**автономная некоммерческая образовательная организация**

**высшего образования Центросоюза Российской Федерации**

**«Сибирский университет потребительской кооперации»**

**Кафедра информатики**

|  |  |
| --- | --- |
|  | Работа допускается к защите |
|  | Заведующий кафедрой |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М.К. Черняков  (подпись) |
|  | «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2025 г. |

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

**(дипломная работа)**

на тему: «ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНТЕРАКТИВНОГО КОНФИГУРАТОРА СИСТЕМЫ ВЕНТИЛЯЦИИ (на материалах АО «ТИОН Умный микроклимат» г. Новосибирска)»

Обучающегося очной формы обучения

факультета экономики и управления

КОЛЕСНИКОВА Сергея Андреевича

Шифр ПИБ-О-01-21-021

Направление подготовки: 09.03.03 Прикладная информатика

*Направленность (профиль) образовательной программы: Прикладная информатика в информационной сфере*

Руководитель: преподаватель (практик)

БЛОК Иван Николаевич

Новосибирск 2025

Дата защиты

Оценка после защиты

Протокол заседания ГЭК №

Секретарь ГЭК

(подпись) (Фамилия И.О.)

Текст выпускной квалификационной работы

на \_\_\_\_ страницах

Приложение на \_\_\_\_\_ листах

(подпись обучающегося) (дата)

(подпись руководителя) (дата)

**автономная некоммерческая образовательная организация**

**высшего образования Центросоюза Российской Федерации**

**«Сибирский университет потребительской кооперации»**

Кафедра информатики

|  |  |
| --- | --- |
|  | Работа допускается к защите |
|  | Заведующий кафедрой |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М.К. Черняков  (подпись) |

**ЗАДАНИЕ**

на выпускную квалификационную работу обучающегося 4 курса

очной формы обучения

Факультета экономики и управления

Колесникова Сергея Андреевича

Шифр (группа) ПИБ-О-01-21-021 (ПИБ-11)

Направление подготовки: 09.03.03 Прикладная информатика,

*направленность (профиль) образовательной программы: Прикладная*

*информатика в информационной сфере*

1. Тема выпускной квалификационной работы: «Проектирование и разработка интерактивного конфигуратора системы вентиляции (на материалах АО «ТИОН Умный микроклимат» г. Новосибирска)»

утверждена приказом ректора от «\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2025 г., № \_\_\_\_\_\_

2. Срок предоставления выполненной работы на кафедру \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Перечень подлежащих разработке вопросов и общее направление работы:

- провести анализ деятельности образовательного учреждения;

- провести сравнительный анализ существующих аналогов тестирования знаний обучающихся;

- спроектировать архитектуру информационной системы;

- выбрать инструментальные средства разработки;

- спроектировать и разработать базу данных;

- разработать модуль тестирования кода;

- разработать серверную часть приложения и пользовательский интерфейс;

- выбрать технологию развертывания.

4. Контрольный график выполнения ВКР:

|  |  |  |
| --- | --- | --- |
| № п/п | Наименование структурного элемента ВКР  (глава / раздел) | Срок исполнения |
| 1 | Изучение литературных источников | февраль 2025 |
| 2 | Исследование предметной области и проектирование архитектуры | февраль 2025 |
| 3 | Разработка приложения и написание теоретической части работы | март 2025 |
| 4 | Разработка приложения и написание практической части работы | апрель 2025 |
| 5 | Формирование введения, заключения, списка источников и приложений | май 2025 |
| 6 | Оформление и сдача работы руководителю | июнь 2025 |

5. Рекомендуемый объём ВКР 70 стр.

6. Рекомендуемые места прохождения преддипломной практики и сбора   
фактического материала *г. Новосибирск, пр. К. Маркса, д. 26, автономная некоммерческая образовательная организация высшего образования Центросоюза Российской Федерации «Сибирский университет потребительской кооперации»*

7. Консультанты по смежным вопросам выпускной квалификационной работы не предусмотрены

8. Срок предоставления ВКР для проверки в системе «Антиплагиат. ВУЗ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

9. Дата выдачи задания \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель

выпускной квалификационной работы \_\_\_\_\_\_\_\_\_\_\_\_Блок И. Н.

*(подпись)*

Задание принял к исполнению \_\_\_\_\_\_\_\_\_\_\_\_ Колесников С. А.

*(подпись обучающегося)*

# ОГЛАВЛЕНИЕ

[ОГЛАВЛЕНИЕ 5](#_Toc199432864)

[ВВЕДЕНИЕ 6](#_Toc199432865)

[Глава 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И СУЩЕСТВУЮЩИХ РЕШЕНИЙ 8](#_Toc199432866)

[1.1 Общая характеристика и инфраструктура университета 8](#_Toc199432867)

[1.2 Обзор и сравнительный анализ существующих решений 11](#_Toc199432868)

[Глава 2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ РАЗРАБОТКИ ИНФОРМАЦИОННОЙ СИСТЕМЫ 12](#_Toc199432869)

[2.1 Выбор серверного фреймворка 13](#_Toc199432870)

[2.2 Обоснование выбора ORM 15](#_Toc199432871)

[2.3 Выбор клиентского фреймворка 17](#_Toc199432872)

[2.4 Выбор системы управления базами данных 19](#_Toc199432873)

[2.5 Использование Docker для разработки и развертывания 22](#_Toc199432874)

[Глава 3. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ ТЕСТИРОВАНИЯ ЗНАНИЙ ОБУЧАЮЩИХСЯ 25](#_Toc199432875)

[3.1 Анализ требований к конфигуратору 25](#_Toc199432876)

[3.2. Фрагменты спецификаций требований в виде диаграмм 28](#_Toc199432877)

[3.2 Проектирование архитектуры системы 31](#_Toc199432878)

[3.3 Проектирование базы данных 33](#_Toc199432879)

[3.4 Разработка серверной части приложения 37](#_Toc199432880)

[3.5 Разработка клиентской части приложения и описание пользовательского интерфейса 64](#_Toc199432881)

[ЗАКЛЮЧЕНИЕ 73](#_Toc199432882)

[СПИСОК ИСТОЧНИКОВ 75](#_Toc199432883)

[ПРИЛОЖЕНИЯ 77](#_Toc199432884)

# ВВЕДЕНИЕ

В последние годы вопросы обеспечения качественного микроклимата в жилых, коммерческих и производственных помещениях приобретают всё большую актуальность. Современные здания предъявляют высокие требования к системам вентиляции и очистки воздуха, что связано с ростом урбанизации, ужесточением санитарных норм, развитием технологий энергосбережения и повышением стандартов комфорта. В этих условиях особое значение приобретает автоматизация процессов подбора и конфигурирования вентиляционного оборудования, позволяющая повысить эффективность работы специалистов, снизить вероятность ошибок и обеспечить индивидуальный подход к каждому объекту.

Актуальность темыобусловлена необходимостью внедрения интеллектуальных цифровых инструментов, способных оптимизировать процесс выбора оборудования, повысить точность расчётов и упростить взаимодействие между производителем, дилерами и конечными клиентами. Внедрение таких решений способствует повышению конкурентоспособности предприятий, улучшению качества обслуживания и росту лояльности клиентов.

Цель работы — разработка и внедрение информационной системы для автоматизированного подбора и конфигурирования вентиляционного оборудования на примере компании ТИОН.

Задачи исследования:

* Проанализировать современное состояние рынка вентиляционного оборудования и существующие решения для автоматизации подбора.
* Изучить бизнес-процессы компании ТИОН и выявить основные требования к системе.
* Разработать архитектуру и структуру информационной системы, выбрать оптимальные инструментальные средства.
* Реализовать прототип конфигуратора вентиляционного оборудования с использованием современных технологий.
* Провести тестирование системы и оценить её эффективность.
* Сформулировать рекомендации по внедрению и дальнейшему развитию системы.

Объект исследования — процессы автоматизации подбора и конфигурирования вентиляционного оборудования.

Предмет исследования — методы и программные средства проектирования и реализации информационных систем для автоматизации подбора оборудования.

Методы исследования: анализ предметной области, сравнительный анализ существующих решений, проектирование архитектуры, программная реализация, тестирование, анализ эффективности.

Научная новизна работы заключается в разработке и внедрении интеллектуального конфигуратора, учитывающего специфику российского рынка и бизнес-процессов компании ТИОН, а также интеграции с современными цифровыми сервисами.

Практическая значимость заключается в возможности применения разработанной системы для повышения эффективности работы специалистов, сокращения времени на подбор оборудования и улучшения качества обслуживания клиентов.

Структура работы. Дипломная работа состоит из введения, трёх глав, заключения, списка литературы и приложений. В первой главе проводится анализ предметной области и существующих решений. Во второй главе описывается проектирование системы и выбор инструментальных средств. В третьей главе рассматривается реализация и внедрение системы, а также анализируются результаты её использования.

# Глава 1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И СУЩЕСТВУЮЩИХ РЕШЕНИЙ

## 1.1 Общая характеристика и инфраструктура университета

ТИОН – это группа научно-исследовательских, производственных и торговых подразделений, обеспечивающих полный цикл создания современных высокотехнологичных продуктов в области умной и энергоэффективной вентиляции, очистки и обеззараживания воздуха [3].

Компания «ТИОН» является одним из ведущих российских предприятий в области разработки, производства и внедрения систем увлажнения и очистки воздуха. Основанная в 2006 году, организация успешно заняла свою нишу на рынке климатического оборудования, предлагая инновационные решения в сегменте бытового и промышленного увлажнения воздуха. Основным направлением деятельности компании является производство аппаратуры, способствующей созданию комфортных и здоровых микроклиматических условий в жилых, коммерческих и промышленных помещениях.

Основной целью деятельности компании «ТИОН» является разработка передовых технологий, обеспечивающих высокое качество воздуха и оптимальные параметры влажности, что напрямую влияет на здоровье и самочувствие пользователей. В рамках стратегии развития компания активно инвестирует в научно-исследовательские работы, модернизацию производства и расширение ассортимента продукции.

Организационная структура компании построена по функциональному принципу и включает в себя несколько основных подразделений: производственное, маркетинговое, научно-исследовательское и сбытовое. Производственный отдел отвечает за изготовление и контроль качества выпускаемой продукции. Научно-исследовательское подразделение разрабатывает новые технологические решения и совершенствует существующие модели оборудования. Маркетинговая служба занимается анализом рынка, продвижением продукции и взаимодействием с клиентами. Сбытовой отдел осуществляет поставки и логистику.

Четкое распределение функций в организации способствует эффективному управлению процессами, оперативному решению производственных задач и своевременному реагированию на изменения рыночной среды.

Анализ подбора оборудования в компании:

В настоящее время процесс подбора оборудования ТИОН для клиентов осуществляется менеджерами по продажам вручную, с использованием каталогов продукции. Часто требуется консультация с техническими специалистами для проверки совместимости оборудования.

Основные этапы этого процесса:

1. Сбор требований от клиента: менеджер по продажам связывается с клиентом (по телефону, электронной почте или лично) и выясняет его потребности в системе вентиляции и очистки воздуха. Собирается информация о типе помещения, его площади, количестве людей, находящихся в помещении, и других факторах, влияющих на выбор оборудования.
2. Предварительный подбор оборудования: менеджер, основываясь на полученных требованиях, подбирает подходящее оборудование из каталога продукции. Используются таблицы технических характеристик и рекомендации по применению оборудования в различных типах помещений.
3. Согласование конфигурации с техническими специалистами: предварительно подобранная конфигурация согласовывается с техническими специалистами для проверки совместимости оборудования и соответствия требованиям клиента.
4. Составление коммерческого предложения: на основе согласованной конфигурации составляется коммерческое предложение, в котором указывается перечень оборудования, его стоимость и сроки поставки.
5. Согласование коммерческого предложения с клиентом: Коммерческое предложение направляется клиенту для согласования. В процессе согласования могут вноситься изменения в конфигурацию оборудования.
6. Оформление заказа: после согласования коммерческого предложения оформляется заказ на поставку оборудования.

