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 _____ А.В. Шелест
(подпись)

зачтено _____
(дата защиты)

Выполнила _____ К.А. Отякова
(подпись) гр. 214371

Минск, 2025

СОДЕРЖАНИЕ

Ссылки на репозитории

https://github.com/Ksenchez/OtyakovaK.A_214371_RIOPK_Server.git
https://github.com/Ksenchez/OtyakovaK.A_214371_RIOPK_Front
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 анализа политики ценообразования и автоматизации расчётной методики. Участниками системы выступают экономист и менеджер по продажам, каждый из которых обладает определённым набором действий в рамках своих обязанностей.

Экономист имеет доступ к полному управлению товарной базой, включая добавление, редактирование и удаление товаров (через расширение варианта использования «Работа с товарами»), а также может управлять политиками ценообразования, добавляя новые ценовые правила. Ему также доступны расчёты цен (включающие учёт налогов и автоматический пересчёт) и просмотр статистики с возможностью формирования отчётов по себестоимости.

Менеджер по продажам имеет доступ к функционалу расчёта цен с учётом налогов и автоматического пересчёта, а также к просмотру статистики и формированию отчётов по себестоимости. Таким образом, менеджер ориентирован на аналитическую и прогностическую работу с ценами, не вмешиваясь в политику и структуру товарной базы.

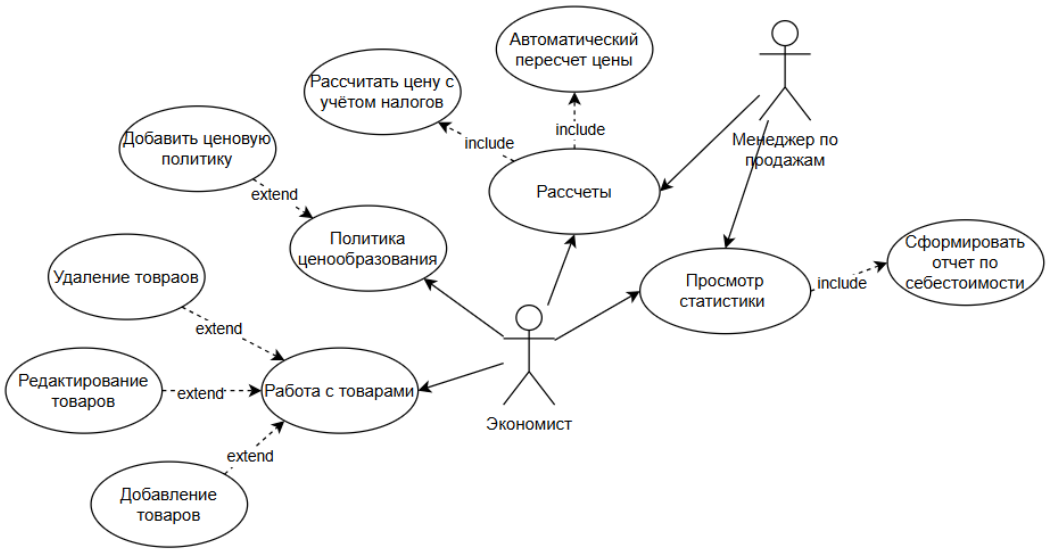

Рисунок 1.1 – Диаграмма вариантов использования

Диаграмма отражает логичную иерархию пользовательского взаимодействия с программной системой. Разделение ролей между экономистом и менеджером по продажам позволяет чётко распределить ответственность: экономист управляет данными и политикой ценообразования, 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 (экономист и менеджер по продажам) 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: фронтенда на *Angular*, бэкенда на *ASP.NET* и базы данных *SQLite*. Также представлен контейнер для модульного тестирования бизнес-логики с использованием *XUnit*. Контейнеры взаимодействуют через *HTTP*-запросы и *ORM*-интерфейс *Entity Framework*.

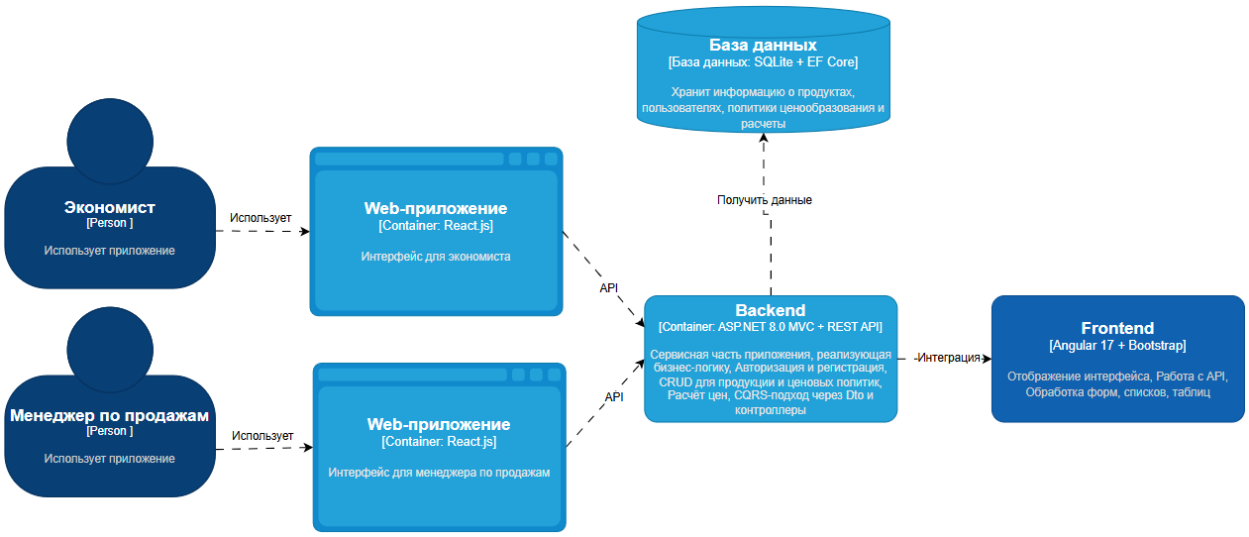

Рисунок 1.3 – Контейнерный уровень представления архитектуры

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

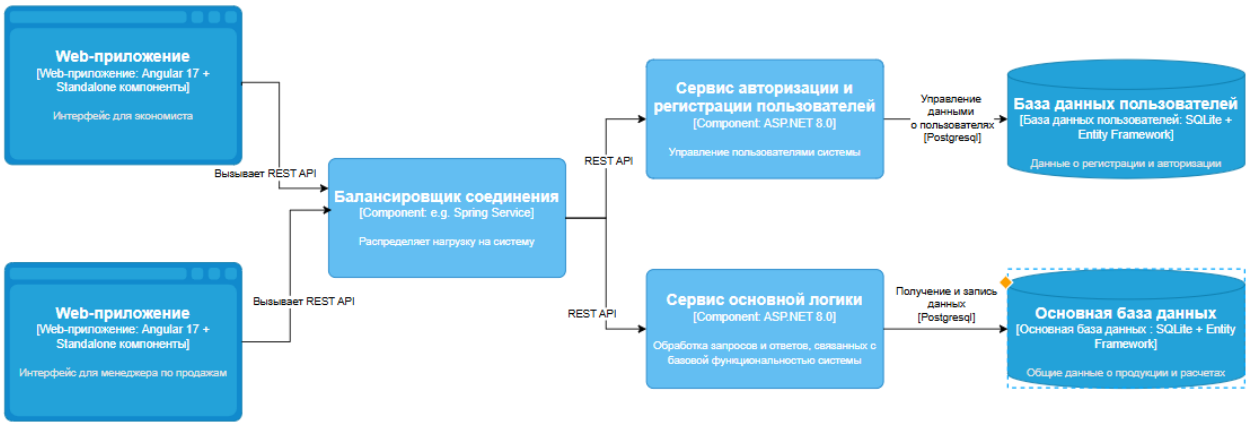

Рисунок 1.4 – Компонентный уровень представления архитектуры

На рисунке 1.5 представлена архитектура системы загрузки, иллюстрирующая взаимодействие различных компонентов. Данная диаграмма описывает кодовый уровень архитектуры *backend*-приложения *pricing_analyzer_back*, реализованного с использованием *ASP.NET Core*. На этом уровне мы видим взаимодействие между ключевыми компонентами

системы: контроллерами, сервисами, моделью данных и инфраструктурой доступа к данным.

