|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  РЕСПУБЛИКИ ТАДЖИКИСТАН | | | | |
| ХУДЖАНДСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ ТАДЖИКСКОГО ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА  имени академика М.С. Осими | | | | |
|  |  | |  | |
|  |  | | *“Утверждаю”* | |
|  |  | | *Зав. кафедрой* | |
|  |  | | | *Худойбердиев Х.А.* |
|  |  | |  | |
|  |  | | | *\_\_\_\_\_\_\_\_\_ “\_\_\_\_” \_\_\_ 2023 г.* |
|  |  | | | (подпись) |
| dpdtt_orig.png | | | | |
| **КОНТРОЛИРУЕМАЯ САМОСТОЯТЕЛЬНАЯ РАБОТА**  1, 2  **Часть № \_\_\_** | | | | |
|  | | | | |
| Предмет: |  | | | |
|  |  | | | |
| Кафедра: | *Программирование и информационные системы* | | | |
|  |  | | | |
| Факультет: | *Информатика и энергетика* | | | |
| Цикл: | *1* | полугодие 2, *2023 – 2024* у.г. | | |
|  |  | | | |
| Выполнила: |  | | | |
| Курс: |  | Группа: | 400101 рА | |
| Преподаватель: |  | | | |

Результат защиты 1:

|  |  |  |  |
| --- | --- | --- | --- |
| *№ попытки* | *Оценка* | *Дата защиты* | *Подпись преподавателя* |
|  |  |  |  |

Результат защиты 2:

|  |  |  |  |
| --- | --- | --- | --- |
| *№ попытки* | *Оценка* | *Дата защиты* | *Подпись преподавателя* |
|  |  |  |  |

Худжанд – 2024

**СОДЕРЖАНИЕ**

[1. ПРОЕКТИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ 3](#_Toc164866890)

[1.1. Описание предметной области 3](#_Toc164866891)

[1.2. Создание диаграммы классов 5](#_Toc164866892)

[1.3. Создание GitHub-репозитории проекта 8](#_Toc164866893)

[2. ВВЕДЕНИЕ В ASP.NET WEB API 11](#_Toc164866894)

[2.1. Организация структуры проекта по Clean Architecture 11](#_Toc164866895)

[2.2. Создание моделей приложения 13](#_Toc164866896)

[3. РАБОТА С БАЗОЙ ДАННЫХ 17](#_Toc164866897)

[3.1. Реализация архитектурного паттерна репозиторий 17](#_Toc164866898)

[3.2. Использование EntityFramework для работы с базой данных 19](#_Toc164866899)

[3.3. Миграция данных 21](#_Toc164866900)

[4. РЕАЛИЗАЦИЯ МЕТОДОВ КОНТРОЛЛЕРОВ ДЛЯ ОБРАБОТКИ HTTP-ЗАПРОСОВ 24](#_Toc164866901)

[4.1. Создание сервисов 24](#_Toc164866902)

[4.2. Создание DTO 27](#_Toc164866903)

[4.3. Маппинг объектов с использованием библиотеки AutoMapper 29](#_Toc164866904)

[4.4. Создание методов контроллеров для CRUD операций 32](#_Toc164866905)

[5. АУТЕНТИФИКАЦИЯ И АВТОРИЗАЦИЯ ПОЛЬЗОВАТЕЛЯ 35](#_Toc164866906)

[5.1. Аутентификация и авторизация c помощью ASP.NET Identity 35](#_Toc164866907)

[5.2. Тестирование endpoint-ов Identity в Swagger 37](#_Toc164866908)

[6. СОЗДАНИЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА С BLAZOR. РАБОТА С ДАННЫМИ В BLAZOR 40](#_Toc164866909)

[6.1. Создание прототипа клиентского приложения в Figma 40](#_Toc164866910)

[6.3. Привязка данных к компонентам 43](#_Toc164866911)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 47](#_Toc164866912)

1. Проектирование предметной области

1.1. Описание предметной области

Сайт для связи инвесторов со стартапами, основанный на структуре, представленной в изображении, может охватывать широкий спектр функций, направленных на содействие взаимодействию между инвесторами и перспективными начинающими компаниями.

**1. Профили стартапов:**

* **Подробная информация:** Профили стартапов должны содержать исчерпывающую информацию о компании, ее команде, продукте или услуге, бизнес-модели, финансовых показателях, планах на будущее и инвестиционных потребностях.
* **Мультимедийный контент:** Стартапы должны иметь возможность размещать видео, презентации и другие медиаматериалы, чтобы наглядно представить свой проект и его потенциал.
* **Обновления и новости:** Сайт должен предоставлять платформу для публикации новостей и обновлений о деятельности стартапов, позволяя инвесторам быть в курсе их прогресса и достижений.

**2. Поиск и фильтрация:**

* **Расширенный поиск:** Инвесторы должны иметь возможность искать стартапы по различным критериям, включая отрасль, стадию развития, местоположение, ключевые слова, сумму инвестиций и другие релевантные параметры.
* **Персонализированные рекомендации:** Сайт может использовать алгоритмы машинного обучения, чтобы предлагать инвесторам стартапы, соответствующие их интересам и инвестиционным целям.
* **Сохранение результатов поиска:** Инвесторы должны иметь возможность сохранять свои результаты поиска, чтобы легко возвращаться к ним в будущем.

**3. Инструменты для связи:**

* **Безопасная коммуникация:** Сайт должен обеспечивать безопасные и удобные каналы связи между инвесторами и стартапами, такие как электронная почта, чат и видеоконференции.
* **Система уведомлений:** Инвесторы должны получать уведомления о новых сообщениях, запросах на связь и других важных событиях, происходящих на платформе.
* **Инструменты для управления контактами:** Инвесторы должны иметь возможность создавать списки контактов, отслеживать ход взаимодействия со стартапами и вести историю коммуникации.

**4. Инструменты due diligence:**

* **Доступ к документам:** Инвесторам должен предоставляться доступ к финансовым отчетам, юридическим документам, презентациям и другим материалам, необходимым для проведения due diligence.
* **Инструменты для оценки:** Сайт может включать инструменты для оценки инвестиционных рисков и потенциала стартапов, помогая инвесторам принимать обоснованные решения.
* **Сравнительный анализ:** Инвесторы должны иметь возможность сравнивать различные стартапы по ключевым показателям, облегчая процесс выбора инвестиционных объектов.

**5. Сообщество и ресурсы:**

* **Форумы и группы:** Сайт может иметь форумы и группы, где инвесторы и стартапы могут общаться, делиться опытом, задавать вопросы и налаживать связи.
* **События:** Сайт может организовывать онлайн- и офлайн-мероприятия, такие как питч-сессии, конференции и семинары, для содействия нетворкингу и обмену знаниями.
* **Образовательные материалы:** Сайт может предоставлять доступ к статьям, блогам, вебинарам и другим информационным ресурсам, помогая инвесторам и стартапам повышать свою квалификацию и осведомленность в сфере инвестиций.

