МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет
имени К.И. Сатпаева

Институт автоматики и информационных технологий

Кафедра «Программная инженерия»

Калиев Б.Е.
Миргалимов Э.Ф.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к дипломному проекту

Образовательная программа 6B06102 – Computer Science

Алматы 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет имени К.И.Сатпаева

Институт автоматики и информационных технологий

Кафедра «Программная инженерия»

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
канд. тех. наук, ассоц.профессор
_____ Ф.Н. Абдолдина
“ 30 ” 05 2025 г.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к дипломному проекту

На тему: «Разработка системы тикетов»

Образовательная программа: 6B06102 Computer Science

Выполнил Калиев Бекжан Ержанович
Миргалимов Эльмир Фуатович

Рецензент
PhD, профессор-исследователь
_____ Сансызбай К.М.
"30" 05 2025 г.

Научный руководитель
PhD, ассоциированный профессор
_____ Мукажанов Н.К.
"28" 05 2025 г.

Алматы 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ КАЗАХСТАН

Казахский национальный исследовательский технический университет имени К.И.Сатпаева

Институт автоматики и информационных технологий

Кафедра «Программная инженерия»

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
канд. тех. наук, ассоц.профессор
_____ Ф.Н. Абдолдина
«30» 05 2025 г.

ЗАДАНИЕ
на выполнение дипломного проекта

Обучающемуся: *Калиев Б.Е., Миргалимов Э.Ф.*
Тема: Разработка системы тикетов
Утверждена приказом проректора по академической работе:

№ *26- П/Ө*	*от «29» января 2025 г.*
Срок сдачи законченного проекта	*«_____»_____2025 г.*

Исходные данные к дипломному проекту:
А) Анализ предметной области, анализ аналогичных проектов;
Б) Разработка технического задания;
В) Разработка архитектуры ПО;
Г) Проектирование и разработка понятного и интерактивного дизайна;
Д) Разработка и тестирование ПО;
Перечень подлежащих разработке в дипломном проекте вопросов: (с точным указанием обязательных чертежей): *представлены_____слайда презентации.*
Рекомендуемая основная литература: *из_____наименований.*

ГРАФИК

подготовки дипломного проекта

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю и консультантам	Ответственный	Примечание
1. Анализ предметной области, разработка технического задания	01.02 – 14.02.2025	Калиев Б.Е.	Выполнено
2. Выбор технологий для разработки	15.02 – 25.02.2025	Калиев Б.Е.	Выполнено
3. Разработка логики взаимодействия	26.02 – 20.03.2025	Миргалимов Э.Ф.	Выполнено
4. Разработка функционала системы	10.03 – 05.04.2025	Миргалимов Э.Ф.	Выполнено
5. Тестирование и оптимизация	16.04 – 30.04.2025	Калиев Б.Е.	Выполнено
6. Написание пояснительной записки к дипломному проекту	16.04 – 30.04.2025	Калиев Б.Е., Миргалимов Э.Ф.	Выполнено

Подписи

консультантов и нормоконтролера на законченную дипломный проект
с указанием относящихся к ним разделов проекта

Наименования разделов	Консультанты, И.О.Ф. (уч. степень, звание)	Дата подписания	Подпись	Оценка
Программное обеспечение	Кабылжан М. преподаватель, магистр.	30.05.25 г.		
Нормоконтролер	Ердалиева З.У. преподаватель, магистр.	30.05.25		

Научный руководитель _____ Мукажанов Н.К.

Задание принял к исполнению обучающиеся _____ Калиев Б.Е.,
_____ Миргалимов Э.Ф..

Дата «27» 05. 2025 г.

АҢДАТПА

Бұл дипломдық жоба корпоративтік ортада ішкі өтінімдерді (тикеттерді) басқаруға арналған ақпараттық жүйені әзірлеуге арналған. Жүйе ASP.NET Core платформасында Razor Pages, Entity Framework және MS SQL Server технологияларын қолдану арқылы жүзеге асырылды. Жобаның басты мақсаты – өтінімдерді өңдеу үдерісін автоматтандыру, қызметкерлер мен бөлімдер арасындағы өзара әрекеттестіктің ашықтығы мен тиімділігін арттыру. Тикеттерді құру, бақылау, сүзу және өңдеу, сонымен қатар мәліметтерді экспорттау мен пайдаланушыларды авторизациялау функциялары іске асырылған. Жүйе сәтті сынақтан өтіп, енгізуге дайын.

АННОТАЦИЯ

Данный дипломный проект посвящён разработке информационной системы для управления внутренними заявками (тикетами) в корпоративной среде. Система реализована на платформе ASP.NET Core с использованием Razor Pages, Entity Framework и MS SQL Server. Основная цель проекта — автоматизация процессов обработки заявок, повышение прозрачности и эффективности взаимодействия между сотрудниками и подразделениями. Реализована функциональность для создания, отслеживания, фильтрации и обработки тикетов, а также модуль экспорта данных и авторизации пользователей. Система успешно протестирована и подготовлена к внедрению.

ABSTRACT

This diploma project is focused on the development of an internal ticket management information system for a corporate environment. The system is implemented using ASP.NET Core, Razor Pages, Entity Framework, and MS SQL Server. The main goal is to automate the request handling process and improve transparency and efficiency of communication between employees and departments. The application includes features for creating, tracking, filtering, and processing tickets, as well as data export and user authentication. The system has been successfully tested and is ready for deployment.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В текущей ситуации, где информационные технологии очень быстро проникают во все сферы деятельности, компании всё чаще сталкиваются с необходимостью комплексного подхода к организации внутренних процессов. Одна из важнейших задач, стоящая перед любой организацией, является организация быстрой, безопасной и ориентированной на простого пользователя системой между сотрудниками, особенно в вопросах постановки, распределения и контроля за скорейшим исполнением задач. Внутренние заявки и обращения становятся важным аспектом повседневной операционной деятельности, и от того, как хорошо, эффективно и грамотно выстроен процесс их обработки, напрямую зависит скорость принятия решений, общая слаженность работы команды и эффективность бизнеса в целом.

Отсутствие централизованной, логически выстроенной системы приводит к множеству проблем: информация теряется или дублируется, задачи выполняются с задержками, нарушаются сроки, сотрудники испытывают сложности в отслеживании текущего статуса своих заявок, а руководители — при контроле над исполнением. Всё это влечёт за собой рост операционных издержек, снижение уровня доверия внутри команды и, в итоге, негативно отражается на репутации и устойчивости компании.

Именно поэтому разработка информационной системы управления внутренними заявками приобретает особую значимость в условиях современных требований к цифровой трансформации бизнеса. Целью данного дипломного проекта стало создание удобной, интуитивно понятной и, в то же время, надёжной, гибкой и масштабируемой системы управления тикетами, способной эффективно решать задачи по регистрации, отслеживанию, фильтрации и контролю выполнения заявок. При разработке особое внимание уделялось соответствию системы реальным потребностям бизнеса, а также её способности органично встраиваться в существующую корпоративную инфраструктуру без необходимости радикальных изменений текущих процессов.

Таким образом, данная работа направлена на решение актуальной и практически значимой задачи, лежащей на стыке информационных технологий и менеджмента, и отражает современные тенденции в области цифровизации и оптимизации внутренней организационной среды. Проект демонстрирует, насколько важно сегодня не просто разрабатывать технические решения, но делать это с пониманием бизнес-контекста, потребностей пользователей и требований к гибкости и устойчивости систем в условиях постоянных изменений.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ И ПОСТАНОВКА ЗАДАЧИ

1.1 Актуальность и цели проекта

В современном обществе цифровые технологии занимают всё более важное место в различных сферах деятельности. Автоматизация и информатизация процессов становятся ключевыми факторами повышения эффективности работы организаций любого уровня и направления. В условиях увеличения объёма информации, усложнения организационных структур и роста числа участников взаимодействия особенно остро стоит вопрос об упорядочивании и оптимизации внутренних коммуникаций.

Одним из важнейших направлений развития цифровых систем является создание эффективных инструментов для обработки внутренних обращений, запросов, жалоб и предложений от различных категорий пользователей. Такие системы позволяют структурировать потоки информации, обеспечивают прозрачность исполнения задач и создают условия для своевременного контроля и анализа процессов.

На сегодняшний день во многих организациях для этих целей часто используются разрозненные и неформализованные инструменты, такие как электронная почта, мессенджеры, бумажные носители, а также различные формы и опросники, созданные на сторонних платформах. Несмотря на относительную простоту и доступность, подобные решения имеют существенные ограничения. В частности, они не обеспечивают должного уровня систематизации данных, затрудняют контроль статусов заявок, не всегда позволяют определить ответственных лиц и часто не позволяют проводить полноценный анализ по характеру и частоте обращений.

Таким образом, актуальной становится задача разработки специализированной системы управления обращениями — так называемой системы тикетов. Такая система должна учитывать специфику внутренней организационной структуры, распределённость участников, различные роли пользователей, а также требования к безопасности и сохранности данных. Важным аспектом является создание удобного и интуитивно понятного интерфейса, который позволит эффективно формулировать запросы, отслеживать их исполнение и обеспечивать прозрачность для всех заинтересованных сторон.

Целью настоящего проекта является создание веб-приложения, ориентированного на управление заявками и обращениями внутри организации, которое позволит повысить уровень автоматизации, прозрачности и контроля процессов взаимодействия между пользователями и ответственными исполнителями.

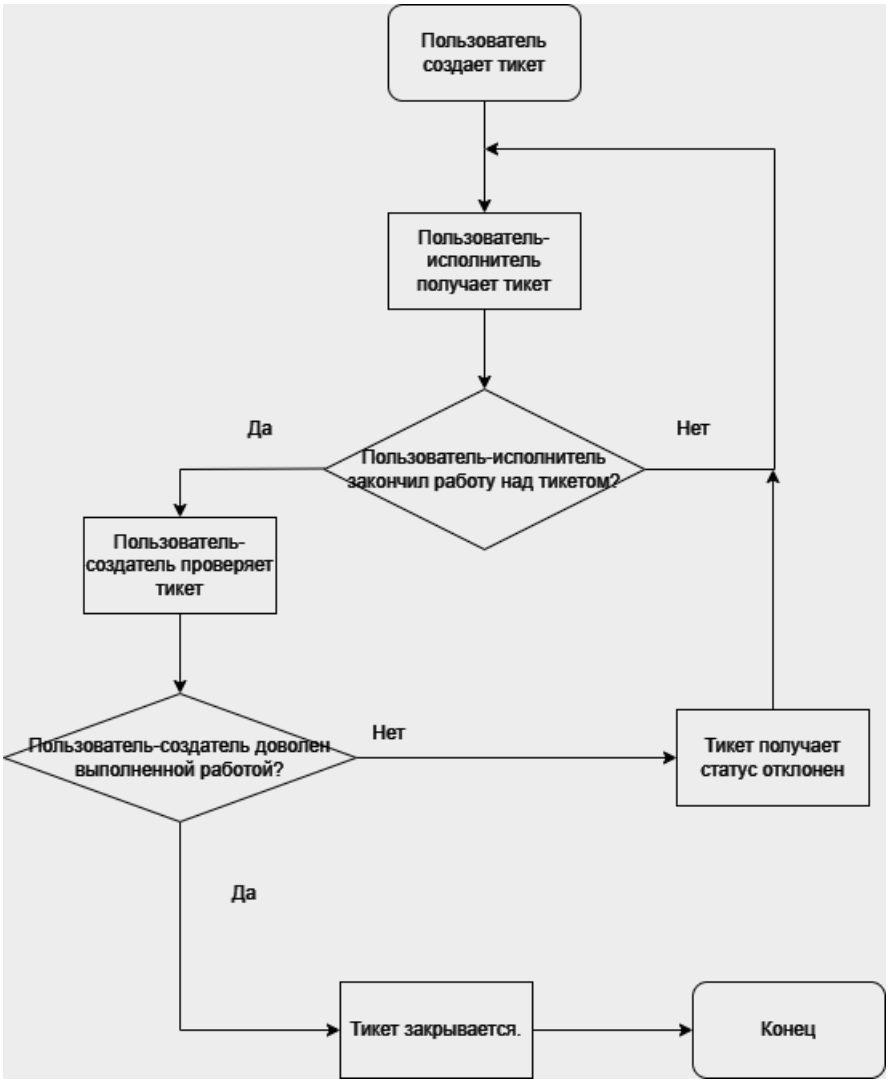


Рисунок 1.1 — Блок-схема жизненного цикла проекта

Для реализации поставленной цели необходимо решить следующие задачи:

– провести анализ существующих решений в области систем управления обращениями и выявить их преимущества и недостатки;

– определить основные требования и потребности конечных пользователей, а также специфику бизнес-процессов, связанных с обработкой заявок;

– спроектировать логическую и физическую архитектуру создаваемой системы с учётом функциональных и нефункциональных требований;

– разработать полноценное веб-приложение с многоуровневым интерфейсом, обеспечивающим доступ и функционал для различных категорий пользователей;

– провести тестирование разработанного решения и оценить потенциальное влияние его внедрения на эффективность работы организации.

Таким образом, данный проект направлен на решение актуальной задачи цифровизации внутренних коммуникаций и оптимизации процессов обработки заявок и обращений, что способствует улучшению качества управления и повышению общей эффективности деятельности.

1.2 Общие требования к системе

Для успешной реализации и последующего внедрения системы внутренней обработки заявок необходимо чётко определить и учесть комплекс функциональных и нефункциональных требований. Эти требования задают базовые ориентиры для разработки, обеспечивают соответствие системы ожиданиям пользователей и способствуют достижению высоких показателей надёжности, производительности и удобства эксплуатации.

Функциональные требования

Функциональные требования отражают ключевые возможности и поведение системы, определяя набор операций, которые должна обеспечивать система в повседневной работе:

– Создание заявок пользователями: Система должна позволять авторизованным пользователям формировать новые обращения, чётко структурируя информацию, включающую описание проблемы, приоритет, категорию и необходимые прикреплённые материалы (файлы, скриншоты и пр.). Важной особенностью является обеспечение удобного и понятного интерфейса для ввода данных, что снижает вероятность ошибок и повышает качество заявок;

– Редактирование и удаление заявок: Пользователи должны иметь возможность корректировать свои обращения до момента их обработки, а также при необходимости удалять устаревшие или ошибочные заявки. При этом должны сохраняться соответствующие ограничения — например, запрет на удаление уже обработанных или закрытых заявок для сохранения целостности данных;

– Назначение ответственных исполнителей: В системе должна быть реализована функциональность распределения заявок между сотрудниками, ответственными за их рассмотрение и выполнение. Это может быть как автоматическое назначение на основании категорий и компетенций, так и ручное распределение администраторами или модераторами. Контроль текущего статуса исполнения позволяет отслеживать этапы обработки — от регистрации заявки до её полного закрытия;

– Ведение переписки через комментарии: Для эффективного взаимодействия по каждой заявке необходимо предоставлять возможность обмена комментариями между пользователями и исполнителями. Такая коммуникация помогает уточнять детали, оперативно реагировать на изменения и фиксировать результаты промежуточных этапов работы;

– Хранение истории изменений и действий: Важным элементом системы является ведение полной аудиторской записи всех действий с заявкой — создание, редактирование, смена статуса, комментарии и пр. Это обеспечивает прозрачность процессов, помогает в разрешении спорных ситуаций и позволяет проводить анализ эффективности обработки обращений;

– Разграничение прав доступа: В системе должна быть реализована гибкая модель управления доступом на основе ролей пользователей. Для разных категорий (например, обычный пользователь, модератор, администратор) предусматриваются разные возможности и ограничения — от простого просмотра и создания заявок до управления справочниками и полномасштабного администрирования системы;

– Административная панель: для удобства управления системой администраторам и ответственным лицам предоставляется специальный интерфейс, позволяющий управлять справочными данными (типы заявок, категории, приоритеты), контролировать роли пользователей, просматривать и анализировать статистику по обработке обращений, мониторить активность и загруженность исполнителей.

Нефункциональные требования

Нефункциональные требования направлены на обеспечение качества, надёжности и удобства эксплуатации системы. Они включают аспекты, не связанные напрямую с функционалом, но влияющие на общую производительность и восприятие пользователями:

– стабильность и масштабируемость: Система должна гарантировать корректную и бесперебойную работу даже при высокой нагрузке и большом количестве одновременных пользователей. Например, в периоды пиковых обращений (таких как сессии, отчетные периоды и пр.) важно сохранить

скорость отклика и стабильность без сбоев и задержек. Архитектура должна предусматривать возможность горизонтального и вертикального масштабирования для увеличения ресурсов по мере роста числа пользователей;

– простота и интуитивность интерфейса: Интерфейс должен быть разработан с акцентом на удобство конечных пользователей — минимальный порог входа, понятная навигация и логичное расположение элементов управления. Это снижает затраты времени на обучение и способствует более быстрому и эффективному использованию системы;

– адаптивность интерфейса: Система должна корректно отображаться и функционировать на различных устройствах — от полноразмерных мониторов настольных компьютеров до экранов смартфонов и планшетов. Использование адаптивной вёрстки и современных фронтенд-технологий обеспечит единообразие пользовательского опыта вне зависимости от платформы;

– безопасность и защита данных: Авторизация и аутентификация пользователей должны осуществляться с использованием надёжных и современных методов (например, токенов JWT, OAuth, либо интеграция с корпоративными системами учёта). Кроме того, система должна обеспечивать конфиденциальность данных, защиту от несанкционированного доступа и устойчивость к распространённым видам атак (SQL-инъекции, XSS, CSRF и др.). Необходимо предусмотреть шифрование данных как при передаче, так и при хранении;

– лёгкость сопровождения и поддержки: Архитектура и кодовая база должны быть построены с учётом простоты обслуживания и расширяемости. Это позволит оперативно исправлять ошибки, внедрять новые функции и адаптировать систему под изменяющиеся требования. Использование современных подходов и шаблонов проектирования способствует снижению затрат на сопровождение;

– высокая производительность: важно, чтобы время отклика системы при выполнении основных операций (создание заявки, обновление статуса, загрузка списка) было минимальным, что положительно скажется на пользовательском опыте и общей эффективности работы;

– отказоустойчивость и резервирование: Для минимизации рисков потери данных или простоя системы должна быть предусмотрена система резервного копирования, аварийного восстановления и мониторинга состояния компонентов;

– логирование и мониторинг: Система должна вести подробные журналы событий и ошибок, что позволит своевременно выявлять и устранять проблемы, а также анализировать работу системы для её оптимизации.

Таким образом, комплексное соблюдение перечисленных функциональных и нефункциональных требований является основой для создания качественной, надёжной и удобной системы внутренней обработки заявок, способной эффективно поддерживать процессы взаимодействия внутри организации.

1.3 Архитектура системы

Система управления тикетами реализована на основе классической двухуровневой архитектуры «клиент–сервер». Этот подход является одной из наиболее распространённых и проверенных временем архитектурных моделей, широко применяемых в разработке корпоративных и веб-приложений. Он позволяет чётко разграничить зоны ответственности между серверной и клиентской частями, что способствует повышению гибкости, масштабируемости и удобству поддержки системы.

