

РЕФЕРАТ

В состав проектной работы входят: пояснительная записка 63 страницы, 77 рисунков, 1 таблица, 13 источников, 5 приложений.

ASP.NET CORE, SQL SERVER, WEB-API, MVC, БАЗА ДАННЫХ, СЕРВЕР

Целью данного проекта является разработка серверной части web-сервиса по обучению программированию «Programming-training service» по типу SPA с элементами МРА и предназначенного для

Также было проведено исследование систем-аналогов, для определения недостатков и достоинств существующих решений.

Для разработки системы была использована среды разработки MS Visual Studio 2019, MS SQL Server Management Studio, база данных SQL Server, язык программирования C#, и платформа ASP.NET Core. Для управления версиями приложения использовалась система Git.

СОДЕРЖАНИЕ

СОДЕРЖАНИЕ	2
ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ	3
ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	4
ВВЕДЕНИЕ.....	5
1 Постановка задачи	6
1.1 Характеристика объекта разработки.....	6
1.2 Обзор аналогов	6
2 Анализ поставленной задачи.....	8
2.1 Требования к разрабатываемой системе.....	8
2.2 Выбор средств разработки	9
2.3 Выбор вариантов реализации требований	11
3 Описание результатов разработки	12
3.1 Проектирование таблиц базы данных.....	12
3.2 Создание моделей сущностей и классов для взаимодействия с БД	15
3.3 Организация работы SPA части сервиса	19
3.4 Организация работы МРА части сервиса	27
3.5 Тестирование	39
3.6 Итоговый продукт	46
ЗАКЛЮЧЕНИЕ	50
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	51
ПРИЛОЖЕНИЯ.....	52

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей пояснительной записке к ВКР применяют следующие термины с соответствующими определениями:

1. База данных - именованная совокупность данных, отражающая состояние объектов и их отношений в рассматриваемой предметной области
2. Веб-сервис - идентифицируемая уникальным веб-адресом (URL-адресом) программная система со стандартизированными интерфейсами, а также HTML-документ сайта, отображаемый браузером пользователя.
3. ASP.NET Core - свободно-распространяемый кросс-платформенный фреймворк для создания веб-приложений с открытым исходным кодом.
4. C# - язык программирования, сочетающий объектно-ориентированные и контекстно-ориентированные концепции
5. Entity Framework Core - объектно-ориентированная, легковесная и расширяемая технология от компании Microsoft для доступа к данным.
6. MS SQL Server - система управления реляционными базами данных (РСУБД), разработанная корпорацией Microsoft.
7. Git - это распределенная система контроля версий и управления исходным кодом
8. Back-end/сервер/бэкенд - программно-аппаратная часть сервиса.
9. Front-end/фронтенд/клиент - клиентская сторона приложения т.е. та, с которой пользователь может взаимодействовать и контактировать напрямую.
10. MVC (Model-View-Controller) – Модель-Представление-Контроллер. Фундаментальный паттерн проектирования приложений, который позволяет отделить графический интерфейс от бизнес логики, а бизнес логику от данных.
11. Файлы Cookie - используются веб-серверами для идентификации пользователей и хранения данных о них.
12. Tag-хелперы - функциональность для генерации HTML-разметки в ASP.NET Core
13. Валидация – процесс проверки данных приходящих от пользователя, на соответствие всем требованиям и ограничениям, установленным для этих данных
14. Прoxy – сервер – промежуточный сервер, выполняющий роль посредника между пользователем и целевым сервером.
15. OAuth – схема авторизации, которая позволяет предоставить третьей стороне ограниченный доступ к защищённым ресурсам пользователя без необходимости раскрывать третьей стороне логин и пароль.
16. Класс-репозиторий – специальный класс для доступа к данным определенного типа

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящей пояснительной записке к ВКР применяют следующие сокращения и обозначения:

1. IT - Information Technology (Информационные технологии)
2. DI - Dependency injection (внедрение зависимостей)
3. ORM - object-relational mapping - отображения данных на реальные объекты
4. SPA - single page application (одностраничное приложение)
5. MPA (Multi Page Application) – многостраничные приложения
6. БД – база данных
7. CORS (Cross Origin Resource Sharing) - Совместное Использование Ресурсов Разных Источников (Кроссдоменные запросы)
8. ER - Entity-Relationship model (модель «сущность — связь»)

ВВЕДЕНИЕ

Цель работы

В настоящее время онлайн обучение приобретает всё большую популярность и IT сфера не исключение. В интернете можно найти множество онлайн курсов и сервисов по изучению языков программирования и различных технологий. Тем не менее опыт каждого программиста уникален, поэтому я, совместно с моим товарищем, решили создать веб-сервис с информацией, которой нам не хватало на этапе нашего обучения и изучения различных областей IT. Он будет включать как общие сведения и уроки по разным языкам программирования, так и разбор авторских проектов с описанием возникших проблем и их решением.

Сервис призван помочь начинающим программистам улучшить знания, а также найти способы и идеи для реализации их проектов. Кроме того, сервис позволит нам лучше изучить веб-разработку и различные технологии, связанные с этой областью, а также имеет реальную перспективу развития в будущем, поскольку со временем мы сможем дополнять его всё более качественным материалом.

Цель данной работы заключается в анализе предметной области, определение требований к системе, изучении средств и подходов к разработке Web API веб-приложений, выбор среды и технологий разработки, языка программирования, СУБД и других вспомогательных технологий, а также проектировании базы данных и связей между таблицами. Кроме того, необходимо начать программирование логики создаваемого приложения.

1 Постановка задачи

1.1 Характеристика объекта разработки

На основании собственного опыта и видения готового продукта были определены следующие функциональные возможности, которые должен предоставлять веб-сервис:

- Наличие следующих разделов: теория/тесты/заметки/статьи/авторы.
- В разделе теория пользователи могут выбрать язык программирования и изучить интересующую их тему, посмотреть примеры и пройти небольшой опрос в конце.
- В разделе тесты собраны тесты и задачи на различные темы, связанные с разработкой и программированием, решая которые пользователь может проверить свои знания. Зарегистрированный пользователь может как добавлять новые колоды, так и удалять их.
- В разделе статьи собраны авторские тексты с советами по применению и способами реализации различных технологий и решения проблем при разработке в разных областях программирования.
- У пользователей есть возможность зарегистрироваться в системе
- У авторизованных пользователей должна быть возможность создавать заметки по мере изучения материалов.
- Для авторизованных пользователей сервис сохраняет информацию о пройденных уроках и результатах тестов.

1.2 Обзор аналогов

Функционал обучающих веб-сервисов по программированию

Большинство сайтов и сервисов по обучению программированию можно условно разделить на 2 категории – платные и бесплатные. Платные сервисы, как правило, представляют собой не столько сайты, сколько масштабные веб-платформы. Над их разработкой и поддержкой трудятся десятки разработчиков разных уровней и специализаций. К таким сервисам относятся, например, praktikum.yandex.ru, skillbox.ru, geekbrains.ru и другие. Обычно, подобные сервисы содержат множество курсов на различные тематики, а процесс обучения проходит с онлайн лекциями, домашними заданиями и преподавателями. Реализовать подобный функционал в рамках этой работы не представляется возможным, поэтому основой для создания нашего сервиса выступили бесплатные веб-ресурсы, вот некоторые из них:

- Metanit.com (см. рисунок 1.1) – сайт для тех, кто хочет изучить программирование с нуля. Содержит руководства по множеству популярных

технологий, а также учебников по языкам программирования. Из преимуществ можно отметить высокое качество материала, наличие поясняющих видео к некоторым урокам, довольно простой и лаконичный дизайн – на сайте отсутствуют анимации, яркие цвета и другие элементы дизайна, вся информация представлена сжато и по сути. Из минусов – отсутствие проверочных тестов и других средств самопроверки.

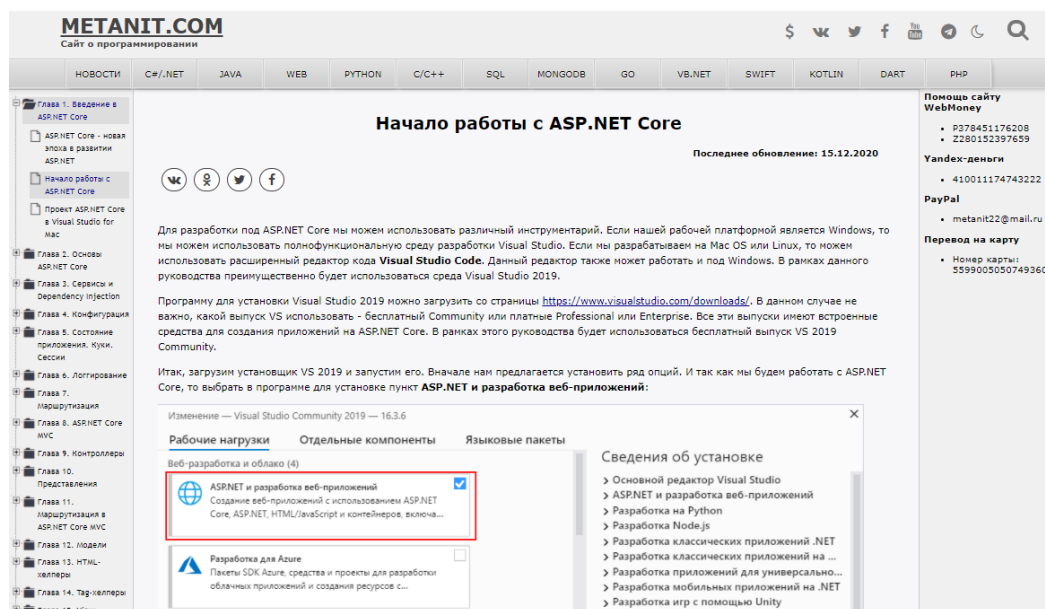


Рисунок 1.1 - Внешний вид статьи на сайте Metanit.com

- habr.com (см. рисунок 1.2) – крупный интернет блог для коллективного создания контента и его обсуждения. Контент сайта формируется пользователями-добровольцами, которые пишут коллективные и персональные блоги, публикуют и переводят иностранные статьи, проводят опросы и общаются с другими пользователями. На данном ресурсе можно найти ответы на многие вопросы, связанные с разработкой в различных отраслях и обсудить их другими пользователями. Из минусов можно отметить сомнительное качество некоторых материалов и публикаций.

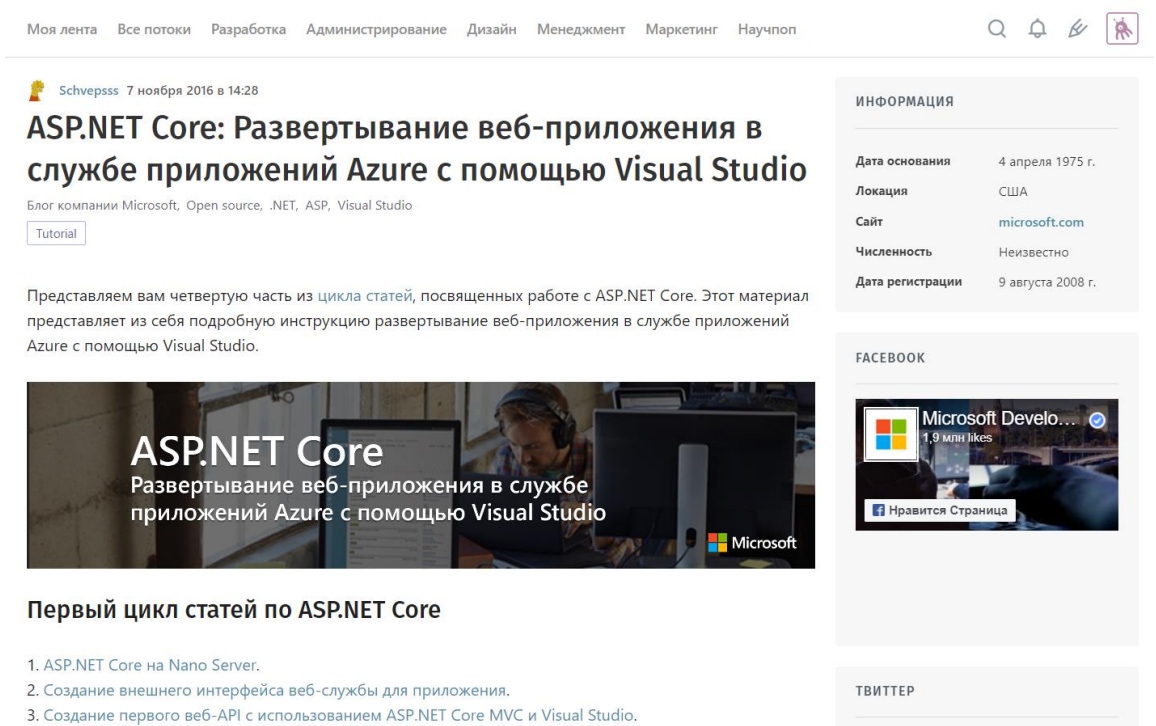


Рисунок 1.2 - Внешний вид статьи на сайте habr.com

2 Анализ поставленной задачи

2.1 Требования к разрабатываемой системе

Проанализировав необходимые функциональные возможности, аналогичные платформы и собственный опыт разработки были сформулированы следующие требования непосредственно к серверной части веб-приложения:

- В первую очередь сервер должен обеспечивать доступ к данным, в связи с чем необходимо проектирование, создание и подключение базы данных
- Отказоустойчивость – при разработке необходимо учесть возможные случаи неправильных запросов на сервер и некорректных данных. Сервер должен обрабатывать такие ситуации и выдавать корректное описание ошибок.
- Механизм авторизации и аутентификации. Кроме того, для повышения удобства использования сервер должен поддерживать аутентификацию через внешние сервисы, например, Google и VK.
- Поскольку большая часть сервиса работает по принципу SPA необходима поддержка клиентской маршрутизации
- Безопасное взаимодействие с БД – необходимо использовать такие средства доступа к данным, которые позволят избежать прямого включения данных, полученных в запросах к серверу, в запросы к БД т.е. устранить уязвимость перед SQL Injection.

2.2 Выбор средств разработки

- Направление разработки – SPA или MPA приложение

MPA (Multi Page Application) – многостраничные приложения. Это стандартные веб-приложения и сайты, на данный момент большая часть интернет ресурсов работают по принципу MPA. Суть состоит в том, что практически каждое действие пользователя принимается и обрабатывается сервером, после чего ему возвращается новая страница. Данный подход отлично подходит для ресурсов не требующих сложной логики на стороне клиента, также он довольно прост в реализации. Минусом является повышенная нагрузка на сервер и сложность параллельной разработки бекенда и фронтенда.

SPA (Single Page Application) – одностраничное веб приложение. Суть данного типа веб-приложение состоит в том, что они используют единственный HTML-документ для всего приложения, это означает, что при взаимодействии с интерфейсом страница будет не перезагружаться, а будет запрашивать с сервера данные посредством, например, технологии AJAX и получать их в виде HTML, CSS, JavaScript файлов, либо в формате JSON и затем встраивать на страницу. Такие приложения мгновенно реагируют на все действия пользователей, что делает их более привлекательными и позволяют разгрузить сервер.

В результате анализа обоих подходов было принято решение взять лучшее от обоих этих технологий - разрабатывать основное рабочее пространство пользователь по принципу SPA, а некоторые дополнительные элементы интерфейса с помощью MPA.

- Технология и язык разработки - PHP или ASP.NET

Согласно статистике [1] на текущий момент большая часть интернет ресурсов написана на PHP, на втором месте находится ASP.NET (см. рисунок 2.1).

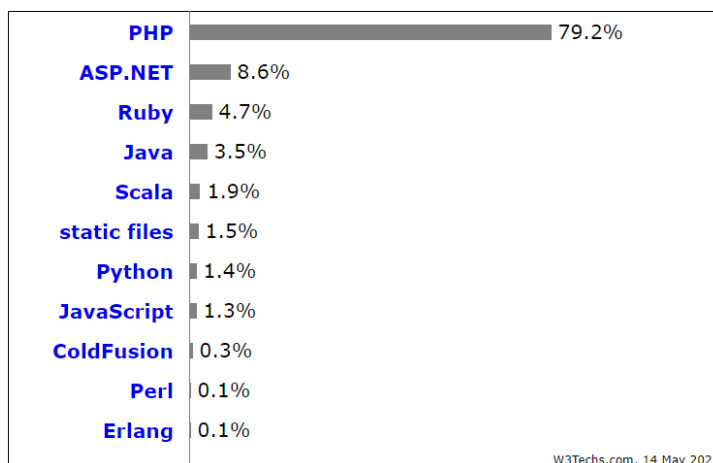


Рисунок 2.1 - Статистика использования серверных языков программирования

Несмотря на то, что PHP является более популярным, он используется как правило в небольших проектах, в отличие от ASP.NET. Для написания сайтов и приложений на платформе ASP.NET требуется несколько больше знаний и опыта, но и результат получается качественней, поэтому будем использовать эту платформу. Кроме того, наличие большего опыта разработки на языке C#, который является основным языком на данной платформе также повлияло на выбор технологии ASP.NET.

- IDE (Среда разработки) - Microsoft Visual Studio или WebStorm

Обе среды является достаточно мощными средами разработки и позволяют разрабатывать как консольные приложения, так и приложения с графическим интерфейсом. Обе включают большое количество инструментов и функций для разработки и отладки кода. Например, подсветку разметки и быстрый переход с помощью полос прокрутки, а также приостановку приложения и получение значений переменных на любом этапе. Однако, для Microsoft Visual Studio, в отличие от WebStorm, существует бесплатная лицензионная версия, поэтому будем использовать её.

- База данных - SQL Server или SQLite

SQL Server — система управления реляционными базами данных (СУБД) разработанная корпорацией Microsoft. Основным используемый язык запросов — Transact-SQL. Для администрирования и управления SQL Server будет применяться MS SQL Server Management Studio. Можно было использовать SQLite вместо SQL Server. Sqlite – очень легкая версия SQL, которая не поддерживает часть функций SQL, например, хранимые процедуры, которые могут понадобиться в будущем при расширении приложения и усложнении логики работы с данными, поэтому было принято решение использовать именно SQL Server.

