

# Учебное приложение «Опросник»

## Быстрая разработка на Blazor

Автор: Минюров Сергей

Версия: 0, июль 2021

**Требования к аудитории:** начальный опыт программирования на C#, работы с базой данных, HTML и CSS (в т.ч. Bootstrap).

**Основная цель:** обучение базовой техники программирования на Blazor при разработке веб-приложения.

**Дополнительная цель:** отработка стандартных решений простых задач в методологии «быстрая разработка» (RAD – Rapid Application Development).

### Задачи верхнего уровня:

- Разработка объектной модели данных (подход code-first) на C# с помощью EntityFramework (библиотека для работы с базами данных, отображение программных объектов на реляционные данные, ORM – object-relation mapping).
- Разработка веб-сервисов как Web API контроллеров на C# для удаленного вызова программных функций на сервере.
- Разработка пользовательского интерфейса с помощью HTML, CSS и C# в формате программного кода Razor.

### Программные ресурсы:

- Для программирования: Visual Studio Edition 2019, бесплатная редакция Community.
- Система управления базами данных: SQL Server, бесплатные редакции<sup>1</sup> Express или Developer.

Программный код в учебном приложении не является оптимальным образцом для использования в высоконагруженных и защищенных решениях. Его целью является демонстрация для начинающих разработчиков простых и базовых практик для решения стандартных задач.

Полный код приложения в данном руководстве не представлен для краткости изложения. Доступен для скачивания по ссылке: <https://github.com/sergeyminyurov/Inquirer>. Рекомендуется внимательно изучать программный код в соответствии с разделом, скачав исходный код учебного приложения.

## Что такое Blazor

Blazor это программная платформа, обеспечивающая разработку серверной и клиентской части приложения на общем языке программирования. Принципиальным при этом является возможность совместного использования программных компонентов на сервере и клиенте.



Официальный сайт Blazor: <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>

---

<sup>1</sup> Microsoft SQL Server Express можно также использовать бесплатно в рабочих средах для небольших приложений. Редакция Developer может использоваться только для разработки, тестирования или обучения.

Имеется два способа исполнения: на сервере и на клиенте при достаточно схожем процессе разработки. При исполнении на сервере при изменении состояния программы для каждого пользователя (сессии) вычисляются изменения визуального представления и с помощью SignalR передаются на клиент.

При исполнении на клиенте используется открытая технология WebAssembly: запускается виртуальная машина в браузере, в которую загружаются программные компоненты и выполняют вычисление изменений визуального представления.

Таблица 1. Сравнение способов исполнения Blazor

Сценарии	Исполнение на сервере (SignalR)	Исполнение на клиенте (WebAssembly)
Запуск приложения	Загрузка только разметки и используемых ресурсов (картинки, javascript и пр.)	Дополнительно загружаются программные компоненты и запускается виртуальная машина в браузере
	<ul style="list-style-type: none"> <li>Быстрая загрузка (как обычное веб-приложение)</li> </ul>	<ul style="list-style-type: none"> <li>Дополнительная задержка при загрузке (оптимизируется с помощью отложенной загрузки компонентов или разработки традиционных веб-компонентов для начальных страниц)</li> </ul>
Исполнение приложения	На сервере для каждого пользователя генерируются представления и вычисляются их последующие изменения. Результирующая разметка передается на клиент с помощью SignalR.	Пользовательский интерфейс генерируется в виртуальной машине браузера на клиенте. Между сервером и клиентом передаются только данные и ресурсы.
	<ul style="list-style-type: none"> <li>Могут быть задержки с обновлением из-за канала и перегрузки сервера</li> </ul>	<ul style="list-style-type: none"> <li>Быстрое обновление клиента.</li> </ul>
	<ul style="list-style-type: none"> <li>Повышенная нагрузка на сервер.</li> </ul>	<ul style="list-style-type: none"> <li>Сбалансированная нагрузка на сервер и клиент.</li> </ul>
Безопасность приложения	Зависит от техники программирования. Все данные и клиентские скрипты легко доступны.	Зависит от техники программирования. Более сложный доступ к программному коду на клиенте.

В учебном приложении используется исполнение на клиенте на основе WebAssembly.



Официальный сайт WebAssembly: <https://webassembly.org/>

## Функциональные требования

1. Пользователь может регистрироваться в приложении.
2. Зарегистрированный пользователь может создать опросник с набором вопросов и вариантами ответов и проходить опрос.
3. В опроснике можно создавать группы вопросов.
4. Опросник, каждый ответ или группа вопросов имеет количественный параметр «Баллы» для оценки результатов.

- 4.1. Правильный ответ имеет положительное значение.
- 4.2. Неправильный ответ имеет нулевое или отрицательное значение
- 4.3. Если вопрос содержит только один правильный ответ, то можно выбирать один ответ.
- 4.4. Если вопрос содержит несколько правильных ответов, то можно выбирать несколько ответов одновременно.
- 4.5. После проведения опроса числовые значения суммируются и сравниваются с параметром для опросника и групп вопросов, если они есть.
5. При проведении опроса фиксируется время начала и окончания.

## Архитектура приложения

Для хранения данных используется баз данных, в которой может быть реализована часть бизнес-логики для повышения производительности в виде хранимых процедур, запросов и функций.

Логически веб-приложение состоит двух частей:

- Сервер (backend) – хранилище данных, компоненты доступа к данным, обработки и интеграции, веб-сервисы.
- Клиент (frontend) – разметка, ресурсы и клиентские скрипты для построения пользовательского интерфейса.

Учебное решение состоит из следующих проектов<sup>2</sup>:

- Inquirer.Client – содержит разметку в форматах HTML и RAZOR (смесь HTML с ASP.NET), ресурсы и клиентскую логику (при использовании WebAssembly будет исполняться в браузере пользователя).
- Inquirer.Server – содержит логику доступа к данным и контроллеры для предоставления удаленного доступа к программным функциям (веб-сервисы). Также содержит разметку и логику для стандартной реализации функций безопасности.
- Inquirer.Shared – содержит общие типы, используемые на клиенте и сервере.



Рисунок 1. Системная архитектура веб-приложения

## Создание решения

Blazor стремительно развивается и для успешной работы важно своевременно обновлять версию Visual Studio и SDK. Учебное приложение разработано на Visual Studio 2019 версии 16.10.1.

Для создания решения нужно выбрать шаблон Blazor WebAssembly App. На следующем шаге нужно ввести название решения Inquirer. После выбрать опции как на следующем рисунке.

<sup>2</sup> При выборе шаблона решения и соответствующих опций создаются по умолчанию три проекта с такими именами (см. далее). В учебных целях демонстрируется самый простой подход при проектировании решения. При разработке рабочих приложений разделение уровней и аспектов функциональности может быть более строгое и, соответственно, потребуются создать дополнительные проекты, например, вынести логику доступа к данным в отдельный компонент.

Опция Progressive Web Application – позволяет пользователю скачать приложение через браузер на свой компьютер и выполнять его без подключения к серверу, для учебного проекта необязательная.

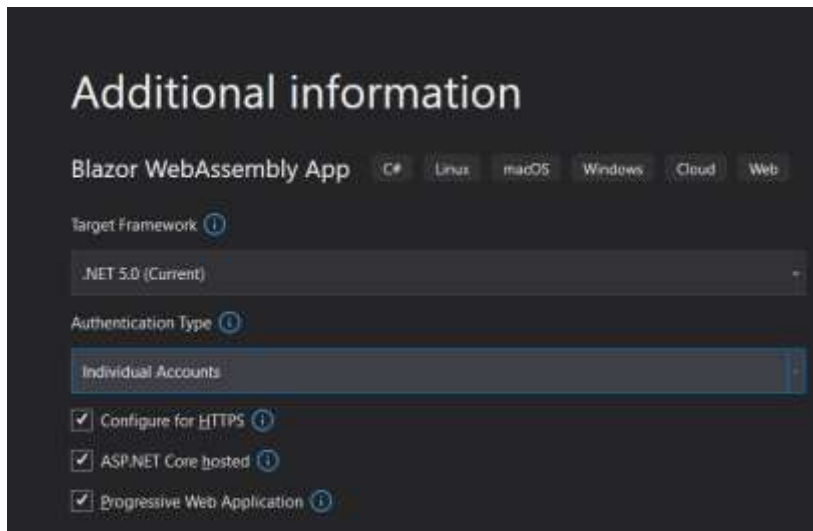


Рисунок 2. 3-й шаг создания решения.



