Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный технический
университет имени П. О. Сухого»

Кафедра «Информатика»

Т. Л. Романькова

ИЗБРАННЫЕ ГЛАВЫ ИНФОРМАТИКИ

ПРАКТИКУМ
по выполнению лабораторных работ для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Гомель 2024

Библиотека ГГТУ им.П.О.Сухого

УДК 004(075.8)
ББК 32.81я73
Р69

Рекомендовано научно-методическим советом
факультета автоматизированных и информационных систем
ГГТУ им. П. О. Сухого
(*протокол № 13 от 29.06.2022 г.*)

Рецензент: зав. каф. «Информационные технологии» ГГТУ им. П. О. Сухого
канд. техн. наук, доц. *К. С. Курочка*

Романькова, Т. Л.

Р69 Избранные главы информатики : практикум по выполнению лаборатор. работ для студентов специальности 1-40 04 01 «Информатика и технологии программирования» / Т. Л. Романькова. – Гомель : ГГТУ им. П. О. Сухого, 2024. – 48 с. – Систем. требования: PC не ниже Intel Celeron 300 МГц ; 32 Mb RAM ; свободное место на HDD 16 Mb ; Windows 98 и выше ; Adobe Acrobat Reader. – Режим доступа: https://elib.gstu.by. – Загл. с титул. экрана.

Содержит краткие теоретические сведения, методические указания для решения задач, задания для освоения разработки программ на базе платформы .Net.

Для студентов специальности 1-40 04 01 «Информатика и технологии программирования».

УДК 004(075.8)
ББК 32.81я73

© Учреждение образования «Гомельский государственный технический университет имени П. О. Сухого», 2024

Библиотека ГГТУ им. П.О.Сухого

Содержание

Библиотека ГГТУ им. П.О. Сухого

1. Архитектурные принципы разработки приложений

При разработке архитектуры приложений и проектировании программных решений нужно учитывать удобство поддержки, предусматривать возможность расширения и модификации разработанных программных продуктов.

Можно выделить следующие общие принципы проектирования, позволяющие создавать приложения, состоящие из слабо связанных друг с другом компонентов:

- разделение задач;
- инкапсуляция;
- инверсия зависимостей;
- явные зависимости;
- единственная обязанность;
- отсутствие дублирования;
- независимость сохранения состояния.

Принцип **разделения задач** подразумевает разделение программного обеспечения на компоненты в соответствии с выполняемыми ими функциями. Для соблюдения этого принципа нужно отделять бизнес-логику от функций пользовательского интерфейса и инфраструктуры доступа к данным. В идеальном случае код, реализующий основной функционал приложения, следует размещать в отдельном проекте, который **не должен зависеть от других проектов в приложении**. Такое разделение обеспечивает простоту тестирования бизнес-модели и ее возможное изменение.

Инкапсуляция отдельных компонентов приложения позволяет изолировать их друг от друга. Использование этого принципа позволяет получить слабо связанную модульную структуру приложения, так как при сохранении интерфейса объекты можно легко заменять другими реализациями. Части приложений и сами приложения должны предоставлять интерфейсы, которые будут использоваться участниками совместной работы вместо конкретных реализаций.

Инверсия зависимостей подразумевает, что модули верхних уровней не должны зависеть от модулей нижних уровней, оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. На соблюдении принципа инверсии зависимостей базируется реализация внедрения зависимостей.

Библиотека ГГТУ им. П.О. Сухого

Принцип **явных зависимостей** предполагает, что методы и классы должны явно требовать наличия всех объектов, которые необходимы для их корректного функционирования. Для соблюдения этого принципа необходимо, чтобы конструкторы и методы содержали в списке параметров все объекты, которые им необходимы для сохранения корректного состояния и правильного функционирования.

Принцип **единственной обязанности** применяется к объектно-ориентированному проектированию, но также может рассматриваться и как архитектурный принцип аналогично разделению задач. Этот принцип подразумевает, что объекты должны иметь только одну обязанность и только одну причину для изменения. Принцип единственной обязанности может применяться к слоям приложения. Например, обязанность представления относится к проекту пользовательского интерфейса, а обязанность доступа к данным будет отнесена к проекту инфраструктуры. Соответственно, бизнес-логика также должна находиться в отдельном проекте приложения, где ее можно тестировать и модернизировать независимо от других обязанностей.

Принцип **отсутствия дублирования** подразумевает, что повторяющуюся в приложении логику следует оформить в виде отдельной конструкции программирования, которая должна быть **единственным исполнителем нужного поведения** и использоваться в любых других частях приложения в тех случаях, когда требуется реализовать это поведение.

Принцип **независимости сохранения состояния** позволяет сохранять состояние бизнес-модели различными способами. Способы хранения данных со временем могут изменяться. Например, вместо одной технологии базы данных может использоваться другая. Принцип независимости сохранения состояния относится к типам, для которых требуется сохранение состояния, но код которых не зависит от выбираемой для этих целей технологии. В .NET такие типы иногда называются простыми объектами CLR (POCO).

Существуют различные виды архитектуры приложений, реализующие описанные выше принципы. Многие традиционные приложения .NET развертываются в виде одного элемента, соответствующего исполняемому файлу, или одного веб-приложения. Это так называемые **монолитные** приложения. Монолитное приложение полностью замкнуто в контексте поведения. Приложения с монолитной архитектурой можно разделить на **комплексные** и **традиционные приложения с *N*-слойной архитектурой.**

Библиотека ГГТУ им. П.О. Сухого

Архитектура комплексного приложения содержит как минимум один проект. В таком случае вся логика приложения заключена в одном проекте, компилируется в одну сборку и развертывается как один элемент. В сценарии с одним проектом разделение задач реализуется с помощью папок. Создаваемый в Visual Studio проект ASP.NET Core MVC представляет собой комплексный монолитный проект. На рис. 1.1 приводится файловая структура приложения, состоящего из одного проекта. При такой организации детали презентации данных в максимально возможной степени размещаются в папке представлений (Views), а детали реализации доступа к данным должны быть ограничены классами, содержащимися в папке данных. Бизнес-логика при этом размещается в службах и классах, находящихся в папке моделей (Models).

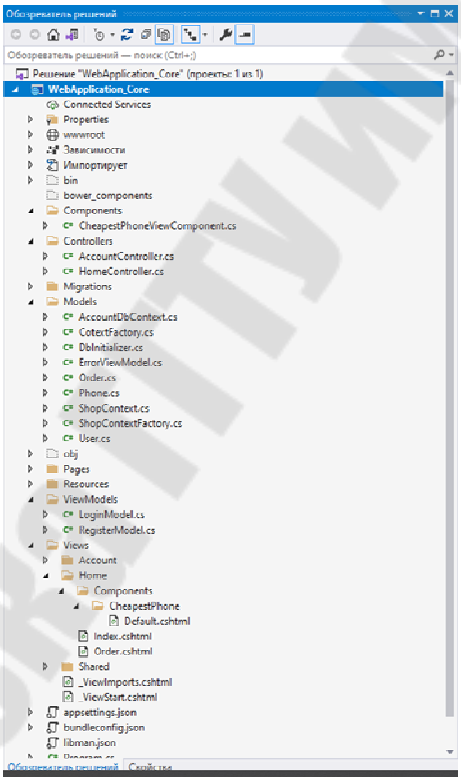

Рис. 1.1. Приложение ASP.NET Core MVC, состоящее из одного проекта

Приложения с *N*-слойной архитектурой организуются в виде решений, состоящих из нескольких проектов, где каждый проект размещается в отдельном слое приложения. Общепринятая организация логики приложения по слоям показана на рис. 1.2. Как правило,

Библиотека ГГТУ им. П.О. Сухого

в приложении определяются слои пользовательского интерфейса, бизнес-логики и доступа к данным.

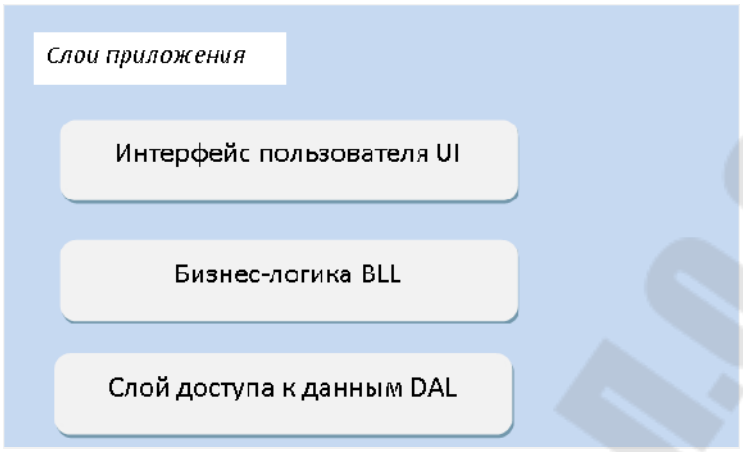

Рис.1.2. Архитектура трехслойного приложения

На рис. 1.3 показан пример решения, в котором приложение разделено на три проекта (слоя) в соответствии с определенными обязанностями.

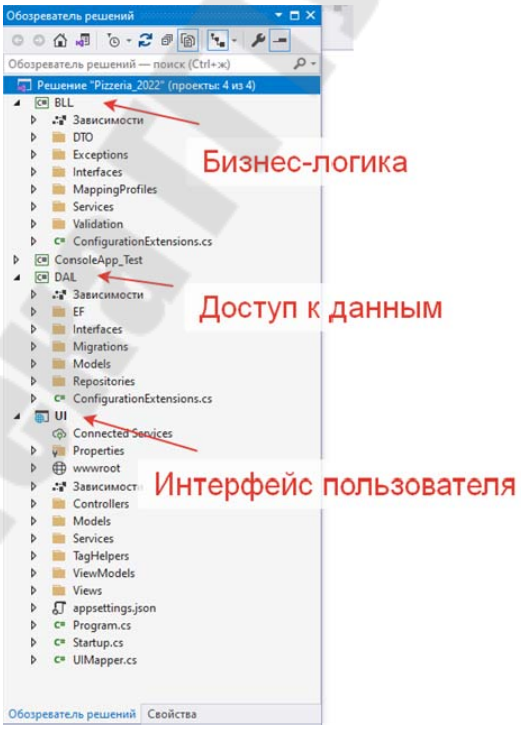

Рис. 1.3. Структура монолитного трехслойного приложения, состоящего из трех проектов

Библиотека ГГТУ им. П.О.Сухого

Еще один вид архитектуры приложений – микросервисная архитектура, подход к разработке программного обеспечения, при котором приложение разбивается на небольшие автономные компоненты (микросервисы) с четко определенными интерфейсами. Микросервисы отталкиваются от бизнес-логики. Каждый из сервисов отвечает за конкретную бизнес-задачу, имеет собственное хранилище данных и общается с другими сервисами через простые API-интерфейсы для решения более сложных задач.