Проблемы и недостатки существующего процесса:

* Длительное время подбора оборудования: процесс подбора оборудования может занимать значительное время, особенно при работе со сложными конфигурациями и большими объемами заказов.
* Высокая вероятность ошибок при ручном подборе оборудования: ручной подбор оборудования подвержен ошибкам, связанным с человеческим фактором, таким как невнимательность и недостаточная квалификация менеджеров по продажам.
* Зависимость от квалификации менеджера по продажам: качество подбора оборудования во многом зависит от квалификации и опыта менеджера по продажам.
* Неудобство для клиентов, желающих самостоятельно подобрать оборудование: клиенты, желающие самостоятельно подобрать оборудование, сталкиваются с трудностями при изучении каталогов продукции и технических характеристик.
* Сложность визуализации конечного результата: клиентам сложно представить, как будет выглядеть система вентиляции и очистки воздуха в их помещении на основе предоставленных технических данных.

Целевой аудиторией интерактивного конфигуратора являются:

Менеджеры по продажам: им нужен инструмент, который позволит быстро и точно подбирать оборудование, формировать коммерческие предложения и сокращать время на обслуживание клиентов. Они должны иметь возможность легко ориентироваться в каталоге продукции, настраивать параметры оборудования и получать информацию о его стоимости и наличии на складе.

Технические специалисты: им нужен инструмент, который позволит проверять правильность конфигурации и избегать технических ошибок. Они должны иметь возможность анализировать технические характеристики оборудования, проверять совместимость компонентов и получать информацию о возможных ограничениях.

Конечные клиенты (дилеры, строительные компании, частные лица): Им нужен инструмент, который позволит самостоятельно подобрать оборудование, соответствующее их потребностям, и получить информацию о его характеристиках и стоимости. Они должны иметь возможность визуализировать конечный результат, сравнивать различные варианты конфигурации и получать консультации от специалистов компании.

Уровень технической подготовки целевой аудитории:

Уровень технической подготовки целевой аудитории варьируется в широких пределах. Менеджеры по продажам и конечные клиенты могут не обладать глубокими техническими знаниями, поэтому конфигуратор должен быть интуитивно понятным и простым в использовании. Технические специалисты, напротив, обладают высоким уровнем технической подготовки и нуждаются в инструменте, который предоставляет им доступ к детальной информации о продуктах и их характеристиках.

## 1.2 Обзор и сравнительный анализ существующих решений

# Глава 2. ВЫБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ РАЗРАБОТКИ ИНФОРМАЦИОННОЙ СИСТЕМЫ

В современном мире разработки программного обеспечения выбор правильных инструментов является критическим фактором успеха любого проекта. Удачное сочетание технологий позволяет повысить эффективность разработки, обеспечить стабильность и масштабируемость создаваемого приложения. В рамках данного проекта, целью которого является создание высокопроизводительного веб-приложения с архитектурой клиент-сервер, был выбран следующий стек:

* Сервер: ASP.NET Core Web API
* ORM: Entity Framework (EF) Core
* Клиент: React
* База данных: PostgreSQL
* Контейнеризация: Docker

Ниже представлено подробное обоснование выбора каждой из этих технологий, с учетом их функциональных возможностей, совместимости, преимуществ и роли Docker в разработке и развертывании.

**Предварительное замечание о выборе языка:** для реализации серверной части (ASP.NET Core Web API) был выбран язык C#. Это решение основывается на следующих факторах [4]:

* Зрелость и надежность платформы .NET: C# и .NET предоставляют стабильную и зрелую платформу для разработки корпоративных приложений.
* Производительность: .NET Core (и его более поздние версии) показывает отличную производительность, особенно при работе с веб-приложениями.
* Экосистема и инструменты: .NET имеет богатую экосистему библиотек, фреймворков и инструментов, включая Entity Framework Core, используемый в этом проекте.
* Сообщество и поддержка: Microsoft предоставляет обширную документацию, поддержку и активное сообщество разработчиков C#.
* Типизация: C# - статически типизированный язык, что способствует более раннему выявлению ошибок на этапе разработки и повышает надежность кода.
* Ориентация на объектно-ориентированное программирование: C# предоставляет мощные возможности для объектно-ориентированного программирования, что упрощает разработку сложных приложений.

2.1 Выбор серверного фреймворка

Выбор серверного фреймворка — важный этап разработки, определяющий производительность, масштабируемость и поддерживаемость приложения. Для проекта, где ключевыми требованиями стали обработка высокой нагрузки, кроссплатформенность и минимизация ошибок на этапе компиляции, рассматривались три решения: ASP.NET Core, Node.js (Express.js) и PHP (Laravel/Symfony). Сравнение проведено по параметрам: производительность, экосистема, типизация, интеграция с ORM и поддержка со стороны разработчика.

**Рассмотренные варианты:**

1. ASP.NET Core — кроссплатформенный фреймворк от Microsoft с поддержкой C#.
2. Node.js (Express.js) — JavaScript-фреймворк для создания высокопроизводительных API.
3. PHP (Laravel/Symfony) — популярные фреймворки для веб-разработки.

Сравнительный анализ ASP.NET Core, Node.js и PHP приведен в табл. 1.

Таблица 1 — Сравнительный анализ серверных фреймворков [2]

| **Характеристика** | **ASP.NET Core** | **Node.js (Express.js)** | **PHP (Laravel/Symfony)** |
| --- | --- | --- | --- |
| Язык | C# | JavaScript/TypeScript | PHP |
| Производительность | Высокая | Высокая (при оптимизации) | Средняя (зависит от версии PHP и фреймворка) |
| Кроссплатформенность | Да | Да | Да |
| Типизация | Статическая | Динамическая | Динамическая |
| Сложность разработки | Средняя (сложность C#) | Низкая (для JS-разработчиков) | Низкая (синтаксис PHP) |
| Экосистема | Богатая (NuGet) | Богатая (npm) | Очень богатая (Composer) |
| Поддержка Microsoft | Да | Нет | Нет |
| ORM интеграция | Отличная (EF Core) | Sequelize, Mongoose, TypeORM и другие | Doctrine, Eloquent (Laravel) и другие |
| Масштабируемость | Высокая | Высокая | Средняя (зависит от архитектуры) |

**Обоснование выбора:**

При разработке серверной части ключевым требованием стала высокая производительность, способная обеспечить стабильную работу приложения в условиях высокой нагрузки. Среди рассмотренных фреймворков — ASP.NET Core, Node.js (Express.js) и PHP (Laravel/Symfony) — каждый обладает уникальными особенностями, однако выбор остановлен на ASP.NET Core. Ниже приведено обоснование этого решения.

**ASP.NET Core** (C#, Microsoft) отличается высокой скоростью обработки запросов, что подтверждается независимыми тестами производительности. Его кроссплатформенность позволяет разворачивать приложения на различных операционных системах, включая Linux и macOS, что особенно важно для гибкой инфраструктуры. Интеграция с Entity Framework Core упрощает работу с базами данных, а статическая типизация C# снижает риск ошибок на этапе компиляции.

**Node.js с Express.js** (JavaScript) популярен благодаря простоте освоения для фронтенд-разработчиков и асинхронной модели, эффективной для I/O-операций. Однако для достижения максимальной производительности требуется глубокая оптимизация, например, использование кластеризации или потоков.

**PHP-фреймворки** (Laravel, Symfony) обеспечивают быстрое создание MVP за счёт удобного синтаксиса и богатой экосистемы. Тем не менее, их производительность уступает .NET и Node.js, особенно в сценариях с интенсивными вычислениями. Масштабирование PHP-решений часто требует дополнительных инструментов, таких как кеширование или балансировка нагрузки.

2.2 Обоснование выбора ORM

Object-Relational Mapping (ORM) — инструмент, который преобразует объекты приложения в структуры реляционной базы данных и обратно, упрощая взаимодействие с БД. Для проекта конфигуратора важно минимизировать ручное написание SQL-запросов, обеспечить безопасность данных и ускорить разработку.

Для взаимодействия с базой данных рассматривались три подхода: Entity Framework Core, Dapper и ADO.NET.

1. **Entity Framework Core** — ORM от Microsoft, интегрированный с ASP.NET Core. Поддерживает миграции, LINQ-запросы и автоматическую генерацию SQL.
2. **Dapper** — легковесный микроОRM, требующий ручного написания SQL, но обеспечивающий высокую производительность.
3. **ADO.NET** — низкоуровневая технология для прямого управления подключениями и запросами, что повышает контроль, но усложняет разработку.

Сравнительный анализ EF Core, Dapper иADO.NET приведен в табл. 2.

Таблица 2 — Сравнительный ORM-инструментов [2]

| **Характеристика** | **Entity Framework Core** | **Dapper** | **ADO.NET** |
| --- | --- | --- | --- |
| Уровень абстракции | Высокий (объектно-ориентированный) | Низкий (близкий к SQL) | Низкий (прямой доступ к базе данных) |
| Простота разработки | Упрощает разработку, снижает ручной ввод SQL | Требует написания SQL-запросов вручную | Требует написания SQL-запросов и ручного управления соединениями |
| Производительность | Средняя | Высокая | Средняя |
| Читаемость кода | Высокая | Может быть низкой (зависит от сложности SQL) | Низкая |

**Обоснование выбора:**

Entity Framework Core был выбран для абстракции работы с базой данных. Его высокоуровневый подход позволяет описывать запросы через LINQ, минимизируя ручное написание SQL. Например, LINQ-запросы автоматически транслируются в оптимизированный SQL, снижая риск синтаксических ошибок.

Миграции в EF Core упрощают управление схемой БД. При изменении модели данных фреймворк генерирует скрипты для обновления таблиц, что критично для итеративной разработки. Интеграция с ASP.NET Core через DbContext обеспечивает простую настройку и поддержку транзакций. Несмотря на меньшую производительность в некоторых сценариях, EF Core выигрывает за счёт удобства и скорости разработки.

2.3 Выбор клиентского фреймворка

Клиентская часть конфигуратора должна обеспечивать динамичное обновление интерфейса при изменении параметров оборудования. Требуется высокая производительность и возможность интеграции с REST API.

**Рассмотренные варианты:**

1. React — библиотека с компонентной архитектурой и виртуальным DOM.
2. Angular — полноценный фреймворк с встроенными инструментами, но сложной архитектурой.
3. Vue.js — прогрессивный фреймворк с простым синтаксисом, но меньшей экосистемой.

Сравнительный анализ React, Angular и Vue.js представлен в табл. 3.

Таблица 3 — Сравнительный анализ клиентских фреймворков [4]

| **Характеристика** | **React** | **Angular** | **Vue.js** |
| --- | --- | --- | --- |
| Архитектура | Компонентно-ориентированный (с использованием JSX) | Компонентно-ориентированный (TypeScript, MVC/MVVM) | Компонентно-ориентированный (с использованием шаблонов или JSX) |
| Язык | JavaScript/TypeScript | TypeScript | JavaScript/TypeScript |
| Кривая обучения | Более пологая (основы легко освоить) | Более крутая (большой объем для изучения) | Более пологая (проще, чем Angular) |
| Размер приложения | Может быть минимальным (при грамотной организации) | Большой (за счет функциональности) | Меньше, чем Angular |
| Производительность | Высокая (виртуальный DOM) | Высокая (оптимизация изменений DOM) | Высокая (виртуальный DOM) |

|  |  |  |  |
| --- | --- | --- | --- |
| Экосистема | Богатая (широкий выбор библиотек для управления состоянием, маршрутизации и т.д.) | Богатая (развитая экосистема, поддержка Google) | Хорошая (широкий выбор библиотек, активное сообщество) |
| Управление состоянием | Redux, Context API, Zustand и другие | NgRx, NgRX/Store (похоже на Redux), Services | Vuex, Pinia и другие |
| Шаблонизация | JSX (JavaScript XML) | HTML-шаблоны с директивами | HTML-шаблоны, JSX, или Render-функции |
| Связывание данных | Одностороннее | Двустороннее | Двустороннее (по умолчанию) и одностороннее (при необходимости) |
| Масштабируемость | Высокая (модульность, компонентный подход) | Высокая (структурированность, поддержка больших проектов) | Высокая (компонентный подход, подходит для проектов разного масштаба) |

Продолжение Таблицы 3 — Сравнительный анализ клиентских фреймворков

**Обоснование выбора React:**

* **Компонентный подход.** Позволяет разбить интерфейс на переиспользуемые компоненты. Например, блок выбора параметров помещения (площадь, количество людей) может быть отдельным компонентом, что упрощает тестирование и поддержку.
* **Виртуальный DOM.** Оптимизирует рендеринг при частых изменениях данных. Например, при динамическом обновлении списка подобранного оборудования React перерисовывает только измененные элементы, а не весь интерфейс.
* **Интеграция с REST API.** Использование fetch или Axios для взаимодействия с ASP.NET Core Web API делает процесс передачи данных прозрачным. Например, после выбора параметров конфигурации React отправляет POST-запрос на сервер и отображает результат без перезагрузки страницы.