**6. Безопасность и конфиденциальность:**

* **Верификация:** Сайт должен иметь надежные механизмы верификации пользователей для защиты от мошенничества и обеспечения безопасности данных.
* **Шифрование:** Все данные, передаваемые на сайте, должны быть зашифрованы для защиты конфиденциальности информации.
* **Политика конфиденциальности:** Сайт должен иметь четкую и прозрачную политику конфиденциальности, защищающую личные данные пользователей.

1.2. Создание диаграммы классов

* **Стартап:** Представляет собой начинающую компанию, ищущую инвестиции.
* **Инвестор:** Представляет собой физическое или юридическое лицо, готовое инвестировать в стартапы.
* **Проект:** Представляет собой бизнес-идею или предложение, которое стартап хочет реализовать с помощью инвестиций.
* **Категория:** Представляет собой отрасль или область деятельности, к которой относится проект.
* **Этап:** Представляет собой стадию развития стартапа (например, идея, прототип, MVP).
* **Инвестиция:** Представляет собой сумму денег, которую инвестор вкладывает в стартап.
* **Пользователь:** Представляет собой любого человека, зарегистрированного на сайте.

**2. Описание классов:**

**Стартап:**

* **id:** Уникальный идентификатор стартапа
* **название:** Название стартапа
* **описание:** Краткое описание проекта
* **проблема:** Описание проблемы, которую решает проект
* **решение:** Описание предлагаемого решения
* **команда:** Информация о команде стартапа
* **этап:** Текущий этап развития стартапа
* **требуемая сумма:** Сумма инвестиций, необходимых для реализации проекта
* **категория:** Отрасль или область деятельности проекта
* **пользователь:** Пользователь, который создал профиль стартапа

**Инвестор:**

* **id:** Уникальный идентификатор инвестора
* **имя:** Имя инвестора
* **фамилия:** Фамилия инвестора
* **электронная почта:** Адрес электронной почты инвестора
* **пароль:** Пароль инвестора
* **описание:** Краткое описание инвестиционных интересов
* **портфель:** Список инвестиций инвестора
* **сайт:** Веб-сайт или профиль инвестора
* **пользователь:** Пользователь, который создал профиль инвестора

**Проект:**

* **id:** Уникальный идентификатор проекта
* **название:** Название проекта
* **описание:** Подробное описание проекта
* **проблема:** Описание проблемы, которую решает проект
* **решение:** Описание предлагаемого решения
* **команда:** Информация о команде проекта
* **этап:** Текущий этап развития проекта
* **требуемая сумма:** Сумма инвестиций, необходимых для реализации проекта
* **категория:** Отрасль или область деятельности проекта
* **стартап:** Стартап, которому принадлежит проект

**Категория:**

* **id:** Уникальный идентификатор категории
* **название:** Название категории

**Этап:**

* **id:** Уникальный идентификатор этапа
* **название:** Название этапа

**Инвестиция:**

* **id:** Уникальный идентификатор инвестиции
* **сумма:** Сумма инвестиции
* **дата:** Дата инвестиции
* **инвестор:** Инвестор, который сделал инвестицию
* **стартап:** Стартап, в который была сделана инвестиция
* **проект:** Проект, который был профинансирован инвестицией

**Пользователь:**

* **id:** Уникальный идентификатор пользователя
* **имя:** Имя пользователя
* **фамилия:** Фамилия пользователя
* **электронная почта:** Адрес электронной почты пользователя
* **пароль:** Пароль пользователя
* **тип:** Тип пользователя (инвестор или стартап)

**3. Отношения между классами:**

* **Стартап:**
  + **Содержит:** 0 или более проектов
* **Инвестор:**
  + **Делает:** 0 или более инвестиций
  + **Интересуется:** 0 или более категориями
* **Проект:**
  + **Принадлежит:** 1 стартапу
  + **Находится в:** 1 категории
  + **Получает:** 0 или более инвестиций
* **Категория:**
  + **Содержит:** 0 или более проектов
* **Этап:**
  + **Описывает:** 0 или более стартапов
* **Инвестиция:**
  + **Делается:** 1 инвестором
  + **Вкладывается в:** 1 стартап
  + **Финансирует:** 1 проект

В то время как определение классов и их отношений является основой диаграммы классов, вы также можете добавить атрибуты и методы для каждого класса, чтобы более подробно описать их функциональность.

1.3. Создание GitHub-репозитории проекта

Перейдите на сайт <https://github.com/> и зарегистрируйтесь, используя свой адрес электронной почты и пароль. Вам будет предложено создать имя пользователя и указать личную информацию.

**2.** **Создание нового репозитория:**

1. Войдите в свою учетную запись GitHub.
2. Нажмите кнопку "**New**" в правом верхнем углу экрана.
3. Выбрать "**Public**" или "**Private**" для вашего репозитория.
4. Введите название вашего репозитория (например, "investor-finder-website").
5. (Необязательно) Добавьте краткое описание вашего проекта.
6. Выбрать "**Initialize this repository with a README**", если хотите создать файл README по умолчанию.
7. Нажмите кнопку "**Create repository**".

**3. Инициализация локального репозитория:**

1. Откройте терминал или командную строку.
2. Перейдите в каталог, где вы хотите хранить локальные файлы вашего проекта.
3. Выполните команду:

git init

1. Добавьте все локальные файлы вашего проекта в систему контроля версий Git:

git add .

1. Зафиксируйте изменения с сообщением:

git commit -m "Initial commit"

**4. Связь локального репозитория с удаленным:**

1. Скопируйте URL-адрес вашего удаленного репозитория GitHub из адресной строки браузера.
2. В локальном репозитории выполните команду:

git remote add origin <URL-адреса-репозитория>

1. Загрузите локальные изменения на удаленный сервер:

git push -u origin master

**5. (Необязательно) Создание файла README:**

1. Создайте файл с именем README.md в корневом каталоге вашего проекта.
2. Добавьте информацию о вашем проекте, такую как:
   * Описание проекта
   * Инструкции по установке и использованию
   * Ссылки на документацию
   * Информацию о лицензировании

Создание GitHub-репозитория проекта обеспечивает централизованное хранилище для кода, способствует совместной работе команды разработчиков, упрощает отслеживание изменений и позволяет эффективно управлять жизненным циклом разработки.

2. Введение в ASP.NET Web API

2.1. Организация структуры проекта по Clean Architecture

**Clean Architecture** (Чистая архитектура) - это шаблон проектирования программного обеспечения, который способствует разработке модульного, тестируемого и простого в сопровождении кода.

**Применение Clean Architecture к ASP.NET Web API** позволяет:

* **Разделить код на независимые слои:**
  + **Сущности:** Модели данных, не зависящие от внешних факторов.
  + **Используемые случаи:** Бизнес-логика, реализующая функциональность приложения.
  + **Интерфейсы:** Определения границ между слоями, без реализации.
  + **Адаптеры:** Реализация взаимодействия между слоями.
  + **Внешняя зависимость:** Библиотеки, фреймворки и другие внешние компоненты.
* **Сделать код более тестируемым:**
  + Легко тестировать каждый слой независимо.
  + Использовать фикстуры для имитации внешних зависимостей.
* **Улучшить сопровождаемость кода:**
  + Четкая структура папок, отражающая архитектуру.
  + Легко найти и изменить код, относящийся к определенной функциональности.