Структура проекта: <https://docs.microsoft.com/en-us/aspnet/core/blazor/project-structure>

При успешном создании решения в проводнике (Solution Explorer) будет следующая структура:

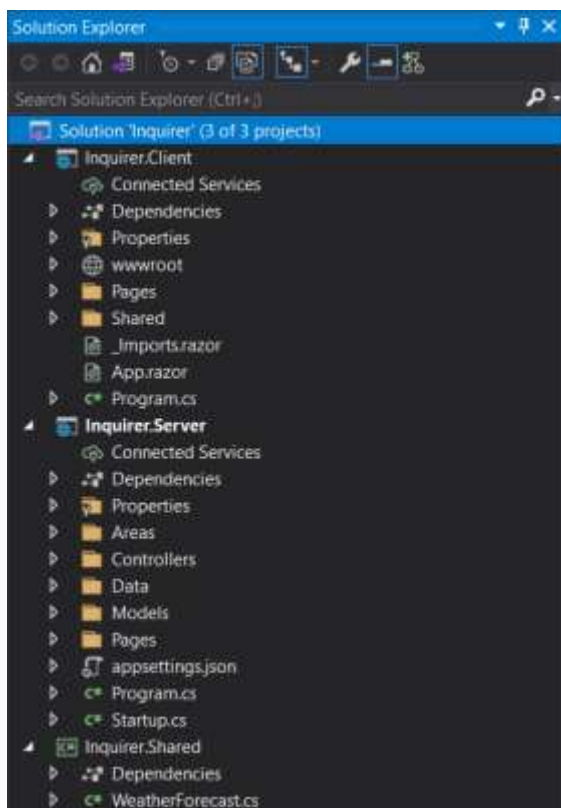


Рисунок 3. Начальная структура решения

Для проверки правильной настройки Visual Studio можно выполнить тестовый запуск приложения. Все функции в нем могут не работать из-за настроек базы данных (по умолчанию используется LocalDb, для которого требуется установка SQL Server Express Edition), которые требуются для регистрации пользователей. Это будет рассмотрено далее.

## Объектная модель данных

Для разработки объектной модели данных нужно внимательно изучить функциональные требования (см. предыдущий раздел). Анализ требований позволяет выделить следующие сущности (сохраняемые в базе данных объекты):

- Пользователь – используется готовый класс ApplicationUser, первоначально находится в проекте Inquirer.Server в папке Models.
- Вопросник – класс Questionnaire, содержит вопросы и группы вопросов.
- Вопрос – класс Question, содержит варианты ответов.
- Вариант ответа на вопрос – класс QuestionAnswer.
- Группа вопросов – класс QuestionGroup, содержит вопросы.
- Опрос – класс Survey, содержит ответы.
- Выбранный ответ в опросе – класс SurveyAnswer.

Объектная модель создается в проекте Inquirer.Shared, в папке Data, которую нужно создать. Класс ApplicationUser нужно перенести в этот проект, поскольку для пользователя создаются объектные ссылки.

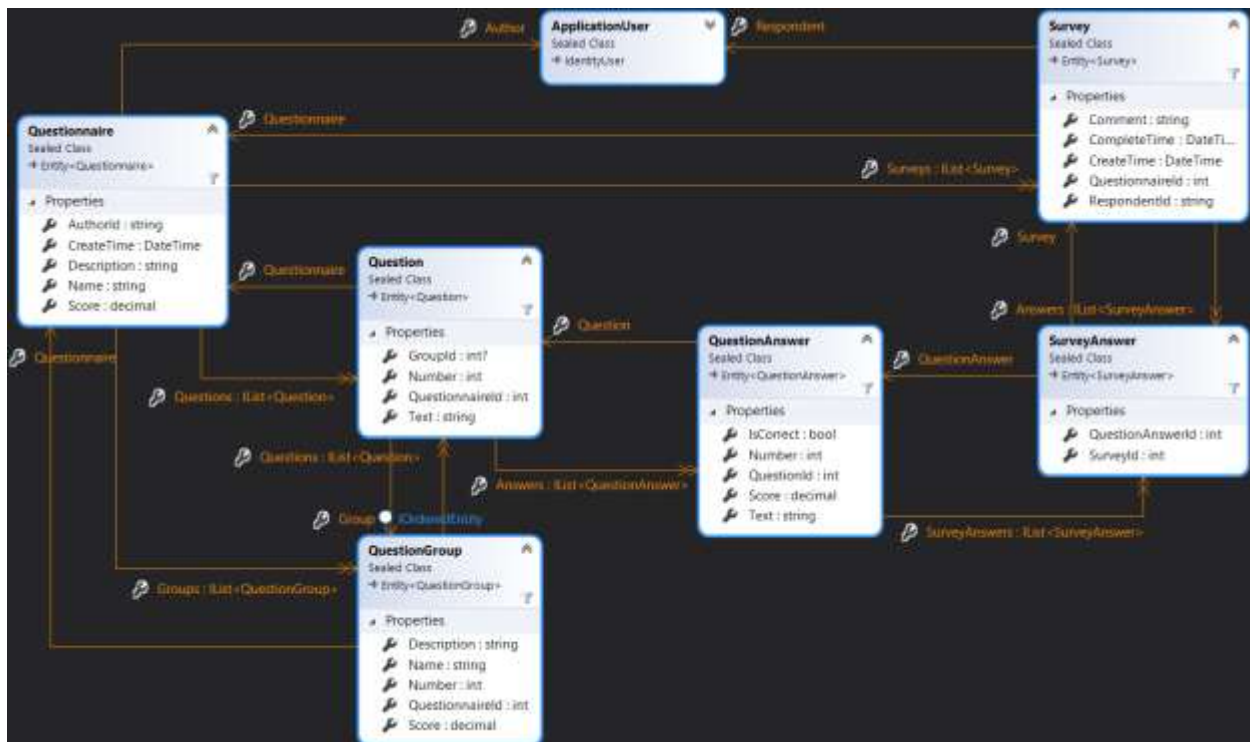


Рисунок 4. Объектная модель данных.

Для сохранения данных и их связывания в базе данных каждая сущность должна иметь идентификатор. В учебном проекте создан абстрактный тип Entity, который содержит

целочисленное свойство Id. Entity Framework по умолчанию использует свойства классов с таким названием как первичный ключ таблицы данных, в которой сохраняются его данные.

ApplicationUser имеет строковый идентификатор. Для обобщения логики контроллеров (см. далее раздел «Контроллеры (веб-сервисы)») создадим обобщенный интерфейс IEntity, позволяющий задавать тип идентификатора.

```
public interface IEntity<T>
{
    T Id { get; set; }
}
```

Листинг 1. Обобщенный интерфейс сущности

```
public sealed class ApplicationUser : IdentityUser, IEntity<string> { }
```

Листинг 2. Класс пользователя со строковым идентификатором

В учебном приложении новые данные имеют значение 0 в свойстве идентификатора. Для удобства добавлено вычисляемое логическое поле IsNew, которое возвращает значение true (истина), если данные новые и пока не сохранены в базе данных.

```
public abstract class Entity : IEntity<int>
{
    public int Id { get; set; }
    public bool IsNew => Id == 0;
    public abstract Entity ForDatabase();
}
```

Листинг 3. Базовый тип типов сущностей, сохраняемых в базе данных

Также в классе Entity определен абстрактный метод для сохранения сущности в базе данных: этот метод должен быть реализован в каждом производном конкретном классе перед его сохранением в базе данных, чтобы сохранять только собственные данные.

Для удобства программирования создан обобщенный тип сущности на основе исходного типа: возвращается собственный тип производной сущности.

```
public abstract class Entity<T> : Entity where T : Entity, new()
{
    public override abstract T ForDatabase();
}
```

Листинг 4. Обобщенный тип сущности для сохранения в базе данных.

```
public sealed class Questionnaire : Entity<Questionnaire>
```

Листинг 5. Определение класса вопросника на основе базового типа обобщенной сущности.

Метод ForDatabase создает новый экземпляр такого же класса, но не содержит никаких ссылочных данные, только скалярные, сохраняемые в соответствующей таблице базы данных.

```
public override Questionnaire ForDatabase() => new()
{
    Id = Id,
    Name = Name,
    Description = Description,
    Score = Score,
    CreateTime = CreateTime,
    AuthorId = AuthorId,
}
```

```
};
```

Листинг 6. Реализация метода в классе `Questionnaire`.

При проектировании объектной модели важно понимать связи между данными и их зависимости при создании и редактировании. Базовый вид связей — это композиция — структурные зависимости между данными. Они позволяют понимать, какие сущности создаются вначале и содержат остальные, которые могут создаваться на их основе. Композиционные группы данных:

- Вопросник содержит вопросы и группы (которые также содержат вопросы), а вопросы содержат варианты ответов. Кроме этого, каждый вопросник содержит опросы, созданные на его основе, а варианты ответов содержат фактические ответы, данные в опросах.
- Опросы содержат фактические ответы.

Композиция определяется в программном коде в виде списка<sup>3</sup>, если содержится множество экземпляров, либо объектной ссылки, если содержится только единственный экземпляр.

```
public IList<QuestionGroup> Groups { get; set; }  
public IList<Question> Questions { get; set; }  
public IList<Survey> Surveys { get; set; }
```

Листинг 7. Пример списков для композиции данных в классе `Questionnaire`.

Класс, которые содержится в другом классе может содержать его идентификатор (внешних ключ в терминах баз данных) и объектную ссылку на него.

```
public int QuestionnaireId { get; set; }  
[JsonIgnore]  
public Questionnaire Questionnaire { get; set; }
```

Листинг 8. Внешний ключ и объектная ссылка.