Angular отвергнут из-за избыточной сложности для проекта среднего масштаба, Vue.js — из-за меньшего количества готовых решений для управления состоянием.

2.4 Выбор системы управления базами данных

Система управления базами данных (СУБД) — это комплекс программных и языковых средств, который отвечает за хранение и управление информацией в базах данных. [1]

Она предоставляет инструменты для организации, хранения, извлечения, обновления и удаления данных, а также для обеспечения их безопасности и целостности.

**Рассмотренные варианты:**

1. PostgreSQL — объектно-реляционная СУБД с поддержкой JSONB и ACID-транзакций.
2. MySQL — популярная СУБД, но с ограниченной функциональностью для сложных данных.
3. MS SQL Server — мощное решение, но требующее лицензирования.

Сравнительный анализ PostgreSQL, MySQL и MS SQL Server в табл. 4.

Таблица 4 — Сравнительный анализ СУБД [9]

| **Характеристика** | **PostgreSQL** | **MySQL** | **MS SQL Server** |
| --- | --- | --- | --- |
| Лицензия | Открытый исходный код (BSD) | Открытый исходный | Средняя |
| Надежность | Высокая (транзакции, ACID, WAL) | Высокая (но зависит от конфигурации) | Высокая (транзакции, ACID) |
| Производительность | Высокая (оптимизация запросов, многопоточность) | Высокая (оптимизация для чтения, масштабируемость) | Высокая (оптимизация, инструменты мониторинга) |
| Сложные типы данных | Поддержка (JSONB, массивы, PostGIS) | Ограниченная | Поддержка (XML, пространственные типы) |
| Поддержка Windows | Отличная | Отличная | Отличная |
| Функциональность | Богатая (поддержка оконных функций, рекурсивных запросов) | Хорошая | Богатая (аналитические функции, интеграция с .NET) |

**Обоснование выбора:**

PostgreSQL была выбрана из-за надёжности (поддержка ACID, WAL) и расширяемости (JSONB, PostGIS). Например, JSONB позволяет хранить динамические настройки продуктов, а PostGIS — работать с геоданными. Оптимизатор запросов и параллельное выполнение операций обеспечивают высокую производительность даже при больших объёмах данных.

**Надежность и безопасность**  
PostgreSQL гарантирует целостность данных благодаря поддержке ACID-транзакций (Atomicity, Consistency, Isolation, Durability). Это означает, что даже в случае сбоев или параллельных операций данные остаются непротиворечивыми. Например, при одновременном обновлении каталога продукции несколькими менеджерами транзакции предотвращают конфликты и потерю информации. Дополнительную надежность обеспечивает механизм WAL (Write-Ahead Logging), который фиксирует все изменения перед их записью в основную базу, что критично для восстановления данных после аварийных ситуаций.

**Гибкость работы с данными**Одним из ключевых преимуществ PostgreSQL является поддержка JSONB — формата для хранения вложенных и полуструктурированных данных. Это позволяет гибко описывать технические характеристики оборудования. Например, параметры вентиляционной системы могут включать как статические значения (мощность, уровень шума), так и динамические (список фильтров, габариты)

Индексация JSONB-полей ускоряет поиск по таким данным, делая запросы вроде «Найти все устройства мощностью более 1000 Вт» эффективными даже для больших каталогов. Кроме того, PostgreSQL поддерживает расширения, такие как PostGIS для работы с геопространственными данными. Это открывает возможности для анализа регионального спроса или интеграции с картографическими сервисами.

**Производительность и оптимизация**  
Встроенный оптимизатор запросов PostgreSQL автоматически выбирает наиболее эффективные пути выполнения операций, что особенно важно для сложных выборок. Например, формирование отчета о популярности оборудования в разных категориях выполняется быстрее, чем в MySQL, благодаря параллельной обработке запросов и использованию многопоточности. Механизм MVCC (Multiversion Concurrency Control) обеспечивает высокую скорость работы при одновременном доступе множества пользователей, что критично для SaaS-решений.

2.5 Использование Docker для разработки и развертывания

Контейнеризация обеспечивает единообразие сред разработки, тестирования и production, что снижает риск ошибок из-за различий в окружении.

**Рассмотренные варианты:**

1. Docker — контейнеризация приложений с изоляцией зависимостей.
2. Виртуальные машины (VM) — полная виртуализация, но с высокими накладными расходами.
3. Ручное развертывание — установка приложения напрямую на сервер.

Сравнительный анализ Docker, VM и традиционного развертывания представлено в табл. 5.

Таблица 5 — Сравнительный анализ методов развертывания

| **Характеристика** | **Docker** | **Виртуальные машины (VM)** | **Традиционное развертывание (на сервере)** |
| --- | --- | --- | --- |
| Изоляция | Высокая (контейнеры) | Высокая (VM) | Низкая (зависимости могут конфликтовать) |
| Ресурсы | Эффективное использование (меньше ресурсов, чем VM) | Высокое потребление ресурсов (каждая VM - отдельная ОС) | Зависит от конфигурации сервера и приложений |
| Переносимость | Высокая (Docker images) | Высокая (VM images) | Низкая (зависит от ОС и конфигурации) |
| Масштабируемость | Легкая (быстрое создание и удаление контейнеров) | Более сложная (запуск новых VM) | Сложная (требует настройки и развертывания приложений) |
| Управление | Упрощенное (Docker Compose, Kubernetes) | Сложное (управление несколькими VM) | Сложное (требует ручной настройки серверов) |

|  |  |  |  |
| --- | --- | --- | --- |
| Время развертывания | Быстрое (секунды/минуты) | Долгое (минуты/часы) | Зависит от сложности развертывания |
| Автоматизация | Высокая (CI/CD, Dockerfiles) | Высокая (но требует дополнительных инструментов) | Средняя (требует скриптов и инструментов автоматизации) |
| Размер | Небольшой | Большой (ОС, приложения, зависимости) | Зависит от размера приложений и зависимостей |

Продолжение Таблицы 5 — Сравнительный анализ методов развертывания

**Преимущества использования Docker:**

* Изоляция: Контейнеры изолированы друг от друга и от хост-системы, что предотвращает конфликты зависимостей и обеспечивает стабильность работы.
* Переносимость: Контейнеры Docker легко переносятся между различными средами разработки, тестирования и production, что упрощает процесс развертывания.
* Масштабируемость: Docker позволяет легко масштабировать приложение путем создания нескольких контейнеров.
* Управление зависимостями: Docker упрощает управление зависимостями приложения, позволяя упаковывать все необходимые библиотеки и инструменты в контейнер.
* Автоматизация: Docker автоматизирует процесс сборки, тестирования и развертывания, что сокращает время разработки и улучшает качество.
* Воспроизводимость: Контейнеры Docker обеспечивают воспроизводимость окружения, что упрощает отладку и тестирование.
* В рамках проекта Docker будет использоваться для контейнеризации:
* ASP.NET Core Web API: для обеспечения стабильности и удобства развертывания серверной части.
* PostgreSQL: для обеспечения согласованной среды разработки и развертывания базы данных.
* React приложение (опционально): для удобства разработки и развертывания клиентской части (использование Docker для frontend зависит от конкретной архитектуры и предпочтений разработчика).

**Вывод:** использование Docker позволяет обеспечить:

Согласованную среду - упрощает настройку и поддержание среды разработки, делая ее одинаковой для всех разработчиков.

Быстрое развертывание - ускоряет процесс развертывания на различных серверах.

Масштабируемость - упрощает масштабирование приложения при увеличении нагрузки. Docker является отличным выбором для современных веб-приложений, обеспечивая гибкость, эффективность и упрощенное управление.

Вывод:

Таким образом, выбранный стек инструментальных средств (ASP.NET Core Web API, Entity Framework Core, React, PostgreSQL, Docker) обеспечивает высокую производительность, надежность и удобство разработки на всех уровнях программного продукта.

# Глава 3. ПРОЕКТИРОВАНИЕ И РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ ТЕСТИРОВАНИЯ ЗНАНИЙ ОБУЧАЮЩИХСЯ

Реализация информационной системы для автоматизации подбора вентиляционного оборудования в компании ТИОН осуществлялась поэтапно, с акцентом на практическую применимость, надёжность и удобство для конечных пользователей. В данной главе подробно рассматриваются ключевые аспекты проектирования, разработки, интеграции и внедрения системы, а также анализируются результаты её эксплуатации.

3.1 Анализ требований к конфигуратору

Данный раздел содержит анализ требований к конфигуратору, разделенных на функциональные и нефункциональные требования.

**Функциональные требования**

Функциональные требования описывают функциональность, которую должен предоставлять конфигуратор для пользователей.

1. **Управление каталогом продукции:** обеспечение возможностей добавления, редактирования и удаления продукции из каталога, а также загрузки изображений и технических характеристик. Доступно только администраторам.

* Добавление продукции: реализация возможности добавления новой продукции в каталог с указанием следующих параметров: название, артикул/код, описание (краткое и подробное), категория, цена, производитель. Должна быть обеспечена возможность сохранения данных о новом продукте в базе данных.
* Редактирование продукции: реализация возможности изменения информации о существующей продукции в каталоге. Предоставление возможности редактирования всех полей, доступных при добавлении продукта. Изменения должны быть сохранены в базе данных.
* Удаление продукции: реализация возможности удаления продукции из каталога с обязательным подтверждением действия. Удаление должно удалять данные продукта из базы данных.
* Загрузка изображений: предоставление возможности прикрепления изображений к продукции, с поддержкой распространенных графических форматов (например, JPG, PNG, GIF) и возможностью оптимизации изображений для отображения на веб-страницах. Предусмотреть хранение изображений (на сервере).
* Дополнительные возможности: предусмотреть категоризацию продукции, поиск по каталогу (по названию, артикулу, описанию и другим параметрам), фильтрацию по каталогу (по категориям, производителям и другим параметрам).

1. **Конфигуратор оборудования:**списки, переключатели, ползунки, текстовые поля) и визуального отображения выбранных значений.

* Ввод данных пользователем: Реализация текстовых полей ввода и выпадающих списков для ввода параметров оборудования.
* Подбор оборудования: Реализация алгоритма, который на основе введенных данных подбирает подходящие модели оборудования из каталога.
* Отображение результата: Отображение карточек подходящих продуктов с информацией (название, изображение, краткое описание, цена).

1. **ИИ-Консультант:** реализация чат-бота, способного вести диалог с пользователем в реальном времени. Возможность подбора оборудования через диалог с чат-ботом.
2. **Управление справочниками:** реализация возможности добавления, редактирования и удаления: возможных характеристик, требований к помещениям, типов устройств. Доступно только для администраторов.
3. **Корзина:** обеспечение возможности добавлять товары с каталога и конфигуратора в корзину
4. **Заказы:** оформление заказов для выбранных товаров.

**Нефункциональные требования**

Нефункциональные требования определяют характеристики системы, которые не относятся непосредственно к функциональности, но влияют на ее качество.

**Производительность:** конфигуратор должен обеспечивать быструю загрузку данных и мгновенный отклик на действия пользователя. Время загрузки каталога продукции и данных о конфигурации должно быть минимальным. Необходимо использовать оптимизированные запросы к базе данных, кэширование данных, оптимизацию изображений и технологию ленивой загрузки. Система должна обеспечивать минимальное время отклика на действия пользователя и быть способной обрабатывать большое количество данных и пользователей без снижения производительности.

**Безопасность:** конфигуратор должен обеспечивать защиту от несанкционированного доступа и взлома. Необходимо реализовать аутентификацию и авторизацию пользователей, защиту от SQL-инъекций, межсайтового скриптинга (XSS), CSRF, шифрование данных и регулярные обновления безопасности. Должна быть предусмотрена система резервного копирования и восстановления данных.