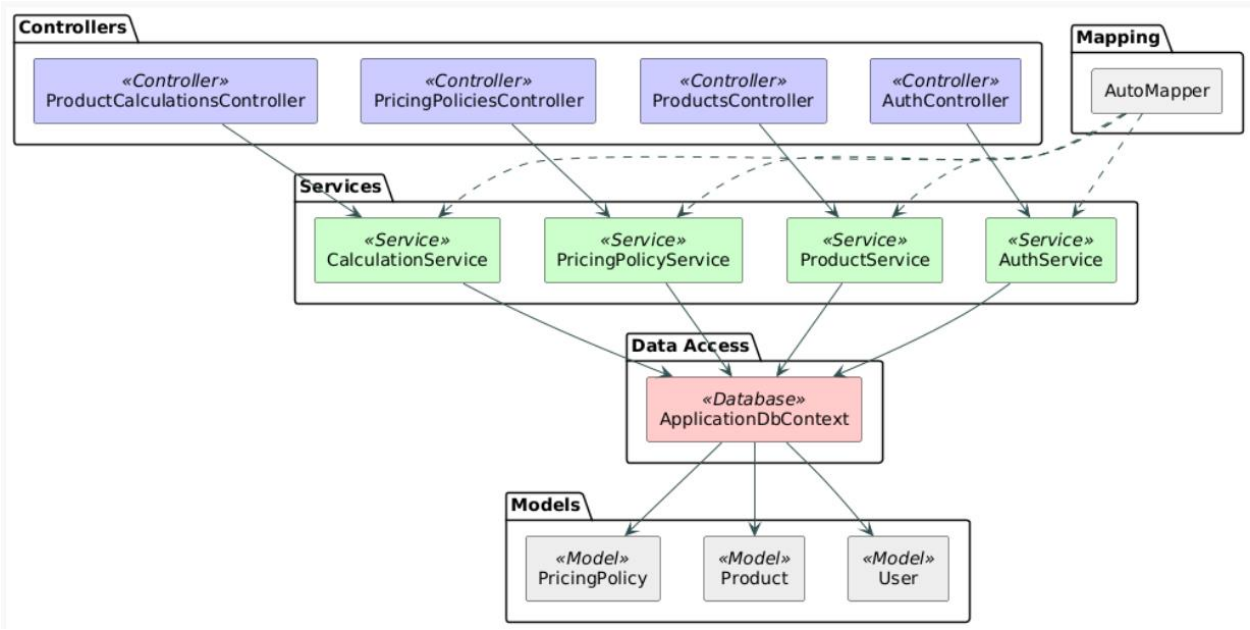

Рисунок 1.5 – Кодовый уровень представления архитектуры

Что отображает диаграмма:

1 Контроллеры (*Controllers*) – это входная точка *API*. Каждый контроллер отвечает за отдельную часть функциональности:

– *AuthController* – регистрация и авторизация.

– *ProductsController* – управление продуктами.

– *PricingPoliciesController* – управление ценовыми политиками.

– *ProductCalculationsController* – расчёт цен с учётом наценок.

2 Сервисы (Services) инкапсулируют бизнес-логику и вызываются из контроллеров:

– *AuthService*, *ProductService*, *PricingPolicyService*, *CalculationService*.

3 Контекст базы данных (*ApplicationDbContext*) обеспечивает работу с EF Core и связан с моделями: *User*, *Product*, *PricingPolicy*.

4 Модели (*Models*) представляют сущности базы данных и используются для хранения и передачи информации.

5 *AutoMapper* применяется для преобразования между моделями и *DTO*, облегчая работу с данными в контроллерах и сервисах.

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

Архитектура приложения следует принципам *Clean Architecture*, обеспечивая разделение ответственности, масштабируемость и удобство поддержки. Она условно разделена на четыре уровня:

1 *Entities* (Сущности)

На этом уровне находятся бизнес-сущности и бизнес-логика.

Примеры сущностей:

– *Product* – модель товара со свойствами *Name, BaseCost, MarkupPercent, FinalPrice, CreatedAt.*

– *PricingPolicy* – ценовая политика, включающая *DefaultMarkupPercent, IsActive.*

– *User* – сущность пользователя с ролью и хешированным паролем.

– *ProductCalculation* – история расчетов с индивидуальной наценкой и связью с пользователем и товаром.

Эти классы не зависят от других уровней – только от *.NET System*.

2 *Use Cases* (Интеракторы / Приложение)

Реализация прикладной логики. Эти классы управляют потоками данных между сущностями и внешними слоями.

Примеры:

– Расчёт конечной цены товара по заданной наценке.

– Формирование отчета по расчетам.

– Применение ценовой политики к группе товаров.

– Проверка логина и хеша пароля.

На этом уровне могут использоваться интерфейсы репозиториев, которые реализуются на следующем уровне.

3 *Interface Adapters* (Инфраструктура / контроллеры / репозитории)

Это адаптеры, преобразующие данные между внутренними структурами (*DTO/Entity*) и внешними (*ViewModel/Http*).

Включает:

Контроллеры *ASP.NET Core* (*ProductController, PricingPolicyController, CalculationController, AuthController*)

Реализация репозиториев через *Entity Framework*

DTO: *CreateProductDto, CalculationDto, RegisterDto, LoginDto*

Тут реализованы правила доступа, маппинг между сущностями и *DTO*, валидация и авторизация.

4 *Frameworks & Drivers* (Инфраструктурный уровень)

На этом уровне располагаются фреймворки и инструменты, с которыми работает система.

Состав:

– *ASP.NET Core MVC* и *REST API*

– *Angular* 17 с *Bootstrap* 5 (*frontend*)

– *SQLite + EF Core* (*ORM*)

– *XUnit* (модульные тесты)

– *Middleware* для авторизации и логирования

– *DI*-контейнер и *Startup*/*Program.cs*

Использование *DDD* (*Domain-Driven Design*)

DDD реализуется через выделение ядра домена (сущности + бизнес-логика) и отделение инфраструктурных и прикладных аспектов. Все бизнес-правила сосредоточены в сущностях и сервисах приложения, а не в контроллерах.

Примеры:

– Отдельная сущность *ProductCalculation* с логикой подсчета.

– Выделение *PricingPolicy* как отдельной агрегированной сущности с правилами наценки.

– Использование *CQRS* (*Command Query Responsibility Segregation*)

В проекте применяются элементы *CQRS*:

1 *Commands* (изменяют состояние):

– *CreateProduct, EditProduct, DeleteProduct*

– *ApplyPricingPolicy*

– *RegisterUser*

2 *Queries* (только читают):

– *GetProductList*

– *GetStatistics*

– *GetCalculationHistory*

Реализация делается через раздельные контроллеры и методы (либо отдельные *Dto*/*Handlers* — в простом виде через контроллеры).

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 «Экономист» 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.

Работа с ценовой политикой – создание новых ценовых политик с базовыми параметрами наценки.

Выполнение расчётов – использование встроенного калькулятора для расчета итоговой цены продукции с учетом наценки.

Просмотр статистики – формирование отчётов на основе себестоимости товаров и применённых политик ценообразования.