Серверная часть системы отвечает за реализацию бизнес-логики, обработку запросов, управление данными и обеспечение безопасности. Для её создания выбран фреймворк ASP.NET Core Web API — современное, кроссплатформенное решение от Microsoft, ориентированное на высокую производительность и устойчивость. Данный стек характеризуется эффективным использованием ресурсов, поддержкой асинхронных операций, что особенно важно при одновременной обработке множества запросов, а также возможностью интеграции с разнообразными внешними сервисами и компонентами. Кроме того, использование языка программирования C# обеспечивает строгую типизацию, богатый инструментарий для разработки и отладки, что сокращает количество ошибок и ускоряет процесс создания надёжного программного продукта.

Клиентская часть реализована с помощью Razor Pages — технологии, входящей в состав ASP.NET Core, которая сочетает преимущества серверного рендеринга и простой организации пользовательского интерфейса. В отличие от сложных SPA-решений (Single Page Application), таких как Angular или React, Razor Pages обеспечивает более лёгкий и быстрый старт проекта, снижает требования к ресурсам клиента и упрощает процесс отладки и сопровождения. Это особенно актуально для внутренних корпоративных решений, где приоритетом является стабильность и предсказуемость работы интерфейса, а не сложные интерактивные визуальные эффекты.

Для работы с базой данных в проекте используется ORM (Object-Relational Mapping) Entity Framework Core. Этот инструмент значительно упрощает взаимодействие с реляционной базой данных, автоматически генерируя SQL-запросы на основе моделей данных, что снижает вероятность ошибок, связанных с ручным написанием кода для работы с БД. EF Core поддерживает миграции — механизм автоматического обновления структуры базы данных в соответствии с

изменениями моделей, что облегчает развитие и масштабирование системы. В сравнении с альтернативными решениями, такими как Dapper, EF Core предоставляет более высокий уровень абстракции и позволяет сосредоточиться на бизнес-логике, не отвлекаясь на тонкости оптимизации запросов.

Архитектура системы включает несколько ключевых модулей, обеспечивающих основные функциональные возможности и разделение ответственности:

- модуль авторизации и аутентификации пользователей, реализующий процессы входа в систему, управление правами доступа и безопасность пользовательских данных. Это гарантирует, что каждый пользователь будет иметь доступ только к тем функциям и данным, которые соответствуют его роли и полномочиям;
- модуль управления тикетами, предоставляющий средства для создания, редактирования, удаления и отслеживания заявок. Он обеспечивает хранение подробной информации о каждой заявке, контроль её статуса, а также назначение ответственных исполнителей;
- модуль комментариев и внутренней переписки, позволяющий участникам взаимодействовать внутри системы, уточнять детали, обмениваться дополнительной информацией и документировать процесс решения вопросов;
- модуль аналитики и фильтрации, предоставляющий инструменты для систематического анализа поступающих обращений, выявления типовых проблем, формирования статистических отчётов и принятия управленческих решений на основе собранных данных;

Таким образом, архитектура системы ориентирована на обеспечение высокой производительности, безопасности, удобства использования и возможности масштабирования в будущем. Благодаря использованию современных технологий и чёткой модульной структуре, система легко адаптируется под изменения требований и интегрируется с внешними компонентами.

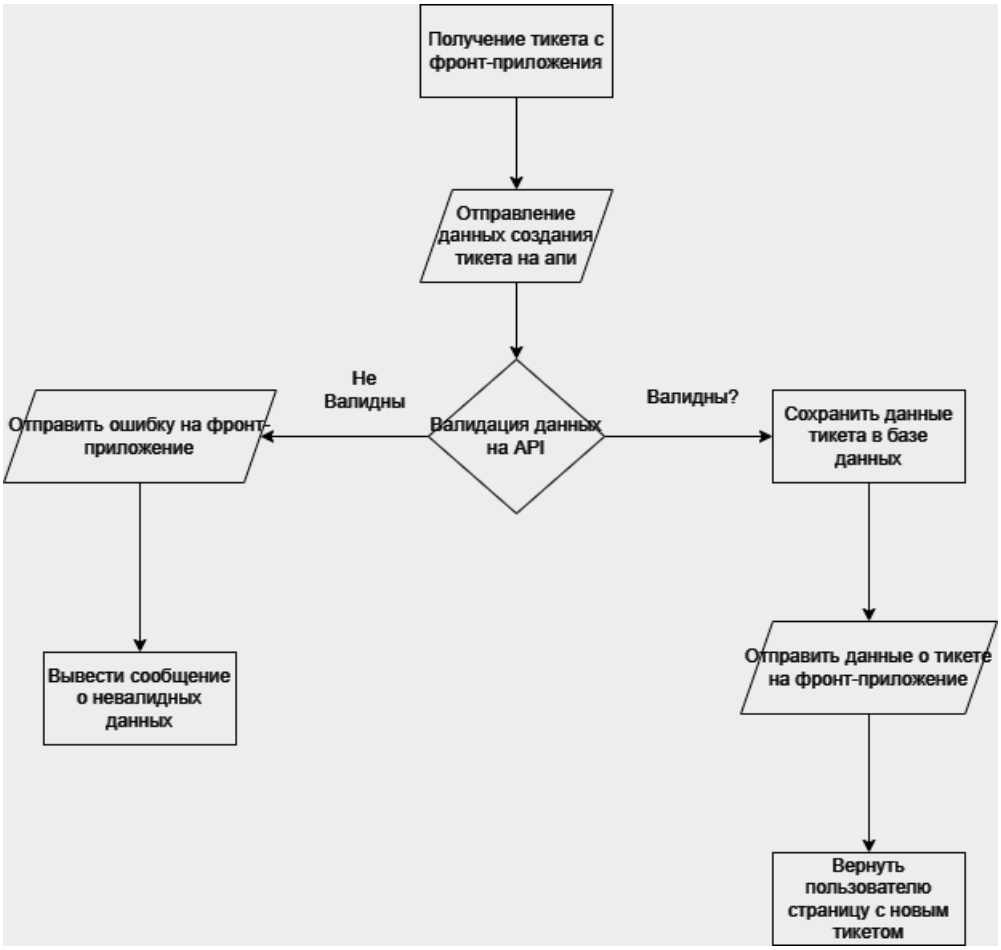


Рисунок 1.2 Блок-схема системы.

1.4 Структура базы данных

Для хранения и обработки данных используется реляционная СУБД Microsoft SQL Server. Структура базы данных разрабатывалась на основе принципов нормализации и гибкости. Основное внимание уделялось сохранению логической целостности, минимизации избыточности и обеспечению быстрого доступа к информации.

Основные таблицы:

- Users — содержит данные пользователей системы, включая их уникальные идентификаторы, ФИО, контактные данные, а также информацию о ролях, отделах и правах доступа. Эта таблица является основной для управления пользователями внутри системы и отвечает за разграничение полномочий;
- Ticket — центральная таблица, в которой хранятся заявки (тикеты). В ней содержатся ключевые поля: тема заявки, описание, статус, приоритет, исполнитель, категория, а также информация о дате создания и обновления.

Таблица связана с другими сущностями для детального учёта жизненного цикла заявки;

– Comment — таблица для хранения комментариев к заявкам, позволяющая вести историю коммуникаций, уточнений и пояснений по каждой заявке. Это обеспечивает прозрачность взаимодействия между пользователями и исполнителями;

– TicketResponse — фиксирует действия и события, связанные с заявками, такие как изменение статуса, назначение исполнителей, отклонения и прочее. Данная таблица помогает отслеживать все изменения и действия, что важно для аудита и контроля;

– dic_Category, dic_Division, dic_Position, dic_Status — справочники, содержащие категории заявок, подразделения, должности пользователей и статусы заявок соответственно. Использование таких справочников повышает консистентность данных и облегчает фильтрацию и группировку информации;

– Roles — таблица, определяющая роли пользователей в системе и их права доступа, что обеспечивает безопасность и разграничение полномочий при работе с данными.

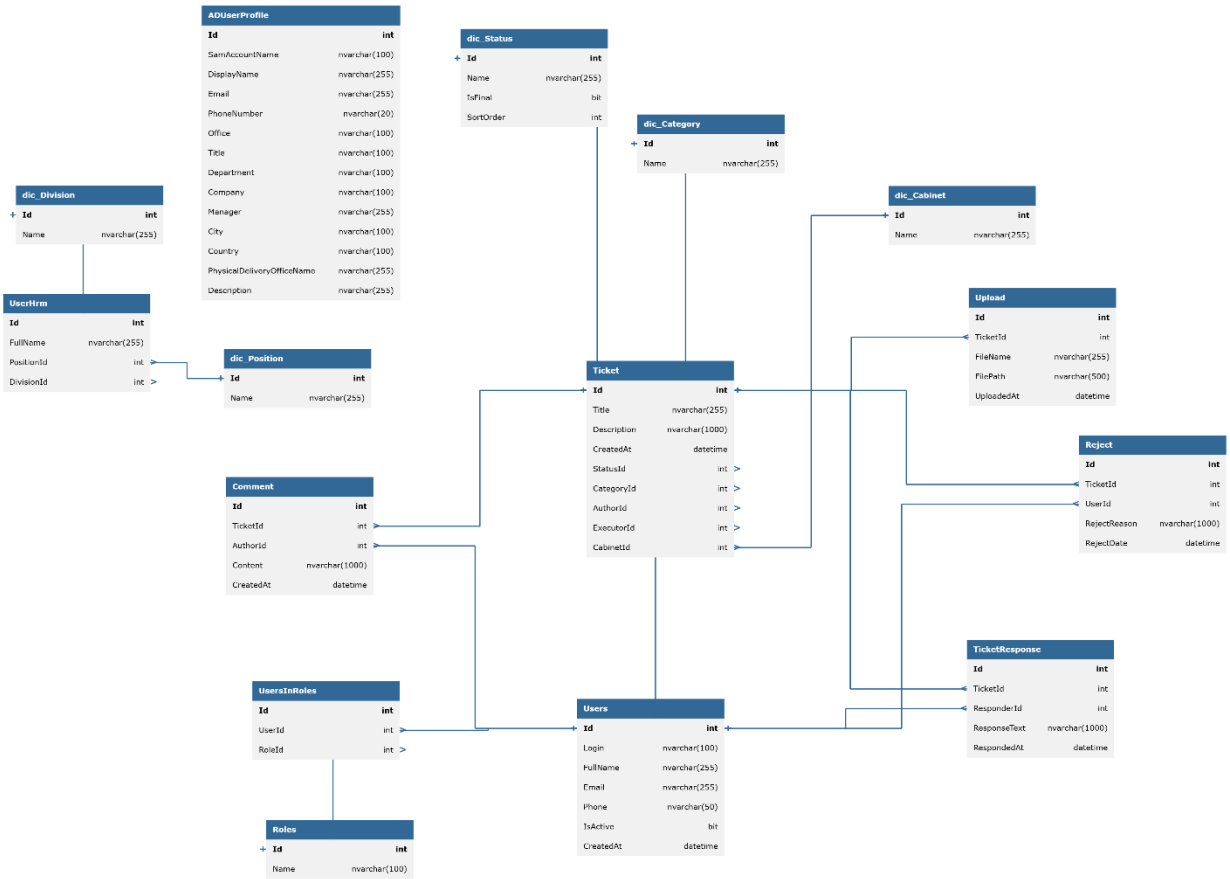


Рисунок 1.3 — ER-диаграмма базы данных с основными сущностями и связями между ними.

Сравнительно с альтернативными СУБД, такими как PostgreSQL или MySQL, выбор в пользу MSSQL был обусловлен полной совместимостью с .NET-экосистемой, наличием встроенных средств репликации, высокой скоростью обработки транзакций и удобными средствами администрирования (SQL Server Management Studio).

1.5 Методология и планирование разработки

В ходе проектирования и разработки использовалась гибкая методология Agile с внедрением принципов Scrum. Такой подход позволил создавать продукт итерационно, своевременно вносить изменения и проводить регулярное тестирование, что существенно повышало качество конечного результата.

Команда состояла из двух человек: один отвечал за разработку API-составляющей (backend), а второй — за фронтенд-часть приложения. Такое распределение ролей позволило сфокусироваться на отдельных модулях, обеспечивая параллельную работу и сокращая общее время разработки.

Каждая итерация (спринт) продолжалась 1–2 недели и включала следующие этапы:

- планирование — определение задач и приоритетов на спринт, оценка трудозатрат и распределение работы между участниками;
- реализация — разработка функционала согласно плану спринта, с частыми коммитами и интеграцией изменений;
- тестирование — выполнение модульного и интеграционного тестирования, а также ручного тестирования интерфейса;
- анализ результатов — оценка выполненных задач, обсуждение возникших проблем и подготовка плана на следующий спринт.

Для управления задачами и контроля прогресса использовались инструменты Trello и Jira, что обеспечивало прозрачность процесса и возможность отслеживать статус каждой задачи в реальном времени.

Ежедневные стендапы (daily scrum) помогали поддерживать постоянную коммуникацию между разработчиками, быстро выявлять и устранять препятствия. Ретроспективы в конце каждого спринта способствовали анализу эффективности процессов и внедрению улучшений.

В сравнении с традиционной каскадной моделью (Waterfall), Scrum обеспечил большую гибкость в управлении изменениями и поддержании высокого уровня вовлечённости команды, что было особенно важно в условиях ограниченного ресурса и необходимости быстрого реагирования на изменения требований.

1.6 Принципы проектирования пользовательского интерфейса

При проектировании пользовательского интерфейса системы основной акцент делался на удобство, простоту и универсальность. Интерфейс должен быть интуитивно понятен даже тем пользователям, которые не имеют опыта работы с ИТ-системами, что обеспечивает минимальный порог вхождения и снижает необходимость в дополнительном обучении.

Основные принципы проектирования включают:

- Лаконичность — интерфейс не перегружен лишними элементами и визуальными отвлекающими факторами. Все необходимые функции и элементы управления расположены логично и доступны без излишних переходов, что позволяет пользователю быстро ориентироваться и выполнять задачи с минимальными усилиями;
- предсказуемость — архитектура интерфейса и навигация построены таким образом, чтобы пользователь всегда понимал, где найти нужную функцию или информацию. Использование общепринятых стандартов и шаблонов интерфейса повышает узнаваемость элементов управления и снижает когнитивную нагрузку;
- адаптивность — интерфейс обеспечивал корректное отображение и удобное взаимодействие на различных устройствах: настольных ПК, ноутбуках, планшетах и смартфонах. Это достигается использованием современных технологий адаптивного веб-дизайна, позволяющих автоматически подстраиваться под размер экрана и особенности устройств ввода;
- контекстные подсказки и уведомления — система предоставляет пользователю своевременную и релевантную информацию о текущем состоянии, результатах действий и возможных ошибках. Это достигается за счёт всплывающих подсказок, модальных окон и информативных сообщений, что повышает комфорт и снижает вероятность ошибок.

В отличие от некоторых промышленных решений, ориентированных на профессиональных пользователей с техническим бэкграундом (например, ServiceNow), пользовательский интерфейс данного проекта разрабатывался с

прицелом на повседневную простоту и лёгкость обучаемости. Это расширяет круг потенциальных пользователей и обеспечивает их эффективную работу с системой без длительного обучения.

Для реализации интерфейса использовался фреймворк Bootstrap, который позволил быстро создать современный и адаптивный дизайн с использованием готовых компонентов и сеток. Для динамического взаимодействия с серверной частью применялись современные JavaScript-технологии — fetch API и AJAX — что обеспечило плавную и отзывчивую работу интерфейса без полной перезагрузки страниц. Такой подход позволил реализовать удобный пользовательский опыт и повысил общую производительность системы.

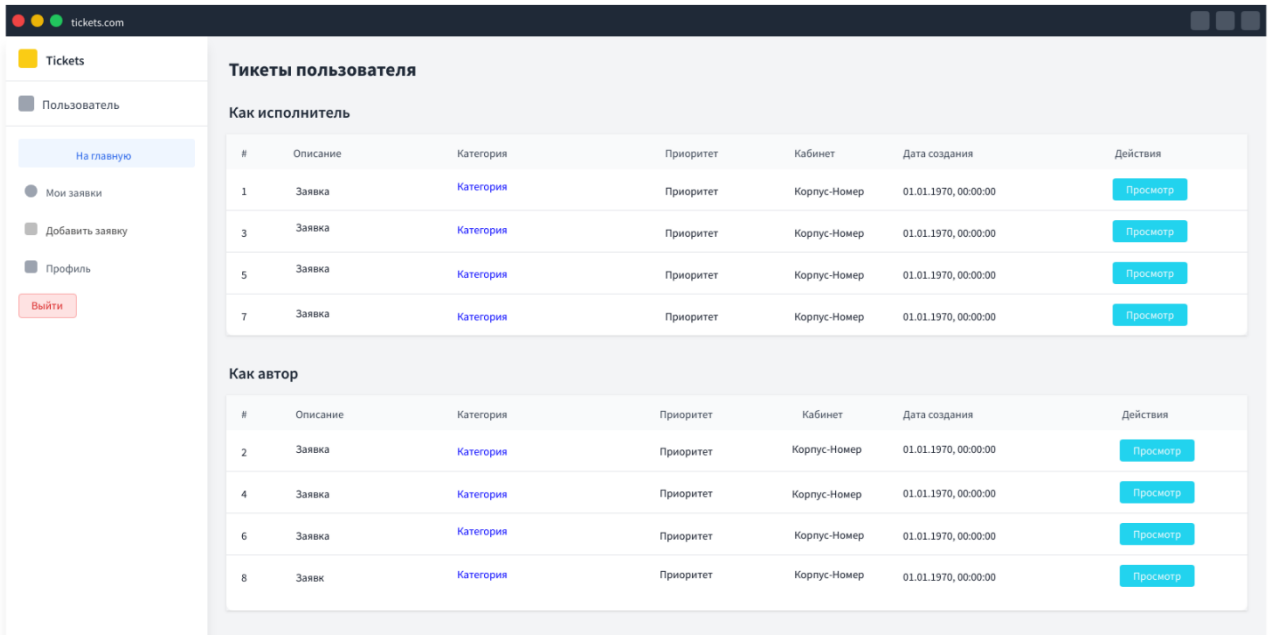


Рисунок 1.4 — Пример макета главной страницы: меню, список заявок, фильтры и кнопки действий (wireframe).

1.7 Ожидаемый эффект от внедрения

Внедрение информационной системы управления заявками направлено на существенное улучшение процессов внутри организации и повышение общей эффективности работы. Ожидается, что реализованное решение обеспечит ряд значимых положительных изменений, которые будут касаться как операционной деятельности, так и стратегического управления.