**Удобство использования:** конфигуратор должен быть интуитивно понятным и простым в использовании даже для пользователей с небольшим опытом работы с компьютером. Интерфейс должен быть понятным, логичным и адаптированным для различных устройств. Необходимо предоставлять обратную связь пользователям и обеспечить наличие документации и помощи.

3.2. Фрагменты спецификаций требований в виде диаграмм

Диаграмма прецедентов служит инструментом для описания системы на начальном, концептуальном этапе разработки. Она демонстрирует основные функции системы и взаимодействие между пользователями (акторами) и этими функциями. Благодаря такой диаграмме становится возможным проанализировать ключевые сценарии использования системы, а также определить, какие действия могут выполнять различные участники процесса. Это позволяет лучше понять требования к системе и заложить прочную основу для дальнейшей разработки. Рис.1.

Изображение выглядит как текст, снимок экрана, круг

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1. Use case диаграмма системы отдела кадров университета

Представлена диаграмма классов, которая была разработана на основании анализа Use-Case диаграммы. Данная диаграмма демонстрирует структуру системы и взаимосвязи между основными сущностями, выделенными из сценариев использования. Она позволяет более детально описать внутреннее строение системы и роль каждого класса в реализации функциональных требований, заложенных в Use-Case модели. Рис.2.

Изображение выглядит как текст, снимок экрана, Шрифт, черный

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 2. Диаграмма классов

Представлена диаграмма компонентов, которая демонстрирует логическое разделение системы на отдельные модули и их взаимодействие между собой. Эта диаграмма позволяет получить обобщенное понимание структуры сложной системы, представив ее в виде взаимосвязанных компонентов, каждый из которых выполняет свою конкретную роль. Таким образом, она упрощает восприятие архитектуры системы, делая ее более доступной для анализа и понимания. Рис.3.

Изображение выглядит как текст, снимок экрана, визитная карточка, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 3. Диаграмма компонентов

Демонстрируется диаграмма состояний, охватывающая работу всей системы и отражающая последовательность действий при взаимодействии с пользователями. Данная диаграмма подробно описывает жизненный цикл системы, показывая, как она переходит из одного состояния в другое в зависимости от выполняемых операций и событий, инициируемых пользователями. Рис.4.

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 4. Диаграмма состояний

Демонстрируется диаграмма размещения, которая позволяет определить физическое расположение компонентов разрабатываемого программного продукта. С ее помощью становится возможным визуализировать, на каких узлах или устройствах будут развернуты различные части системы, а также как эти компоненты взаимодействуют между собой в пределах инфраструктуры. Рис.5

Изображение выглядит как текст, снимок экрана, Прямоугольник, линия

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 5. Диаграмма размещений

3.2 Проектирование архитектуры системы

Архитектура системы конфигуратора состоит из трех основных компонентов: Frontend, Backend и Database [12].

**Frontend (React.js)** отвечает за предоставление пользовательского интерфейса для взаимодействия с конфигуратором. Он разработан с использованием React.js, JavaScript (ES6+), HTML и CSS (или CSS-in-JS библиотеки). Для управления состоянием приложения может использоваться Redux или Context API. Основные функции включают отображение каталога продукции и интерфейса конфигурации, обработку пользовательского ввода, взаимодействие с Backend API, отображение результатов конфигурации и обеспечение удобного интерфейса. При проектировании учитывается компонентная архитектура React.js, оптимизация производительности, адаптивный дизайн и обработка ошибок.

**Backend (ASP.NET Core Web API)** предоставляет API для доступа к данным и выполнения бизнес-логики. Он разработан с использованием ASP.NET Core, C#, .NET Runtime и Entity Framework Core для работы с базой данных. Основные функции включают обработку HTTP-запросов, авторизацию и аутентификацию пользователей, взаимодействие с базой данных, реализацию бизнес-логики конфигуратора, формирование ответов в формате JSON и предоставление API для управления каталогом продукции. При проектировании учитывается RESTful API, использование Dependency Injection, реализация обработки ошибок и, использование индексов, резервное копирование, безопасность базы данных и выбор типа данных для каждого поля.

**Схема взаимодействия компонентов:**

1. Пользователь взаимодействует с Frontend (React.js).
2. Frontend отправляет HTTP-запросы к Backend API (ASP.NET Core Web API).
3. Backend обрабатывает запросы, взаимодействует с базой данных (PostgreSQL).
4. Backend возвращает ответы в формате JSON Frontend.
5. Frontend отображает данные пользователю.

**Преимущества данной архитектуры [12]:**

* Разделение ответственности: Каждый компонент отвечает за свою конкретную задачу, упрощая разработку, тестирование и поддержку.
* Масштабируемость: Каждый компонент может быть масштабирован независимо.
* Гибкость: Можно легко заменить один компонент другим.
* Поддержка современных технологий: Использование React.js, ASP.NET Core и PostgreSQL обеспечивает современные подходы к разработке.
* Независимое развертывание: Каждый компонент можно развертывать независимо.

Архитектура системы представлена на рисунке 1.

Изображение выглядит как текст, снимок экрана, Прямоугольник, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 1. Архитектура системы

3.3 Проектирование базы данных

**Проектирование инфологической модели БД**

Проектирование инфологической модели является основной задачей при создании БД. Цель инфологической модели – обеспечение наиболее естественных для человека способов сбора и представления той или иной информации, которую предполагается хранить в создаваемой базе. Поэтому инфологическую модель данных пытаются строить по аналогии с естественным языком (последний не может быть использован в чистом виде из-за сложности компьютерной обработки текстов и неоднозначности любого естественного языка). Основными конструктивными элементами инфологических моделей являются сущности, связи между ними и их свойства [10].

Поэтому необходимо четко поставить цель данной БД, а также установить, какую информацию получает пользователь в результате работы с программой.

Для данной ИС были выделены следующие сущности:

1. Устройство
2. Тип устройства
3. Тип помещения
4. Пользователь
5. Корзина
6. Заказ
7. УстройстваКорзина
8. УстройстваЗаказ

В ходе проектирования была разработана следующая концептуальная модель базы данных:

Изображение выглядит как текст, снимок экрана, Шрифт, дизайн

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 2. Концептуальная ER-Диаграмма

**Проектирование даталогической модели БД**

Проектирование даталогической модели - важный этап в проектировании базы данных. На этом этапе важно правильно выделить сущности и описать их атрибуты. Ошибка на этом этапе может обернуться разработчику значительными потерями времени и сил в дальнейшем.

Датологическая модель - модель, описывающая логику организации данных. Датологическое проектирование заключается в проектировании логической структуры БД. Таким образом, главное отличие даталогической модели от инфологической состоит в том, что инфологическая модель хранит в себе всю информацию о предметной области, необходимую и достаточную для проектирования базы данных, но она не привязана к определенной СУБД [5].

Даталогическое проектирование сводится к следующим этапам:

1. Определение таблиц;

2. Определение полей таблиц;

3. Определение типов данных в соответствии с выбранной СУБД;

4. Определение длины каждого поля таблиц;

5. Определение обязательности каждого поля;

6. Определение индексации каждого поля.

Далее будут представлены сформированные таблицы:

Таблица 1 - Устройство, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Название | Текстовый | 255 значений, обязательное поле: да |
| Описание | Текстовый | 255 значений, обязательное поле: нет |
| Путь к изображению | Текстовый | 255 значений, обязательное поле: нет |
| Мощность | Числовой | Длинное целое, обязательное поле: да |
| Уровень шума | Числовой | Длинное целое, обязательное поле: да |
| Максимальный поток воздуха | Числовой | Длинное целое, обязательное поле: да |
| Цена | Числовой | Длинное целое, обязательное поле: да |
| ID Тип устройства | Числовой | Длинное целое, обязательное поле: да |

Таблица 2 – Тип устройства, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Название | Текстовый | 255 значений, обязательное поле: да |
| Описание | Текстовый | 255 значений, обязательное поле: нет |

Таблица 3 – Требование к помещению, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Коэффициент площади | Числовой | Дробное, обязательное поле: да |
| Коэффициент людей | Числовой | Дробное, обязательное поле: да |

Таблица 4 – Пользователь, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Email | Текстовый | 255 значений, обязательное поле: да |
| Пароль | Текстовый | 255 значений, обязательное поле: да |
| Роль | Текстовый | 255 значений, обязательное поле: да |
| Дата создания | Дата/время | Обязательное поле: да |

Таблица 5 - УстройстваКорзина, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| ID Корзина | Числовой | Длинное целое, обязательное поле: да |
| ID Устройство | Числовой | Длинное целое, обязательное поле: да |
| Сумма | Числовой | Длинное целое, обязательное поле: да |
| Дата создания | Дата/время | Обязательное поле: да |
| Дата изменения | Дата/время | Обязательное поле: да |

Таблица 6 - Корзина, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| ID Пользователь | Числовой | Длинное целое, обязательное поле: да |
| ID УстройстваКорзина | Числовой | Длинное целое, обязательное поле: да |
| Сумма | Числовой | Длинное целое, обязательное поле: да |
| Дата создания | Дата/время | Обязательное поле: да |
| Дата изменения | Дата/время | Обязательное поле: да |

Таблица 7 - УстройстваЗаказ, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| ID Заказ | Числовой | Длинное целое, обязательное поле: да |
| ID Устройство | Числовой | Длинное целое, обязательное поле: да |
| Сумма | Числовой | Длинное целое, обязательное поле: да |
| Единичная цена | Числовой | Длинное целое, обязательное поле: да |
| Дата создания | Дата/время | Обязательное поле: да |

Таблица 8 – Заказ, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| ID Пользователь | Числовой | Длинное целое, обязательное поле: да |
| ID УстройстваЗаказ | Числовой | Длинное целое, обязательное поле: да |
| Сумма | Числовой | Длинное целое, обязательное поле: да |
| Адрес | Текстовый | 255 значений, обязательное поле: да |
| Номер телефона | Текстовый | 255 значений, обязательное поле: да |
| Email | Текстовый | 255 значений, обязательное поле: да |
| Примечания | Текстовый | 255 значений, обязательное поле: нет |
| Статус | Текстовый | 255 значений, обязательное поле: да |
| Дата создания | Дата/время | Обязательное поле: да |
| Дата изменения | Дата/время | Обязательное поле: да |

Таблица 9 – Возможные характеристики, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Название | Текстовый | 255 значений, обязательное поле: да |
| Единица измерения | Текстовый | 255 значений, обязательное поле: нет |
| ID Тип устройства | Числовой | Длинное целое, обязательное поле: да |
| Тип данных | Текстовый | 255 значений, обязательное поле: да |
| Обязательность | Булевой | обязательное поле: да |
| ID Характеристика | Числовой | Длинное целое, обязательное поле: нет |

Таблица 10 – Характеристика, ключевое поле «ID»

|  |  |  |
| --- | --- | --- |
| Имя поля | Тип данных | Ограничения |
| ID | Счетчик | Номер по порядку уникален |
| Значение | Текстовый | 255 значений, обязательное поле: да |
| ID Тип устройства | Числовой | Длинное целое, обязательное поле: да |
| ID Возможные характеристики | Числовой | Длинное целое, обязательное поле: нет |

* 1. **Разработка серверной части приложения**

На первом этапе была реализована серверная часть системы на платформе ASP.NET Core. Основное внимание уделялось построению надёжного API, обеспечивающего взаимодействие между клиентским приложением, базой данных и внешними сервисами.

Для организации доступа к данным на уровне серверной части приложения было решено использовать **Entity Framework Core (EF Core)** – современный и мощный ORM (Object-Relational Mapper) от Microsoft. EF Core позволяет взаимодействовать с базами данных как с набором объектов C#, существенно упрощая разработку и поддержку приложения.

В качестве подхода к работе с EF Core был выбран **Code First**.

**EF Core Code First: Создание базы данных из кода**

В подходе Code First, разработчик сначала определяет структуру базы данных в виде классов C#. Затем EF Core использует эти классы для создания схемы базы данных. Это предоставляет большую гибкость и контроль над процессом, особенно когда вы уже имеете четкое представление о структуре данных, необходимой для вашего приложения.