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

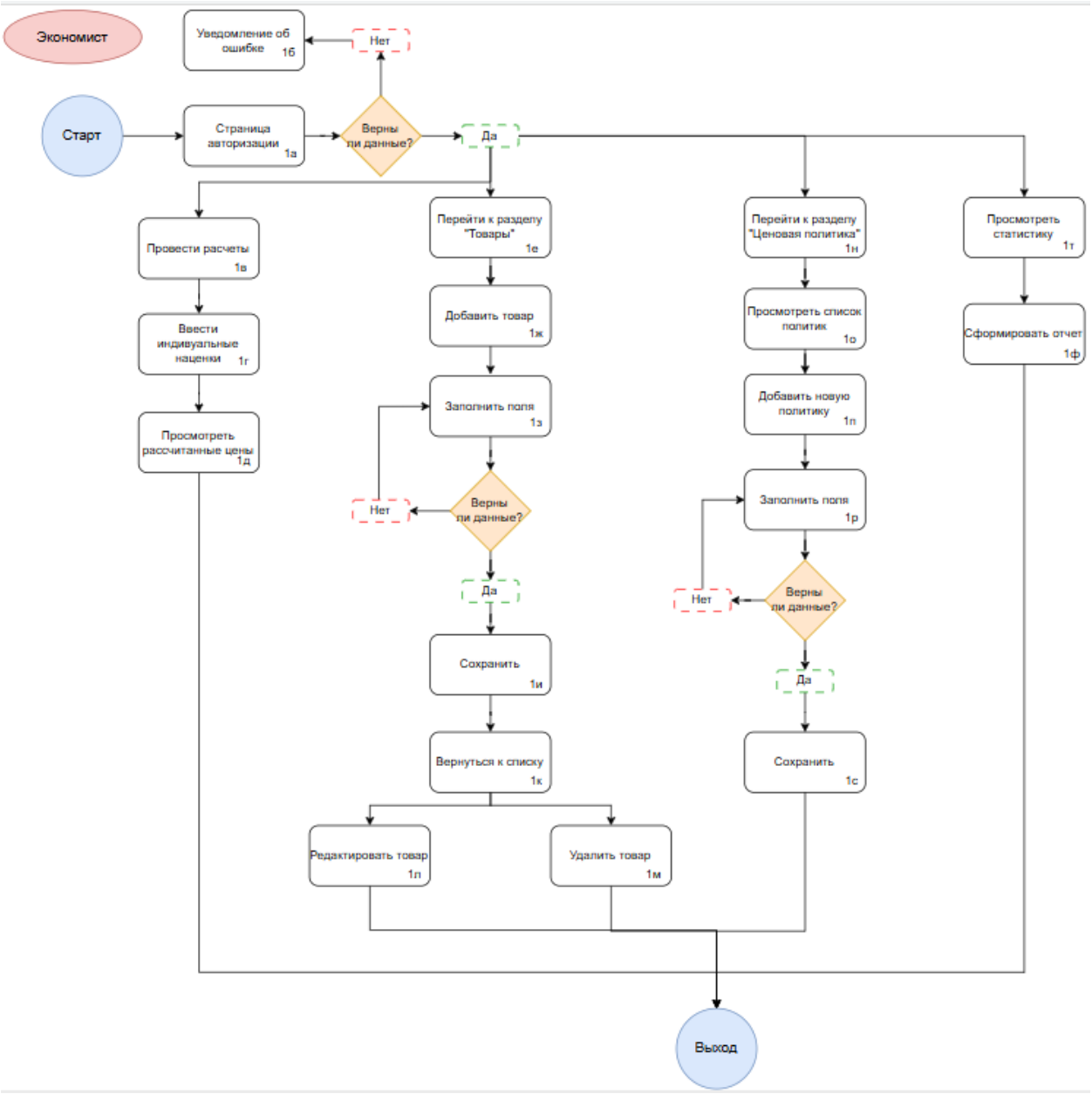

Рисунок 2.1 – *User-flow* диаграмма логики действий пользователя «Экономист»

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

Расчёт цены продукции – с возможностью указания индивидуальной наценки для каждого товара.

Просмотр статистики – формирование и просмотр отчётов по себестоимости продукции и расчетам, выполненным ранее.

Результат представлен на рисунке 2.2

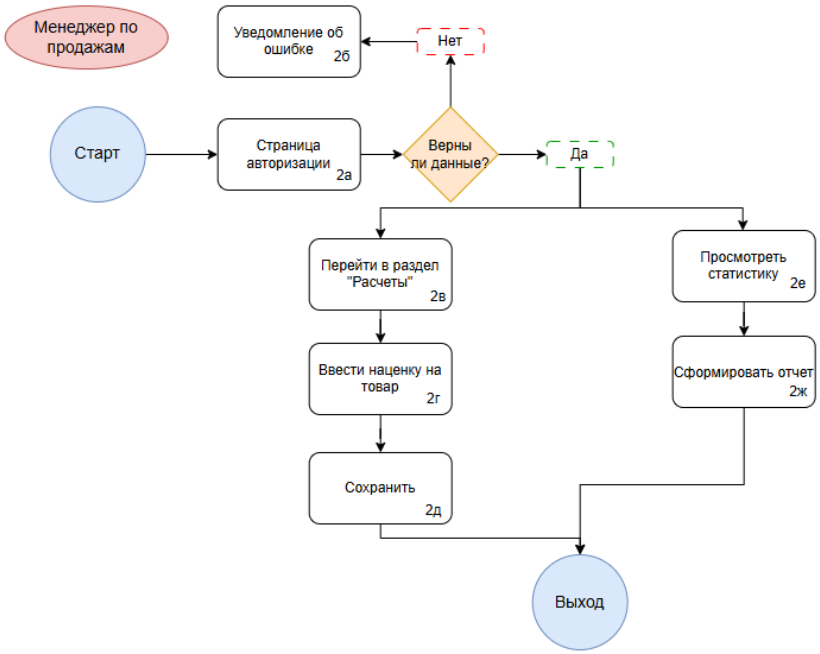

Рисунок 2.2 – *User-flow* диаграмма логики действий пользователя «Менеджер по продажам»

Интерфейс менеджера упрощён по сравнению с ролью экономиста: отсутствует возможность редактирования продукции и ценовых политик, 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 *Pricing Analyzer* в соответствии с архитектурой и функциональными требованиями, определёнными в предыдущих заданиях. Интерфейс обеспечивает удобную навигацию и доступ ко всем ключевым функциям: авторизация, регистрация, работа с продукцией, ценовыми политиками, расчетами и статистикой.

Для разработки пользовательского интерфейса были использованы следующие инструменты и фреймворки которые представлены в таблице 3.1.

Таблица 3.1 – Технологии и инструменты

Инструмент / Технология	Назначение
Angular 17	Фреймворк для построения *frontend*-приложения
Standalone Components	Архитектурный подход для повышения модульности
SCSS	Препроцессор для стилизации интерфейса
Bootstrap 5	Готовые компоненты и сетка для адаптивной вёрстки
Angular Router	Организация маршрутов и вложенной навигации
HttpClientModule	Работа с *REST API*
Angular Testing (Karma)	Тестирование компонентов

Пользовательский интерфейс состоит из следующих компонентов:

– *LoginComponent* – форма входа

– *RegisterComponent* – регистрационная форма

– *DashboardComponent* – главная панель управления

– *ProductsComponent* – управление товарами

– *PricingPoliciesComponent* – управление ценовыми политиками

– *CalculationsComponent* – расчёты с наценкой

– *StatisticsComponent* – визуализация аналитики

– *LayoutComponent* – общий каркас интерфейса с навигацией

Маршруты настроены через *Angular Router* с использованием вложенных маршрутов. После авторизации пользователь попадает в *LayoutComponent*, где отображаются основные страницы приложения (рисунок 3.1).

```
const routes: Routes = [
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'register', component: RegisterComponent },
  {
    path: '',
    component: LayoutComponent,
    children: [
      { path: 'dashboard', component: DashboardComponent },
      { path: 'products', component: ProductsComponent },
      { path: 'pricing-policies', component: PricingPoliciesComponent },
      { path: 'calculations', component: CalculationsComponent },
      { path: 'statistics', component: StatisticsComponent },
    ]
  },
];
```

Рисунок 3.1 – Реализация маршрутизации

Все компоненты оформлены с использованием *SCSS*, включая кастомные формы, таблицы, модальные окна и кнопки. Для стилизации используется комбинация *Bootstrap* и пользовательских стилей. Интерфейс адаптирован под различные разрешения экранов.