- Тестирование WEB Api

Для тестирования WEB Api была выбрана довольно популярная среди разработчиков программа Postman. Она позволяет удобно и просто отправлять POST, GET, PUT и другие виды запросов на сервер, отлаживать логику обработки запросов и просматривать результат. Кроме того, она обладает графическим интерфейсом в связи с чем намного удобнее, чем, например, схожая по возможностям консольная утилита CURL.

- Система контроля версий – Git

Git – одна из самых популярных в мире систем управления версиями. По ней много документации и бесплатных сервисов для хостинга проектов. В качестве хостинга для git-

репозитории был выбран крупнейший веб-сервис для совместной разработки проектов - GitHub.

2.3 Выбор вариантов реализации требований

В первую очередь необходимо было выбрать способ доступа к данным и взаимодействия с БД. На платформе ASP.NET Core, согласно рекомендациям Microsoft, лучше всего взаимодействовать с БД через официальный пакет Microsoft.EntityFrameworkCore, поэтому будем использовать его, а поскольку БД является SQL Server, то воспользуемся решением для SQL сервера - Microsoft.EntityFrameworkCore.SqlServer. EF Core является ORM-инструментом, то есть позволяет работать с базами данных, но позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными независимо от типа хранилища.

В случае изменения моделей или добавления новых свойств необходимо эффективно применять эти изменения к БД без потери данных, для этого будем пользоваться механизмом миграций, который позволяет строить планы переходов от старых схем БД к новым. Для этого также воспользуемся официальным пакетом Microsoft.EntityFrameworkCore.Tools.

В связи с тем, что сервис имеет образовательную направленность и значительная часть материала – это большие объемы текста с изображениями необходимо было решить вопрос со способом хранения данной информации. Для этого были изучены описанные в предыдущей главе аналогичные сервисы и сайты.

Например, исходя из анализа ресурсов с помощью консоли разработчика, на сайте metanit.com (см. рисунок 2.2) статьи и изображения хранятся в виде статических файлов в файловой системе сервера. Для каждого раздела существует отдельная папка, которая хранит список статей, относящихся к данной теме и папку со всеми изображениями, используемыми в данных статьях.

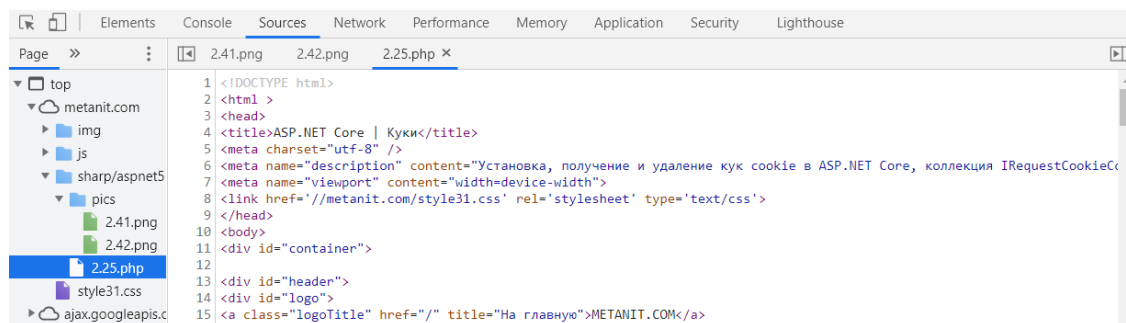


Рисунок 2.2 - Хранение статей на сайте Metanit.com

Похожим образом ситуация обстоит на сайте habr.com (см. рисунок 2.3). Здесь также посты хранятся в виде статических файлов в отдельных папках, поэтому было принято решение использовать такой же подход.

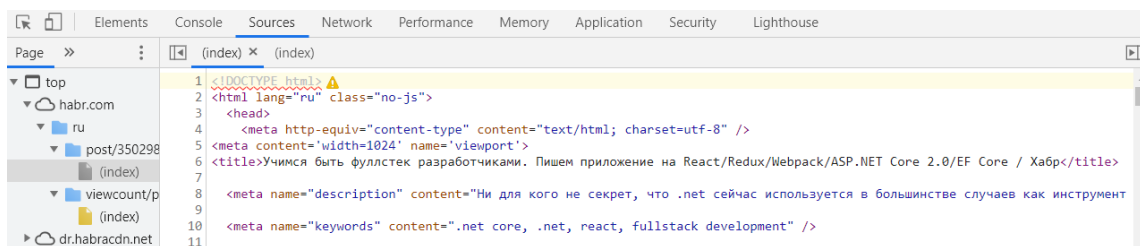


Рисунок 2.3 - Хранение статей на сайте habr.com

Отличием между двумя рассматриваемыми ресурсами является хранение изображений. Поскольку на сайте habr.com. посты могут писать не только создатели сайта, но и обычные пользователи – было создано глобальное хранилище (habrastorage) в котором хранятся все изображения, добавляемые пользователями. Это позволяет иметь доступ к изображениям из любого места сайта и удобно встраивать ссылки на изображения в html документ. Данный подход является более универсальным и позволит в будущем добавить возможность публикации материалов для пользователей сервиса, поэтому будем использовать его.

Для авторизации пользователей будем использовать систему ASP.NET Identity, которая представляет встроенный в ASP.NET механизм аутентификации и авторизации. Данная система позволяет удобно создавать учетные записи пользователей, аутентифицироваться их, управлять учетными записями, кроме того, данная система может использовать для входа на сайт учетные записи внешних провайдеров, таких как Facebook, Google, Microsoft, Twitter что в будущем также пригодится.

3 Описание результатов разработки

3.1 Проектирование таблиц базы данных

База данных должна содержать информацию о пользователях, заметках, уроках, статьях, тестах и заметках пользователей. Исходя из этих требований была спроектирована следующая структура базы данных:

Поскольку на сервере используется встроенная в ASP.NET система аутентификации и авторизации - ASP.NET Identity, то нет необходимости проектировать таблицы, связанные с пользователями и ролями. Данная система автоматически добавит необходимые таблицы при создании базы данных, а именно:

- AspNetUsers - таблица пользователей
- AspNetRoles - таблица ролей
- AspNetRoleClaims - таблица связи ролей и объектов claims
- AspNetUserLogins - таблица связи пользователей с их логинами во внешних сервисах
- AspNetUserClaims - таблица связи пользователей и объектов claims
- AspNetUserRoles - таблице связи пользователей и их ролей

- AspNetUserTokens - таблице токенов пользователей

Для хранения информации об уроках будет использоваться структура из 3 таблиц: Topics, Sections и Lessons.

Таблица Topics содержит информацию об темах изучения (C#, JavaScript и т.д.) имеет поля:

- Id – уникальный идентификатор темы (первичный ключ), тип данных int,
- Name – название темы, тип данных - string

Каждой теме из таблицы Topics соответствует несколько разделов, которые хранятся в таблице Sections, которая имеет поля:

- Id – идентификатор раздела (Primary Key), тип данных int
- TopicId – идентификатор темы (Primary Key), внешний ключ на поле Id таблицы Topics, тип данных int
- Name – название темы, тип данных - string

Для каждой темы индексация разделов начинается заново, поэтому поля Id и TopicId образуют составной первичный ключ.

Аналогичным образом, каждому разделу из таблицы Sections соответствует некоторое количество уроков, которые хранятся в таблице Lessons, и которая имеет поля:

- Id – идентификатор урока (Primary Key), тип данных int
- SectionId – идентификатор раздела (Primary Key), внешний ключ на поле Id таблицы Sections, тип данных int
- SectionTopicId – идентификатор темы (Primary Key), внешний ключ на поле TopicId таблицы Sections, тип данных int
- Name – название урока, тип данных - string
- Path – путь к html документу с уроком в файловой системе сервера, тип данных - string

Так же, как и в предыдущей таблице, для каждой темы и раздела индексация уроков начинается заново, поэтому поля Id, SectionId и SectionTopicId образуют составной первичный ключ.

Для хранения пользовательских статей не предусматривается подобной древовидной структуры, поэтому будет достаточно одной таблицы Articles со следующими полями:

- Id – уникальный идентификатор статьи (Primary Key), тип данных int
- UserId – ссылка на автора статьи, внешний ключ на поле Id таблицы AspNetUsers, тип данных int

- SectionTopicId – идентификатор темы (Primary Key), внешний ключ на поле Id таблицы Topics, тип данных int
- Title – заголовок статьи, тип данных - string
- Description – краткое описание содержания статьи, тип данных - string
- Path – путь к html документу со статьей в файловой системе сервера, тип данных – string
- ImagePath – путь к изображению (preview) статьи, тип данных - string

У пользователей должна быть возможность в процессе обучения создавать заметки, информация о них будет храниться в таблице Notes со следующими полями:

- Id – уникальный идентификатор статьи (Primary Key), тип данных int
- UserId – ссылка на автора статьи, внешний ключ на поле Id таблицыAspNetUsers, тип данных int
- Name – название заметки, тип данных – string
- Text – текст заметки, тип данных - string

Для проверки изученного материала на сервисе должны быть доступны тесты, за это будут отвечать таблицы Tests и Questions:

Таблица Tests содержит список тестов и имеет следующие поля:

- Id – уникальный идентификатор теста (Primary Key), тип данных int
- Title – название теста, тип данных int
- ImagePath – путь к изображению (preview) теста, тип данных – string
- TopicId - внешний ключ на поле Id таблицы Topics, тип данных int

Каждый тест содержит ряд вопросов, которые хранятся в таблице Questions со следующими полями:

- Id – идентификатор вопроса (Primary Key), тип данных int
- TestId – идентификатор теста (Primary Key), внешний ключ на поле Id таблицы Tests, тип данных int
- Type – тип вопроса (1 – с выбором ответа, 0 – с текстовым вводом), тип данных bool
- Options – список вариантов ответов для тестов с типом 1, тип данных string
- Correct – правильный ответ

Для авторизованных пользователей необходимо сохранять результат прохождения тестов. Эта информация будет храниться в таблице UsersTests, которая связывает пользователей с тестами и имеет следующие поля:

- UserId – идентификатор пользователя (Primary Key), внешний ключ на поле Id таблицыAspNetUsers, тип данных int
- TestId – идентификатор теста (Primary Key), внешний ключ на поле Id таблицы Tests, тип данных int
- Rating – результат последнего прохождения теста в процентах

Упрощенная структура БД и связей между таблицами представлена на рисунке 3.1.

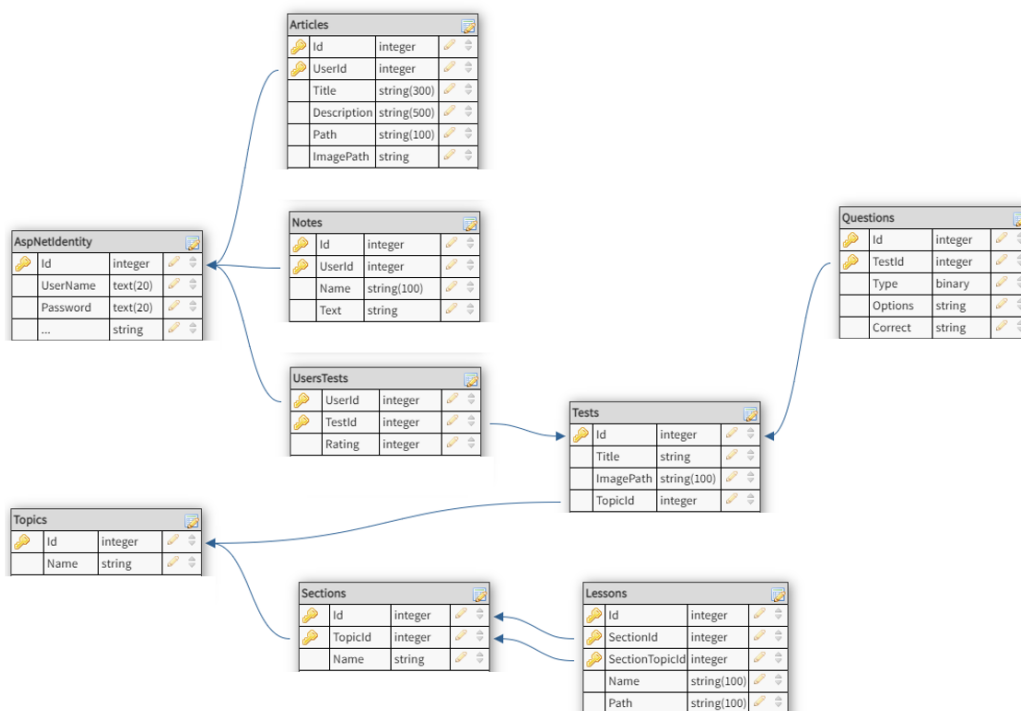


Рисунок 3.1 - Упрощенная структура БД

Стрелки в схеме обозначают внешние ключи. Направление стрелки - от внешнего ключа к первичному. Полная ER-диаграмма БД приведена в приложении А.

3.2 Создание моделей сущностей и классов для взаимодействия с БД

При создании классов для доступа в данным воспользуемся довольно популярным паттерном «Репозиторий» [2]. Использование классов - репозиториев позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными (как правило это класс контекста данных, который наследуется от базового класса DbContext из пакета EF Core), и остальной программой.

Реализация данного паттерна состоит из трех основных компонентов:

- Интерфейсов репозиториев, которые описывают обязательные для классов - репозиториев методы.
- Классов – репозиториев, которые реализуют соответствующие интерфейсы
- Класса доступа к данным

В результате в проект была добавлена папка DBRepository со следующей структурой (см. рисунок 3.2).

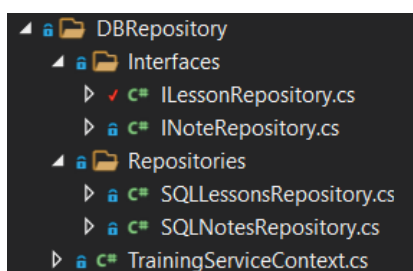


Рисунок 3.2 - Структура каталога DBRepository

В качестве примера рассмотрим взаимодействие с таблицей Notes. Сперва в проект была добавлена модель, описывающая сущность заметки (см. рисунок 3.3). Исходный код остальных моделей сущностей БД приведен в приложении Б.1.

```
public class Note
{
    //Внешний ключ
    public string UserId { get; set; }
    //Навигационное свойство
    public User User { get; set; }
    public int Id { get; set; }
    [MaxLength(100)]
    public string Name { get; set; }
    public string Text { get; set; }
}
```

Рисунок 3.3 - Класс описывающий сущность заметки пользователя

Для доступа к данным используется класс TrainingServiceContext (см. рисунок 3.4), который использует подключение к SQL Серверу. Для автоматического добавления таблиц системы ASP.NET Identity [3][4] данный класс необходимо унаследовать от класса IdentityDbContext<User>.

```
public class TrainingServiceContext: IdentityDbContext<User>
{
    public
    TrainingServiceContext(DbContextOptions<TrainingServiceContext> options) :
    base(options) { }
    //Добавление таблиц
    public DbSet<Note> Notes { get; set; }
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);
        //Установка первичных ключей
        modelBuilder.Entity<Note>().HasKey(u => new { u.Id, u.UserId
    });
}
```

Рисунок 3.4 - Класс контекста данных «TrainingServiceContext»

Для репозитория определен следующий интерфейс (см. рисунок 3.5). Так, репозиторий заметок должен содержать три метода: создание новой заметки, удаление заметки и получение всех заметок пользователя.

```
public interface INoteRepository
```



```

{
    List<Note> GetUserNotes(string userId);
    void DeleteNote(string userId, int noteId);
    void AddNote(Note newNote);
}

```

Рисунок 3.5 - Интерфейс INoteRepository

В качестве репозитория выступает класс SQLNotesRepository (см. рисунок 3.6), который реализует интерфейс INoteRepository и адаптирован для использования подключения к SQL Серверу.

```

public class SQLNotesRepository : INoteRepository
{
    private TrainingServiceContext context;
    //определяем контекст данных в конструкторе
    public SQLNotesRepository(TrainingServiceContext _db)
    {
        context = _db;
    }
    //Метод получения пользовательских заметок
    public List<Note> GetUserNotes(string userId)
    {
        return context.Notes.Where(note => note.UserId==userId).ToList();
    }
    //Метод добавления новой заметки
    public void AddNote(Note newNote)
    {
        context.Notes.Add(newNote);
        Save();
    }
    //Метод удаления заметки
    public void DeleteNote(string userId, int noteId)
    {
        //Находим заметку по идентификаторам пользователя и заметки
        Note noteToRemove = context.Notes
            .Where(note => note.Id == noteId && note.UserId == userId)
            .FirstOrDefault();
        //Удаляем заметку
        context.Notes.Remove(noteToRemove);
        Save();
    }
    public void Save()
    {
        context.SaveChanges();
    }
}

```

Рисунок 3.6 - Класс репозитория «SQLNotesRepository»

Для удобного доступа к контексту данных воспользуемся механизмом Dependency Injection (DI) [5] и зарегистрируем контекст данных в качестве сервиса в классе Startup (см. рисунок 3.7).

```

public void ConfigureServices(IServiceCollection services)
{
    //получаем строку подключения из файла appsettings.json
    string connection =
        Configuration.GetConnectionString("DefaultConnection");
    //Регистрируем контекст данных в качестве сервиса через механизм DI
    services.AddDbContext<TrainingServiceContext>(options =>
        options.UseSqlServer(connection));
}

```

Рисунок 3.7 - Регистрация контекста данных в качестве сервиса

Теперь, в контроллере воспользуемся созданным репозиторием (см. рисунок 3.8).

```
[Route("api/[controller]")]
public class NotesController : Controller
{
    IRepository _noteRepository;
    private readonly UserManager<User> _userManager;
    public NotesController(UserManager<User> userManager,
        TrainingServiceContext dbContext)
    {
        _userManager = userManager;
        //Создаем репозиторий для взаимодействия с таблицей Notes
        _noteRepository = new SQLNotesRepository(dbContext);
    }
}
```

Рисунок 3.8 - Контроллер NotesController

Таким образом, все методы контроллера работают с методами интерфейса IRepository, а тип репозитория определяется непосредственно в конструкторе. Это позволяет практически избавиться от зависимости к определенному типу подключения и позволит в будущем легко переключиться на другую БД при необходимости.

