**ОГЛАВЛЕНИЕ**

1. ВВЕДЕНИЕ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2. КРАТКОЕ ОПИСАНИЕ ШАБЛОНА ПРОЕКТА . . . . . . . . . . . . . . . . . . 4
3. ВЫБОР ТИПА БД И ОПРЕДЕЛЕНИЕ ОСНОВНЫХ СУЩНОСТЕЙ. ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ. МИГРАЦИИ . . . . . . . . . . . . . . . . .7
4. СОЗДАНИЕ ПЕРВОГО СЕРВИСА. «СЛУЧАЙНЫЕ ДАННЫЕ» . . . .11
5. СЕРВИС ПОИСКА ДОКУМЕНТОВ . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
6. СЕРВИС ЧАСТЫХ ПОИСКОВЫХ ЗАПРОСОВ . . . . . . . . . . . . . . . . . 19
7. СЕРВИС НЕДАВНО ОПУБЛИКОВАННЫХ ДОКУМЕНТОВ . . . . . .25
8. РЕАЛИЗАЦИЯ МИНИМАЛЬНОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТФЕЙСА. ВИЗУАЛЬНЫЙ КОМПОНЕНТ «ЛЕНТА ПУБЛИКАЦИЙ» . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .26
9. ОПРЕДЕЛЕНИЕ POST-ЗАПРОСА ДЛЯ ПОИСКА ДОКУМЕНТОВ .29
10. АВТОРИЗАЦИЯ. СЕРВИС ПАРОЛЕЙ . . . . . . . . . . . . . . . . . . . . . . . . . 30
11. АВТОРИЗАЦИЯ. МЕНЕДЖЕР СЕССИЙ . . . . . . . . . . . . . . . . . . . . . . . 31
12. АВТОРИЗАЦИЯ. ЗАЩИТА ОТ ПОВТОРНОГО ВОСПРОИЗВЕДЕНИЯ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .32
13. АВТОРИЗАЦИЯ. СОЗДАНИЕ ИНТЕРФЕЙСА . . . . . . . . . . . . . . . . . . 35
14. СЕРВИС ЗАЩИТЫ ОТ СПАМА . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
15. ФУНКЦИОНАЛ ПУБЛИКАЦИИ ДОКУМЕНТОВ . . . . . . . . . . . . . . . 39
16. ТЕСТИРОВАНИЕ. ПРИМЕНЕНИЕ MOQ . . . . . . . . . . . . . . . . . . . . . . .40
17. ЗАВЕРШЕНИЕ РАЗРАБОТКИ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .42
18. ВЫВОДЫ. ЗАКЛЮЧЕНИЕ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .43
19. ИСТОЧНИКИ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
20. ИСХОДНЫЙ КОД

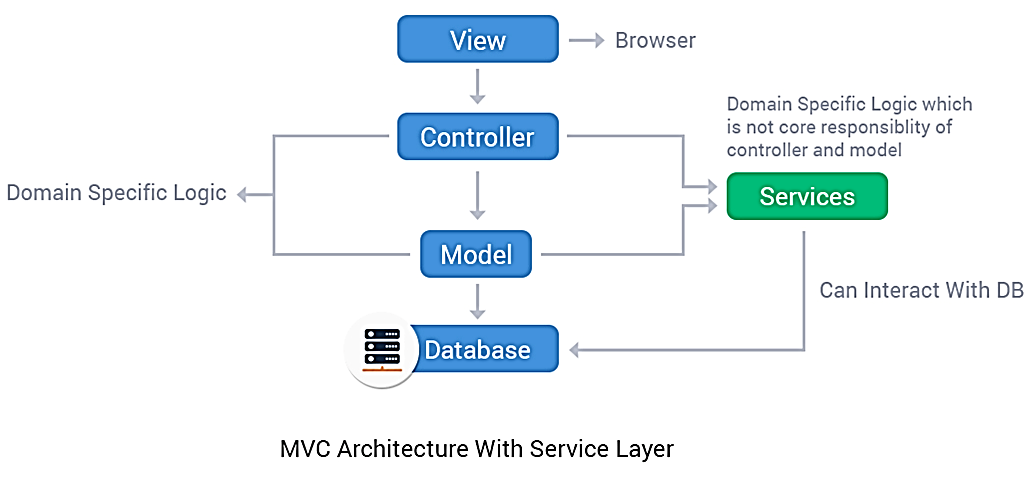
**ВВЕДЕНИЕ**

В настоящей работе мной будет описан процесс разработки веб-приложения для создания и публикации документации различных программных продуктов с применением платформы ASP. Формальная цель работы – создание приложения, решающего указанную проблему, фактическая – обучение работе с платформой ASP и фреймворком MVC.

Ограниченный объем изучаемого материала заставляет сделать упор на одну из частей веб-приложения – клиентскую или серверную. В настоящей работе была выбрана вторая как более интересная составителю. Изучение разработки клиентской части должным образом подразумевает изучение соответствующий современных технологий: React или Blazor.

В связи с невозможностью однозначно определить необходимую глубину описания процесса разработки (так, например, создание строки подключения к локальной базе данных состоит из ввода логина и пароля, однако не для всех пользователей этот процесс является очевидным), а также ограниченностью конечного объема документа, в настоящей работе не будут отражены все без исключения его этапы.

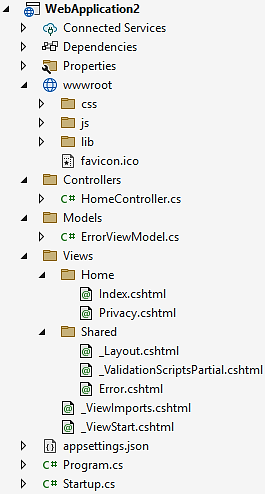
Настоящая работа описывает процесс разработки приложения с ошибками, в том числе известными на момент написания настоящей работы, однако дабы не засорять текст вставками с их описанием, анализ ошибок будет произведен в заключающих главах.

Перед началом основной части работы следует кратко описать, в чем состоит идея MVC (Model-View-Controller, Модель-Представление-Контроллер) как паттерна проектирования. Почти любое MVC приложение можно разделить на четыре основные части: модель, контроллер, представление, сервисы. Модели представляют собой не более чем структуры данных - классы со свойствами. Контроллер представляет собой обработчик запросов. Когда пользователь обращается к приложению, контроллер принимает его запрос, запрашивает данные из базы данных или сервисов, производит их обработку с помощью сервисов, передаёт обработанные данные в представление для их отображения. Блок-схема 1 представляет описанное визуально.

Блок-схема 1

**КРАТКОЕ ОПИСАНИЕ ШАБЛОНА ПРОЕКТА**

С целью понимания отправной точки разработки, в данной главе будет кратко описан автоматически создаваемый dotNET шаблон проекта MVC-приложения для SDK версии 3.1.

На рисунке 1 изображена файловая структура шаблона. Рассмотрим некоторые её элементы:

1. wwwroot – папка, предназначенная для хранения статических файлов – изображений или библиотек, используемых, как правило, в клиентской части приложения.
2. Program.cs – файл, несущий код метода Main.
3. Startup.cs – файл, несущий код методов Configure и ConfigureServices, вызываемых из метода Main.
4. Controllers – папка, несущая файлы с кодом контроллеров.
5. Models – папка, несущая файлы с кодом моделей.
6. Views – папка, несущая в себе файлы с визуальными представлениями.
7. appsettings.json – файл json-формата с настройками проекта.

Рассмотрим примеры упрощенного содержания некоторых файлов.

Рисунок 1

Программный код 1 – содержание файла Program.cs. Данный код указывает MVC, как и с какими параметрами ему следует создавать т.н. хост (сервер).

Программный код 1

public class Program

{

public static void Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

public static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

});

}

Программный код 2 – содержание файла Startup.cs. Данный код подключает части MVC-фреймворка к приложению. Также данный код указывает стандартный синтаксис запроса к сайту. Параметры обращения к методу MapControllerRoute указывают, что запрос может включать в себя указание на контроллер, который должен его обработать, конкретный метод контроллера, а также нести в себе мету в виде Id.

Программный код 2

public class Startup

{

public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)

{ Configuration = configuration; }

public void ConfigureServices(IServiceCollection services)

{ services.AddControllersWithViews(); }

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

app.UseDeveloperExceptionPage();

app.UseStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}");

});

}

}

Как можно заметить, сгенерированный код уже активно использует внедрение зависимостей. Сейчас я не буду останавливаться на данной теме.

Программный код 3 – содержание файла HomeController.cs. В данном случае можно увидеть обработку контроллером Home GET-запроса действия Index. Такая обработка будет выполняться, в частности, при обращении по адресу localhost:5000/Home/Index (естественно, при условии, что приложение запущено локально и прослушивает порт 5000, что является стандартным для разрабатываемых MVC-приложений). Контроллер возвращает представление, передавая ему модель, в данном случае – текст.

Программный код 3

public class HomeController : Controller

{

[HttpGet]

public IActionResult Index()

{

return View(model:"MyText");

}

}

Программный код 4 – содержание файла Index.cshtml. Представление получает от контроллера модель (напомню, модель, в контексте MVC, - не более, чем представление данных) и отображает её для пользователя, возвращая HTML-ответ.

Программный код 4

@model System.String

<div class="text-center">

<h1 class="display-4">@Html.Raw(Model)</h1>

</div>

На рисунке 2 изображен результат обращения к странице Index контроллера Home. За рамками описания оставлен т.н. Layout, формирующий остальную часть страницы, не описанную в файле Index.cshtml.

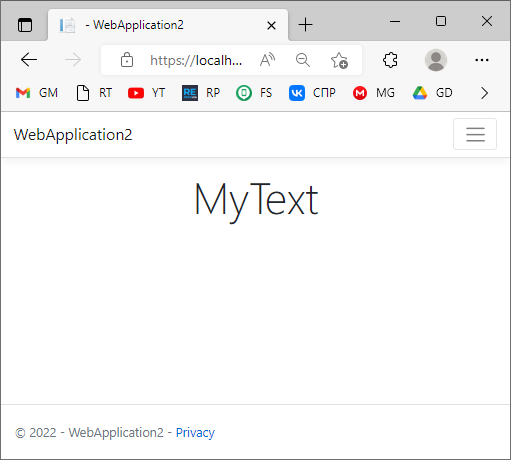


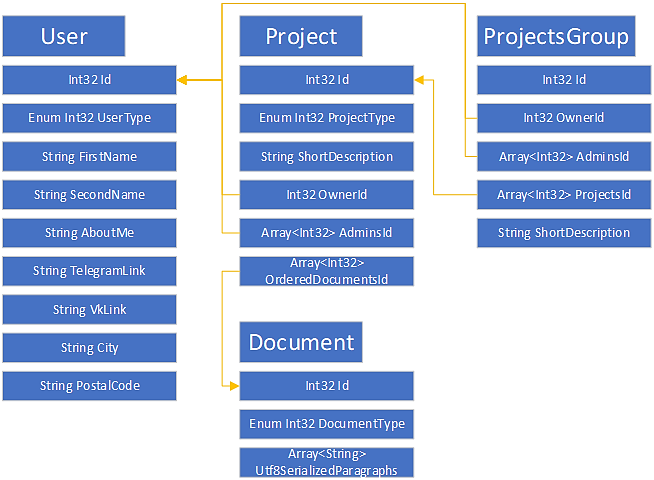
Рисунок 2

В данной главе была описана минимальная конфигурация MVC-приложения с целью создать у читателя примерное понимание структуры соответствующих приложений. Далее по ходу настоящей работы столь подробное описание применяться не будет, иначе объем работы легко сможет выйти более чем за 100 страниц.

**ВЫБОР ТИПА БД И ОПРЕДЕЛЕНИЕ ОСНОВНЫХ СУЩНОСТЕЙ. ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ. МИГРАЦИИ**

В своей работе я буду применять PostgreSQL как одну из самых популярных реализаций реляционных баз данных. При работе c ASP распространено применение ORM (Object-Relational Mapping, объектно-реляционное преобразование) EntityFramework, конкретной реализацией которого для PostgreSQL является NpgSQL. В проект EntityFramework и NpgSQL добавляются путем установки через Nuget (менеджер пакетов, встроенный в dotNET). ORM позволяет работать с объектами из базы данных как с обычными C#-классами, тем самым обеспечивая бесшовность взаимодействия C#-кода с SQL. Минусами такого подхода является сравнительно низкая производительность.