**Структура проекта по Clean Architecture:**

src

├── Application

│ ├── Entities // Сущности

│ ├── UseCases // Используемые случаи

│ ├── Interfaces // Интерфейсы

│ └── Adapters // Адаптеры

├── Infrastructure

│ ├── Repositories // Репозитории

│ ├── Database // База данных

│ └── Logging // Логирование

└── WebApi // ASP.NET Web API

├── Controllers // Контроллеры

├── Filters // Фильтры

└── Startup // Настройка приложения

**Описание слоев:**

* **Entities:** Содержит модели данных, представляющие бизнес-объекты приложения.
* **UseCases:** Реализует бизнес-логику приложения, используя сущности и не завися от внешних факторов.
* **Interfaces:** Определяет границы между слоями, не содержая реализации.
* **Adapters:** Реализует взаимодействие между слоями, например, маппирование сущностей в запросы к базе данных.
* **Infrastructure:** Содержит классы, реализующие работу с инфраструктурой, такой как база данных, логирование и т.д.
* **WebApi:** Содержит контроллеры ASP.NET Web API, обрабатывающие HTTP-запросы и ответы.

**Преимущества использования Clean Architecture:**

* **Модульность:** Легко добавлять новые функции и изменять существующие без влияния на другие части кода.
* **Тестируемость:** Простота написания тестов для каждого слоя.
* **Поддержка:** Легкость в сопровождении и внесении изменений в код.
* **Независимость от фреймворков:** Бизнес-логика не зависит от конкретного фреймворка ASP.NET.

**Следование принципам Clean Architecture при разработке ASP.NET Web API позволяет создавать надежные, масштабируемые и простые в сопровождении приложения.**

**Важно отметить:**

* Clean Architecture - это лишь один из шаблонов проектирования, и его применение может варьироваться в зависимости от конкретного проекта.
* Необязательно строго следовать описанной выше структуре папок.
* Главное - понимать принципы Clean Architecture и применять их таким образом, чтобы улучшить качество кода.

**2.2.** **Создание моделей приложения**

**Описание моделей:**

Для каждой сущности необходимо создать класс-модель, который будет представлять ее в коде.

**Пример модели стартапа:**

public class Startup

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public string Problem { get; set; }

public string Solution { get; set; }

public string Team { get; set; }

public Stage Stage { get; set; }

public int RequiredAmount { get; set; }

public Category Category { get; set; }

public User User { get; set; }

}

**Связи между моделями:**

Модели могут быть связаны между собой с помощью отношений один к одному, один ко многим или многие ко многим.

**Пример отношений между моделями:**

* **Стартап:**
  + **Один к одному:** Пользователь (создатель стартапа)
  + **Один ко многим:** Проекты
* **Инвестор:**
  + **Один к одному:** Пользователь (создатель профиля инвестора)
  + **Многие ко многим:** Стартапы (через инвестиции)
  + **Многие ко многим:** Категории (интересы)
* **Проект:**
  + **Один к одному:** Стартап (компания, владеющая проектом)
  + **Один к одному:** Категория

**Дополнительные атрибуты:**

В зависимости от требований проекта, в модели можно добавить дополнительные атрибуты.

**Пример:**

* **Стартап:**
  + Дата создания
  + Ссылка на веб-сайт
  + Финансовые показатели
  + Местоположение
* **Инвестор:**
  + Опыт инвестирования
  + Предпочитаемая сумма инвестиций
  + Стратегия инвестирования
* **Проект:**
  + План развития
  + Команда проекта
  + Статус проекта
* **Категория:**
  + Родительская категория (для иерархической структуры)
* **Этап:**
  + Описание этапа
  + Критерии перехода на следующий этап
* **Инвестиция:**
  + Статус инвестиции (ожидается, подтверждена, отменена)
  + Условия инвестиции
* **Пользователь:**
  + Дата регистрации
  + Роль пользователя (администратор, модератор)

**Валидация данных:**

Для обеспечения целостности данных необходимо использовать методы валидации в моделях.

**Пример:**

public class Startup

{

// ... другие свойства

public bool IsValid()

{

if (string.IsNullOrEmpty(Name))

{

return false;

}

if (string.IsNullOrEmpty(Description))

{

return false;

}

// ... другие проверки

return true;

}

}

**Использование DTO (Data Transfer Objects):**

В некоторых случаях может быть целесообразно использовать DTO для передачи данных между слоями приложения.

**Пример DTO для стартапа:**

public class StartupDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public string CategoryName { get; set; }

public string StageName { get; set; }

}

**Заключение:**

Создание моделей приложения является важным этапом разработки.

* Четко определенные модели позволяют улучшить структуру кода и сделать его более понятным.
* Использование валидации данных обеспечивает целостность информации.
* DTO могут упростить передачу данных между слоями приложения.

3. Работа с базой данных

3.1. Реализация архитектурного паттерна репозиторий

**Понятие репозитория:**

Паттерн репозитория - это архитектурный шаблон, который обеспечивает абстракцию над слоем доступа к данным.

**Цель использования репозитория:**

* **Скрыть реализацию доступа к данным:** Репозиторий позволяет разработчикам работать с данными, не зная о том, как они хранятся (в базе данных, файлах и т.д.).
* **Упростить код:** Использование репозитория делает код более читаемым, понятным и простым в сопровождении.
* **Обеспечить унифицированный доступ к данным:** Репозиторий обеспечивает единый интерфейс для доступа к данным из разных частей приложения.

**Реализация репозитория в ASP.NET Web API:**

В ASP.NET Web API для реализации репозитория можно использовать различные подходы:

* **Использование Entity Framework Core:** Entity Framework Core (EF Core) - это фреймворк ORM (Object-Relational Mapping), который упрощает работу с базами данных. EF Core позволяет создавать репозитории, автоматически генерируя код для доступа к данным.
* **Использование собственных классов репозитория:** Можно написать собственные классы репозитория, используя ADO.NET или другие технологии доступа к данным.

**Пример реализации репозитория с помощью Entity Framework Core:**

public interface IStartupRepository

{

Startup GetById(int id);

IEnumerable<Startup> GetAll();

void Add(Startup startup);

void Update(Startup startup);

void Delete(int id);

}

public class StartupRepository : IStartupRepository

{

private readonly ApplicationDbContext \_context;

public StartupRepository(ApplicationDbContext context)

{

\_context = context;

}

public Startup GetById(int id)

{

return \_context.Startups.FirstOrDefault(s => s.Id == id);

}

public IEnumerable<Startup> GetAll()

{

return \_context.Startups.ToList();

}

public void Add(Startup startup)

{

\_context.Startups.Add(startup);

\_context.SaveChanges();

}

public void Update(Startup startup)

{

\_context.Startups.Update(startup);

\_context.SaveChanges();

}

public void Delete(int id)

{

var startup = \_context.Startups.FirstOrDefault(s => s.Id == id);

if (startup != null)

{

\_context.Startups.Remove(startup);

\_context.SaveChanges();

}

}

}

**Преимущества использования репозитория:**

* **Снижение зависимости от технологии доступа к данным:** Репозиторий позволяет легко заменить одну технологию доступа к данным другой, не влияя на код приложения.
* **Улучшение тестируемости:** Легко тестировать репозиторий, изолируя его от остального кода.
* **Повышение гибкости:** Репозиторий позволяет легко добавлять новые функции, связанные с доступом к данным.