Основные ожидаемые эффекты включают:

– сокращение времени обработки заявок на 30–40% - благодаря автоматизации ключевых этапов работы — регистрации, распределения,

обработки и контроля исполнения заявок — значительно уменьшается временной лаг между поступлением запроса и его решением. Это достигается за счёт исключения ручных операций, автоматических уведомлений ответственных сотрудников и прозрачной системы приоритетов;

– снижение нагрузки на сотрудников и повышение прозрачности взаимодействий - система позволяет централизованно хранить всю информацию по заявкам и коммуникациям, что минимизирует необходимость повторных запросов, снижает количество ошибок и упрощает обмен информацией между отделами. Ответственные сотрудники получают своевременные уведомления и могут легко контролировать текущий статус задач, что уменьшает стресс и повышает удовлетворённость работой;

– формирование аналитических отчётов и мониторинг на основе реальных данных - внедрение системы открывает новые возможности для сбора и анализа данных, позволяя руководству принимать решения на основе объективной информации. Автоматизированные отчёты по времени обработки, загруженности сотрудников, категориям заявок и другим ключевым метрикам помогут выявлять узкие места и оптимизировать бизнес-процессы;

– повышение исполнительской дисциплины и общей продуктивности команды - за счёт чёткой фиксации сроков, ответственных исполнителей и этапов выполнения задач, система стимулирует соблюдение регламентов и внутреннего распорядка. Видимость статусов заявок и истории изменений мотивирует сотрудников работать эффективнее и дисциплинированнее, что положительно отражается на качестве обслуживания и результатах деятельности;

– улучшение коммуникации и качества обслуживания - в отличие от традиционных каналов связи, таких как электронная почта или бумажные журналы регистрации, информационная система обеспечивает мгновенную обратную связь, историю взаимодействий и возможность быстрого реагирования на запросы. Это значительно повышает уровень удовлетворённости внутренних и внешних пользователей, снижает вероятность потери или забывания заявок;

– доступность и прозрачность документооборота в реальном времени - все заинтересованные стороны имеют круглосуточный доступ к актуальной информации, что упрощает контроль и управление процессами. Возможность быстро находить нужные данные и видеть полную картину взаимодействия способствует оперативному принятию решений и снижению административных затрат.

Стратегическое значение и влияние на конкурентоспособность организации:

Внедрение данной системы способствует созданию более организованной, управляемой и технологически продвинутой внутренней среды. Это позволяет повысить скорость и качество выполнения бизнес-задач, адаптироваться к изменяющимся требованиям и быстро реагировать на вызовы рынка. В итоге организация получает конкурентное преимущество за счёт:

- быстрого и точного выполнения заявок и заказов;
- уменьшения операционных рисков и ошибок;
- улучшения взаимодействия между подразделениями;
- повышения мотивации и вовлечённости сотрудников.

Таким образом, проект не только решает текущие задачи автоматизации, но и создаёт фундамент для устойчивого развития и масштабирования бизнес-процессов в будущем. Данная теоретическая база подтверждает актуальность и целесообразность реализации решения, а также демонстрирует его потенциал в значительном повышении эффективности и качества работы организации.

2. Проектирование информационной системы

2.1 Разработка back-end приложения

2.1.1 Архитектурный подход и выбор технологий

На начальном этапе реализации серверной части дипломного проекта особое внимание было уделено выбору среды разработки, технологий и инструментов, которые бы обеспечили эффективность, надёжность и удобство при построении веб-приложения. Главной задачей этого этапа стало формирование технологической базы, на которой будет реализована основная бизнес-логика приложения.

Одним из важнейших принципов, лежащих в основе современного программирования, является **разделение ответственности (Separation of Concerns, SoC)**. В рамках этого подхода клиентская и серверная части приложения функционируют независимо друг от друга: клиент отвечает за отображение интерфейса и обработку пользовательских действий, в то время как сервер занимается хранением данных, обработкой запросов и реализацией бизнес-правил. Такой подход позволяет повысить модульность системы, облегчить её сопровождение и упростить масштабирование.

В качестве основной платформы для разработки серверной части было выбрано **ASP.NET Core Web API** — современный, кроссплатформенный фреймворк от компании Microsoft. Среди множества доступных решений ASP.NET Core был предпочтён за счёт своей высокой производительности, гибкости и активной поддержки со стороны сообщества разработчиков. Он идеально подходит для построения REST-сервисов, что делает его особенно удобным для интеграции с различными клиентами — будь то веб-интерфейс, мобильное приложение или внешние системы.

Дополнительным преимуществом ASP.NET Core Web API стало наличие встроенной поддержки таких важных элементов, как маршрутизация HTTP-запросов, внедрение зависимостей (Dependency Injection), автоматическая сериализация и десериализация данных, а также возможность быстрой интеграции со средствами документирования, такими как Swagger. Это существенно ускоряет процесс разработки и тестирования, а также делает API более понятным и доступным для других участников команды.

Также важным фактором в выборе технологий стало наличие **Entity Framework Core** — объектно-реляционного маппера, входящего в состав платформы .NET. Он позволяет работать с базой данных через объектную модель, без необходимости написания большого количества SQL-кода вручную.

Это ускоряет процесс разработки и снижает вероятность ошибок при работе с данными. Entity Framework Core автоматически управляет миграциями базы данных, что облегчает поддержку и обновление схемы хранения данных по мере развития проекта.

Средой разработки, используемой в процессе написания серверной части, выступила Microsoft Visual Studio 2022 — профессиональная интегрированная среда разработки (IDE), предоставляющая мощные инструменты для создания, тестирования и отладки .NET-приложений. Visual Studio предлагает широкий функционал: подсветку синтаксиса, средства автоматической генерации кода, встроенную поддержку систем контроля версий (например, Git), интеграцию с базами данных и прочее. Использование данной среды позволило сосредоточиться на бизнес-логике приложения, минимизируя временные затраты на технические настройки и поиск ошибок.

Кроме того, при подготовке проекта к разработке были установлены и настроены необходимые вспомогательные инструменты:

- SQL Server Management Studio (SSMS) — для работы с базой данных;
- Git и GitHub — для организации системы контроля версий и хранения исходного кода;
- .NET CLI (Command Line Interface) — для работы с проектом через консоль, включая запуск миграций и сборку проекта;

Также была настроена система управления зависимостями и конфигураций через встроенные механизмы .NET. Использование конфигурационных файлов, таких как appsettings.json, позволяет удобно управлять строками подключения, параметрами логирования и другими настройками без необходимости модифицировать исходный код.

На основании вышеописанных критериев и требований к проекту, выбранный стек технологий оказался оптимальным для решения поставленных задач. Все выбранные инструменты обеспечили необходимый уровень производительности, расширяемости и сопровождения, а также позволили ускорить реализацию ключевых функций серверной части дипломного проекта.

2.1.2 Планирование и проектирование серверной логики

Разработка серверной части дипломного проекта началась с этапа проектирования модели предметной области, что является ключевым моментом в построении любой информационной системы. Основной целью данного этапа являлось формирование чёткого понимания того, какие сущности будут существовать в системе, как они будут взаимодействовать между собой, и каким образом можно будет управлять данными с помощью API-интерфейса.

Проектирование предметной области

На начальном этапе была составлена схема предметной области. Для этого были выделены ключевые сущности, отражающие бизнес-логику и реальные объекты системы, а именно:

- Ticket (Заявка) — основная сущность, представляющая собой сообщение пользователя о проблеме или запросе;
- User (Пользователь) — отправитель или исполнитель заявки;
- Comment (Комментарий) — сообщения, прикреплённые к заявке;
- Category (Категория) — классификация заявок по тематике;
- Status (Статус) — текущее состояние заявки (например: «Создано», «В работе», «Закрыто»);
- Reject (Причина отклонения) — используется, если заявка была отклонена;
- Role (Роль пользователя) — определяет уровень доступа (например, «Пользователь», «Администратор»).

После выделения основных сущностей были установлены **взаимосвязи** между ними:

- один пользователь может иметь несколько заявок;
- одна заявка может содержать несколько комментариев;
- каждая заявка принадлежит к одной категории и имеет один текущий статус;
- заявка может быть отклонена по конкретной причине (reject);
- пользователь может иметь одну или несколько ролей.

На основе этих данных была построена ER-диаграмма, отражающая структуру и связи в базе данных. Данная диаграмма легла в основу создания таблиц в базе данных с использованием Entity Framework Core.

Рисунок 2.1 Классы моделей сгенерированных EF Core.

Определение ресурсов и маршрутов

Следующим этапом стало определение ресурсов API и их маршрутов, что необходимо для реализации REST-интерфейса взаимодействия между клиентом и сервером. Каждый тип ресурса был сопоставлен с соответствующим маршрутом:

- GET /api/tickets — получение списка всех заявок;
- GET /api/tickets/{id} — получение информации о конкретной заявке;
- POST /api/tickets — создание новой заявки;
- PUT /api/tickets/{id} — редактирование существующей заявки;
- DELETE /api/tickets/{id} — удаление заявки.

Аналогичным образом были определены маршруты для других ресурсов:

- /api/users — для управления пользователями;
- /api/comments — для работы с комментариями;
- /api/categories, /api/statuses, /api/rejects, /api/roles — для работы со справочниками.

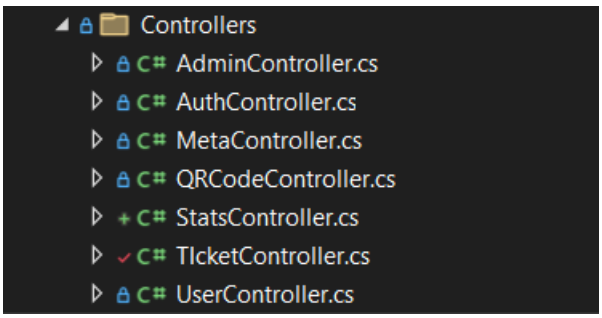


Рисунок 2.2 Основные контроллеры, отвечающие за маршрутизацию.

Данный подход обеспечил логичную и структурированную маршрутизацию в рамках REST-парадигмы.

Формирование структуры проекта

После определения логики взаимодействия с данными был создан сам проект на основе шаблона ASP.NET Core Web API. Внутри него была реализована многоуровневая архитектура, предполагающая разделение логики по слоям. Такая структура улучшает читаемость кода, упрощает сопровождение и способствует повторному использованию компонентов. Были выделены следующие основные папки и компоненты:

- Controllers — контроллеры, реализующие конечные точки API, получающие и обрабатывающие HTTP-запросы;
- Services — бизнес-логика приложения. Каждый сервис реализует операции, связанные с конкретной сущностью;
- Entities (или Models) — C#-классы, представляющие сущности базы данных. Содержат свойства, отражающие колонки таблиц;
- DTOs (Data Transfer Objects) — модели, предназначенные для передачи данных между клиентом и сервером. DTO позволяют скрыть внутреннюю структуру сущностей и обеспечить безопасность;
- Data — класс контекста базы данных (DbContext) и миграции базы данных. Именно здесь определяется, какие сущности будут включены в модель базы данных и каким образом они будут связаны.

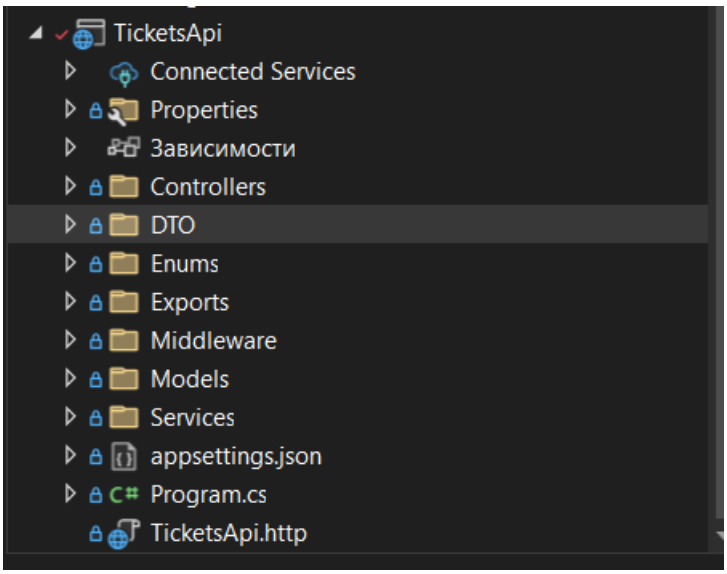


Рисунок 2.3 Схема слоев приложения

Каждый слой имеет чётко определённую ответственность. Такой подход соответствует принципам чистой архитектуры и значительно упрощает разработку и тестирование отдельных модулей.

Настройка проекта и инициализация

На этом этапе были выполнены следующие технические действия:

- установка необходимых NuGet-пакетов Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.SqlServer, AutoMapper, Swashbuckle.AspNetCore (для Swagger) и других;
- создание файла подключения к базе данных в appsettings.json;
- регистрация зависимостей в Startup.cs или Program.cs через встроенный механизм Dependency Injection;
- настройка Entity Framework Core и инициализация первой миграции, содержащей схему таблиц.

Таким образом, на данном этапе была заложена прочная основа для всей дальнейшей серверной логики дипломного проекта. Выбранная структура проекта позволила чётко разграничить зоны ответственности между слоями, обеспечить масштабируемость и упростить реализацию как простых, так и сложных бизнес-процессов. Такой подход обеспечивает высокую надёжность и гибкость разрабатываемой системы, а также облегчает сопровождение кода на всех этапах жизненного цикла приложения.

2.1.3 Реализация контроллеров

После завершения проектирования структуры приложения и создания моделей предметной области следующим этапом стала реализация контроллеров — компонентов, отвечающих за приём и обработку HTTP-запросов от клиента. Контроллеры представляют собой связующее звено между пользовательскими запросами и бизнес-логикой, реализованной в сервисах.

В соответствии с архитектурными принципами ASP.NET Core Web API, для каждого основного ресурса системы был создан отдельный контроллер. Это позволило организовать код по REST-принципам и обеспечить читаемость, модульность и расширяемость приложения.

Контроллер для заявок (TicketController)

Одним из ключевых компонентов стал контроллер TicketController, реализующий взаимодействие с сущностью **заявки (Ticket)**. Он был размещён в папке Controllers и включал в себя методы, соответствующие основным HTTP-глаголам:

- GET /api/tickets — получение списка всех заявок;
- GET /api/tickets/{id} — получение информации о конкретной заявке по идентификатору;
- POST /api/tickets — создание новой заявки;
- PUT /api/tickets/{id} — обновление существующей заявки;
- DELETE /api/tickets/{id} — удаление заявки.

Каждый метод контроллера использует соответствующий сервисный класс, который инкапсулирует бизнес-логику. Таким образом, контроллер не содержит тяжёлой логики обработки данных, а выполняет роль маршрутизатора: получает входные данные, вызывает нужный метод сервиса и возвращает ответ клиенту.

Такой подход позволяет:

- соблюдать принцип единственной ответственности (Single Responsibility Principle);
- повысить тестируемость контроллеров;
- облегчить сопровождение и расширение API.

Контроллеры других сущностей (пользователи, комментарии, категории, статусы и др.) реализованы аналогичным образом — каждый ресурс имеет свой набор HTTP-методов и маршрутов, соответствующих CRUD-операциям.

```
namespace TicketsApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    Ссылок: 1
    public class TicketController : ControllerBase
    {
        private readonly ITicketService _ticketService;
        Ссылок: 0
        public TicketController(ITicketService ticketService)
        {
            _ticketService = ticketService;
        }

        [Authorize]
        [HttpPost("add")]
        Ссылок: 0
        public async Task<IActionResult> AddTicket(CreateTicketDTO ticketDto)
        {
            var userIdClaim = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
            if (string.IsNullOrEmpty(userIdClaim))
                return Unauthorized("Пользователь не авторизован.");

            int authorId = int.Parse(userIdClaim);

            int createdTicketId = await _ticketService.CreateTicketAsync(ticketDto, authorId);

            return CreatedAtAction(nameof(GetTicketById), new { ticketId = createdTicketId }, new { createdTicketId });
        }
```

Рисунок 2.4 Пример контроллера

2.1.4 Вынесение логики в сервисы

Для обеспечения чистоты архитектуры и повышения удобства поддержки проекта была принята практика разделения ответственности между контроллерами и сервисами. Контроллеры отвечают только за приём и отправку HTTP-запросов и ответов, а всю бизнес-логику — обработку данных, валидацию, выполнение правил и взаимодействие с данными — перенесли в отдельные классы, называемые сервисами.

Процесс создания сервисов

Разработка сервисов начиналась с выделения ключевых операций, которые должны выполняться над каждой сущностью предметной области. Например, для работы с тикетами сервисы обеспечивают создание новой заявки, обновление существующей, получение списка с возможностью фильтрации и сортировки, а также удаление. Для каждой из таких задач был определён набор методов в сервисном классе.

При этом сервисы получают данные от контроллеров, обрабатывают их согласно бизнес-правилам, могут выполнять дополнительные проверки и преобразования. После этого сервисы взаимодействуют с базой данных через слои репозиториев, обеспечивая тем самым удобный и централизованный доступ к данным.

Для интеграции сервисов в общую архитектуру приложения была реализована система внедрения зависимостей (Dependency Injection), которая

позволяет автоматически создавать и передавать сервисы в контроллеры. Благодаря этому контроллеры остаются максимально простыми — они вызывают только необходимые методы сервисов для выполнения задач.

Такой подход обеспечил удобство тестирования, так как сервисы можно было легко заменить на их мок-версии при написании юнит-тестов, что значительно ускоряло процесс отладки и проверки бизнес-логики. Кроме того, вынос логики в сервисы позволил многократно использовать одни и те же методы в разных частях приложения без дублирования кода.

Преимущества выделения сервисного слоя

- модульность — каждая бизнес-операция реализована в отдельном сервисном методе, что облегчает сопровождение и расширение функционала;
- читаемость кода — контроллеры содержат минимум кода, связанного с логикой, что улучшает понимание структуры проекта;
- тестируемость — сервисы можно изолированно тестировать, что повышает качество программного продукта;
- гибкость — сервисы позволяют централизованно менять логику без необходимости изменения контроллеров и других частей системы.

В итоге внедрение сервисного слоя стало важным шагом в построении надёжной, масштабируемой и удобной для сопровождения серверной части веб-приложения.