**Преимущества Code First:**

* **Гибкость:** Полный контроль над структурой базы данных.
* **Удобство:** Определение структуры данных в терминах объектно-ориентированного программирования (ООП).
* **Миграции:** Поддержка миграций позволяет легко изменять схему базы данных и переносить эти изменения между различными средами (разработка, тестирование, продакшен).

**Архитектура: Слои, паттерны и Dependency Injection**

Для организации кода серверной части приложения была выбрана многослойная архитектура, с использованием следующих ключевых компонентов:

**Модели (Models):** Представляют собой классы C#, которые отражают структуру таблиц в базе данных. Они содержат свойства, соответствующие полям таблицы, и используются EF Core для взаимодействия с базой данных. В данном контексте, модели представляют основные сущности предметной области: Device, DeviceType, RoomType, User, Order, Cart и т.д.

**Репозитории (Repositories):** Репозитории реализуют паттерн Repository, который инкапсулирует логику доступа к данным. Каждый репозиторий отвечает за работу с определенным типом сущностей (например, IdeviceRepository для работы с устройствами). Репозитории предоставляют методы для выполнения операций CRUD (Create, Read, Update, Delete) и других операций, связанных с данными. Они скрывают детали реализации доступа к данным от других слоев приложения.

**Сервисы (Services):** Содержат бизнес-логику приложения. Они координируют работу репозиториев и выполняют операции, связанные с обработкой данных, взаимодействием с внешними сервисами, и другими задачами, не относящимися непосредственно к отображению данных или управлению запросами. Сервисы используют репозитории для доступа к данным и могут выполнять сложные операции, такие как расчет стоимости заказа, подбор оптимального оборудования, и т.д. Примеры сервисов: AddDeviceService, UpdateDeviceService, ImageService, DeviceConfiguratorService, GigaChatService.

**Контроллеры (Controllers):** Принимают HTTP-запросы от клиентского приложения, вызывают соответствующие методы сервисов для обработки запроса, и возвращают HTTP-ответ с результатами. Контроллеры отвечают за маршрутизацию запросов, аутентификацию и авторизацию пользователей, валидацию данных и формирование ответов в формате JSON. Примеры контроллеров: DevicesController, AIConsultantController.

Такое разделение на слои помогает изолировать различные части приложения друг от друга, что упрощает тестирование, поддержку и расширение функциональности. Для обеспечения гибкости и расширяемости системы, использован механизм **Dependency Injection (DI)**, который позволяет легко заменять реализации репозиториев и сервисов, а также упрощает тестирование.

Для работы с данными использовался Entity Framework Core, что позволило упростить реализацию CRUD-операций и обеспечить поддержку миграций.

Для обеспечения гибкости и расширяемости системы были разработаны следующие модели данных:

public sealed class Device

    {

        public int Id { get; set; }

        [Required]

        public string Name { get; set; }

        public string Description { get; set; }

        public string ImagePath { get; set; }

        public int PowerConsumption { get; set; }

        public int NoiseLevel { get; set; }

*//Максимальный расход/поток воздуха*

        [Required]

        public int MaxAirflow { get; set; }

        [Range(1, int.MaxValue)]

        public int Price { get; set; }

        public int DeviceTypeId { get; set; }

        [ForeignKey(“DeviceTypeId”)]

        public required DeviceType DeviceType { get; set; }

        public Icollection<Characteristic>? Characteristics { get; set; }

    }

public class DeviceType

{

    public int Id { get; set; }

    [Required]

    [StringLength(100)]

    public string Name { get; set; }

    [StringLength(500)]

    public string? Description { get; set; }

    public Icollection<PossibleCharacteristic> PossibleCharacteristics { get; set; }

}

public class PossibleCharacteristic

{

    public int Id { get; set; }

    [Required]

    [StringLength(100)]

    public string Name { get; set; }

    [StringLength(200)]

    public string? Unit { get; set; } *// Единица измерения (например, Вт, Дб)*

    [Required]

    public int DeviceTypeId { get; set; }

    [ForeignKey(“DeviceTypeId”)]

    public DeviceType DeviceType { get; set; }

    [Required]

    [StringLength(20)]

    public string Type { get; set; } *// ‘bool’, ‘number’, ‘string’*

    public bool IsRequired { get; set; }

    public Icollection<Characteristic> Characteristics { get; set; }

}

public class Characteristic

{

    public int Id { get; set; }

    [Required]

    public string Value { get; set; }

    [Required]

    public int DeviceId { get; set; }

    [ForeignKey(“DeviceId”)]

    public Device Device { get; set; }

    [Required]

    public int PossibleCharacteristicId { get; set; }

    [ForeignKey(“PossibleCharacteristicId”)]

    public PossibleCharacteristic PossibleCharacteristic { get; set; }

}

public class User

    {

        public int Id { get; set; }

        [Required]

        [EmailAddress]

        public string Email { get; set; }

        [Required]

        public string PasswordHash { get; set; }

        [Required]

        public string Role { get; set; } *// “User” и “Admin”*

        public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

    }

public class RoomType

    {

        public int Id { get; set; }

        [Required]

        public string Name { get; set; }

*//Коэф. Куб. метров воздуха в час для каждого квардратного метра (м³/час на м²)*

        [Required]

        public double AreaCoefficient { get; set; }

*//Коэф. Куб. метров воздуха в час для каждого человека (м³/час на человека)*

        [Required]

        public double PeopleCoefficient { get; set; }

    }

public class Order

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int UserId { get; set; }

    [ForeignKey(“UserId”)]

    public User User { get; set; }

    public List<OrderItem> Items { get; set; } = new();

    public decimal TotalPrice => Items.Sum(item => item.Quantity \* item.Device.Price);

    [Required]

    public string ShippingAddress { get; set; }

    [Required]

    public string PhoneNumber { get; set; }

    [Required]

    public string Email { get; set; }

    public string? Notes { get; set; }

    public OrderStatus Status { get; set; } = OrderStatus.Pending;

    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

    public DateTime? UpdatedAt { get; set; }

}

public class OrderItem

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int OrderId { get; set; }

    [ForeignKey(“OrderId”)]

    public Order Order { get; set; }

    [Required]

    public int DeviceId { get; set; }

    [ForeignKey(“DeviceId”)]

    public Device Device { get; set; }

    [Required]

    [Range(1, int.MaxValue)]

    public int Quantity { get; set; }

    [Required]

    public decimal UnitPrice { get; set; }

    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

}

public class Cart

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int UserId { get; set; }

    [ForeignKey(“UserId”)]

    public User User { get; set; }

    public List<CartItem> Items { get; set; } = new();

    public decimal TotalPrice => Items.Sum(item => item.Quantity \* item.Device.Price);

    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

    public DateTime? UpdatedAt { get; set; }

}

public class CartItem

{

    [Key]

    public int Id { get; set; }

    [Required]

    public int CartId { get; set; }

    [ForeignKey(“CartId”)]

    public Cart Cart { get; set; }

    [Required]

    public int DeviceId { get; set; }

    [ForeignKey(“DeviceId”)]

    public Device Device { get; set; }

    [Required]

    [Range(1, int.MaxValue)]

    public int Quantity { get; set; }

    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

    public DateTime? UpdatedAt { get; set; }

}

В рамках данной архитектуры, контроллеры (Controllers) получают запросы от клиента, передают их в сервисы (Services), которые используют репозитории (за кодом репозиториев стоит обращаться в документацию, так как он занимает довольно много места), взаимодействующие с базой данных через EF Core.

API системы построено по принципу RESTful с использованием паттерна Repository и Unit of Work. Это обеспечивает чистоту кода, тестируемость и возможность легкого расширения функционала. Основные контроллеры:

[Route(“api/[controller]”)]

    [ApiController]

    public class DevicesController : ControllerBase

    {

        private readonly IdeviceRepository \_deviceRepo;

        private readonly AddDeviceService \_addDeviceService;

        private readonly UpdateDeviceService \_updateDeviceService;

        private readonly ImageService \_imageService;

        private readonly DeviceMappingService \_mappingService;

        private readonly DeviceConfiguratorService \_configuratorService;

        public DevicesController(

            IdeviceRepository deviceRepo,

            AddDeviceService addDeviceService,

            UpdateDeviceService updateDeviceService,

            ImageService imageService,

            DeviceMappingService mappingService,

            DeviceConfiguratorService configuratorService)

        {

            \_deviceRepo = deviceRepo;

            \_addDeviceService = addDeviceService;

            \_updateDeviceService = updateDeviceService;

            \_imageService = imageService;

            \_mappingService = mappingService;

            \_configuratorService = configuratorService;

        }

        [HttpGet]

        public async Task<IactionResult> GetAll()

        {

            var devices = await \_deviceRepo.GetAllAsync();

            var result = devices.Select(device => \_mappingService.MapToDetailsDto(device)).ToList();

            return Ok(result);

        }

        [HttpGet(“{id}”)]

        public async Task<IactionResult> GetById(int id)

        {

            var device = await \_deviceRepo.GetByIdAsync(id);

            if (device == null)

                return NotFound();

            var result = \_mappingService.MapToDetailsDto(device);

            return Ok(result);

        }

        [HttpPost]

        [DisableRequestSizeLimit]

        [RequestFormLimits(MultipartBodyLengthLimit = int.MaxValue)]

        public async Task<IactionResult> Create([FromForm] DeviceDto deviceDto, IformFile image)

        {

            if (!ModelState.IsValid)

                return BadRequest(ModelState);

*// --- Универсальный парсинг характеристик ---*

            if (Request.Form.TryGetValue(“characteristics”, out var characteristicsJson))

            {

                try

                {

                    deviceDto.Characteristics = JsonSerializer.Deserialize<List<CharacteristicCreateDto>>(

                        characteristicsJson,

                        new JsonSerializerOptions { PropertyNameCaseInsensitive = true }

                    );

                }

                catch (JsonException ex)

                {

                    return BadRequest($”Некорректный формат JSON для характеристик: {ex.Message}”);

                }

            }

            else

            {

*// Fallback: если пришли пары possibleCharacteristicId и value*

                var possibleCharacteristicIds = Request.Form[“possibleCharacteristicId”];

                var values = Request.Form[“value”];

                if (possibleCharacteristicIds.Count > 0 && values.Count > 0 && possibleCharacteristicIds.Count == values.Count)

                {

                    deviceDto.Characteristics = new List<CharacteristicCreateDto>();

                    for (int I = 0; I < possibleCharacteristicIds.Count; i++)

                    {

                        if (int.TryParse(possibleCharacteristicIds[i], out int pcId))

                        {

                            deviceDto.Characteristics.Add(new CharacteristicCreateDto

                            {

                                PossibleCharacteristicId = pcId,

                                Value = values[i]

                            });

                        }

                    }

                }

            }

*// --- конец универсального парсинга ---*

            if (deviceDto.Characteristics == null)

                deviceDto.Characteristics = new List<CharacteristicCreateDto>();

            try

            {

                var device = await \_addDeviceService.AddDeviceAsync(deviceDto, image);

                var result = \_mappingService.MapToDetailsDto(device);

                return CreatedAtAction(nameof(GetById), new { id = device.Id }, result);

            }

            catch (ArgumentException ex)

            {

                return BadRequest(ex.Message);

            }

            catch (Exception ex)

            {

                return StatusCode(500, $”Внутренняя ошибка сервера: {ex.Message}”);

            }

        }

        [HttpDelete(“{id}”)]

        [Authorize(Roles = “Admin”)]

        public async Task<IactionResult> Delete(int id)

        {

            try

            {

                var device = await \_deviceRepo.GetByIdAsync(id);

                if (device == null)

                    return NotFound();

                if (!string.IsNullOrEmpty(device.ImagePath))

                    \_imageService.DeleteImage(device.ImagePath);

                await \_deviceRepo.DeleteAsync(id);

                return NoContent();

            }

            catch (Exception ex)

            {

                return StatusCode(500, $”Внутренняя ошибка сервера: {ex.Message}”);

            }

        }

        [HttpPut(“{id}”)]

        [DisableRequestSizeLimit]

        [RequestFormLimits(MultipartBodyLengthLimit = int.MaxValue)]

        public async Task<IactionResult> Update(int id, [FromForm] DeviceDto deviceDto, IformFile? Image)

        {

            if (!ModelState.IsValid)

                return BadRequest(ModelState);