**Применение репозитория:**

* Репозиторий используется в сервисах приложения для работы с данными.
* Контроллеры ASP.NET Web API взаимодействуют с сервисами, а не напрямую с репозиторием.
* Это обеспечивает разделение ответственности и соблюдение принципов Clean Architecture.

3.2. Использование EntityFramework для работы с базой данных

1. **Установка Entity Framework Core:**
   * Добавьте пакет Entity Framework Core NuGet в ваш проект ASP.NET Web API.
2. **Создание контекста данных:**
   * Создайте класс контекста данных, который наследуется от DbContext.
   * Определите в классе контекста данных DbSet для каждой сущности, с которой будет работать приложение.
3. **Регистрация контекста данных:**
   * В методе Startup класса Startup зарегистрируйте контекст данных в системе зависимостей ASP.NET Core.

**Миграции базы данных:**

1. **Создание миграций:**
   * Используйте команду dotnet ef migrations add для создания миграций базы данных.
   * Миграции автоматически генерируют код SQL, необходимый для создания или обновления схемы базы данных.
2. **Применение миграций:**
   * Используйте команду dotnet ef database update для применения миграций к базе данных.
   * Это создаст или обновит таблицы базы данных в соответствии с текущей моделью сущностей.

**Работа с данными с помощью Entity Framework Core:**

* **Доступ к данным:**
  + Используйте контекст данных для получения, добавления, обновления и удаления сущностей из базы данных.
  + EF Core автоматически преобразует операции с сущностями в команды SQL.
* **Запросы к данным:**
  + Используйте метод Where для фильтрации данных.
  + Используйте методы OrderBy и OrderByDescending для сортировки данных.
  + Используйте методы Select и Include для проекции данных.
  + EF Core поддерживает LINQ-запросы для работы с данными.

**Пример использования Entity Framework Core:**

using (var context = new ApplicationDbContext())

{

// Получение стартапа по идентификатору

var startup = context.Startups.FirstOrDefault(s => s.Id == 1);

// Добавление нового стартапа

var newStartup = new Startup

{

Name = "Новый стартап",

Description = "Описание нового стартапа",

// ... другие свойства

};

context.Startups.Add(newStartup);

context.SaveChanges();

// Обновление информации о стартапе

startup.Description = "Обновленное описание";

context.SaveChanges();

// Удаление стартапа

context.Startups.Remove(startup);

context.SaveChanges();

// Получение списка всех стартапов

var startups = context.Startups.ToList();

// Получение стартапов по категории

var startupsByCategory = context.Startups

.Where(s => s.Category.Id == 1)

.ToList();

}

**Преимущества использования Entity Framework Core:**

* **Абстракция от SQL:** EF Core позволяет работать с данными без необходимости писать SQL-запросы.
* **Отслеживание изменений:** EF Core автоматически отслеживает изменения в сущностях и обновляет базу данных.
* **Валидация данных:** EF Core позволяет выполнять валидацию данных на стороне сервера.
* **Поддержка транзакций:** EF Core поддерживает транзакции для обеспечения согласованности данных.

3.3. Миграция данных

**Оценка необходимости миграции:**

Перед началом миграции данных необходимо оценить ее необходимость.

**Факторы, влияющие на необходимость миграции:**

* **Структура данных:** Если структура данных в новой системе существенно отличается от старой, может потребоваться миграция.
* **Объем данных:** Если объем данных большой, миграция может быть трудоемкой и требовать тщательного планирования.
* **Совместимость данных:** Если данные в старой системе несовместимы с новой, их необходимо преобразовать перед миграцией.
* **Зависимости данных:** Если данные в разных системах взаимосвязаны, миграцию необходимо выполнять согласованно, чтобы избежать потери данных.

**Выбор стратегии миграции:**

Существует несколько стратегий миграции данных:

* **Прямая миграция:** Данные переносятся из старой системы в новую без промежуточных этапов.
* **Поэтапная миграция:** Данные мигрируются по частям, например, по модулям или таблицам.
* **Параллельная миграция:** Новая система работает параллельно со старой, пока не будет выполнена полная миграция данных.

**Выбор стратегии миграции зависит от:**

* Срока, выделенного на миграцию
* Доступных ресурсов
* Сложности миграции
* Влияния миграции на пользователей

**Планирование миграции:**

План миграции должен включать:

* **Определение целей и задач миграции.**
* **Анализ существующих данных.**
* **Выбор стратегии миграции.**
* **Разработку плана действий.**
* **Тестирование миграции.**
* **Внедрение миграции.**
* **Отслеживание результатов миграции.**

**Инструменты для миграции данных:**

Существует множество инструментов для миграции данных, как коммерческих, так и бесплатных.

**Выбор инструмента зависит от:**

* Объема и сложности данных
* Навыков разработчиков
* Бюджета проекта

**Пример поэтапной миграции:**

1. **Экспорт данных из старой системы:** Данные экспортируются из старой системы в формат, удобный для обработки.
2. **Преобразование данных:** Данные преобразуются в формат, совместимый с новой системой.
3. **Загрузка данных в новую систему:** Данные загружаются в новую систему.
4. **Тестирование миграции:** Проводится тестирование миграции для проверки целостности и корректности данных.
5. **Внедрение миграции:** Новая система вводится в эксплуатацию, а старая система выводится из эксплуатации.

**Заключение:**

Миграция данных - это важный этап при внедрении новой системы.

4. Реализация методов контроллеров для обработки HTTP-запросов

4.1. Создание сервисов

**Назначение сервисов:**

Сервисы приложения - это классы, которые инкапсулируют бизнес-логику и взаимодействуют с репозиториями для доступа к данным.

**Преимущества использования сервисов:**

* **Модульность:** Сервисы позволяют разделить код на модули, что делает его более понятным и простым в сопровождении.
* **Повторное использование:** Логику, реализованную в сервисах, можно повторно использовать в разных частях приложения.
* **Тестируемость:** Сервисы легко тестировать изолированно от контроллеров и других компонентов.
* **Разделение ответственности:** Сервисы фокусируются на бизнес-логике, а контроллеры - на обработке HTTP-запросов и ответов.