Entity Framework автоматически распознает свойство как внешний ключ (см. на «Листинг 8» свойство `QuestionnaireId`), если оно соответствует наименованию объектного свойства и в конце добавлено сокращение идентификатора — `Id`.

Для обратной объектной ссылки используется атрибут `JsonIgnore`, который позволяет избежать заикливания<sup>4</sup> при сериализации данных в формат данных JSON при обмене данными между сервером и клиентом.

### Валидация данных на стороне сущностей

Для обеспечения качества данных в классах сущностей с помощью атрибутов (специальных типов метаданных) можно задавать правила для проверки данных. Атрибут `RequiredAttribute` (сокращенный вариант `Required`) указывает для механизма проверки данных, что последующее свойство данных является обязательным и должно содержать данные для сохранения сущности в базе данных. Иначе для пользователя должно выводиться сообщение, заданное в параметре `ErrorMessage`.

<sup>3</sup> Здесь и далее показывается только один вариант реализации в программном коде. На практике можно увидеть разные приемы программирования, которые не описываются в данном руководстве для краткости и понятности.

<sup>4</sup> В более сложных приложениях разделяют сущности, сохраняемые в базе данных и модели данных, с помощью которых выполняется обмен данных между сервером и клиентом. В учебном приложении для упрощения одни и те же классы используются как сущности и как модели. Метод `ForDatabase` (см. выше) помогает использовать сущности как модели при сохранении данных.



```
[Required(ErrorMessage = "Введите наименование опросника")]
public string Name { get; set; }
[Range(1, 100, ErrorMessage = "Введите значение от 1 до 100 для баллов")]
public decimal Score { get; set; }
```

Листинг 9. Атрибуты для валидации данных в свойствах сущности

Атрибут `RangeAttribute` (сокращенный вариант `Range`) позволяет задавать диапазон разрешенных значений для числового свойства или даты-времени.

Как используются атрибуты проверки данных при разработке веб-формы рассказывается далее.

### Контекст данных для доступа к базе данных

Для сохранения состояния объектов (сущностей) в базе данных применяется низкоуровневое программирование на основе ADO.NET. Либо используются библиотеки ORM, например, `EntityFramework`, `NHibernate`, `Dapper` и пр.

В учебном приложении используется `EntityFramework`. При создании решения по шаблону и выборе соответствующих опций (см. выше) в проекте `Inquirer.Server`, папка `Data` создается класс контекста данных `ApplicationDbContext`. В качестве базового типа для него определен специальный тип `ApiAuthorizationDbContext`, обеспечивающий создание стандартного набора типов для аутентификации и авторизации, который давно используется в ASP.NET приложениях.



Библиотека доступа к данным `EntityFramework`: <https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore?view=efcore-5.0>

В папке `Models` находится класс `ApplicationUser`, который нужно<sup>5</sup> перенести в проект `Inquirer.Shared`, папка `Data`. Поскольку этот тип будет использоваться в других сущностях как автор вопросов и респондент при проведении опросов.

Класс контекста данных `ApplicationDbContext` переименован в `InquirerDbContext`. В нем добавлены свойства для доступа к множествам данных:

```
public DbSet<Questionnaire> Questionnaires { get; set; }
public DbSet<Question> Questions { get; set; }
public DbSet<QuestionGroup> QuestionGroups { get; set; }
public DbSet<QuestionAnswer> QuestionAnswers { get; set; }
public DbSet<Survey> Surveys { get; set; }
public DbSet<SurveyAnswer> SurveyAnswers { get; set; }
```

Листинг 10. Множества данных для сущностей приложения

Для настройки базы данных используется перегруженный (`override` – определен в базовом типе, и может меняться в производном типе) защищенный (`protected` – видимость только в исходном типе и производных типах).

В учебном приложении настраиваются индексы и определяются правила удаления данных: для упрощения решения настроено каскадное удаление данных. При этом, если удаляется какая-либо сущность, то удаляются и связанные с ней данные. Например, если удалить вопросник, то будут удалены вопросы и группы, а также опросы, созданные на его основе.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
```

<sup>5</sup> Необходимость переноса обусловлена тем, что в учебном проекте сущность (сохраняемый в базе данных тип) используется как модель – обмен данными между клиентом и сервером.



```

        modelBuilder.Entity<ApplicationUser>().HasIndex(t => t.Email).IsUnique();

        modelBuilder.Entity<Questionnaire>().HasMany(t => t.Groups).WithOne(t =>
t.Questionnaire).OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<Questionnaire>().HasMany(t => t.Questions).WithOne(t =>
t.Questionnaire).OnDelete(DeleteBehavior.NoAction);
        modelBuilder.Entity<Questionnaire>().HasMany(t => t.Surveys).WithOne(t =>
t.Questionnaire).OnDelete(DeleteBehavior.Cascade);

        modelBuilder.Entity<QuestionGroup>().HasMany(t => t.Questions).WithOne(t =>
t.Group).OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<Question>().HasMany(t => t.Answers).WithOne(t =>
t.Question).OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<QuestionAnswer>().HasMany(t => t.SurveyAnswers).WithOne(t =>
t.QuestionAnswer).OnDelete(DeleteBehavior.Cascade);
        modelBuilder.Entity<Survey>().HasMany(t => t.Answers).WithOne(t =>
t.Survey).OnDelete(DeleteBehavior.NoAction);
    }

```

Листинг 11. Настройка базы данных в контексте данных

Для пользователя (класс ApplicationUser) правила каскадного удаления не настроены, соответственно, при попытке выполнить пользователя, у которого есть вопросники или опросы, то программа выдаст ошибку и выполнит отмену выполнения команды удаления.

## Настройка сервера

В зависимости от используемой редакции СУБД в проекте Inquirer.Server, файл конфигурации appsettings.json нужно прописать строки подключения в разных форматах<sup>6</sup>:

- LocalDb (файловая база данных по умолчанию, требуется SQL Server Express):  
Server=(localdb)\\mssqllocaldb;Database=Inquirer;Trusted\_Connection=True;  
MultipleActiveResultSets=true
- SQL Server: Server=localhost;Database=Inquirer;Trusted\_Connection=True;  
MultipleActiveResultSets=true

В классе Program добавлен код для автоматического создания базы данных, если она не существует (посмотрите полный код):

```

var context = services.GetRequiredService<InquirerDbContext>();
context.Database.EnsureCreated();

```

Листинг 12. Проверка существования базы данных и ее создание, если нет

Основные настройки сервера выполняются в классе Startup в виде определения и конфигурации программных сервисов. Обратите внимание на определение контекста данных:

```

services.AddDbContext<InquirerDbContext>(options =>
options.UseSqlServer(
Configuration.GetConnectionString("DefaultConnection")));

```

Листинг 13. Инъекция контекста данных как программного сервиса

<sup>6</sup> В строке подключения нет учетных данных: используется интегрированный режим безопасности и передаются учетные данные текущего пользователя, т.е. наши собственные. При коллективной разработке, в тестовой или рабочей среде могут использоваться служебные учетные данные. А сам файл конфигурации в может быть защищен и недоступен пользователю.

Шаблон проектирования программных сервисов обеспечивает слабую связанность программных компонентов и простоту использования этих сервисов (см. пример с контроллерами далее).

## Контроллеры (веб-сервисы)

Контроллеры обеспечивают обмен данными между клиентом и сервером, удаленный вызов программных функций в виде веб-сервисов.

При создании решения создается контроллер `OidcConfigurationController` для поддержки стандартных функций по безопасности для клиента.

Также создается демонстрационный контроллер `WeatherForecastController`, который можно удалить.



Аутентификация и авторизация: <https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity-api-authorization>

Для каждой сущности создается отдельный контроллер для чтения и сохранения данных. Поскольку эти операции имеют много общего, сначала разрабатывается базовый контроллер `EntityController` как абстрактный класс.

Атрибут `ApiController` определяет, что это контроллер только для выполнения команд, а не для формирования клиентских представлений. Этому соответствует базовый тип `ControllerBase`.



Создание ASP.NET Web API: <https://docs.microsoft.com/en-us/aspnet/core/web-api>

Атрибут `Route` определяет относительный путь к методам контроллера. Префикс `api` является необязательным, но общепринятым соглашением. Вместо значения `controller` с квадратными скобками подставляется имя текущего класса без постфикса `Controller`.

```
[ApiController]
[Route("api/[controller]")]
public abstract class EntityController<TEntity, TIdentity> : ControllerBase
    where TEntity : class, IEntity<TIdentity>, new()
```

Листинг 14. Заголовок контроллера `EntityController`

Обобщенные параметры контроллера указывают тип сущности и тип данных ее идентификатора. Конструктор класса в качестве параметра получает контекст данных с помощью инъекции сервисов (см. выше раздел «Настройка сервера»).

Контроллер реализует стандартные методы для чтения и сохранения данных:

- `GetAll` – чтение всех данных
- `Get` – чтение сущности по идентификатору
- `AddNew` – создание новой сущности
- `Update` – сохранение изменений сущности
- `Delete` – удаление сущности

Для настройки функциональности, специфической для сущности также определены защищенные (доступные только в производном типе) виртуальные (с возможностью переопределения) методы:

- `GetQuery` – определяет запрос для чтения данных, в т.ч. какие связанные данные могут дополнительно читаться.

- NewEntity – дополнительные операции перед сохранением новой сущности, например, добавления времени создания (пустой метод по умолчанию).
- DeleteDependencies – удаление зависимых данных при удалении сущности (пустой метод по умолчанию).