*// --- Универсальный парсинг характеристик ---*

            if (Request.Form.TryGetValue(“characteristics”, out var characteristicsJson))

            {

                try

                {

                    deviceDto.Characteristics = JsonSerializer.Deserialize<List<CharacteristicCreateDto>>(

                        characteristicsJson,

                        new JsonSerializerOptions { PropertyNameCaseInsensitive = true }

                    );

                }

                catch (JsonException ex)

                {

                    return BadRequest($”Некорректный формат JSON для характеристик: {ex.Message}”);

                }

            }

            else

            {

*// Fallback: если пришли пары possibleCharacteristicId и value*

                var possibleCharacteristicIds = Request.Form[“possibleCharacteristicId”];

                var values = Request.Form[“value”];

                if (possibleCharacteristicIds.Count > 0 && values.Count > 0 && possibleCharacteristicIds.Count == values.Count)

                {

                    deviceDto.Characteristics = new List<CharacteristicCreateDto>();

                    for (int I = 0; I < possibleCharacteristicIds.Count; i++)

                    {

                        if (int.TryParse(possibleCharacteristicIds[i], out int pcId))

                        {

                            deviceDto.Characteristics.Add(new CharacteristicCreateDto

                            {

                                PossibleCharacteristicId = pcId,

                                Value = values[i]

                            });

                        }

                    }

                }

            }

*// --- конец универсального парсинга ---*

            if (deviceDto.Characteristics == null)

                deviceDto.Characteristics = new List<CharacteristicCreateDto>();

            foreach (var c in deviceDto.Characteristics)

            {

                if (string.IsNullOrEmpty(c.Value))

                    return BadRequest(«Все характеристики должны иметь значения»);

            }

            try

            {

                var updatedDevice = await \_updateDeviceService.UpdateDeviceAsync(id, deviceDto, image);

                var result = \_mappingService.MapToDetailsDto(updatedDevice);

                return Ok(result);

            }

            catch (ArgumentException ex)

            {

                return BadRequest(ex.Message);

            }

            catch (Exception ex)

            {

                return StatusCode(500, $”Внутренняя ошибка сервера: {ex.Message}”);

            }

        }

        [HttpPost(“suitable”)]

        public async Task<IactionResult> GetSuitableDevicePost([FromBody] DeviceConfigRequest request)

        {

            try

            {

                var result = await \_configuratorService.FindBestDeviceWithReasonAsync(

                    request.DeviceTypeId,

                    request.RoomTypeId,

                    request.RoomSize,

                    request.PeopleCount,

                    request.Characteristics?.Select(c => (c.PossibleCharacteristicId, c.Value)).ToList() ?? new List<(int, string)>()

                );

                if (result.Device != null)

                {

                    var deviceDto = \_mappingService.MapToDetailsDto(result.Device);

                    return Ok(new { device = deviceDto });

                }

                return NotFound(new { reason = result.Reason });

            }

            catch (ArgumentException ex)

            {

                return BadRequest(ex.Message);

            }

            catch (Exception ex)

            {

                return StatusCode(500, $”Внутренняя ошибка сервера: {ex.Message}”);

            }

        }

    }

**Репозитории**

Репозитории предоставляют абстракцию над уровнем доступа к данным. Они реализуют паттерн Repository, который позволяет изолировать бизнес-логику от деталей работы с базой данных. Каждый репозиторий отвечает за работу с определенным типом сущностей (например, IdeviceRepository для работы с устройствами).

Репозитории предоставляют методы для выполнения операций CRUD (Create, Read, Update, Delete) и других операций, связанных с данными. Они могут использовать EF Core для взаимодействия с базой данных, но детали реализации скрыты от других слоев приложения.

Для работы с EF Core и определения структуры базы данных используется класс ApplicationDbContext, который наследуется от DbContext.

Public class ApplicationDbContext : DbContext

    {

        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)

        {

        }

        public DbSet<Device> Devices { get; set; }

        public DbSet<RoomType> RoomTypes { get; set; }

        public DbSet<DeviceType> DeviceTypes { get; set; }

        public DbSet<User> Users { get; set; }

        public DbSet<Cart> Carts { get; set; }

        public DbSet<Order> Orders { get; set; }

        public DbSet<CartItem> CartItems { get; set; }

        public DbSet<OrderItem> OrderItems { get; set; }

        public DbSet<PossibleCharacteristic> PossibleCharacteristics { get; set; }

        public DbSet<Characteristic> Characteristics { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)

        {

            modelBuilder.ApplyConfiguration(new DeviceConfiguration());

            modelBuilder.ApplyConfiguration(new RoomTypeConfiguration());

            modelBuilder.ApplyConfiguration(new DeviceTypeConfiguration());

*// Конфигурация для User*

            modelBuilder.Entity<User>()

                .HasIndex(u => u.Email)

                .IsUnique();

*// Конфигурация для PossibleCharacteristic*

            modelBuilder.Entity<PossibleCharacteristic>()

                .HasOne(pc => pc.DeviceType)

                .WithMany(dt => dt.PossibleCharacteristics)

                .HasForeignKey(pc => pc.DeviceTypeId)

                .OnDelete(DeleteBehavior.Cascade);

*// Конфигурация для Characteristic*

            modelBuilder.Entity<Characteristic>()

                .HasOne(c => c.Device)

                .WithMany(d => d.Characteristics)

                .HasForeignKey(c => c.DeviceId)

                .OnDelete(DeleteBehavior.Cascade);

            modelBuilder.Entity<Characteristic>()

                .HasOne(c => c.PossibleCharacteristic)

                .WithMany(pc => pc.Characteristics)

                .HasForeignKey(c => c.PossibleCharacteristicId)

                .OnDelete(DeleteBehavior.Restrict);

            base.OnModelCreating(modelBuilder);

        }

    }

Базовые репозитории от которых унаследуются остальные:

public interface Irepository<T> where T : class

    {

        Task<Ienumerable<T>> GetAllAsync();

        Task<T?> GetByIdAsync(int id);

        Task AddAsync(T entity);

        Task UpdateAsync(T entity);

        Task DeleteAsync(int id);

    }

public class Repository<T> : Irepository<T> where T : class

    {

        protected readonly ApplicationDbContext \_context;

        private readonly DbSet<T> \_dbSet;

        public Repository(ApplicationDbContext context)

        {

            \_context = context;

            \_dbSet = context.Set<T>();

        }

        public async Task AddAsync(T entity)

        {

            try

            {

                await \_dbSet.AddAsync(entity);

                await \_context.SaveChangesAsync();

            }

            catch (DbUpdateException ex)

            {

                throw new Exception($”Ошибка при сохранении в базу данных: {ex.InnerException?.Message ?? ex.Message}”);

            }

            catch (Exception ex)

            {

                throw new Exception($”Неожиданная ошибка при добавлении: {ex.Message}”);

            }

        }

        public async Task DeleteAsync(int id)

        {

            try

            {

                var entity = await GetByIdAsync(id);

                if (entity != null)

                {

                    \_dbSet.Remove(entity);

                    await \_context.SaveChangesAsync();

                }

            }

            catch (Exception)

            {

                throw;

            }

        }

        public virtual async Task<Ienumerable<T>> GetAllAsync()

        {

            try

            {

                return await \_dbSet.ToListAsync();

            }

            catch (Exception)

            {

                throw;

            }

        }

        public virtual async Task<T?> GetByIdAsync(int id)

        {

            try

            {

                return await \_dbSet.FindAsync(id);

            }

            catch (Exception)

            {

                throw;

            }

        }

        public async Task UpdateAsync(T entity)

        {

            try

            {

                \_dbSet.Update(entity);

                await \_context.SaveChangesAsync();

            }

            catch (Exception)

            {

                throw;

            }

        }

    }

Продолжая описание разработки серверной части, следует отметить ключевую роль сервисов. Сервисы, как уже было сказано, содержат основную бизнес-логику приложения. Они инкапсулируют сложные алгоритмы и операции, обеспечивая чистоту и тестируемость контроллеров.

**Пример: AddDeviceService**

Сервис AddDeviceService отвечает за добавление нового устройства в каталог. Этот процесс включает несколько этапов:

* **Валидация входных данных:** Проверка корректности данных, полученных от клиентского приложения (например, проверка на обязательные поля, соответствие типов данных).
* **Обработка изображения:** Сохранение изображения устройства на сервере и получение пути к файлу.
* **Маппинг данных:** Преобразование данных из DTO (Data Transfer Object) в модель Device, которая соответствует структуре базы данных.
* **Сохранение устройства в базе данных:** Использование репозитория IdeviceRepository для сохранения нового устройства в базе данных.

**Пример: DeviceConfiguratorService**

Сервис DeviceConfiguratorService реализует логику подбора оптимального оборудования на основе параметров помещения и других характеристик. Алгоритм подбора оборудования может включать сложные расчеты и правила, которые инкапсулированы в этом сервисе.

**Dependency Injection**

Для обеспечения гибкости и тестируемости системы, используется Dependency Injection (DI). DI позволяет внедрять зависимости (например, репозитории, сервисы) в другие классы (например, контроллеры, сервисы) через конструктор.

Это позволяет легко заменять реализации зависимостей, а также упрощает модульное тестирование. Например, при тестировании контроллера можно внедрить мок-репозиторий, который возвращает заранее определенные данные, что позволяет изолировать тестирование контроллера от работы с реальной базой данных.

Теперь, посмотрим на код сервисов, включающих в себя реализацию паттерна Repository и Dependency Injection:

public class AddDeviceService

    {

        private readonly IdeviceRepository \_deviceRepository;

        private readonly ImageService \_imageService;

        private readonly DeviceMappingService \_mappingService;

        private readonly ApplicationDbContext \_context;

        public AddDeviceService(IdeviceRepository deviceRepository, ImageService imageService, DeviceMappingService mappingService, ApplicationDbContext context)

        {

            \_deviceRepository = deviceRepository;

            \_imageService = imageService;

            \_mappingService = mappingService;

            \_context = context;

        }

        public async Task<Device> AddDeviceAsync(DeviceDto deviceDto, IformFile imageFile)

        {

            var ventilationType = await \_deviceRepository.GetDeviceTypeByIdAsync(deviceDto.DeviceTypeId);

            if (ventilationType == null)

            {

                throw new ArgumentException(“Указанный тип вентиляции не найден.”);

            }

*// Обработка изображения*

            if (imageFile != null && imageFile.Length > 0)

            {

                deviceDto.ImagePath = await \_imageService.SaveImageAsync(imageFile, “devices”);

            }

            else

            {

                deviceDto.ImagePath = string.Empty;

            }

*// Проверяем характеристики*

            if (deviceDto.Characteristics == null)

                deviceDto.Characteristics = new List<CharacteristicCreateDto>();

*// Логируем и фильтруем характеристики*

            if (deviceDto.Characteristics != null)

            {

                Console.WriteLine(“=== DTO Characteristics ===”);

                foreach (var c in deviceDto.Characteristics)

                {

                    Console.WriteLine($”PossibleCharacteristicId: {c.PossibleCharacteristicId}, Value: {c.Value}”);

                }

                deviceDto.Characteristics = deviceDto.Characteristics

                    .Where(c => c != null && c.PossibleCharacteristicId > 0 && !string.IsNullOrEmpty(c.Value))

                    .ToList();

            }

            var device = \_mappingService.MapToModel(deviceDto, ventilationType);

*// Явно подгружаем существующую PossibleCharacteristic для каждой характеристики*

            if (device.Characteristics != null)

            {

                Console.WriteLine(«=== Характеристики для сохранения ===»);

                foreach (var c in device.Characteristics)

                {

                    Console.WriteLine($”PossibleCharacteristicId: {c.PossibleCharacteristicId}, Value: {c.Value}”);

                    c.PossibleCharacteristic = await \_context.PossibleCharacteristics.FindAsync(c.PossibleCharacteristicId);

                    if (c.PossibleCharacteristic == null)

                        throw new Exception($”PossibleCharacteristic с id={c.PossibleCharacteristicId} не найдена в базе!”);

                }

            }

            try

            {

                await \_deviceRepository.AddAsync(device);

*// Логируем характеристики после сохранения*

                var savedDevice = await \_context.Devices

                    .Include(d => d.Characteristics)

                    .FirstOrDefaultAsync(d => d.Id == device.Id);