Взаимодействие с остальными сущностями БД реализовано по такому же принципу. Для каждой таблицы БД была создана соответствующая модель. Также были добавлены следующие интерфейсы:

- IArticleRepository
 - метод GetArticlesList – получение списка статей
 - метод Get Article – получение конкретной сатьи
 - метод AddArticle – добавление статьи
- ILessonRepository
 - GetLesson – получение урока
 - AddLesson – добавление урока
 - GetLessons – получение списка разделов с уроками
- IRepository
 - GetUserNotes – получение пользовательских заметок
 - DeleteNote – удаление заметки
 - AddNote – добавление заметки
 - GetNewUserNoteId – следующий id заметки для текущего пользователя
- ITestRepository
 - GetTestsWithUserRating – список тестов выбранной темы с рейтингом пользователя
 - GetTestWithQuestions – получение теста с вопросами
 - UpdateRating – обновление рейтинга пользователя

Исходный код классов-репозитория, реализующих эти интерфейсы приведен в приложении В.

3.3 Организация работы SPA части сервиса

3.3.1 Обмен данными между клиентской и серверной частью

Для успешного функционирования фронтенда – сервер должен принимать запросы с сервера, обрабатывать их и возвращать результат. За это отвечают контроллеры.

Исходя из описания функциональных возможностей - сервис должен предоставлять пользователям несколько основных возможностей – чтение авторских статей, изучение языков программирования и других технологий посредством уроков, создание заметок и прохождение тестов. Для реализации этих функций на сервер были добавлены соответствующие контроллеры, `ArticlesController`, `LessonsController`, `NotesController` и `TestsController` каждый из которых обрабатывает запросы по определенным маршрутам

3.3.1.1 Контроллер `ArticlesController`

Метод `GetArticles` (см. рисунок 3.9) – обрабатывает запрос `api/articles`. Тип запроса – GET. Тип возвращаемого значения – `JsonResult(List<ArticleResponse>)`. Возвращает список всех статей на сервере.

```
[Route("")]
[HttpGet]
public JsonResult GetArticles()
{
    return new JsonResult(_articlesRepository.GetArticlesList());
}
```

Рисунок 3.9 - Метод `GetArticles` контроллера `ArticlesController`

Каждая статья в списке должна содержать идентификатор, путь к картинке, заголовок, описание, дату создания и имя автора статьи, что довольно сильно отличается от полей в модели `Article`. Для решения данной проблемы для ответа на этот и некоторые другие запросы были созданы отдельные модели – `ResponsesModels` (см. приложение Б.2). В случае с методом `GetArticles` – возвращаемое значение - список с экземплярами класса `ArticleResponse` (см. рисунок 3.10) в котором отсутствует поле с идентификатором пользователя (`UserId`), вместо которого есть поле с именем автора статьи (`Author`), а также нет поля `Path`, хранящего путь к документу со статьей.

```
public class ArticleResponse
{
    public int Id { get; set; }
    public string Image { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string Date { get; set; }
    public string Author { get; set; }
}
```

Рисунок 3.10 - Класс `ArticleResponse`

Получить информацию об имени автора можно только объединив таблицы Users и Articles и выбрав из результата необходимые данные, для этого в репозитории статей был добавлен соответствующий метод GetArticlesList (см. рисунок 3.11). Полный код класса SQLArticlesRepository приведен в приложении В.

```
public List<ArticleResponse> GetArticlesList()
{
    //выбираем все статьи из таблицы Articles
    return db.Articles.Select(article => article)
    //Объединяем по внешнему ключу с таблицей Users
        .Join(db.Users,
            //внешний ключ
            article => article.UserId,
            //первичный ключ
            user => user.Id,
            //преобразовываем данные в необходимый вид т.е. в экземпляры
            //класса ArticleResponse и превращаем в список
            (article, user) => new ArticleResponse {
                Id = article.Id,
                Image = article.ImagePath,
                Title = article.Title,
                Description = article.Description,
                Date=article.Date,
                Author=user.Email })
        .ToList();
}
```

Рисунок 3.11 - Метод GetArticlesList класса SQLArticlesRepository

Метод GetArticle (см. рисунок 3.12) – обрабатывает запрос api/articles/{id}. Тип запроса – GET. Принимает id статьи (тип-int), который передается в запросе. Тип возвращаемого значения – VirtualFileResult. Возвращает html документ со статьей, id которой было передано в запросе.

```
[Route("{articleId:int}")]
[HttpGet]
public VirtualFileResult GetArticle(int articleId)
{
    //получаем информацию о статье с переданным id из БД
    var article = _articlesRepository.GetArticle(articleId);
    //Если статья не найдена - возвращаем страницу и информацией об ошибке
    if (article == null) return File("/Files/Error.html", "text/html");
    //Обращаемся к файловой системе и возвращаем файл используя полученный из
    //БД пути и заданный тип
    return File(article.Path, "text/html");
}
```

Рисунок 3.12 - Метод GetArticle контроллера ArticlesController

3.3.1.2 Контроллер LessonsController

Метод GetLessons (см. рисунок 3.13) – обрабатывает запрос api/lessons/{topicName}. Тип запроса – GET. Тип возвращаемого значения – JsonResult(List<ResponseSection>). Возвращает список разделов и входящих в них уроков, которые принадлежат переданной в запросе теме (sharp, js или sql).

```
[Route("{topicString:maxlength(10)}")]
[HttpGet]
public JsonResult GetLessons(string topicName)
{
    //...
```

```

        return new JsonResult(_lessonsRepository.GetLessonsList(topicName));
    }

```

Рисунок 3.13 - Метод GetLessons контроллера LessonsController

В соответствующем репозитории - SQLLessonsRepository (см. приложение В) также реализован метод GetLessonsList (см. рисунок 3.14)

```

public List<ResponseSection> GetLessonsList(string topicName)
{
    //по имени темы находим id
    int topicId = context.Topics.FirstOrDefault(topic => topic.Name ==
topicName).Id;
    //получаем все разделы этой темы и собираем данные каждого раздела в
класс ResponseSection, после чего формируем список
    return context.Sections.Where(section => section.TopicId == topicId)
        .Select(section => new ResponseSection { TopicId =
section.TopicId, Id = section.Id, SectionName = section.Name, Lessons =
section.Lessons })
        .ToList();
}

```

Рисунок 3.14 - Метод GetLessonsList класса SQLLessonsRepository

Как и в предыдущем контроллере, для ответа была добавлена дополнительная модель – ResponseSection (см. рисунок 3.15).

```

public class ResponseSection
{
    public int Id { get; set; }
    public string SectionName { get; set; }
    public int TopicId { get; set; }
    public List<Lesson> Lessons { get; set; }
}

```

Рисунок 3.15 - Класс ResponseSection

Метод GetLesson (см. рисунок 3.16) – обрабатывает запрос api/lessons/{topicName}/{sectionId}/{lessonId}. Тип запроса – GET. Тип возвращаемого значения – VirtualFileResult. Возвращает html документ с уроком, id, раздел и тема которого были переданы в запросе.

```

[Route("{topicString:maxlength(10)}/{sectionId:int}/{lessonId:int}")]
[HttpGet]
public VirtualFileResult GetLesson(string topicName, int sectionId, int
lessonId)
{
    //находим урок по переданным параметрам
    Lesson lesson = _lessonsRepository.GetLesson(topicName, sectionId,
lessonId);
    //если урок не найден - возвращаем страницу с ошибкой
    if (lesson == null) return File("/Files/Error.html", "text/html");
    //по указанному в БД пути получаем html документ и возвращаем его
    return File(lesson.Path, "text/html");
}

```

Рисунок 3.16 - Метод GetLesson контроллера LessonsController

3.3.1.3 Контроллер NotesController

Данный контроллер защищён атрибутом [Authorize], поскольку только у зарегистрированных пользователей есть возможность создавать заметки и в последствии их читать. Поскольку для авторизации и аутентификации используется система Identity, то

информацию о текущем пользователе можно через специальный класс UserManager, а также через имеющееся в пространстве имен ASP.NET Core.Mvc свойство User. Механизм авторизации и аутентификации будет подробнее рассмотрен в следующей главе.

Метод GetUserNotes (см. рисунок 3.17) – обрабатывает запрос api/notes. Тип запроса – GET. Тип возвращаемого значения – JsonResult(List<NodeResponse>). Возвращает список всех заметок принадлежащих пользователю.

```
[Route("")]
[HttpGet]
public JsonResult GetUserNotes()
{
    return new
    JsonResult(_notesRepository.GetUserNotes(_userManager.GetUserId(User)));
}
```

Рисунок 3.17 - Метод GetUserNotes контроллера NotesController

Метод AddNotes (см. рисунок 3.18) – обрабатывает запрос api/notes/addnote. Получает из тела запроса новую заметку и добавляет её в БД. Тип запроса – POST. Тип возвращаемого значения – JsonResult(string). Возвращает результат добавления новой заметки – успешно/не успешно.

```
[Route("addnote")]
[HttpPost]
public JsonResult AddNote()
{
    //определяем специальный класс StreamReader для чтения данных из тела
    запроса
    var stream = new StreamReader(Request.Body);
    //считываем данные
    var body = stream.ReadToEndAsync().Result;
    //создаем экземпляр класса описывающего заметку для добавления
    NewNoteRequest newNote;
    //пытаемся конвертировать данные JSON в объект класса NewNoteRequest
    try
    {
        newNote = JsonConvert.DeserializeObject<NewNoteRequest>(body);
    }
    //в случае возникновения отлавливаем ошибки
    catch (JsonReaderException e)
    {
        return new JsonResult(e.ToString());
    }
    //проверяем, что все поля заполнены
    if (!TryValidateModel(newNote)) return new JsonResult("Note Content
    Error!");
    //получаем id пользователя
    string userId = _userManager.GetUserId(User);
    //определяем id новой заметки
    int newId = _notesRepository.GetNewUserId(userId);
    //добавляем новую заметку, преобразуя данные в класс модели БД - Note
    _notesRepository.AddNote(new Note { UserId = userId, Id = newId, Text
    = newNote.Text, Title = newNote.Title });
    //сообщаем об успешном добавлении
    return new JsonResult("success");
}
```

Рисунок 3.18 - Метод AddNotes контроллера NotesController

Из соображений безопасности, для получения данных о новой заметке также была добавлена специальная модель `NewNoteRequest` (см. рисунок 3.19), содержащая только заголовок и текст заметки. Это сделано для того, чтобы нельзя было подменить пользователя или указать некорректный `id`. Кроме того, используя атрибуты `[Required]`, можно удобным способом проверять заполнение всех полей при валидации данных в контроллере.

```
public class NewNoteRequest : IValidatableObject
{
    [Required]
    public string Title { get; set; }
    [Required]
    public string Text { get; set; }
}
```

Рисунок 3.19 - Класс `NewNoteRequest`

3.3.1.4 Контроллер `TestsController`

Аналогично контроллеру `NotesController`, данный контроллер защищён атрибутом `[Authorize]` т.к. доступ к тестам есть только у зарегистрированных пользователей.

Метод `GetTestsWithUserRating` – обрабатывает запрос `api/tests/{topic}`. Тип запроса – GET. Тип возвращаемого значения – `JsonResult(List<TestResponse>)`. Возвращает список тестов принадлежащих переданной теме и рейтинг пользователя по каждому тесту. Контроллер принимает название темы, определяет текущего пользователя и затем передает эти данные в одноименный метод (см. рисунок 3.20) репозитория тестов `ITestRepository` (см. приложение В). Принцип работы метода заключается в том, что информация в таблицу `UsersTests` добавляется только после первого прохождения теста пользователем, поэтому по умолчанию результат всех тестов считается равным 0, после чего проверяются записи в таблице `UsersTests` для текущего пользователя и для соответствующих тестов в итоговом списке поле `Rating` корректируется.

```
public List<TestResponse> GetTestsWithUserRating(string userId, string topicName)
{
    //получаем Id темы по названию
    int topicId = db.Topics.FirstOrDefault(topic => topic.Name ==
topicName).Id;
    //создаем список объектов TestResponse, по умолчанию полю рейтинг
    присваиваем значение 0
    List<TestResponse> result = db.Tests.Where(test => test.TopicId==
topicId).Select(test => new TestResponse { Id = test.Id, Image = test.ImagePath,
Title = test.Title, Rating = 0 }).ToList();
    //если данные не найдены - возвращаем пустой список
    if (result == null) return result;
    //Получаем результаты всех тестов, пройденных пользователем
    List<UserTest> usersTestsList = db.UsersTests.Where(userTest =>
userTest.UserId==userId).ToList();
    //Для каждого теста пройденного пользователем
    foreach (var usertest in usersTestsList)
    {
        //ищем соответствующий тест в итоговом списке
        var test = result.Find(testResp => testResp.Id ==
usertest.TestId);
```

```

        //меняем рейтинг с 0 на результат пользователя
        test.Rating = usertest.Rating;
    }
    return result;
}

```

Рисунок 3.20 - Метод GetTestsWithUserRating класса SQLTestsRepository

Аналогично методам предыдущих контроллеров, результатом данного метода является объединение нескольких таблиц, поэтому была добавлена соответствующая модель TestResponse (см. рисунок 3.21).

```

public class TestResponse
{
    public int Id { get; set; }
    public string Image { get; set; }
    public string Title { get; set; }
    public int Rating { get; set; }
}

```

Рисунок 3.21 - Класс TestResponse

Метод GetTestWithQuestions – обрабатывает запрос api/tests/{topic}/{id}. Тип запроса – GET. Тип возвращаемого значения – JsonResult(TestResponse). Возвращает тест со списком вопросов. Контроллер принимает id теста передает его в одноименный метод (см. рисунок 3.22) репозитория тестов ITestRepository. В БД варианты ответов на вопрос хранятся в виде строки. Для разделения вариантов была выбрана подстрока "&&&". Данный метод разделяет исходную строку из БД по данной подстроке и преобразует результат в массив.

```

public TestWithQuestionsResponse GetTestWithQuestions(int testId)
{
    //определяем символ по которому будет разбиваться строка
    string[] questionSeparator = { "&&&" };
    //выбираем вопросы данного теста
    var questions = db.Questions.Where(question =>
question.TestId==testId)
        //преобразуем данные в необходимый формат
        .Select(question => new QuestionResponse {
            Id = question.Id, Question= question.QuestionText,
            Type= Convert.ToInt32(question.Type),
            Correct= question.Correct,
            //Преобразуем строку в массив вариантов ответов
            Options = question.Options.Split(questionSeparator,
System.StringSplitOptions.RemoveEmptyEntries).ToList())
        .ToList();
    //преобразуем данные в необходимый формат
    return new TestWithQuestionsResponse { Id = testId, Questions =
questions };
}

```

Рисунок 3.22 - Метод GetTestWithQuestions класса SQLTestsRepository

Реализация данной функциональности потребовала добавления двух дополнительных моделей TestWithQuestionsResponse (см. рисунок 3.23) и QuestionResponse (см. рисунок 3.24) ввиду того, что итоговый объект должен был иметь специфический набор полей – тест должен содержать массив вопросов, а каждый вопрос массив вариантов ответов, если его тип предполагает выбор ответа.


```
public class TestWithQuestionsResponse
{
    public int Id { get; set; }
    //список вопросов
    public List<QuestionResponse> Questions { get; set; }
}
```

Рисунок 3.23 - Класс TestWithQuestionsResponse

```
public class QuestionResponse
{
    public int Id { get; set; }
    //вопрос
    public string Question { get; set; }
    //тип - с выбором ответа/с вводом
    public int Type { get; set; }
    //для вопросов 1 типа - список вариантов
    public List<string> Options { get; set; }
    //правильный ответ
    public string Correct { get; set; }
}
```

Рисунок 3.24 - Класс QuestionResponse

Метод UpdateRating – обрабатывает запрос api/tests/updaterating. Тип запроса – PUT. Тип возвращаемого значения – JsonResult(string). Данный метод вызывается после завершения теста пользователем и обновляет результат в БД. Контроллер извлекает из тела запроса модель NewRatingResponse (см. приложение Б.3), содержащую данные об id теста и результате прохождения, после чего передает их в одноименный метод репозитория тестов ITestRepository (см. приложение В). Механизм обновления результата теста для пользователя в целом аналогичен механизму добавления новой заметки, с той разницей, что запись добавляется в таблицу UsersTests только один раз, после чего она только обновляется посредством метода Update.

В плане обмена информацией, взаимодействие основного SPA приложения и сервера ограничивается этими четырьмя контроллерами.

3.3.2 Обработка запроса к корневому адресу и отправка приложения клиенту

После сборки всех компонентов проекта клиентской части на выходе получается, как правило, следующий набор файлов (см. таблицу 1).

Таблица 1 – Файлы клиентской части после сборки проекта

Имя файла	Описание
index.html	документ, который подключает и запускает скрипт index-bundle.js
index-bundle.js	скрипт, содержащий всю логику приложения, упакованную в один файл и преобразованную в язык JavaScript. Использует стили из файла style.css и файлы из src
style.css	файл со стилями элементов
каталог src	изображения, иконки и другие статические файлы.

При обращении к корневому адресу сервера «/» необходимо либо передать все эти файлы сразу, либо вернуть представление, подключающее скрипт index-bundle.js. Чтобы не нарушать общий принцип обработки запросов на сервере – MVC, был использован второй вариант.

Сперва в методе Configure класса Startup (см. приложение Г) в конвейер обработки запроса был добавлен компонент, позволяющий использовать статические файлы – UseStaticFiles(), а также компонент, подключающий функционал MVC – UseMvc с маршрутом по умолчанию (см. рисунок 3.25). Таким образом, все обращения к корневому пути «/» будут перенаправляться на метод Index контроллера Home.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    //подключение статических файлов
    app.UseStaticFiles();
    //включение маршрутизации
    app.UseRouting();
    //включение механизма MVC
    app.UseMvc(routes =>
    {
        //определение маршрута по умолчанию
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}");
    });
}
```

Рисунок 3.25 - Метод Configure Класс Startup

Контроллер Home (см. рисунок 3.26) возвращает соответствующее представление.