2. Разработка слоя доступа к данным в приложении с многослойной архитектурой

2.1. Краткие теоретические сведения

2.1.1. Использование ADO.Net для доступа к данным

ADO.Net – часть библиотеки .Net, в которых определены пространства имен для работы с реляционными базами данных. Существует три режима (уровня) использования ADO.NET.

Подключенный уровень: приложение явно подключается к хранилищу данных, работает с ним, а затем отключается от него. Используются объекты подключения, команд и чтения данных.

Автономный (отключенный) уровень: приложение работает с копией внешних данных через объект DataSet, содержащий набор объектов DataTable. При этом данные обрабатываются без затрат на сетевой трафик. При необходимости внести изменение в хранилище данных подключение открывается заново для проведения обновлений в базе, а затем сразу же закрывается.

Entity Framework: приложение взаимодействует с базой данных через объектную модель. То есть данные представляются не как таблица из строк и столбцов, а как коллекция строго типизированных объектов (сущностей).

Подключенный уровень ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, чтения данных и команд.

Базовый класс для всех объектов подключения – абстрактный класс **DbConnection** из пространства имен **System.Data.Common**. Позволяет подключаться к хранилищу данных и отключаться от него.

Библиотека ГГТУ им. П.О. Сухого

Базовый класс для всех объектов команд – абстрактный класс **DbCommand** из пространства имен **System.Data.Common**. Представляет SQL-запрос или хранимую процедуру, объекты команд предоставляют доступ к объекту чтения данных.

Базовый класс для всех объектов чтения – абстрактный класс **DbDataReader** из пространства имен **System.Data.Common.** Предоставляет доступ к данным только для чтения в прямом направлении.

Существуют различные поставщики данных (провайдеры), в которых определены производные от этих базовых типов. Для каждого источника данных в ADO.NET может быть свой провайдер. Поставщик данных – это набор классов ADO.NET, предназначенных для работы с определенной базой данных, которые представляют собой конкретные реализации абстрактных базовых классов.

Классы в поставщиках имеют *префикс*, указывающий на имя провайдера: **SqlDataReader** , **OleDbCommand, OdbcConnection и др.**

Для соединения с базой данных нужно создать объект подключения соответствующего поставщика, передав ему строку подключения через параметр конструктора или через свойство ConnectionString.

Например,

```
SqlConnection cnn = new SqlConnection(ConnectionString);
```

Строка подкючения содержит ряд пар «имя/значение», разделенных точками с запятой. Смысл каждой пары «имя/значение» для специфичной СУБД можно узнать в описании свойства ConnectionString объекта подключения для соответствующего поставщика данных в документации .NET. Например,

```
string connectString =
                  @"Data Source=(localdb)\mssqllocaldb;
                     Initial Catalog=Library;
                     Integrated Security=true;";
```

или

```
string connectString = @"Server=(localdb)\mssqllocaldb;
                       Database=Library;
                       Trusted_Connection=True;";
```

Для формирования строки подключения в провайдерах данных есть специальный класс **ConnectionStringBuilder (с соответствующим префиксом).** Например,

```
OleDbConnectionStringBuilder csb =
                        new OleDbConnectionStringBuilder();
csb.DataSource = @"F:\Univer.mdb";
```


Библиотека ГГТУ им. П.О. Сухого

```
csb.Provider = "Microsoft.Jet.OLEDB.4.0";

OleDbConnection cnn =
                    OleDbConnection(csb.ConnectionString);
```

Задавать строки подключения можно в файле конфигурации **App.config,** например, в секции **connectionStrings.** Например,

```
<connectionStrings>
  <add name="library"
     connectionString="Data source=(localdb)\mssqllocaldb;
     Initial catalog=Library;
     Integrated security=true;"/>
 </connectionStrings>
```

Тогда в программе для чтения строки из этой секции можно применить класс **ConfigurationManager** из пространства имен **System.Configuration** (необходимо добавить ссылку на сборку System.Configuration.dll**).** Например**,**

```
stringcnStr =
        ConfigurationManager.ConnectionStrings["library"]
                                .ConnectionString;
```

В типах данных, производных от DbConnection, есть следующие элементы.

Метод **Open()** устанавливает соединение с СУБД.

Метод **Close()** закрывает подключение к источнику данных.

Метод **ChangeDatabase()** изменяет базу данных, связанную с открытым подключением.

Свойство **ConnectionTimeout** возвращает время ожидания при подключении, после которого ожидание прекращается и выдается сообщение об ошибке (значение по умолчанию 15 секунд). Только для чтения. Изменить этот параметр можно через строку подключения.

Тип DbConnection реализует интерфейс IDisposable, поэтому объекты этого типа можно использовать в инструкции using.

Для отправки команд базе данных нужно использовать объект команды – объект класса, производного от **DbCommand**, соответствующего провайдера.

DbCommand – это объектно-ориентированное представление SQL-запроса, имени таблицы или хранимой процедуры.

Создать объект команды можно с использованием конструктора без параметров, после чего задать подключение и строку запроса через свойства. Например,

```
OleDbCommand comand = new OleDbCommand();
```


Библиотека ГГТУ им. П.О.Сухого

```
comand.Connection = cnn;
comand.CommandText = "SELECT * FROM Students";
```

Здесь cnn – объект класса OleDbConnection.

Можно использовать конструктор с параметрами.

```
OleDbCommand comand =
   new OleDbCommand("SELECT * FROM Students",cnn);
```

Можно также воспользоваться методом **CreateCommand()** объекта подключения.

Ниже приводятся некоторые элементы типа **DbCommand.**

Свойство **Parameters** представляет коллекцию параметров (объектов класса **DbParameter**), используемых для параметризированного запроса.

Метод **Cancel ()** отменяет выполнение команды.

Метод **ExecuteReader()** выполняет SQL-запрос и возвращает объект **DbDataReader** соответствующего поставщика данных, который предоставляет доступ к результату запроса.

Метод **ExecuteNonQuery()** выполняет SQL-оператор, отличный от запроса (например, вставка, обновление, удаление или создание таблицы).

Ниже приведен пример параметризованного запроса.

```
    string sql = string.Format("Insert Into Authors" +
"(FirstName, LastName) Values(@FirstName, @LastName)");
    using (SqlCommand cmd = new SqlCommand(sql, connect))
     {
       // Добавить параметры
       cmd.Parameters.AddWithValue("@FirstName", fn);
       cmd.Parameters.AddWithValue("@LastName", ln);
       cmd.ExecuteNonQuery();
     }
```

Объекты чтения данных – это объекты классов, производных от **DbDataReader,** которые представляют поток данных, допускающий только чтение в прямом направлении, и возвращают каждый раз по одной записи. Объект чтения данных можно получить из объекта команды вызвав метод **ExecuteReader ().**

OleDbDataReader dtReader = cmd.ExecuteReader();

Объект чтения данных представляет текущую запись, прочитанную из базы данных. Получить значения полей этой записи можно с помощью индексатора. Есть индексатор с целым индексом (нумера-

Библиотека ГГТУ им. П.О. Сухого

ция с нуля) и индексатор со строковым индексом. В качестве индекса выступает имя поля. Например, **dtReader["Fam"], dtReader[1].**

Метод **Read()** перемещает объект чтения к следующей записи. Возвращает true, если чтение возможно.

```
SqlCommand cmd = new SqlCommand();
cmd.Connection = cnn;
cmd.CommandText = "SELECT * FROM Authors";
using (SqlDataReader dtReader = cmd.ExecuteReader())
{
     while (dtReader.Read())
     {
         richTextBox1.Text +=
              dtReader["FirstName"].ToString() + "  " +
              dtReader["LastName"].ToString() + "\n";
     }
 }
```

2.1.2. Паттерн Репозиторий

Паттерн **Репозиторий** – один из наиболее часто используемых паттернов при работе с данными. Назначение: отделение бизнес-логики от деталей реализации слоя доступа к данным.

Репозитории – это классы или компоненты, которые содержат логику, необходимую для доступа к источникам данных. Репозитории предоставляют функционал для доступа к данным, позволяя отделить технологию, используемую для доступа к данным, от модели предметной области.

В реализацию «Репозитория» обычно включается следующий функционал (**CRUD**):

- создание (Create): добавление записи в хранилище данных;
- чтение (Read): выборка имеющихся записей из хранилища данных;
- обновление (Update): редактирование имеющихся записей;
- удаление (Delete) имеющихся записей.

Хорошей практикой считается создание отдельных репозиториев для каждого бизнес-объекта (POCO) или контекста, например: BooksRepository, UsersRepository, AdminRepository.

Схема паттерна Репозиторий показана на рис. 2.1.

Библиотека ГГТУ им. П.О. Сухого

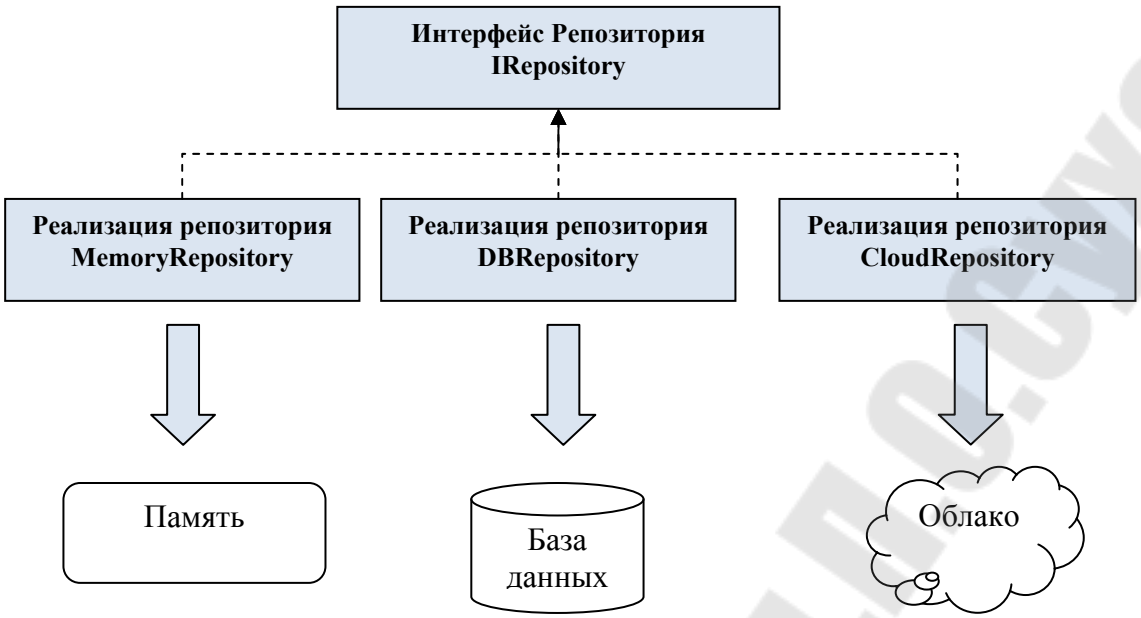

Рис. 2.1. Схема паттерна Репозиторий

Можно создавать обобщенный (**Generic**) **Репозиторий,** но только если приложение работает с данными одинаково. Например,

```
public interface IRepository<T> where T:class
    {
        IEnumerable<T> GetAll();
        T Get(int id);
        IEnumerable<T> Find(Func<T, Boolean> predicate);
        void Create(T item);
        void Update(T item);
        void Delete(int id);
    }
```

