывание системы сопровождается подробной инструкцией, включая настройку среды, установку зависимостей и запуск обеих частей проекта. Поддерживается автоматическое формирование схемы БД на основе моделей.

Система также включает документацию к *API*, а также разработанное пользовательское руководство, что значительно упрощает внедрение программного продукта в эксплуатацию. Описание функциональных ролей и операций, а также подготовленные тест-кейсы, подтверждают пригодность системы для использования в реальной среде.

Таким образом, выполненная работа представляет собой законченный программный продукт, способный эффективно решать задачи автоматизации процессов поставок, анализа данных и поддержки управленческих решений. Разработка отличается высоким уровнем технической реализации, методологической обоснованностью и практической применимостью. Полученные результаты полностью соответствуют поставленным целям и задачам и могут быть расширены или адаптированы под нужды конкретного предприятия.