```
public class HomeController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        return View();
    }
}
```

Рисунок 3.26 - Контроллер Home

В представлении (см. рисунок 3.27) был подключен вышеупомянутый файл index-bundle.js. При загрузке страницы этот скрипт встроит весь контент в элемент «app».

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div id="app">
    </div>
    <script type="text/javascript" src="@Url.Content("~/assets/index-
bundle.js")"></script>
</body>
</html>
```

Рисунок 3.27 - Представление Index.cshtml контроллера Home

После добавления в каталог `wwwroot`, который является хранилищем статических файлов по умолчанию, файлов `index-bundle.js`, `style.css` и каталога `src`, обратившись к корневому адресу, пользователи смогут получать наше приложение.

3.3.3 Добавление поддержки клиентской маршрутизации

В случае с SPA приложениями необходимо предусмотреть поддержку клиентской маршрутизации, поскольку на клиенте могут быть свои, отличные от сервера, маршруты.

Для решения данной проблемы у Microsoft есть специальный пакет - `Microsoft.AspNetCore.SpaServices`, который позволяет передавать запросы по всем неизвестным маршрутам на клиент. Таким образом, для подключения клиентской маршрутизации, необходимо добавить функционал данного пакета в существующий механизм маршрутизации (см. рисунок 3.28).

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}");
    routes.MapSpaFallbackRoute("spa-fallback", new { controller =
"Home", action = "Index" });
});
```

Рисунок 3.28 - Добавление клиентской маршрутизации в методе `Configure` класса `Startup`

3.4 Организация работы МРА части сервиса

3.4.1 Добавление механизма авторизации и аутентификации

Как было указано в пункте 2.3 – на сервере будет использоваться встроенная в ASP.NET система аутентификации и авторизации - `ASP.NET Identity` [3],[4]. Для взаимодействия с MS SQL Server через `ASP.NET Core Identity` необходимо добавить пакет `Microsoft.AspNetCore.Identity.EntityFrameworkCore` [4].

В классе `Startup` в методе `ConfigureServices` необходимо добавить соответствующий сервис (см. рисунок 3.29). Также, для упрощения процесса регистрации, изменим некоторые опции механизма `Identity`.

```
services.AddIdentity<User, IdentityRole>(options => {
    options.Password.RequiredLength = 5; // минимальная длина
    options.Password.RequireDigit = false; // требуются ли цифры
    options.User.RequireUniqueEmail = false; // подтверждение Email
    options.User.AllowedUserNameCharacters =
"abcdefghijklmnopqrstuvwxyz@.1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // допустимые
СИМВОЛЫ
}).AddEntityFrameworkStores<TrainingServiceContext>();
```

Рисунок 3.29 - Добавление сервисов `ASP.NET Core Identity`

Добавляем использование аутентификации и авторизации в методе `Configure` (см. рисунок 3.30).

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseAuthentication();
```

```

        app.UseAuthorization();
    }

```

Рисунок 3.30 - Добавление аутентификации и авторизации в проект

Аутентификация и авторизация пользователей состоит, как правило, из трёх основных компонентов:

- Регистрация нового пользователя
- Вход пользователя в систему
- Выход пользователя из системы

Для регистрации и входа в систему будут использоваться отдельные страницы т.е. принцип МРА. Рассмотрим процесс обмена данными на примере регистрации нового пользователя.

Для начала необходимо определиться с данными, которые необходимы для регистрации. На текущем этапе достаточно просто указать имя, пароль и повтор пароля. Для описания таких сущностей, которые используются в информационном обмене между представлениями и сервером применяют специальные классы, называемые ViewModels или модели представлений. Созданная модель представления для регистрации (см. рисунок 3.31). Атрибуты, используемые над полями класса, в последствии используются при генерации представления и при проверке валидности данных.

```

using System.ComponentModel.DataAnnotations;
public class RegisterViewModel
{
    [Required]
    [Display(Name = "Email")]
    public string Email { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Пароль")]
    public string Password { get; set; }

    [Required]
    [DataType(DataType.Password)]
    [Compare("Password", ErrorMessage = "Пароли не совпадают")]
    [Display(Name = "Подтвердить пароль")]
    public string PasswordConfirm { get; set; }
}

```

Рисунок 3.31 - Модель представления регистрации RegisterViewModel

За авторизацию и аутентификацию будет отвечать контроллер Account. Поскольку в классе Startup система ASP.NET Identity была зарегистрирована в качестве сервиса, то теперь в контроллерах можно получить доступ к специальным классам – UserManager, который позволяет управлять пользователями и SignInManager, который позволяет аутентифицировать пользователя и устанавливать или удалять cookie (см. рисунок 3.32).

```

public class AccountController : Controller
{
    private readonly UserManager<User> _userManager;

```

```

        private readonly SignInManager<User> _signInManager;
        public AccountController(UserManager<User> userManager,
            SignInManager<User> signInManager)
        {
            _userManager = userManager;
            _signInManager = signInManager;
        }
    }
}

```

Рисунок 3.32 - Конструктор контроллера Account

За регистрацию отвечает метод Register. Get-версия метода возвращает представление с формой для регистрации, POST-версия принимает модель RegisterViewModel и непосредственно регистрирует пользователя (см. рисунок 3.33). Сперва проверяется модель проверяется на валидность, т.е. соответствуют ли значения полей указанным в классе атрибутам. После чего создается новый пользователь и производится попытка его добавления в БД. Если всё проходит успешно, то пользователя авторизуют и перенаправляют на корневой адрес сервиса. По умолчанию в системе Identity используется авторизация на основе файлов cookie. Данный тип авторизации довольно прост в использовании, вместе с тем он отлично справляется со своей задачей, поэтому будем использовать его.

```

public async Task<IActionResult> Register()
{
    return View();
}
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model)
{
    // Проверяем модель на валидность
    if (ModelState.IsValid)
    {
        User user = new User { Email = model.Email, UserName = model.Email };

        // добавляем пользователя
        var result = await _userManager.CreateAsync(user, model.Password);
        if (result.Succeeded)
        {
            // установка куки
            await _signInManager.SignInAsync(user, false);
            return Redirect("~/");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }
    }
    return View(model);
}

```

Рисунок 3.33 - Метод Register контроллера Account

Теперь, остается только добавить форму регистрации в представление (см. рисунок 3.34). В данном случае используются не стандартные html-элементы, а так называемые tag-

хелперы. Они выглядят как обычные html-элементы или атрибуты, но при запросе представления обрабатываются специальным движком Razor конечном счете преобразуются в стандартные html-элементы [6]. В этом представлении они позволяют автоматически связать элементы html и методы контроллера Account, используя при этом интуитивно понятные атрибуты.

```
<form class="register" asp-action="Register" asp-controller="Account" asp-anti-
forgery="true">
    <div class="validation" asp-validation-summary="ModelOnly" />
    <div class="form-body">
        <section class="form-section">
            <label asp-for="Email">Введите логин</label>
            <input class="input-field" type="text" asp-for="Email" />
            <span asp-validation-for="Email"></span>
        </section>
        <section class="form-section">
            <label asp-for="Password">Введите пароль</label>
            <input class="input-field" type="Password" asp-for="Password"
/>
            <span asp-validation-for="Password"></span>
        </section>
        <section class="form-section">
            <label asp-for="PasswordConfirm">Повторите пароль</label>
            <input class="input-field" type="Password" asp-
for="PasswordConfirm" />
            <span asp-validation-for="PasswordConfirm"></span>
        </section>
        <section class="form-section">
            <input class="submit-button" type="submit" value="Регистрация"
/>
        </section>
    </div>
</form>
```

Рисунок 3.34 - Представление регистрации

Теперь при обращении по адресу «/account/register» пользователь получит форму регистрации (см. рисунок 3.35), после отправки которой, сервер добавит информации о новом пользователю в БД, установит аутентификационные cookie и перенаправит пользователя на корневой адрес, по которому пользователь получит SPA приложение.

Рисунок 3.35 - Форма регистрации

Для упрощения процесса авторизации в приложение была добавлена возможность входа через внешние сервисы, в частности Google и Vkontakte [7],[8]. Для этого необходимо было зарегистрировать приложение на соответствующих ресурсах – Google Cloud Console и VK Api, после чего получить Id приложения и секретный ключ. Авторизация в данном случае будет работать по схеме OAuth.

Для подключения механизма самой авторизации, в проект были добавлены NuGet пакеты Microsoft.AspNetCore.Authentication.Google и AspNet.Security.OAuth.Vkontakte, которые затем были подключены в методе ConfigureServices, используя ранее полученные ClientId и ClientSecret (см. рисунок 3.36).

```
services.AddAuthentication().AddGoogle(options =>
{
    options.ClientId = Configuration.GetConnectionString("GoogleClientId");
    options.ClientSecret =
Configuration.GetConnectionString("GoogleClientSecret");
}).AddVkontakte(options =>
{
    options.ClientId = Configuration.GetConnectionString("VkontakteClientId");
    options.ClientSecret =
Configuration.GetConnectionString("VkontakteClientSecret");
    options.Scope.Add("email");
});
```

Рисунок 3.36 - Добавление внешних сервисов авторизации

Также в контроллер AccountController были добавлены методы GoogleLogin и GoogleResponse. Метод GoogleLogin (см. рисунок 3.37) перенаправляет пользователя на страницу входа Google, дополнительно передавая ссылку на метод GoogleResponse.

```
public IActionResult GoogleLogin()
{
    //Формируем адрес на который Google будет возвращать результат
    //авторизации
    string redirectUrl = Url.Action("GoogleResponse", "Account");
    //формируем свойства проверки подлинности
    var properties =
_signInManager.ConfigureExternalAuthenticationProperties("Google", redirectUrl);
    //Возвращаем ChallengeResult с указанной схемой проверки подлинности и
    //свойствами.
    return new ChallengeResult("Google", properties);
}
```

Рисунок 3.37 - Метод GoogleLogin контроллера Account

Метод GoogleResponse (см. рисунок 3.38) вызывается после прохождения аутентификации в Google. Данный метод принимает данные о пользователе к которым Google разрешил доступ.

```
public async Task<IActionResult> GoogleResponse()
{
    //Получаем данные авторизации
    ExternalLoginInfo info = await
_signInManager.GetExternalLoginInfoAsync();
    if (info == null) return RedirectToAction(nameof(Login));
    //Пытаемся авторизовать пользователя в нашей системе
```

```

        //ищем данные о пользователе в таблице AspNetLogins
        var result = await
_signInManager.ExternalLoginSignInAsync(info.LoginProvider, info.ProviderKey,
false);
        //если всё прошло успешно - перенаправляем пользователя обратно на
главную страницу
        if (result.Succeeded)
            return Redirect("~/");
        else
        {
            //Если вход не удался т.е. пользователь впервые авторизуется -
создаем нового пользователя
            User user = new User
            {
                Email = info.Principal.FindFirst(ClaimTypes.Email).Value,
                UserName = info.Principal.FindFirst(ClaimTypes.Email).Value,
            };
            //добавляем пользователя в БД
            IdentityResult identResult = await _userManager.CreateAsync(user);
            if (identResult.Succeeded)
            {
                //добавляем данные в таблицу AspNetLogins
                identResult = await _userManager.AddLoginAsync(user, info);
                if (identResult.Succeeded)
                {
                    //авторизуем пользователя (устанавливаем cookie)
                    await _signInManager.SignInAsync(user, false);
                    return Redirect("~/");
                }
            }
            return AccessDenied();
        }
    }
}

```

Рисунок 3.38 - Метод GoogleResponse контроллера Account

Авторизация ВКонтакте работает аналогично.

Финальным шагом стало добавление ссылок на методы GoogleLogin и VkontakteLogin в представления Login и Register (см. рисунок 3.39).

```

<section class="form-section">
    <div class="auth-form">
        <a class="btn btn-google" style="background-image: url('./google.png')"
asp-action="GoogleLogin" asp-controller="Account"></a>
        <a class="btn btn-vk" style="background-image: url('./vk.png')" asp-
action="VkontakteLogin" asp-controller="Account"></a>
    </div>
</section>

```

Рисунок 3.39 - Ссылки на методы GoogleLogin и VKontakteLogin

Внешний вид представления входа в систему с добавленными ссылками представлен на рисунке 3.40.

Вход на сайт

Введите логин

Введите пароль

Запомнить ☐

Войти **Регистрация**




 

Рисунок 3.40 - Форма регистрации с ссылками на внешние сервисы авторизации

Перейдя по данным ссылкам получит соответствующие окна для входа в Google (см. рисунок 3.41) и VKontakte (см. рисунок 3.42).

 Войдите в аккаунт Google

Вход

Переход в приложение "somee.com"


Телефон или адрес эл. почты

[Забыли адрес электронной почты?](#)

Приложению "somee.com" будет предоставлен доступ к вашим данным: имени, адресу электронной почты, языковым настройкам и фото профиля.

[Создать аккаунт](#) [Далее](#)

Рисунок 3.41 - Форма входа в аккаунт Google

 Регистрация

Для продолжения вам необходимо войти **ВКонтакте**.

Телефон или email

Пароль

[Войти](#)

[Забыли пароль?](#)

Рисунок 3.42 - Форма входа в аккаунт VKontakte

После входа, например, в Google, в настройках безопасности аккаунта можно убедиться в предоставлении доступа сервису к необходимым данным и в любой момент отозвать разрешения (см. рисунок 3.43).

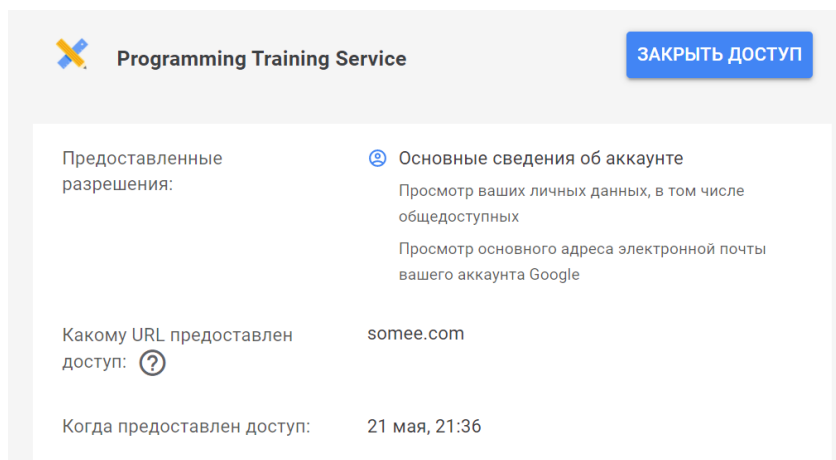


Рисунок 3.43 - Информация об использовании сервисом данных аккаунта Google

3.4.2 Механизм хранения и добавления изображений и html документов

Для хранения изображений использовалась файловая система сервера [9] (см. п.2.3). За добавление изображений отвечает контроллер PicturesStorageController. Для доступа к информации о файловой системе, в конструкторе контроллера присваивается переменная _appEnvironment (см. рисунок 3.44).

```
public class PicturesStorageController : Controller
{
    IWebHostEnvironment _appEnvironment;

    public PicturesStorageController( IWebHostEnvironment appEnvironment)
    {
        _appEnvironment = appEnvironment;
    }
}
```

Рисунок 3.44 - Определение переменной окружения в конструкторе контроллера PicturesStorageController

По запросу GET по адресу api/picturesstorage контроллер возвращает представление с простой формой для добавления файлов (см. рисунок 3.45).

```
@ViewData["Title"] = "Добавление картинки";
<h3>Выберите картинку для загрузки</h3>
<form asp-action="AddPicture" asp-controller="PicturesStorage" method="post"
    enctype="multipart/form-data">
    <p>
        <b>Прикрепите изображение</b><br>
        <input type="file" name="uploadedPicture" /><br>
    </p>
    <p>
        <input type="submit" value="Загрузить" />
    </p>
</form>
```

Рисунок 3.45 - Представление с формой добавления файлов

После загрузки файла, форма отправляет его методом POST на адрес `api/picturesstorage`, который обрабатывается методом `AddPicture` (см. рисунок 3.46).

```
public static async Task<String> AddPicture(IFormFile uploadedPicture,
IWebHostEnvironment appEnvironment)
{
    //проверка существования файла
    if (uploadedPicture != null)
    {
        //id картинки получаем посредством вычисления хэша
        int pictureId = uploadedPicture.OpenReadStream().GetHashCode();
        //формируем относительный путь с именем файла
        string CommonPicturePath = "/Files/PicturesStorage/" +
pictureId.ToString() + new Random().Next(0, 100).ToString() + ".jpg";
        //формируем полный путь с именем файла, используя информацию о
среде выполнения
        string LocalPicturePath = appEnvironment.WebRootPath +
CommonPicturePath;
        //создаем файл используя полный путь
        using (var fileStream = new FileStream(LocalPicturePath,
        FileMode.Create))
        {
            //копируем загруженный файл в файловую систему
            await uploadedPicture.CopyToAsync(fileStream);
        }
        //возвращаем путь, по которому можно получить доступ к файлу
        return CommonPicturePath;
    }
    return "error";
}
```

Рисунок 3.46 - Метод `AddPicture` контроллера `PicturesStorageController`

Внешний вид формы добавления изображений представлен на рисунке 3.47

Выберите картинку для загрузки

Прикрепите изображение

Выберите файл Файл не выбран

Загрузить

Рисунок 3.47 - Форма добавления изображений

Для добавления статей и уроков использовался похожий подход, с той разницей, что информация о новых статьях и уроках сохраняется дополнительно в БД. В контроллере `ArticlesController` был создан дополнительный метод `AddArticle` с GET и POST версиями, обрабатывающими запрос по маршруту `api/articles/addarticle`

GET-версия метода `AddArticle` возвращает представление с формой для добавления статьи (см. рисунок 3.48), которая содержит элементы для ввода названия и описания статьи, а также два блока для добавления файлов – изображения-превью и самой статьи в формате html. Если в самой статье также необходимо использовать изображения, то они должны быть предварительно добавлены через контроллер `PicturesStorageController`, а полученные адреса вставлены в документ html в необходимые тэги ``.

```
@ViewData["Title"] = "Добавление статьи";
```

```

<h3>Введите информацию о статье и добавьте файлы</h3>
<form asp-action="AddArticle" asp-controller="Articles" method="post"
enctype="multipart/form-data">
    <p>
        <b>Название статьи:</b><br>
        <input type="text" name="title">
    </p>
    <p>
        <b>Описание:</b><br>
        <input type="text" name="description">
    </p>
    <p>
        <b>Загрузите превью статьи:</b><br>
        <input type="file" name="uploadedArticlePicture" /><br>
    </p>
    <p>
        <b>Загрузите HTML файл с уроком:</b><br>
        <input type="file" name="uploadedNewArticleHTML" /><br>
    </p>
    <p>
        <input type="submit" value="Загрузить" />
    </p>
</form>

```

Рисунок 3.48 - Представление с формой добавления статьи

Внешний вид формы добавления статьи представлен на рисунке 3.49.