```
public class ApplicationController : EntityController<ApplicationUser, string>
{
    public ApplicationController(InquirerDbContext db) : base(db) { }
}
```

Листинг 15. Стандартная реализация контроллера для пользователя

## Переопределение метода чтения данных

Метод GetQuery по умолчанию читает только данные сущности:

```
protected virtual IQueryable<TEntity> GetQuery([CallerMemberName] string methodName = null) => DbContext.Set<TEntity>();
```

Листинг 16. Чтение данных по умолчанию

Параметр methodName с помощью атрибута CallerMemberName обеспечивает автоматическую передачу названия другого метода, в котором был вызван GetQuery.

Для отображения связанных с сущностью данных этот метод переопределяется:

```
protected override IQueryable<Questionnaire> GetQuery([CallerMemberName] string methodName = null)
{
    switch (methodName)
    {
        case nameof(Get):
            return DbContext.Questionnaires
                .Include(t => t.Groups)
                .Include(t => t.Questions)
                .ThenInclude(t => t.Answers);
        case nameof(GetAll):
            return DbContext.Questionnaires
                .Include(t => t.Author);
    }
    return base.GetQuery(methodName);
}
```

Листинг 17. Чтение связанных данных для вопросника

Для вопросника обрабатывается наименование метода: если метод Get (чтение сущности по идентификатору), то данные читаются для редактирования вопросника и дополнительно будут переданные данные по группам, вопросам и ответам (определяется с помощью методов Include и ThenInclude).



Программный запросы к данным (Linq): <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities>

Метод GetAll используется для отображения списка сущностей. Для вопросника дополнительно считываются данные об авторе для отображения в таблице (см. далее раздел «

Разметка страниц»).

## Переопределение метода создания новой сущности

При создании нового опроса с помощью переопределенного метода `NewEntity` устанавливается код пользователя и текущее время:

```
protected override void NewEntity(Survey entity)
{
    entity.RespondentId = HttpContext.User.GetId();
    entity.CreateTime = System.DateTime.Now;
}
```

Листинг 18. Установка значений для новой сущности (опросника)

Идентификатор пользователя читается из контекста http-запроса (класс `HttpContext` и одноименное свойство контроллера) с помощью метода `GetId` статического класса `EntityExtensions`, содержащего набор методов утилит, расширяющих функционал типов:

```
public static string GetId(this ClaimsPrincipal user, bool throwIfUndefined = true)
{
    Claim claim = user.Claims.FirstOrDefault(t =>
t.Type.Equals(ClaimTypes.NameIdentifier));
    if (claim is null)
    {
        if (throwIfUndefined)
            throw new ArgumentNullException(ClaimTypes.NameIdentifier);
        return null;
    }
    return claim.Value;
}
```

Листинг 19. Получение идентификатора текущего пользователя из контекста запроса (`HttpContext`)

## Настройка клиента

В проекте `Inquirer.Client` автоматически создается класс `Program`, в методе `Main` выполняется настройка программных сервисов для клиента и запуск клиентского приложения. В этот метод нужно добавить клиентские сервисы. Также добавляется сервис кэширования данных в памяти, если он используется для снижения нагрузки на сервер.

```
builder.Services.AddMemoryCache();

builder.Services.AddScoped<IDataService, DataService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<ISurveyService, SurveyService>();
```

Листинг 20. Настройка клиентских сервисов



Внедрение зависимостей: <https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/dependency-injection?view=aspnetcore-5.0&pivots=server>

## Клиентские сервисы

Для операций с данными, соответствующих методам контроллера `EntityController` разработан программный интерфейс `IDataService` и класс `DataService`, его реализующий.

```
public interface IDataService
{
    Task<List<T>> GetList<T>() where T : Entity;
    Task<T> Get<T>(int id) where T : Entity;
    Task<T> AddNew<T>(T newEntity) where T : Entity;
```

```

    Task Update<T>(T entity) where T : Entity;
    Task Delete<T>(int id) where T : Entity;
    Task Swap<T>(T entity1, T entity2) where T : Entity, IOrderedEntity;
}

```

Листинг 21. Интерфейс сервиса данных

Класс DataService использует класс HttpClient для взаимодействия с сервером и интерфейсы IMemoryCache и ILogger<DataService> для кэширования данных и логирования операций. Эти типы внедряются через параметры конструктора.



Вызов веб-API из ASP.NET Core Blazor: <https://docs.microsoft.com/en-us/aspnet/core/blazor/call-web-api?view=aspnetcore-5.0&pivots=webassembly>

```

public sealed class DataService : IDataService
{
    const int ExpirationInterval = 300;
    private HttpClient Http { get; }
    private IMemoryCache Cache { get; }
    private ILogger<DataService> Logger { get; }
    public DataService(HttpClient http, IMemoryCache cache, ILogger<DataService> logger)
    {
        Http = http;
        Cache = cache;
        Logger = logger;
    }
}

```

Листинг 22. Инъекция сервисов с помощью параметров конструктора

Для универсальной обработки данных используется соглашение, что наименование контроллера соответствует сущности. Например, контроллер для сущности Question называется QuestionController. По правилам маршрутизации ASP.NET Web API постфикс Controller в наименовании контроллера отбрасывается.

Обобщенный метод GetRequestUri по параметру типа и необязательному параметру аргументов формирует относительный путь для контроллера, соответствующего типу:

```

private string GetRequestUri<T>(object arg = null) =>
    $"api/{typeof(T).Name.ToLower()}/{arg}";

```

Листинг 23. Определение относительного адреса для контроллера сущности

Каркас метода обеспечивает логирование вызова метода контроллера в режиме отладки (метод LogDebug) и ошибки с помощью программной конструкции try catch (метод LogError). При этом ошибка не обрабатывается, а передается далее по стеку вызова с помощью инструкции throw.

```

public async Task<T> Get<T>(int id) where T : Entity
{
    try
    {
        ...
        Logger.LogDebug($"Get<{typeof(T).Name}>: {requestUri}");
        ...
    }
    catch (Exception ex)
    {
        Logger.LogError($"Get<{typeof(T).Name}>: id={id};
        {ex.GetBaseException().Message}");
    }
}

```

```

        throw;
    }
}

```

Листинг 24. Каркас метода сервиса данных для логирования ошибок и отладочной информации

При чтении списка сущностей данные кэшируются по типу как ключу на определенный период времени (ExpirationInterval):

```

public async Task<List<T>> GetList<T>() where T : Entity
{
    try
    {
        return await Cache.GetOrCreateAsync(typeof(T), async e =>
        {
            e.SetOptions(new MemoryCacheEntryOptions
            {
                AbsoluteExpirationRelativeToNow =
                TimeSpan.FromSeconds(ExpirationInterval)
            });
        });
    }
}

```

Листинг 25. Кэширование данных

Для чтения списка сущностей используется расширенный метод GetFromJsonAsync для HttpClient (свойство Http):

```

public async Task<List<T>> GetList<T>() where T : Entity
{
    ...
    string requestUri = GetRequestUri<T>("list");
    return await Http.GetFromJsonAsync<List<T>>(requestUri);
}

```

Листинг 26. Чтение списка сущностей

Для сохранения новой сущности выполняется метод ForDatabase для удаления возможных ссылок на другие сущности. Затем, после выполнения операции на стороне сервера с помощью расширенного метода ReadFromJsonAsync для HttpContent.