Определим основные сущности проекта. Ими будут являться «Пользователь», «Документ», «Проект». В будущем предусмотрено добавление «групп проектов». Блок-схема 2 демонстрирует их отношения.



Блок-схема 2

Создадим файлы DbDocument.cs, User.cs, Project.cs, в которых опишем соответствующие модели для базы данных, они представлены в программных кода 5, 6, 7 соответственно. Отмечу, что представленный код может содержать дополнительные поля (корректно называемыми свойствами в контексте C#), которые не отражены на блок-схеме 2, ибо были добавлены позднее. Также, вместо хранения JSON-строк они хранятся как массив байт (что не отражено на блок-схеме).

Программный код 7

public class Project

{

public enum PROJECT\_TYPE // заглушка

{

Common = 0

}

public int Id { get; set; }

public PROJECT\_TYPE ProjectType { get; set; }

public string? Name { get; set; }

public string? ShortDescription { get; set; }

public User? Owner { get; set; }

public User[]? Admins { get; set; }

public IEnumerable<DbDocument>? OrderedDocumentsId { get; set; }  
}

Программный код 5

Программный код 6

public class User

{

public int Id { get; set; }

public USER\_TYPE UserType { get; set; } = USER\_TYPE.Common;

public string DisplayedName { get; set; } = "New User";

public string? AboutMe { get; set; } = null!;

public string? TelegramLink { get; set; } = null!;

public string? VkLink { get; set; } = null!;

public string? City { get; set; } = null!;

public string? PostalCode { get; set; } = null!;

public string? String64\_ProfileImage { get; set; } = null!;

public string? EmailAdress { get; set; } = null!;

}

public class DbDocument

{

public int Id { get; set; }

public string Title { get; set; } = null!;

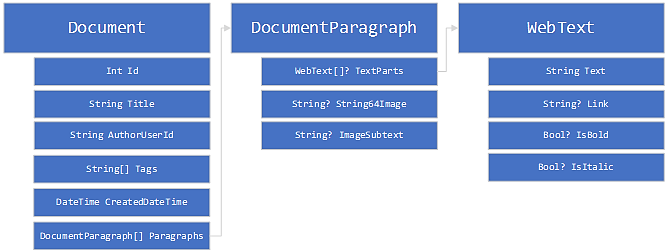
public User? Author { get; set; }

public DateTime CreatedDateTime { get; set; } = DateTime.UtcNow;

public DateTime UpdatedDateTime { get; set; } = DateTime.UtcNow;

public byte[] Utf8JsonSerializedParagraphs { get; set; } = null!;

}

Как уже было показано на блок-схеме 2 и в программном коде 5, в базе данных текст документа хранится в виде сериализованных (англ. serialized) его параграфов. Для этих целей в проекте используется Nuget-пакет Utf8JsonSerializer. Сущность документа, с которой осуществляется работа в C#-коде имеет другую, сложную многослойную структуру, она отражена на блок-схеме 3. Её можно описать словесно как «каждый документ несет несколько параграфов, каждый параграф несет несколько отрывков текста, каждый отрывок текста несет, помимо текста, набор его свойств».

Блок-схема 3

Таким образом, после извлечения из БД объект типа DbDocument («Database Document») должен быть преобразован в объект типа Document для дальнейшего использования. Основное действие преобразования состоит в десериализации свойства Utf8JsonSerializedParagraphs и получения вместо него массива объектов типа DocumentParagraph («параграф документа»). Для этих целей в классе DbDocument написаны статический метод FromDocument (реализующий т.н. фабричный метод) и нестатический метод ToDocument.

Теперь, когда структура базы данных заложена в C#-коде, нам необходимо предоставить её EntityFramework’у. Для этого мы создаём наследника класса DbContext и создаём в нём свойства с generic-типом DbSet, которому передаём типы, таблицы для которых необходимо создать в БД. Программный код 8 демонстрирует описанное.

public class WebsiteContext : DbContext

{

public WebsiteContext(DbContextOptions<WebsiteContext> options) : base(options) => this.Database.Migrate();

public DbSet<Website.Models.DocumentModel.DbDocument> DbDocuments

{ get; set; } = null!;

public DbSet<Website.Models.UserModel.User> Users

{ get; set; } = null!;

public DbSet<Website.Models.ProjectModel.Project> Projects

{ get; set; } = null!;

public DbSet<Website.Models.ProjectsGroupModel.ProjectsGroup> ProjectsGroups { get; set; } = null!;

}

Программный код 8

Данного наследника класса DbContext следует «зарегистрировать в MVC» путем прописывания соответствующего программного кода 9 в методе ConfigureServices. В качестве параметра метода UseNpgsql передается строка подключения. Должным образом её следует хранить в файлах appsettings.json или secrets.json (предпочтительнее, поскольку в общем случае срока подключения является конфиденциальной). В данном случае она «захардкожена» (англ. hardcoded) для упрощения примера.

Программный код 9

services.AddDbContext<WebsiteContext>(opts =>

{

opts.UseNpgsql(

"Server=localhost;Database=LuminodiodeWebsiteDb1;"

+ "Password=qwerty;username=postgres");

});

Для создания таблиц с объектами, вложенными в DbSet следует запустить терминал (например, PowerShell) в директории проекта и прописать команду  
dotnet ef migrations add Initial, где Initial – произвольное имя создаваемой миграции. Данная команда добавляет миграцию в очередь. Она будет выполнена при создании экземпляра класса WebsiteContext – как можно заметить в программном коде 8, в конструкторе класса. Суть миграции состоит в приведении таблиц базы данных в соответствие с объектам, переданными в DbSet’ы. Основное преимущество миграций состоит в возможности легко откатиться на предыдущие версии БД.

В дальнейшем для доступа к таблицам БД следует использовать класс WebsiteContext. DbSet позволяет работать с таблицами как с IQueryable типом. Данный тип, если описывать его кратко, позволяет сначала создать выражение, например Users.Where(…).Select(…).OrderBy(…).ThenBy(…), после чего выполнить его (англ. resolve), вызвав, например .ToList(). Это позволяет не создавать новые экземпляры объектов при каждом применении сортирующего/фильтрующего/преобразовывающего/группирующего метода в коллекции объектов DbSet.

**СОЗДАНИЕ ПЕРВОГО СЕРВИСА. «СЛУЧАЙНЫЕ ДАННЫЕ»**

В данной главе описывается создание первого сервиса разрабатываемого веб-приложения. В связи с этим процесс будет описан несколько более подробно, чем для последующих сервисов. Задача разрабатываемо сервиса – заполнить БД случайными данными (англ. seed data) для дальнейшего тестирования.

В начале разработки любого сервиса следует задать себе вопрос – «должны ли у данного сервиса быть настраиваемые параметры?». У данного сервиса – должны. Для их хранения обычно используется файл appsettings.json. Создадим в нём соответствующий раздел и запишем в него предполагаемые настройки – программный код 10. Такой подход позволяет изменять параметры сервисов без рекомпиляции приложения (англ. soft coded).

Программный код 10

{

**. . .**

"DataSeederSettings": {

"SeederIsEnabled": true,

"SeedIfQuantityOfUsersIsLessThan": 3,

"SeedIfQuantityOfDocumentsIsLessThan": 1000,

"SeedIfQuantityOfProjectsIsLessThan": 100,

"SeedIfQuantityOfProjectsGroupsIsLessThan": 0

},  
**. . .**  
}

Данные настройки предполагают следующее:

1. Сервис включен ("SeederIsEnabled": true).
2. Сервис будет добавлять случайные данные до тех пор, пока в базе данных менее чем три пользователя ("SeedIfQuantityOfUsersIsLessThan": 3).
3. Сервис будет добавлять случайные данные до тех пор, пока в базе данных менее чем тысяча документов ("SeedIfQuantityOfDocumentsIsLessThan": 1000).
4. Сервис будет добавлять случайные данные до тех пор, пока в базе данных менее чем 100 проектов "SeedIfQuantityOfProjectsIsLessThan": 100).
5. Сервис не будет добавлять группы проектов (SeedIfQuantityOfProjectsGroupsIsLessThan": 0).

Далее следует написать вспомогательный сервис для извлечения настроек из данного файла. В программном коде 11 приведен его частичный код, который демонстрирует извлечение одной из настроек из файла appsettings.json, извлечение остальных отличается лишь их именем или типом возвращаемого значения.

В данном сервисе я впервые применил внедрение зависимостей в самописном коде, конкретно – передал в конструктор IConfiguration. В рамках настоящей работы я не буду описывать данный механизм слишком подробно. На настоящем этапе скажем, что внедрение зависимостей состоит в регистрации тех или иных классов (например, WebsiteContext) в MVC с дальнейшей возможностью получить их в любом месте кода. Но не только. Внедрение зависимостей также позволяет автоматически передавать экземпляры зарегистрированных классов в конструкторы других классов. Данная особенность порождает инверсию управления – один из принципов ООП, который можно описать как «не программист управляет фреймворком, а фреймворк управляет кодом программиста», ведь именно фреймворк будет решать, как создавать экземпляры объектов, какие конкретно экземпляры передавать в их конструктор.

Программный код 11

public class RandomDataSeederSettingsProvider

{

protected readonly IConfiguration config;

public RandomDataSeederSettingsProvider(IConfiguration configuration)

=> this.config = configuration;

public int SeedIfQuantityOfUsersIsLessThan

{

get {

try {

return this.config

.GetRequiredSection

("DataSeederSettings:SeedIfQuantityOfUsersIsLessThan")

.Get<int>();

}

catch (Exception){

return 3; // default

}

}

}

. . .  
}

Стоит также затронуть тему «времени жизни» сервисов (англ. lifetime). Существуют сервисы, которые создаются один раз при запуске приложения и существуют с ним до его завершения (англ. Singleton, рус. Одиночка). Примером такого сервиса может служить менеджер сессий – он должен хранить сессии всё время, дабы пользователь мог продолжить пользоваться сайтом без повторной аутентификации, например, спустя неделю. С другой стороны, существуют сервисы, длительная жизни которых нецелесообразна – например, сервис поиска документов: если он обработал поисковой запрос, его дальнейшее существование нецелесообразно (англ. Scope, рус. Переходящий). Конечно, такое разделение весьма условно, и в целом почти любой функционал можно реализовать в любом формате, но для логической структурированности программы принято вносить разделения.

Конкретно данный сервис будет иметь тип Singleton, так как при обращении всегда возвращает одни и те же параметры, пересоздавать его экземпляр не имеет смысла. Зарегистрируем его соответствующей строкой программного кода 12 в метода ConfigureServices.

Программный код 12

public void ConfigureServices(IServiceCollection services)

{

. . .

services.AddSingleton<RandomDataSeederSettingsProvider>();  
. . .  
}

Теперь перейдем к разработке непосредственно сервиса случайных данных. Сразу понятно, что наш сервис будет внедрять зависимость от написанного только-что провайдера настроек (RandomDataSeederSettingsProvider), а также от контекста базы данных (WebsiteContext через IServiceScopeFactory). Программный код 13 демонстрирует конструктор класса, его поля, единственный открытый метод.

public class RandomDataSeederService

{

private readonly WebsiteContext context;

private readonly RandomDataSeederSettingsProvider SettingsProvider;

public RandomDataSeederService

(RandomDataSeederSettingsProvider SettingsProvider,

IServiceScopeFactory ctxFactory)

{

this.context = ctxFactory.CreateScope().ServiceProvider

.GetRequiredService<WebsiteContext>();

this.SettingsProvider = SettingsProvider;

}

public void SeedData()

{

if (SettingsProvider.SeederIsEnabled)

{

this.SeedData\_Users();

this.SeedData\_DbDocuments();

this.SeedData\_Projects();

this.context.SaveChanges();

}

}  
. . .  
}

Программный код 13

В программном коде 14 продемонстрирован один из закрытых методов класса – SeedData\_Users. демонстрировать все из них не имеет смысла, они делают ни что иное, как просто создают объект со случайными данными и добавляют его в базу данных.