Рисунок 3.49 - Форма добавления новой статьи на сайт

POST-версия метода AddArticle (см. рисунок 3.50) принимает введенные в форму данные и обрабатывает их, в частности добавляет изображение и документ со статьей в файловую систему и создает запись о новой статье в БД.

```

[Route("addarticle")]
[HttpPost]
public async Task<IActionResult> AddArticle(string title, string description,
IFormFile uploadedNewArticleHTML, IFormFile uploadedArticlePicture)
{
    if (uploadedNewArticleHTML != null)
    {
        //формируем путь к папке с уроками
        string newLessonFolderPath = _appEnvironment.WebRootPath +
"/Files/Articles";
        //формируем путь к самому файлу
        string htmlPath = newLessonFolderPath + "/" +
uploadedNewArticleHTML.FileName;
        //создаем файл используя созданный путь
        using (var fileStream = new FileStream(htmlPath, FileMode.Create))

```

```

{
    // сохраняем файл в папку Files в каталоге wwwroot
    await uploadedNewArticleHTML.CopyToAsync(fileStream);
}
//создаем новую статью используя полученную информацию
Article newArticle = new Article {
    //присваиваем id автора
    UserId = _userManager.GetUserId(User),
    Title =title,
    Description=description,
    Path = "/Files/Articles/" + uploadedNewArticleHTML.FileName,
    //добавляем картинку в файловой системе и сохраняем путь к ней
    ImagePath = await
PicturesStorageLogic.AddPicture(uploadedArticlePicture, _appEnvironment),
    //определяем дату добавления
    Date = DateTime.Now.Date.ToString().Split(' ')[0]
};
//добавляем информацию в соответствующую таблицу в БД
_articlesRepository.AddArticle(newArticle);
}
return Redirect("~/");
}

```

Рисунок 3.50 Метод AddArticle контроллера ArticlesController

Новые уроки добавляются аналогичным образом, отличием является только отсутствие добавления изображения и наличие дополнительных сведений о принадлежности к теме и разделу.

3.4.3 Добавление административных ресурсов

Пока что, возможность добавлять изображения, уроки и статьи, не очень адаптирована для конечных пользователей. Поэтому в систему аутентификации были добавлены роли, а весь функционал по управлению контентом, а также пользователями и их правами был вынесен в отдельное окно, доступное только пользователям с ролью администратора.

За все административные настройки отвечает контроллер AdminController, который содержит следующие методы:

- Index – тип – GET. Возвращает представление с основным административным окном (см. рисунок 3.51).

Управление контентом

[Добавить урок](#) [Добавить статью](#) [Добавить изображение](#)

Список ролей

admin

[Добавить роль](#)

Список Пользователей

Email	Роль	
vadim	admin	Изменить Сменить пароль Удалить
vadKor	user	Изменить Сменить пароль Удалить
sasha	admin	Изменить Сменить пароль Удалить
Alexandr Butuzov	admin	Изменить Сменить пароль Удалить
AlexanderButuzov	user	Изменить Сменить пароль Удалить
sasha.butuzov9924@gmail.com	user	Изменить Сменить пароль Удалить
maxim	admin	Изменить Сменить пароль Удалить

[Добавить пользователя](#)[Web hosting by Somee.com](#)

2021 - Training Service

Рисунок 3.51 - Главная страница администратора

- CreateUser - тип – GET/POST. Возвращает представление с формой добавление нового пользователя. Принимает модель с именем и паролем.
- DeleteUser тип – POST. Принимает Id пользователя и удаляет его из БД.
- Edit - тип – GET/POST. Возвращает представление с формой для редактирования данных о пользователе. Позволяет изменить имя пользователя и его права. Принимает модель с новым именем пользователя и списком ролей.
- ChangePassword тип – GET/POST. Возвращает представление с формой для смены пароля пользователя. Принимает модель с идентификатором, именем и новым паролем пользователя и в случае корректности пароля – обновляет информацию в БД.
- CreateRole - тип – GET/POST. Возвращает представление с формой для создания новой роли. Принимает строку с именем новой роли.
- AddPicture – возвращает представление AddPicture контроллера PictureStorage
- AddLesson – возвращает представление AddLesson контроллера Lessons
- AddArticle – возвращает представление AddArticle контроллера Articles

Исходный код класса AdminController представлен в приложении Д.

3.5 Тестирование

3.5.1 Тестирование сервера

Тестирование сервера проводилось с использованием Postman [10]. Это специальная программа для тестирования работы различных API или просто отправки различных запросов на сервер. Кроме того, она позволяет настраивать передаваемые Cookie, HTTP заголовки, а также параметры форм.

В ходе тестирования была проверена и отлажена логика работы методов контроллеров, а также формат передаваемых данных. Далее приведены ответы сервера на основные запросы, которые приходят с фронтенда.

Ответ сервера по маршруту /api/articles (получение всех статей) изображен на рисунке 3.52 и представляет собой массив объектов, каждый из которых описывает статью и содержит информацию о названии, описании, дате создания и авторе, а также ссылки на изображение и документ со статьей.

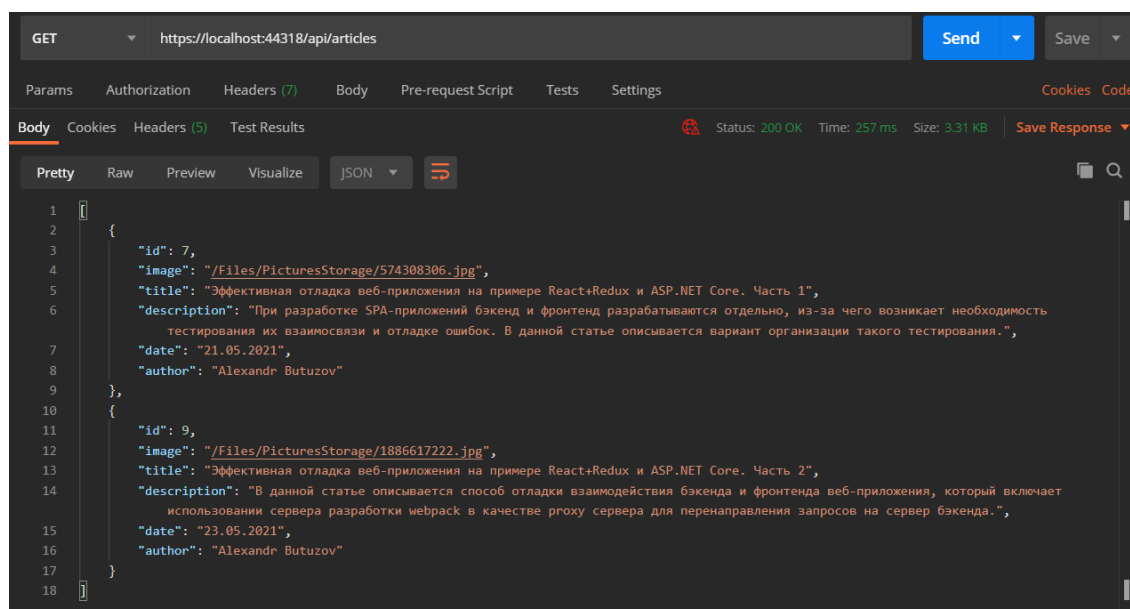
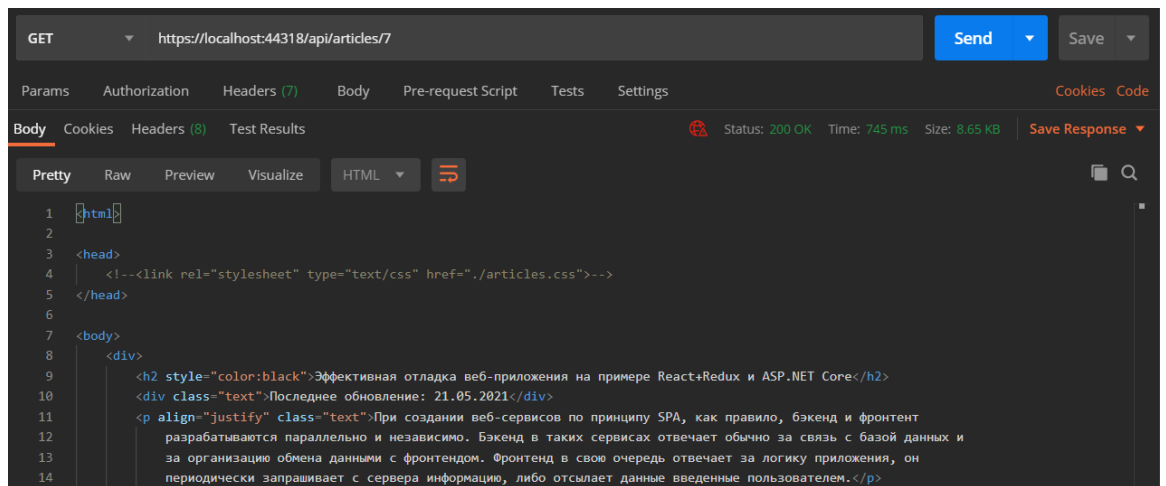


Рисунок 3.52 - Ответ сервера по маршруту /api/articles

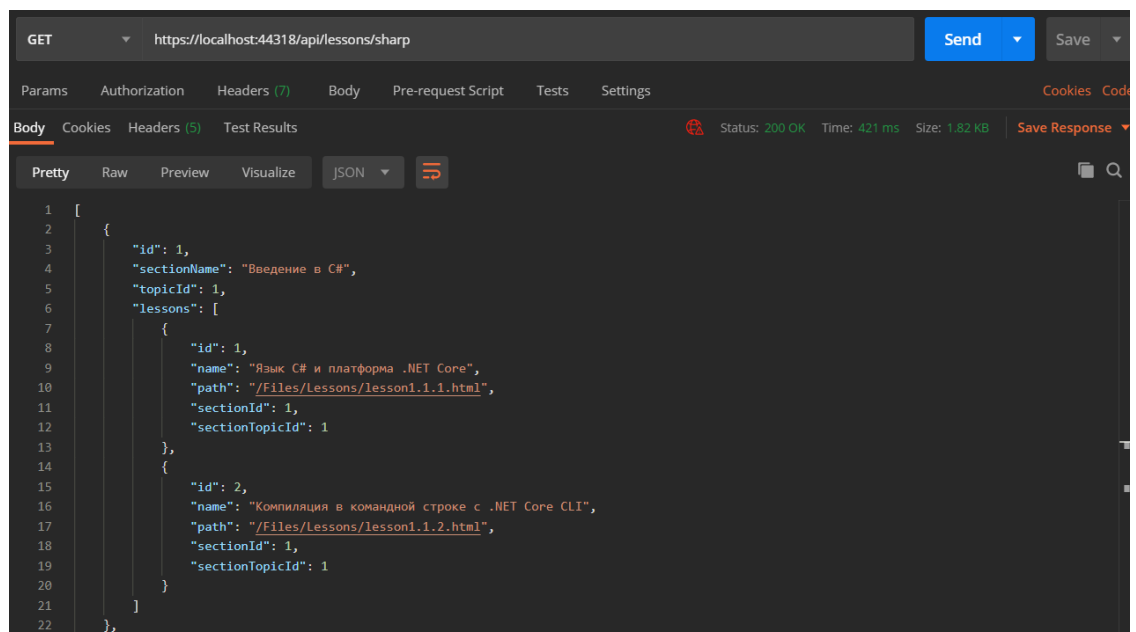
Ответ сервера по маршруту /api/articles/{id} (получение конкретной статьи) изображен на рисунке 3.53. Используя полученные в предыдущем запросе идентификаторы статей, фронтенд может запросить любую из них при соответствующих действиях пользователя. Ответ представляет собой html-документ.



```
GET https://localhost:44318/api/articles/7
Status: 200 OK Time: 745 ms Size: 8.65 KB
Body
Pretty Raw Preview Visualize HTML
1 <html>
2
3 <head>
4 <!--<link rel="stylesheet" type="text/css" href="/articles.css"-->
5 </head>
6
7 <body>
8 <div>
9 <h2 style="color:black">Эффективная отладка веб-приложения на примере React+Redux и ASP.NET Core</h2>
10 <div class="text">Последнее обновление: 21.05.2021</div>
11 <p align="justify" class="text">При создании веб-сервисов по принципу SPA, как правило, бэкенд и фронтенд
12 разрабатываются параллельно и независимо. Бэкенд в таких сервисах отвечает обычно за связь с базой данных и
13 за организацию обмена данными с фронтендом. Фронтенд в свою очередь отвечает за логику приложения, он
14 периодически запрашивает с сервера информацию, либо отправляет данные введенные пользователем.</p>
```

Рисунок 3.53 - Ответ сервера по маршруту /api/articles/{id}

Ответ сервера по маршруту /api/lessons/{topicName} (получение разделов конкретной темы с уроками) изображен на рисунке 3.54 и представляет собой массив объектов, каждый из которых описывает раздел и содержит информацию о названии раздела, идентификаторе раздела и идентификаторе темы, а также массив с уроками, каждый из которых, в свою очередь, хранит информацию о разделе, теме, названии урока и местонахождении самого урока.



```
GET https://localhost:44318/api/lessons/sharp
Status: 200 OK Time: 421 ms Size: 1.82 KB
Body
Pretty Raw Preview Visualize JSON
1 [
2   {
3     "id": 1,
4     "sectionName": "Введение в C#",
5     "topicId": 1,
6     "lessons": [
7       {
8         "id": 1,
9         "name": "Язык C# и платформа .NET Core",
10        "path": "/Files/Lessons/lesson1.1.1.html",
11        "sectionId": 1,
12        "sectionTopicId": 1
13      },
14      {
15        "id": 2,
16        "name": "Компиляция в командной строке с .NET Core CLI",
17        "path": "/Files/Lessons/lesson1.1.2.html",
18        "sectionId": 1,
19        "sectionTopicId": 1
20      }
21    ]
22  },
23 ]
```

Рисунок 3.54 - Ответ сервера по маршруту /api/lessons/{topicName}

Ответ сервера по маршруту /api/lessons/{topicName}/{sectionId}/{lessonId} (получение конкретного урока) изображен на рисунке 3.55. Также, как и в случае со статьями, используя полученные в предыдущем запросе идентификаторы темы, раздела и самой статьи, фронтенд может запросить любой из уроков. Ответ представляет собой html-документ.

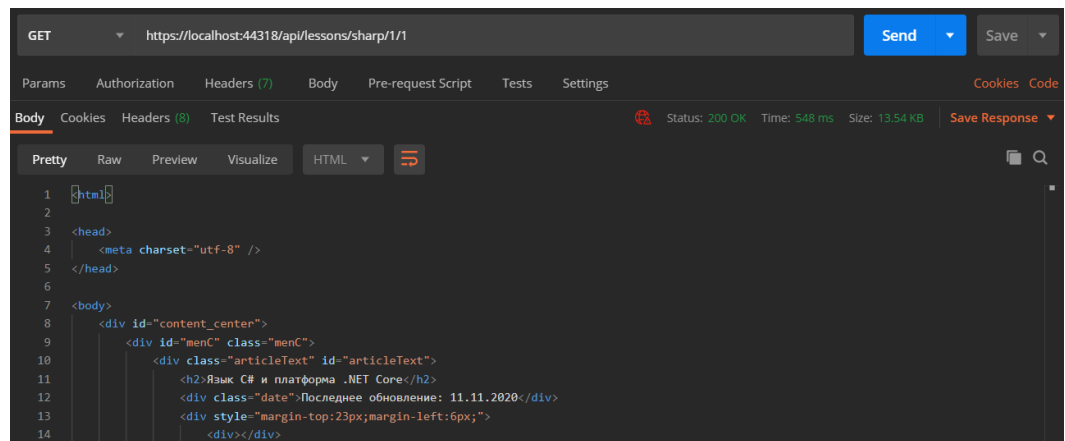


Рисунок 3.55 - Ответ сервера по маршруту /api/lessons/sharp/1/1

Доступ к остальным методам имеют только авторизованные пользователи, поэтому для их тестирования необходимо, чтобы запрос содержал аутентификационные куки. Для их установки необходимо симитировать отправку формы авторизации на соответствующий адрес (см. рисунок 3.56). После чего, сервер вернёт ответ с установленными cookies и можно будет продолжать тестирование.