```

public async Task<T> AddNew<T>(T entity) where T : Entity
{
    ...
    string requestUri = GetRequestUri<T>();
    T entity2 = (T)entity.ForDatabase();
    var result = await Http.PostAsJsonAsync<T>(requestUri, entity2);
    entity = await result.Content.ReadFromJsonAsync<T>();
    Reset<T>();
}

```

Листинг 27. Сохранение новой сущности

Метод Reset сбрасывает кэш данных по ключу типа:

```

private void Reset<T>() where T : Entity => Cache.Remove(typeof(T));

```

Листинг 28. Очистка кэша данных по типу сущности

## Изменение порядка данных

Для групп, вопросов и ответов можно менять порядок представления для пользователя, сортируя данные по свойству Number. Эта функциональность определяется с помощью интерфейса IOrderedEntity:

```

public interface IOrderedEntity
{
    int Id { get; }
    int Number { get; set; }
}

```

```
}
```

Листинг 29. Интерфейс для упорядочивания данных по номеру

На стороне сервера (проект `Inquirer.Server`) в классе контекста данных `InquirerDbContext` добавлен метод `SwapOrdered` для изменения порядка данных в виде обмена номера для двух сущностей:

```
public void SwapOrdered<T>(T entity1, T entity2) where T : Entity, IOrderedEntity
{
    int number = entity1.Number;
    entity1.Number = entity2.Number;
    entity2.Number = number;
    Set<T>().UpdateRange(entity1, entity2);
    SaveChanges();
}
```

Листинг 30. Изменение порядка данных в контексте данных

Для взаимодействия клиента и сервера создан класс `SwapRequestData` как параметр метода контроллера:

```
public sealed class SwapRequestData<T> where T : Entity, IOrderedEntity
{
    public T Entity1;
    public T Entity2;
}
```

Листинг 31. Класс параметров для метода контроллера по изменению порядка данных

```
[HttpPut("swap")]
public void Swap(SwapRequestData<QuestionGroup> data)
{
    DbContext.SwapOrdered(data.Entity1, data.Entity2);
}
```

Листинг 32. Метод контроллера для обмена порядковыми номерами группы вопросов

Для обработки на клиентской стороне в сервис данных добавлен соответствующий метод:

```
public async Task Swap<T>(T entity1, T entity2) where T : Entity, IOrderedEntity
{
    ...
    string requestUri = GetRequestUri<T>();
    SwapRequestData<T> value = new()
    {
        Entity1 = entity1,
        Entity2 = entity2,
    };
    await Http.PutAsJsonAsync(requestUri, value);
    int number = entity1.Number;
    entity1.Number = entity2.Number;
    entity2.Number = number;
}
```

Листинг 33. Метод для изменения порядка на клиенте

Поскольку при изменении порядка мы не читаем повторно данные с сервера, то на клиенте повторно выполняется обмен номерами между сущностями.

## Безопасность

В учебном приложении используется стандартная программная инфраструктура ASP.NET по безопасности «как есть». Поэтому в данном руководстве она не рассматривается. Сценарии и



варианты использования описываются в технической документации: <https://docs.microsoft.com/en-us/aspnet/core/blazor/security>.

## Навигация

Шаблон проекта содержит набор файлов, в т.ч. страницы и компоненты для клиента в проекте `Inquirer.Client`. Самый простой способ разработки простого приложения, как наше, является их редактирование:

- `wwwroot/index.html` – стартовая страница, в которой задаются заголовок приложения, сообщения о загрузке приложения и возможной программной ошибке.
- `App.razor` – стартовый компонент приложения, в нем устанавливаются размещение по умолчанию и обработка ограничений и ошибок доступа.
- `Shared/NavMenu.razor` – боковая панель приложения.
- `Shared/LoginDisplay.razor` – локализация наименований команд для аутентификации.
- `Shared/MainLayout.razor` – размещение визуальных компонентов на странице.
- `Pages/Index.razor` – начальная страница приложения.



Размещение визуальных элементов: <https://docs.microsoft.com/en-us/aspnet/core/blazor/components/layouts>

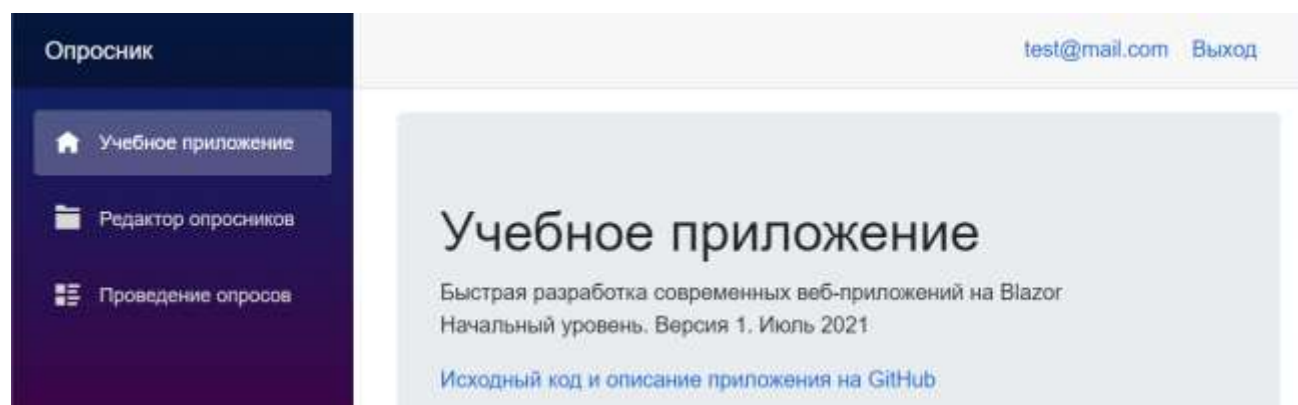


Рисунок 5. Начальная страница приложения и панель навигации

Файлы формата `razor` позволяют совмещать `html`-разметку и программный код на `C#`, который для приложений в формате `WebAssembly` выполняется на клиенте. В разметке можно использовать функции, свойства и поля, определенные в программном коде клиента или в проекте `Inquirer.Shared`: для этого перед именем добавляется символ «@». Например, в следующем листинге для класса или адресов используются локальные поля или статические свойства.

```
<div class="@NavMenuCssClass" @onclick="ToggleNavMenu">
  <ul class="nav flex-column">
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="" Match="NavLinkMatch.All">
        <span class="oi oi-home" aria-hidden="true"></span> Учебное приложение
      </NavLink>
    </li>
    <li class="nav-item px-3">
      <NavLink class="nav-link" href="@Routes.Questionnaire.List">
        <span class="oi oi-folder" aria-hidden="true"></span> Редактор опросников
      </NavLink>
    </li>
    <li class="nav-item px-3">
```

```

        <NavLink class="nav-link" href="@Routes.Survey.List">
            <span class="oi oi-list-rich" aria-hidden="true"></span> Проведение опросов
        </NavLink>
    </li>
</ul>
</div>

```

Листинг 34. Разметка навигации приложения (боковая панель)

Для страниц с помощью атрибута `page` задается один или несколько статических относительных путей, по которому данная страница доступна:

```

@page "/"

<div class="jumbotron">
    <h1>Учебное приложение</h1>
    <p>
        Быстрая разработка современных веб-приложений на Blazor<br />

```

Листинг 35. Навигация в разметке (Index.razor)

Функциональная структура приложения описывается с помощью навигации. Для наглядности можно описать ее в виде вложенных статических классов:

```

public static class Routes
{
    public static class Questionnaire
    {
        public const string List = "/questionnaire/list";
        public const string Form = "/questionnaire/form/{Id:int}";
    }
    public static class Survey
    {
        public const string List = "/survey/list";
        public const string Conducting = "/survey/conducting/{Id:int}";
    }
}

```

Листинг 36. Справочник маршрутов

Для использования маршрутов, определенных в программном коде вместо атрибута `@page` нужно использовать атрибут `Route`:

```

@attribute [Route(@Routes.Questionnaire.List)]
@attribute [Authorize]
@Inject IDataService DataService

<h1>Конструктор опросников</h1>

<div class="table-responsive">

```

Листинг 37. Навигация с помощью справочника маршрутов (QuestionnaireList.razor)

Атрибут `Authorize` обеспечивает проверку прав доступа пользователя к данной странице.

Директива `@inject` обеспечивает инъекцию сервиса данных для их чтения с сервера и определяет соответствующее свойство `DataService` (см. выше раздел «Настройка клиента»).



Маршрутизация ASP.NET Core Blazor: <https://docs.microsoft.com/en-us/aspnet/core/blazor/fundamentals/routing>

## Разметка страниц



Синтаксис Razor: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor>

Если страница содержит только статическую разметку, то ее можно разработать в формате html.

Динамические страницы, разметка которых генерируется на основании данных разрабатываются в формате razor. В этом формате можно совмещать в одном файле разметку html и программный код. Для этого после разметки создается секция @code. Программный код страницы можно выносить в отдельный файл .razor.cs.



Частичный файл с кодом компонента: <https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#partial-class-support>

В учебном приложении демонстрируются разные подходы при редактировании данных:

- Отдельные страницы для списков данных и их редактирования: QuestionnaireList.razor и QuestionnaireForm.razor (конструктор опросников).
- Встроенное редактирование данных в списке (таблице): QuestionGroupList.razor (группа вопросов).
- Редактирование данных в списке через диалог: QuestionList.razor (вопросы).

## Опросники

В списке опросников решаются следующие задачи:

1. Чтение списка сущностей с сервера (запрос к контроллеру).
2. Выбор определенного опросника и отображение контекстных команд для редактирования и удаления.
3. Переход к форме редактирования при выполнении команд для создания или редактирования.
4. Удаление выбранного опросника.

# Конструктор опросников

[Создать](#) | [Редактировать](#) | [Удалить](#)

Наименование	Автор	Баллы	Описание	Дата создания
Досуг и увлечения	test@mail.com	10.00	Вопросы по увлечениям и занятиям в свободное время	16.06.2021 19:22
Обучение и интересы	test@mail.com	5.00	Вопросы про обучение и интересы	26.06.2021 12:48
Работа и карьера	test@mail.com	5.00		26.06.2021 12:50

Рисунок 6. Список опросников

Чтение данных выполняется в перегруженном методе инициализации страницы, который имеет синхронную и асинхронную версии<sup>7</sup>:

```
@code {
    private List<Questionnaire> _data;

    protected override async Task OnInitializedAsync()
    {
        try
        {
            _data = await DataService.GetList<Questionnaire>();
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Листинг 38. Чтение данных для списка опросников (асинхронная версия).