Позволю себе сделать отступление и заметить, то данный код имеет некоторую небезопасность в связи с тем, что опирается на то, что база данных создаёт линейные индексы, т.е. 1,2,3,4,5 и т.д., что не всегда актуально для БД наподобие SqlServer, они могут создавать нелинейные индексы, например, таблица из пяти записей может нести совершенно случайные индексы, вроде 1,2,1001,1003,9991. Подробно описывать данную проблему не вижу смысла в связи с использованием другой реализации SQL.

Программный код 14

private void SeedData\_Users()

{

int NumToAdd = this.SettingsProvider.SeedIfQuantityOfUsersIsLessThan –

this.context.Users.Count();

if (NumToAdd <= 0) return;

var ToAddUsers = new Website.Models.UserModel.User[NumToAdd];

var startEmailIndex = this.context.Users.Count();

for (int i = 0; i < ToAddUsers.Length; i++)

{

var ToAdd = new Models.UserModel.User

{

EmailAdress = $"testemail{startEmailIndex + i}@gmail.com",

DisplayedName = "Admin",

};

this.context.Users.Add(ToAdd);

}

this.context.SaveChanges();

}

Возвращаясь к сервису – он закончен. Теперь его следует зарегистрировать в MVC точно также, как это было сделано в программном коде 12. Для данного сервиса можно использовать тип Singleton, дабы его можно было запустить при запуске приложения (в отличии от Scoped-сервисов). Добавлю, что именно поэтому зависимость от WebsiteContext была внедрена через IServiceScopeFactory, а не через WebsiteContext напрямую – в MVC невозможно внедрить зависимость Singleton-сервиса от Scoped-сервиса. Вызов единственного метода сервиса при запуске приложения показан в программном коде 15.

Программный код 15

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)

{

. . .

app.ApplicationServices.GetRequiredService<RandomDataSeederService>()

.SeedData();

. . .  
}

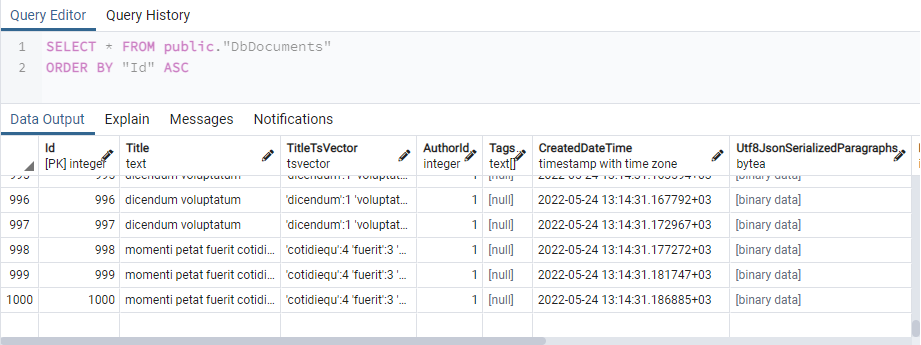
 Работу созданного сервиса проверим путем простого запуска приложения. В базе данных появились записи – «заказанная» нами в параметрах сервиса тысяча документов – рисунок 3, остальные таблицы также были сгенерированы в соответствии с настройками. В качестве случайного текста я применил «Lorem ipsum», но это останется за рамками работы.

Рисунок 3

**СЕРВИС ПОИСКА ДОКУМЕНТОВ**

В данной главе описывается создание сервиса поиска документов. Поиск документов будет осуществляться по заголовку. Очевидно, что при поиске должно использоваться т.н. нечеткое сравнение. Так, запросы «купить апельсины в москве» и «Москва купить апельсины» должны нести одинаковую смысловую нагрузку. Для этих целей в SQL существует тип tsVector. Данный тип представляет собой векторизованную форму того или иного текста. Векторизация ведет к исключению из текста служебных слов языка (предлогов, частиц, союзов) и приведению к единому стилю оставшихся (по падежу, числу). Далее происходит вычисление обратного индекса для каждого из слов. Так, заголовок текста «This title is a title where is a words which is repeating» будет преобразован в tsVector «'repeat':12 'titl':2,5 'word':9». Легко заметить преобразования «repeating» в «repeat», «title» в «titl», «words» в «word», а также исключение всех служебных слов.

Таким образом, первой задачей определяется добавить tsVector к каждому из документов в базе данных. В класс модели DbDocuments добавим соответствующее свойство: public NpgsqlTsVector TitleTsVector { get; set; }. Данное поле должно автоматически генерироваться при добавлении документа в БД. Для этого следует переопределить метод OnModelCreating используемого наследника DbContext (в данном случае – WebsiteContext) следующим образом – программный код 16.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Website.Models.DocumentModel.DbDocument>()

.HasGeneratedTsVectorColumn

(p => p.TitleTsVector, "english", p => new { p.Title })

.HasIndex(p => p.TitleTsVector)

.HasMethod("GIN");

}

Программный код 16

Вызов можно описать следующим образом:

1. Указываем modelBuilder’у на модифицируемую сущность вызовом метода Entity с соответствующим типом.
2. Указываем, какого типа автогенерируемую колонку следует создать – в данном случае её тип, tsVector, определяется вызываемым методом.
3. Три параметра метода HasGeneratedTsVectorColumn указывают на: название свойства, содержащего tsVector в классе модели; язык обработки текста; название свойства класса модели, несущего обрабатываемый текст. Таким образом, данный вызов указывает EF: «возьми значение из свойства Title, векторизуй его как английский текст, запиши результат в свойство TitleTsVector».
4. Вызов метода HasIndex указывает, какая колонка в базе данных соответствует генерируемому значению.
5. Вызов метода HasMethod с параметром “GIN” указывает EF, каким методом сохранить полученный tsVector. Для текстового поиска более предпочтительным является GIN, т.к. GiST допускает неточности.

Внесенные в код изменения подразумевают изменение структуры базы данных, посему следует создать новую миграцию путем powershell-команды dotnet ef migrations add AddingTsVector.

Далее переходим к созданию непосредственно сервиса. На мой взгляд, данный сервис не требует реализации извлечения настроек из файла, в нём нечего настраивать. Конструктор, поля и свойства класса – программный код 17.

Программный код 17

public class DocumentSearchService

{

private readonly IServiceScopeFactory DbContextScopeFactory;

private readonly FrequentSearchRequestsService FreqReqService;

public DocumentSearchService(

IServiceScopeFactory DbContextScopeFactory,

FrequentSearchRequestsService FreqReqService)

{

this.DbContextScopeFactory = DbContextScopeFactory;

this.FreqReqService = FreqReqService;

}

public DateTime ProceedDateTime { get;private set; }

public string? Request { get; private set; }

public List<DbDocument>? Response { get; private set; }

Как можно заметить, помимо внедрения зависимости от WebsiteContext через IServiceScopeFactory с целью обращения к базе данных, через конструктор внедряется зависимостей от некого FrequentSearchRequestsService – сервиса частых поисковых запросов. Разработка данного сервиса будет полностью описана в следующей главе. Сейчас скажем, что данный сервис может позволить получить ответ на запрос без обращения к базе данных, а также он должен получать уведомления от сервиса поиска об обрабатываемых запросах.

Рассмотрим единственный метод сервиса – программный код 18.

public async Task<List<DbDocument>> ProceedRequest(string UserRequest)

{

this.ProceedDateTime = DateTime.UtcNow;

var TryGetFromFreq = FreqReqService.GetSimilarRequestOrNull(UserRequest); //\*

if (TryGetFromFreq is not null && TryGetFromFreq.Response is not null) //\*

return TryGetFromFreq.Response; //\*

this.Request = UserRequest;

this.Response = await this.DbContextScopeFactory.CreateScope().ServiceProvider

.GetRequiredService<WebsiteContext>().DbDocuments.OrderByDescending(

d=>d.TitleTsVector.Rank(EF.Functions.WebSearchToTsQuery(UserRequest)))

.Take(20).Include("Author").ToListAsync();

this.FreqReqService.AddDocumentSearchServiceScope(this); //\*

return this.Response;

}

Программный код 18

Звездочками был помечен код, который может быть исключен без потери функциональности и служит исключительно для взаимодействия с сервисом частых поисковых запросов.

При обработке строки пользовательского запроса UserRequest метод сначала пытается запросить ответ от сервиса частых поисковых запросов. Если ответа не нашлось – будет выполнен запрос к базе данных. Запрос состоит в следующем: создание экземпляра WebsiteContext через метод CreateScope; получение IQueryable типа, содержащего DbDocuments; добавление команд запроса, в данном случае – сортировки по убыванию совпадения с векторизованной строкой запроса, где уровень совпадения вычисляется SQL-методом rank; выбор первых 20 элементов последовательности; добавление сущности автора к полученным элементам; выполнение (англ. resolve) запроса посредствам вызова ToListAsync. Минусами простого сравнения функцией rank является отсутствие учета «ассоциаций», когда разные слова могут определяться как «несущие одинаковую смысловую нагрузку», но такие алгоритмы – прерогатива крупнейших (и сложнейших) поисковых систем.

Перед возвратом результата метод передаёт свою сущность в сервис частых поисковых запросов для анализа. О нём – далее.

**СЕРВИС ЧАСТЫХ ПОИСКОВЫХ ЗАПРОСОВ**

В данной главе я попытаюсь воссоздать лишь частицу сложнейших алгоритмов оптимизации, используемых в больших поисковых системах. Крупные поисковики, маркетплейсы и т.д. часто используют сохранение самых популярных поисковых запросов с целью существенно снизить количество обращений к базе данных с миллионами записей. Данная оптимизация неактуальна для проектов с небольшими базами данных, но однозначно актуальна в учебных целях. Сервис должен будет определять наиболее часто встречающиеся поисковые запросы и сохранять определенное их количество вместе с ответами. Оптимальное количество сохраняемых запросов, в идеале, должно быть определено математически или эмпирически, но в рамках данной работы я возьму произвольные значения.

Опишу придуманный мной алгоритм словесно. Сервис частых поисковых запросов хранит в себе список поисковых запросов вместе с их частотой в смысле количества обработанных запросов. Для этих целей создан класс FrequenciedScopes, несущий в себе два свойства – частоту данного запроса и экземпляр сервиса поиска документов, обработавшего данный запрос – программный код 19.

public class FrequenciedScopes

{

public int Frequency { get; private set; }

public DocumentSearchService DocumentSearchServiceScope { get; set; }

public FrequenciedScopes(int Frequency, DocumentSearchService DocSearchScope)

{

this.Frequency = Frequency;

this.DocumentSearchServiceScope = DocSearchScope;

}

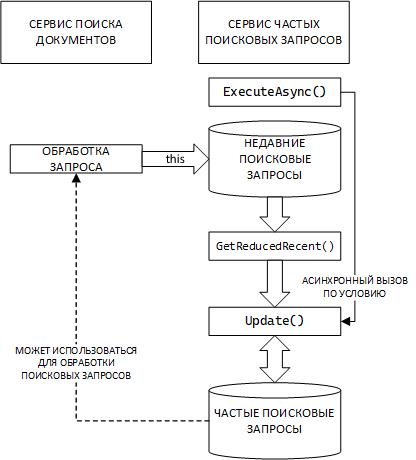
public void IncFreq(int count = 1) => Frequency = Frequency + count;

public void DecFreq(int count = 1) => Frequency = Frequency - count;

}

Программный код 19

Сервис поиска документов уведомляет сервис частых поисковых запросов о каждом поисковом запросе путем передачи в него своей сущности, несущей в себе три основных свойства: время обработки запроса, текст запроса, ответ на запрос, они же ProceedDateTime, Request, Response. Первоначально эти сущности сохраняются в коллекцию «недавних запросов». В определенный момент вызывается метод Update происходит обработка всей коллекции. В первую очередь необходимо редуцировать коллекцию временного хранения: если в ней найдутся несколько похожих запросов – свести их к одному с условной частотой, равной количеству ликвидированных экземпляров + 1 (включая оставшийся экземпляр). Далее происходит аналогичное сравнение запросов с постоянной коллекцией частых поисковых запросов. Если в них найдется похожий запрос, добавить к нему частоту, которую имеет элемент недавних запросов, если нет – добавить в неё сам элемент из коллекции недавних поисковых запросов. Движение данных изображено на блок-схеме 4, где широкие стрелки – движение данных, тонкие – вызовы.