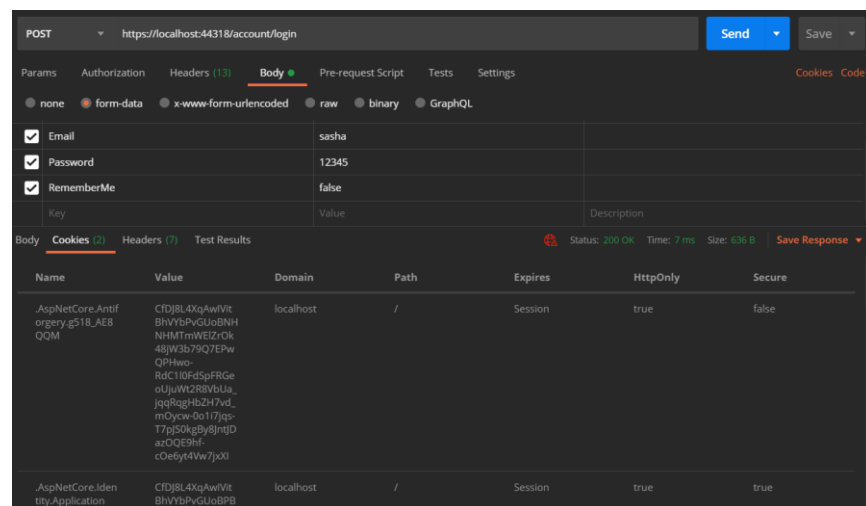


Рисунок 3.56 - Ответ сервера по маршруту /api/lessons/sharp/1/1

Ответ сервера по маршруту /api/notes (получение заметок пользователя) изображен на рисунке 3.57 и представляет собой массив объектов, каждый из которых описывает заметку пользователя и содержит информацию об идентификаторе заметки, заголовке и тексте.

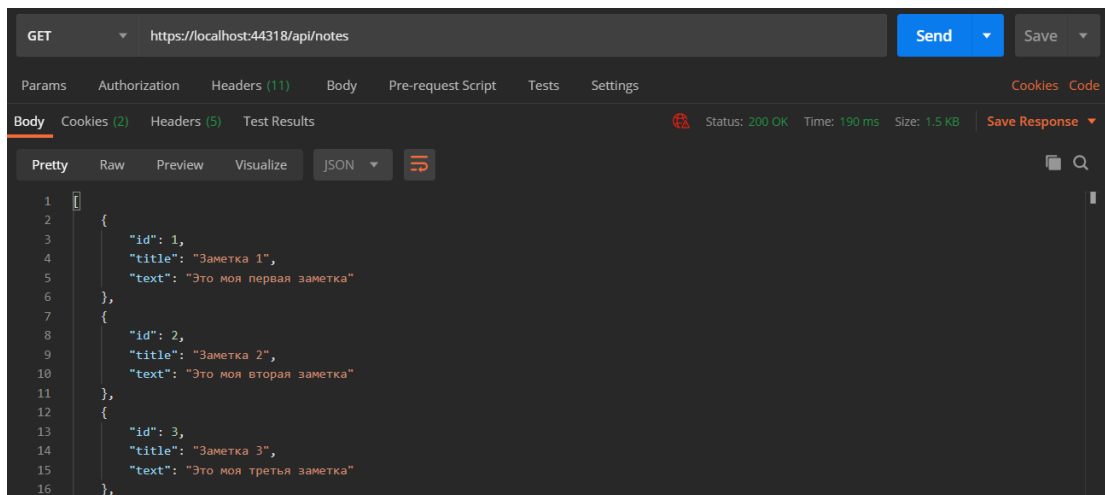


Рисунок 3.57 - Ответ сервера по маршруту /api/notes

Ответ сервера по маршруту /api/tests/{topicName} (получение тестов по соответствующей теме) изображен на рисунке 3.58 и представляет собой массив объектов, каждый из которых описывает тест и содержит информацию об идентификаторе теста, заголовке и прогрессе текущего пользователя, а также ссылку на изображение.

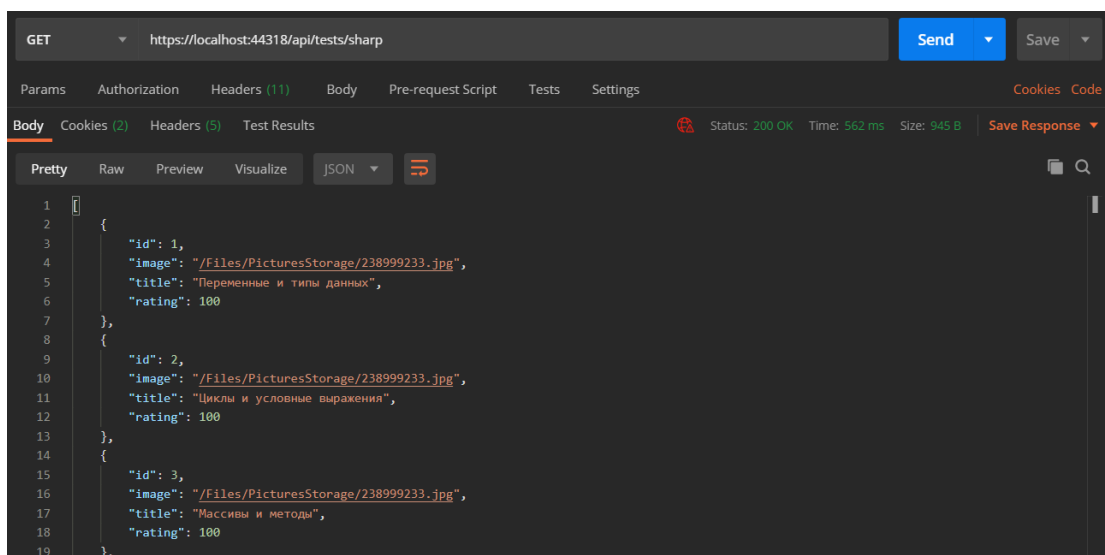


Рисунок 3.58 - Ответ сервера по маршруту /api/tests/sharp

Ответ сервера по маршруту /api/tests/{topicName}/{testId} (получение конкретного теста) изображен на рисунке 3.59 и представляет собой объект, содержащий свойства id и questions. Id – идентификатор теста, после прохождения теста он используется для обновления прогресса пользователя. Questions – массив вопросов, каждый из которых содержит сам вопрос, тип вопроса, варианты ответов и правильный ответ.

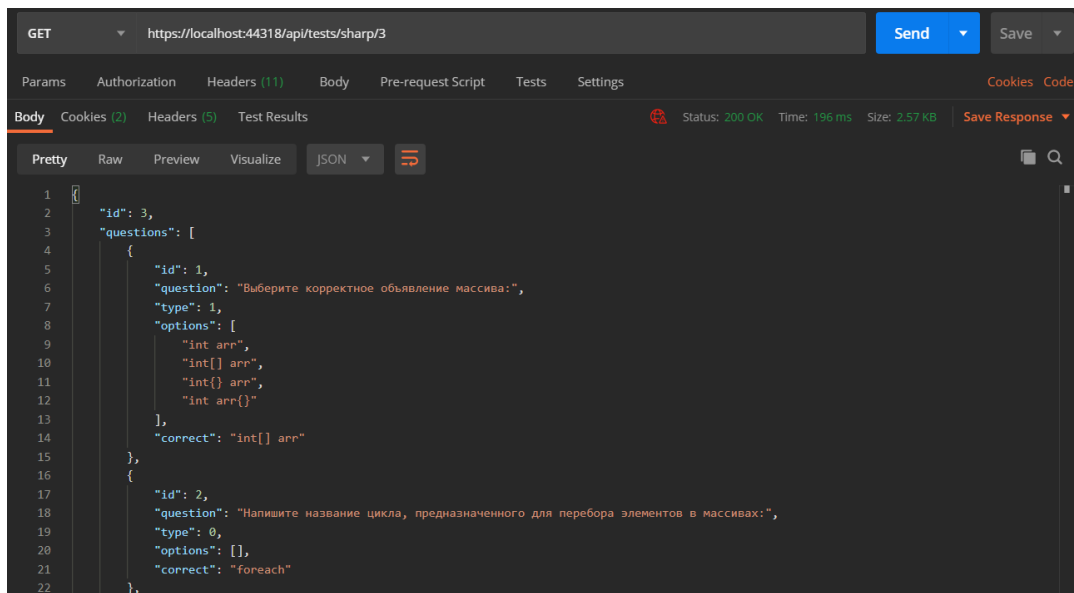


Рисунок 3.59 - Ответ сервера по маршруту /api/tests/sharp/3

Поскольку пользователю, под которым был выполнен вход, присвоена роль администратора, то можно также проверить корректность формирования данных на административной странице (см. рисунок 3.60).

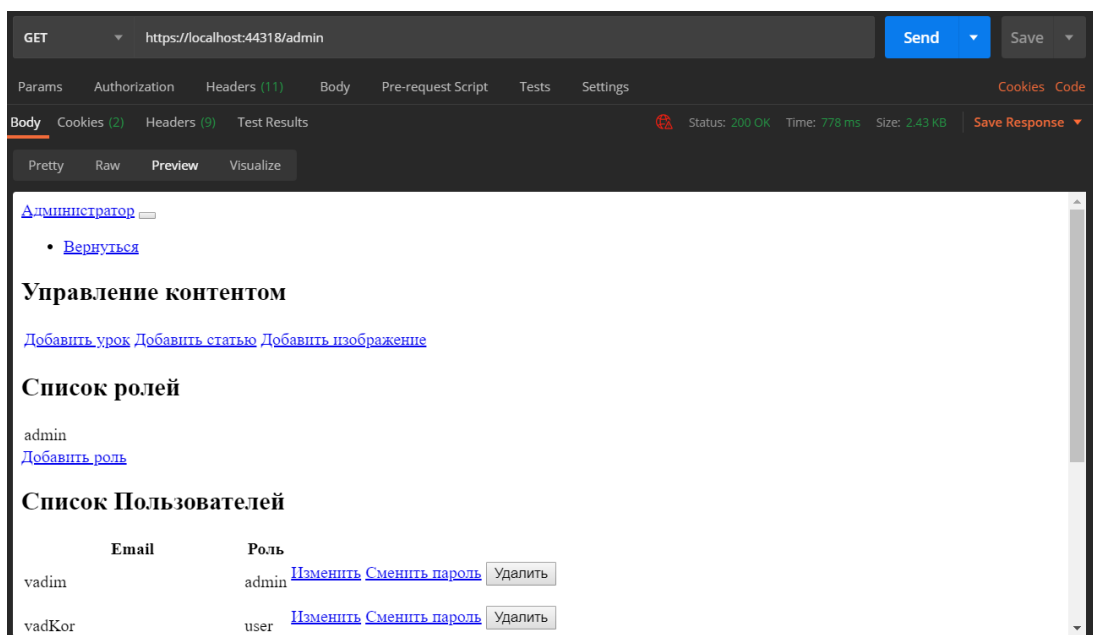


Рисунок 3.60 - Ответ сервера по маршруту /admin

3.5.2 Организация отладки взаимодействия клиентской и серверной частей

Поскольку большая часть данного сервиса работает по принципу SPA, то бэкенд и фронтенд разрабатывался параллельно и преимущественно независимо. Тем не менее для проверки и отладки взаимодействия необходимо периодически совмещать эти части и тестировать работоспособность. Самый простой способ это сделать – каждый раз собирать фронтенд-часть приложения, помещать готовые файлы в папку со статическими данными в проекте бэкенда и запускать сервер. После чего выполнять тестирование -

взаимодействовать с элементами, предполагающими обращение к серверу и с помощью точек останова, проверять корректность запросов и отлаживать код на сервере.

Очевидным минусом такого подхода является невозможность эффективной отладки фронтенда (как вариант, можно выводить в консоль требуемые данные, но это неудобно) т.к. при необходимости внести корректировки в код клиентской части - нужно заново перестраивать фронтент и производить манипуляции с файлами, что довольно неудобно.

Поэтому было принято решение использовать более эффективный подход, а именно запускать бэкенд и фронтенд в режиме разработки. Для сборки фронтенда используется webpack, а для отладки webpack-dev-server - специальный сервер для разработки, который не содержит никакой логики, однако умеет собирать приложение и открывать доступ к нему на определенном порту, кроме того он поддерживает функцию “горячей замены” т.е. возможность при изменении исходников очень быстро отобразить изменения в браузере без перезагрузки страницы.

Для того, чтобы фронтенд, запущенный на сервере-разработки webpack мог обмениваться данными с основным сервером ASP.Net Core необходимо его настроить, а именно включить перенаправление запросов по определенным маршрутам.

Все настройки сервера webpack производятся в файле webpack.config.js. в свойстве devServer. Правила перенаправления описываются в свойстве proxy (см. рисунок 3.61). Т.к. сервер ASP.NET Core разворачивался на порту 44318, туда и перенаправлялись запросы. Основная задача заключалась в отладке обмена данными, получении статических файлов, а также корректности запросов и правильности отображения интерфейса для авторизованного и не авторизованного пользователя, поэтому все запросы с префиксами /api, /files и /account перенаправлялись на бэкенд и обрабатывались соответствующими контроллерами.

```
devServer: {
  port: 8080,
  historyApiFallback: true,
  proxy: {
    //взаимодействие с основными SPA контроллерами
    '/api': {
      target: 'https://localhost:44318',
      secure: false
    },
    //возможность получать изображения, статьи и уроки (статические файлы)
    '/files': {
      target: 'https://localhost:44318',
      secure: false
    },
    //доступ к механизму аутентификации и авторизации
    '/account': {
      target: 'https://localhost:44318',
      secure: false
    },
  },
}
```

```
//возможность отслеживать работу приложения через исходные файлы  
devtool: 'source-map'
```

Рисунок 3.61 - Настройка dev-server в файле webpack.config.js

Для того, чтобы сервер бэкенда не блокировал запросы со стороннего домена, необходимо разрешить кроссдоменные запросы, для этого необходимо в методе Configure класса Startup (см. приложение Г) в конвейер обработки запросов были добавлены сервисы CORS (см. рисунок 3.62). Для упрощения настройки, запросы к нашему приложению были разрешены с любого адреса, любого типа и с любыми заголовками.

```
app.UseCors(options => options.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
```

Рисунок 3.62 - Добавление сервисов CORS в методе Configure класса Startup

Схема взаимодействия бэкенда и фронтенда при данном подходе изображена на рисунке 3.63.

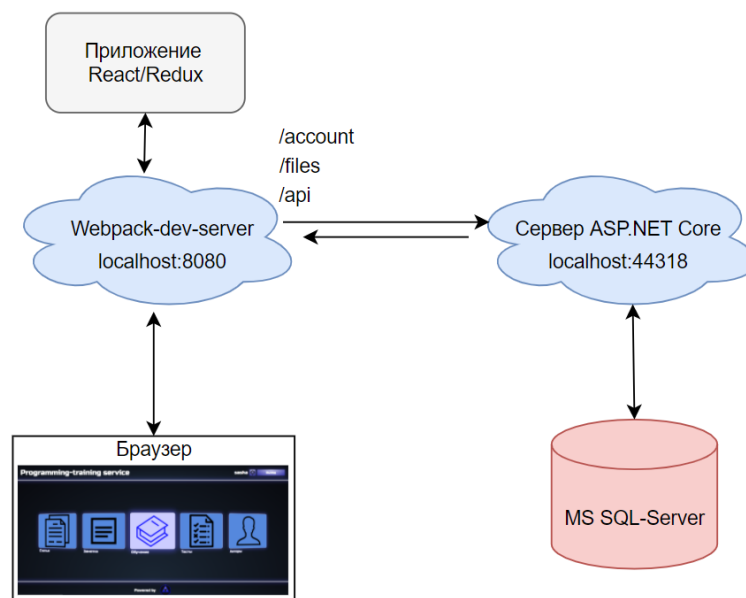


Рисунок 3.63 - Схема взаимодействия бэкенда и фронтенда в режиме отладки

Использование перенаправления запросов при одновременной отладке бэкенда и фронтенда значительно ускорило отладку, поскольку позволило в реальном времени взаимодействовать с интерфейсом фронтенда, отслеживать значения переменных, поэтапно проходить по коду и отправлять запросы на сервер, который в свою очередь обрабатывал запросы, имел сложную логику и доступ к базе данных и точно также, используя точки останова, отлаживать серверный код отслеживая данные полученные с клиента и данные посылаемые обратно.

3.6 Итоговый продукт

После отладки взаимодействия клиентской и серверной частей и тестирования на ошибки, разработка была завершена. Внешний вид главного меню представлен на рисунке 3.64. По умолчанию анонимному пользователю доступны разделы «Статьи», «Обучение» и «Авторы». После прохождения регистрации открывается доступ ко всем разделам. Это связано с тем, что заметки и результаты тестов привязываются к идентификатору пользователя.

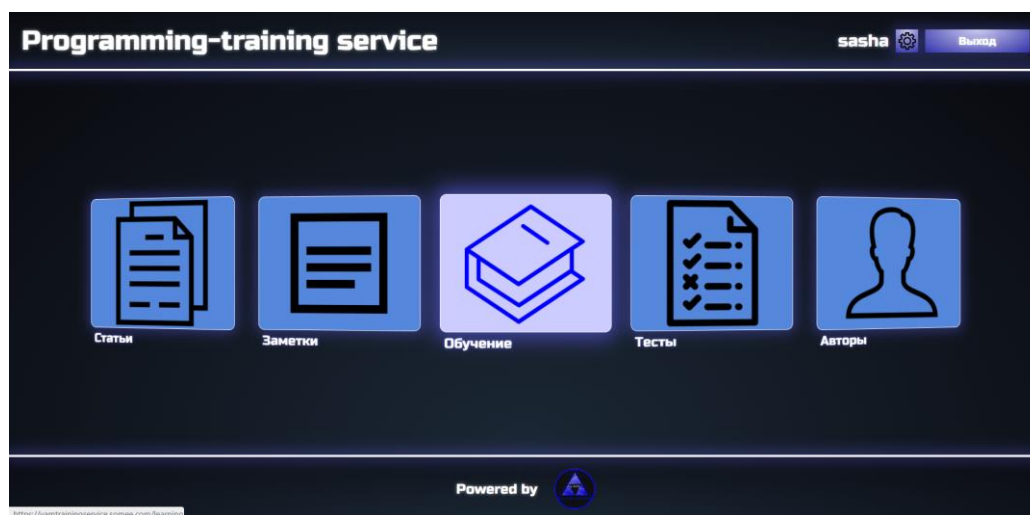


Рисунок 3.64 – Главное меню веб-сервиса

При входе в раздел «Обучение» предлагается выбрать направление (см. Рисунок 3.65)

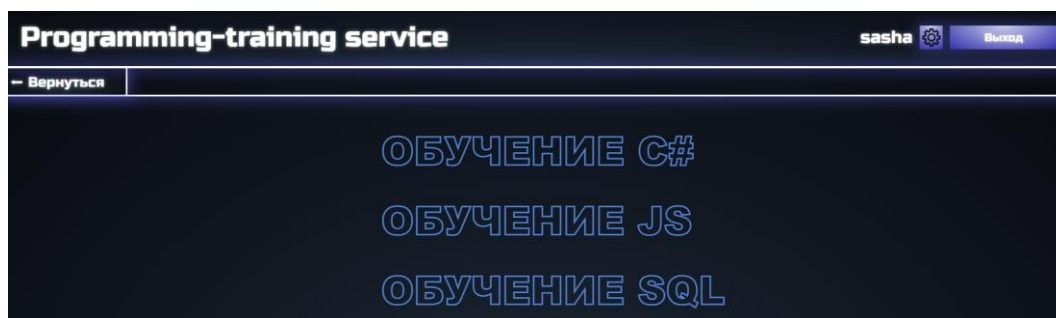


Рисунок 3.65 – Выбор направления обучения

После выбора открывается страница с рабочей областью, слева расположено меню с разделами и уроками, справа область с выбранным уроком (см. рисунок 3.66). Информация о списке разделов с уроками и сами уроки запрашиваются с сервера.