Для обработки исключения доступа используется конструкция try catch, и при выполнении ошибки выполняется перенаправление пользователя для аутентификации пользователя.

В разметке можно использовать программный код в виде условий или циклов, для этого нужно, также как для директив и атрибутов в начале страницы, использовать символ «@».

При чтении данных с помощью проверки поля с данными на null выводится соответствующее сообщение:

```
@if (_data is null)
{
```

<sup>7</sup> Обычно асинхронные методы имеют в конце своего названия постфикс Async. Если все методы являются асинхронными, тогда этот постфикс может опускаться.

```

    }
    <p><em>Чтение данных...</em><span class="spinner-border text-primary"/> </p>

```

Листинг 39. Вывод сообщения при чтении данных



При изменении состояния класса страницы производится автоматическое обновление виртуальной страницы (в памяти), вычисление разницы между старым и новым образом, и инкрементное обновление разметки отображаемой в браузере страницы.

Для выбора опросника в таблице и визуализации выбранной строки в коде определяется свойство `SelectedItem` и метод `GetSelectedRowStyle`:

```

private Questionnaire SelectedItem { get; set; }
private string GetSelectedRowStyle(int id) => SelectedItem?.Id == id ? "bg-info" : "";

```

Листинг 40. Выбранная сущность и стиль для выбранной строки

Для html-элементов поддерживается привязка обработчиков событий, аналогично как java-script. Для этого имя события начинается с символа «@»:

```

foreach (var item in _data)
{
    <tr class="@GetSelectedRowStyle(item.Id)" @onclick="(a) => SelectedItem = item">
        <td>@item.Name</td>
        <td>@item.Author?.UserName</td>
    </tr>
}

```

Листинг 41. Разметка для выбора строки в таблице

В разметке задаются гиперссылка для перехода на форму редактирования для выбранного опросника, и кнопка для удаления. Обратите внимание на то, как для события нажатия кнопки `@onclick` вызывается асинхронный обработчик.

```

<a class="d-inline" href="@Routes.Parameterization(Routes.Questionnaire.Form,
0)">Создать</a>
@if (SelectedItem is not null)
{
    <span> | </span>
    <a class="d-inline" href="@Routes.Parameterization(Routes.Questionnaire.Form,
@SelectedItem.Id)">Редактировать</a>
    <span> | </span>
    <button class="btn btn-link d-inline" @onclick="(async () => { await DeleteAsync();
})">Удалить</button>
}

```

Листинг 42. Разметка для команд редактирования в списке опросников



Обработка событий: <https://docs.microsoft.com/en-us/aspnet/core/blazor/components/event-handling>

## Форма редактирования опросника

Для формы редактирования опросника решаются следующие задачи:

1. Редактирование данных по опроснику.
2. Просмотр и редактирование списка групп вопросов (встроенное редактирование в таблице).
3. Просмотр и редактирования списка вопросов (редактирование с помощью диалога).

# Опросник

Дата создания: 16.06.2021 19:22

[Список](#)

Форма	Группы <b>2</b>	Вопросы <b>4</b>
-------	-----------------	------------------

Наименование:	<input type="text" value="Досуг и интересы"/>
Автор:	<input type="text" value="test@mail.com"/>
Баллы для прохождения:	<input type="text" value="10.00"/>
Описание:	<input type="text" value="Вопросы по интересам и занятиям в свободное время"/>

[Сохранить](#)


Рисунок 7. Страница формы редактирования опросника со списком групп и вопросов

С помощью атрибута `Parameter`, применяемого для свойства компонента, можно передавать значения как значения или объектные ссылки:

```
[Parameter]  
public int Id { get; set; }
```

Листинг 43. Параметр идентификатора опросника

Значения параметров могут передаваться с помощью маршрутизации (в url, см. выше пример в «Листинг 42») или при использовании одного компонента в другом (композиции, см. ниже пример в «Листинг 48»).

	Параметры компонентов: <a href="https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#component-parameters">https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#component-parameters</a>
	Параметры маршрута: <a href="https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#route-parameters">https://docs.microsoft.com/en-us/aspnet/core/blazor/components/?view=aspnetcore-5.0#route-parameters</a>
	Каскадные значения и параметры: <a href="https://docs.microsoft.com/en-us/aspnet/core/blazor/components/cascading-values-and-parameters?view=aspnetcore-5.0">https://docs.microsoft.com/en-us/aspnet/core/blazor/components/cascading-values-and-parameters?view=aspnetcore-5.0</a>

Если параметр `Id` равен 0, то создается новый опросник (устанавливается по умолчанию первый пользователь в списке), иначе читается с сервера сущность вместе с группами, вопросами и ответами (см. выше «Листинг 17»):

```
public Questionnaire Model { get; set; }  
private EditContext editContext { get; set; }  
private List<ApplicationUser> users { get; set; }
```

```
protected override async Task OnInitializedAsync()
{
    users = await UserService.GetList();
    if (Id == 0)
    {
        Model = new() { AuthorId = users[0].Id };
    }
    else
    {
        try
        {
            Model = await DataService.Get<Questionnaire>(Id);
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
    editContext = new(Model);
}
```

Листинг 44. Чтение данных для формы редактирования опросника

При создании нового опросника либо редактировании существующего выводится соответствующий заголовок. Также выводится ссылка для возврата в список опросников:

```
@if (Id == 0)
{
    <h1>Новый опросник</h1>
}
else
{
    <h1>Опросник</h1>
    @if (model is not null)
    {
        <div>Дата создания: @model.CreateTime.ToString("dd.MM.yyyy HH:mm")</div>
    }
}

<p>
    <a href="@Routes.Questionnaire.List">Список</a>
</p>
```

Листинг 45. Разметка заголовка формы редактирования опросника

```
@if (model is not null && users is not null)
{
    @if (!model.IsNew)
    {
        <p class= "btn-group mt-2">
            <button type="button" class=@GetButtonStyleByMode(EditMode.Form) @onclick="()
=> { Mode = EditMode.Form; }">Форма</button>
            <button type="button" class=@GetButtonStyleByMode(EditMode.Groups) @onclick="()
=> { Mode = EditMode.Groups; }">
                Группы
                @if (model.Groups.Count > 0)
                {
                    <span class= "badge badge-primary">@model.Groups.Count</span>
                }
            </button>
        </p>
    }
```



```

        <button type="button" class=@GetButtonStyleByMode(EditMode.Questions)
@onclick="() => { Mode = EditMode.Questions; }">
            Вопросы
            @if (model.Questions.Count > 0)
            {
                <span class="badge badge-primary">@model.Questions.Count</span>
            }
        </button>
    </p>
}

```

Листинг 46. Разметка для выбора редактируемых данных

Помимо редактирования самого опросника редактируются также группы вопросов и сами вопросы. Соответственно, в разметке и в коде реализованы перечисления для выбора режима редактирования и функция для выбора стилей для переключателей:

```

public enum EditMode { Form, Groups, Questions }
private EditMode Mode { get; set; } = EditMode.Form;
private string GetButtonStyleByMode(EditMode mode) => Mode == mode ? "btn btn-info" :
"btn btn-outline-info";

```

Листинг 47. Код для выбора редактируемых данных

Для редактирования данных в форме Blazor предоставляет набор компонентов. На форме редактирования опросника используются следующие компоненты:

- **EditForm** – группирует элементы ввода данных для отправки на сервер.
- **DataAnnotationsValidator** – проверяет данные в элементах внутри формы на основе декларативных правил (см. выше «Листинг 9»).
- **ValidationSummary** – выводит сообщение об ошибке при проверке данных, если есть.
- **InputText**, **InputSelect**, **InputNumber**, **InputTextArea** – элементы ввода данных, обрабатываемые внутри формы.

```

@if (Mode == EditMode.Form)
{
    <EditForm EditContext="@editContext" OnValidSubmit="@HandleValidSubmit">
        <div class="card mt-2 mb-3 p-3">
            <DataAnnotationsValidator />
            <ValidationSummary />
            <div class="row m-2">
                <label class="col-4">Наименование:</label>
                <InputText class="form-control col-8" @bind-Value="model.Name"
DisplayName="Наименование" />
            </div>
            <div class="row m-2">
                <label class="col-4">Автор:</label>
                <InputSelect class="form-control col-8" @bind-Value="model.AuthorId">
                    @foreach (var user in users)
                    {
                        <option value="@user.Id">@user.UserName</option>
                    }
                </InputSelect>
            </div>
            <div class="row m-2">
                <label class="col-4">Баллы для прохождения:</label>
                <InputNumber class="form-control col-xl-1 col-2" @bind-Value="model.Score"
DisplayName="Баллы для прохождения" />
            </div>
        </div>
    </EditForm>
}

```

```

        <div class="row m-2">
            <label class="col-4">Описание:</label>
            <InputTextArea class="form-control col-8" @bind-Value="model.Description" />
        </div>
    </div>
    <button class="btn btn-primary" type="submit">Сохранить</button>
</EditForm>
}
else if (Mode == EditMode.Groups)
{
    <QuestionGroupList ParentComponent="this"/>
}
else if (Mode == EditMode.Questions)
{
    <QuestionList ParentComponent="this"/>
}
}

```

Листинг 48. Разметка для формы редактирования опросника

Сохранение данных происходит при возникновении стандартного события Submit, которое запускается кнопкой с соответствующим типом ([https://www.w3schools.com/html/html\\_forms.asp](https://www.w3schools.com/html/html_forms.asp)).