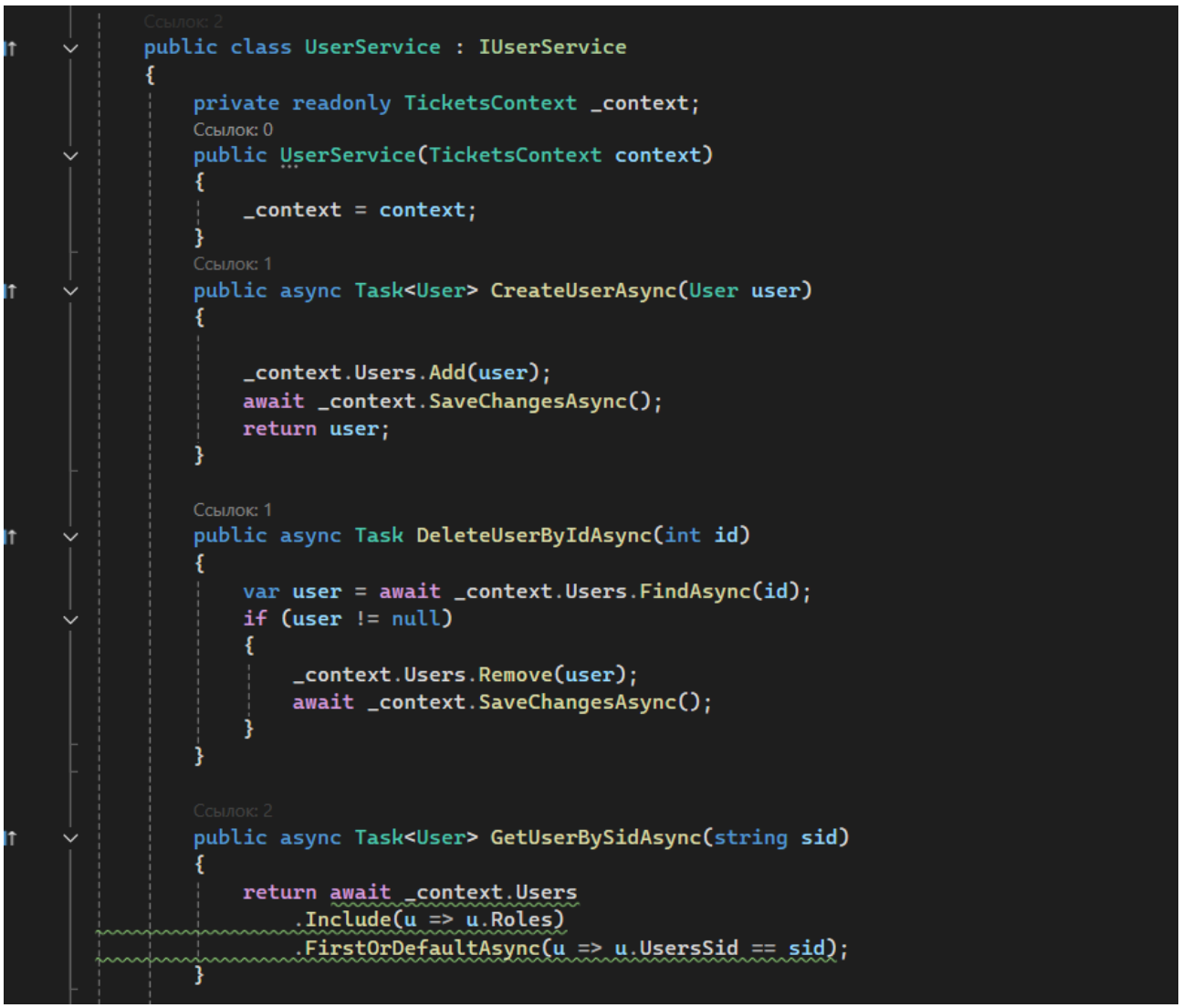


Рисунок 2.5 Пример сервиса

2.1.5 Работа с базой данных: Entity Framework Core

Для эффективного взаимодействия с базой данных в рамках проекта была использована объектно-реляционная библиотека Entity Framework Core (EF Core). EF Core представляет собой современную, лёгкую и расширяемую ORM (Object-Relational Mapper) от Microsoft, которая позволяет разработчику работать с базой данных на уровне объектов C#, избавляя от необходимости писать сложные SQL-запросы вручную.

Использование EF Core обеспечило более быструю разработку, упрощённое сопровождение и гибкое управление данными в приложении.

Основные этапы работы с EF Core:

Создание моделей сущностей - На первом этапе были созданы классы, которые отражают структуру данных предметной области. К ним относятся основные сущности: тикеты (Ticket), пользователи (User), комментарии

(Comment), категории (Category), статусы (Status) и другие. Каждая сущность описывает свойства, соответствующие столбцам таблиц базы данных, а также связи между ними — например, один пользователь может иметь множество тикетов, а каждый тикет может иметь несколько комментариев.

Описание контекста базы данных - Следующим шагом был создан класс контекста базы данных — TicketsDbContext. Этот класс является основным связующим звеном между приложением и базой данных. В нем определяется набор DbSet-свойств, каждое из которых соответствует таблице в базе. Контекст управляет подключением к базе, отслеживанием изменений в объектах и выполнением запросов.

Конфигурация DbSet-ов и Fluent API - Для точного описания структуры таблиц, ограничений, типов данных и отношений между сущностями использовались возможности Fluent API — гибкого и мощного инструмента конфигурации EF Core. С помощью Fluent API можно задавать правила каскадного удаления, уникальные индексы, обязательность или необязательность полей, типы связей «один-ко-многим», «многие-ко-многим» и другие.

Применение миграций - Для автоматизации управления схемой базы данных использовались миграции — механизм EF Core, который позволяет последовательно изменять структуру базы данных на основе изменений моделей сущностей. С помощью команд Add-Migration создавались миграции, фиксирующие изменения схемы, а команда Update-Database применяла эти изменения к базе данных. Такой подход обеспечивает синхронизацию кода и структуры базы, облегчая обновления и развёртывание.

Настройка строки подключения - Для подключения к SQL Server была настроена строка подключения, содержащая адрес сервера, имя базы данных и параметры безопасности. Она хранится в конфигурационных файлах проекта и используется контекстом базы данных для установления соединения.

Значимость использования EF Core -Применение Entity Framework Core значительно упростило работу с данными и позволило разработчикам сосредоточиться на бизнес-логике, а не на низкоуровневых деталях взаимодействия с базой. Благодаря EF Core:

- Процесс разработки стал быстрее за счёт возможности оперировать привычными объектами и коллекциями, а не сложными SQL-запросами;
- Уменьшилась вероятность ошибок, связанных с ручным написанием запросов и неправильной обработкой данных;
- Стала возможной простая миграция схемы базы данных при изменении требований;
- Улучшилась читаемость и поддерживаемость кода, что важно для дальнейшего развития и масштабирования проекта.

В целом, использование EF Core стало одним из ключевых факторов успешной реализации серверной части дипломного проекта, обеспечив удобный и эффективный способ работы с реляционной базой данных.

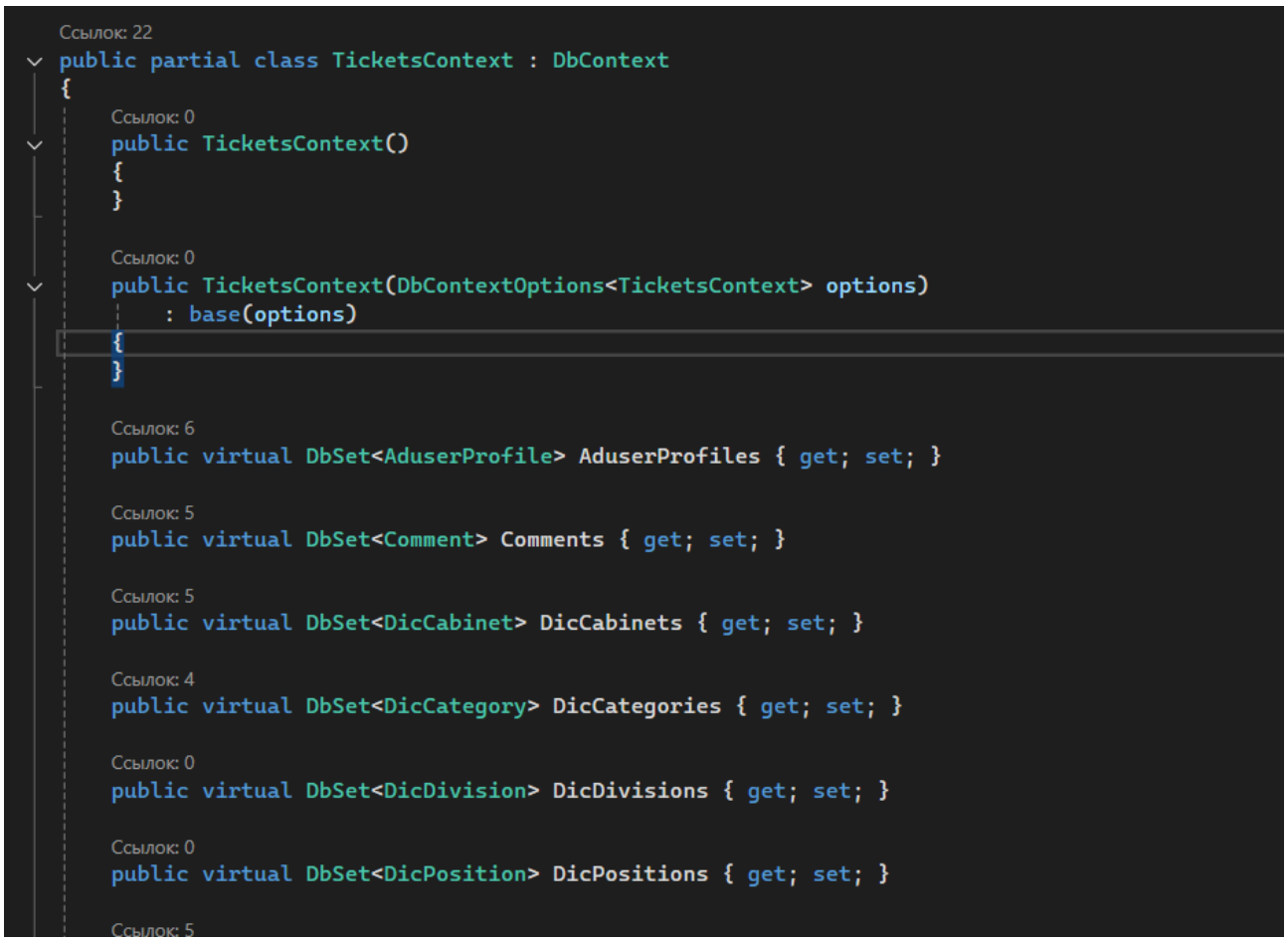


Рисунок 2.6 Использование DbContext для взаимодействий с БД

2.1.6 Использование DTO-моделей

В процессе разработки серверной части дипломного проекта особое внимание было уделено организации обмена данными между клиентской и серверной частями приложения. Для этого применялись модели передачи данных — DTO (Data Transfer Object). DTO представляют собой специализированные классы, которые содержат только те данные, которые необходимо отправить или получить через API, скрывая при этом внутреннюю структуру базовых сущностей и обеспечивая контроль над передаваемой информацией.

Использование DTO-моделей решает сразу несколько важных задач:

– сокрытие внутренней структуры сущностей: DTO позволяют абстрагироваться от конкретной реализации базы данных и модели предметной области. Это предотвращает прямой доступ клиента к внутренним деталям сущностей, таким как служебные поля или поля, используемые исключительно на сервере;

– безопасность данных: При передаче данных через API часто требуется исключать чувствительную информацию. Например, модель пользователя (User) может содержать поля с паролями, хешами или другими конфиденциальными сведениями, которые не должны быть раскрыты клиенту. DTO позволяют исключить такие поля из передаваемых данных, снижая риск утечки информации;

– упрощение и стандартизация данных: DTO могут включать только необходимые для клиента поля, что уменьшает объём передаваемой информации и улучшает производительность. Кроме того, с помощью DTO можно консолидировать данные из нескольких сущностей в одну модель для удобства отображения на клиенте;

– гибкость и масштабируемость API: DTO обеспечивают возможность изменять внутреннюю структуру данных без нарушения контракта с клиентом. Это особенно важно при развитии и расширении функционала, когда изменяются модели базы данных, но старые версии API должны продолжать корректно работать.

Примеры использования DTO в проекте

– TicketDto — содержит основные поля заявки, необходимые для отображения в списке и деталях, например, номер, заголовок, статус, дату создания. При этом в эту модель не включались служебные поля или внутренние идентификаторы;

– CreateTicketDto — отдельная модель, используемая для создания новых тикетов. Она включает только те поля, которые клиент должен предоставить при создании заявки, исключая, например, поля статуса или даты, которые устанавливаются автоматически на сервере;

– UserDto — модель для передачи информации о пользователях, исключающая конфиденциальные данные, такие как пароль, хеш пароля и токены аутентификации.

Применение DTO в API

В процессе обработки запросов контроллеры принимают DTO-модели от клиента и передают их сервисам для дальнейшей обработки. Аналогично, при отправке данных клиенту, сервер конвертирует внутренние сущности в DTO, обеспечивая единообразный и безопасный формат данных.

Для преобразования между сущностями и DTO использовались специализированные мапперы — классы или сторонние библиотеки (например, AutoMapper), которые автоматизируют и упрощают процесс конвертации, снижая количество повторяющегося кода.

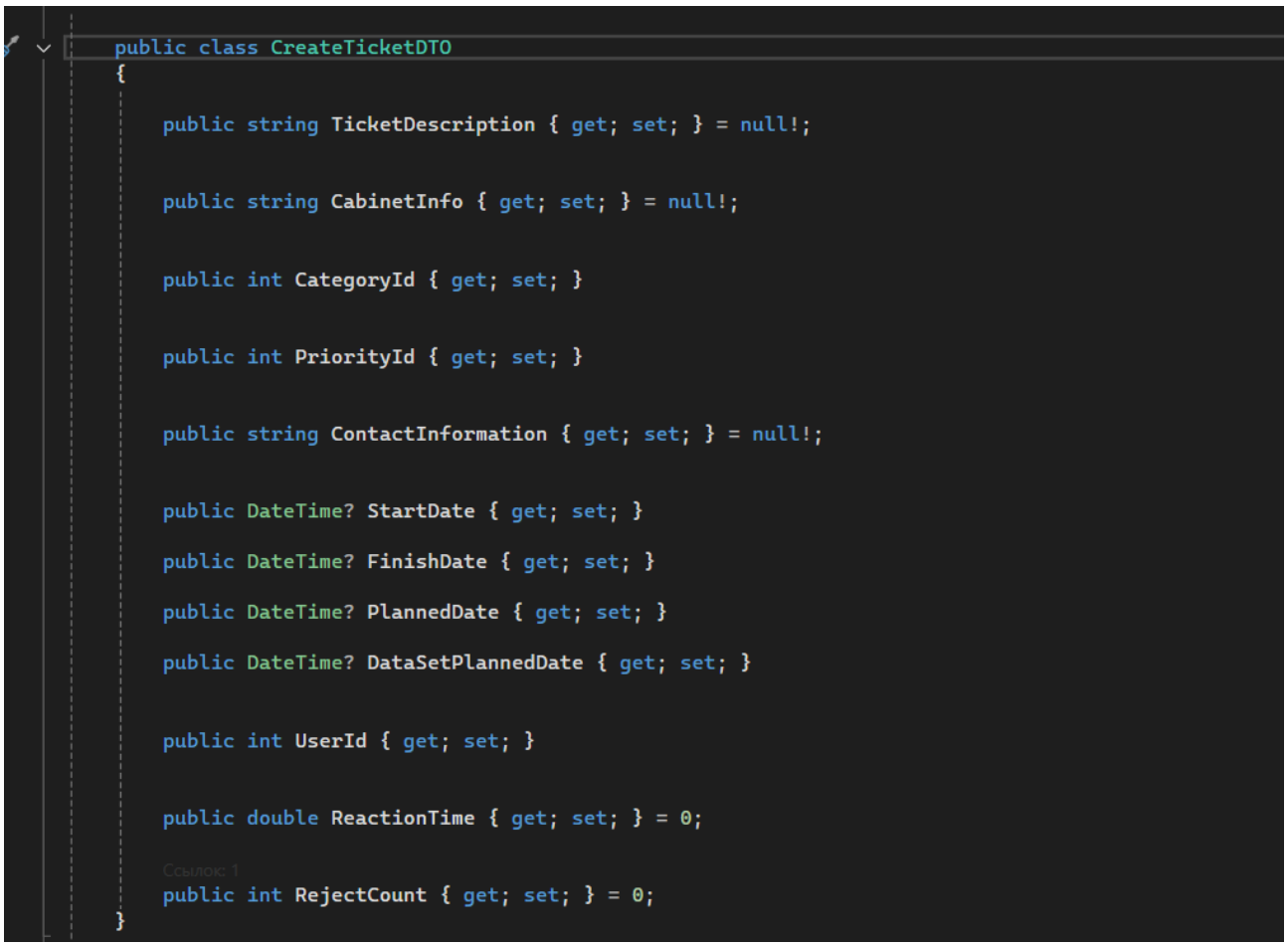


Рисунок 2.7 Пример файла ДТО.

Стоит отметить, что Web API в ASP.NET Core изначально спроектирован с учётом расширяемости и масштабируемости. Он легко может быть адаптирован под рост нагрузки, добавление новых функций и интеграцию с другими сервисами. Благодаря использованию стандартов REST и протокола HTTP, API можно использовать в самых разных типах клиентов — от традиционных браузеров до мобильных устройств и даже промышленных систем автоматизации.

2.1.7 Реализация безопасности: JWT и авторизация

Одной из важнейших задач при разработке серверной части веб-приложения является обеспечение безопасности — как передаваемых данных, так и ограничение доступа к ресурсам. В рамках дипломного проекта для защиты API была реализована система аутентификации и авторизации на основе JWT (JSON Web Token) — современного и широко применяемого подхода к безопасной передаче информации между участниками взаимодействия.

JWT представляет собой компактный и самодостаточный токен, содержащий в себе всю необходимую информацию о пользователе и его правах. Такой токен формируется на сервере, подписывается секретным ключом, и затем передаётся клиенту. Клиент использует его при каждом последующем запросе, указывая в заголовке авторизации. Сервер, получив токен, может его проверить, извлечь закодированные в нём данные и принять решение об уровне доступа пользователя.

Этапы реализации авторизации на основе JWT:

– генерация токена при логине - при успешной аутентификации (например, когда пользователь вводит правильные логин и пароль), сервер формирует JWT-токен. В него включаются данные, позволяющие идентифицировать пользователя, например, его идентификатор, имя, роль, срок действия токена. Этот токен возвращается клиенту и используется для всех дальнейших запросов.

– настройка middleware для аутентификации - в ASP.NET Core реализована поддержка JWT на уровне встроенного middleware. На этапе настройки приложения подключается схема аутентификации, в которой указывается, как именно обрабатывать входящие токены: какой алгоритм использовать для их валидации, где хранится секретный ключ, как извлекать данные из токена и т. д. Это позволяет централизованно проверять каждый запрос, содержащий токен, без необходимости вручную проверять его в каждом контроллере

– лграничение доступа к API-методам - для защиты конкретных участков API использовались атрибуты [Authorize], которые применяются к контроллерам или их методам. Это означает, что доступ к ним возможен только для аутентифицированных пользователей. Кроме того, при необходимости можно дополнительно указывать роли, например [Authorize(Roles = "Admin")], что позволяет реализовать более гибкую модель управления доступом и разграничение прав.

– хранение и защита секретного ключа - для обеспечения безопасности токена необходим секретный ключ, с помощью которого он подписывается и валидируется. Этот ключ хранится в конфигурационном файле проекта — **appsettings.json**. При запуске приложения ключ считывается и используется для настройки **JWT** middleware. Особое внимание было уделено защите этого ключа: при деплое приложения на сервер важно ограничить к нему доступ и использовать безопасные способы хранения (например, переменные окружения или секретные хранилища).