Можно использовать обобщенный интерфейс в качестве базового для других. Например,

```
public  interface IDoctorRepository:IRepository<Doctor>
    {

     }
```

Библиотека ГГТУ им. П.О. Сухого

Преимущества паттерна Репозиторий:

– независимость бизнес-логики от способа хранения;

– возможность использовать разные способы хранения: ORM, облачное хранилище, файл и т.д, заменять их и комбинировать;

– работая через интерфейсы, можно создать несколько реализаций репозитория.

2.2. Практическая часть

Задание. Разработать модель заданной предметной области в соответствии с вариантом задания, приведенным в табл. 2.1.

Разработать слой доступа к данным (Data Access Layer) в виде библиотеки классов для создаваемой системы в рамках выбранной предметной области.

Библиотека должна содержать:

– классы сущностей (Entities), моделирующие не менее чем три таблицы базы данных;

– интерфейсы и классы, реализующие паттерн **Репозиторий** для доступа к данным в файле в формате CSV и к базе данных с использованием подключенного режима ADO.Net.

Разработать консольное приложение, использующее описанную выше библиотеку, (слой UI), предоставляющее пользователю удобный интерфейс для работы с системой. Приложение должно работать под управлением меню и выполнять следующие функции:

– вывод информации по запросу пользователя;

– добавление информации;

– удаление указанных пользователем данных;

– поиск информации.

Порядок выполнения. В первую очередь, необходимо подготовить файлы с данными в формате CSV и создать базу данных с необходимыми таблицами.

Для выполнения поставленной задачи нужно создать решение с двумя проектами: консольным приложением и библиотекой классов.

В библиотеке следует создать систему папок для размещения разрабатываемых классов моделей предметной области, интерфейсов и реализаций репозитория для каждой сущности для работы с файлами в формате CSV и с таблицами базы данных. Как правило, создают отдельные папки для моделей данных, отдельно хранят абстракции, также нужна папка для реализации абстракций.

Библиотека ГГТУ им. П.О. Сухого

Пример структуры библиотеки классов для доступа к данным приведен на рис. 2.2.

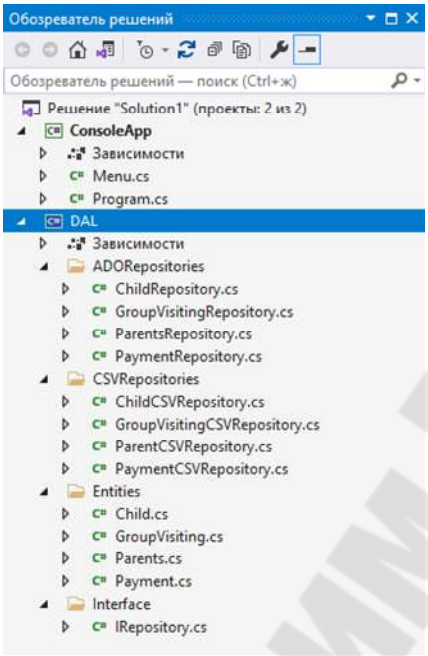

Рис. 2.2. Пример структуры приложения

Затем необходимо создать классы сущностей предметной области. Например,

```
class Dog
{
	public int Id { get; set; }
	public string Name { get; set; }
	public int Age { get; set; }
}
```

В папке для интерфейсов создать необходимые интерфейсы для контрактов репозиториев. Например,

```
interface IRepository<T> where T : class
{
	IEnumerable<T> GetAll (); // получение всех объектов
	T Get(int id); // получение одного объекта по id
	void Create(T item); // создание объекта
	void Update(T item); // обновление объекта
	void Delete(int id); // удаление объекта по id
}
```

Затем необходимо создать реализации репозиториев для каждой модели данных для каждого вида хранилища. Например,

Библиотека ГГТУ им. П.О.Сухого

```
class TextCSVDogRepository : IRepository<Dog>
{
    List<Dog> dogs;
    string pathFile;
    public TextCSVDogRepository(string pathFile)
    {
        this.pathFile = pathFile;
        dogs = new List<Dog>();
    }
    public void Create(Dog item)
    {
       if (!dogs.Any(d => d.Id == item.Id))
       {
           dogs.Add(item);
           Write();
       }
       else
        throw new Exception("This dog already exists!");
    }
     …
}
```

В консольное приложение следует добавить ссылку на библиотеку. Реализовать все функции, указанные в задании, используя объекты классов из разработанной библиотеки.

Таблица 2.1

Вариант	Система для разработки
1	Система записи пациентов на прием в поликлинику
2	Приложение для автоматизации работы фитнес-центра
3	Автоматизированная система учета заявок станции техобслуживания
4	Автоматизированная система заказа такси
5	Система продажи авиабилетов
6	Сайт туристического агентства
7	Интернет-магазин по продаже компьютерных комплектующих
8	Сайт заведения общественного питания

Библиотека ГГТУ им. П.О.Сухого

Окончание табл. 2.1

Вариант	Система для разработки
9	Система продажи электронных железнодорожных билетов
10	Система управления производством и реализацией мебели
11	Система продажи театральных билетов
12	Приложение «Новостной портал»
13	Система записи клиентов салона красоты
14	Система заказа обедов в офис организации
15	Система для автоматизации деятельности фирмы организации мероприятий

3. Разработка модели предметной области с использованием Entity Framework в трехслойном приложении

3.1. Краткие теоретические сведения

Entity Framework Core представляет специальную объектно-ориентированную технологию на базе .NET для работы с данными. Является object-relational mapping (**ORM**) решением от Microsoft.

Объектная модель – это группа классов приложения, связанных между собой и использующихся для хранения, обработки и отображения данных из БД.

ORM – Object Relationl Mapping или «отображение объектов в связанные таблицы».

Чтобы воспользоваться функционалом Entity Framework Core для MS SQL Server, надо добавить соответствующий пакет: **Microsoft.EntityFrameworkCore.SqlServer**.

Если надо создать базу данных, нужно добавить через NuGet пакет **Microsoft.EntityFrameworkCore.Tools.**

Для подключения к существующей БД нужно добавить **Microsoft.EntityFrameworkCore.SqlServer.Design.**

Главными классами Entity Framework являются **DbContext, DbSet и DbContextOptionsBuilder**: устанавливает параметры подключения**.** Контекст базы данных – это специальный класс, производный от системного класса DbContext и предназначенный для установ-

Библиотека ГГТУ им. П.О. Сухого

ления связи с БД и для выполнения запросов к БД. Контекст данных содержит одно или несколько свойств типа **DbSet<T>**, где T представляет имена сущностей модели, т. е. имена классов, соответствующих таблицам БД. Для настройки подключения нужно переопределить метод **OnConfiguring**. Передаваемый в него параметр **DbContextOptionsBuilder** с помощью метода UseSqlServer позволяет настроить строку подключения для соединения с MS SQL Server.

Например,

```
class ApplicationContext:DbContext
{
    public ApplicationContext()
    {
        Database.EnsureCreated();
    }
    public DbSet<DogKind> DogKinds { get; set; }
    public DbSet<Dog> Dogs { get; set; }
    protected override void
    OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
                          "Server=(localdb)\\mssqllocaldb;
                           Database=dogdb;
                           Trusted_Connection=True;");
    }
}
```

Второй способ установки конфигурации подключения к базе данных – передача конфигурации в конструктор базового класса DbContext.

```
public ApplicationContext
     (DbContextOptions<ApplicationContext> options)
                                      :base(options)
{
    Database.EnsureCreated();
}
```

Создается объект класса DbContextOptions следующим образом:

```
var optionsBuilder =
new DbContextOptionsBuilder<ApplicationContext>();
DbContextOptions<ApplicationContext> options =
     optionsBuilder.UseSqlServer(connectionString)
                   .Options;
```


Библиотека ГГТУ им. П.О.Сухого

Все сущности в Entity Framework определяются в виде классов моделей. Есть три подхода к созданию моделей:

- соглашение (conventions);
- Fluent API;
- аннотации данных.

При подходе по соглашению нужно придерживаться следующих правил.

Названия столбцов таблицы должны соответствовать названиям свойств.

Каждая сущность сопоставляется с таблицей, которая называется по имени свойства **DbSet<T>** в контексте данных, представляющего данную сущность. Если в контексте данных подобного свойства не определено, то для названия таблицы используется имя класса сущности.

В качестве ключа используется свойство, которое называется **Id** или **[имя_класса]Id**.

Свойство является необязательным к установке, если оно допускает значение **null**.

Свойство является обязательным, если значение null не является для него корректным. Это свойства типов **int, decimal, bool** и т. д.

По умолчанию провайдер СУБД выбирает для столбцов типы данных *на основе типов данных свойств сущности.*

Fluent API – это набор методов, которые определяют сопоставление между классами и таблицами, между свойствами классов и столбцами таблиц. Чаще всего методы Fluent API применяются при переопределении метода **OnModelCreating** в контексте данных.