В атрибуте OnValidSubmit элемента EditForm вызывается метод для сохранения опросника, который вызывается при нажатии кнопки и успешной проверке данных:

```

private async Task HandleValidSubmit()
{
    if (model.IsNew)
    {
        model = await DataService.AddNew(model);
    }
    else
    {
        await DataService.Update(model);
    }
}

```

Листинг 49. Сохранение данных формы редактирования опросника



Форма и валидация данных: <https://docs.microsoft.com/en-us/aspnet/core/blazor/forms-validation>

Списки групп и вопросов реализованы в виде отдельных компонентов.

## Группа вопросов

Список группы вопросов для выбранного опросника реализован в виде отдельного компонента QuestionGroupList.razor, который в качестве параметров получает родительский компонент.

```

[Parameter]
public QuestionnaireForm ParentComponent { get; set; }
public Questionnaire Model => ParentComponent.Model;

```

Листинг 50. Параметры списка групп вопросов

Просмотр и редактирование данных выполняется в таблице. В режиме просмотра показываются команды для перемещения, создания, редактирования и удаления.

Форма

Группы 2

Вопросы 4

#	Наименование	Баллы	Описание	Создать
1 ▼	Отдых	10.00		Изменить Удалить
2 ▲	Спорт	20.00		Изменить Удалить

Рисунок 8. Список групп вопросов в опроснике

При переходе в режим редактирования они скрываются, и отображаются команды для сохранения изменений данных или отмены редактирования.

Форма

Группы 2

Вопросы 4

#	Наименование	Баллы	Описание	
1	Отдых	10.00		
2	<input type="text" value="Спорт"/>	<input type="text" value="20.00"/>	<input type="text"/>	Сохранить Отмена

Рисунок 9. Редактирование в таблице групп вопросов

Для редактируемой группы определено свойство EditedItem, которое устанавливается и очищается в методах редактирования и его завершения:

```

private QuestionGroup EditedItem { get; set; }

private void AddNew()
{
    Model.Groups.Add(EditedItem = new QuestionGroup
    {
        QuestionnaireId = Model.Id,
        Number = Model.Groups.Count == 0 ? 1 : Model.Groups.Max(t => t.Number) + 1
    });
}
private void Edit(QuestionGroup item)
{
    EditedItem = item;
}
private async Task DeleteAsync(QuestionGroup item)
{
    await DataService.Delete<QuestionGroup>(item.Id);
    Model.Groups.Remove(item);
    ParentComponent.Refresh();
}

```

Листинг 51. Программный код редактирования группы вопросов в таблице

В родительском компоненте определен метод для обновления представления Refresh, поскольку при вложенных компонентах не всегда Blazor определяет корректно измененные области.

```
private async Task Save()
{
    if (EditedItem.IsNew)
    {
        var savedItem = await DataService.AddNew(EditedItem);
        EditedItem.Id = savedItem.Id;
        ParentComponent.Refresh();
    }
    else
    {
        await DataService.Update(EditedItem);
    }
    EditedItem = null;
}
```

Листинг 52. Сохранение и завершение редактирования группы вопросов

## Список вопросов

Список вопросов для выбранного опросника также реализован в виде отдельного компонента QuestionList.razor, который в качестве параметров получает родительский компонент. В нем реализован третий подход при построении интерфейса пользователя для редактирования данных: таблица данных для просмотра, и диалог для редактирования.

Форма

Группы 3

Вопросы 4

Создать

Редактировать

Удалить

#	Вопрос	Группа	Ответы	Баллы
1	Когда вы засыпаете?	Отдых	Ранко	1.00
			Позднее	0.00
2	Делаете ли вы зарядку?	Спорт	Да	2.00
			Нет	0.00
3	Отдыхаете ли вы после обеда?	Отдых	Да	1.00
			Нет	0.00

Рисунок 10. Список вопросов в режиме просмотра

В таблице выводятся два уровня данных: для каждого вопроса еще выводятся ответы и баллы.

```
foreach (var item in Model.Questions)
{
    <tr class="@GetSelectedRowStyle(item.Id)" @onclick="(a) => SelectedItem = item">
        <th scope="row" rowspan="@item.Answers.Count">@item.Number</th>
        <td rowspan="@item.Answers.Count">@item.Text</td>
        @if (Model.Groups.Count > 0)
        {
            <td rowspan="@item.Answers.Count">@GetGroupName(item)</td>
        }
        <td>
            @if (item.Answers.Count > 0)
            {
                @item.Answers.OrderBy(t => t.Number).First().Text
            }
        }
    </tr>
}
```

```

    }
    </td>
    <td>
        @if (item.Answers.Count > 0)
        {
            @item.Answers.OrderBy(t => t.Number).First().Score
        }
    </td>
</tr>
@foreach (var ans in item.Answers.OrderBy(t => t.Number).Skip(1))
{
    <tr class="@GetSelectedRowStyle(item.Id)">
        <td>@ans.Text</td>
        <td>@ans.Score</td>
    </tr>
}
}

```

Листинг 53. Разметка для таблицы вопросов с ответами

Для этого используется атрибут `rowspan` для колонок вопросов, а для вывода колонок с ответами используется вложенный цикл.

Для отображения диалога используется техника с выделенной строкой (см. в разделе «Опросы») и командой редактирования, а также соответствующие стили из css библиотеки Bootstrap.

```

@if (EditedItem is not null)
{
    <div class="modal show" tabindex="-1" role="dialog" style="display:block">
        <div class="modal-dialog modal-dialog-centered" role="document">
            <div class="modal-content">
                <div class="modal-header">
                    <h5 class="modal-title">Вопрос</h5>
                </div>
                <div class="modal-body">
                    <QuestionForm Questionnaire="Model" Question="EditedItem" />
                </div>
                <div class="modal-footer">
                    <button type="reset" class="btn btn-secondary" data-bs-dismiss="modal"
@onclick="() => EditedItem = null">Закрыть</button>
                </div>
            </div>
        </div>
    </div>
}

```

Листинг 54. Разметка для отображения формы редактирования как диалога

### Форма редактирования вопроса и ответов

Форма редактирования вопроса с ответами реализована как отдельный компонент `QuestionForm`. В ней совмещается собственно форма для вопроса и таблица с вариантами ответов:

Вопрос

Текст вопроса:

Группа:

[Сохранить вопрос](#)

Ответы

#	Текст	Баллы	Создать
1	Рано	1.00	<a href="#">Изменить</a> <a href="#">Удалить</a>
2	Поздно	0.00	<a href="#">Изменить</a> <a href="#">Удалить</a>

[Закреть](#)

Рисунок 11. Форма для редактирования вопроса и ответов

Обратите внимание, что сохранение данных по вопросу и ответам выполняется отдельными командами и не зависит друг от друга. Техника реализации аналогична описанным в предыдущих разделах.

## Проведение опросов

Для проведения опросов выполняются следующие задачи:

1. Выбор опросника.
2. Проведение опроса и сохранение результатов.

Для выбора опросника на странице SurveyList.razor отображаются карточки с их наименованием и описанием:

## Проведение опросов

Досуг и увлечения

Вопросы по увлечениям и занятиям в свободное время

[Начать опрос](#)

Обучение и интересы

Вопросы про обучение и интересы

[Начать опрос](#)

Работа и карьера

[Начать опрос](#)

Рисунок 12. Выбор опросника

Для разметки используются стили Bootstrap:

```
<div class="card-deck">
  @foreach (var item in _data)
  {
    <div class="card thumbnail">
      <div class="card-body">
        <h5 class="card-title">@item.Name</h5>
        <p class="card-text">@item.Description</p>
      </div>
      <div class="card-footer">
```

```

        <a href="@Routes.Parameterization(@Routes.Survey.Conducting,
@item.Id)">Начать опрос</a>
    </div>
</div>
}
</div>

```

Листинг 55. Разметка для карточек опросников



Размещение карточек: <https://getbootstrap.com/docs/4.0/components/card/#card-groups>,  
<https://getbootstrap.com/docs/4.0/components/card/#card-decks>

При выборе опросника выполняется переход на страницу для проведения опроса SurveyConducting.razor.