Рисунок 2.8 Использование и конфигурация секретного jwt-ключа на апи

Преимущества выбранного подхода

Использование JWT обеспечило проекту следующие преимущества:

- Масштабируемость и независимость — токены хранятся на клиенте, не требуется серверной сессии;
- Удобная работа с ролями и правами доступа — информация о правах доступа содержится в самом токене;
- Безопасность передачи данных — токен невозможно подделать без знания секретного ключа;
- Совместимость — стандарт JWT поддерживается всеми современными клиентами, включая веб-браузеры, мобильные приложения и другие серверные системы.

Таким образом, реализация авторизации на основе JWT позволила обеспечить безопасное и эффективное взаимодействие клиента с сервером, а также гибко управлять доступом к различным частям API. Это особенно важно в корпоративной среде, где необходимо чётко разграничивать права пользователей, таких как сотрудники службы поддержки, администраторы и обычные пользователи.

2.1.8 Экспорт в CSV и интеграция с ИИ

В дополнение к основному функционалу API, в проекте была реализована возможность экспорта данных в формате CSV (Comma-Separated Values). Это оказалось особенно полезным для административных задач, аналитики и ведения отчётности. Возможность выгрузки тикетов, пользователей и комментариев в структурированный текстовый формат обеспечивает удобную интеграцию с другими системами и инструментами обработки данных, такими как Excel, Google Sheets или BI-платформы.

Реализация экспорта в CSV

Реализация функционала экспорта данных происходила поэтапно. Первоначально был спроектирован отдельный сервис, отвечающий

исключительно за формирование CSV-отчётов. Такой подход соответствовал принципу единственной ответственности и обеспечил модульность кода, позволяя легко расширять или изменять формат экспорта в будущем.

Основные шаги реализации:

– Создание сервиса экспорта - Отдельный класс, реализующий интерфейс ICsvExportService, был добавлен в слой бизнес-логики. Этот сервис отвечал за выборку данных из базы, преобразование их в строковое представление и формирование итогового CSV-файла.

– Генерация CSV-строки - На этом этапе происходило форматирование данных в соответствии с правилами CSV. Каждая строка представляла собой отдельную сущность (например, тикет или пользователя), а столбцы — поля этой сущности: идентификатор, имя, дата создания, статус, категория и т. д. При необходимости, в формате учитывалась кодировка (например, UTF-8 с BOM), чтобы избежать проблем с отображением в Excel.

– Передача файла клиенту - В контроллере API была реализована конечная точка (endpoint), которая вызывала экспортирующий сервис и возвращала сформированный файл через FileContentResult. Пользователь получал ссылку на скачивание файла с расширением .csv или мог сразу его открыть в браузере.

Такой подход обеспечивает удобную интеграцию экспортируемых данных с аналитическими инструментами, а также позволяет пользователям получать полную информацию по всем заявкам и действиям в системе.

```
Ссылок: 2
public async Task<string> ExportTicketsToCsvAsync()
{
    var trainData = await _context.Tickets
        .Include(t => t.Category)
        .Include(t => t.Cabinet)
        .Where(t => !string.IsNullOrEmpty(t.TicketDescription))
        .Select(t => new TicketCategoryTrainDTO
        {
            InputText = $"Cabinet: {t.Cabinet.CabinetBuilding}-{t.Cabinet.CabinetNumber} | Description: {t.TicketDescr
            CategoryName = t.Category.CategoryName
        })
        .ToListAsync();

    var fileName = $"tickets_category_train_{Guid.NewGuid():N}.csv";
    var filePath = Path.Combine(_exportDir, fileName);

    using var writer = new StreamWriter(filePath, false, Encoding.UTF8);
    using var csv = new CsvWriter(writer, CultureInfo.InvariantCulture);
    await csv.WriteRecordsAsync(trainData);

    return filePath;
}
```

Рисунок 2.9 Создание CSV-файла

Для повышения интеллектуальности системы и автоматизации части рутинных процессов был внедрён отдельный модуль, основанный на предобученной модели BERT, адаптированной под задачи обработки заявок.

Модуль реализован в виде микросервиса на языке Python, с использованием фреймворка FastAPI и библиотеки HuggingFace Transformers.

```python
def tokenize(batch):
    return tokenizer(batch["text"], padding=True, truncation=True)

train_dataset = train_dataset.map(tokenize, batched=True)
test_dataset = test_dataset.map(tokenize, batched=True)

num_labels = len(le.classes_)
model = BertForSequenceClassification.from_pretrained( pretrained_model_name_or_path: "bert-base-uncased", num_labels=num_

training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=tokenizer,
)

trainer.train()

metrics = trainer.evaluate()
print(metrics)
```

Рисунок 2.10 Пример обучения модели BERT

Этот микросервис обеспечивает автоматическую обработку текстов заявок и реализует следующие функции:

1. автоматическая классификация заявок по категориям - на основе анализа текста обращения ИИ-модуль самостоятельно определяет, к какой категории относится заявка: например, "Проблемы с доступом", "Оборудование", "Программное обеспечение" и т.д. Полученный результат передаётся обратно в ASP.NET Core API и используется для автозаполнения поля "Категория" при создании тикета;

2. генерация предложений по формулировке текста - в случае, если введённый текст является недостаточно точным или содержит неоднозначные формулировки, система предлагает отредактированный вариант — краткий и ясный, помогающий ускорить обработку обращения;

3. обнаружение дубликатов - при создании новой заявки ИИ-модуль выполняет сравнение текста с существующими тикетами в базе данных. Если

найдены схожие обращения, пользователю выводится предупреждение о возможном дубликате, с указанием номера и заголовка найденного тикета.

Архитектура взаимодействия между сервисами:

- ASP.NET Core API отправляет текст заявки и (при необходимости) другую метаинформацию на Python-модуль через HTTP POST-запрос;
- Python-сервис обрабатывает данные, возвращает предсказания (категория, вероятность дубликата, отредактированный текст);
- Результаты анализируются серверной частью и используются для автоматического предзаполнения полей или показа рекомендаций клиенту.

Безопасность и производительность

Для защиты межсервисного взаимодействия используются безопасные токены доступа и контроль IP-адресов. Кэширование часто используемых результатов и предобработка данных позволяют добиться высокой скорости отклика даже при большом количестве пользователей.

2.1.9 Результаты и заключение

Разработка серверной логики в рамках дипломного проекта завершилась созданием надёжного, расширяемого и безопасного API, отвечающего современным требованиям к архитектуре корпоративных веб-приложений. Реализация велась на платформе ASP.NET Core Web API, что обеспечило высокую производительность, широкую совместимость с клиентскими приложениями, а также возможность интеграции с внешними сервисами и библиотеками.

В процессе работы были достигнуты следующие ключевые результаты:

1. создана полнофункциональная серверная часть, обеспечивающая весь необходимый цикл обработки заявок (тикетов): от их создания и редактирования до удаления и фильтрации по различным параметрам. Поддерживаются также операции над связанными сущностями — пользователями, комментариями, статусами и категориями;

2. сформирована логически чистая и модульная архитектура, основанная на принципах разделения ответственности и SOLID. Это позволило повысить читаемость кода, упростить его сопровождение и подготовить почву для последующей масштабируемости;

3. реализована многоуровневая структура проекта, включающая отдельные слои:

- контроллеров (взаимодействие с клиентом через HTTP-запросы);
- сервисов (реализация бизнес-логики);
- репозиториев (работа с данными);

– моделей DTO (передача данных между слоями);
– сущностей (отображение структуры таблиц базы данных).

4. обеспечена надёжная работа с базой данных, включая конфигурацию контекста EF Core, настройку миграций и работу с SQL Server. Использование ORM позволило сократить количество шаблонного SQL-кода и повысить стабильность операций с данными;

5. интегрированы механизмы безопасности, включая авторизацию на основе JWT-токенов. Это обеспечивает разграничение прав доступа, защиту чувствительной информации и возможность масштабирования системы безопасности при расширении функционала;

6. внедрена система экспорта в CSV, что обеспечивает выгрузку данных в машиночитаемом виде. Данная функция особенно полезна для анализа, статистики и отчётности, а также для обмена данными между отделами;

7. добавлен интеллектуальный модуль на базе искусственного интеллекта (ИИ). Он обеспечивает автоматическую классификацию тикетов, выявление дубликатов и генерацию текстов. Это повысило степень автоматизации и снизило нагрузку на персонал, обрабатывающий обращения;

8. обеспечена гибкость и масштабируемость API, благодаря использованию REST-подхода, расширяемым маршрутам, стандартным HTTP-глаголам и возможности лёгкого добавления новых эндпоинтов и сущностей. Добавление нового функционала (например, новых фильтров, ролей или логики) не требует существенного изменения существующей структуры кода;

9. проведено модульное тестирование ключевых компонентов, включая сервисы и контроллеры. Это позволило выявить и устранить возможные ошибки на ранних этапах разработки, а также убедиться в корректной работе бизнес-логики;

10. произведена интеграция с клиентским приложением, в результате чего API прошло «боевое» тестирование в интерфейсе конечного пользователя. Работа интерфейса с сервером показала стабильность отклика, надёжную обработку ошибок и правильную реализацию всех предусмотренных сценариев взаимодействия.

Таким образом, разработанная серверная часть полностью соответствует поставленным требованиям и представляет собой устойчивый фундамент для дальнейшего развития информационной системы. Гибкая архитектура и использование современных технологий позволяют безболезненно масштабировать систему, расширять набор функций, интегрироваться с другими платформами и системами аналитики.

Реализация данного API может служить как основой для крупных корпоративных решений, так и как модуль для интеграции в другие цифровые экосистемы. Благодаря строгому соблюдению архитектурных подходов и принципов проектирования, разработка отличается высокой надёжностью,

удобством сопровождения и готовностью к эксплуатации в реальной рабочей среде.

2.2 Создание пользовательского интерфейса

2.2.1 Основы HTML и его роль в клиентской части

HTML — это язык разметки, который служит основой для создания структуры веб-страниц и веб-приложений. Его основная задача — организовать содержимое так, чтобы браузер мог правильно его отобразить, а пользователь — удобно с ним взаимодействовать. В практической разработке HTML задаёт скелет страницы, на который позже накладываются стили и интерактивность с помощью CSS и JavaScript.

Одним из важнейших принципов HTML является логическое структурирование контента. Каждый элемент страницы оборачивается в теги, которые указывают браузеру, что именно это за элемент — заголовок, параграф, список, изображение или форма. Использование правильных тегов помогает не только браузеру, но и разработчикам лучше понимать структуру документа, а также способствует улучшению доступности и SEO-оптимизации сайта.

Современный веб активно использует семантические теги — такие как «заголовок», «раздел», «статья», «нижний колонтитул». Они не только структурируют контент, но и делают страницы более понятными для поисковых систем и вспомогательных технологий, например, экранных читалок для людей с нарушениями зрения. Это особенно важно при создании сайтов, доступных для всех категорий пользователей.

Практическое применение HTML включает работу с мультимедийным контентом — изображениями, видео и аудио. Правильное внедрение мультимедийных элементов помогает сделать сайт более привлекательным и информативным, улучшая пользовательский опыт. При этом важно учитывать особенности браузеров и устройств, а также обеспечивать наличие альтернативного контента на случай, если мультимедийный элемент не загрузится.

Работа с формами — ещё один ключевой аспект практического применения HTML. Формы позволяют собирать данные от пользователей, будь то регистрация, обратная связь, поиск или оформление заказа. Правильное использование элементов формы и их атрибутов обеспечивает удобство ввода и базовую валидацию данных еще на стороне клиента, что повышает качество взаимодействия и уменьшает нагрузку на сервер.

Навигация на сайте строится с помощью гиперссылок — основного способа перемещения между страницами и разделами. Практическая задача

разработчика — сделать меню и ссылки интуитивно понятными и удобными, чтобы пользователь легко находил нужную информацию. При этом важно учитывать такие моменты, как открытие ссылок в новой вкладке и безопасность переходов на внешние ресурсы.

HTML тесно интегрирован с CSS и JavaScript. В то время как HTML отвечает за структуру, CSS отвечает за внешний вид, а JavaScript — за динамику и интерактивность. Понимание того, как эти три технологии взаимодействуют, необходимо для построения современных динамичных приложений, которые не только красиво выглядят, но и быстро реагируют на действия пользователя.

В процессе разработки полезно регулярно проверять корректность и валидность HTML-кода с помощью специальных инструментов. Это помогает избежать ошибок, которые могут приводить к неправильному отображению страниц или ухудшению индексации сайта поисковыми системами.

В целом, HTML — это не просто язык для разметки текста, а фундаментальная технология, на которой строится вся клиентская часть веб-приложений. Знание его основ и умение грамотно использовать возможности HTML является обязательным навыком для любого веб-разработчика.

```html
<div class="container mt-4">
    <h2 id="ticketTitle" class="mb-4">Детали тикета</h2>

    <div id="ticketDetails" class="card p-4 shadow-sm d-none">
        <p><strong>ID тикета:</strong> <span id="ticketId"></span></p>
        <p><strong>Автор:</strong> <span id="ticketAuthorName"></span></p>
        <p><strong>Описание:</strong> <span id="ticketDescription"></span></p>
        <p><strong>Кабинет:</strong> <span id="ticketCabinet"></span></p>
        <p><strong>Категория:</strong> <span id="ticketCategory"></span></p>
        <p><strong>Дата создания:</strong> <span id="ticketCreateDate"></span></p>
        <p><strong>Планируемая дата:</strong> <span id="ticketPlannedDate"></span></p>
        <p><strong>Приоритет:</strong> <span id="ticketPriority"></span></p>
        <p><strong>Время реакции:</strong> <span id="ticketReactionTime"></span> минут</p>
        <p><strong>Выполняющий:</strong> <span id="ticketUserName"></span></p>

        <button id="respondButton" class="btn btn-success mt-3">Откликнуться</button>
        <div id="respondMessage" class="mt-2 text-success d-none">Вы откликнулись на тикет!</div>
    </div>

    <p id="errorMessage" class="text-danger d-none">Ошибка загрузки данных.</p>
</div>
```

Рисунок 2.11 Пример HTML-кода

2.2.2 Организация интерфейса с Razor Pages

Razor Pages — это технология в ASP.NET Core, которая позволяет удобно создавать веб-страницы, объединяя серверную логику и HTML-разметку в одном месте. В отличие от классического MVC, где контроллеры и представления отделены, Razor Pages организуют работу вокруг отдельных страниц, каждая из которых имеет собственный набор файлов: разметку и связанный код.

Структура Razor Pages

В проекте на ASP.NET Core Razor Pages файлы страниц располагаются в папке Pages. Каждая страница представлена двумя основными файлами: файл с расширением .cshtml, содержащий HTML и Razor-синтаксис, и связанный с ним файл .cshtml.cs, где реализована серверная логика страницы на C#.

Например, файл Pages/Index.cshtml — это главная страница сайта, а Pages/_Layout.cshtml — это шаблон (Layout), который определяет общий каркас для всех страниц приложения. Layout включает повторяющиеся элементы, такие как шапка сайта, меню навигации и футер, что позволяет избежать дублирования кода и облегчает поддержку интерфейса.

Использование шаблонов Layout

Шаблоны Layout в Razor Pages служат для организации общего дизайна приложения. Они задают структуру страницы, в которую динамически подставляется содержимое каждой конкретной страницы. Это упрощает создание и изменение общего внешнего вида: если требуется обновить меню или добавить новый элемент в шапку, достаточно сделать это в одном месте — в файле Layout.

Практически каждый .cshtml файл страницы указывает, какой Layout использовать, либо наследует стандартный, определённый в _ViewStart.cshtml. Это обеспечивает консистентность интерфейса на всех страницах и ускоряет разработку.

Razor-синтаксис и динамическое формирование HTML

Одним из ключевых преимуществ Razor Pages является использование Razor-синтаксиса — гибкого способа интеграции C# кода непосредственно в HTML. Благодаря этому можно динамически формировать содержимое страницы на сервере в зависимости от данных модели или бизнес-логики.

Например, с помощью Razor можно циклом вывести список элементов, отобразить данные из базы, условно показать или скрыть части интерфейса, а также внедрять пользовательские параметры. Это делает страницы интерактивными и адаптивными под текущие данные.

Razor синтаксис прост и выразителен: он поддерживает встроенные конструкции, такие как циклы, условия, объявления переменных и вызовы методов, что позволяет реализовать сложную логику прямо в представлении, сохраняя при этом чистоту и читаемость кода.

Практические преимущества Razor Pages

Использование Razor Pages упрощает разработку веб-интерфейсов благодаря близости HTML и C# кода. Это ускоряет разработку, уменьшает количество файлов и связей между ними, а также облегчает отладку. Для небольших и средних приложений данный подход часто оказывается более удобным, чем классический MVC.

Кроме того, Razor Pages хорошо интегрируются с возможностями ASP.NET Core, включая систему маршрутизации, валидацию форм, защиту от CSRF и многое другое, что позволяет создавать надёжные и безопасные приложения.

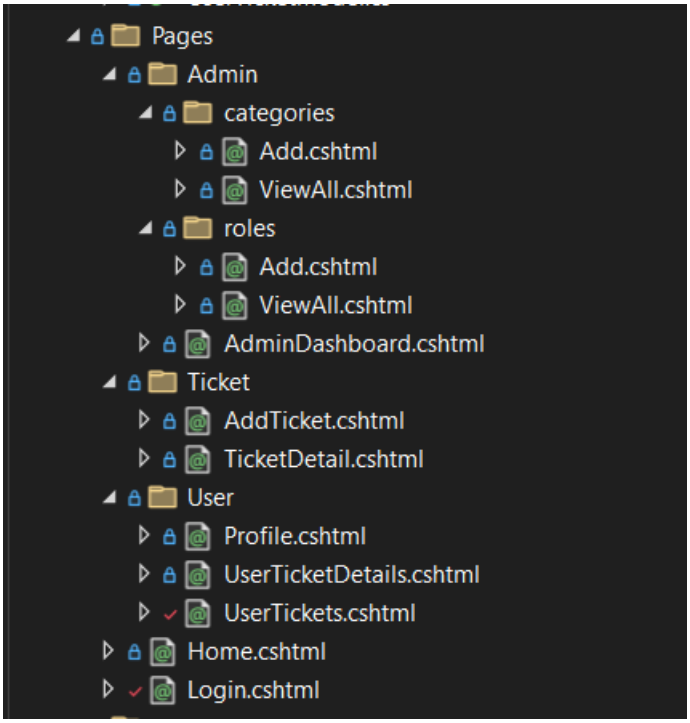


Рисунок 2.12 Папка страниц Page.

2.2.3 Использование Layout-ов и Razor-компонентов

В современных веб-приложениях очень важна единообразная и удобная для пользователя визуальная структура интерфейса. Для достижения консистентности внешнего вида и повторного использования общих элементов интерфейса в ASP.NET Core Razor Pages применяются **Layout** — шаблоны, задающие общий каркас страницы. Использование Layout-ов позволяет централизованно управлять такими элементами, как заголовок сайта, меню навигации, футер и другие части интерфейса, которые должны присутствовать на всех или большинстве страниц приложения.