Блок-схема 4

Перед переходом к конкретной реализации добавим настройки сервиса в файл appsettings.json. Существует смысл определить следующие параметры: максимальный интервал обновления значений сервиса; максимальное число сохраняемых частых запросов; частота проверки необходимости обновления (или же «обновлять не чаще, чем раз в …»); максимальное число новых запросов в очереди на обработку; необходимое число «очков сходства» для зачитывания запросов как одинаковых; время жизни ответа на запрос, после которого ответ на него должен быть обновлен путем обращения к базе данных. В соответствующем порядке названные параметры представлены в программном коде 20

Программный код 20

"FrequentSearchRequestsServiceSettings": {

"Interval\_msec": 300000,

"Interval\_numOfRecentRequests": 50,

"Interval\_updateNeededCheck\_msec": 10000,

"NumOfFrequentRequestsStored": 50,

"TokenSortRationNeededToCountAsSimilar": 90,

"ResponseLifetime\_msec": 300000

},

Переходим к конкретной реализации. Все поля класса являются закрытыми. Помимо них в программном коде 21 демонстрируются обращения к вспомогательному сервису-провайдеру настроек, созданному для данного сервиса. Код сервиса-провайдера настроек не демонстрируется в связи с его шаблонностью и почти полной аналогичностью такового для первого сервиса, описанного в данной работе, за исключением имен параметров.

Программный код 21

public sealed class FrequentSearchRequestsService : BackgroundService

{

// Хранит последние, еще не обработанные запросы

private readonly List<DocumentSearchService>

RecentRequests\_DocumentSearchServiceScopes = new();

// Хранит наиболее частые запросы

private List<FrequenciedScopes>

FrequentRequests = new();

// Интервал обработки последних запросов, даже если не набрано количество, мс

private int Interval\_msec

=> this.SettingsProvider.Interval\_msec;

// Количественный интервал обработки последних запросов

private int Interval\_numOfRecentRequests

=> this.SettingsProvider.Interval\_numOfRecentRequests;

// Интервал проверки необходимости провести обработку

private int Interval\_updateNeededCheck\_msec

=> this.SettingsProvider.Interval\_updateNeededCheck\_msec;

// Количество хранимых частых запросов

private int NumOfFrequentRequestsStored

=> this.SettingsProvider.NumOfFrequentRequestsStored;

// Очки сходства по Fuzz.TokenSortRation для засчитывания как одинаковых

private int TokenSortRationNeededToCountAsSimilar

=> this.SettingsProvider.TokenSortRationNeededToCountAsSimilar;

// Время жизни частого запроса. Обновлять ответ каждые… мс

private int ResponseLifetime\_msec

=> this.SettingsProvider.ResponseLifetime\_msec;

// Хранит время последнего обновления данных

private System.DateTime LastUpdateTime;

private readonly IServiceScopeFactory ScopeFactory;

private readonly FrequentSearchRequestsServiceSettingsProvider SP;

public FrequentSearchRequestsService(

IServiceScopeFactory ScopeFactory,

FrequentSearchRequestsServiceSettingsProvider SettingsProvider)

{

this.ScopeFactory = ScopeFactory;

this.SettingsProvider = SettingsProvider;

}

Рассмотрим два открытых метода класса. Один из них – для передачи данных в сервис, второй – для извлечения. Метод AddDocumentSearchServiceScope, как указывалось ранее, принимает экземпляры сервисов поиска документов. Метод GetSimilarRequestOrNull пытается найти ответ на пользовательский запрос среди частых запросов, если ничего не найдено – возвращает null, если ответ найден, но он устарел – обновляет его и возвращает. Код методов представлен в программном коде 22.

public async Task<DocumentSearchService?> GetSimilarRequestOrNull(string UserReq)

{

// Пытается найти подходящий ответ на переданный запрос среди частых запросов

try

{

var tryFind = this.FrequentRequests.First(x =>

Fuzz.TokenSortRatio(UserReq,x.DocumentSearchServiceScope.Request)

> this.TokenSortRationNeededToCountAsSimilar);

tryFind.IncFreq();

// Если ответ устарел - обновить сейчас

if ((tryFind.DocumentSearchServiceScope.ProceedDateTime - DateTime.UtcNow).Seconds > this.ResponseLifetime\_msec)

{

tryFind.DocumentSearchServiceScope = this.ScopeFactory

.CreateScope().ServiceProvider

.GetRequiredService<DocumentSearchService>();

await tryFind.DocumentSearchServiceScope.ProceedRequest(UserReq);

}

return tryFind.DocumentSearchServiceScope;

}

// Ничего не найдено (следует обработать запрос через БД)

catch (System.InvalidOperationException)

{

return await Task.FromResult<DocumentSearchService?>(null)

}

}

Программный код 22

Теперь рассмотрим алгоритмическую часть сервиса. Метод GetReducedRecent, как указывалось ранее, минимизирует коллекцию недавних запросов путем объединения похожих запросов.

Программный код 23

private List<FrequenciedScopes> GetReducedRecent()

{

var recent = this.RecentRequests\_DocumentSearchServiceScopes

.Select(SearchServiceScope => new // Присвоение частоты 0

FrequenciedScopes(0,SearchServiceScope)).ToList();

// Вычисление частоты для каждого недавнего запроса и удаление похожих

for (int i = 0; i < recent.Count - 1; i++){

for (int r = i + 1; r < recent.Count; r++){

/\*Если найдены схожие строки: одной добавить частоты, вторую удалить.

\* Данный алгоритм должен быть улучшен: сначала из двух похожих

\* строк следует выводить наиболее популярную, далее оставлять её, а не \* просто первую попавшуюся \*/

if (Fuzz.TokenSortRatio(recent[i].DocumentSearchServiceScope.Request, recent[r].DocumentSearchServiceScope.Request)

> this.TokenSortRationNeededToCountAsSimilar)

{

recent[i].IncFreq(recent[r].Frequency + 1);

recent.RemoveAt(r--);

}

}

}

return recent;

}

Теперь применим полученные данные о частоте недавних запросов в коллекции частых запросов. В программном коде 24 представлен код метода Update и код вспомогательно метода IndexOfMostSimilarByTokenSortRatio, возвращающего индекс экземпляра сервиса поиска из переданного списка, запрос которого наиболее поход на переданный.

Программный код 24

private int IndexOfMostSimilarByTokenSortRatio(

IList<FrequenciedScopes> SearchServices,

string SearchRequest,

out int SimilarityScore)

{

SimilarityScore = 0;

int IndexOfMostSimilar = 0;

for (int i = 0; i < SearchServices.Count; i++)

{

var score =Fuzz.TokenSortRatio(SearchServices[i]

.DocumentSearchServiceScope.Request, SearchRequest);

if (score > SimilarityScore)

{ SimilarityScore = score; IndexOfMostSimilar = i; }

}

return IndexOfMostSimilar;

}

private void Update()

{

var recent = GetReducedRecent();

// Учет обработанных недавних запросов в общей статистике частых запросов

for (int i = 0; i < recent.Count; i++)

{

// Если похожая строка существует, то инкрементировать её частоту int score;

int index =this.IndexOfMostSimilarByTokenSortRatio

(this.FrequentRequests,

recent[i].DocumentSearchServiceScope.Request, out score);

if (score > this.TokenSortRationNeededToCountAsSimilar)

{

this.FrequentRequests[index].IncFreq(recent[i].Frequency + 1);

recent.RemoveAt(i--);

}

}

// Если строка не совпадает ни с одним из существующих частых - добавить её

this.FrequentRequests.AddRange(recent);

// Сортировка запросов по убыванию популярности и ограничение их количества

this.FrequentRequests = this.FrequentRequests.OrderByDescending(

x => x.Frequency).Take(this.NumOfFrequentRequestsStored).ToList();

// Снижение частоты каждого запроса с целью вытеснения устаревших частых

for (int i = 0; i < this.FrequentRequests.Count; i++)

this.FrequentRequests[i].DecFreq();

// Очистка коллекции недавних запросов

this.RecentRequests\_DocumentSearchServiceScopes.Clear();

}

Единственный оставшийся метод сервиса – ExecuteAsync. Данный метод вызывает метод Update по одному из двух условий – либо по времени, согласно настройкам, либо по превышению максимального количества хранимых недавних запросов, согласно настройкам, но делает это не чаще определенного интервала, согласно настройкам. Код метода представлен в программном коде 25.

Программный код 25

protected async override Task ExecuteAsync(CancellationToken ct)

{

while (!ct.IsCancellationRequested)

{

// Обновить если накоплено достаточное число недавних запросов

if (this.RecentRequests\_DocumentSearchServiceScopes.Count()

> this.Interval\_numOfRecentRequests) this.Update();

// Обновить если прошел интервал обновления

if (((DateTime.UtcNow - this.LastUpdateTime).TotalSeconds

> (this.Interval\_msec / 1000))

&& this.RecentRequests\_DocumentSearchServiceScopes.Count

> 0) this.Update();

// Интервал проверки необходимости обновления

await Task.Delay(this.Interval\_updateNeededCheck\_msec);

}

}

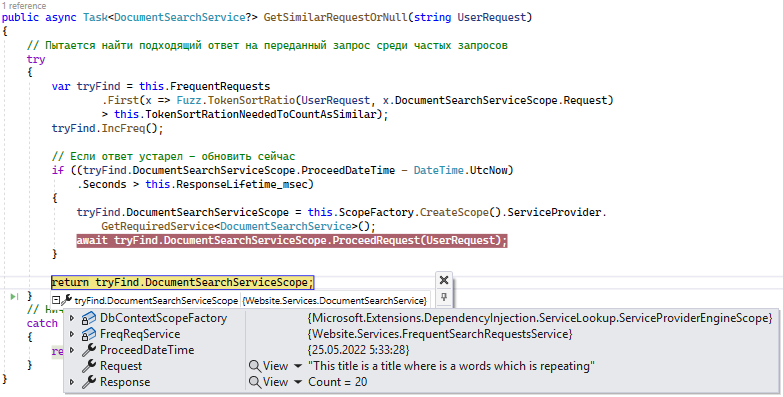
 Ручное тестирование сервисов поиска я проведу путем постановки двух брейкпоинтов (англ. breakpoint) в методе GetSimilarRequestOrNull. Первый – на код внутри условия, проверяющего необходимость обращения к БД, дабы проверить, что оно не будет выполнено, второй - на возврат значения из коллекции частых запросов. Далее я отправлю несколько одинаковых запросов, выжду интервал обновления сервиса, который заранее поставлю на 10 секунд, чтобы не ждать слишком долго, после чего отправлю ещё один такой же запрос, проверив работу брейкпоинта. Срабатывание продемонстрировано на рисунке 5. Автоматизированное тестирование будет рассмотрено в следующих главах.

Рисунок 4

**СЕРВИС НЕДАВНО ОПУБЛИКОВАННЫХ ДОКУМЕНТОВ**

В данной главе описывается реализация простого сервиса, который предоставляет коллекцию недавно опубликованных на сайте документов. Его функция – избавиться от необходимости обращаться к БД каждый раз, когда пользователь обращается к общим страницам сайта. Данный сервис позволяет реализовать ленту опубликованных документов.

Поскольку в предыдущих трех главах достаточно подробно описан процесс создания сервисов, в последующих я буду использовать преимущественно словесное описание и минимум кода, дабы не увеличивать объем работы.

Назовем сервис RecentDocumentsBackgroundService. Данный сервис будет иметь одну настройку – интервал обновления Interval-msec. Сервис представляет собой Singleton и имеет один экземпляр. Класс сервиса является наследником BackgroundService. В методе ExecuteAsync с периодом, определяемым настройками, происходит извлечение последних документов из базы данных и запись их в свойство RecentDocuments – программный код 26.

Программный код 26

protected async override Task ExecuteAsync(CancellationToken ct)

{

while (!ct.IsCancellationRequested)

{

var NewRecent = this.context.DbDocuments

.OrderByDescending(x=> x.CreatedDateTime)

.Take(this.NumberOfStoredDocs).Include("Author");

this.RecentDocuments = NewRecent.Select(x => x.ToDocument()).ToList();

await Task.Delay(this.Interval\_msec);

}

}

}

Неплохой идеей является добавлять документ в «недавно опубликованные» в момент его добавления в БД. Например, пользователь публикует новый документ – он сразу же добавляется не только в БД, но и в коллекцию сервиса недавних документов, таким образом сервис узнает о нем без обращения к БД, однако, поскольку я не использую т.н. репозиторий (англ. repository pattern), реализация подобного может вылиться в чрезмерную связанность компонентов сайта, поэтому было принято решение отказаться от реализации подобного функционала.