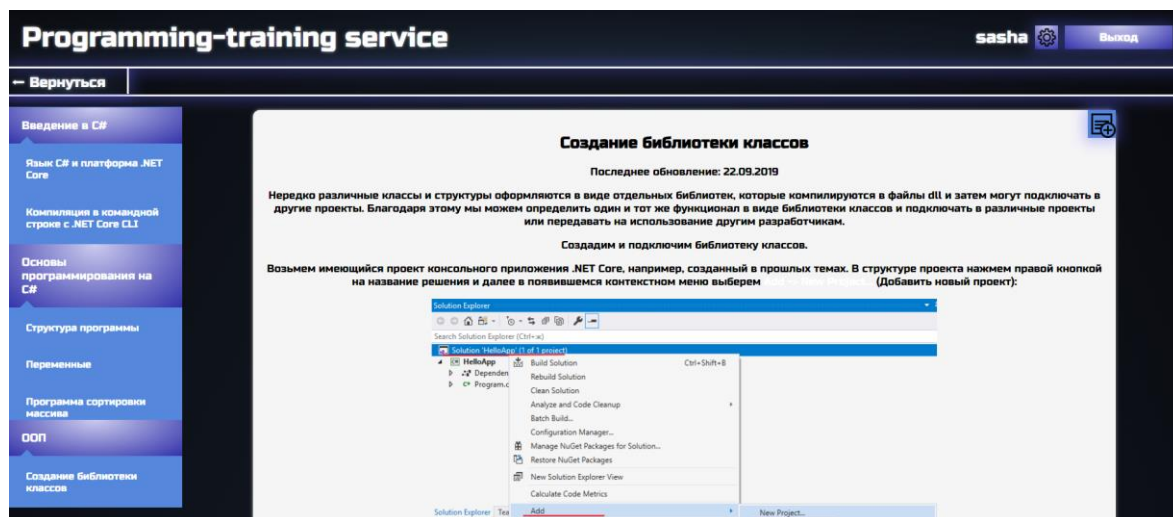


Рисунок 3.66 – Страница с разделами и уроками

При переходе в раздел «Тесты» алгоритм действий аналогичный. Сперва необходимо выбрать направление тестирования (см. рисунок 3.67)

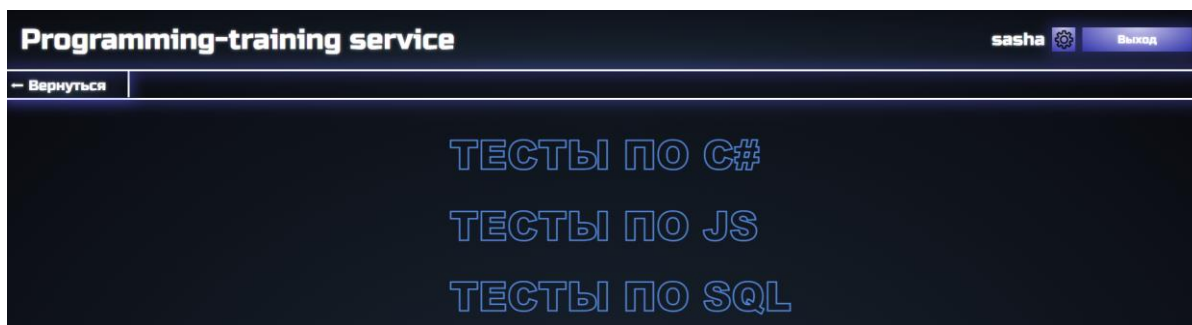


Рисунок 3.67 – Выбор направления тестирования

После чего открывается страница с тестами, принадлежащими данному направлению (см. рисунок 3.68). Данная информация также приходит с сервера.

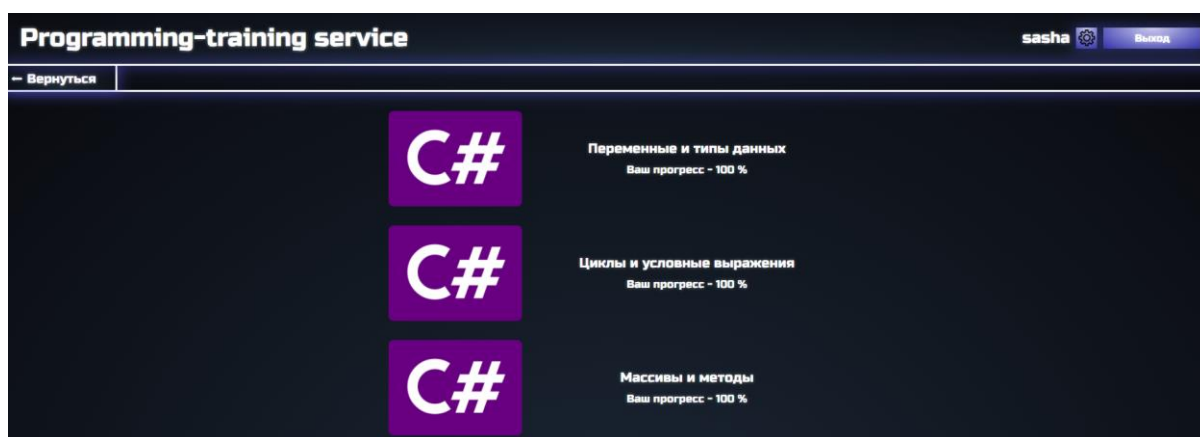


Рисунок 3.68 – Выбор теста

После выбора теста открывается страница с вопросами, вариантами ответов и кнопкой «Завершить» (см. рисунок 3.69).

Рисунок 3.69 – Страница с тестом

Внешний вид раздела «Статьи» приведен на рисунке 3.70. Список статей формируется после получения с сервера соответствующих данных. Каждая статья содержит заголовок, краткое описание, дату публикации и автора.

Рисунок 3.70 – Раздел «Статьи»

Авторизованным пользователям также доступны «Заметки» (см. рисунок 3.71). В процессе обучения или чтения статей пользователь может с помощью заметок записывать ключевую информацию или другие сведения, которые он посчитал интересными.

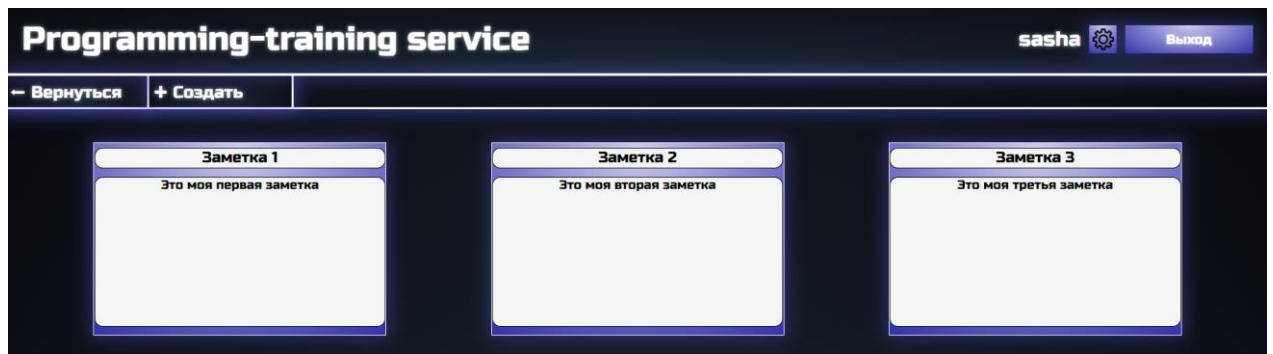


Рисунок 3.71 – Раздел «Заметки»

ЗАКЛЮЧЕНИЕ

В ходе данной работы была разработана серверная часть веб-сервиса для изучения программирования «Programming-training service». Созданный продукт отвечает всем поставленным требованиям.

В процессе разработки была спроектирована и развернута база данных, включающая все необходимые сущности. Также, был создан сам сервер, умеющий обрабатывать запросы, принимать и отправлять данные, подключаться к базе данных и т.д. По завершении разработки сервер был неоднократно протестирован с помощью соответствующего ПО. Кроме того, на финальном этапе работы было проведено объединение серверной части с клиентской, которое включало отладку обмена данными, объединение файлов двух проектов и сборку веб-приложения.

В дальнейшем планируется поддерживать и развивать проект, также в планах добавление новых функций, таких как возможность встраивания видеоплеера в статьи и уроки при необходимости или предоставление пользователям возможности самим составлять и добавлять статьи, которые на текущем этапе могут быть загружены только администратором.

Итоговый продукт был опубликован в сети интернет и в настоящее время (июнь 2021 года) доступен по адресу <https://vamtrainingservice.somee.com>

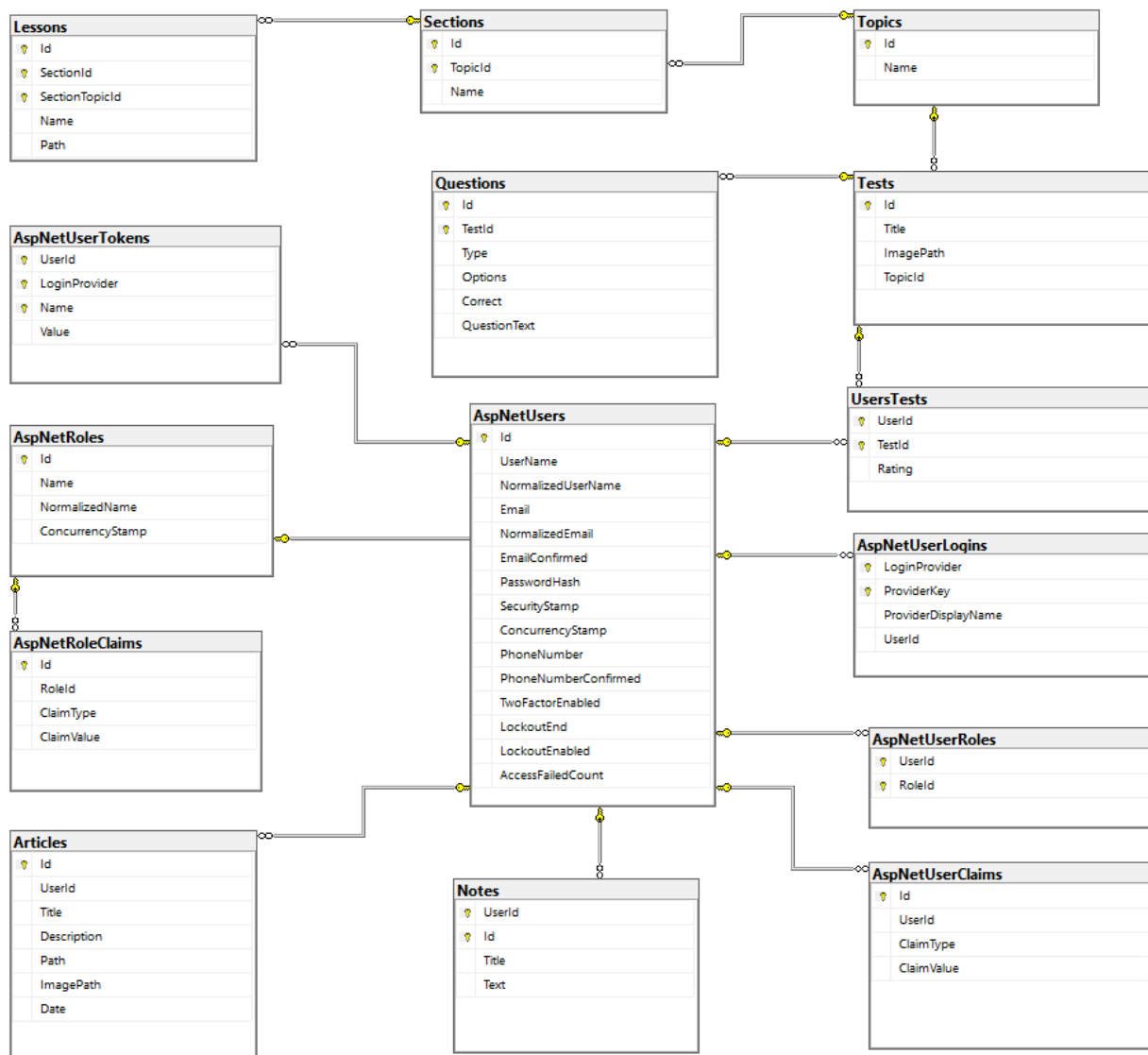
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Usage statistics of server-side programming languages for websites // W3Techs - World Wide Web Technology Surveys: сайт. – 2021. – URL: https://w3techs.com/technologies/overview/programming_language (дата обращения 15.05.2021)
2. Паттерн 'Репозиторий' в ASP.NET // Metanit : сайт. – 2015. – URL: <https://metanit.com/sharp/articles/mvc/11.php> (дата обращения 10.05.2021)
3. Добавление Identity в проект // Metanit : сайт. – 2021. – URL: <https://metanit.com/sharp/aspnet5/16.2.php> (дата обращения 26.04.2021)
4. Введение в Identity ASP.NET Core // Microsoft : официальный сайт. – 2020. – URL: <https://docs.microsoft.com/ru-ru/aspnet/core/security/authentication/identity> (дата обращения 08.05.2021)
5. Внедрение зависимостей в ASP.NET Core // Microsoft : официальный сайт. – 2020. – URL: <https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/dependency-injection> (дата обращения 15.04.2021)
6. Справочник по синтаксису Razor для ASP.NET Core // Microsoft : официальный сайт. – 2020. – URL: <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/views/razor> (дата обращения 10.05.2021)
7. Интеграция входа через Google в ваше веб-приложение // Google Developers : официальный сайт. – 2021. – URL: <https://developers.google.com/identity/sign-in/web/sign-in> (дата обращения 12.05.2021)
8. Проверка подлинности Facebook, Google и внешних поставщиков в ASP.NET Core // Microsoft : официальный сайт. – 2020. – URL: <https://docs.microsoft.com/ru-ru/aspnet/core/security/authentication/social> (дата обращения 12.05.2021)
9. Загрузка файлов на сервер // Metanit : сайт. – 2019. – URL: <https://metanit.com/sharp/aspnet5/21.3.php> (дата обращения 15.05.2021)
10. Postman Learning Center // Postman : официальный сайт. – 2021. – URL: <https://learning.postman.com/docs/getting-started/introduction/> (дата обращения 20.05.2021)

ПРИЛОЖЕНИЯ

Приложение А.