                Console.WriteLine(«=== Сохранённые характеристики устройства ===»);

                if (savedDevice?.Characteristics != null)

                {

                    foreach (var c in savedDevice.Characteristics)

                    {

                        Console.WriteLine($”PossibleCharacteristicId: {c.PossibleCharacteristicId}, Value: {c.Value}”);

                    }

                }

                else

                {

                    Console.WriteLine(«Характеристики отсутствуют!»);

                }

                return device;

            }

            catch (Exception ex)

            {

                if (!string.IsNullOrEmpty(device.ImagePath))

                {

                    \_imageService.DeleteImage(device.ImagePath);

                }

                throw new Exception($»Ошибка при сохранении устройства: {ex.Message}», ex);

            }

        }

    }

public class UpdateDeviceService

    {

        private readonly IdeviceRepository \_deviceRepository;

        private readonly ImageService \_imageService;

        private readonly DeviceMappingService \_mappingService;

        public UpdateDeviceService(IdeviceRepository deviceRepository, ImageService imageService, DeviceMappingService mappingService)

        {

            \_deviceRepository = deviceRepository;

            \_imageService = imageService;

            \_mappingService = mappingService;

        }

        public async Task<Device> UpdateDeviceAsync(int id, DeviceDto deviceDto, IformFile? Image)

        {

            var existingDevice = await \_deviceRepository.GetByIdAsync(id);

            if (existingDevice == null)

            {

                throw new ArgumentException($”Устройство с ID {id} не найдено”);

            }

            var ventilationType = await \_deviceRepository.GetDeviceTypeByIdAsync(deviceDto.DeviceTypeId);

            if (ventilationType == null)

            {

                throw new ArgumentException(“Указанный тип вентиляции не найден”);

            }

*// Обработка изображения*

            if (image != null && image.Length > 0)

            {

                if (!string.IsNullOrEmpty(existingDevice.ImagePath))

                {

                    \_imageService.DeleteImage(existingDevice.ImagePath);

                }

                deviceDto.ImagePath = await \_imageService.SaveImageAsync(image, “devices”);

            }

            else

            {

                deviceDto.ImagePath = existingDevice.ImagePath;

            }

*// Обновляем основные данные устройства*

            existingDevice.Name = deviceDto.Name;

            existingDevice.Description = deviceDto.Description;

            existingDevice.ImagePath = deviceDto.ImagePath;

            existingDevice.PowerConsumption = deviceDto.PowerConsumption;

            existingDevice.NoiseLevel = deviceDto.NoiseLevel;

            existingDevice.MaxAirflow = deviceDto.MaxAirflow;

            existingDevice.Price = deviceDto.Price;

            existingDevice.DeviceTypeId = deviceDto.DeviceTypeId;

            existingDevice.DeviceType = ventilationType;

*// Обновляем характеристики*

            if (deviceDto.Characteristics == null)

                deviceDto.Characteristics = new List<CharacteristicCreateDto>();

*// Проверяем все характеристики на наличие значений*

            foreach (var c in deviceDto.Characteristics)

            {

                if (string.IsNullOrEmpty(c.Value))

                    throw new ArgumentException(«Все характеристики должны иметь значения»);

            }

*// Очищаем существующие характеристики*

            existingDevice.Characteristics?.Clear();

*// Добавляем новые характеристики*

            foreach (var charDto in deviceDto.Characteristics)

            {

                existingDevice.Characteristics.Add(new Characteristic

                {

                    PossibleCharacteristicId = charDto.PossibleCharacteristicId,

                    Value = charDto.Value,

                    Device = existingDevice

                });

            }

            try

            {

                await \_deviceRepository.UpdateAsync(existingDevice);

                return existingDevice;

            }

            catch (Exception ex)

            {

*// Если произошла ошибка и мы сохранили новое изображение, удаляем его*

                if (image != null && !string.IsNullOrEmpty(deviceDto.ImagePath))

                {

                    \_imageService.DeleteImage(deviceDto.ImagePath);

                }

                throw new Exception($”Ошибка при обновлении устройства: {ex.Message}”, ex);

            }

        }

    }

public class ImageService

    {

        private readonly IwebHostEnvironment \_environment;

        private readonly string[] \_allowedExtensions = { “.jpg”, “.jpeg”, “.png”, “.gif” };

        public ImageService(IwebHostEnvironment environment)

        {

            \_environment = environment;

        }

        public async Task<string> SaveImageAsync(IformFile image, string subfolder = “devices”)

        {

            if (image == null || image.Length == 0)

            {

                throw new ArgumentException(«Изображение обязательно для загрузки»);

            }

            var fileExtension = Path.GetExtension(image.FileName).ToLowerInvariant();

            if (!\_allowedExtensions.Contains(fileExtension))

            {

                throw new ArgumentException($”Недопустимый формат файла. Разрешены только: {string.Join(“, “, \_allowedExtensions)}”);

            }

            var uploadsFolder = Path.Combine(\_environment.WebRootPath ?? “wwwroot”, “uploads”, subfolder);

            if (!Directory.Exists(uploadsFolder))

            {

                Directory.CreateDirectory(uploadsFolder);

            }

            var uniqueFileName = $”{Guid.NewGuid()}\_{image.FileName}”;

            var filePath = Path.Combine(uploadsFolder, uniqueFileName);

            using (var fileStream = new FileStream(filePath, FileMode.Create))

            {

                await image.CopyToAsync(fileStream);

            }

            return $”/uploads/{subfolder}/{uniqueFileName}”;

        }

        public void DeleteImage(string imagePath)

        {

            if (string.IsNullOrEmpty(imagePath))

            {

                return;

            }

*// Удаляем начальный слэш, если он есть*

            var relativePath = imagePath.TrimStart(‘/’);

            var fullPath = Path.Combine(\_environment.WebRootPath ?? “wwwroot”, relativePath);

            if (File.Exists(fullPath))

            {

                File.Delete(fullPath);

            }

        }

    }

Продолжая описание разработки серверной части, следует выделить особенности интеграции с **GigaChat**. Для реализации функциональности ИИ-консультанта, была выполнена интеграция с сервисом GigaChat от Сбербанка. Этот сервис позволяет обрабатывать естественный язык и предоставлять рекомендации на основе запросов пользователей. GigaChat был выбран в качестве платформы для реализации ИИ-консультанта благодаря своей развитой модели обработки естественного языка и возможности интеграции с различными API.

**Интеграция с GigaChat**

Интеграция с GigaChat включает в себя следующие шаги:

**Получение Access Token:** для взаимодействия с GigaChat API требуется аутентификация. Получение токена доступа осуществляется через защищенный канал с использованием ключа авторизации. Этот процесс включает в себя отправку запроса на сервер аутентификации GigaChat и получение токена доступа в ответ. Токен доступа используется для аутентификации последующих запросов.

**Формирование запроса:** запросы к GigaChat API формируются в формате, требуемом сервисом. В запросе передается:

**Системное сообщение:** содержит инструкции для ИИ-консультанта, описывающие его роль и задачи, а также информацию о доступных устройствах. Системное сообщение играет важную роль в определении поведения ИИ-консультанта.

**История чата:** предыдущие сообщения пользователя и ответы ИИ, обеспечивающие контекст для текущего запроса. История чата позволяет ИИ-консультанту учитывать контекст разговора и предоставлять более релевантные рекомендации.

**Отправка запроса и получение ответа:** запрос отправляется на GigaChat API. Полученный ответ содержит рекомендацию по выбору оборудования.

**Обработка ответа:** Ответ обрабатывается, и из него извлекается рекомендация, которая затем возвращается пользователю. Обработка ответа может включать в себя различные этапы, такие как парсинг JSON, валидация данных и форматирование ответа.

В рамках данного подхода, были созданы классы GigaChatService и AIConsultantController.

**GigaChatService**: этот класс инкапсулирует логику взаимодействия с GigaChat API. Он отвечает за получение токена доступа, формирование запросов, отправку запросов и обработку ответов. Класс GigaChatService реализует все необходимые методы для взаимодействия с GigaChat API, обеспечивая централизованное управление процессом интеграции.

[ApiController]

    [Route(“api/[controller]”)]

    public class AIConsultantController : ControllerBase

    {

        private readonly GigaChatService \_gigaChatService;

        public AIConsultantController(GigaChatService gigaChatService)

        {

            \_gigaChatService = gigaChatService;

        }

        [HttpPost(“recommend”)]

        public async Task<IactionResult> GetRecommendation([FromBody] ChatRequest request)

        {

            if (string.IsNullOrEmpty(request.Query))

            {

                return BadRequest(«Запрос не может быть пустым»);

            }

            try

            {

*// Исправляем проблему с оператором ??*

                var recommendation = await \_gigaChatService.GetDeviceRecommendation(

                    request.Query,

                    request.History ?? new List<ChatMessage>()

                );

                return Ok(new { recommendation });

            }

            catch (Exception ex)

            {

                return StatusCode(500, new { error = «Произошла ошибка при получении рекомендации» });

            }

        }

    }

 public class GigaChatService

    {

        private readonly HttpClient \_httpClient;

        private readonly string \_authKey;

        private readonly string \_tokenUrl;

        private readonly string \_apiUrl;

        private string \_accessToken;

        private DateTime \_tokenExpiration;

        private readonly IdeviceRepository \_deviceRepository;

        private readonly string \_sessionId;

        public GigaChatService(Iconfiguration configuration, IdeviceRepository deviceRepository)

        {

            \_httpClient = new HttpClient();

            \_authKey = configuration[“GigaChat:AuthKey”];

            \_tokenUrl = “https://ngw.devices.sberbank.ru:9443/api/v2/oauth”;

            \_apiUrl = configuration[“GigaChat:ApiUrl”];

            \_deviceRepository = deviceRepository;

            \_sessionId = Guid.NewGuid().ToString();

        }

        private async Task<string> GetAccessTokenAsync()

        {

            if (!string.IsNullOrEmpty(\_accessToken) && DateTime.UtcNow < \_tokenExpiration)

            {

                return \_accessToken;

            }

            var requestBody = new FormUrlEncodedContent(new[]

            {

                new KeyValuePair<string, string>(“scope”, “GIGACHAT\_API\_PERS”)

            });

            var request = new HttpRequestMessage(HttpMethod.Post, \_tokenUrl);

            request.Headers.Add(“Authorization”, $”Basic {\_authKey}”);

            request.Headers.Add(“RqUID”, “0b862e8b-90bd-4c63-a97b-49bc3f8ae5f9”);

            request.Headers.Add(“Accept”, “application/json”);

            request.Content = requestBody;

            var response = await \_httpClient.SendAsync(request);

            var responseContent = await response.Content.ReadAsStringAsync();

            response.EnsureSuccessStatusCode();

            var tokenResponse = JsonSerializer.Deserialize<TokenResponse>(responseContent);

            \_accessToken = tokenResponse.AccessToken;

            \_tokenExpiration = DateTime.UtcNow.AddSeconds(tokenResponse.ExpiresIn – 60);

            return \_accessToken;

        }

        public async Task<string> GetDeviceRecommendation(string userQuery, List<Models.ChatMessage> chatHistory = null)

        {

            try

            {

*// Получаем все устройства из базы данных*

                var devices = await \_deviceRepository.GetAllAsync();

                var devicesList = devices.ToList();

                if (!devicesList.Any())

                {

                    return «Извините, в базе данных нет доступных устройств.»;

                }

*// Формируем информацию об устройствах для ИИ*

                var devicesInfo = string.Join(“\n”, devicesList.Select(d =>

                {

                    var characteristicsInfo = d.Characteristics != null && d.Characteristics.Any()

                        ? “, Характеристики: “ + string.Join(“, “, d.Characteristics.Select(c => $”{c.PossibleCharacteristic.Name}: {c.Value} {c.PossibleCharacteristic.Unit}”))

                        : “”;

                    return $”Устройство: {d.Name}, “ +

                           $”Описание: {d.Description}, “ +

                           $»Макс. Расход воздуха: {d.MaxAirflow} м³/ч, « +

                           $»Потребляемая мощность: {d.PowerConsumption} Вт, « +

                           $»Уровень шума: {d.NoiseLevel} Дб, « +

                           $”Цена: {d.Price} руб.” +

                           characteristicsInfo;

                }));

                var accessToken = await GetAccessTokenAsync();

*// Создаем список сообщений для отправки в GigaChat*

                var messages = new List<object>();

*// Добавляем системное сообщение (инструкции для ИИ)*

                messages.Add(new

                {

                    role = “system”,

                    content = «Ты – консультант по подбору устройств вентиляции компании Tion. « +

                            «Вот список доступных устройств:\n» + devicesInfo + «\n\n» +

                            «Твоя задача – помочь пользователю выбрать подходящее устройство на основе его запроса. « +

                            «Используй информацию из списка устройств для рекомендации. « +

                            «Если пользователь ссылается на предыдущие рекомендации, учитывай контекст разговора.»