**РЕАЛИЗАЦИЯ МИНИМАЛЬНОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТФЕЙСА. ВИЗУАЛЬНЫЙ КОМПОНЕНТ «ЛЕНТА ПУБЛИКАЦИЙ»**

На сайте реализовано несколько сервисов. Для оценки их функционально «вживую» следует написать минимальный интерфейс пользователя. Возвращаясь к первой после введения главе данной работы, повторю, что создание «страницы» веб-приложения состоит в создании соответствующего действия контроллера и визуального представления. Создадим контроллер DocumentController, отвечающий за работу с документами. В нём определим страницу Summary, которая будет являться его страницей по-умолчанию. На ней разместим панель поиска и ленту последних опубликованных документов. Страницы по-умолчанию, как правило, не требуют каких-либо действий по части контроллера, поэтому его код в данном случае будет просто возвращать визуальную часть – программный код 27. О классе AMyController будет рассказано позднее – сейчас скажем, что он внедряет зависимость от фабрики IServiceScopeFactory.

Программный код 27

public sealed class DocumentController : AMyController

{

private readonly Website.Services.RecentDocumentsBackgroundService

recentDocumentsProvider;

public DocumentController(IServiceScopeFactory ScopeFactory)

: base(ScopeFactory) { }

[HttpGet]

public ViewResult Summary()

{

return this.View();

}

}

Итак, при обращении к сайту по адресу localhost:5000/Document в качестве ответа будет отправлено визуальное представление Summary. Сейчас стоит пояснить, что в MVC принято делать почти все представления частичными. Для приложения создаётся т.н. Layout – базовая часть любой возвращаемой HTML-страницы. В моём случае она включает navbar в верхней части страницы и footer в нижней. Я не буду демонстрировать её код, так как он является шаблонным. Между верхней и нижней частями страницы вызывается метод RenderBody – программный код 28, который совмещает возвращенное представление со страницей Layout. Данный код пишется в cshtml файлах, они совмещают C# и HTML, сродни совмещению JS и HTML в файлах jsx, которые использует React.

Программный код 28

<main role="main" class="container">

@RenderBody()

</main>

Само возвращаемое представление будет содержать не более чем input для ввода запроса поиска документов и ленту недавно опубликованных документов.

Для этих целей следует написать визуальный компонент (C#-тип ViewComponent), использующий сервис недавно опубликованных документов. Визуальные компоненты ASP похожи на элементы визуального фреймворка React, но их рендер происходит на сервере. Отличие визуального компонента от частичного представления (C# PartialView, часть html страницы, которая может быть вынесена в отдельный файл) состоит в том, что он может иметь дополнительную серверную логику. Визуальные компоненты поддерживают внедрение зависимостей. Таким образом, если на странице требуется отобразить какую-то часть, которая является унифицированной и не привязана к конкретной странице, то лучше вынести её обработку из метода контроллера в визуальный компонент с целью реализации разделения обязанностей. Так, например, лента последних документов, которая может быть реализована как для всех последних документов, так и для последних публикаций отдельного пользователя, путем простого применения фильтра по Id автора, может быть вынесена во ViewComponent.

Создадим соответствующий класс RecentDocumentsCardsViewComponent, наследуемый от ViewComponent. Данный класс внедряет зависимости от WebsiteContext и RecentDocumentsBackgroundService. В классе имеется единственный открытый метод InvokeAsync с параметрами   
‘(int? filterByUserId = null, int? MaxQuantity = null)’. Данный метод, если не передан фильтр по Id автора, возвращает представление с последними опубликованными документами из соответствующего сервиса, описанного в предыдущей главе, в ином случае он выполняет запрос к БД; также есть возможность определить количество отображаемых документов. В папке Views\Shared (см. первую после вступления главу), где Shared – общий, означает, что компонент будет доступен в различных частях приложения, создаём папку Components\RecentDocumentsCards c файлом Default.cshtml. В нем я помещу визуальную часть создаваемого компонента. Данный компонент будет не более чем осуществлять рендер переданных документов в качестве «карточек» (используя bootstrap-класс card) – программный код 29.

Программный код 29

@model IEnumerable<Website.Models.DocumentModel.Document>

@addTagHelper \*, Microsoft.AspNetCore.Mvc.TagHelpers

@foreach(var doc in Model)

{

<div class="p-1">

<**partial** **name**="~/Views/Shared/PartialViews/Document/\_DocumentCardPartialView.cshtml" **model**=doc />

</div>

}

Как можно заметить, компонент сразу же обращается к частичному представлению. В моём проекте используется многослойная структура частичных представлений для отображения документов. Я не буду описывать её в текущей работе, она является достаточно простой, но займет много места.

Вернемся к первоначально разрабатываемой странице. Помимо формы поиска добавим в неё только что созданный компонент. Частичный код созданной страницы представлен в программном коде 30.

Программный код 30

<div class="p-3">

<**partial** **name**="~/Views/Shared/PartialViews/Document/\_DocumentSearchFormPartialView.cshtml" **model**=string.Empty />

</div>

. . .

<div class="border border-primary">

<div class="w-100 text-center p-1"><h5>Recent docs</h5></div>

@await Component.InvokeAsync(

Website.ViewComponents.RecentDocumentsCardsViewComponent.nameofThis)

</div>

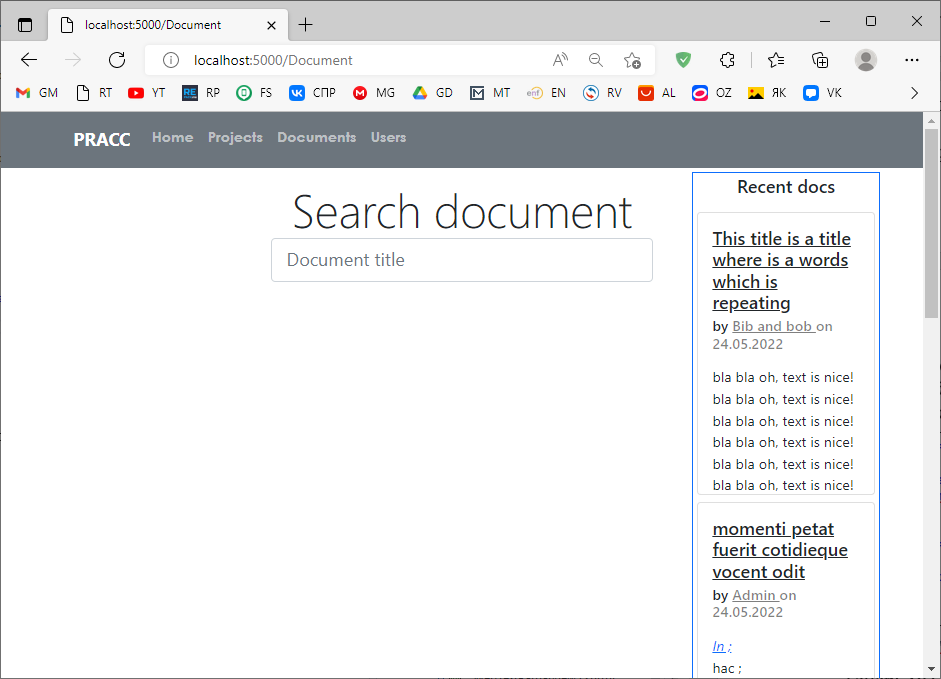
 Теперь мы имеем возможность открыть сайт на созданной странице – рисунок 5.

Рисунок 5

**ОПРЕДЕЛЕНИЕ POST-ЗАПРОСА ДЛЯ ПОИСКА ДОКУМЕНТОВ**

В конце предыдущей главы была определена форма поиска документа. Теперь следует определить соответствующий метод контроллера для обработки запроса поиска – программный код 31.

[HttpPost]

public async Task<ViewResult> Search(string SearchRequest)

{

this.TempData.Add("OriginalRequest", SearchRequest);

return this.View((await base.ServiceProvider

.GetRequiredService<DocumentSearchService>()

.ProceedRequest(SearchRequest)).Select(x => x.ToDocument()));

}

Программный код 31

Код визуальной части страницы не будет демонстрироваться в связи с большим размером кода. При вводе запроса будет возвращена следующая страница – рисунок 6.

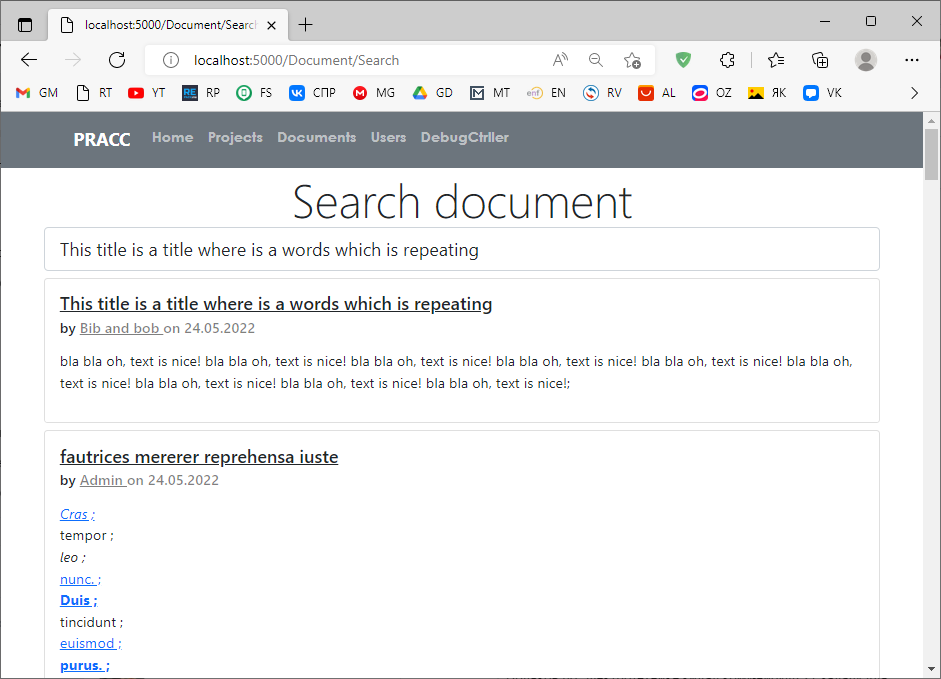
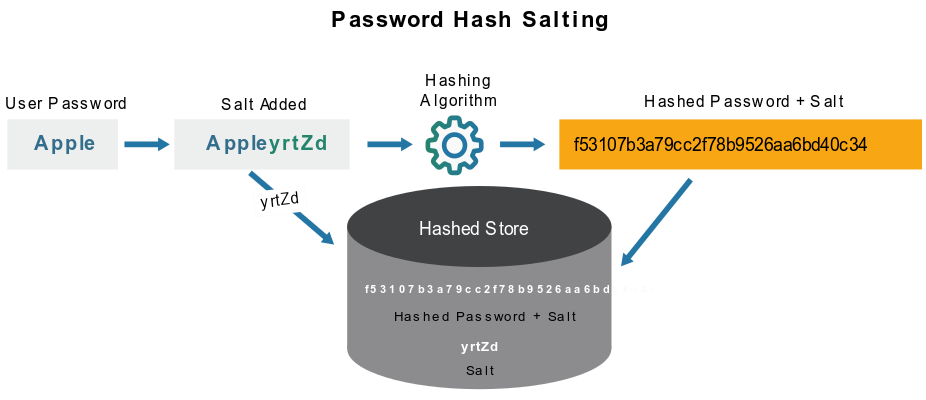


Рисунок 6

**АВТОРИЗАЦИЯ. СЕРВИС ПАРОЛЕЙ**

В данной главе будет рассмотрено создание сервиса паролей. Его суть состоит в выполнении операций хеширования и проверки паролей.