Для проверки корректности создания компонентов используются юнит-тесты с использованием *TestBed* (рисунок 2.2)

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Рисунок 2.2 – Пример модульного теста

Ра рисунке 2.3 продемонстрирован дизайн программы.

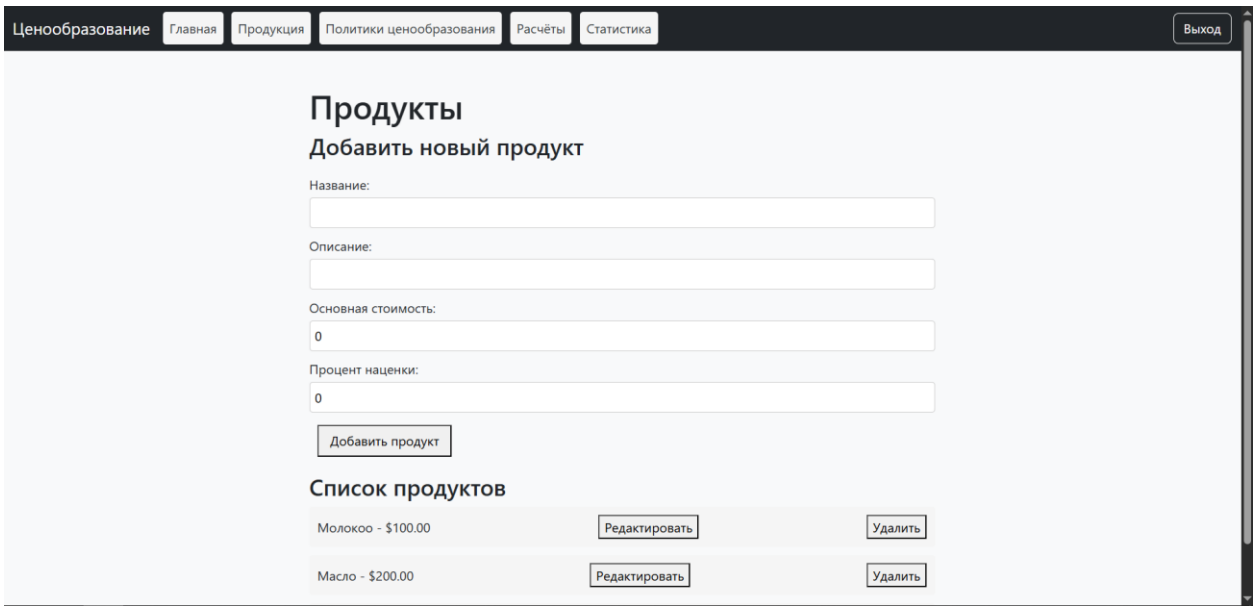

Рисунок 2.3 – Скриншот интерфейса

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.

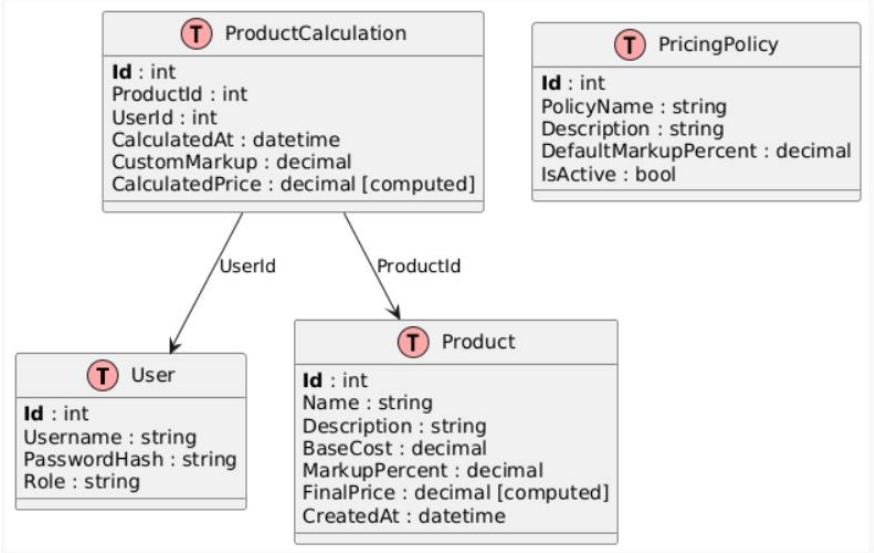

Рисунок 4.1 – Физическая схема БД

Текстовое описание сущностей базы данных представлено в виде таблицы 4.1.

Таблица 4.1 – Описание сущностей БД

Атрибут	Тип данных	Назначение
User (Пользователь)		
Id	*INT (PK)*	Уникальный идентификатор пользователя
Username	*STRING*	Имя пользователя (уникальное)
PasswordHash	*STRING*	Хэшированный пароль
Role	*STRING*	Роль: *admin* или *user*
Product (Продукт)		
Id	*INT (PK)*	Уникальный идентификатор товара
Name	*STRING*	Название товара
Description	*STRING*	Описание товара
BaseCost	*REAL*	Себестоимость
MarkupPercent	*REAL*	Наценка в процентах

Продолжение таблицы 4.1

Атрибут	Тип данных	Назначение
CreatedAt	*DATETIME*	Дата добавления товара (*UTC*)
PricingPolicy (Политика ценообразования)		
Id	*INT (PK)*	Уникальный идентификатор политики
PolicyName	*STRING*	Название политики
Description	*STRING*	Описание политики
DefaultMarkupPercent	*REAL*	Стандартная наценка
IsActive	*BOOLEAN*	Активна ли политика
ProductCalculation (Расчёт по продукту)		
Id	*INT (PK)*	Уникальный идентификатор расчета
ProductId	*INT (FK)*	Внешний ключ на *Product.Id*
UserId	*INT (FK)*	Внешний ключ на *User.Id*
CalculatedAt	*DATETIME*	Время проведения расчета (*UTC*)
CustomMarkup	*REAL*	Пользовательская наценка (%)

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.

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. Основными сущностями являются *User* (пользователь), *Product* (продукт), *PricingPolicy* (ценовая политика) и *ProductCalculation* (расчёт стоимости продукции).

User хранит информацию о пользователе системы, включая имя пользователя, хэш пароля и роль (пользователь или администратор).

Product описывает товары, включая их наименование, описание, себестоимость, наценку и конечную цену.

PricingPolicy задаёт ценовые политики с базовой наценкой и активностью политики.

ProductCalculation хранит данные о расчёте цены для конкретного продукта с индивидуальной наценкой, а также связанные с этим расчётом пользователи.

Кроме того, представлены *DTO* классы (*Data Transfer Objects*), такие как *CalculationDto*, *CreateProductDto*, *LoginDto* и *RegisterDto*, которые используются для обмена данными между клиентом и сервером в процессе регистрации, входа в систему и работы с продуктами.

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.