Создание общего макета (Layout)

Layout — это специальная Razor-страница, обычно располагающаяся в папке Pages/Shared под названием _Layout.cshtml. В этом файле задаётся базовая HTML-структура, в которую динамически подставляется контент каждой конкретной страницы.

В макете обычно включаются:

- заголовок (header) — логотип, название сайта, главное меню навигации;

– основной контент — динамическое содержимое, которое меняется от страницы к странице;

– футер (footer) — информация о копирайте, ссылки на политику конфиденциальности и контакты.

Такой подход позволяет значительно упростить поддержку и развитие проекта — для изменения общего вида достаточно внести правки в один файл Layout, и они автоматически отразятся на всех страницах, использующих этот шаблон.

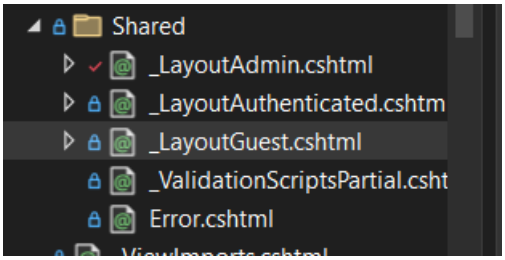


Рисунок 2.13 Папка Layout-ов

Подключение CSS и JavaScript

В файле Layout также подключаются глобальные стили CSS и скрипты JavaScript, необходимые для корректного отображения и работы интерфейса. Это делается с помощью стандартных HTML-тегов <link> для стилей и <script> для скриптов.

Подключение общих файлов стилей и скриптов в Layout обеспечивает:

– единообразие визуального оформления на всех страницах;

– оптимизацию загрузки, так как ресурсы загружаются один раз и доступны всем страницам;

– централизованное управление внешним видом и функционалом.

Рисунок 2.14 Скрипты и Razor-элементы которые будут наследованы в других страницах Razor

Передача параметров и рендеринг вложенных Razor-страниц

Razor Pages поддерживают передачу параметров и использование вложенных компонентов для создания более гибких и модульных интерфейсов.

– рендеринг вложенных страниц и компонентов: С помощью Razor можно вставлять одну страницу или компонент внутрь другой, что помогает строить сложные интерфейсы из простых элементов.

– передача параметров: Вложенные Razor-компоненты могут принимать параметры, что позволяет им динамически изменять своё содержимое или поведение в зависимости от переданных данных.

Этот механизм способствует более чистой архитектуре UI, где каждый элемент отвечает за свою часть интерфейса, упрощая поддержку и масштабирование приложения.

Преимущества модульного подхода в UI-проектировании

Использование Layout-ов и Razor-компонентов поддерживает модульный подход, который имеет следующие преимущества:

– повторное использование кода: Общие части интерфейса не дублируются, что снижает вероятность ошибок и упрощает внесение изменений;

– разделение ответственности: Каждый компонент отвечает за свою конкретную функцию — меню, форму, блок с новостями и т.п.;

– удобство поддержки и расширения: Модульная структура облегчает добавление новых элементов и изменение существующих без влияния на всю систему;

– ускорение разработки: Разработчики могут сосредоточиться на создании отдельных частей, которые затем легко собираются в единое целое.

Таким образом, использование Layout-ов и Razor-компонентов обеспечивает эффективную организацию интерфейса и способствует созданию качественных, удобных и масштабируемых веб-приложений.

2.2.4 Повышение интерактивности с JavaScript

JavaScript играет ключевую роль в создании интерактивных и отзывчивых веб-приложений. В отличие от статичной разметки HTML, JavaScript позволяет изменять содержимое страницы в реальном времени, реагировать на действия пользователя и взаимодействовать с сервером без полной перезагрузки страницы. В контексте Razor Pages, JavaScript служит важным инструментом для улучшения пользовательского опыта.

Добавление обработчиков событий

Обработчики событий — это функции JavaScript, которые запускаются в ответ на действия пользователя, такие как нажатие кнопки, ввод текста в форму, наведение мыши или изменение состояния элементов. Например:

– клик по кнопке может открыть дополнительное меню или форму;
– ввод данных в поле может сразу проверяться на корректность;
– наведение на элемент может показать подсказку или изменить стиль.

Использование событий позволяет делать интерфейс более отзывчивым и удобным, так как пользователь получает мгновенную обратную связь на свои действия.

Работа с DOM (Document Object Model)

DOM — это модель документа, которая представляет HTML-структуру страницы в виде дерева объектов. JavaScript может динамически изменять DOM, добавляя, удаляя или изменяя элементы страницы без необходимости её полной перезагрузки.

Примеры динамических изменений с помощью DOM:

– добавление новых элементов списка или таблицы на основе введённых данных;
– скрытие или отображение блоков информации в зависимости от выбора пользователя;
– обновление содержимого без обновления всей страницы.

Работа с DOM позволяет реализовывать сложные интерфейсы, которые адаптируются под действия пользователя и изменяются в реальном времени.

Интеграция JavaScript в Razor Pages

JavaScript можно подключать к Razor Pages двумя основными способами:

1. встроенные скрипты - JavaScript-код размещается непосредственно внутри Razor-страницы в тегах <script>. Этот способ удобен для небольших сценариев, специфичных для одной страницы;

2. внешние JS-файлы - для более крупного и структурированного кода используют подключение внешних файлов с помощью тега <script src="..."></script>. Такие файлы могут содержать общие функции и библиотеки, используемые на нескольких страницах;

В файле Layout обычно подключают основные скрипты, чтобы обеспечить доступность JavaScript на всех страницах, а специфичные скрипты — на отдельных страницах через встроенный или внешний код.

Практическое применение JavaScript в Razor Pages

- валидация форм на клиенте - JavaScript проверяет правильность введённых данных до отправки на сервер, что экономит время и ресурсы;
- асинхронное обновление данных - с помощью AJAX-запросов JavaScript может получать данные с сервера и обновлять содержимое страницы без её перезагрузки;
- анимации и визуальные эффекты - улучшают восприятие интерфейса, делают работу пользователя комфортнее;
- управление элементами интерфейса - раскрывающиеся меню, вкладки, модальные окна и другие интерактивные элементы реализуются при помощи JavaScript.

Таким образом, интеграция JavaScript с Razor Pages позволяет создавать динамичные и современные веб-приложения с высокой степенью интерактивности и удобства для пользователя. Это неотъемлемая часть современного фронтенд-разработки, которая дополняет статическую HTML-разметку и обеспечивает богатый пользовательский опыт.

2.2.5 Асинхронные запросы с AJAX

AJAX (Asynchronous JavaScript and XML) — это подход в веб-разработке, позволяющий выполнять обмен данными с сервером без полной перезагрузки страницы. Это особенно важно при создании современных интерактивных веб-интерфейсов, где отзывчивость и непрерывность взаимодействия с пользователем играют ключевую роль.

Принципы работы AJAX

Основная идея AJAX заключается в том, чтобы с помощью JavaScript отправлять HTTP-запросы к серверу в фоновом режиме и обрабатывать полученные данные для отображения их на странице без её обновления. Это позволяет реализовывать такие функции, как:

- подгрузка контента без перезагрузки страницы;

– обновление отдельных элементов интерфейса (например, таблицы с данными);

– отправка данных форм с мгновенным отображением результата;

– подтверждение действий пользователя без лишней навигации.

Создание интерактивных форм и таблиц

Один из наиболее частых сценариев использования AJAX — это динамическая работа с формами и таблицами. Например:

– пользователь заполняет форму обратной связи;

– нажимает кнопку отправки;

– AJAX отправляет данные формы на сервер;

– полученный от сервера результат отображается прямо на той же странице — без её перезагрузки.

Также с помощью AJAX можно динамически подгружать строки таблиц, обновлять статистику, выводить сообщения об ошибках валидации и многое другое.

Отправка формы с использованием XMLHttpRequest или jQuery AJAX

AJAX-запросы можно реализовать разными способами:

– с помощью XMLHttpRequest — это встроенный в браузеры интерфейс, позволяющий на низком уровне отправлять асинхронные HTTP-запросы и обрабатывать ответы. Этот метод даёт полный контроль над процессом, но требует больше кода;

– с помощью jQuery AJAX — более простой и лаконичный способ, особенно популярен в проектах, где уже используется jQuery. Он позволяет отправить запрос всего в несколько строк и обрабатывать результат в удобной форме.

Оба подхода позволяют выполнять POST-запросы (например, для отправки формы) или GET-запросы (для получения данных) и обновлять DOM в зависимости от ответа сервера.

Динамическое отображение данных с использованием append()

После получения данных с сервера их можно динамически отобразить на странице с помощью методов, таких как append(). Это особенно удобно при работе с таблицами, списками или галереями, когда нужно подгрузить новые элементы и отобразить их без удаления уже существующих.

Примерные сценарии использования:

– подгрузка новых строк в таблицу при прокрутке вниз;

– добавление сообщений в чат в реальном времени;

– отображение уведомлений или результатов поиска.

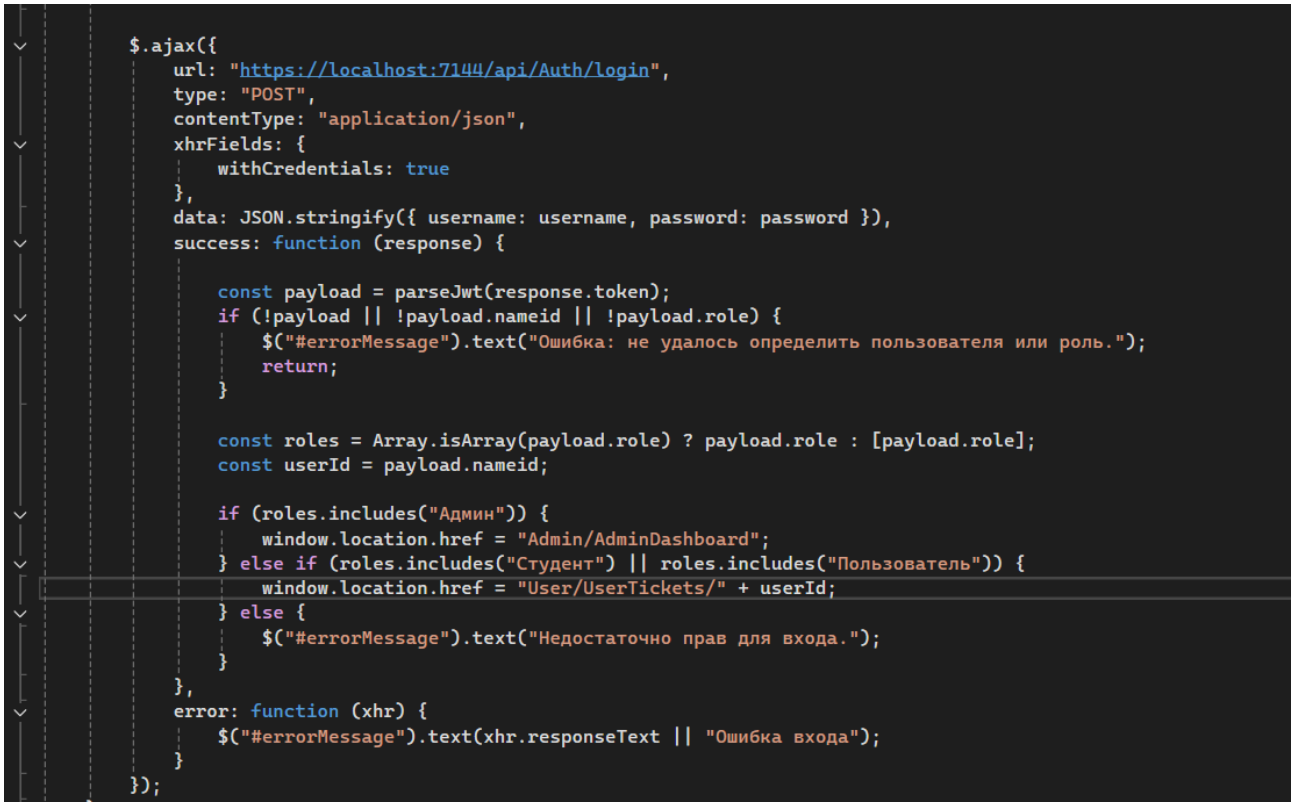
Роль AJAX в Razor Pages

В рамках Razor Pages использование AJAX позволяет сохранить преимущества серверного рендеринга, одновременно обеспечивая интерактивность клиентской части. Страница может рендериться сервером, а

обновление её частей происходит асинхронно, в зависимости от действий пользователя.

Это особенно важно для:

- форм обратной связи;
- административных панелей и дашбордов;
- фильтров и сортировок без перезагрузки страницы;
- постраничной навигации (pagination) и подгрузки данных.



```
$.ajax({
    url: "https://localhost:7144/api/Auth/login",
    type: "POST",
    contentType: "application/json",
    xhrFields: {
        withCredentials: true
    },
    data: JSON.stringify({ username: username, password: password }),
    success: function (response) {

        const payload = parseJwt(response.token);
        if (!payload || !payload.nameid || !payload.role) {
            $("#errorMessage").text("Ошибка: не удалось определить пользователя или роль.");
            return;
        }

        const roles = Array.isArray(payload.role) ? payload.role : [payload.role];
        const userId = payload.nameid;

        if (roles.includes("Админ")) {
            window.location.href = "Admin/AdminDashboard";
        } else if (roles.includes("Студент") || roles.includes("Пользователь")) {
            window.location.href = "User/UserTickets/" + userId;
        } else {
            $("#errorMessage").text("Недостаточно прав для входа.");
        }
    },
    error: function (xhr) {
        $("#errorMessage").text(xhr.responseText || "Ошибка входа");
    }
});
```

Рисунок 2.15 Использование Ajax-запроса

Таким образом, AJAX — это не просто удобная технология, а важный инструмент повышения качества пользовательского интерфейса. Он делает веб-приложения быстрее, гибче и отзывчивее, позволяя реализовать современные стандарты UX/UI.

2.2.6 Работа с Fetch API

С развитием веб-технологий возникла потребность в более современном, гибком и простом инструменте для выполнения асинхронных HTTP-запросов. Таким инструментом стал Fetch API — современный стандарт, заменяющий

устаревший XMLHttpRequest и значительно упрощающий работу с запросами на стороне клиента.

Fetch API является частью спецификации JavaScript и встроен во все современные браузеры. Он обеспечивает простой и лаконичный способ отправки запросов к серверу и получения ответов, в том числе в формате JSON.

Основные преимущества Fetch API

Fetch API предоставляет разработчику ряд преимуществ:

- лаконичный синтаксис: использование fetch() делает код более коротким и понятным;
- работа с промисами (Promises): структура запросов строится на промисах, что упрощает обработку асинхронных операций;
- совместимость с async/await: современные конструкции JavaScript позволяют писать асинхронный код в синхронном стиле;
- гибкость: легко настраивается метод, заголовки, тело запроса и другие параметры;
- расширяемость: Fetch API подходит как для простых GET-запросов, так и для сложных POST/PUT/DELETE-операций с отправкой данных в различных форматах.

Работа с Promises и async/await

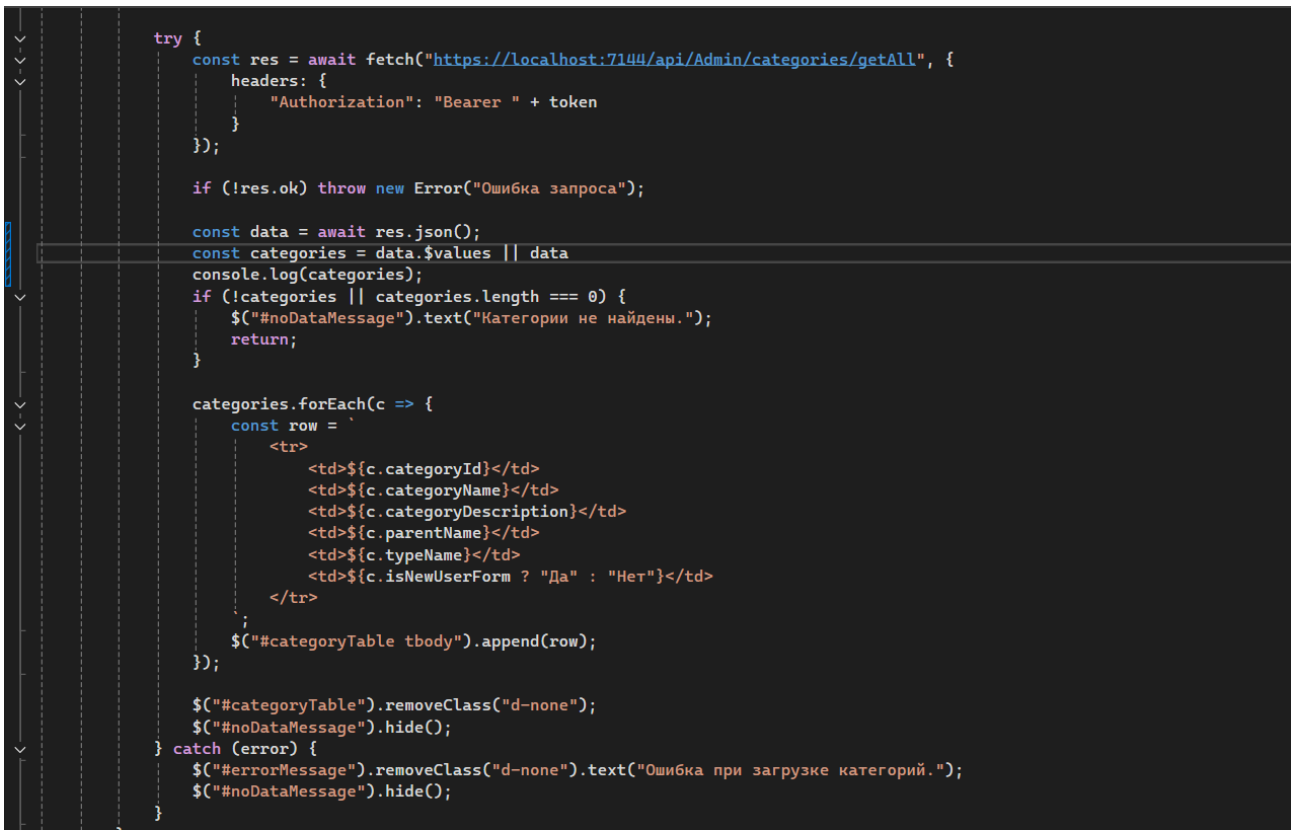
Ключевой особенностью Fetch API является то, что он возвращает промис, который резолвится в объект ответа (Response). Этот объект можно обрабатывать с помощью цепочек .then() или, что более предпочтительно — с использованием конструкции async/await.

Такой подход позволяет избежать «адского колбэка» и сделать асинхронный код визуально линейным и легко читаемым. Например, сначала можно ожидать получения ответа от сервера, затем преобразовать его в JSON, и далее — отобразить полученные данные на странице.