Говоря о теории, «лучший способ хранить пароли – не хранить пароли», хранить их хеш. Так, при регистрации пользователь отправляет пароль на сервер, который добавляет к нему случайный набор байт (т.н. соль) и вычисляет хеш получившейся последовательности, после чего пароль выкидывается, хеш и соль сохраняются в базу данных – блок-схема 5.



Блок-схема 5

По сути, мы получаем хеш и два ключа, один из которых открытый — соль, второй закрытый — сам пароль. Соль хранится на сервере, пароль у пользователя — для вычисления хеша нужны оба. Потенциальный злоумышленник, даже выкачав базу данных, не получит доступа к паролям, а только к бессмысленному набору байт.

Частичный код сервиса: функций хеширования и проверки представлен в программном коде 32.

Программный код 32

public byte[] HashPass (string PlainTextPassword, out byte[] GeneratedSalt)

{

GeneratedSalt = GenerateSalt();

var PasswordBytes = Encoding.UTF8.GetBytes(PlainTextPassword);

var SaltedPassword = PasswordBytes.Concat(GeneratedSalt).ToArray();

return SHA512.HashData(SaltedPassword); // should better use PBKDF2 ?

}

public bool ConfirmPass

(string PlainTextPassword, byte[] HashedPassword, byte[] Salt)

{

var PasswordBytes = Encoding.UTF8.GetBytes(PlainTextPassword);

var SaltedPossiblePassword = PasswordBytes.Concat(Salt).ToArray();

return SHA512.HashData(SaltedPossiblePassword).SequenceEqual(HashedPassword);

}

**АВТОРИЗАЦИЯ. МЕНЕДЖЕР СЕССИЙ**

В данной главе будет рассмотрено создание сервиса, управляющего сессиями. В момент авторизации на сайте пользователю должен быть выдан токен сессии. Обычно данный токен записывает в куки. Токен отправляется на сервер при каждом запросе и указывает, от имени какого аккаунта совершается действие. При развитой клиентской части существует возможно не отправлять куки с запросами, которые этого не требуют, однако в данной работе такой подход не рассматривается. В случае написания «реалистичного» менеджера сессий требуется добавить возможность их сохранения в базу данных, дабы пользователь мог продолжить работу даже спустя месяц не повторяя вход в аккаунт, однако в настоящей работе сессии будут хранится исключительно в оперативной памяти.

Задача сервиса состоит не более чем в создании токена по требованию c возможностью его последующей валидации. Токен должен иметь время жизни. Для этих целей создадим раздел с настройками данного сервиса в файле appsettings.json и вспомогательный сервис для их извлечения. Настройки содержат три параметра: время жизни сессии, длину идентификатора сессии, интервал очистки словаря сервиса от устаревших сессий - SessionLifetime\_secs, SessionIdStringLength\_bytes, SessionsCleanUpInterval\_secs соответственно.

Сервис содержит в качестве приватного поля словарь с ключами типа String и значениями типа SessionInfo – специально созданного для целей этого сервиса типа, содержащего поля, указывающие на Id пользователя и дату окончания действия токена. В методе, помимо ExecuteAsync, выполняющего очистку словаря от устаревших сессий, присутствуют два метода – CreateSession и ValidateSession, отвечающих за создание и валидацию сессий соответственно – программный код 33.

Программный код 33

public void CreateSession(int UsedId, out string CreatedSessionId) {

string SessionId = null!;

do {

SessionId =Convert.ToBase64String(

System.Security.Cryptography.RandomNumberGenerator

.GetBytes(this.SessionIdLength));

} while(this.Sessions.ContainsKey(SessionId));

var ValidThrough = DateTime.UtcNow.AddSeconds(this.SessionLifeTimeSecs);

CreatedSessionId = SessionId;

this.Sessions.Add(SessionId, new SessionInfo(UsedId, ValidThrough));

}

public bool ValidateSession(string SessionId, out SessionInfo? FoundSession) {

FoundSession = null;

if (!this.Sessions.TryGetValue(SessionId, out FoundSession)) return false;

if (!FoundSession.IsValidNow) return false;

FoundSession.ValidThrough = DateTime.UtcNow.AddSeconds( SessionLifeTimeSecs);

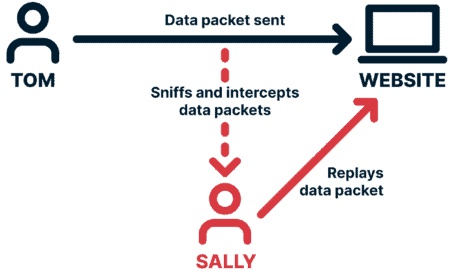
return true;

}

**АВТОРИЗАЦИЯ. ЗАЩИТА ОТ ПОВТОРНОГО ВОСПРОИЗВЕДЕНИЯ**

В данной главе будет описана реализация защита от атаки повторного воспроизведения (англ. replay attack) и перехвата пароля в незащищенном соединении. Заранее отмечу, что в ходе данной главы т.н. соль хеширования именуется ключом, что не совсем корректно, ибо ей нельзя ничего «открыть», однако само словосочетание более привычно.

Атака повторного воспроизведения является разновидностью атаки «человек посередине» (англ. Man-In-The-Middle). Суть атаки состоит в простом перехвате и повторной отправке на сервер запроса настоящего клиента – рисунок 7. Например, пользователь пытается авторизоваться на сайте. Он отправляет на сервер POST-запрос по защищенному HTTPs соединению, в теле запроса передавая свой логин и пароль в виде простого текста. Злоумышленник перехватывает соединение, после чего дублирует вышеуказанный запрос. В данном случае не требуется расшифровка данных, благодаря чему **атака может быть произведена даже на защищенное соединение**. Сервер, получая оба запроса, отправляет на них два ответа с разными токенами сессии, посему авторизация произойдет как на компьютере пользователя, так и на клиенте злоумышленника.



Защита от данной атаки состоит в создании уникальных токенов авторизации. Один из конкретных вариантов — передача хеша пароля, вычисленного по двум открытым ключам, один из которых — уникальный. **Первым открытым ключом** является т.н. соль, которая хранится в базе данных сервера и является постоянной до смены пароля (описана в главе «сервис паролей»). При GET-запросе страницы авторизации сервер должен сгенерировать уникальный токен авторизации с определенным временем жизни, например, 5 минут — для него формируется **второй открытый ключ**. Оба ключа передаются клиенту. Перед отправкой на сервер клиент сначала вычисляет хэш введенного пользователем пароля по первому ключу, потом по второму, после чего отправляет на сервер вычисленный хэш и токен авторизации. Сервер уже имеет вычисленный по первому ключу хэш — это есть хэш пароля, хранимый в базе данных, ему остается вычислить его по второму ключу, который он сохранил привязанным к токену авторизации. После первого же вычисления токен авторизации выбрасывается. Теперь, если злоумышленник попытается повторить запрос — сервер не сможет его обработать — токен и ключ уже были уничтожены, для сервера они невалидны.

Рисунок 7

Таким образом, сервер всё еще не хранит пароль пользователя, в то же время клиент не отправляет на сервер никаких паролей. Последнее архитектурно исключает возможность перехвата пароля по незащищенному соединению — что актуально, ведь многие люди используют один пароль для множества сервисов. Однако стоит понимать, что в случае использования незащищенного соединения возможность перехвата токена сессии никак не исключена.

Отмечу, что в отличии от аутентификации, при регистрации пользователя, пароль всё-таки должен будет быть передан на сервер, иначе не представляется возможным проверить его валидность. В таком случае не следует производить автоматическую аутентификацию пользователя после регистрации — при использовании защищенного соединения повторная отправка запроса ничего не даст злоумышленнику. При использовании же незащищенного соединения передаваемый пароль может быть легко извлечен со всеми последствиями.

Перейдем к конкретной реализации. Сначала требуется реализовать сервис, отвечающий за создание токенов аутентификации - AuthTockenService. В качестве настроек сервис будет иметь время жизни токена, длину токена, длину ключа, интервал очистки устаревших токенов. Функции создания и валидации токена почти эквивалентны таковым у сервиса паролей, описанного ранее, поэтому не будут демонстрироваться. Далее я реализую клиентскую часть для выполнения хеширования пароля. Понятно, что ввод логина и пароля должен являться отдельным действием в такой системе аутентификации, однако при использовании React/Blazor данный функционал может быть реализован «бесшовно», т.е. несмотря на одновременный ввод пользователем логина и пароля в форму, условный React сначала отправит на сервер запрос с логином, получит для него токен авторизации и ключи, применит их к паролю, отправит следующий запрос с токеном авторизации и хешированным паролем. Поскольку в рамках данной работы у меня не было возможности изучить React, я реализую данный функционал на двух отдельных страницах. Первая будет отправлять на сервер логин и получать токен и ключ, записывая их в куки, вторая – выполнять действия с паролем. Исходя из этого напишем скрипт для страницы ввода пароля, приняв название input’а для пароля за ‘passwordInputId’, – программный код 34.

<script>

function getCookie(name) {

var match = document.cookie.match(new RegExp('(^| )' + name + '=([^;]+)'));

if (match) return match[2];

}

function ToCommonBase64(str) {

return ((str + '='.repeat(4 - (str.length % 4)))

.replaceAll('-', '+').replaceAll('\_', '/'));

}

function ToUrlBase64(str) {

return (str.replaceAll('+', '-').replaceAll('/', '\_').replaceAll('=', ''));

}

function BufToBase64(bufferArray) {

return btoa(String.fromCharCode.apply(null, new Uint8Array(bufferArray)));

}

function Base64ToBuf(str64) {

return Uint8Array.from(atob(str64), c => c.charCodeAt(0));

}

function OnPasswordSubmitClick() {

PasswordSalt64 = ToCommonBase64(getCookie(@Html.Raw(

'\''+Website.Controllers.UserController.PasswordSaltString64+'\'')));

AuthSalt64 = ToCommonBase64(getCookie(@Html.Raw(

'\''+Website.Controllers.UserController.AuthHashKeyString64+'\'')));

var PasswordSaltBytes = Base64ToBuf(PasswordSalt64);

var AuthSaltBytes = Base64ToBuf(AuthSalt64);

var UserPasswordBytes = new Uint8Array(new TextEncoder()

.encode(document.getElementById("passwordInputId").value));

var Salted = Array.prototype.concat.apply(

[...UserPasswordBytes, ...PasswordSaltBytes]);

var SaltedHashed = sha512.array(Salted);

var SaltedHashedWithKey = Array.prototype.concat.apply(

[...SaltedHashed, ...AuthSaltBytes]);

var SaltedHashedWithKeyHashed = sha512.array(SaltedHashedWithKey);

var AsBase64url = ToUrlBase64(BufToBase64(SaltedHashedWithKeyHashed));

var Req = new XMLHttpRequest();

Req.open('POST', '/User/NewAuthPassword', false);

Req.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');

Req.send('@Html.Raw(Website.Controllers.UserController

.PasswordHashedByClientFormName)' + "=" + AsBase64url);

document.open();

document.write(Req.responseText);