Для сопоставления класса с таблицей базы данных используется метод **ToTable()** класса **Entity,** который можно получить через метод класса **DbModelBuilder.** Например,

```
protected override void OnModelCreating
                              (ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Dog>().ToTable("Psiny");
}
```

Если по сущности не нужно создавать таблицу, то ее можно проигнорировать с помощью метода **Ignore():**

```
modelBuilder.Ignore<Cat>();
```

Для переопределения первичного ключа используется метод **HasKey()** для сущности, например,

Библиотека ГГТУ им. П.О. Сухого

modelBuilder.Entity<Dog>().HasKey(u => u.PesId);

Для сопоставления свойства с определенным столбцом, используется метод **HasColumnName().**

```
modelBuilder.Entity<Dog>()
                .Property(d => d.Name)
                .HasColumnName("Nickname");
```

С помощью метода **IsRequired()** можно указать, что значение для этого столбца и свойства требуется обязательно. Например,

```
modelBuilder.Entity<User>()
                .Property(u => u.Login)
                .IsRequired();
```

Если надо указать, что столбец может принимать значения NULL, используется метод **IsOptional().**

Аннотации представляют настройку сопоставления моделей и таблиц с помощью атрибутов. Классы аннотаций располагаются в пространстве имен **System.ComponentModel.DataAnnotations.**

Атрибут **[Key]** используется для установки свойства в качестве первичного ключа.

Атрибут **[Required]** указывает, что данное свойство обязательно для установки, то есть будет иметь определение NOT NULL.

Атрибуты **[MaxLength]** и [**MinLength]** устанавливают максимальное и минимальное количество символов в свойстве типа string.

Атрибут **[NotMapped]** используется, чтобы для свойства не создавался столбец в таблице.

Атрибут **[Table]** используется для сопоставления с таблицей.

Атрибут **[Column]** используется для сопоставления свойства со столбцом.

Атрибут **[ForeignKey]** устанавливает внешний ключ для связи с другой сущностью. Применяется к свойству навигации**.**

Например,

```
[Table("Svora")]
 class Dog
 {
     [Key]
     public int Ident { get; set; }
     [MinLength(3,ErrorMessage ="маловато будет(")]
     public string Name { get; set; }
     [Column("DogWeight")]
     public double Weight { get; set; }
    public int DogKindId { get; set;}
```


Библиотека ГГТУ им. П.О.Сухого

```
    }
```

Настройки для моделей и их свойств можно вынести в объект класса, реализующего **IEntityTypeConfiguration<T>**.

Например,

```
class DogConfiguration : IEntityTypeConfiguration<Dog>
{
    public void Configure(EntityTypeBuilder<Dog> builder)
    {
       builder.ToTable("Svora").HasKey(p => p.Ident);
       builder.Property(p => p.Name)
              .IsRequired()
              .HasMaxLength(7);
    }
}
```

Тогда в классе контекста данных:

```
protected override void
       OnModelCreating(ModelBuilder modelBuilder)
        {
           modelBuilder.ApplyConfiguration(
                              new DogConfiguration());
        }
```

Для внесения изменений в базу данных при изменении моделей и контекста данных нужно использовать миграции.

Для создания миграции в окне **Package Manager Console** вводится следующая команда:

Add-Migration название_миграции.

Например, **Add-Migration AddDogAgeMigration.**

При использовании миграций нужно убрать вызов метода Database.EnsureCreated() из контекста данных.

Если настройки подключения к базе данных передаются через параметр конструктора, нужно создать класс, который реализует интерфейс **IDesignTimeDbContextFactory** и задает конфигурацию контекста. Этот класс явно нигде не вызывается и никак не используется, он вызывается инфраструктурой Entity Framework при создании миграции.

Чтобы выполнить миграцию нужно набрать в консоли **Package Manager** команду **Update-Database.**

Чтобы удалить миграцию нужно набрать в той же консоли команду **Remove-Migration.**

Библиотека ГГТУ им. П.О. Сухого

Если планируется использовать миграции, то правильно будет использовать их сразу при создании базы данных. При этом метод Database.EnsureCreated() не нужен. При выполнении миграции он вызывает ошибку.

Для внесения редактирования данных в таблицах в классе DbSet<T> есть соответствующие методы. Ниже приводятся примеры использования некоторых из них.

Для добавления:

```
db.Dogs.Add(new Dog
                {
                    Name = "Ли",
                    Weight = 2.3,
                    DogKindId=1
                });
```

Автоинкрементное свойство задавать не нужно. Возникнет ошибка во время выполнения.

Удаление:

```
 Dog dog = new Dog
                {
                    Ident=4,
                    Name = "Липа",
                    Weight = 2.3,
                    DogKindId = 1
                };
  db.Remove(dog);
```

Если необходимо удалить сразу несколько объектов, то можно использовать метод **RemoveRange()**.

Для сохранения изменений нужно использовать метод **SaveChanges()** контекста данных.

3.2. Практическая часть

Задание. Разработать слой доступа к данным (Data Access Layer) в виде библиотеки классов для создаваемой системы в рамках выбранной предметной области.

Библиотека должна содержать:

– классы сущностей (Entities), моделирующие не менее чем три таблицы базы данных;

Библиотека ГГТУ им. П.О.Сухого

– интерфейсы и классы, реализующие паттерн **Репозиторий** для доступа к БД с использованием Entity Framework Core, интерфейсы должны быть public, а реализации internal;

Разработать слой бизнес-логики (Buisiness Logic Layer) в виде библиотеки классов, со ссылкой на DAL, содержащей:

– объекты DTO для сущностей слоя DAL;

– классы, реализующие функцию трансляции данных из DTO в сущности и обратно

– классы (сервисы), реализующие операции с данными в рамках выбранной предметной области с использованием классов библиотеки DAL; обязательные операции: получение всех данных, добавление, удаление, поиск данных, удовлетворяющих заданному критерию, валидация данных; для выполнения запросов использовать LINQ;

– классы для реализации внедрения зависимостей.

Разработать приложение WPF (слой UI), предоставляющее пользователю удобный интерфейс для работы с системой. Это приложение должно содержать ссылку на слой BLL и не должно содержать ссылку на DAL.

На рис. 3.1 отображена структура приложения.

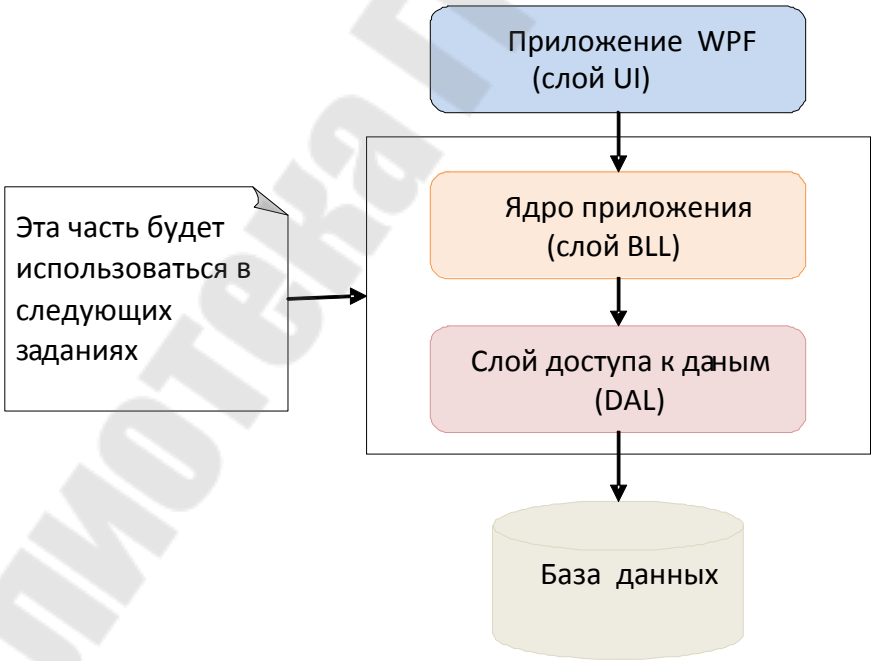

Рис. 3.1. Структура программного комплекса

Библиотека ГГТУ им. П.О. Сухого

Критерии для поиска указаны в табл. 3.1.

Порядок выполнения. Для выполнения поставленной задачи нужно создать решение с тремя проектами: приложением WPF и двумя библиотеками классов (слой DAL и слой BLL).

Добавить в проект DAL пакеты:

- Microsoft.EntityFrameworkCore.SqlServer;
- Microsoft.EntityFrameworkCore.Tools.

Создать в этом слое классы моделей данных, класс контекста данных, интерфейсы репозиториев. Реализовать репозитории для каждой сущности данных с использованием Entity Framework. Например,

```
internal class IngredientRepository
                          : IRepository<Ingredient>
{
    public ApplicationContext Database { get; set; }
    public IngredientRepository(ApplicationContext db)
    {
        Database = db;
    }
    public void Create(Ingredient item)
    {
        Database.Ingredients.Add(item);
        Database.SaveChanges();
    }
   public void Delete(Ingredient item)
    {
        Database.Ingredients.Remove(item);
        Database.SaveChanges();
    }
    public void Update(Ingredient item)
    {
        Database.Ingredients.Update(item);
        Database.SaveChanges();
    }
    public IQueryable<Ingredient> GetAll() =>
    Database.Ingredients.AsQueryable().AsNoTracking();
}
```

Создать статический класс для конфигурирования внедрения зависимостей и в нем метод расширения типа IServiceCollection для до-

Библиотека ГГТУ им. П.О. Сухого

бавления контекста данных и объектов репозиториев в коллекцию сервисов. Например,

```
public static class ConfigurationExtensions
{
    public static void ConfigureDAL(
     this IServiceCollection services,string connection)
    {
       services.AddDbContext<ApplicationContext>(
             options => options.UseSqlServer(connection));
       services.AddScoped<IRepository<Ingredient>,
                                    IngredientRepository>();
       services.AddScoped<IRepository<IngredientCategory>,
IngredientCategoryRepository>();
             …
     }
}
```

Далее следует добавить в проект BLL пакеты:

- AutoMapper;
- AutoMapper.Extensions.Microsoft.DependencyInjection.

Добавить ссылку на библиотеку DAL.

Создать в этом слое классы объектов DTO, интерфейсы сервисов.

Например,

```
public class IngredientCategoryDTO
{
    public int Id { get; set; }
    public string Name { get; set; }
}
public interface IService<T> where T:IDTO
{
    void Add(T item);
    void Update(T item);
    void DeleteById(int id);
    T FindById(int id);
    List<T> GetAll();
}
public interface IPizzaService:IService<PizzaDTO>
{
   List<(IngredientDTO, int)> GetIngredients(int id);
}
```


Библиотека ГГТУ им. П.О.Сухого

Создать классы профилей для отображения моделей данных с помощью AutoMapper. Например,

```
class PizzaProfile:Profile
{
    public PizzaProfile()
    {
        CreateMap<Pizza, PizzaDTO>().ReverseMap();
    }
}
```

Реализовать сервисы, исходя из требуемого функционала.