**Примеры сервисов:**

* **Сервис стартапов:**
  + Получение списка стартапов
  + Получение информации о стартапе по ID
  + Добавление нового стартапа
  + Обновление информации о стартапе
  + Удаление стартапа
* **Сервис инвесторов:**
  + Получение списка инвесторов
  + Получение информации об инвесторе по ID
  + Добавление нового инвестора
  + Обновление информации об инвесторе
  + Удаление инвестора
* **Сервис проектов:**
  + Получение списка проектов
  + Получение информации о проекте по ID
  + Добавление нового проекта
  + Обновление информации о проекте
  + Удаление проекта
* **Сервис категорий:**
  + Получение списка категорий
  + Получение информации о категории по ID
  + Добавление новой категории
  + Обновление информации о категории
  + Удаление категории
* **Сервис этапов:**
  + Получение списка этапов
  + Получение информации об этапе по ID
  + Добавление нового этапа
  + Обновление информации об этапе
  + Удаление этапа
* **Сервис инвестиций:**
  + Получение списка инвестиций
  + Получение информации об инвестиции по ID
  + Добавление новой инвестиции
  + Обновление информации об инвестиции
  + Удаление инвестиции

**Реализация сервисов:**

* Сервисы должны быть простыми, понятными и лаконичными.
* Сервисы должны инкапсулировать бизнес-логику и не должны напрямую взаимодействовать с HTTP-запросами или ответами.
* Сервисы должны использовать репозитории для доступа к данным.
* Сервисы должны возвращать DTO (Data Transfer Objects) для передачи данных между слоями приложения.

**Пример реализации сервиса стартапов:**

C#

public class StartupService

{

private readonly IStartupRepository \_startupRepository;

public StartupService(IStartupRepository startupRepository)

{

\_startupRepository = startupRepository;

}

public IEnumerable<StartupDTO> GetStartups()

{

var startups = \_startupRepository.GetAll();

return startups.Select(s => MapToDTO(s));

}

public StartupDTO GetStartupById(int id)

{

var startup = \_startupRepository.GetById(id);

if (startup != null)

{

return MapToDTO(startup);

}

return null;

}

public void AddStartup(StartupDTO startupDTO)

{

var startup = MapFromDTO(startupDTO);

\_startupRepository.Add(startup);

}

public void UpdateStartup(StartupDTO startupDTO)

{

var startup = MapFromDTO(startupDTO);

\_startupRepository.Update(startup);

}

public void DeleteStartup(int id)

{

\_startupRepository.Delete(id);

}

private StartupDTO MapToDTO(Startup startup)

{

return new StartupDTO

{

Id = startup.Id,

Name = startup.Name,

Description = startup.Description,

CategoryName = startup.Category.Name,

StageName = startup.Stage.Name

};

}

private Startup MapFromDTO(StartupDTO startupDTO)

{

return new Startup

{

Id = startupDTO.Id,

Name

4.2. Создание DTO

**Назначение DTO:**

DTO (Data Transfer Objects) - это объекты, предназначенные для передачи данных между слоями приложения.

**Преимущества использования DTO:**

* **Декупация:** DTO позволяют декоуплировать слои приложения, делая их независимыми друг от друга.
* **Снижение зависимости от ORM:** DTO не зависят от конкретной технологии ORM, что упрощает их использование.
* **Улучшение сериализации:** DTO оптимизированы для сериализации в JSON и другие форматы данных.
* **Повышение читаемости кода:** DTO делают код более понятным и читаемым.

**1.3.2. Примеры DTO:**

* **DTO стартапа:**
  + Id: int
  + Name: string
  + Description: string
  + CategoryName: string
  + StageName: string
* **DTO инвестора:**
  + Id: int
  + Name: string
  + Surname: string
  + Email: string
  + Description: string
  + Portfolio: string
  + Website: string
* **DTO проекта:**
  + Id: int
  + Name: string
  + Description: string
  + Problem: string
  + Solution: string
  + Team: string
  + StageName: string
  + RequiredAmount: int
  + CategoryName: string
  + StartupName: string
* **DTO категории:**
  + Id: int
  + Name: string
* **DTO этапа:**
  + Id: int
  + Name: string
* **DTO инвестиции:**
  + Id: int
  + Amount: int
  + Date: DateTime
  + InvestorName: string
  + StartupName: string
  + ProjectName: string

**Свойства DTO:**

* Свойства DTO должны соответствовать полям сущностей, но не должны содержать бизнес-логику.
* DTO должны быть простыми и лаконичными.
* DTO должны быть сериализуемыми в JSON и другие форматы данных.

**Пример реализации DTO стартапа:**

public class StartupDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public string CategoryName { get; set; }

public string StageName { get; set; }

}

**Преимущества использования DTO:**

* **Повышение модульности:** DTO позволяют модулизировать код приложения.
* **Улучшение тестируемости:** DTO легко тестировать изолированно от других компонентов.
* **Повышение безопасности:** DTO можно использовать для фильтрации и проверки данных, что повышает безопасность приложения.

4.3. Маппинг объектов с использованием библиотеки AutoMapper

**AutoMapper** - это библиотека .NET для автоматического маппинга объектов.

**Преимущества использования AutoMapper:**

* **Уменьшение кода:** AutoMapper позволяет сократить количество кода, необходимого для маппинга объектов.
* **Повышение читаемости кода:** AutoMapper делает код более читаемым и понятным.
* **Уменьшение ошибок:** AutoMapper помогает избежать ошибок, связанных с ручным маппингом объектов.
* **Улучшение тестируемости:** AutoMapper упрощает тестирование кода маппинга объектов.

**Основные возможности AutoMapper:**

* **Автоматическое маппинг объектов:** AutoMapper может автоматически сопоставлять свойства объектов с одинаковыми названиями.
* **Настройка маппинга:** AutoMapper позволяет настроить маппинг объектов, используя профили и атрибуты.
* **Поддержка сложных типов:** AutoMapper поддерживает маппинг сложных типов, таких как коллекции и вложенные объекты.
* **Расширяемость:** AutoMapper можно расширить, используя плагины.

**Установка AutoMapper:**

1. Добавьте пакет AutoMapper NuGet в ваш проект ASP.NET Web API.

**Пример использования AutoMapper:**

using AutoMapper;

public class StartupDTO

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

}

public class Startup

{

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

}

public class StartupProfile : Profile

{

public StartupProfile()

{

CreateMap<Startup, StartupDTO>();

}

}

public class StartupController : ControllerBase

{

private readonly ISstartupRepository \_startupRepository;

private readonly IMapper \_mapper;

public StartupController(IStartupRepository startupRepository, IMapper mapper)

{

\_startupRepository = startupRepository;

\_mapper = mapper;

}

[HttpGet]

public IEnumerable<StartupDTO> GetStartups()

{

var startups = \_startupRepository.GetAll();

return \_mapper.Map<IEnumerable<StartupDTO>>(startups);

}

[HttpGet("{id}")]

public StartupDTO GetStartupById(int id)

{

var startup = \_startupRepository.GetById(id);

if (startup != null)

{

return \_mapper.Map<StartupDTO>(startup);

}

return null;

}

}

**В этом примере:**

* **StartupDTO** - это DTO, который представляет сущность Startup.
* **StartupProfile** - это профиль AutoMapper, который определяет правила маппинга между Startup и StartupDTO.
* **StartupController** - это контроллер ASP.NET Web API, который использует AutoMapper для маппинга объектов Startup в StartupDTO.

**AutoMapper - это мощный инструмент, который может значительно упростить маппинг объектов в ASP.NET Web API.**

4.4. Создание методов контроллеров для CRUD операций

**Определение методов контроллеров:**

Для каждой сущности в вашей предметной области необходимо создать методы контроллера для реализации CRUD-операций:

* **Create (POST):** Создает новый объект.
* **Read (GET):** Получает объект по ID.
* **Update (PUT):** Обновляет существующий объект.
* **Delete (DELETE):** Удаляет объект по ID.