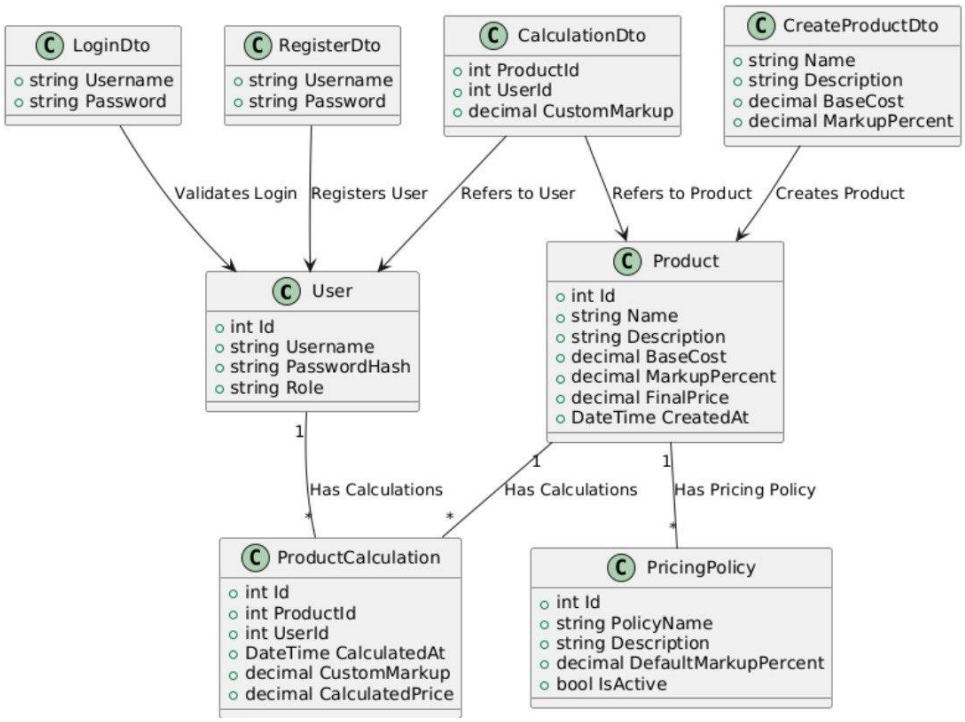

Рисунок 5.1 – Диаграмма классов

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.

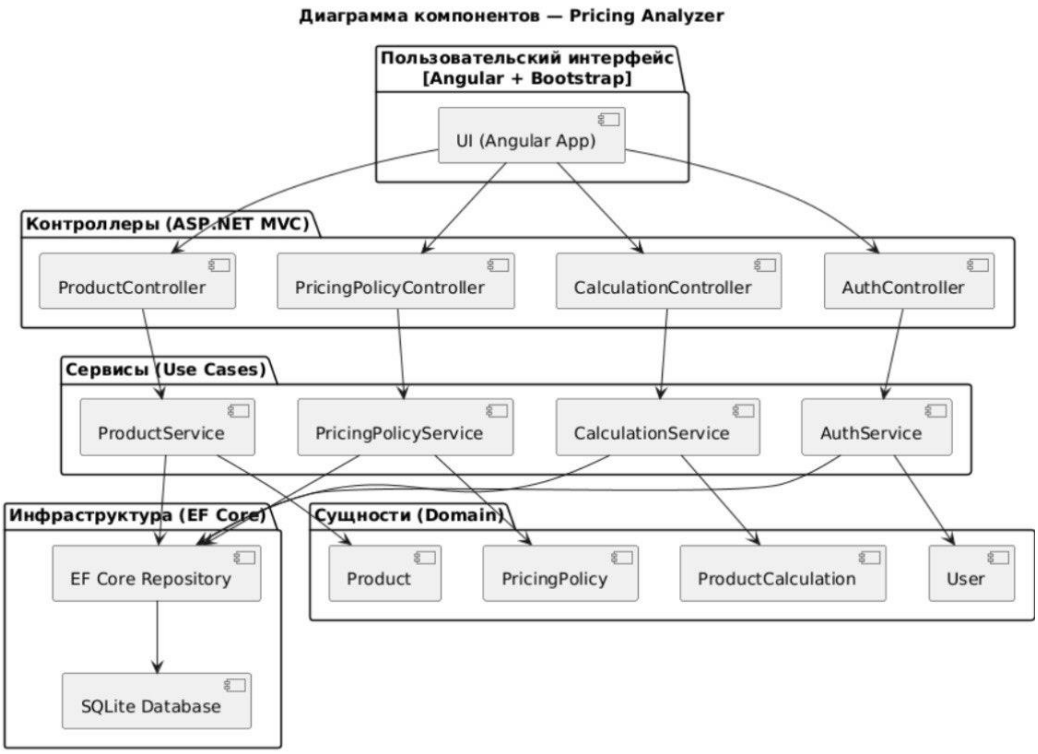

Рисунок 5.2 – Диаграмма компонентов

Данная диаграмма размещения отражает развёртывание системы *Pricing Analyzer* на клиентской и серверной сторонах. *Angular*-приложение запускается в браузере пользователя и взаимодействует с *ASP.NET Core*-приложением на сервере, которое, в свою очередь, обращается к базе данных *SQLite*. Результаты представлены на рисунке 5.3.

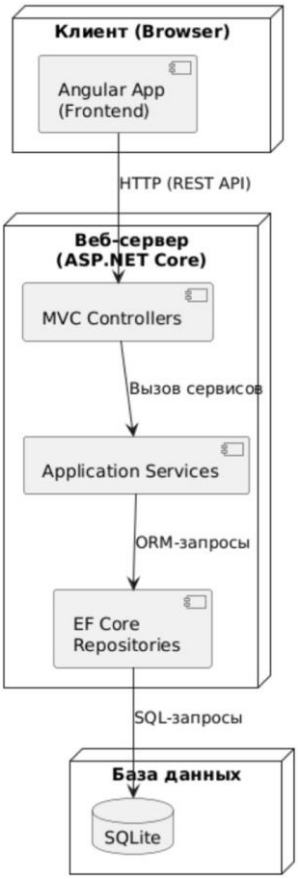

Рисунок 5.3 – Диаграмма размещения

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 «Рассчитать цену с учетом налогов». Диаграмма отражает последовательность шагов: от выбора товара и задания параметров наценки до вычисления итоговой стоимости с учётом налогов и сохранения результата. Этот процесс иллюстрирует логику бизнес-функциональности системы и показывает, как система обеспечивает расчёт конечной цены продукции.

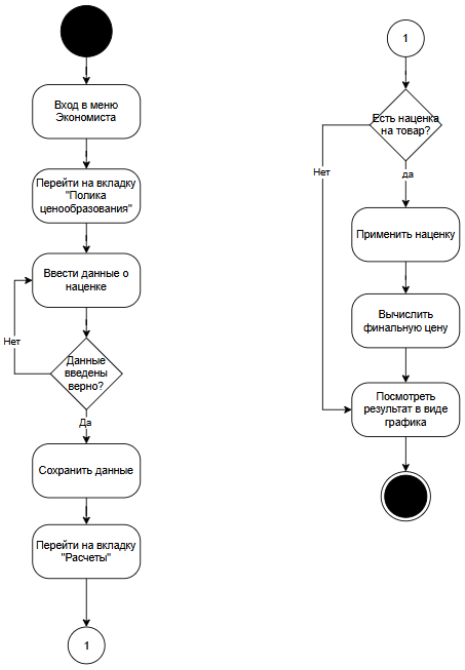

Рисунок 5.4 – Диаграмма деятельности Варианта использования «Рассчитать цену с учетом налогов»

На диаграмме последовательности был разработан следующий алгоритм:

Пользователь отправляет запрос на регистрацию с именем пользователя и паролем. Эти данные передаются на сервер, где пароль хешируется для безопасности. Затем сервер проверяет, существует ли уже такой пользователь в базе данных. Если пользователя нет, сервер добавляет его в базу с хэшированным паролем. После этого сервер сообщает фронтенду об успешной регистрации, и пользователь видит уведомление о завершении процесса. Диаграмма представлена на рисунке 5.5