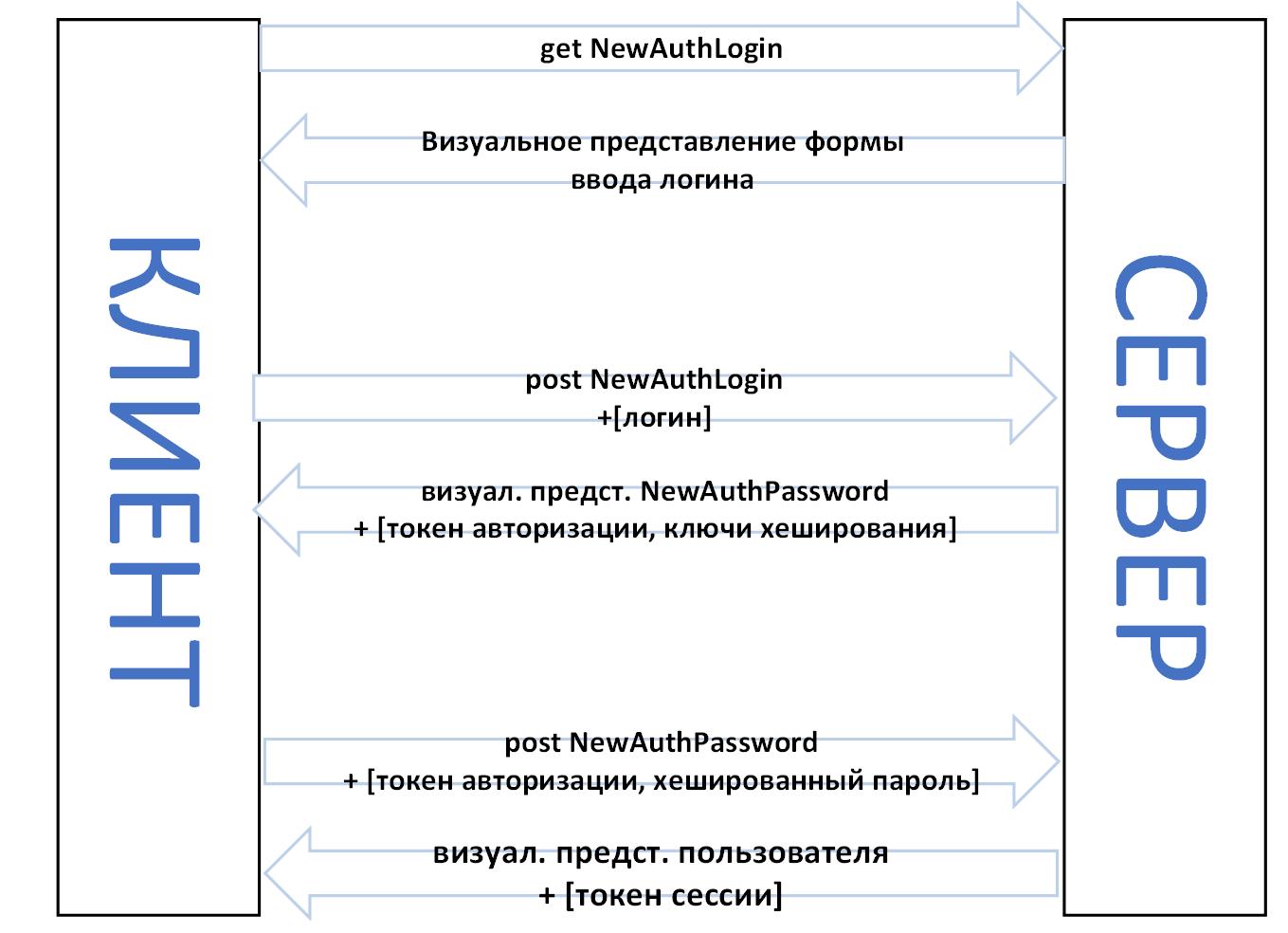
document.close();

}

</script>

Программный код 34

**АВТОРИЗАЦИЯ. СОЗДАНИЕ ИНТЕРФЕЙСА**

 Основные компоненты самописной авторизации настоящего проекта были описаны в трех предшествующих главах. В этой главе будет произведено их объединение путем создания соответствующих страниц (действий) в контроллере UserController – контроллере пользователей. Написание регистрации я оставлю за рамками данного текста, ибо она имеет более простую логику. Создадим два действия контроллера – NewAuthLogin и NewAuthPassword. Для каждого определим Get и Post запросы. Они должны реализовать примерно следующую последовательность действий – блок-схема 6.

Блок-схема 6

Get-запрос NewAuthLogin возвращает визуальной представление с простой формой. Она передаёт на сервер логин в виде простого текста. В настоящем проекте в качестве логина используется адрес электронной почты, который был указан EntityFramework’у как AlternateKey (т.е. как второй уникальный ключ, аналог Id) с целью оптимизации поиска по БД. После получения логина, логика Post-запроса NewAuthLogin производит перенаправление на действие NewAuthPassword – программный код 35.

Get-запрос NewAuthPassword производит проверку переданного логина. Далее он возвращает страницу ввода пароля пользователю, передавая вместе с ней на клиент куки c токеном авторизации и ключами хеширования – программный код 36.

Программный код 35

[HttpPost]

public async Task<IActionResult> NewAuthLogin(Website.Models.Auth.LoginOnly LO) {

if (!ModelState.IsValid) return View(LO);

Response.Cookies.Append(CurrentLoginCoockieName, LO.EmailPlainText);

return RedirectToAction(nameof(NewAuthPassword));

}

Программный код 36

[HttpGet] public async Task<IActionResult> NewAuthPassword() {

var passedLogin = Request.Cookies[CurrentLoginCoockieName];

if (passedLogin is null) return new StatusCodeResult(400);

Website.Models.Auth.LoginOnly LO = new() { EmailPlainText = passedLogin };

if (!TryValidateModel(LO)){

AddAlertToPageTop(new("Bad email format", Alert.ALERT\_COLOR.Red));

return this.RedirectToAction(nameof(NewAuthLogin));

}

var found = await this.context.Users.FirstOrDefaultAsync(

x => x.EmailAdress!.Equals(LO.EmailPlainText));

if (found is null){

base.AddAlertToPageTop(new("User not found", Alert.ALERT\_TYPE.Danger));

return this.RedirectToAction(nameof(NewAuthLogin));

}

authTockenService.CreateTocken(found.Id, out var AuthTocken, out var AuthHashKey);

Response.Cookies.Append(PasswordSaltString64,

Base64UrlEncoder.Encode(found.AuthPasswordSalt));

Response.Cookies.Append(AuthTockenName, AuthTocken);

Response.Cookies.Append(AuthHashKeyString64,Base64UrlEncoder.Encode(AuthHashKey));

return View();

}

Визуальное представление страницы с вводом пароля является простой формой с одним input’ом и кнопкой отправки запроса. Скрипт, обрабатывающий нажатие был приведен в предыдущей главе. Рассмотрим обработчик Post-запроса NewAuthPassword. Сначала метод убеждается в наличии переданного хеша пароля как такового. Далее он убеждается в корретном формате base64-строки, в виде которой он передается. Далее метод убеждается в наличии переданного токена авторизации, проверяет его с помощью соответствующего сервиса; если токен найден и валиден – загружает из него него Id пользователя, которому он принадлежит; сущность пользователя загружается из базы данных, после чего хеш пароля из базы данных передается для хеширования по уникальному ключу токена авторизации в соответствующий сервис, после чего полученный хеш сверяется с хешем, переданным клиентом. Если хеши совпадают – с помощью соответствующего сервиса создаётся сессия; её идентификатор передается клиенту, остальные же куки удаляются – программный код 37.

Перед окончанием данной главы добавлю, что авторизация в момент выполнения каждого запроса производится классом AMyController, упоминаемым, но не описываемым в данной работе. Все контроллеры проекта являются наследниками данного абстрактного класса. Данный контроллер обращается к менеджеру сессий, если с запросом был передан токен сессии. Он делает это путем переопределения (override) метода OnActionExecuting. Если токен сессии найден – контроллер подгрузит сущность авторизованного пользователя из базы данных в поле класса AuthedUser – таким образом доступ к авторизованному на данный момент пользователю может быть получен в любой момент обработки запроса.

Программный код 37

[HttpPost] public async Task<IActionResult> NewAuthPassword(string hashedPassword64)

{

if (string.IsNullOrEmpty(hashedPassword64))

return new StatusCodeResult(400);

byte[] PasswordHashedByClient = null!;

try {PasswordHashedByClient = Base64UrlEncoder.DecodeBytes(hashedPassword64);}

catch (System.FormatException) { return new StatusCodeResult(422); }

var AuthTockenId = this.Request.Cookies[AuthTockenName];

if (string.IsNullOrEmpty(AuthTockenId)) {

AddAlertToPageTop(new("Cannot get auth tocken",Alert.ALERT\_COLOR.Red));

return this.RedirectToAction(nameof(NewAuthLogin));

}

if (!this.authTockenService.TryGetTocken(AuthTockenId, out var Tocken)) {

AddAlertToPageTop(new("Auth tocken outdated",Alert.ALERT\_COLOR.Red));

return this.RedirectToAction(nameof(NewAuthLogin));

}

var FoundUser = await this.context.Users.FindAsync(Tocken.UserId);

if (FoundUser is null){

base.AddAlertToPageTop(new("User not found", Alert.ALERT\_TYPE.Danger));

return this.RedirectToAction(nameof(NewAuthLogin));

}

if (!this.authTockenService.ConfirmPasswordAndInvalidateTocken(AuthTockenId,

PasswordHashedByClient, FoundUser.AuthHashedPassword, out var dummy)) {

base.AddAlertToPageTop(new("Wrong password", Alert.ALERT\_COLOR.Red));

return this.RedirectToAction(nameof(NewAuthPassword));

}

SM.CreateSession(FoundUser.Id, out var CreatedSessId);

Response.Cookies.Append(SessionManager.SessionIdCoockieName, CreatedSessId);

Response.Cookies.Delete(PasswordSaltString64);

Response.Cookies.Delete(AuthTockenName);

Response.Cookies.Delete(AuthHashKeyString64);

return this.RedirectToAction("Show", "User", new { id = FoundUser.Id });

}

**СЕРВИС ЗАЩИТЫ ОТ СПАМА**

В данной главе описывается создание сервиса зашиты от «слишком частых запросов». Сервис имеет простую реализацию и следит за тем, чтобы с одного IP-адреса не поступало слишком много запросов за единицу времени, устанавливаемую настройками. Сервис имеет следующие настройки в файле appsettings.json: ControlledPeriod\_secs – период времени, контролируемый сервисом; AllowedNumOfRequestsPerPeriod – число запросов от адреса, допустимое за контролируемый период; UnbanInterval\_secs – интервал обновления сервиса, раз в который снимается блокировка с ранее заблокированных адресов. Сервис работает со словарем, ключем в котором служит IP-адрес, значением – число, указывающее на число запросов за контролируемый период. В сервисе имеются методы CountRequest – «учитывает» запрос с переданного адреса; IPAddressIsBanned – проверяет адрес на предмет блокировки. Сервис является наследником BackgroundService. В переопредленном методе ExecuteAsync происходит уменьшение значений для каждого и ключа на число, вычисляемое как произведения допустимых в период запросов на отношение периода обновления к контролируемому периоду. Так, если за контролиуемый периож, равный, например, минуте, допускается 10 запросов, а период обновления составляет 20 секунд, то каждые 20 секунд по каждому адресу число запросов за контролиуемый период будет снижаться на 3.33, что в конкретной реализации будет округлено до 4 – программный код 38.

Программный код 38

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

while (!stoppingToken.IsCancellationRequested)

{

await Task.Delay(UnbanInterval\_secs \* 1000);

int DecreaseInDelay =(int)System.Math.Ceiling(

(float)this.MaxRequestsPerPeriod \* ((float)this.UnbanInterval\_secs / (float)this.Period\_secs));

foreach (var k in this.RequestsByIpLastTime.Keys)

{

this.RequestsByIpLastTime[k] =

this.RequestsByIpLastTime[k] - DecreaseInDelay;

if (this.RequestsByIpLastTime[k] < 0)

this.RequestsByIpLastTime.Remove(k);

}

}

}

С целью введения сервиса в работу следует внедрить зависимость от него в конструкторе AMyController и добавить в переопределение его метода OnActionExecuting строчки с проверкой IP на предмет бана – программный код 39.

Программный код 39

if (this.RC.IPAddressIsBanned(context)) context.Result = new StatusCodeResult(429);

**ФУНКЦИОНАЛ ПУБЛИКАЦИИ ДОКУМЕНТОВ**

В этой главе описывается выполнение формальной задачи работы – добавление функционала публикации документов. На текущем этапе разработки для этого уже не требуется создания новых функциональных частей проекта.

Сначала создадим простую страницу с двумя input’ами, служащими для ввода названия и текста документа, и кнопкой отправки документа на сервер. В связи с простотой код не демонстрируется. В рамках этой работы я не буду реализовывать заложенный в модель данных функционал добавления в документ жирного/наклонного текста, а также текста со ссылками, ибо они требуют исключительно написания клиентской части, на что не делается упор в данной работе.

Далее следует создать модель постинга документа с возможностью автоматической валидации. Для этого следует добавить соответствующие аннотации к данным, указывающие, каким условиям должно соответствовать их содержание – программный код 40. Валидирующими аннотациями являются Required – означающая, что поле должно иметь значение и не может быть пустым; MinLength и MaxLength – указывающие минимальную и максимальную длину соответствующих полей; RegularExpression – позволяющая ввести обязательное соответствие данному регулярному выражению.