**Пример реализации методов контроллера для сущности "Стартап"**

[Route("api/[controller]")]

[ApiController]

public class StartupController : ControllerBase

{

private readonly IStartupRepository \_startupRepository;

private readonly IMapper \_mapper;

public StartupController(IStartupRepository startupRepository, IMapper mapper)

{

\_startupRepository = startupRepository;

\_mapper = mapper;

}

[HttpPost]

public IActionResult CreateStartup([FromBody] StartupDTO startupDTO)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

var startup = \_mapper.Map<Startup>(startupDTO);

\_startupRepository.Add(startup);

\_startupRepository.SaveChanges();

var newStartupDTO = \_mapper.Map<StartupDTO>(startup);

return CreatedAtAction(nameof(GetStartupById), new { id = newStartupDTO.Id }, newStartupDTO);

}

[HttpGet("{id}")]

public IActionResult GetStartupById(int id)

{

var startup = \_startupRepository.GetById(id);

if (startup == null)

{

return NotFound();

}

var startupDTO = \_mapper.Map<StartupDTO>(startup);

return Ok(startupDTO);

}

[HttpPut("{id}")]

public IActionResult UpdateStartup(int id, [FromBody] StartupDTO startupDTO)

{

if (!ModelState.IsValid)

{

return BadRequest(ModelState);

}

if (id != startupDTO.Id)

{

return BadRequest();

}

var startup = \_mapper.Map<Startup>(startupDTO);

\_startupRepository.Update(startup);

\_startupRepository.SaveChanges();

return NoContent();

}

[HttpDelete("{id}")]

public IActionResult DeleteStartup(int id)

{

var startup = \_startupRepository.GetById(id);

if (startup == null)

{

return NotFound();

}

\_startupRepository.Delete(id);

\_startupRepository.SaveChanges();

return NoContent();

}

}

**В этом примере:**

* **StartupController** - это контроллер ASP.NET Web API, который управляет сущностью "Стартап".
* **CreateStartup** - метод POST, который создает новый стартап.
* **GetStartupById** - метод GET, который получает стартап по ID.
* **UpdateStartup** - метод PUT, который обновляет существующий стартап.
* **DeleteStartup** - метод DELETE, который удаляет стартап по ID.

**Обработка ошибок:**

* Используйте ModelState для проверки входных данных и обработки ошибок валидации.
* Возвращайте соответствующие коды HTTP-статусов (например, 400 для неверных данных, 404 для несуществующих объектов).

**Заключение:**

Создание методов контроллеров для CRUD-операций - это важная часть разработки RESTful API.

5. Аутентификация и авторизация пользователя

**5.1. Аутентификация и авторизация c помощью ASP.NET Identity**

ASP.NET Identity - это встроенная в ASP.NET Core система аутентификации и авторизации, которая позволяет реализовать регистрацию, вход в систему, управление профилями пользователей и многое другое.

**Настройка ASP.NET Identity**

1. **Добавление пакета NuGet:**

Install-Package Microsoft.AspNetCore.Identity

1. **Регистрация сервисов ASP.NET Identity:**

В файле Startup.cs добавьте код для регистрации сервисов ASP.NET Identity:

public void ConfigureServices(IServiceCollection services)

{

services.AddDbContext<ApplicationDbContext>(options =>

options.UseSqlServer(connectionString));

services.AddIdentity<ApplicationUser, IdentityRole>()

.AddEntityFrameworkStores<ApplicationDbContext>()

.AddDefaultTokenProviders();

}

1. **Создание базы данных:**

ASP.NET Identity автоматически создаст таблицы в базе данных для хранения информации о пользователях, ролях и разрешениях. Вы можете настроить этот процесс с помощью миграций базы данных.

1. **Настройка регистрации и входа в систему:**

Создайте контроллеры и представления для регистрации и входа в систему пользователей. ASP.NET Identity предоставляет множество готовых компонентов для упрощения этой задачи.

**Авторизация на основе ролей**

ASP.NET Identity позволяет реализовать авторизацию на основе ролей. Вы можете создавать роли и назначать им разрешения на доступ к различным частям вашего приложения.

1. **Создание ролей:**

Используйте класс RoleManager для создания ролей:

var roleManager = services.GetRequiredService<RoleManager<IdentityRole>>();

await roleManager.CreateAsync(new IdentityRole("Admin"));

await roleManager.CreateAsync(new IdentityRole("User"));

1. **Назначение ролей пользователям:**

Используйте класс UserManager для назначения ролей пользователям:

var userManager = services.GetRequiredService<UserManager<ApplicationUser>>();

var user = await userManager.FindByNameAsync("sohibkor@example.com");

await userManager.AddToRoleAsync(user, "Admin");

1. **Авторизация в контроллерах:**

Используйте атрибут [Authorize] для ограничения доступа к контроллерам и действиям:

[Authorize(Roles = "Admin")]

public class AdminController : Controller

{

// ...

}

**Преимущества использования ASP.NET Identity:**

* **Простота использования:** ASP.NET Identity предоставляет множество готовых компонентов для упрощения реализации аутентификации и авторизации.
* **Расширяемость:** ASP.NET Identity можно расширить с помощью пользовательских классов и политик.
* **Безопасность:** ASP.NET Identity использует современные методы аутентификации и авторизации для защиты вашего приложения.

**5.2. Тестирование endpoint-ов Identity в Swagger**

Swagger - это инструмент для документирования и тестирования API. Он позволяет генерировать интерактивную документацию для вашего API, которая включает в себя описание endpoint-ов, запросов и ответов.

**Настройка Swagger**

1. **Добавление пакета NuGet:**

Install-Package Swashbuckle.AspNetCore

1. **Регистрация сервисов Swagger:**

В файле Startup.cs добавьте код для регистрации сервисов Swagger:

public void ConfigureServices(IServiceCollection services)

{

// ...

services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo

{

Version = "v1",

Title = "Sohibkor API",

Description = "API для сайта Sohibkor.tj",

TermsOfService = new Uri("https://sohibkor.tj/terms"),

Contact = new OpenApiContact

{

Name = "Sohibkor Team",

Email = "info@sohibkor.tj",

Url = new Uri("https://sohibkor.tj")

},

License = new OpenApiLicense

{

Name = "Apache License 2.0",

Url = new Uri("https://www.apache.org/licenses/LICENSE-2.0")

}

});

c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme

{

In = ParameterLocation.Header,

Type = "apiKey",

Name = "Authorization",

Description = "JWT authentication token",

Scheme = "bearer"

});

c.AddSecurityRequirement(new Dictionary<string, List<string>>

{

{ "Bearer", new List<string>() }

});

});

}

1. **Добавление middleware Swagger:**

В файле Startup.cs добавьте код для добавления middleware Swagger:

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

// ...

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "Sohibkor API v1");

});

}

**Тестирование endpoint-ов Identity**

После настройки Swagger вы можете использовать Swagger UI для тестирования endpoint-ов Identity.

1. **Откройте Swagger UI:**

Перейдите по адресу https://localhost:5001/swagger/index.html в вашем браузере.