Примеры отправки и получения JSON-данных

Fetch API активно используется при взаимодействии с RESTful API, где данные передаются в формате JSON. С его помощью можно легко реализовать следующие сценарии:

- GET-запрос на получение списка данных (например, заявок или пользователей);
- POST-запрос с передачей тела запроса в формате JSON (например, при отправке формы обратной связи);
- обработка ошибок ответа сервера (например, 400 или 500), что позволяет корректно уведомлять пользователя;
- подгрузка данных по кнопке или при прокрутке страницы (lazy loading).



```
try {
    const res = await fetch("https://localhost:7144/api/Admin/categories/getAll", {
        headers: {
            "Authorization": "Bearer " + token
        }
    });

    if (!res.ok) throw new Error("Ошибка запроса");

    const data = await res.json();
    const categories = data.$values || data
    console.log(categories);
    if (!categories || categories.length === 0) {
        $("#noDataMessage").text("Категории не найдены.");
        return;
    }

    categories.forEach(c => {
        const row = `
            <tr>
                <td>${c.categoryId}</td>
                <td>${c.categoryName}</td>
                <td>${c.categoryDescription}</td>
                <td>${c.parentName}</td>
                <td>${c.typeName}</td>
                <td>${c.isNewUserForm ? "Да" : "Нет"}</td>
            </tr>
        `;
        $("#categoryTable tbody").append(row);
    });

    $("#categoryTable").removeClass("d-none");
    $("#noDataMessage").hide();
} catch (error) {
    $("#errorMessage").removeClass("d-none").text("Ошибка при загрузке категорий.");
    $("#noDataMessage").hide();
}
```

Рисунок 2.16 Использование fetch запроса.

Интерактивность интерфейса через динамическое обновление

После получения ответа с сервера, данные часто нужно отобразить на странице — например, в виде списка, таблицы или карточек. Для этого используется взаимодействие с DOM, включая методы, такие как append(), innerHTML или insertAdjacentHTML().

Таким образом, можно:

- отображать результаты фильтрации или поиска;
- динамически добавлять элементы интерфейса (например, карточки товаров);
- обновлять статус или содержимое без перезагрузки страницы;
- создавать реализацию бесконечной прокрутки (infinite scroll).

Рисунок 2.17 Использование append для динамического добавления контента на странице

Практические преимущества Fetch API в Razor Pages

В контексте Razor Pages Fetch API используется для создания гибких и отзывчивых пользовательских интерфейсов. Например:

- динамическая подгрузка данных в разделе "Мои заявки";
- отправка форм (создание тикета, комментария) без перезагрузки страницы;
- работа с фильтрами и переключателями (статус, категория) в реальном времени;
- автоматическое обновление таблиц, уведомлений, статистики и т.д.

Fetch API отлично интегрируется с архитектурой, основанной на Web API, позволяя фронтенду быстро получать данные и обновлять представление без сложных переходов между страницами или дублирования логики.

2.2.7 Реализация ленивой загрузки данных (Lazy Loading)

Одной из современных практик в разработке производительных веб-интерфейсов является реализация ленивой загрузки данных (Lazy Loading) — динамической подгрузки содержимого по мере необходимости. В отличие от традиционного подхода, при котором все данные загружаются сразу при открытии страницы, ленивый подход позволяет загружать только те данные, которые действительно требуются пользователю в текущий момент, например, при прокрутке списка или таблицы.

Современные веб-приложения часто работают с большими объемами данных — например, списками заявок, пользователей, сообщений, логов или товаров. Загрузка всей информации сразу может:

- существенно замедлить отклик страницы;
- привести к перерасходу памяти и ресурсов браузера;
- вызвать ненужную нагрузку на сервер и базу данных.

Lazy Loading позволяет избежать этих проблем за счёт того, что контент подгружается поэтапно, например, постранично, при прокрутке до конца списка или при клике на кнопку "Загрузить ещё".

Механизм работы ленивой загрузки

1. определение момента подгрузки — с помощью JavaScript отслеживается прокрутка страницы или позиция элемента на экране;
2. отправка запроса к API — как правило, используется fetch() с передачей параметров пагинации (например, page и pageSize);
3. получение и отображение новых данных — новые элементы добавляются на страницу без перезагрузки, например, через append() или innerHTML;
4. обновление состояния — номер страницы, общее количество записей или флаг hasMore сохраняются и обновляются в зависимости от ответа.

```
function loadTickets(page = 1) {
    $("#ticketTable tbody").empty();
    $("#pagination").addClass("d-none");
    $("#noTicketsMessage").text("Загрузка тикетов...").show();
    $("#errorMessage").addClass("d-none");

    $.ajax({
        url: `https://localhost:7144/api/Ticket/getDashboardTicket?page=${page}`,
        type: "GET",
        success: function (response) {
            const tickets = response.tickets?.$values || [];

            if (tickets.length === 0) {
                $("#noTicketsMessage").text("Нет доступных тикетов.");
                return;
            }
```

Рисунок 2.18 Отправление данных для ленивой загрузки

```
function renderPagination(current, total) {
    const pagination = $("#pagination");
    pagination.empty().removeClass("d-none");

    const createPageItem = (page, text, active = false, disabled = false) => `
        <li class="page-item ${active ? 'active' : ''} ${disabled ? 'disabled' : ''}">
            <button class="page-link" ${disabled ? 'disabled' : ''} onclick="loadTickets(${page})">${text}</button>
        </li>`;

    pagination.append(createPageItem(current - 1, "«", false, current === 1));

    for (let i = 1; i <= total; i++) {
        pagination.append(createPageItem(i, i, i === current));
    }

    pagination.append(createPageItem(current + 1, "»", false, current === total));
}

$(document).ready(function () {
    loadTickets(currentPage);
});
```

Рисунок 2.19 Использование полученных данных при ленивой загрузке

Реализация на JavaScript с использованием Fetch

На практике реализация ленивой загрузки сводится к использованию слушателей событий прокрутки (scroll), которые при достижении конца контейнера инициируют новый запрос к API. Для повышения удобства и читаемости обычно используются async/await, а также логика, предотвращающая дублирующие запросы (например, при двойной прокрутке).

Ключевые шаги:

- проверка, достигнут ли низ страницы;
- установка флага «в процессе загрузки»;
- отправка fetch-запроса к серверу;
- добавление новых данных к уже отображённым;
- снятие флага загрузки.

Постраничная выборка на стороне сервера

Для поддержки ленивой загрузки сервер должен предоставлять API с постраничной выборкой данных. Это означает, что при каждом запросе клиент указывает, какие записи он хочет получить — например, с 41 по 60. Такой механизм обычно реализуется через параметры skip и take, page и pageSize или offset и limit.

Сервер возвращает:

- массив элементов;
- общее количество записей (по желанию);
- индикатор hasMore или isLastPage.

Такой API легко расширяется и масштабируется, обеспечивая высокую производительность даже при работе с миллионами записей.

Преимущества ленивой загрузки

- скорость — страница загружается быстрее за счёт меньшего объёма данных;

- экономия трафика и ресурсов — не загружается лишняя информация;
- повышение отзывчивости — пользователь быстрее получает доступ к интерфейсу;
- масштабируемость — подходит для больших списков, не перегружая интерфейс.

Универсальность AJAX и Fetch

Хотя ленивая загрузка реализуется чаще всего через Fetch, также можно использовать XMLHttpRequest или библиотеки типа jQuery.ajax() в более старых проектах. Кроме того, Fetch поддерживает работу не только с JSON, но и с другими форматами — XML, plain text, HTML, что позволяет адаптировать логику под различные потребности.

Интеграция с серверной частью через Web API или Razor Pages позволяет клиенту запрашивать только необходимые данные и отображать их мгновенно, улучшая пользовательский опыт.

Разделение клиентской и серверной логики

Одним из ключевых архитектурных преимуществ применения Fetch и Lazy Loading является чёткое разделение логики:

- сервер занимается только обработкой запросов и возвращением нужных данных;
- клиент отвечает за отображение информации и обработку действий пользователя.

Такой подход упрощает поддержку, тестирование и масштабирование системы. Серверные методы могут быть переиспользованы в других клиентах (например, мобильных приложениях), а фронтенд можно развивать независимо.

2.2.8 Результаты и заключение

В ходе реализации клиентской части веб-приложения была достигнута основная цель — создание современного, динамичного и удобного интерфейса, соответствующего требованиям пользователя и актуальным стандартам веб-разработки. На всех этапах разработки применялись технологии, обеспечивающие интерактивность, производительность и масштабируемость.

Основные достижения:

- фронтенд-приложение построено на Razor Pages — это современный и удобный подход в рамках ASP.NET Core, позволяющий органично объединить серверную и клиентскую части. Razor Pages упрощает организацию структуры представлений, обеспечивая читаемость и поддержку кода;
- HTML, CSS и JavaScript — легли в основу визуального представления и взаимодействия пользователя с системой. Использовались адаптивные

элементы, семантическая разметка и стили, соответствующие принципам UX/UI-дизайна;

- интеграция JavaScript обеспечила возможность работы с DOM, обработки событий, модификации элементов интерфейса в реальном времени;
- асинхронные запросы через AJAX и Fetch API позволили добиться высокой отзывчивости интерфейса, избежать полной перезагрузки страниц, а также реализовать функционал динамической подгрузки данных и обновления информации;
- поддержка ленивой загрузки (Lazy Loading) и фильтрации — улучшила производительность, позволив эффективно обрабатывать большие объёмы данных, отображая только те фрагменты, которые действительно необходимы пользователю в конкретный момент;
- взаимодействие с серверным API через стандартизированные запросы (JSON, метод GET, POST, пагинация) обеспечило устойчивую и масштабируемую коммуникацию между фронтендом и бекендом.

Характеристики разработанного интерфейса:

- отзывчивый и адаптивный — интерфейс корректно отображается на различных устройствах и разрешениях экрана;
- интерактивный — элементы интерфейса реагируют на действия пользователя мгновенно, что повышает удобство и снижает время отклика;
- расширяемый — архитектура и структура страниц позволяют без труда внедрять новый функционал или перерабатывать существующий;
- пользовательски ориентированный — особое внимание уделено эргономике, простоте навигации, логике взаимодействия и визуальной доступности информации.

Современные подходы к клиентской разработке

В проекте были реализованы ключевые принципы современного веб-развития:

- разделение логики клиента и сервера;
- минимизация количества запросов к серверу;
- использование событийно-ориентированной модели взаимодействия с DOM;
- применение асинхронного программирования (async/await) для повышения читаемости кода;
- гибкая архитектура данных, позволяющая использовать API и клиентский рендеринг независимо от конкретного интерфейса;
- прогрессивное улучшение — интерфейс работает даже без JavaScript (в базовом режиме), но с JS становится значительно функциональнее.

Использование технологий AJAX и Fetch API стало ключевым шагом в реализации интуитивно понятного и высокопроизводительного веб-приложения. Эти инструменты позволяют эффективно обрабатывать данные,

обновлять содержимое страниц в реальном времени, реагировать на действия пользователя без задержек и сохранять целостность пользовательского опыта.

Внедрение асинхронных запросов и динамических компонентов интерфейса существенно повышает конкурентоспособность приложения, делает его более удобным, быстрым и технологичным. Такой подход соответствует современным требованиям к разработке, способствует повышению производительности командной работы и обеспечивает основу для масштабируемости проекта в будущем.

Таким образом, клиентская часть разработанного веб-приложения полностью удовлетворяет функциональным, визуальным и технологическим критериям, предъявляемым к современным информационным системам.

3. Безопасность и производительность системы

3.1 Процесс разработки

Разработка информационной системы велась на платформе ASP.NET Core с использованием Web API для серверной части и Razor Pages для клиентского интерфейса. Целью проекта было создание системы управления заявками (тикетами) для малого бизнеса с возможностью масштабирования и дальнейшего развития.

На начальном этапе был проведён сбор требований: выделены ключевые роли пользователей (администратор, оператор, сотрудник), определены сценарии взаимодействия с системой, а также бизнес-правила обработки заявок.

Архитектура системы построена по принципу разделения ответственности: бизнес-логика реализована в сервисах, контроллеры обрабатывают HTTP-запросы, а доступ к данным организован через Entity Framework Core. Также внедрён шаблон Repository и применены механизмы Dependency Injection.

Для взаимодействия с клиентской частью использованы Razor Pages, HTML, CSS, JavaScript, а также технологии AJAX и Fetch для отправки асинхронных запросов к API без перезагрузки страницы. Это обеспечило плавную и интерактивную работу интерфейса.

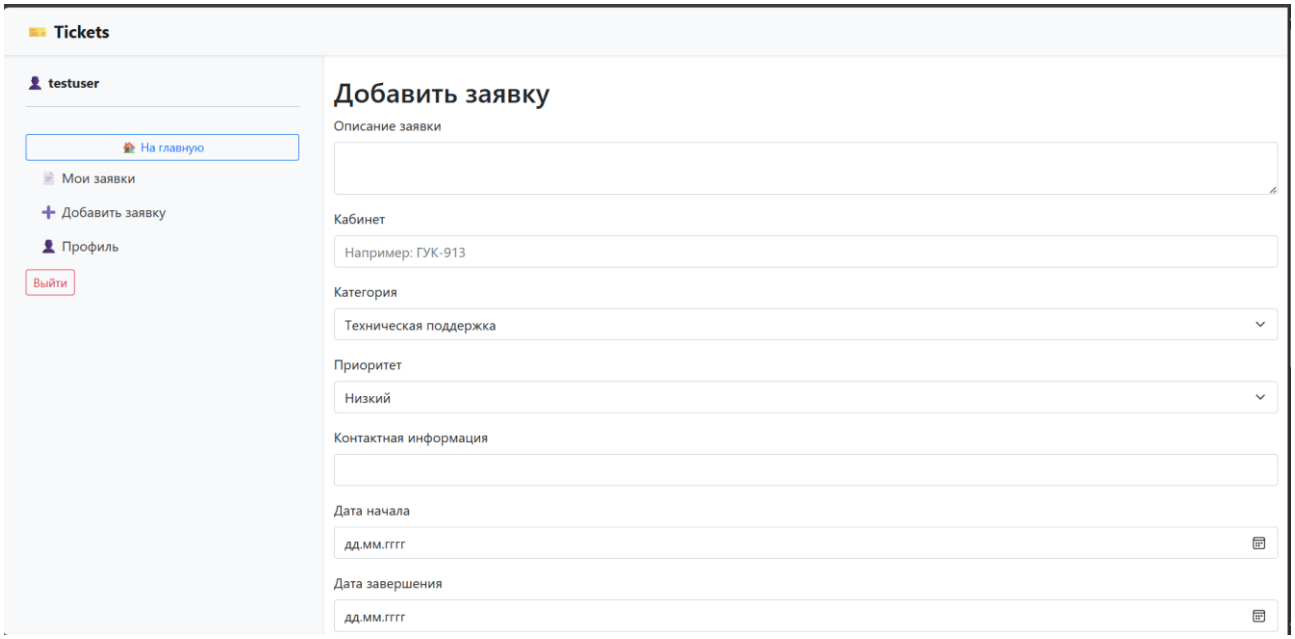


Рисунок 3.1 – Страница создания заявки (Razor Pages + Fetch)

3.2 Тестирование информационной системы

В процессе разработки большое внимание уделялось проверке корректности работы системы. Тестирование проводилось как вручную, так и с помощью автоматических средств.

Также выполнялось интеграционное тестирование API через Swagger. Была подключена библиотека Swashbuckle, которая автоматически сгенерировала документацию и предоставила удобный веб-интерфейс для проверки всех эндпоинтов. Это позволило оперативно тестировать работу API в реальных условиях.

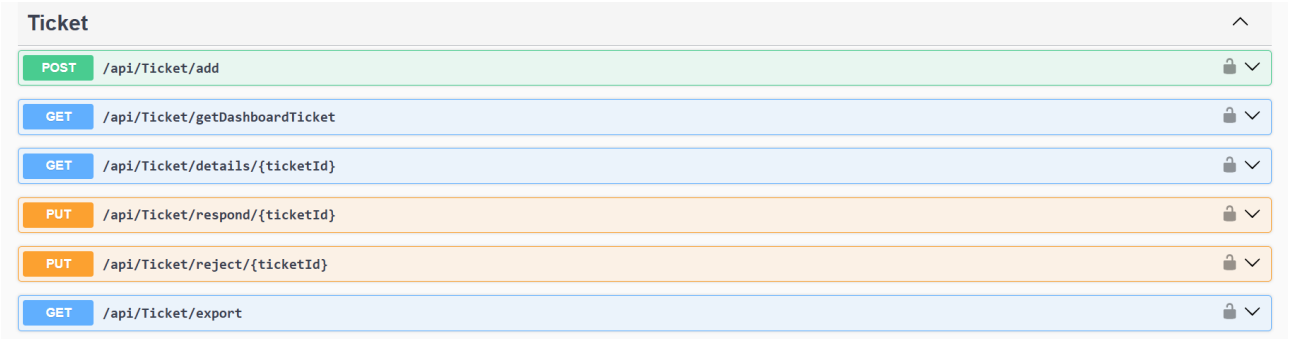


Рисунок 3.2 – Swagger UI: тестирование POST /api/tickets

В клиентской части вручную тестировались формы создания и просмотра тикетов, переключение фильтров, отправка AJAX-запросов и обработка ошибок. В приложении реализована централизованная обработка ошибок с использованием конструкции try-catch, благодаря чему исключения логируются и не приводят к сбоям в работе интерфейса.