                });

*// Добавляем историю чата (если есть)*

                if (chatHistory != null && chatHistory.Any())

                {

                    foreach (var message in chatHistory)

                    {

                        messages.Add(new

                        {

                            role = message.Role,

                            content = message.Content

                        });

                    }

                }

*// Формируем запрос к GigaChat API*

                var requestBody = new

                {

                    model = “GigaChat”,

                    messages = messages

                };

                var jsonContent = JsonSerializer.Serialize(requestBody);

                var content = new StringContent(jsonContent, Encoding.UTF8, “application/json”);

                var request = new HttpRequestMessage(HttpMethod.Post, \_apiUrl);

                request.Headers.Add(“Authorization”, $”Bearer {accessToken}”);

                request.Headers.Add(“X-Session-ID”, \_sessionId);

                request.Content = content;

*// Отправляем запрос и получаем ответ*

                var response = await \_httpClient.SendAsync(request);

                var responseContent = await response.Content.ReadAsStringAsync();

                if (!response.IsSuccessStatusCode)

                {

                    throw new HttpRequestException($”GigaChat API error: {response.StatusCode} – {responseContent}”);

                }

                var responseObject = JsonSerializer.Deserialize<GigaChatResponse>(responseContent);

                if (responseObject?.Choices == null || responseObject.Choices.Length == 0)

                {

                    return «Извините, не удалось получить рекомендацию.»;

                }

                var recommendation = responseObject.Choices[0].Message?.Content;

                if (string.IsNullOrEmpty(recommendation))

                {

                    return «Извините, не удалось получить рекомендацию.»;

                }

                return recommendation;

            }

            catch (Exception ex)

            {

                throw;

            }

        }

    }

## 3.5 Разработка клиентской части приложения и описание пользовательского интерфейса

Проектирование интерфейса интерактивного конфигуратора осуществлялось с учетом требований к удобству использования, функциональности и адаптивности. Основной задачей являлось создание интуитивно понятного инструмента для пользователей с разным уровнем технической подготовки. Процесс включал следующие этапы:

1. **Анализ пользовательских сценариев**: на основе требований, описанных в разделе 3.1, были определены ключевые сценарии взаимодействия:
   * Ввод параметров помещения (площадь, количество людей).
   * Подбор оборудования с фильтрацией по характеристикам.
   * Формирование заказа и экспорт спецификации.
2. **Разработка макетов**: для визуализации структуры интерфейса использовались эскизы на бумаге и цифровые инструменты. Основной упор делался на:
   * Минималистичный дизайн с акцентом на функциональность.
   * Логичную группировку элементов управления.
   * Адаптацию под разные устройства (десктоп, мобильные).
3. **Выбор элементов управления**:
   * Для ввода числовых параметров (площадь, бюджет) использованы текстовые поля с валидацией.
   * Фильтрация оборудования реализована через выпадающие списки и ползунки.
   * Интерактивные карточки продуктов содержат изображения, краткое описание и кнопки действий.
4. **Прототипирование и тестирование**: на этапе проектирования проводилась проверка логики переходов между экранами и удобства навигации. Например, кнопка «Назад» добавлена на всех этапах подбора, чтобы пользователь мог корректировать параметры.

**Основные экраны приложения**

**Конфигуратор (рисунок 3):** на данном экране пользователь вводит параметры помещения (площадь, количество людей). Элементы размещены вертикально для последовательного заполнения. Валидация данных реализована через подсказки при некорректном вводе (например, отрицательные значения).

**Каталог (рисунок 4):** Страница с карточками всех товаров. Карточки продуктов содержат изображения, ключевые характеристики (мощность, уровень шума) и кнопку «В корзину».

**Корзина (рисунок 6):** интерфейс корзины включает таблицу с выбранными товарами, итоговой суммой и кнопкой «Оформить заказ». Для удобства добавлена возможность удаления и изменения количества позиций.

**Формы авторизации и регистрации (рисунок 7 и 8):** минималистичный дизайн форм снижает когнитивную нагрузку. Обязательные поля помечены, ошибки выводятся под соответствующими input-полями.

**Заказы (рисунок 9):** интерфейс раздела «Заказы» предназначен для просмотра истории оформленных заказов и управления их статусами. Основные элементы: таблица заказов, детализация заказа, изменение статуса

**ИИ-консультант (рисунок 10):** чат для помощи пользователю по определению необходимого устройства или уточнения нужной информации.

**Обоснование дизайн-решений**

* Цветовая схема: Использованы корпоративные цвета компании ТИОН (синий и белый) для укрепления брендинга.
* Навигация: Минималистичное меню с быстрым доступом к основным разделам.
* Обратная связь: Интерактивные элементы (например, анимация загрузки) визуализируют процесс подбора оборудования.

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 3. Конфигуратор

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 4. Результат подбора

Изображение выглядит как текст, снимок экрана, программное обеспечение, Операционная система

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 4 Каталог

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 5. Форма создания устройства

Изображение выглядит как текст, снимок экрана, число, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 6. Корзина

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 6. Заказы

Изображение выглядит как текст, снимок экрана, дисплей, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 7. Форма входа

Изображение выглядит как текст, снимок экрана, число, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 8. Форма регистрации

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 9. Заказы

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 10. ИИ-консультант

Изображение выглядит как текст, Шрифт, программное обеспечение, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 10. Справочник типов помещения

Изображение выглядит как текст, Шрифт, число, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 10. Справочник типов устройств

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Контент, сгенерированный ИИ, может содержать ошибки.

Рис. 10. Справочник возможных характеристик

Спроектированный интерфейс соответствует требованиям к usability и функциональности. Логичная структура экранов, понятная навигация и визуальные подсказки обеспечивают комфортное взаимодействие как для клиентов, так и для менеджеров компании.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения дипломной работы была разработана и внедрена информационная система для автоматизированного подбора и конфигурирования вентиляционного оборудования на примере компании ТИОН. Проведённый анализ предметной области и существующих решений позволил выявить основные проблемы и требования, предъявляемые к современным системам автоматизации в данной сфере.

В результате проектирования была создана гибкая архитектура, учитывающая специфику бизнес-процессов компании и обеспечивающая возможность масштабирования и интеграции с внешними сервисами. Реализация конфигуратора на основе современных технологий позволила автоматизировать ключевые этапы подбора оборудования, повысить точность расчётов, снизить нагрузку на специалистов и улучшить качество обслуживания клиентов.

Проведённое тестирование системы показало её высокую эффективность, удобство использования и положительное влияние на бизнес-показатели компании. Внедрение конфигуратора позволило сократить время на обработку заказов, снизить количество ошибок и повысить удовлетворённость клиентов.

Практическая значимость работы заключается в возможности применения разработанной системы не только в компании ТИОН, но и на других предприятиях, занимающихся производством и продажей климатического оборудования. Разработанный подход может быть адаптирован для решения аналогичных задач в смежных отраслях. В дальнейшем перспективным направлением развития системы является расширение функциональности конфигуратора, интеграция с системами управления складом и логистикой, внедрение дополнительных интеллектуальных сервисов для поддержки пользователей, а также развитие мобильных и облачных решений.

Таким образом, поставленные в работе цели и задачи были успешно реализованы, а полученные результаты подтверждают актуальность и эффективность выбранного подхода к автоматизации подбора вентиляционного оборудования.

СПИСОК ИСТОЧНИКОВ

1. Агальцов, В. П. Базы данных: в 2 книгах. Книга 2. Распределенные и удаленные базы данных: учебник / В.П. Агальцов. — Москва: ФОРУМ : ИНФРА-М, 2021. — 271 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-8199-0713-9. - Текст: электронный. - URL: https://znanium.com/catalog/product/1514118 (дата обращения: 09.05.2025). – Режим доступа: по подписке.

2. TechEmpower [Электронный ресурс]. – Режим доступа: <https://www.techempower.com/benchmarks/> (дата обращения: 11.05.2025).

3. Официальный сайт ТИОН [Электронный ресурс]. – Режим доступа: <https://tion.ru/about/> (дата обращения: 11.05.2025).

4. Stack Overflow Trends [Электронный ресурс]. – Режим доступа: <https://insights.stackoverflow.com/trends> (дата обращения: 11.05.2025).

5. Шустова, Л. И. Базы данных: учебник / Л.И. Шустова, О.В. Тараканов. — Москва: ИНФРА-М, 2021. — 304 с. + Доп. материалы [Электронный ресурс]. — (Высшее образование: Бакалавриат). — DOI 10.12737/11549. - ISBN 978-5-16-010485-0. - Текст: электронный. - URL: https://znanium.com/catalog/product/ 1362122 (дата обращения: 12.05.2025). – Режим доступа: по подписке.

6. Белов, В. В. Алгоритмы и структуры данных : учебник / В. В. Белов, В. И. Чистякова. - Москва: КУРС: ИНФРА-М, 2020. - 240 с. - (Бакалавриат). - ISBN 978-5-906818-25-6. - Текст: электронный. - URL: https://znanium.com/catalog/ product/1057212 (дата обращения: 12.05.2025). – Режим доступа: по подписке.

7. Хорев, П. Б. Объектно-ориентированное программирование с примерами на С# : учебное пособие / П.Б. Хорев. — Москва : ФОРУМ : ИНФРА-М, 2020. — 200 с. — (Высшее образование: Бакалавриат). - ISBN 978-5-00091-680-3. - Текст : электронный. - URL: https://znanium.com/catalog/product/1069921 (дата обращения: 13.05.2025). – Режим доступа: по подписке.

8. Кокоса, К. Управление памятью в .NET для профессионалов : практическое руководство / К. Кокоса. - Москва : ДМК Пресс, 2020. - 800 с. - ISBN 978-5-97060-800-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1210679 (дата обращения: 13.05.2025). – Режим доступа: по подписке.

9. Новиков, Б. А. Основы технологий баз данных : учебное пособие / Б. А. Новиков, Е. А. Горшкова, Н. Г. Графеева ; под ред. Е. В. Рогова. — 2-е изд. — Москва : ДМК Пресс, 2020. - 582 с. - ISBN 978-5-97060-841-8. - Текст : электронный. - URL: https://znanium.com/catalog/product/1210665 (дата обращения: 14.05.2025). – Режим доступа: по подписке.

10. Сьоре, Э. Проектирование и реализация систем управления базами данных : учебное пособие / Эдвард Сьоре ; пер. с анг. А. Н. Киселева ; научн. ред. Е. В. Рогов. - Москва : ДМК Пресс, 2021. - 466 с. - ISBN 978-5-97060-488-5. - Текст : электронный. - URL: https://znanium.com/catalog/product/1225360 (дата обращения: 14.05.2025). – Режим доступа: по подписке.

11. Федотов, И. Е. Параллельное программирование. Модели и приемы : практическое пособие / И. Е. Федотов. - Москва : СОЛОН-Пресс, 2020. - 390 с. - (Серия «Библиотека профессионала»). - ISBN 978-5-91359-222-4. - Текст : электронный. - URL: https://znanium.com/catalog/product/1858781 (дата обращения: 15.05.2025). – Режим доступа: по подписке.

12. Коваленко, В. В. Проектирование информационных систем : учебное пособие / В.В. Коваленко. — 2-е изд., перераб. и доп. — Москва : ИНФРА-М, 2023. — 357 с. — (Высшее образование: Бакалавриат). — DOI 10.12737/987869. - ISBN 978-5-00091-783-1. - Текст : электронный. - URL: <https://znanium.com/catalog/product/1894610> (дата обращения: 15.05.2025). – Режим доступа: по подписке.

# ПРИЛОЖЕНИЯ

**Приложение 1**