1. **Выбрать endpoint Identity:**

Выбрать endpoint Identity, который хотите протестировать.

1. **Добавьте JWT-токен:**

В поле Authorization добавить JWT-токен, который был получен при авторизации пользователя.

1. **Выбрать метод HTTP:**

Выбрать метод HTTP (GET, POST, PUT, DELETE) для endpoint-а.

1. **Введите запрос:**

Введите запрос в формате JSON.

1. **Нажмите кнопку "Execute":**

Swagger UI отправит запрос на сервер и отобразит ответ.

**Преимущества использования Swagger:**

* **Улучшенная документация:** Swagger позволяет генерировать интерактивную документацию для вашего API, которая упрощает его использование разработчиками.
* **Простота тестирования:** Swagger позволяет легко тестировать endpoint-ы вашего API без необходимости писать код.
* **Обнаружение ошибок:** Swagger может помочь вам обнаружить ошибки в вашем API на ранней стадии разработки.

**Заключение**

ASP.NET Identity - это мощный инструмент для реализации аутентификации и авторизации в приложениях ASP.NET Core.

Использование ASP.NET Identity позволяет вам быстро и легко добавить функции регистрации, входа в систему и управления профилями пользователей в ваше приложение, а также обеспечить надежную защиту вашего приложения от несанкционированного доступа.

Swagger - это ценный инструмент для документирования и тестирования API.

Использование Swagger позволяет вам улучшить качество вашего API, сделать его более понятным для разработчиков и упростить его тестирование.

6. Создание пользовательского интерфейса с Blazor. Работа с данными в Blazor

**6.1. Создание прототипа клиентского приложения в Figma**

**1. Цель:**

Создание прототипа клиентского приложения Sohibkor.tj в Figma для визуализации пользовательского интерфейса и обеспечения согласованности дизайна на всех страницах сайта.

**2. Процесс:**

* **Изучение требований:** Проведен анализ требований к пользовательскому интерфейсу, включая функциональные возможности, целевую аудиторию и желаемый стиль.
* **Создание основных элементов:** Разработаны основные элементы интерфейса, такие как шапка сайта, меню навигации, карточки проектов, формы регистрации и авторизации.
* **Создание прототипов страниц:** Созданы прототипы основных страниц сайта, таких как главная страница, страница с списком проектов, страница с описанием проекта, страница регистрации и страница авторизации.
* **Тестирование и итерации:** Проведено тестирование прототипов с потенциальными пользователями для получения обратной связи и внесения корректировок.

**3. Результат:**

Создан подробный прототип клиентского приложения Sohibkor.tj в Figma, который наглядно демонстрирует пользовательский интерфейс и обеспечивает согласованность дизайна на всех страницах сайта.

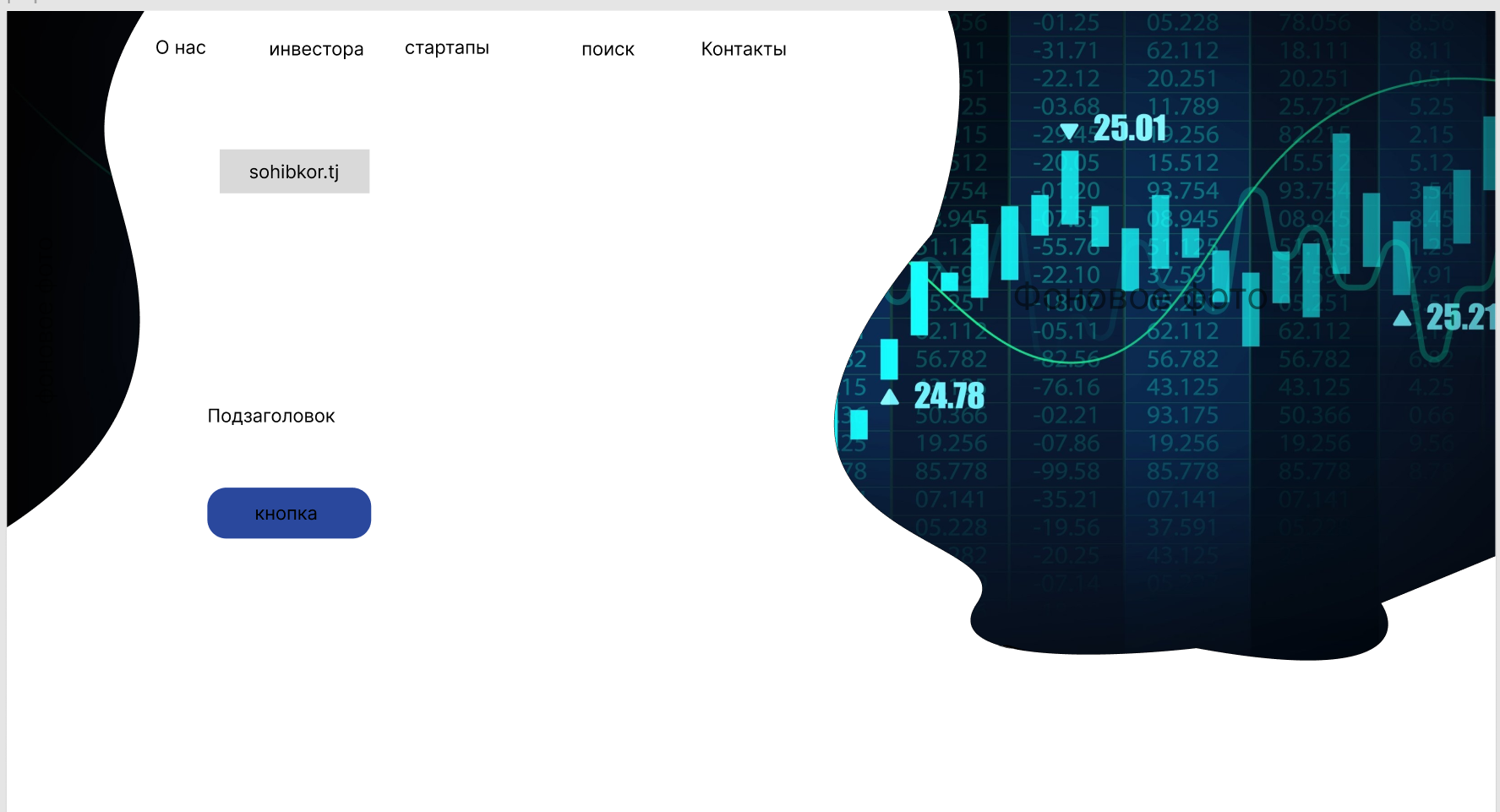


Рис. 1. Прототип сайта на figma

**4. Ссылка на прототип:**

<https://www.figma.com/file/LK9w6G0W8RAP49XtlaOMZe/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF-%D0%A1%D0%B0%D0%B9%D1%82%D0%B0-(Community)-(Copy)?type=design&node-id=0-1&mode=design&t=acCxc6Qdho2VGGGA-0>

**5. Преимущества использования Figma:**

* **Визуальное прототипирование:** Figma позволяет создавать интерактивные прототипы, которые имитируют работу реального приложения.
* **Сотрудничество:** Figma позволяет команде дизайнеров и разработчиков работать над прототипом одновременно.
* **Легкость использования:** Figma имеет простой и понятный интерфейс, что делает его доступным для пользователей с любым уровнем подготовки.6.2. Создание компонентов

**Цель:**

Создание reutilizable компонентов Blazor для клиентского приложения Sohibkor.tj, которые будут использоваться на разных страницах сайта.