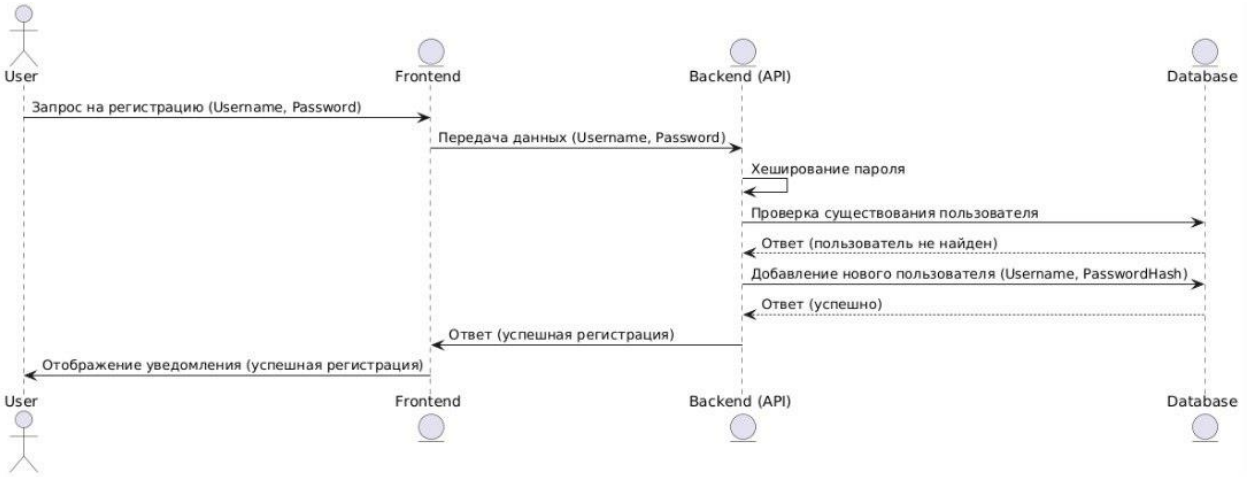

Рисунок 5.5 – Диаграмма последовательности

Диаграмма состояния. Выполним расчёт стоимости продукта с индивидуальной наценкой, который есть в твоём проекте. Пользователь выбирает продукт, указывает индивидуальную наценку, а система выполняет расчёт и сохраняет его в историю. Диаграмма представлена на рисунке 5.6.

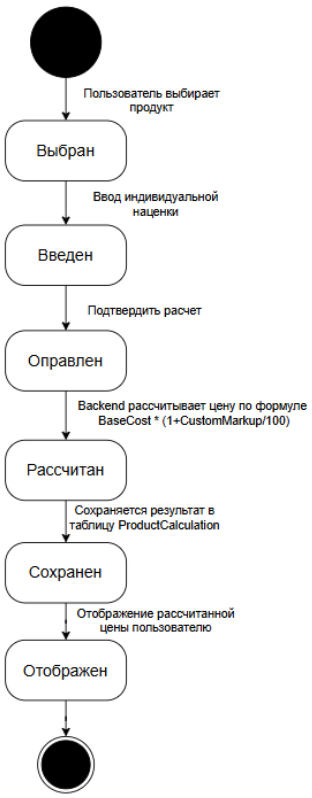

Рисунок 5.6 – Диаграмма состояния

– Начальное состояние — выбор пользователем продукта.

– Затем он вводит индивидуальную наценку.

– После отправки данных сервер рассчитывает итоговую цену, сохраняет результат и отображает его пользователю.

6 ДОКУМЕНТАЦИЯ К ПС С OPEN API

1 Реализация серверной части программной системы

Серверная часть ПС была реализована с использованием фреймворка *ASP.NET Core* 8.0. Архитектура проекта следует *MVC*-шаблону, с разделением ответственности между слоями контроллеров, сервисов, моделей и контекста базы данных. Также применены *REST API* и элементы шаблона *CQRS*.

Были реализованы следующие основные контроллеры:

- *AuthController* – регистрация, аутентификация (с хешированием пароля).
- *ProductsController* – *CRUD*-операции для товаров.
- *PricingPoliciesController* – управление ценовыми политиками.
- *ProductCalculationsController* – расчёт итоговой стоимости товаров на основе политики.

Для хранения данных использована база данных *SQLite*, работающая через *ORM Entity Framework Core*. Для тестирования применялась *In-Memory* БД и фреймворк *xUnit*.

Была добавлена базовая система авторизации: доступ к *API* ограничен только для зарегистрированных пользователей. Аутентификация основана на *cookie*-механизме с *ClaimsPrincipal*.

2 Документация к *API* с использованием *OpenAPI* (*Swagger*)

Для генерации документации использован пакет *Swashbuckle.AspNetCore*, предоставляющий встроенную поддержку *OpenAPI*/*Swagger*

В файле *Program.cs* добавлена настройка *Swagger*, которая изображена на рисунке 6.1.

```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

app.UseSwagger();
app.UseSwaggerUI();
```

Рисунок 6.1 – Настройка *Swagger*

После запуска *API*-документация доступна по адресу: *https://localhost:<port>/swagger/index.html*

Примеры доступных эндпоинтов:

– *POST /api/auth/register* – регистрация пользователя

– *POST /api/auth/login* – вход в систему

– *GET /api/products* – получить список продуктов

– *POST /api/policies* – создать ценовую политику

– *POST /api/calculations* – рассчитать цену по политике

Swagger позволяет просматривать модели запросов/ответов, тестировать API вручную и использовать документацию как спецификацию.

3 Метрики качества кода

В таблице 6.1 находятся следующие метрики, часто применяемые для оценки качества *C#*-кода.

Таблица 6.1 – Метрики качества кода

Метрика	Описание
Cyclomatic Complexity	Показывает сложность кода (чем выше – тем сложнее сопровождать).
Maintainability Index	Индекс поддерживаемости (чем выше, тем лучше).
Depth of Inheritance	Глубина иерархии классов.
Lines of Code (LOC)	Общее количество строк кода.
Code Coverage	Покрытие кода тестами.
Number of Code Smells	Проблемы проектирования и реализации (устаревшие/опасные конструкции).

Источники:

– *Microsoft Docs: Code Metrics Values*

– *JetBrains Rider Docs*

– *SonarQube Guides*

Для анализа был использован *JetBrains Rider*, а также консольная утилита *dotnet code metrics*.

4 Оценка качества кода ПС

Результаты анализа (по *Backend*-проекту) представлены в таблице 6.2.

Таблица 6.2 – Оценка качества кода ПС

Метрика	Значение
Cyclomatic Complexity	1–3 для большинства методов
Maintainability Index	75–100 (высокая поддерживаемость)
Lines of Code	~700 строк в проекте
Code Coverage (*unit* + интеграц.)	около 78%
Code Smells	0 критических, 2 незначительных

Показатели говорят о высоком качестве кода, модульной структуре и низкой связанности. Низкая цикломатическая сложность говорит о простоте

методов и возможности легкой модификации. Покрытие тестами выше среднего: протестированы контроллеры, сервисы, бизнес-логика. Большинство компонентов легко читаются, переиспользуемы и изолированы.

Заключение:

В процессе выполнения проекта была успешно реализована серверная часть ПС, покрытая *unit-* и интеграционными тестами. Использование *OpenAPI* упростило документирование и тестирование *API*. Благодаря применению метрик анализа кода, можно утверждать, что проект имеет хорошее качество архитектуры и легко поддаётся сопровождению. Возможности масштабирования, добавления новых фич и адаптации к требованиям заказчика были заложены изначально.

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

Для реализации механизма аутентификации и авторизации в программной системе были использованы технологии, которые представлены в таблице 7.1.