ER-диаграмма БД



Модели данных используемые в проекте

1. Модели сущностей БД

```

//Модель пользователя
public class User : IdentityUser
{
}

//Модель урока
public class Lesson
{
    public int Id { get; set; }
    [MaxLength(50)]
    public string Name { get; set; }
    public string Path { get; set; }
    public int SectionId { get; set; }
    public int SectionTopicId { get; set; }
}

//Модель раздела
public class Section
{
    public int Id { get; set; }
    public int TopicId { get; set; }
    public Topic Topic { get; set; }
    public string Name { get; set; }
    public List<Lesson> Lessons { get; set; }
}

//Модель темы
public class Topic
{
    public int Id { get; set; }
    [MaxLength(50)]
    public string Name { get; set; }
    public List<Section> Sections { get; set; }
}

//Модель заметки
public class Note
{
    public string UserId { get; set; }
    public User User { get; set; }
    public int Id { get; set; }
    [MaxLength(100)]
    public string Title { get; set; }
    public string Text { get; set; }
}

//Модель статьи
public class Article
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string UserId { get; set; }
    public User User { get; set; }

    public string Title { get; set; }
    public string Description { get; set; }
    [MaxLength(100)]
    public string Path { get; set; }
    [MaxLength(100)]
    public string ImagePath { get; set; }
    public string Date { get; set; }
}

```

```

//Модель теста
public class Test
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public string Title { get; set; }
    public string ImagePath { get; set; }
    public int TopicId { get; set; }
    public Topic Topic { get; set; }
    public List<Question> Questions { get; set; }
}

//Модель вопроса
public class Question
{
    public int Id { get; set; }
    public int TestId { get; set; }
    public Test Test { get; set; }
    public string QuestionText { get; set; }
    public bool Type { get; set; }
    public string Options { get; set; }
    public string Correct { get; set; }
}

//Модель связи пользователя с тестом
public class UserTest
{
    public string UserId { get; set; }
    public User User { get; set; }
    public int TestId { get; set; }
    public Test Test { get; set; }
    public int Rating { get; set; }
}

```

2. Модели данных отправляемые на клиент (ResponsesModels)

```

//Модель ответа с информацией об уроке
public class ResponseLesson
{
    public int Id { get; set; }
    public string Name { get; set; }
}

//Модель ответа с информацией о разделе
public class ResponseSection
{
    public int Id { get; set; }
    public string SectionName { get; set; }

    public int TopicId { get; set; }

    public List<Lesson> Lessons { get; set; }
}

//Модель ответа с информацией о авторизации пользователя
public class UserCheckOutResponse
{
    public bool IsAuthenticated { get; set; }
    public bool IsAdmin { get; set; }
    public string UserId { get; set; }

    public string UserName { get; set; }
}

//Модель ответа с информацией о заметке
public class NoteResponse
{
    public int Id { get; set; }
    public string Title { get; set; }
}

```

```

        public string Text{ get; set; }
    }
    //Модель ответа с информацией о тесте
    public class TestResponse
    {
        public int Id { get; set; }
        public string Image { get; set; }
        public string Title { get; set; }
        public int Rating { get; set; }
    }
    //Модель ответа с информацией о тесте с вопросами
    public class TestWithQuestionsResponse
    {
        public int Id { get; set; }
        public List<QuestionResponse> Questions { get; set; }
    }
    //Модель ответа с информацией о вопросе
    public class QuestionResponse
    {
        public int Id { get; set; }
        public string Question { get; set; }
        public int Type { get; set; }
        public List<string> Options { get; set; }
        public string Correct { get; set; }
    }
    //Модель ответа с информацией о статье
    public class ArticleResponse
    {
        public int Id { get; set; }
        public string Image { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public string Date { get; set; }
        public string Author { get; set; }
    }

```

3. Модели данных, принимаемых с клиентской части

```

//Модель данных запроса с информацией о новой заметке
public class NewNoteRequest : IValidatableObject
{
    [Required]
    public string Title { get; set; }
    [Required]
    public string Text { get; set; }
}
//Модель запроса с информацией о прогрессе пользователя
public class NewRatingRequest
{
    [Required]
    public int TestId { get; set; }
    [Required]
    public int Rating { get; set; }
}

```

Классы-репозитории, используемые для доступа к данным

```

//Репозиторий статей
public class SQLArticlesRepository : IArticleRepository
{
    private TrainingServiceContext db;
    //в конструкторе определяем контекст данных
    public SQLArticlesRepository (TrainingServiceContext _db)
    {
        db = _db;
    }
    //метод получения статьи по идентификатору
    public Article GetArticle (int articleId)
    {
        var result = db.Articles.FirstOrDefault(article => article.Id==articleId);
        return result;
    }
    //метод получения списка статей
    public List<ArticleResponse> GetArticlesList()
    {
        //выбираем все статьи из таблицы Articles
        return db.Articles.Select(article => article)
            //Объединяем по внешнему ключу с таблицей Users
            .Join(db.Users,
                article => article.UserId,
                user => user.Id,
                //преобразовываем данные в необходимый вид т.е. в экземпляр класса
                //ArticleResponse и превращаем в список
                (article, user) => new ArticleResponse {
                    Id = article.Id,
                    Image = article.ImagePath,
                    Title = article.Title,
                    Description = article.Description,
                    Date=article.Date,
                    Author=user.Email })
            .ToList();
    }
    //Метод добавления новой статьи
    public void AddArticle(Article newArticle)
    {
        db.Articles.Add(newArticle);
        db.SaveChanges();
    }
}

//Репозиторий уроков
public class SQLLessonsRepository : ILessonRepository
{
    private TrainingServiceContext context;
    //в конструкторе определяем контекст данных
    public SQLLessonsRepository(TrainingServiceContext _db) {
        context = _db;
    }
    //метод получения урока
    public Lesson GetLesson(string topicName, int sectionId, int lessonId)
    {
        var topic = context.Topics.FirstOrDefault(topic => topic.Name ==
topicName);
        if (topic == null) return null;
        int topicId = topic.Id;

```



```

        var result = new Lesson();
        result = context.Lessons.FirstOrDefault(Lesson => Lesson.Id == lessonId &&
Lesson.SectionId == sectionId && Lesson.SectionTopicId == topicId);
        return result;
    }
    //Метод получения списка уроков с разделами
    public List<ResponseSection> GetLessonsList(string topicName)
    {
        //по имени темы находим id
        var topic = context.Topics.FirstOrDefault(topic => topic.Name ==
topicName);
        if (topic == null) return null;
        int topicId = topic.Id;
        //получаем все разделы этой темы и собираем данные каждого раздела в класс
ResponseSection, после чего формируем список
        return context.Sections.Where(section => section.TopicId == topicId)
            .Select(section => new ResponseSection { TopicId =
section.TopicId, Id = section.Id, SectionName = section.Name, Lessons = section.Lessons
})
            .ToList();
    }
    //метод получения следующего идентификатора урока для темы и раздела
    public int GetNewLessonId(int topicId, int sectionId)
    {
        var lessonsList = context.Lessons.Where(Lesson => Lesson.SectionId ==
sectionId && Lesson.SectionTopicId == topicId).ToList();
        return (lessonsList.Count() == 0) ? 1 : lessonsList.Last().Id+1;
    }
    //метод добавления урока
    public void AddLesson(Lesson newLesson)
    {
        context.Lessons.Add(newLesson);
        Save();
    }
    public void Save()
    {
        context.SaveChanges();
    }
}

//репозиторий заметок
public class SQLNotesRepository : INoteRepository
{
    private TrainingServiceContext context;

    public SQLNotesRepository(TrainingServiceContext _db)
    {
        context = _db;
    }
    //метод получения заметок пользователя
    public List<NoteResponse> GetUserNotes(string userId)
    {
        return context.Notes.Where(note => note.UserId==userId)
            .Select(note => new NoteResponse {
Id=note.Id, Title=note.Title, Text=note.Text})
            .ToList();
    }
    //метод добавления заметки
    public void AddNote(Note newNote)
    {
        context.Notes.Add(newNote);
        Save();
    }
    //метод ролучения нового идентификатора заметки текущего пользователя
    public int GetNewUserNoteId(string userId)

```

```

    {
        var userNotesList = context.Notes.Where(Note => Note.UserId==userId).ToList();
        return (userNotesList.Count() == 0) ? 1 : userNotesList.Last().Id+1;
    }
    //метод удаления заметки
    public void DeleteNote(string userId, int noteId)
    {
        Note noteToRemove = context.Notes
            .Where(note => note.Id == noteId && note.UserId == userId)
            .FirstOrDefault();
        if (noteToRemove == null) return;
        context.Notes.Remove(noteToRemove);
        Save();
    }
    public void Save()
    {
        context.SaveChanges();
    }
}

//репозиторий тестов
public class SQLTestsRepository : ITestRepository
{
    private TrainingServiceContext db;

    public SQLTestsRepository(TrainingServiceContext _db)
    {
        db = _db;
    }
    //метод получения всех тестов с прогрессом пользователя
    public List<TestResponse> GetTestsWithUserRating(string userId, string topicName)
    {
        //получаем Id темы по названию
        int topicId = db.Topics.FirstOrDefault(topic => topic.Name == topicName).Id;
        //создаем список объектов TestResponse, по умолчанию полю рейтинг присваиваем
        значение 0
        List<TestResponse> result = db.Tests.Where(test => test.TopicId==
topicId).Select(test => new TestResponse { Id = test.Id, Image = test.ImagePath, Title =
test.Title, Rating = 0 }).ToList();
        //если данные не найдены - возвращаем пустой список
        if (result == null) return result;
        //Получаем результаты всех тестов, пройденных пользователем
        List<UserTest> usersTestsList = db.UsersTests.Where(userTest =>
userTest.UserId==userId).ToList();
        //Для каждого теста пройденного пользователем
        foreach (var usertest in usersTestsList)
        {
            //ищем соответствующий тест в итоговом списке
            var test = result.Find(testResp => testResp.Id == usertest.TestId);
            if (test == null) continue;
            //меняем рейтинг с 0 на результат пользователя
            else test.Rating = usertest.Rating;
        }
        return result;
    }
    //метод получения теста с вопросами
    public TestWithQuestionsResponse GetTestWithQuestions(int testId)
    {
        //определяем символ по которому будет разбиваться строка
        string[] questionSeparator = { "&&&" };
        //выбираем вопросы данного теста
        var questions = db.Questions.Where(question => question.TestId==testId)
            //преобразуем данные в необходимый формат
            .Select(question => new QuestionResponse {

```

```

        Id = question.Id, Question=
question.QuestionText,
        Type= Convert.ToInt32(question.Type),
        Correct= question.Correct,
        //Преобразуем строку в массив вариантов
        Options =
question.Options.Split(questionSeparator,
System.StringSplitOptions.RemoveEmptyEntries).ToList()))
        .ToList();
        return new TestWithQuestionsResponse { Id = testId, Questions = questions };
    }
    //метод обновления прогресса пользователя
    public void UpdateRating (string userId, int testId, int newRating)
    {
        // пытаемся найти запись о предыдущем результате прохождения теста
        var userTest = db.UsersTests.FirstOrDefault(userTest => userTest.UserId ==
userId && userTest.TestId == testId);
        // если записи нет, т.е пользователь прошёл тест в первый раз – добавляем
        запись
        if (userTest == null)
        {
            db.UsersTests.Add(new UserTest { UserId = userId, TestId = testId, Rating
= newRating });
            Save();
            return;
        }
        // иначе обновляем существующую запись
        userTest.Rating = newRating;
        db.UsersTests.Update(userTest);
        Save();
    }
    public void Save()
    {
        db.SaveChanges();
    }
}

```

Класс Startup

```

public class Startup
{
    public IConfiguration Configuration { get; }
    //в конструкторе определяем объект конфигурации приложения
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public void ConfigureServices(IServiceCollection services)
    {
        //получаем строку подключения из файла appsettings.json
        string connection = Configuration.GetConnectionString("DefaultConnection");
        //добавляем контекст БД в качестве сервиса через механизм DI
        services.AddDbContext<TrainingServiceContext>(options =>
options.UseSqlServer(connection));
        //добавляем сервисы Identity
        services.AddIdentity<User, IdentityRole>(options => {
            options.Password.RequiredLength = 5; // минимальная длина
            options.Password.RequireNonAlphanumeric = false; // требуются ли не алфавитно-
цифровые символы
            options.Password.RequireLowercase = false; // требуются ли символы в нижнем
регистре
            options.Password.RequireUppercase = false; // требуются ли символы в верхнем
регистре
            options.Password.RequireDigit = false; // требуются ли цифры
            options.User.RequireUniqueEmail = false;
            options.User.AllowedUserNameCharacters =
"abcdefghijklmnopqrstuvwxyz@.1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // допустимые
символы
        }).AddEntityFrameworkStores<TrainingServiceContext>();
        // добавляем внешние сервисы авторизации
        services.AddAuthentication().AddGoogle(options =>
        {
            // указываем идентификатор приложения
            options.ClientId = Configuration.GetConnectionString("GoogleClientId");
            // указываем секретный ключ
            options.ClientSecret =
Configuration.GetConnectionString("GoogleClientSecret");
        }).AddVkontakte(options =>
        {
            options.ClientId = Configuration.GetConnectionString("VKontakteClientId");
            options.ClientSecret =
Configuration.GetConnectionString("VKontakteClientSecret");
            options.Scope.Add("email");
        });
        // добавляем сервисы MVC
        services.AddMvc(option => option.EnableEndpointRouting = false);
        // добавляем поддержку контроллеров с представлениями
        services.AddControllersWithViews();
        // добавляем поддержку контроллеров без представлений
        services.AddControllers();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
    }
}

```

```

// включение статических файлов
app.UseStaticFiles();
// включение маршрутизации
app.UseRouting();
// включение механизма аутентификации и авторизации
app.UseAuthentication();
app.UseAuthorization();
// разрешение кросс-доменных запросов
app.UseCors(options =>
options.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
// включение механизма MVC
app.UseMvc(routes =>
{
    routes.MapRoute(
        // определяем маршрут по умолчанию
        name: "default",
        template: "{controller=Home}/{action=Index}");
    // поддержка клиентской маршрутизации
    routes.MapSpaFallbackRoute("spa-fallback", new { controller = "Home", action
= "Index" });
});
}
}

```

Контроллер AdminController

```

[Route("[controller]")]
//доступ только у пользователей с ролью администратора
[Authorize(Roles = "admin")]
public class AdminController : Controller
{
    UserManager<User> _userManager;
    RoleManager<IdentityRole> _roleManager;
    // в конструкторе определяется менеджер ролей и пользователей
    public AdminController(TrainingServiceContext dbContext,
        RoleManager<IdentityRole> roleManager, UserManager<User> userManager)
    {
        _userManager = userManager;
        _roleManager = roleManager;
    }
    // Основной метод, возвращает представление с главным окном администратора
    [Route("")]
    public async Task<IActionResult> Index()
    {
        var users = _userManager.Users.ToList();
        var roles = _roleManager.Roles.ToList();
        var adminList = await _userManager.GetUsersInRoleAsync("admin");
        // создаем модель-представления с необходимыми данными
        IndexViewModel model = new IndexViewModel
        {
            Users = users,
            Roles = roles,
            AdminUsers = adminList,
        };
        return View(model);
    }
    [Route("[action]")]
    public IActionResult CreateUser() => View();
    // метод создания нового пользователя
    [Route("[action]")]
    [HttpPost]
    public async Task<IActionResult> CreateUser(CreateUserViewModel model)
    {
        //проверка валидности полученных данных
        if (ModelState.IsValid)
        {
            // создаем объект нового пользователя
            User user = new User { Email = model.Email, UserName = model.Email };
            // добавляем пользователя в БД
            var result = await _userManager.CreateAsync(user, model.Password);
            // в случае успехе возвращаемся на главную страницу администратора
            if (result.Succeeded)
            {
                return RedirectToAction("Index");
            }
            // в противном случае выводим сообщение об ошибке
            else
            {
                foreach (var error in result.Errors)
                {
                    ModelState.AddModelError(string.Empty, error.Description);
                }
            }
        }
        return View(model);
    }
}
[Route("[action]")]

```

```

public IActionResult CreateRole() => View();
// метод создания новой роли
[Route("[action]")]
[HttpPost]
public async Task<IActionResult> CreateRole(string name)
{
    // проверяем, что имя не пустое
    if (!string.IsNullOrEmpty(name))
    {
        // добавляем роль в БД
        IdentityResult result = await _roleManager.CreateAsync(new
IdentityRole(name));
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }
    }
    return View(name);
}

[Route("[action]")]
public async Task<IActionResult> Edit(string id)
{
    // получаем пользователя
    User user = await _userManager.FindByIdAsync(id);
    if (user != null)
    {
        // получем список ролей пользователя
        var userRoles = await _userManager.GetRolesAsync(user);
        var allRoles = _roleManager.Roles.ToList();
        EditUserViewModel model = new EditUserViewModel
        {
            Id = user.Id,
            Email = user.Email,
            UserRoles = userRoles,
            AllRoles = allRoles
        };
        // передаем данные в представление
        return View(model);
    }

    return NotFound();
}

// метод редактирования данных о пользователе
[Route("[action]")]
[HttpPost]
roles) public async Task<IActionResult> Edit(EditUserViewModel model, List<string>
{
    if (ModelState.IsValid)
    {
        // находим пользователя
        User user = await _userManager.FindByIdAsync(model.Id);
        // меняем данные
        if (user != null)
        {

```

```

        user.Email = model.Email;
        user.UserName = model.Email;
        var result = await _userManager.UpdateAsync(user);
        // изменяем информацию о ролях пользователя
        if (result.Succeeded)
        {
            var userRoles = await _userManager.GetRolesAsync(user);
            // получаем все роли
            var allRoles = _roleManager.Roles.ToList();
            // получаем список ролей, которые были добавлены
            var addedRoles = roles.Except(userRoles);
            // получаем роли, которые были удалены
            var removedRoles = userRoles.Except(roles);

            await _userManager.AddToRolesAsync(user, addedRoles);

            await _userManager.RemoveFromRolesAsync(user, removedRoles);
            return RedirectToAction("Index");
        }
        else
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty, error.Description);
            }
        }
    }
    return View(model);
}
// метод удаления пользователя
[Route("[action]")]
[HttpPost]
public async Task<ActionResult> DeleteUser(string id)
{
    User user = await _userManager.FindByIdAsync(id);
    if (user != null)
    {
        IdentityResult result = await _userManager.DeleteAsync(user);
    }
    return RedirectToAction("Index");
}

[Route("[action]")]
public async Task<IActionResult> ChangePassword(string id)
{
    User user = await _userManager.FindByIdAsync(id);
    if (user == null)
    {
        return NotFound();
    }
    ChangePasswordViewModel model = new ChangePasswordViewModel { Id = user.Id,
Email = user.Email };
    return View(model);
}
// метод изменения пароля пользователя
[Route("[action]")]
[HttpPost]
public async Task<IActionResult> ChangePassword(ChangePasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        User user = await _userManager.FindByIdAsync(model.Id);
        if (user != null)

```



```

        {
            // создаем объект проверяющий пароль на валидность
            var _passwordValidator =
HttpContext.RequestServices.GetService(typeof(IPasswordValidator<User>)) as
IPasswordValidator<User>;
            // создаем объект, вычисляющий хэш пароля
            var _passwordHasher =
HttpContext.RequestServices.GetService(typeof(IPasswordHasher<User>)) as
IPasswordHasher<User>;
            // проверяем пароль на соответствие всем требованиям
            IdentityResult result = await
_passwordValidator.ValidateAsync(_userManager, user, model.NewPassword);
            // в случае успеха находим хэш пароля и обновляем данные в БД
            if (result.Succeeded)
            {
                user.PasswordHash = _passwordHasher.HashPassword(user,
model.NewPassword);
                await _userManager.UpdateAsync(user);
                return RedirectToAction("Index");
            }
            else
            {
                ModelState.AddModelError(string.Empty, "Пользователь не найден");
            }
        }
        return View(model);
    }
    // следующие методы позволяют администратору управлять контентом сайта,
    перенаправляя его на соответствующие методы других контроллеров.
    [Route("[action]")]
    [HttpGet]
    public IActionResult AddPicture()
    {
        return View("~/Views/PicturesStorage/AddPicture.cshtml");
    }
    [Route("[action]")]
    [HttpGet]
    public IActionResult AddLesson()
    {
        return View("~/Views/Lessons/AddLesson.cshtml");
    }
    [Route("[action]")]
    [HttpGet]
    public IActionResult AddArticle()
    {
        return View("~/Views/Articles/AddArticle.cshtml");
    }
}

```