**Подход:**

* **Определение компонентов:** Определить основные компоненты пользовательского интерфейса, такие как кнопки, карточки, формы, таблицы и т.д.
* **Разработка компонентов:** Разработать каждый компонент как отдельный Razor-файл с кодом C# и HTML-разметкой.
* **Использование компонентов:** Использовать созданные компоненты на разных страницах клиентского приложения.

**Пример создания компонента кнопки:**

1. **Создайте Razor-файл:** Button.razor
2. **Добавьте код C#:**

@code {

[Parameter]

public string Text { get; set; } = "Button";

[Parameter]

public EventCallback OnClick { get; set; }

}

<button class="btn btn-primary" @onclick="@OnClick">@Text</button>

**Используйте компонент:**

HTML

<Button Text="Click me" OnClick="@OnClick" />

**Преимущества использования компонентов:**

* **Повторное использование:** Компоненты можно использовать повторно на разных страницах сайта, что упрощает разработку и обслуживание кода.
* **Модульность:** Компоненты инкапсулируют код и разметку, что делает код более модульным и понятным.
* **Согласованность:** Компоненты обеспечивают согласованность дизайна и поведения на всех страницах сайта.

**6.3. Привязка данных к компонентам**

**Цель:**

Отображение данных в компонентах Blazor и обеспечение взаимодействия пользователя с этими данными.

**Подходы к привязке данных:**

* **Односторонняя привязка:** Данные из модели передаются в компонент, но изменения в компоненте не отражаются в модели.
* **Двусторонняя привязка:** Изменения данных в компоненте автоматически отражаются в модели.
* **Событийная привязка:** Компонент подписывается на события модели и обновляет свой интерфейс при их возникновении.

**Пример односторонней привязки:**

@page "/products"

<h1>Products</h1>

<ul>

@foreach (var product in Products)

{

<li>

@product.Name - @product.Price

</li>

}

</ul>

**Пример двусторонней привязки:**

@page "/products"

<h1>Products</h1>

<input type="text" @bind-value="@newProductName" />

<button @onclick="@AddProduct">Add Product</button>

<ul>

@foreach (var product in Products)

{

<li>

@product.Name - @product.Price

</li>

}

</ul>

@code {

private List<Product> Products = new List<Product>();

private string newProductName = "";

private void AddProduct()

{

Products.Add(new Product { Name = newProductName, Price = 100 });

newProductName = "";

}

}

**Преимущества использования привязки данных:**

* **Упрощенная разработка:** Привязка данных упрощает разработку пользовательского интерфейса, так как не требуется вручную обновлять элементы при изменении данных.
* **Реактивность:** Привязка данных обеспечивает реактивность пользовательского интерфейса, что означает, что изменения данных автоматически отражаются в интерфейсе.
* **Разделение ответственности:** Привязка данных разделяет логику работы с данными (модель) и логику отображения данных (компонент).

**Рекомендации по использованию привязки данных:**

* **Выбирайте правильный тип привязки для каждого случая использования.**
* **Используйте выражения Razor для форматирования данных.**
* **Обрабатывайте ошибки привязки данных.**

5.4. Использование MudBlazor

**Цель:**

Использовать MudBlazor для создания красивых и функциональных компонентов пользовательского интерфейса в Blazor-приложении Sohibkor.tj.

**Что такое MudBlazor?**

MudBlazor - это библиотека компонентов пользовательского интерфейса с открытым исходным кодом для Blazor, основанная на Material Design.

Она позволяет создавать современные и привлекательные интерфейсы, используя знакомые элементы Material Design.

**Преимущества использования MudBlazor:**

* **Готовые компоненты:** MudBlazor предоставляет широкий набор готовых компонентов, таких как кнопки, карточки, таблицы, формы, диалоговые окна и многое другое.
* **Простота использования:** MudBlazor прост в использовании и имеет понятный API.
* **Красивый дизайн:** MudBlazor основан на Material Design, что обеспечивает красивый и современный дизайн компонентов.
* **Реактивность:** MudBlazor компоненты реагируют на изменения данных и пользовательский ввод.
* **Расширяемость:** MudBlazor можно расширять с помощью пользовательских компонентов и стилей.

**Пример использования MudBlazor:**

@page "/products"

<h1>Products</h1>

<MudGrid>

@foreach (var product in Products)

{

<MudCard>

<MudCardContent>

<h2>@product.Name</h2>

<p>@product.Description</p>

<MudButton @onclick="@(() => NavigateToProduct(product.Id))">

Details

</MudButton>

</MudCardContent>

</MudCard>

}

</MudGrid>

@code {

private List<Product> Products = new List<Product>();

private void NavigateToProduct(int productId)

{

NavigationManager.NavigateTo($"/product/{productId}");

}

}

**Заключение:**

MudBlazor - это мощный инструмент для создания красивых и функциональных компонентов пользовательского интерфейса в Blazor-приложениях.

Использование MudBlazor позволяет упростить разработку, улучшить дизайн и повысить качество пользовательского опыта.

Привязка данных является essential aspect of Blazor development.

Использование привязки данных позволяет создавать динамичные и интерактивные пользовательские интерфейсы, которые реагируют на изменения данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

* + <https://github.com/>(дата обращения: 04.04.2024)
  + <https://git-scm.com/doc>(дата обращения: 07.04.2024)
  + [https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/managing-your-profile-readme](https://gemini.google.com/app/%3C0%3Ehttps:/docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/customizing-your-profile/managing-your-profile-readme)(дата обращения: 09.04.2024)
  + **Microsoft Docs: ASP.NET Core Entity Framework Core:** <https://learn.microsoft.com/en-us/ef/> (дата обращения: 10.04.2024)
  + **AutoMapper documentation:** <https://github.com/AutoMapper/AutoMapper> (дата обращения: 12.04.2024)
  + **REST API Design Best Practices:** <https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/> (дата обращения: 12.04.2024)
  + <https://swagger.io/tools/swagger-ui/>
  + <https://learn.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-8.0><https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0>
  + <https://m.youtube.com/watch?v=bNrym4PqhgM>
  + <https://mudblazor.com/docs/overview>
  + <https://github.com/MudBlazor/MudBlazor>
  + <https://mudblazor.com/mud/community/getting-help>
  + <https://go.microsoft.com/fwlink/?linkid=851728>
  + <https://mudblazor.com/docs/overview>
  + <https://github.com/parvina88/RealEstateAPI/issues/11>
  + <https://medium.com/@avinash.dhumal/understanding-dependency-injection-a-practical-guide-with-c-examples-aee44eacee32>
  + <https://metanit.com/sharp/tutorial/>
  + <https://metanit.com/sharp/tutorial/3.1.php>
  + https://github.com/parvina88/RealEstateCRM/files/14839101/1.txt