Создать статический класс для конфигурирования внедрения зависимостей и в нем метод расширения типа IServiceCollection для добавления объекта AutoMapper и объектов сервисов в коллекцию. Например,

```
public static class ConfigurationExtensions
{
    public static void ConfigureBLL(
       this IServiceCollection services,string connection)
    {
        services.ConfigureDAL(connection);
        services.AddAutoMapper(
                typeof(IngredientCategoryProfile),
                typeof(IngredientProfile),
                typeof(OrderItemProfile),
                …
         );
        services
           .AddTransient<IService<IngredientCategoryDTO>,
                                IngredientCategoryService>();
       services
          .AddTransient<IService<IngredientDTO>,
                                       IngredientService>();
       services.AddTransient<IPizzaService,
                                       PizzaService>();
       …
     }
}
```

Добавить в проект WPF ссылку на BLL. Разработать удобный пользовательский интерфейс и реализовать требуемый в задании функционал с использованием сервисов слоя BLL.

Библиотека ГГТУ им. П.О. Сухого

Таблица 3.1

Вариант	Система для разработки	Критерии для поиска
1	Система записи пациентов на прием в поликлинику	– по фамилии врача; – по фамилии пациента
2	Приложение фитнес-центра	– по виду тренировок; – по фамилии тренера
3	Автоматизированная система учета заявок станции техобслуживания	– по фамилии владельца авто; – по дате приема заявки
4	Автоматизированная система заказа такси	– по времени заказа; – по адресу клиента
5	Система продажи авиабилетов	– по номеру рейса; – по цене
6	Сайт туристического агентства	– по срокам поездки; – по региону путешествия
7	Интернет–магазин по продаже компьютерных комплектующих	– по наименованию комплектующих; – по цене
8	Сайт заведения общественного питания	– по названию блюда; – по категории блюда
9	Система продажи электронных железнодорожных билетов	– по пункту назначения; – по дате отправления
10	Система управления производством и реализацией мебели	– по наименованию изделия; – по дате выпуска продукции
11	Система продажи театральных билетов	– по дате спектакля; – по цене билета
12	Приложение «Новостной портал»	– по теме новостей; – по дате публикации
13	Система записи клиентов салона красоты	– по виду услуги; – по фамилии мастера
14	Система заказа обедов в офис организации	– по дате заказа; – по названию организации

Библиотека ГГТУ им. П.О.Сухого

Окончание табл. 3.1

Вариант	Система для разработки	Критерии для поиска
15	Система для автоматизации деятельности фирмы организации мероприятий	– по виду мероприятия; – по дате проведения

4. Разработка web-приложения с использованием ASP.NET Core MVC

4.1. Краткие теоретические сведения

ASP.NET Core MVC – платформа для разработки веб-приложений, реализующих паттерн **«модель-представление-контроллер».**

Model-View-Controller (MVC) – шаблон, позволяющий разделять данные приложения, пользовательский интерфейс и управляющую логику на три отдельных компонента: модель, представление и контроллер – таким образом, что модификация каждого компонента может осуществляться независимо. Основная идея этого паттерна в том, что и контроллер и представление зависят от модели, но модель никак не зависит от этих двух компонент.

В инфраструктуре ASP.NET Core MVC контроллеры – это классы C#, обычно производные от класса **Controller.** Каждый открытый метод в классе контроллера является методом действия, который ассоциирован с каким-то URL. В приложении ASP.NET Core MVC представление является файлом, содержащим HTML-элементы и код C#, который обрабатывается для генерации ответа. Модели представляют собой простые классы и располагаются в проекте в каталоге *Models*. Модели описывают логику данных.

В приложении ASP.NET MVC Core модели можно разделить по степени применения на несколько групп:

– модели, объекты которых хранятся в специальных хранилищах данных (например, в базах данных, файлах xml и т.д.);

– модели, которые используются для передачи данных в представление или для получения данных из представления; такие модели называются **моделями представления;**

– вспомогательные модели для промежуточных вычислений.

Библиотека ГГТУ им. П.О. Сухого

За сопоставление запросов с конкретными адресами внутри приложения в ASP.NET Core отвечает система маршрутизации. Маршрутизация в MVC это процесс соответствия входящего запроса и обработчика запроса. Обработчиками запросов в MVC обычно выступают *действия контроллера*. Обработка запроса в ASP.NET Core устроена по принципу конвейера. Данные запроса по очереди получают компоненты в конвейере, которые называются **middleware**. Для направления запросов на соответствующие действия контроллера в MVC существует таблица маршрутизации. Добавление маршрута в таблицу маршрутизации называется ***регистрацией маршрута***. Есть два API, которые можно использовать для регистрации маршрутов:

- Fluent API;
- атрибуты маршрутизации.

Маршрутизация к действиям должна быть добавлена в конвейер обработки запросов. При создании проекта по шаблону ASP.Net Core MVC это происходит через определение и использование конечных точек с помощью компонентов **EndpointRoutingMiddleware** и **EndpointMiddleware.** Метод UseEndpoints() встраивает в конвейер обработки запроса компонент EndpointMiddleware, который определяет набор конечных точек, которые будут сопоставляться с определенными маршрутами и будут обрабатывать соответствующие маршрутам входящие запросы. Например,

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
      name: "default",
      pattern: "{controller=Home}/{action=Index}/{id?}");
 });
```

Маршруты, определенные с помощью атрибутов, имеют приоритет по сравнению с маршрутами, определенными в классе Startup (или Program в .Net 6). Для определения маршрута необходимо использовать атрибут **[Route].** В качестве параметра атрибут Route принимает шаблон URL, с которым будет сопоставляться запрошенный адрес. Этот атрибут применяется к действиям контроллера. Например,

```
[Route("Trepatnja")]
    [HttpGet]
    public ActionResult Comments()
    {
      var comments = CommentsDb.Comments;
```


Библиотека ГГТУ им. П.О. Сухого

```
        return View(comments);
    }
```

Чтобы создать свой класс контроллера, достаточно создать класс, наследующий от Controller и имеющий в имени суффикс Controller.

Методы действий (action methods) представляют такие методы контроллера, которые обрабатывают запросы по определенному URL. Так как запросы бывают разных типов, ASP.NET MVC позволяет определить тип обрабатываемого запроса для действия, применив к нему соответствующий атрибут: [HttpGet], [HttpPost], [HttpDelete] или [HttpPut].

Методы действий всегда имеют модификатор **public**. Закрытых методов действий не бывает. Но контроллер может также включать и обычные методы, которые могут использоваться для вспомогательных целей.

Для контроллеров предназначена папка **Controllers**. По умолчанию при создании проекта в нее добавляется контроллер **HomeController.**

Если нужно, чтобы какой-то public метод контроллера не рассматривался как действие, то можно применить к нему атрибут [**NonAction]**.

Метод контроллера формирует некоторый ответ в виде результата действия, как правило, это объект класса, реализующего интерфейс **IActionResult.**

Существует ряд стандартных результатов действий**.**

ContentResult пишет указанное содержимое напрямую в ответ в виде строки.

EmptyResult отправляет пустой ответ в виде статусного кода 200.

FileResult предназначен для отправки файлов.

ViewResult производит рендеринг представления и отправляет результаты рендеринга в виде html-страницы клиенту. Чтобы возвратить объект ViewResult используется метод **View()**.

По умолчанию контроллер производит поиск представления в проекте по следующему пути:

/Views/Имя_контроллера/Имя_представления.cshtml .

В качестве представления будет использоваться то представление, имя которого совпадает с именем действия. Можно также задать имя представления явным образом:

Библиотека ГГТУ им. П.О.Сухого

return View("Index");

Можно полностью переопределить путь, по которому система будет искать представление:

return View("~/Views/Some/Index.cshtml");

RedirectResult перенаправляет пользователя по другому адресу URL.

Метод **RedirectToAction** позволяет перейти к определенному действию определенного контроллера, а также позволяет задать передаваемые параметры:

return RedirectToAction("Index");

return RedirectToAction("Square", "Home", new { a=10,h=12});

HttpNotFoundResult возвращает клиенту ответ в виде статусного кода HTTP 404, указывая, что запрошенный ресурс не найден.

Представление не является html-страницей. При компиляции приложения на основе требуемого представления генерируется класс на языке C#, наследующий от класса **RazorPage<T>**, где T – это класс модели, которая будет использоваться, или ключевое слово **dynamic**.

Все добавляемые представления, группируются по контроллерам в соответствующие папки в каталоге **Views**. Для рендеринга представления в выходной поток, используется метод View(). Контроллер готовит данные и выбирает представление, а затем объект ViewResult обращается к *движку представления* **Razor** для рендеринга представления в выходной результат.

Способы передачи данных из контроллера в представление:

- ViewData;
- ViewBag ;
- модель представления.

Строго типизированные представления позволяют передавать данные не через объект ViewBag и т.п., а напрямую в представление *через параметр* метода View.

Чтобы связать представление с передаваемым параметром, надо добавить в представление директиву **@model** с указанием **типа передаваемых данных.** Объект **Model** представляет тип модели и будет содержать переданные из контроллера данные.

```
@using MvcSimple.Models;
@model IEnumerable<Product>
    <div>
        <table border="1">
            <tr>
```


Библиотека ГГТУ им. П.О.Сухого

```
                <td><p>Товар</p></td>
                <td><p>Категория</p></td>
                <td><p>Цена</p></td>
                <td><p>Выбор</p></td>
            </tr>
            @foreach(var b in Model)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.ProductCategory</p></td>
                    <td><p>@b.Price</p></td>
                    @*адрес, по которому будет размещаться
форма оформления покупки*@
                    <td><p><a
href="/Home/Buy?id=@b.id&item=@b.Name">Купить</a></p></td>
                </tr>
            }
        </table>
    </div>
```

В классе контроллера:

```
public ActionResult Index()
        {
            // получаем из бд данные
            ViewBag.Title = "Магазин";
            var products = db.Products;

            // возвращаем представление

            return View(products);
        }
```

Для генерирования html-кода можно использовать **HTML-хелперы.** Для создания хелпера в проекте нужно создать папку под названием App_Code. Это специальное зарезервированное имя для папки, которая содержит html-хелперы, поэтому нужно использовать данное название. Хелперы представляют собой методы расширения, поэтому в папку App_Code для создания хелперов нужно добавить статический класс. Ниже приводится пример хелпера для генерирования маркированного списка.