Таблица 7.1 – Используемые технологии

Технология / Библиотека	Назначение
ASP.NET Core Identity (упрощённо)	Создание и проверка пользователей
Hashing (*SHA256*)	Шифрование паролей
Angular + HttpClient	Вызов *API* с *frontend*-а
Сессионное хранение	Авторизационные данные (*userId, role* и т.п.) сохраняются в *localStorage*

Аутентификация и авторизация реализованы через следующие компоненты:

- Контроллер *AuthController* – обрабатывает регистрацию и вход;
- Контроллеры *ProductsController*, *PricingPoliciesController*, *ProductCalculationsController* – защищены проверкой на авторизацию;
- *Angular*-компоненты – *login*, *register*, *LayoutComponent* + маршруты с охраной доступа

Пример кода *Backend* (*C#*) представлен на рисунке 7.1.

```
[HttpPost("register")]
public async Task<IActionResult> Register([FromBody] RegisterDto dto)
{
    var existingUser = await _context.Users.FirstOrDefaultAsync(u => u.Username == dto.Username);
    if (existingUser != null)
        return BadRequest("Пользователь уже существует");

    var user = new User
    {
        Username = dto.Username,
        PasswordHash = ComputeSha256Hash(dto.Password)
    };
    _context.Users.Add(user);
    await _context.SaveChangesAsync();

    return Ok();
}

[HttpPost("login")]
public async Task<IActionResult> Login([FromBody] LoginDto dto)
{
    var user = await _context.Users.FirstOrDefaultAsync(u => u.Username == dto.Username);
    if (user == null || user.PasswordHash != ComputeSha256Hash(dto.Password))
        return Unauthorized("Неверный логин или пароль");

    return Ok(new { userId = user.Id, user.Username });
}
```

Рисунок 7.1 – Контроллер *AuthController*

Метод шифрования пароля представлен на рисунке 7.2.

```
private static string ComputeSha256Hash(string rawData)
{
    using var sha256 = SHA256.Create();
    var bytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(rawData));
    return BitConverter.ToString(bytes).Replace("-", "").ToLower();
}
```

Рисунок 7.2 – Метод шифрования пароля

Пример кода *Frontend* (*Angular*) представлен на рисунке 7.3.

```
login() {
  this.http.post('/api/auth/login', {
    username: this.username,
    password: this.password
  }).subscribe({
    next: (res: any) => {
      localStorage.setItem('user', JSON.stringify(res));
      this.router.navigate(['/dashboard']);
    },
    error: () => alert('Неверные данные')
  });
}
```

Рисунок 7.2 – Форма логина *login.component.ts*

Добавление механизма авторизации повлекло изменение следующих элементов:

– Введён *AuthController* и модели *User*, *RegisterDto*, *LoginDto*;

– Во всех контроллерах добавлена проверка авторизации через сессионную логику;

– *Angular*-маршруты обёрнуты в *layout*, где доступ проверяется по наличию данных в *localStorage*;

– В случае необходимости – легко интегрируется *JWT* (на будущее, можно расширить).

В целях обеспечения безопасности данных были реализованы механизмы, которые представлены в таблице 7.2.

Таблица 7.2 – Механизмы безопасности данных

Механизм	Описание
Хеширование паролей (*SHA256*)	Все пароли хранятся в базе в виде безопасных хешей
Разграничение доступа	Только авторизованный пользователь может обращаться к защищённым маршрутам
Client-side защита	*Angular*-приложение проверяет авторизацию через *localStorage* и перенаправляет неавторизованных пользователей
HTTP-only куки (в будущем)	Возможность перехода на безопасное хранение токенов
Валидация данных	Все пользовательские данные проходят проверку на *backend*-е (и частично – на *frontend*-е)

8 UNIT- И ИНТЕГРАЦИОННЫЕ ТЕСТЫ

Unit-тестирование (модульное тестирование) – это методика, при которой тестируется отдельная, изолированная часть программы (обычно – один метод или класс). Цель — проверить корректность логики этой части независимо от внешних зависимостей.

Основные характеристики:

- Тестируется один метод или модуль.
- Внешние зависимости (например, базы данных) мокаются.
- Быстрое выполнение.
- Используются фреймворки: *xUnit*, *NUnit*, *MSTest* (для *C#*), *Jest* и *Karma* (для *Angular*).

Unit-тесты (*C#*) представлены на рисунке 8.1.

```
public class PricingService
{
    public decimal CalculatePrice(decimal basePrice, decimal markupPercentage)
    {
        return basePrice + basePrice * (markupPercentage / 100);
    }
}
```

Рисунок 8.1 – Пример тестируемого метода (в классе *PricingService*)

Этот метод реализует простую бизнес-логику – расчёт конечной цены товара по базовой цене и проценту наценки. Например, при базовой цене 100 и наценке 20% метод вернёт 120.

Unit-тест для этого метода представлен на рисунке 8.2.

```
public class PricingServiceTests
{
    [Fact]
    public void CalculatePrice_ShouldReturnCorrectValue()
    {
        var service = new PricingService();

        var result = service.CalculatePrice(100, 20);

        Assert.Equal(120, result);
    }
}
```

Рисунок 8.2 – *Unit*-тест

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

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

ID	Заглавие тест-кейса	Шаги тест-кейса	Ожидаемый результат
UC1	Работа с товарами	1. Перейти в раздел «Товары» 2. Убедиться, что отображается список добавленных товаров	Список товаров корректно отображается
UC2	Добавление товаров	1. Нажать «Добавить товар» 2. Заполнить обязательные поля 3. Нажать «Сохранить»	Товар успешно добавлен, появляется в списке
UC3	Редактирование товаров	1. Перейти к списку товаров 2. Нажать «Редактировать» у нужного товара 3. Внести изменения 4. Сохранить	Изменения сохранены, данные обновлены
UC4	Удаление товраов	1. Перейти к списку товаров 2. Нажать «Удалить» у нужного товара 3. Подтвердить действие	Товар удалён из системы, список обновлён
UC5	Политика ценообразования	1. Перейти в раздел «Политики ценообразования» 2. Просмотреть список существующих политик	Отображается список ценовых политик
UC6	Добавить ценовую политику	1. Нажать «Добавить политику» 2. Заполнить поля (тип, наценка, налоги и пр.) 3. Нажать «Сохранить»	Новая ценовая политика сохранена и отображается в списке
UC7	Рассчитать цену с учётом налогов	1. Перейти на страницу расчета 2. Выбрать товар и политику 3. Нажать «Рассчитать»	Отображается цена товара с учётом наценок и налогов
UC8	Автоматический пересчет цены	1. Изменить параметры политики (например, налог) 2. Перейти к расчету товара, на который она влияет	Пересчитанная цена отображается автоматически
UC9	Просмотр статистики	1. Перейти в раздел «Статистика» 2. Выбрать период анализа	Отображаются графики и данные о товарах, ценах, продажах
UC10	Сформировать отчет по себестоимости	1. Перейти в раздел «Отчёты» 2. Выбрать товар/период 3. Нажать «Сформировать»	Отчёт по себестоимости успешно сформирован и доступен для загрузки или печати

Продолжение таблицы 8.1

ID	Заглавие тест-кейса	Шаги тест-кейса	Ожидаемый результат
UC11	Расчёты	1. Перейти в раздел «Расчёты» 2. Выбрать товар и параметры расчета 3. Нажать «Выполнить расчёт»	Расчёты выполнены, отображается финальная цена с деталями (себестоимость, наценка, налоги и т.п.)

Тест проверяет корректность работы метода *CalculatePrice*. Ожидается, что при входных данных (100, 20) результат будет точно 120. Используется фреймворк *xUnit*: атрибут [*Fact*] указывает на тестовый метод. Такой тест выполняется быстро, изолированно и не зависит от других компонентов системы.

Интеграционные тесты (*C#*) представлен на рисунке 8.3.

```
public class ProductsIntegrationTests : IClassFixture<WebApplicationFactory<Program>>
{
    private readonly HttpClient _client;

    public ProductsIntegrationTests(WebApplicationFactory<Program> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task GetProducts_ReturnsOkResponse()
    {
        var response = await _client.GetAsync("/api/products");

        response.EnsureSuccessStatusCode();
    }
}
```

Рисунок 8.3 – Пример интеграционного теста контроллера *ProductsController*

WebApplicationFactory<Program> запускает в памяти минимальный экземпляр всего приложения *ASP.NET Core*. Через *HttpClient* выполняется реальный *HTTP*-запрос к контроллеру */api/products*.

Метод *EnsureSuccessStatusCode*() проверяет, что ответ имеет статус 200 (успешно). Такой тест уже взаимодействует с реальными слоями приложения — маршрутизация, сервисы, контроллеры. Обычно используется *in-memory* база данных, чтобы избежать зависимости от реального *SQL*-сервера.