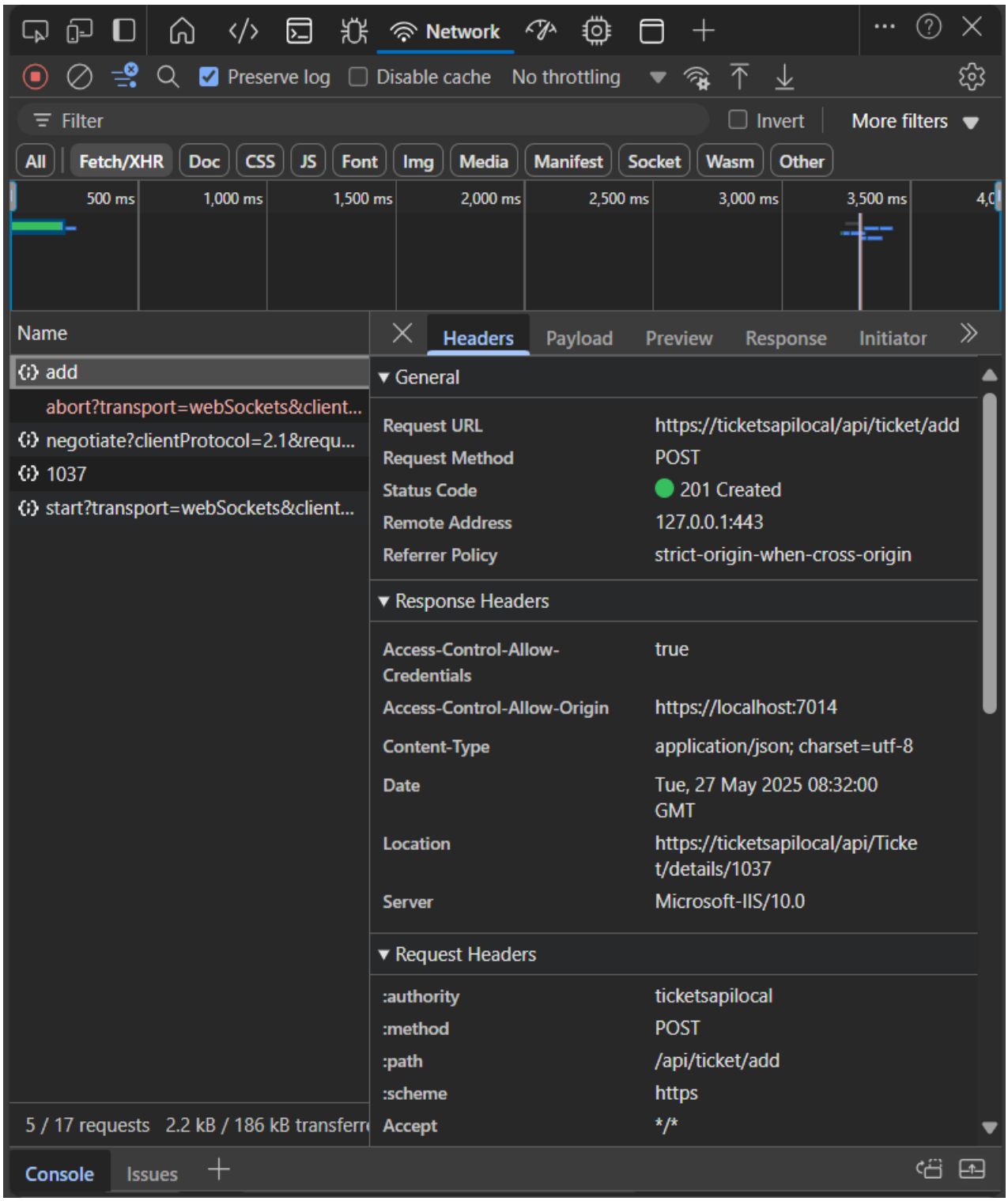


Рисунок 3.3 – Форма создания тикета: отправка AJAX-запроса

Логирование всех событий и ошибок организовано с помощью библиотеки Serilog. Для визуализации логов использовалась система Seq, которая предоставляет удобный интерфейс поиска, фильтрации и анализа журналов. Это облегчает диагностику и выявление проблем в рабочем приложении.

Рисунок 3.4 – Интерфейс Seq: отображение логов приложения Tickets

3.3 Развертывание информационной системы

После завершения этапа разработки система была подготовлена к развёртыванию. Серверная часть развернута на базе IIS. Настроены конфигурации окружений и подключение к SQL Server. Выполнена миграция базы данных, созданы справочники, добавлены роли и тестовые аккаунты.

Клиентская часть была опубликована совместно с серверной на одном хостинге. Весь трафик проходит по защищённому HTTPS-протоколу.

После развёртывания выполнено финальное тестирование доступности системы, проверка авторизации, операций с тикетами и стабильности работы API.

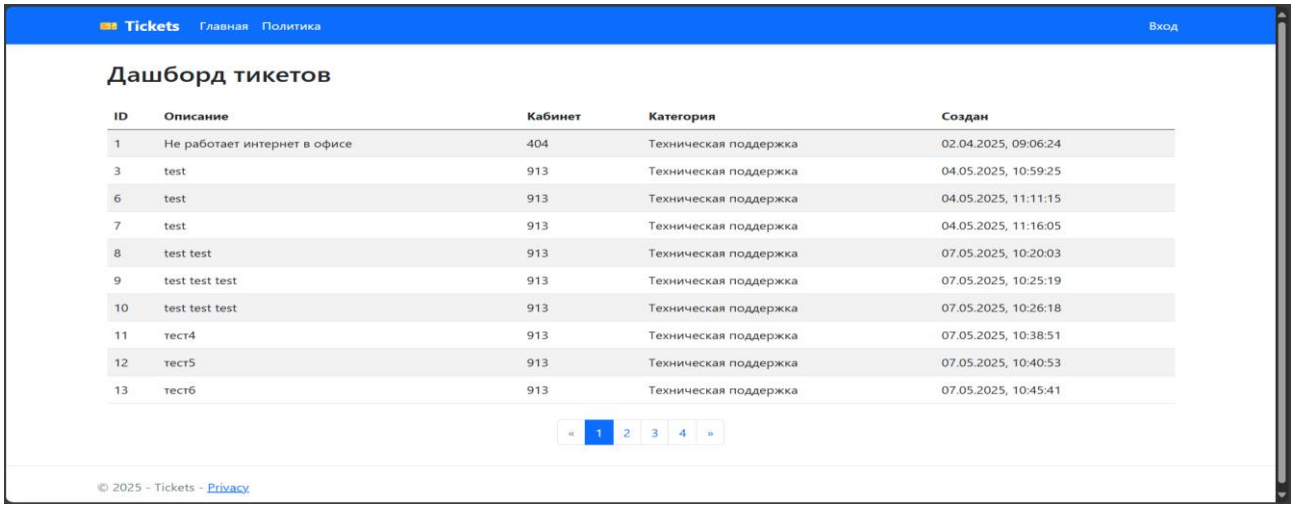


Рисунок 3.5 – Главная страница после развертывания

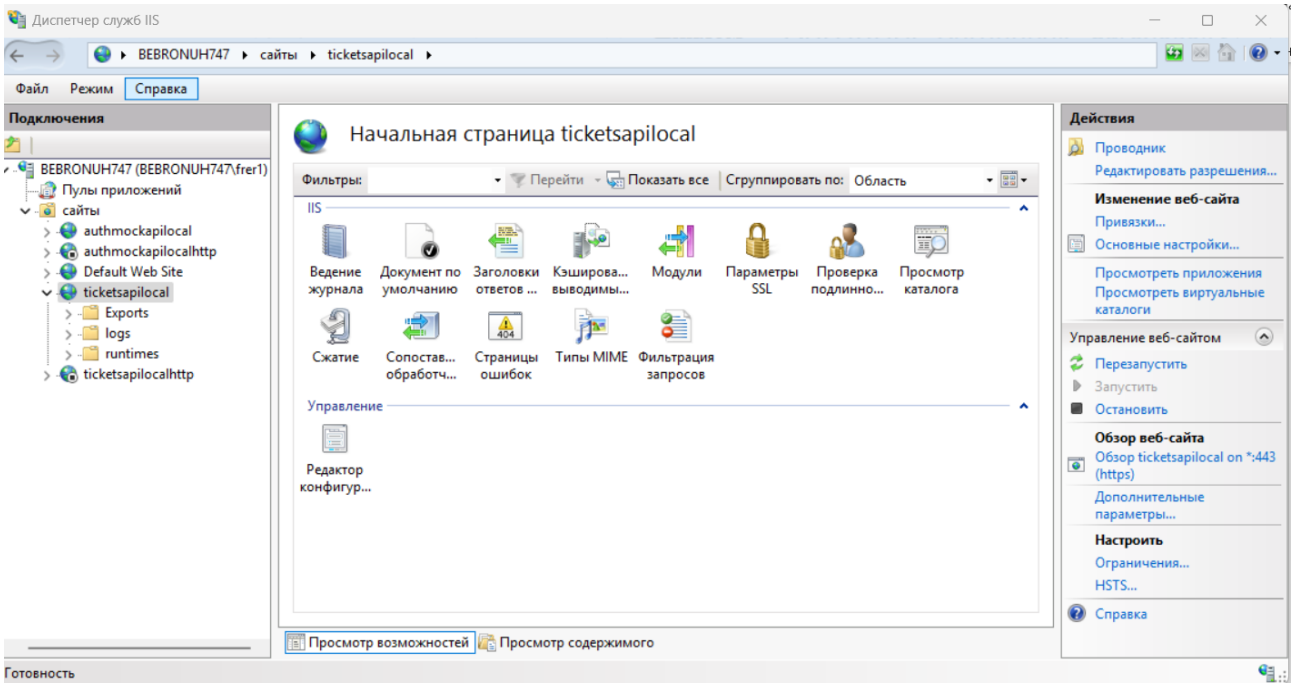


Рисунок 3.6 – IIS: конфигурация сайта Tickets

Таким образом, в результате выполнения всех этапов разработки, тестирования и развертывания была создана устойчивая, безопасная и расширяемая информационная система, готовая к эксплуатации и масштабированию в рамках малого бизнеса.

ЗАКЛЮЧЕНИЕ

В рамках данного дипломного проекта была разработана и внедрена информационная система управления внутренними заявками (тикетами), направленная на оптимизацию и автоматизацию внутренних бизнес-процессов компании. Актуальность проекта обусловлена необходимостью повышения прозрачности, контроля и эффективности работы с заявками в условиях современной корпоративной среды, где быстрое и качественное решение задач имеет критическое значение.

Разработка выполнена с использованием современных технологий, таких как ASP.NET Core Web API, Razor Pages, Entity Framework и Microsoft SQL Server, что обеспечило высокую производительность, надежность и масштабируемость системы. Выбранная архитектура клиент-сервер позволяет гибко развивать и адаптировать приложение под изменяющиеся требования организации.

Реализованная система успешно решает основные задачи:

- централизует приём, обработку и хранение внутренних заявок;
- обеспечивает удобный и прозрачный контроль статусов и исполнения задач;
- снижает вероятность ошибок за счёт автоматизации процессов и чёткой регламентации;
- повышает дисциплину и качество взаимодействия между сотрудниками и подразделениями.

Проведённое функциональное и нагрузочное тестирование подтвердило стабильную работу системы при реальной эксплуатации, её устойчивость к многопользовательским нагрузкам и удобство интерфейса для конечных пользователей.

Особое внимание уделено перспективам дальнейшего развития проекта. Благодаря модульной архитектуре система обладает хорошим потенциалом для масштабирования и интеграции с корпоративными сервисами, такими как ERP и CRM. В числе возможных направлений развития:

- создание мобильной версии для удобного доступа к тикетам в любое время и в любом месте;
- интеграция с существующими корпоративными системами для объединения процессов управления;
- внедрение аналитических инструментов и визуализации ключевых показателей эффективности (KPI), сроков исполнения и загруженности сотрудников;
- добавление расширенных функций приоритизации, SLA и автоматических уведомлений для улучшения качества сервиса.

Таким образом, разработанная информационная система полностью соответствует текущим требованиям компании, обеспечивает значительное улучшение управления внутренними заявками и создает прочную основу для дальнейшей цифровой трансформации бизнес-процессов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Troelsen A., Japikse P. *Pro C# 10 with .NET 6: Foundational Principles and Practices in Programming*. 11th Edition. Apress, 2022.
2. Sanderson S. *Pro ASP.NET Core MVC 2*. Apress, 2017.
3. Freeman E., Robson E. *Head First Design Patterns: A Brain-Friendly Guide*. 2nd Edition. O'Reilly Media, 2020.
4. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
5. Martin R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
6. Albahari J., Albahari B. *C# 10 in a Nutshell: The Definitive Reference*. O'Reilly Media, 2022.
7. Seemann M. *Dependency Injection in .NET*. Manning Publications, 2011.
8. Thomas D., Hunt A. *The Pragmatic Programmer: Your Journey to Mastery*. 20th Anniversary Edition. Addison-Wesley, 2019.
9. McConnell S. *Code Complete: A Practical Handbook of Software Construction*. 2nd Edition. Microsoft Press, 2004.
10. Fowler M. *Refactoring: Improving the Design of Existing Code*. 2nd Edition. Addison-Wesley, 2018.
11. ASP.NET Core: Обзор и руководство по созданию веб-приложений – **https://learn.microsoft.com/en-us/aspnet/core/**
12. Razor Pages в ASP.NET Core: создание страниц и обработка событий – **https://learn.microsoft.com/en-us/aspnet/core/razor-pages/**
13. Entity Framework Core: ORM для .NET – **https://learn.microsoft.com/en-us/ef/core/**
14. C# Programming Guide: синтаксис, типы, ООП и современные возможности – **https://learn.microsoft.com/en-us/dotnet/csharp/**
15. LINQ Guide: запросы к данным в стиле C# – **https://learn.microsoft.com/en-us/dotnet/standard/linq/**
16. ASP.NET Core Identity: аутентификация и авторизация пользователей – **https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity**
17. Конфигурация и управление параметрами в ASP.NET Core – **https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/**
18. SQL Server Documentation: работа с Microsoft SQL Server – **https://learn.microsoft.com/en-us/sql/sql-server/**
19. ADO.NET Overview: доступ к данным в .NET – **https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview**
20. GitHub Actions: автоматизация CI/CD процессов – **https://docs.github.com/en/actions**

21. Git: официальная документация по распределённой системе контроля версий – **https://git-scm.com/doc**

22. React: руководство по созданию компонентов и интерфейсов – **https://reactjs.org/docs/getting-started.html**

23. JavaScript Guide: основы и особенности языка JS от Microsoft – **https://learn.microsoft.com/en-us/scripting/javascript/**

24. Принципы проектирования ПО: DRY, KISS, YAGNI, SOLID – **https://dev.to/** **и** **https://medium.com/**

25. Visual Studio: среда разработки для .NET, C#, Razor и JavaScript – **https://learn.microsoft.com/en-us/visualstudio/**

Приложение А

Техническое задание на разработку информационной системы управления внутренними заявками.

Проект направлен на создание веб-приложения для централизованного управления внутренними заявками (тикетами) внутри компании. Цель — повысить эффективность работы, обеспечить прозрачность процессов и контроль исполнения задач.

Система позволит сотрудникам подавать заявки, отслеживать их статусы, назначать ответственных и комментировать заявки. Также предусмотрена возможность формирования отчетов по выполненным задачам с фильтрацией по времени, статусу, отделу и другим параметрам.

Архитектура решения — клиент-серверная. Backend реализуется с использованием ASP.NET Core Web API, ORM Entity Framework и базы данных Microsoft SQL Server. Frontend создаётся на Razor Pages с применением Bootstrap для адаптивного дизайна, что обеспечивает корректное отображение на различных устройствах и в современных браузерах (Chrome, Firefox, Edge и др.).

Для безопасности предусмотрена регистрация и авторизация пользователей с подтверждением учетных данных (email и пароль). Аутентификация построена на JWT (JSON Web Token). Реализовано разграничение прав доступа — роли администратор и пользователь, с разным уровнем доступа к функциям и данным.

Основной функционал включает создание заявок с указанием заголовка, описания, категории и приоритета, ведение комментариев, смену статусов ("Ожидает", "В процессе", "Завершено", "Отложено" и др.), назначение ответственных, а также уведомления по изменениям статусов.

Система должна обеспечивать производительность — поддержка до 100 одновременных пользователей без заметных задержек. Для повышения надёжности предусмотрено устойчивое хранение данных и отказоустойчивость.

План разработки рассчитан на 15 недель и включает следующие этапы:

- Сбор требований и проектирование — 2 недели;
- Разработка backend (API, база данных) — 4 недели;
- Разработка frontend (пользовательский интерфейс) — 4 недели;
- Интеграционное тестирование, функциональные и нагрузочные тесты — 3 недели;
- Внедрение и обучение пользователей — 2 недели.

Результатом станет полнофункциональное веб-приложение для обработки внутренних заявок с возможностью формирования отчётов и полной документацией по тестированию. Обучение сотрудников позволит эффективно использовать систему.

Требования к нефункциональным характеристикам включают обеспечение безопасности данных, шифрование паролей, масштабируемость и поддержку интеграции с другими системами (ERP, CRM). Система должна быть готова к дальнейшему развитию и расширению функционала.

Приложение Б

код программы

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System.Security.Claims;
using TicketsApi.DTO;
using TicketsApi.Models;
using TicketsApi.Services;

namespace TicketsApi.Controllers
{
   [Route("api/[controller]")]
   [ApiController]
   public class TicketController : ControllerBase
   {
      private readonly ITicketService _ticketService;
      public TicketController(ITicketService ticketService)
      {
         _ticketService = ticketService;
      }

      [Authorize]
      [HttpPost("add")]
      public async Task<IActionResult> AddTicket(CreateTicketDTO ticketDto)
      {
         var userIdClaim = User.FindFirst(ClaimTypes.NameIdentifier)?.Value;
         if (string.IsNullOrEmpty(userIdClaim))
            return Unauthorized("Пользователь не авторизован.");

         int authorId = int.Parse(userIdClaim);

         int createdTicketId = await _ticketService.CreateTicketAsync(ticketDto,
authorId);

         return CreatedAtAction(nameof(GetTicketById), new { ticketId =
createdTicketId }, new { createdTicketId });
      }

      [HttpGet("getDashboardTicket")]
      public async Task<IActionResult> GetTicketsByPageDashboard(int page = 1)
      {
```

Продолжение приложения Б

```
        const int pageSize = 10;

        var totalCount = await _ticketService.GetTicketsCountAsync();

        var totalPages = (int)Math.Ceiling(totalCount / (double)pageSize);

        var tickets = await _ticketService.GetTicketsDashboardByPageAsync(page, pageSize);

        var response = new TicketDashboardResponseDTO
        {
            Tickets = tickets,
            CurrentPage = page,
            TotalPages = totalPages,
            PageSize = pageSize,
            TotalCount = totalCount
        };

        return Ok(response);
    }

    [Authorize]
    [HttpGet("details/{ticketId}")]
    public async Task<IActionResult> GetTicketById(int ticketId)
    {
        var ticket = await _ticketService.GetTicketByIdAsync(ticketId);
        if (ticket == null)
        {
            return NotFound($"Тикет с ID {ticketId} не найден.");
        }
        return Ok(ticket);
    }

    [Authorize]
    [HttpPut("respond/{ticketId}")]
    public async Task<IActionResult> RespondToTicket(int ticketId)
    {
        var userId = int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);

        var ticket = await _ticketService.GetTicketByIdAsync(ticketId);
        if (ticket == null)
            return NotFound(new { message = "Тикет не найден" });
```

```
        if (ticket.UserId != null)
            return BadRequest(new { message = "На тикет уже назначен исполнитель" });

        if (ticket.UserId == userId)
            return BadRequest(new { message = "Вы не можете быть исполнителем своего тикета" });

        var updatedTicket = await _ticketService.UpdateTicketUserAsync(ticketId, userId);
        return Ok(updatedTicket);
    }

    [Authorize]
    [HttpPut("reject/{ticketId}")]
    public async Task<IActionResult> RejectTicket(int ticketId)
    {
        var userId = int.Parse(User.FindFirst(ClaimTypes.NameIdentifier)?.Value!);

        var ticket = await _ticketService.GetTicketByIdAsync(ticketId);
        if (ticket == null)
            return NotFound(new { message = "Тикет не найден" });

        await _ticketService.RemoveUserFromTicketAsync(ticketId);

        return Ok();
    }

  }
}
```