Программный код 40

public class DocumentCreation{

private const string PleaseCorrectStr = "Please enter a correct value";

[DataType(DataType.Text)]

[Required(ErrorMessage = PleaseCorrectStr)]

[MinLength(3), MaxLength(200)]

[RegularExpression(@".\*\S{3}.\*")]

[Display(Name = "Title")]

public string Title { get; set; } = null!;

[DataType(DataType.Text)]

[Required(ErrorMessage = PleaseCorrectStr)]

[MinLength(200), MaxLength(300000)]

[Display(Name = "Document text")]

public string Text { get; set; } = null!;

}

Далее создадим соответствующее действие post-запроса Create в контроллере документов. Действие проверяет наличие авторизованного пользователя, проверяет валидность модели переданного документа встроенный методом Controller.ModelState.IsValid; преобразует тип DocumentCreation в Document, добавляя ему автора и время создания; асинхронно добавляет полученный объект в базу данных; и перенаправляет пользователя на страницу показа (действие контроллера Show) вновь созданного пользователем документа.

**ТЕСТИРОВАНИЕ. ПРИМЕНЕНИЕ MOQ**

«*Современные приложения активно избавляются от монолитности. Микросервисы, модули, “слои” — основные принципы организации рабочего кода, позволяющие достигать независимости, удобства повторного использования, обмена и переноса в системы и т.д. В нашей теме имеет ключевое значение именно слоистая структура и внедрение зависимостей*»,   
- Денис Орлов, habr.com.

В данной главе на простом примере будет рассмотрено тестирование с применением Mock-объектов. Проблема тестирования состоит в необходимости создавать окружение объекта для его функционирования. Так, например, те или иные сервисы могут внять определенные зависимости через конструктор. Если в самописном коде глубина внедрения может быть не выше однозначного числа, то с учетом используемых библиотек она может выходить за десяток.

Рассмотрим антиспам сервис (RequestsFromIpCounterService), создание которого было описано ранее. Он внедряет зависимость от самописного сервиса провайдера настроек (RequestsFromIpCounterServiceSettingsProvider). Такое внедрение обеспечивает разделение обязанностей и уровней абстракции (провайдер настроек работает с содержанием файлов, в то время как сам сервис работает уже исключительно с C#-переменными). В свою очередь провайдер настроек внедряет зависимость от «встроенного» интерфейса IConfiguration, а в данном случае от одного из его конкретных наследников, - экземпляра классов ConfigurationSection, [ConfigurationRoot](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.configuration.configurationroot?view=dotnet-plat-ext-6.0) или [ConfigurationManager](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.configuration.configurationmanager?view=dotnet-plat-ext-6.0). Все из них, кроме последнего, также внедряют зависимости через конструктор. Но и с [ConfigurationManager](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.configuration.configurationmanager?view=dotnet-plat-ext-6.0) всё не так просто. Он, как следует из документации, требует предоставить источники данных в качестве свойства Source, имеющего тип [IList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.ilist-1?view=dotnet-plat-ext-6.0)<[IConfigurationSource](https://docs.microsoft.com/ru-ru/dotnet/api/microsoft.extensions.configuration.iconfigurationsource?view=dotnet-plat-ext-6.0)>. Данную цепочку наследования, внедрения, предоставления можно распутывать и дальше, однако уже понятно, что создание «реального» окружения путем создания экземпляров всех этих объектов может занять невообразимое количество времени и, что даже более важно, нарушает саму концепцию изолированного тестирования (unit-тестирования), ибо затрагивает работу огромного количества сторонних классов.

Для решения проблем, описанных выше, были придуманы т.н. Mock-объекты. Фактически они являются заглушками, позволяющими оборвать цепочку внедрения зависимостей и заменить обращения к внедряемым сервисам предоставлением конкретных литералов. Так, рассматривая упомянутый сервис, провайдер настроек не будет обращаться ни к каким файлам, он просто будет возвращать заданные разработчиком для тестирования значения.

Для создания «заглушек» библиотека Moq создаёт наследника используемого класса, подменяет ему необходимые свойства (например, методом SetupGet), путем их переопределения (англ. override), после чего даёт возможность получить созданный объект. Для обеспечения его работы требуется добавлять к полям, которые предполагается делать подменяемыми, модификатор virtual, иначе Mock не сможет их переопределить.

Для целей тестирования в ASP принято создавать отдельный проект, называемый как тестируемый проект с припиской ‘.Tests’ – например, Website.Tests. Из разнообразных библиотек для тестирования, включая nunit, xunit, mstest я буду использовать xunit, поскольку в dotnet встроена возможность создать проект с данной библиотекой, а также, Адам Фримен в своей книге ‘Pro ASP.NET Core 3’ сообщает, что данная библиотека реализует достаточно удобный интерфейс для проведения тестирования, хотя и замечает, что аналогичного функционала возможно достичь любым из трех представленных средств.

Перейдем к практике. Создаем проект тестирования powershell-командой ‘dotnet new xunit -o Website.Tests --framework netcoreapp3.1’ и добавляем его в решение. В проекте создадим папку ServicesTests. В ней создадим класс RequestsFromIpCounterServiceTests для тестирования антиспам сервиса. Мы будем проверять, соответствует ли поведение сервиса ожидаемому. Функция сервиса проста – блокировать и разблокировать IP-адреса. Создадим 2 метода тестирования с атрибутами [Fact] – WillBan и WillUnban, которые будут проверять способность сервиса производить блокировку и разблокировку соответственно.

Рассмотрим создание метода WillUnban. Создадим в нём Mock-объект сервиса провайдера настроек и зададим ему параметры: число запросов за период – 2, контролируемый период – 3 секунды, интервал проверки возможности разблокировки – 3 секунды. Таким образом, ожидаемое поведение сервиса состоит в том, что при выполнении 3-го запроса в течении 3-х секунд адрес должен быть заблокирован. По прошествии 3-х секунд блокировка должна быть снята. Метод продемонстрирован в программном коде 41.

Программный код 41

[Fact] public async Task WillUnban(){

Mock<RequestsFromIpCounterServiceSettingsProvider> MockAntispamSP = new(null);

MockAntispamSP.SetupGet(sp => sp.AllowedNumOfRequestsPerPeriod).Returns(2);

MockAntispamSP.SetupGet(sp => sp.ControlledPeriod\_secs).Returns(3);

MockAntispamSP.SetupGet(sp => sp.UnbanInterval\_secs).Returns(3);

var serv = new RequestsFromIpCounterService(MockAntispamSP.Object);

await serv.StartAsync(new System.Threading.CancellationToken(false));

for(int i=0;i<3;i++)serv.CountRequest(new IPAddress(this.defaultIPBytes));

Assert.True(serv.IPAddressIsBanned(new IPAddress(this.defaultIPBytes)));

await Task.Delay(3 \* 1000 + 100); //wait 3 secs (and 100 ms for confidence)

Assert.False(serv.IPAddressIsBanned(new IPAddress(this.defaultIPBytes)));

}

**ЗАВЕРШЕНИЕ РАЗРАБОТКИ**

Перед переходом к заключению и выводам я подведу итоги по состоянию проекта на момент написания данной главы. В списке ниже отражены основные разработанные компоненты.

* Модели данных:
  1. Модель документа. Включает текст, дату создания, автора, векторизованный заголовок (для индексации поиска).
  2. Модель пользователя. Включает имя, аватар, дату регистрации, место жительства, электронную почту и т.д.
* Контроллеры:
  1. Контроллер документов. Отвечает за генерацию ответов на запрос поиска/добавления/изменения документов.
  2. Контроллер пользователей. Отвечает за регистрацию/авторизацию пользователей.
  3. Абстрактный контроллер. Привносит во все контроллеры общий функционал авторизации пользователей, защиты от спама и т.д.
* Сервисы:
  1. Сервис случайных данных. Позволяет заполнить базу данных случайными данными разных видов.
  2. Сервис паролей. Реализует хеширование паролей с добавлением т.н. соли на стороне сервера, их добавление в базу данных.
  3. Сервис токенизации авторизации. Реализует хеширование паролей на стороне клиента перед отправкой на сервер.
  4. Сервис поиска документов. Инкапсулирует преобразование поискового запроса в TsVector и его обработку средствами SQL.
  5. Менеджер сессий. Позволяет клиенту производить авторизацию от имени пользователя.
  6. Сервис частых поисковых запросов. Позволяет не обращаться к базе данных при обработке популярных поисковых запросов.
  7. Сервис недавних документов. Позволяет получить недавно добавленные на сайт документы без обращения к базе данных.
  8. Сервис защиты от спама. Блокирует превышение количества запросов с определенного IP-адреса.
* Визуальные страницы и компоненты.
* Проект тестирования с применением xunit и moq.

Проект несет в себе более 2.5 тысяч строк самописного кода, почти 13 тысяч с учетом автогенерации.

**ВЫВОДЫ. ЗАКЛЮЧЕНИЕ**

В качестве вывода можно констатировать, что в ходе разработки проекта, на мой взгляд, была допущена одна грубая ошибка и несколько мелких недоработок.

Серьезной ошибкой является спроектированная структура базы данных. Поскольку это была моя первая проектируемая база данных для относительного большого проекта, я не учел должным образом подход «одной записи и множественных чтений». Дополнительно, что касается структуры, возможно, мне стоит обратить внимание на паттерн проектирования БД ‘Database per service’, что соотносится с применяемой мной микросервисной архитектурой. Также, в свете дальнейшей разработки, я пришел к выводу, что EntityFramework вовсе не является столь удобным. Он освобождает программиста от необходимости использовать язык SQL, однако в остальном несет лишь недостатки – как в плане производительности, так и в плане понимания того, что происходит «под капотом». Возможно, Тим Кори был прав, говоря в одном из своих видео, что «EntityFramework хорош, но для опытных программистов, иначе с ним легко наделать глупостей». Возможно, мне стоит рассмотреть использование паттерна Repository и работать непосредственно с SQL.

Мелкими недоработками можно считать, например, нежелание тратить достаточное количество времени на проектирование в должной степени изолированных микросервисов. В частности, сервис частых поисковых запросов, в идеальном случае, мог бы ничего не знать о сервисе поиска документов, а получать от него информацию во встроенных типах, в моём же случае он получает непосредственно экземпляры сервиса поиска документов (см. блок-схему 4). Такие решения приводят к превращению микросервисной архитектуры в «микросервисный монолит».

В остальном – проект не закончен. Однако можно заметить, что его развитие может идти бесконечно, как и многих IT-продуктов, поэтому это нельзя считать проблемой.

**ИСТОЧНИКИ**

1. Макконнелл С. Совершенный код: руководство по разработке программного обеспечения. – 2-е изд. – «БХВ», 2019, - 867 с.
2. Адам Ф. ASP.NET Core 3 с примерами на C# для профессионалов: руководство по разработке программного обеспечения. – 8-е изд. – «Диалектика», 2019, - 1184 с.
3. Сайт о программировании [Электронный ресурс] - metanit.com (дата обращения: 30.05.2022)
4. Тим Кори [Электронный ресурс] - [youtube.com/user/IAmTimCorey](http://www.youtube.com/user/IAmTimCorey) (дата обращения: 30.05.2022)
5. Техническая документация Майкрософт [Электронный ресурс] - docs.microsoft.com/en-us/ (дата обращения: 30.05.2022)
6. Сайт об IT-индустрии [Электронный ресурс] – habr.com (дата обращения: 30.05.2022)
7. Техническая документация NpgSQL [Электронный ресурс] – npgsql.org (дата обращения: 30.05.2022)
8. Техническая документация PostgreSQL [Электронный ресурс] - postgrespro.ru (дата обращения: 30.05.2022)
9. Сайт о программировании [Электронный ресурс] - github.com (дата обращения: 30.05.2022)
10. Computerphile [Электронный ресурс] - youtube.com/user/Computerphile (дата обращения: 30.05.2022)
11. Сайт о программировании [Электронный ресурс] - stackoverflow.com (дата обращения: 30.05.2022)
12. Википедия [Электронный ресурс] - ru.wikipedia.org (дата обращения: 30.05.2022)