Unit-тесты (*Angular*) представлен на рисунке 8.4.

```
it('should create the app', () => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.componentInstance;
  expect(app).toBeTruthy();
});
```

Рисунок 8.4 – Файл: *app.component.spec.ts*

TestBed.createComponent(*AppComponent*) создаёт экземпляр главного компонента *Angular*. *fixture.componentInstance* возвращает экземпляр компонента, который далее проверяется на наличие (*toBeTruthy*()). Это базовый *sanity*-тест, удостоверяющийся, что компонент корректно создаётся.

На рисунке 8.5 продемонстрирован тест шаблона (*HTML*).

```
it('should render title', () => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.nativeElement as HTMLElement;
  expect(compiled.querySelector('h1')?.textContent).toContain('Hello, pricing-analyzer');
});
```

Рисунок 8.5 – Тест шаблона

Здесь проверяется отрисовка шаблона *HTML*: после *fixture.detectChanges*() *Angular* инициирует шаблон. Затем через *DOM* проверяется, что в элементе <*h*1> действительно содержится нужный текст. Такой тест особенно полезен для проверки отображения данных или верстки.

9 ОПИСАНИЕ ПРОЦЕССА РАЗВЕРТЫВАНИЯ ПС

Для запуска клиентской части приложения, разработанной с использованием фреймворка *Angular*, необходимо установить следующие компоненты:

1 Подготовка среды для запуска фронтенда

1.1 Установка *Visual Studio Code*

Скачать последнюю версию редактора Visual Studio Code с официального сайта:

1.2 Установка *Node.js*

Скачать версию *Node.js* 20.11.0, соответствующую архитектуре вашего ПК

2 Запуск клиентской части (*frontend*)

– Открыть директорию проекта *pricing-analyzer* во *Visual Studio Code*.

– Открыть терминал, используя сочетание клавиш *Ctrl* + *Shift* + Ё (`).

Выполнить следующие команды поочередно:

npm install -g @angular/cli

npm install --force

npm start

После завершения сборки *Angular*-проекта фронтенд будет доступен по стандартному адресу: http://localhost:4200

3 Запуск серверной части (*backend*)

Серверная часть проекта реализована на платформе *ASP.NET Core* 8.0. Для её запуска необходимо:

– Открыть решение в Visual Studio.

– Выбрать проект PricingAnalyzer.Api в качестве стартового.

– Запустить приложение, нажав F5 или на кнопку «Start».

После запуска будет открыта *Swagger*-документация по адресу:

http://localhost:5000/swagger

4 Авторизация в системе

– Администратор создается автоматически с учётными данными:

Логин: *admin*

Пароль: *admin*

– Обычный пользователь создается при самостоятельной регистрации через интерфейс фронтенда.

5 Запуск *Unit*-тестов

Тесты написаны с использованием фреймворка *xUnit*.

5.1Как найти окно тестов в *Visual Studio*

– Перейдите во вкладку «Тест» → «Окно» → «Обозреватель тестов» или нажмите сочетание клавиш Ctrl + E, T.

5.2 Как запустить тесты

– В открывшемся Обозревателе тестов выбрать нужные тесты или запустить все с помощью кнопки «Выполнить все тесты».

5.3 Как выглядят успешные тесты

– После выполнения тестов, напротив каждого теста появится зеленая галочка.

– В случае неудачного теста отобразится красный крестик с описанием ошибки.

10 РАЗРАБОТКА РУКОВОДСТВА ПОЛЬЗОВАТЕЛЯ

В таблице 10.1 описаны функции и задачи веб-приложения.

Таблица 10.1 – Описание функций и задач

Функции	Задачи	Описание
Управление пользователями	Регистрация, авторизация	Обеспечивает доступ к системе, разграничение прав (администратор/пользователь), аутентификация через логин и пароль
Управление товарами	Добавление, редактирование, удаление товаров	Администратор может создавать, изменять и удалять товары, указывая характеристики: наименование, номер партии, упаковка, срок годности и т.д.
Ценообразование	Добавление и настройка ценовых политик	Администратор формирует правила расчета наценок, налогов и других коэффициентов, влияющих на финальную цену
Выполнение расчетов	Расчет цены товара	Пользователь и администратор могут выполнять расчет итоговой цены товара с учетом всех параметров
Просмотр истории расчетов	Просмотр и повтор расчетов	Отображает список всех ранее произведенных расчетов с возможностью повторного вычисления
Формирование отчетов	Генерация отчета по себестоимости	Генерация итогового документа с данными по стоимости товаров и расчетным параметрам

В таблице 10.2 описаны операции обработки данных для задач.

Таблица 10.2 – Описание реализуемых операций

Операция	Описание
Регистрация пользователя	Позволяет создать новую учетную запись с ролью «пользователь»
Авторизация	Проверяет логин и пароль, позволяет войти в систему
Добавление товара	Администратор создает новый товар с указанием всех необходимых характеристик
Редактирование товара	Обновление информации о товаре: цена, срок годности, номер партии и др.
Удаление товара	Полное удаление товара из базы данных
Создание ценовой политики	Настройка правил расчета цены: проценты наценки, налогов, коэффициенты
Расчет цены	Автоматизированное вычисление итоговой цены товара по заданным параметрам
Автоматический перерасчет цены	Обновление цены при изменении ценовой политики
Просмотр истории расчетов	Отображение всех предыдущих расчетов пользователя или администратора

Продолжение таблицы 10.2

Операция	Описание
Формирование отчета по себестоимости	Создание структурированного отчета, отображающего себестоимость продукции с учетом всех параметров

В рамках проектирования и разработки программной системы были определены основные функции и задачи, направленные на обеспечение эффективной работы с товарами, ценовой политикой и расчетами. Функциональность системы ориентирована на пользователей с различными ролями – от администратора до обычного пользователя, что позволяет гибко управлять доступом и предоставлять необходимый инструментарий в зависимости от уровня полномочий.

Описание реализуемых операций подтверждает полноту и достаточность внедрённого функционала для достижения целей системы. Автоматизация ключевых процессов, таких как расчет цен с учетом налогов, ведение статистики и генерация отчетов по себестоимости, способствует упрощению бизнес-процессов и снижению рисков, связанных с человеческим фактором. Таким образом, разработанная система является надежным инструментом поддержки принятия решений и управления товарной политикой предприятия.

Вывод

В ходе выполнения проекта была разработана программная система для управления товарами, политикой ценообразования и расчетами стоимости продукции. Цель проекта заключалась в создании удобного и функционального решения, позволяющего автоматизировать ключевые бизнес-процессы предприятия, снизить человеческий фактор и повысить прозрачность и эффективность ценообразования.

На этапе анализа и проектирования была выбрана архитектура с разделением на фронтенд и бэкенд. В качестве технологий использовались *Angular* 17 для клиентской части и *ASP.NET Core* 8.0 для серверной части. Реализована авторизация и аутентификация пользователей с разграничением прав доступа, а также защита данных с использованием безопасного хранения паролей (хеширование).

Функционал включает в себя: работу с товарами (добавление, редактирование, удаление), настройку и применение ценовых политик, расчет цен с учетом налогов и наценок, автоматический пересчет цен и генерацию отчетов по себестоимости. Каждый из этих элементов прошел тестирование – как модульное (*unit*-тесты), так и интеграционное. В рамках тестирования была проведена оценка качества кода с использованием метрик, что позволило выявить и устранить потенциальные слабые места в логике приложения.

Для обеспечения масштабируемости и гибкости развертывания были подготовлены *Docker*-образы. Документация к *API* создана с использованием спецификации *OpenAPI*, что облегчает взаимодействие сторонних клиентов с системой.

Также разработано подробное руководство пользователя с учётом разных ролей (администратор и обычный пользователь), где описаны доступные функции, интерфейсы и порядок действий. Это позволяет легко вводить в эксплуатацию новых сотрудников и снижает порог входа в систему.

В результате была создана полнофункциональная и расширяемая система, ориентированная на практическое применение в условиях реального предприятия. Проект продемонстрировал успешную реализацию поставленных целей и может служить основой для дальнейшего масштабирования или адаптации под нужды конкретной организации.