```
public static class ListHelper
{
public static HtmlString CreateList(this IHtmlHelper html,
```


Библиотека ГГТУ им. П.О. Сухого

```
string[] items)
  {
      string result = "<ul>";
      foreach (string item in items)
      {
         result = $"{result}<li>{item}</li>";
       }
      result = $"{result}</ul>";
      return new HtmlString(result);
  }
}
```

Пример использования в представлении:

@Html.CreateList(students)

Существует ряд встроенных html-хелперов. Встроенный хелпер **BeginForm** позволяет создать форму. Принимает в качестве параметров имя метода действия и имя контроллера, а также тип запроса. Например,

```
using(Html.BeginForm("Buy","Home",FormMethod.Post))
{
    ...
}
```

Хелпер Html.**TextBox** генерирует тег **input** со значением атрибута type равным text. Перегружен семь раз. Например,

```
@Html.TextBox("Address")
```

Html.Hidden генерирует тег input type="hidden" **.** Например**,**

```
@Html.Hidden("ProductId",(int) ViewBag.ProductId)
```

Один из параметров хелперов позволяет задать атрибуты создаваемого элемента формы, в том числе и классы css.

```
@Html.TextBox("Text", "Наберите текст комментария",
   new { style = "width:350px;height:100px",
         maxlength = 250, minlengh = 5, required = true })
```

Этот хелпер генерирует тег

```
<input    id =" Text "  maxlength   =" 250 "  minlengh   =" 5"
name =" Text "  required    =" True "
style   =" width:350px;height:100px          "
type  =" text   "  value   =" #x41D;#x430;#x431                " />
```

В библиотеке определены также html-хелперы **Html.Password, Html.RadioButton, Html.CheckBox, Html.Label, Html.ListBox.**

Строго типизированные хелперы принимают в качестве параметра лямбда-выражение, в котором указывается то свойство модели, к которому должен быть привязан данный хелпер. Могут использо-

Библиотека ГГТУ им. П.О.Сухого

ваться только в строго типизированных представлениях. Тип модели, которая передается в хелпер, должен быть тем же самым, что указан в директиве **@model**. Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер, только в имени добавлен суффикс **For.** Например, `@Html.TextBoxFor(m=>m.Client).`

Tag-хелперы (тег-хелперы**)** – это специальные элементы в представлении, выглядят как обычные html-элементы или атрибуты, но при работе приложения они обрабатываются движком Razor на стороне сервера и преобразуются в стандартные html-элементы.

Например,

```
<a asp-controller="Home" asp-action="Comments"
                asp-route-header="@Model.Title"
                class="btn btn-primary btn-large">
                 Перейти к комментариям
</a>
```

Для создания тег-хелпера нужно создать класс, наследующий от **TagHelper**. В классе TagHelper определен метод **Process**(), который переопределяется производными классами для реализации поведения, трансформирующего элементы. Имя тег-хелпера образуется из имени трансформируемого элемента и суффикса TagHelper. Метод Process принимает два параметра: объект **TagHelperContext**, представляющий контекст тега (его содержимое, атрибуты), и объект **TagHelperOutput**, отвечающий за генерацию выходного элемента html на основе тега. Ниже приведен пример тег-хелпера для генерирования маркированного списка.

```
public class ListTagHelper:TagHelper
{
   public List<String> Info { get; set; }
   public override void Process(TagHelperContext context,
TagHelperOutput output)
   {
        output.TagName = "ul";
        string temp = "";
        foreach (string item in Info)
        {
           temp = $"{temp}<li>{item}</li>";
        }
        output.Content.SetHtmlContent(temp);
        output.TagMode = TagMode.StartTagAndEndTag;
    }
```


Библиотека ГГТУ им. П.О. Сухого

```
}
```

Чтобы задействовать класс хелпера в представлении, надо подключить его функциональность в представление. Например, использовать директиву addTagHelper в файле *Views/_ViewImports.cshtml*:

```
@addTagHelper "*, WebApp_1".
```

Пример применения в представлении:

```
<list info="@Model.Readers"></list>
```

4.2. Практическая часть

Задание. Разработать веб-приложение на базе *ASP.NET Core MVC*, предоставляющее пользовательский интерфейс для выбранной предметной области и позволяющее пользователю выполнять различные операции с данными. Для выполнения операций нужно использовать слой BLL (разработанный при выполнении задания из раздела 3).

Приложение должно обеспечивать набор базовых операций, таких как редактирование справочников, запрос данных, изменение, удаление, добавление новых. В зависимости от выбранной предметной области, набор операций/действий и данных может отличаться.

Это приложение должно содержать ссылку на слой BLL и **не должно содержать ссылку на слой DAL**.

В приложении должно быть минимум 3 представления. Все страницы должны быть оформлены в едином стиле, основанном на использовании одной или нескольких мастер-страниц, и иметь систему навигации (строка меню, гиперссылки, кнопки).

Для создания внешнего вида веб-страниц можно использовать общеизвестные и популярные фреймворки, например, Bootstrap.

На рис. 4.1 отображена структура программного комплекса после добавления проекта ASP.Net.

Порядок выполнения. В разработанное в задании раздела 3 решение необходимо добавить проект ASP.Net Core MVC.

Добавить в этот проект ссылку на слой BLL.

Создать классы моделей для представлений. Например,

```
public class CatalogPizzaIngredientViewModel
{
    public string Name { get; set; }
    public string Description { get; set; }
    public int Weight { get; set; }
}
```


Библиотека ГГТУ им. П.О.Сухого

Создать необходимые классы контроллеров. Добавить в контроллер методы действий в соответствии с требуемым в задании функционалом. Разработать интерфейс пользователя в виде файлов представлений.

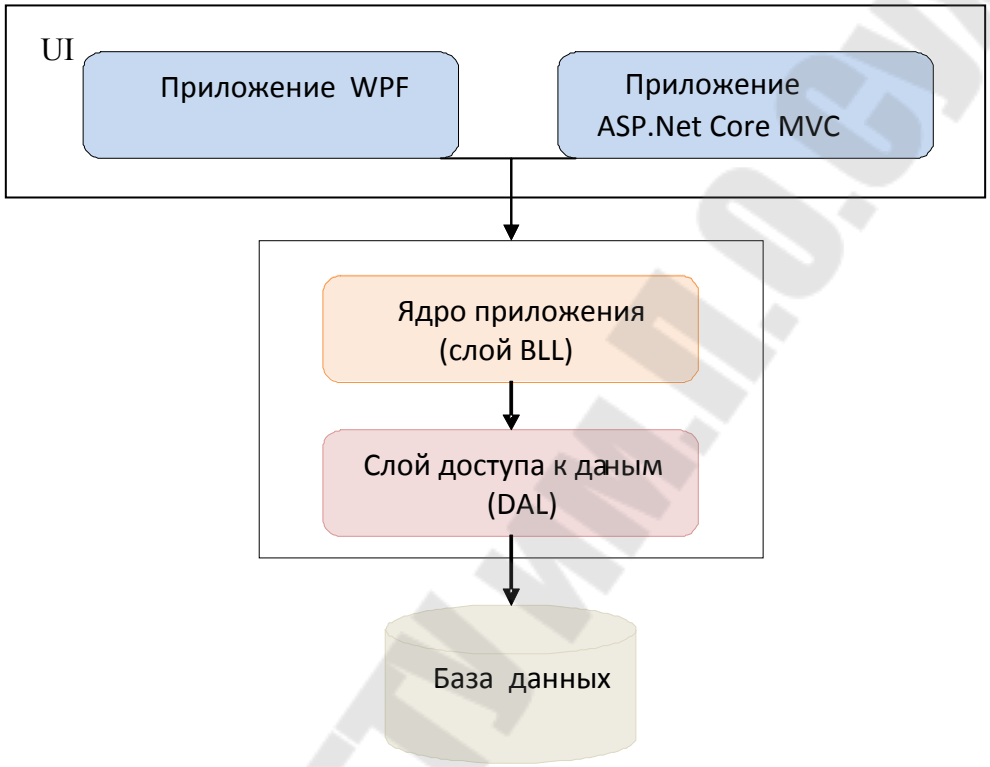

Рис. 4.1. Структура программного комплекса

5. Авторизация и аутентификация пользователей в web -приложении

5.1. Краткие теоретические сведения

В ASP.NET Core используется система авторизации и аутентификации под названием **ASP.NET Identity**.

Основные классы: IdentityDbContext**,** IdentityUser, UserManager, UserStore<T>, IdentityRole, RoleManager<T>.

IdentityDbContext – контекст данных, производный от **DbContext**, который уже содержит свойства, необходимые для управления пользователями и ролями: свойства **Users** и **Roles**. В приложении лучше создавать класс, производный от IdentityDbContext.

Кроме Users и Roles в контексте есть свойства:

– RoleClaims: набор объектов IdentityRoleClaim;

Библиотека ГГТУ им. П.О. Сухого

– UserLogins: набор объектов IdentityUserLogin;
– UserClaims: набор объектов IdentityUserClaim;
– UserRoles: набор объектов IdentityUserRole, соответствует таблице, которая сопоставляет пользователей и их роли;
– UserTokens: набор объектов IdentityUserToken, соответствует таблице токенов пользователей.

Класс **IdentityUser** реализует интерфейс **IUser** и определяет следующие свойства:

– Claims: коллекция специальных атрибутов, которыми обладает пользователь и которые хранят о пользователе определенную информацию;
– Email: email пользователя;
– Id: уникальный идентификатор пользователя;
– Logins: коллекция логинов пользователя;
– PasswordHash: возвращает хэш пароля;
– Roles: коллекция ролей пользователя;
– PhoneNumber: номер телефона;
– UserName: возвращает ник пользователя;
– AccessFailedCount: число попыток неудачного входа в систему;
– EmailConfirmed: true, если email был подтвержден;
– PhoneNumberConfirmed: true, если телефонный номер был подтвержден;
– TwoFactorEnabled: если true, то для данного пользователя включена двухфакторная авторизация.

Обычно для управления пользователями определяют класс, производный от IdentityUser:

```
public class ApplicationUser : IdentityUser
{
    …
}
```

Класс **UserManager** осуществляет непосредственное управление пользователями. Основные методы:

– ChangePasswordAsync(user,old,new): изменяет пароль пользователя;
– CreateAsync(user): создает нового пользователя;
– DeleteAsync(user): удаляет пользователя;
– FindByIdAsync(id): ищет пользователя по идентификатору;
– FindByEmailAsync(email): ищет пользователя по email;

Библиотека ГГТУ им.П.О.Сухого

- FindByNameAsync(name): ищет пользователя по имени;
- UpdateAsync(user): обновляет пользователя;
- Users: возвращает всех пользователей;
- AddToRoleAsync(user, role): добавляет для пользователя user роль role;
- GetRolesAsync (user): возвращает список ролей, к которым принадлежит пользователь user;
- IsInRoleAsync(user, name): возвращает true, если пользователь user принадлежит роли name;
- RemoveFromRoleAsync(user, name): удаляет роль с именем name у пользователя user.

Класс **UserStore<T>** представляет хранилище пользователей. Реализует интерфейс **IUserStore<T>**. Чтобы создать объект UserStore, необходимо использовать контекст данных IdentityDbContext.

Класс **IdentityRole** представляет роль пользователя и содержит следующие свойства:

- Id: уникальный идентификатор роли;
- Name: название роли;
- Users: коллекция объектов IdentityUserRole, который связывают пользователя и роль.

Класс **RoleManager<T>**, где T - реализация интерфейса **IRole**, управляет ролями с помощью следующих методов:

- CreateAsync(role): создает новую роль;
- DeleteAsync(role): удаляет роль;
- FindByIdAsync(id): возвращает роль по идентификатору;
- FindByNameAsync(name): возвращает роль по названию;
- RoleExistsAsync(name): true, если роль с данным именем существует;
- UpdateAsync(role): обновляет роль;
- Roles: возвращает все роли.

Для взаимодействия с базой данных через ASP.NET Core Identity нужно добавить в проект через Nuget соответствующие пакеты. Например, для взаимодействия с MS SQL Server нужно добавить пакеты:

- **Microsoft.AspNetCore.Identity.EntityFrameworkCore**;
- **Microsoft.EntityFrameworkCore.SqlServer**.

Для работы с Identity и базой данных нужно добавить все необходимые сервисы в классе Startup (или Program). Например,

Библиотека ГГТУ им. П.О. Сухого

```
string connection = Configura-
tion.GetConnectionString("DefaultConnection");
            services.AddDbContext<ShopContext>(options =>
options.UseSqlServer(connection));
            ser-
vices.AddDbContext<AccountDbContext>(options => op-
tions.UseSqlServer(connection));
            services.AddIdentity<User, IdentityRole>()

.AddEntityFrameworkStores<AccountDbContext>();
```

В методе Configure() нужно установить компонент middeware, вызвав метод **UseAuthentication()**.

Пример действия контроллера в ответ на запрос входа в систему:

```
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Login(LoginModel model)
{
   if (ModelState.IsValid)
   {
      var result =
    signInManager.PasswordSignInAsync(model.UserName,
         model.Password, model.RememberMe, false).Result;
      if (result.Succeeded)
      {
         return RedirectToAction("Index", "Home");
      }
      else
      {
          ModelState.AddModelError("",
             "Неправильный логин и (или) пароль");
      }
   }
   return View(model);
}
```

Пример действия контроллера для выхода из системы:

```
public async Task<ActionResult> Logout()
{
   await signInManager.SignOutAsync();
   return RedirectToAction("Index", "Home");
}
```


Библиотека ГГТУ им.П.О.Сухого

5.2. Практическая часть

Задание. Дополнить разработанное в предыдущих заданиях веб-приложение функциями аутентификации и авторизации.

Web-приложение должно удовлетворять следующим требованиям:

– любой пользователь должен пройти аутентификацию;

– приложение должно поддерживать реализацию не менее трех ролевых политик для доступа к классам и (или) методам контроллеров; обязательные роли по вариантам приводятся в таблице 5.1;

– приложение должно управлять учетными записями пользователей: просматривать, создавать, удалять и редактировать данные учетных записей;

– использовать не менее одного разработанного html- хэлпера и не менее одного Tag-хэлпера;

– вводимые и редактируемые данные должны проходить валидацию на стороне сервера и клиента;

– обязательно реализовывать ajax-запросы для отображения результатов поиска.

На рис.5.1 отображена структура программного комплекса после добавления модуля аутентификации и авторизации.

Таблица 5.1

Вариант	**Система для разработки**	**Роли пользователя**
1	Система записи пациентов на прием в поликлинику	– пациент; – врач; – администратор
2	Приложение фитнес–центра	– клиент; – тренер; – администратор
3	Автоматизированная система учета заявок станции техобслуживания	– клиент; – мастер; – администратор
4	Автоматизированная система заказа такси	– пассажир; – водитель; – диспетчер

Библиотека ГГТУ им. П.О. Сухого

Окончание таб. 5.1

Вариант	Система для разработки	Роли пользователя
5	Система продажи авиабилетов	– пассажир; – менеджер авиакомпании; – администратор
6	Сайт туристического агентства	–клиент; – менеджер; – администратор
7	Интернет-магазин по продаже компьютерных комплектующих	– покупатель; – курьер; – администратор
8	Сайт заведения общественного питания	– клиент; – ответственный за работу с клиентами; – администратор
9	Система продажи электронных железнодорожных билетов	– пассажир; – менеджер; – начальник ж/д вокзала
10	Система управления производством и реализацией мебели	– клиент; – бригадир; – менеджер по продажам
11	Система продажи театральных билетов	– зритель; – курьер; – менеджер
12	Приложение «Новостной портал»	– пользователь; – журналист; – модератор
13	Система записи клиентов салона красоты	– клиент; – специалист; – администратор
14	Система заказа обедов в офис организации	– клиент; – менеджер – курьер
15	Система для автоматизации деятельности фирмы организации мероприятий	– клиент; – работник фирмы; – администратор

Библиотека ГГТУ им. П.О.Сухого

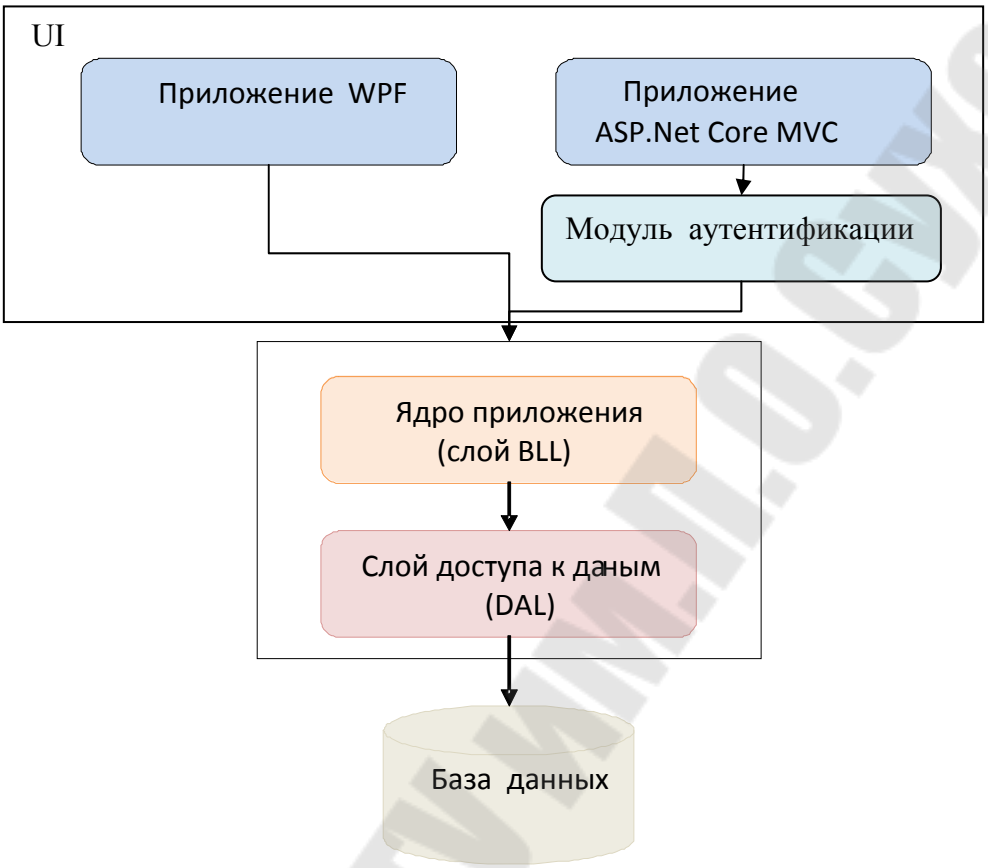

Рис. 5.1. Структура программного комплекса

6. Разработка web-службы с использованием ASP.NET Core

6.1. Краткие теоретические сведения

Web API – это веб-служба, которая может взаимодействовать с различными приложениями: веб-приложениями ASP.NET, мобильными или десктопными.

API (Application programming interface) – это контракт, который предоставляет программа.

API для веб-сервера, веб-браузера или любой другой веб-ориентированной технологии называется Web API. Данная технология позволяет посылать данные в качестве ответа на запросы различного рода клиентов: из сети, от мобильных приложений, от веб-приложений или других Web API.

Структура приложения, созданного по шаблону Web API ASP.Net Core показана на рис. 6.1.

Библиотека ГГТУ им. П.О. Сухого

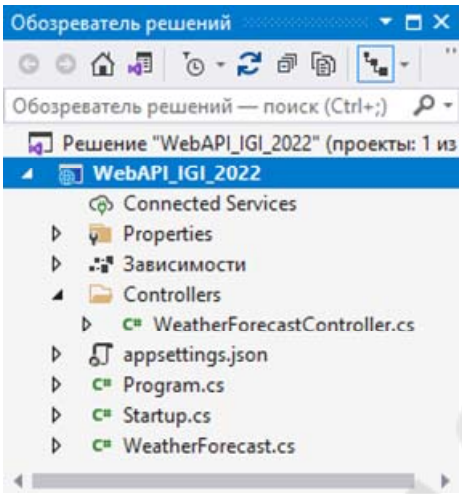

Рис. 6.1. Структура проекта Web API
с примером контроллера

Контроллеры Web API применяют стиль **REST** (**Representation State Transfer** или «передача состояния представления»).

Характеристики контроллера API:

- методы действий возвращают объекты моделей, а не объекты типа ActionResult;
- методы действий выбираются на основе HTTP-метода, используемого в запросе.

Объекты моделей, возвращаемые методом действия контроллера API, кодируются в формате JSON и отправляются клиенту. Контроллеры API предназначены для доставки веб-служб данных, поэтому они не поддерживают представления, компоненты или любые другие средства, которые применялись для генерации HTML-разметки.

Для тестирования контроллера Web API можно применять специальные инструменты, которые устанавливаются в виде отдельных приложений, либо в виде расширений для браузеров, например, **Fiddler** или **Postman**. Visual Studio предоставляет еще один способ – ввод команд в окне **Package Manager Console**.

При создании проекта по шаблону Web API ASP.Net Core в приложение сразу встраивается поддержка Swagger.

Swagger (Open API) – это не зависящая от языка спецификация для описания REST API. Пользовательский интерфейс Swagger обеспечивает пользовательский веб-интерфейс, предоставляющий сведения о службе с использованием созданной спецификации Open API. Этот интерфейс позволяет протестировать каждый метод службы.

Библиотека ГГТУ им. П.О.Сухого

Контроллер API является наследником класса **ControllerBase**, который никак не связан с базовым классом контроллеров MVC - Controller.

Когда в приложение поступает запрос, соответствующий маршруту Web API, действие определяется на основе метода HTTP, используемого для выдачи запроса.

Соглашение по именованию методов действий контроллера API предусматривает снабжение имени метода действия **префиксом в виде поддерживаемого им HTTP-метода** и включение ссылки на тип модели, с которым метод действия работает.

Но не обязательно жестко придерживаться соглашения по именованию, можно использовать любые другие имена без префиксов. Но тогда надо будет явно указать метод HTTP в виде атрибута. Например, [HttpPut]. В ASP.NET Core маршрутизация реализуется без учета регистра символов. Атрибут [Rote] задает общий маршрут к контроллеру. Например,

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
 …
}
```