При переходе на эту страницу автоматически создается новый опрос для текущего пользователя:

```

Questionnaire = await DataService.Get<Questionnaire>(Id);
if (Questionnaire.Questions.Count > 0)
{
    Survey = new Survey { QuestionnaireId = Id };
    Survey = await DataService.AddNew(Survey);
}

```

Листинг 56. Создание опроса на клиенте

В контроллере опроса определяется текущий пользователь и задается время создания:

```

protected override void NewEntity(Survey entity)
{
    entity.RespondentId = HttpContext.User.GetId();
    entity.CreateTime = System.DateTime.Now;
}

```

Листинг 57. Создание опроса на сервере

По выбранному опроснику создается список вопросов и вариантов ответов:

## Досуг и увлечениям

Вопросы по увлечениям и занятиям в свободное время

Дата создания: 21.07.2021 10:47

Пожалуйста, ответьте на следующие вопросы:

Когда вы засыпаете?

- ☐ Рано
- ☐ Поздно

Делаете ли вы зарядку?

- ☐ Да
- ☐ Нет

Отдыхаете ли вы после обеда?

- ☐ Да
- ☐ Нет

С кем вы занимаетесь?

- ☐ Друзья
- ☐ Коллеги
- ☐ Тренер
- ☐ Самостоятельно

Завершение опроса

Рисунок 13. Проведение опроса



В зависимости от количества вариантов правильных ответов (балл больше нуля) для каждого вопроса выбирается тип элемента управления:

- Единственный правильный ответ: radio.
- Несколько правильных ответов: checkbox.

```
@foreach (var ans in qst.Answers)
{
    <div class="form-check">
        <input class="form-check-input"
            id="@("answer" + ans.Id)"
            name="@("answer" + (qst.IsSingle() ? qst.Id : ans.Id))"
            type="@((qst.IsSingle() ? "radio" : "checkbox"))"
            @onchange="async (a) => { await SetAnswer(qst, ans, a.Value); }"/>
        <label class="form-check-label" for="@("answer" + ans.Id)">@ans.Text</label>
    </div>
}
```

Листинг 58. Разметка для вывода вариантов ответов на вопрос

Метод определения типа вопроса IsSingle реализован как расширенный – в отдельном статическом классе EntityExtensions.

```
public static class EntityExtensions
{
    public static bool IsSingle(this Question question)
    {
        if (question.Answers is null)
        {
            throw new
ArgumentNullException($"{nameof(Question)}.{nameof(Question.Answers)} (Id={question.Id})");
        }
        if (question.Answers.Count == 0)
        {
            throw new
IndexOutOfRangeException($"{nameof(Question)}.{nameof(Question.Answers)}.Count == 0
(Id={question.Id})");
        }
        if (question.Answers.Count(t => t.Score > 0) == 0)
        {
            throw new
ArgumentOutOfRangeException($"{nameof(Question)}.{nameof(Question.Answers)}
(Id={question.Id}): no right answers");
        }
        return question.Answers.Count(t => t.Score > 0) == 1;
    }
}
```

Листинг 59. Метод расширения для определения типа вопроса

При выборе варианта ответа формируется стек программных вызовов:

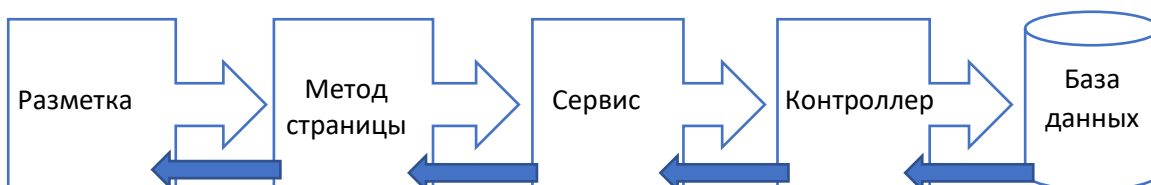


Рисунок 14. Стек программных вызовов и обратный поток данных как результат выполнения

Для корректной обработки выбора пользователем варианта ответа важно определить тип вопроса: альтернативный или множественный (см. выше «Листинг 59»):

```
private async Task SetAnswer(Question question, QuestionAnswer answer, object value)
{
    if (question.IsSingle())
    {
        await SurveyService.SetAnswer(Survey, question, answer, single: true);
    }
    else
    {
        if ((bool)value)
        {
            await SurveyService.SetAnswer(Survey, question, answer, single: false);
        }
        else
        {
            await SurveyService.RemoveAnswer(Survey, answer);
        }
    }
}
```

Листинг 60. Метод страницы для выбора варианта ответа

После выбора вариантов ответа пользователь завершает опрос и программа отмечает время завершения:

```
private void CompleteInterview()
{
    Survey.CompleteTime = DateTime.Now;
    DataService.Update(Survey);
    NavigationManager.NavigateTo(Routes.Survey.List);
}
```

Листинг 61. Метод страницы для завершения опроса и перехода на страницу списка опросников

## Сервис опросника

На клиенте разработан сервис (интерфейс и класс реализации) для обработки выбора пользователем вариантов ответа при проведении опроса:

```
public interface ISurveyService
{
    Task<SurveyAnswer> SetAnswer(Survey survey, Question question, QuestionAnswer answer, bool single);
    Task RemoveAnswer(Survey survey, QuestionAnswer answer);
}
```

Листинг 62. Интерфейс сервиса проведения опроса

При реализации сервиса используются соответствующие методы на стороне сервера – в контроллере вариантов ответа опроса:

```
public async Task<SurveyAnswer> SetAnswer(Survey survey, Question question, QuestionAnswer answer, bool single)
{
    try
    {
        string requestUri =
            $"api/{nameof(SurveyAnswer).ToLower()}/{survey.Id}/{question.Id}/{answer.Id}/{single}";
        Logger.LogDebug($"{nameof(SurveyService)}.{nameof(SetAnswer)}: {requestUri}");
        HttpRequestMessage httpRequest = new ()
```

```

        {
            Method = HttpMethod.Post,
            RequestUri = new Uri(requestUri, UriKind.Relative)
        };
        var result = await Http.SendAsync(httpRequest);
        var entity = await result.Content.ReadFromJsonAsync<SurveyAnswer>();
        return entity;
    }
    catch (Exception ex)
    {
        Logger.LogError($"{nameof(SurveyService)}.{nameof(SetAnswer)}":
{ex.GetBaseException().Message}");
        throw;
    }
}

```

Листинг 63. Метод сервиса для выбора варианта ответа

```

public async Task RemoveAnswer(Survey survey, QuestionAnswer answer)
{
    try
    {
        string requestUri =
            $"{api}/{nameof(SurveyAnswer).ToLower()}/{survey.Id}/{answer.Id}";
        Logger.LogDebug($"{nameof(SurveyService)}.{nameof(RemoveAnswer)}":
{requestUri}");
        await Http.DeleteAsync(requestUri);
    }
    catch (Exception ex)
    {
        Logger.LogError($"{nameof(SurveyService)}.{nameof(RemoveAnswer)}":
{ex.GetBaseException().Message}");
        throw;
    }
}

```

Листинг 64. Метод сервиса для удаления варианта ответа

## Контроллер ответов опросника

На стороне сервера в контроллере вариантов ответов опроса в зависимости от типа вопроса (альтернативный или множественный) выполняется обработка данных: если вопрос альтернативный, то нужно удалить предыдущий вариант ответа, если он есть.

```

[HttpPost("{surveyId}/{questionId}/{answerId}/{single}")]
public async Task<SurveyAnswer> SetAnswer(int surveyId, int questionId, int answerId,
bool single)
{
    DbContext.Database.BeginTransaction();
    if (single)
    {
        var prevAnswer = DbContext
            .SurveyAnswers
            .Include(t => t.QuestionAnswer)
            .Where(t => t.SurveyId == surveyId && t.QuestionAnswer.QuestionId ==
questionId)
            .FirstOrDefault();
        if (prevAnswer is not null)
        {
            DbContext.SurveyAnswers.Remove(prevAnswer);
        }
    }
}

```

```

    var answer = new SurveyAnswer
    {
        SurveyId = surveyId,
        QuestionAnswerId = answerId,
    };
    DbContext.SurveyAnswers.Add(answer);
    await DbContext.SaveChangesAsync();
    await DbContext.Database.CommitTransactionAsync();
    return answer;
}

```

*Листинг 65. Метод контроллера для выбора варианта ответа*

Либо нужно удалить вариант ответа при отмене выбора для множественного вопроса:

```

[HttpDelete("{surveyId}/{answerId}")]
public async Task<bool> RemoveAnswer(int surveyId, int answerId)
{
    var answer = await DbContext.SurveyAnswers.FirstOrDefaultAsync(t => t.SurveyId ==
surveyId && t.QuestionAnswerId == answerId);
    if (answer is not null)
    {
        DbContext.SurveyAnswers.Remove(answer);
        await DbContext.SaveChangesAsync();
        return true;
    }
    return false;
}

```

*Листинг 66. Метод контроллера для удаления варианта ответа*