Локальные маршруты к действиям можно задать с помощью параметров атрибутов [HttpPut], [HttpGet] и т.п. Например,

```
// GET api/values/all
[HttpGet("all")]
public IEnumerable<ToDoTask> GetTasks()
{
    return db.Tasks;
}
//GET api/values/task/5
[HttpGet("task/{id}")]
public ToDoTask GetTask(int id)
 {
    return db.Tasks.Find(id);
 }
```

При запуске приложения из среды Visual Studio в браузере открывается страница, вид которой показан на рис. 6.2.

Библиотека ГГТУ им. П.О.Сухого

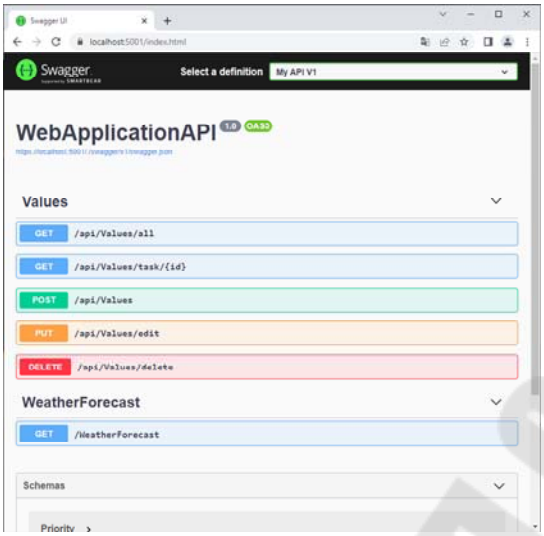

Рис. 6.2. Интерфейс Swagger

6.2. Практическая часть

Задание. Разработать отдельный компонент для приложения, разрабатываемого на протяжении предыдущих заданий, который позволит построить открытый API для возможности доступа к данным и приложению из других источников.

Для выполнения задания нужно создать новое отдельное приложение в виде ASP.NET Core API, которое может быть запущено параллельно web-сайту и представлять набор конечных точек, которые могут быть доступны с использованием протокола HTTP путем вызова разных операций.

Все операции должны соответствовать операциям, которые уже есть в разработанном web-приложении, а также операции, указанные в таблице 6.1.

Добавить в web-приложение, разработанное в задании 3, страницу с документацией API.

На рисунке 6.3 отображена структура программного комплекса после добавления проекта ASP.Net Core Web API.

Библиотека ГГТУ им. П.О.Сухого

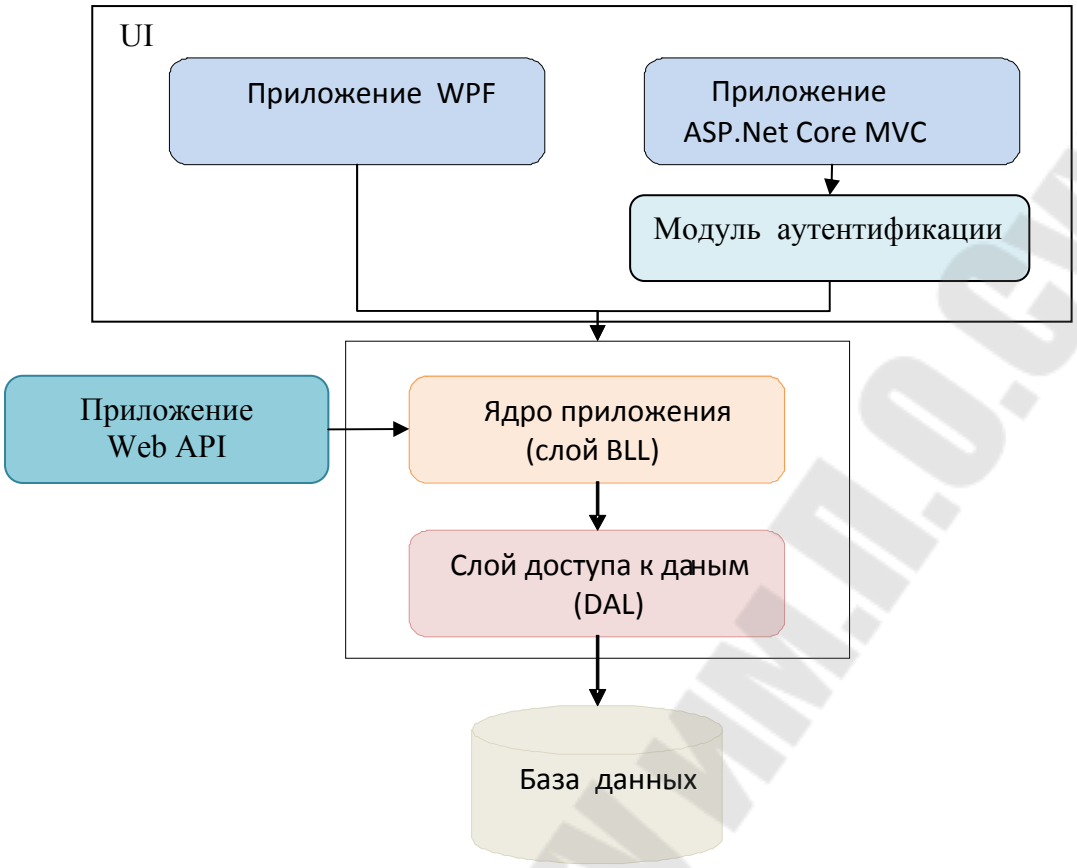

Рис. 6.3. Структура программного комплекса

Таблица 6.1

Вариант	Система для разработки	Операции службы
1	Система записи пациентов на прием в поликлинику	Определение количества пациентов, записанных к указанному врачу на заданную дату
2	Приложение фитнес-центра	Поиск клиентов, записанных более чем к одному тренеру
3	Автоматизированная система учета заявок станции техобслуживания	Определения количества заявок к указанному мастеру за определенный период времени
4	Автоматизированная система заказа такси	Определение среднего количества клиентов, заказывавших такси в период до 8-00
5	Система продажи авиабилетов	Поиск самого дешевого билета на указанное направление

Библиотека ГГТУ им. П.О. Сухого

Окончание табл. 6.1

Вариант	Система для разработки	Операции службы
6	Сайт туристического агентства	Поиск самого дешевого тура на указанный период времени
7	Интернет-магазин по продаже компьютерных комплектующих	Определение суммарной стоимости комплектующих заданного вида, имеющихся в наличии на складе
8	Сайт заведения общественного питания	Поиск самого дорогого напитка
9	Система продажи электронных железнодорожных билетов	Поиск билетов указанного вида до заданного пункта назначения
10	Система управления производством и реализацией мебели	Определение суммарной прибыли от реализации мебели за указанный период
11	Система продажи театральных билетов	Определение количества проданных билетов на указанный спектакль
12	Приложение «Новостной портал»	Поиск статей на заданную тему за указанный период времени
13	Система записи клиентов салона красоты	Определение количества клиентов указанного мастера за указанный период
14	Система заказа обедов в офис организации	Определение суммарной стоимости заказов для заданной организации
15	Система для автоматизации деятельности фирмы организации мероприятий	Поиск количества мероприятий указанного вида за определенный период

Порядок выполнения. Добавить в решение, разработанное в заданиях предыдущих разделов проект Web API. Затем следует добавить ссылку на библиотеку BLL.

Библиотека ГГТУ им. П.О. Сухого

В файл конфигурации нужно добавить строку подключения. Например, если это файл в формате json, строка может выглядеть так:

```
"ConnectionStrings": {
    "DefaultConnection": "Data Source=DESKTOP-
S6MC2AM;Initial Catalog=Pizzeria;Integrated Security=True"
}
```

Сконфигурировать внедрение зависимостей, вызвав метод расширения со слоя BLL в классе Startup. Например,

```
var connection =
   Configuration.GetConnectionString("DefaultConnection");
services.ConfigureBLL(connection);
```

Создать в папке Controllers контроллер. Реализовать в контроллере CRUD-модель и требуемые в задании операции. Например,

```
[ApiController]
[Route("[controller]")]
public class PizzaController : ControllerBase
{
    private readonly IService<PizzaDTO> _pizzaService;
    public PizzaController(
                        IService<PizzaDTO> pizzaService)
   {
       _pizzaService = pizzaService;
   }

  [HttpGet]
   public IEnumerable<PizzaDTO> GetAll()
   {
       return _pizzaService.GetAll();
   }
   [HttpPost]
   public void Add(PizzaDTO pizza)
   {
       _pizzaService.Add(pizza);
   }
   …

}
```

Протестировать действия контроллера с помощью Swagger.

Добавить в web-приложение страницу для отображения документации по предоставляемым web-сервисам.

Библиотека ГГТУ им. П.О. Сухого

Романькова Татьяна Леонидовна

ИЗБРАННЫЕ ГЛАВЫ ИНФОРМАТИКИ

Практикум
по выполнению лабораторных работ
для студентов специальности 1-40 04 01 «Информатика и технологии программирования» дневной формы обучения

Подписано к размещению в электронную библиотеку
ГГТУ им. П. О. Сухого в качестве электронного
учебно-методического документа 01.02.24.

Рег. № 66Е.
http://www.gstu.by

Библиотека ГГТУ им. П.О.Сухого