Лабораторная работа 2
по конструированию Интернет-приложений
Разработка веб-API. Реализация функциональности CRUD

20-03-2019

Цель лабораторной работы: научиться создавать контроллер веб-API с методами получения, создания, обновления и удаления данных.

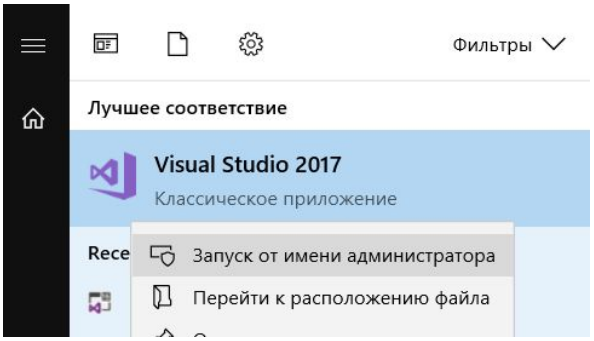
Задания:

Сумма баллов ПК-1 состоит из трех пунктов (по 20 марта):

1. Создать модели своих сущностей и контекст базы данных по выбранной темы. Создать свой контроллер или контроллеры веб-API с CRUD. (Лабораторная работа 2).
2. Создать БД в СУБД и реализовать работу со связанными данными (Лабораторная работа 3). Заполнить тестовыми данными.
3. Продемонстрировать результаты и загрузить отчеты по лабораторным работам 2 и 3 на гугл-диск.

Процесс проверялся на Windows 10 с Visual Studio Community 2017 версии 15.9.7 и .NET Core 2.2.104.

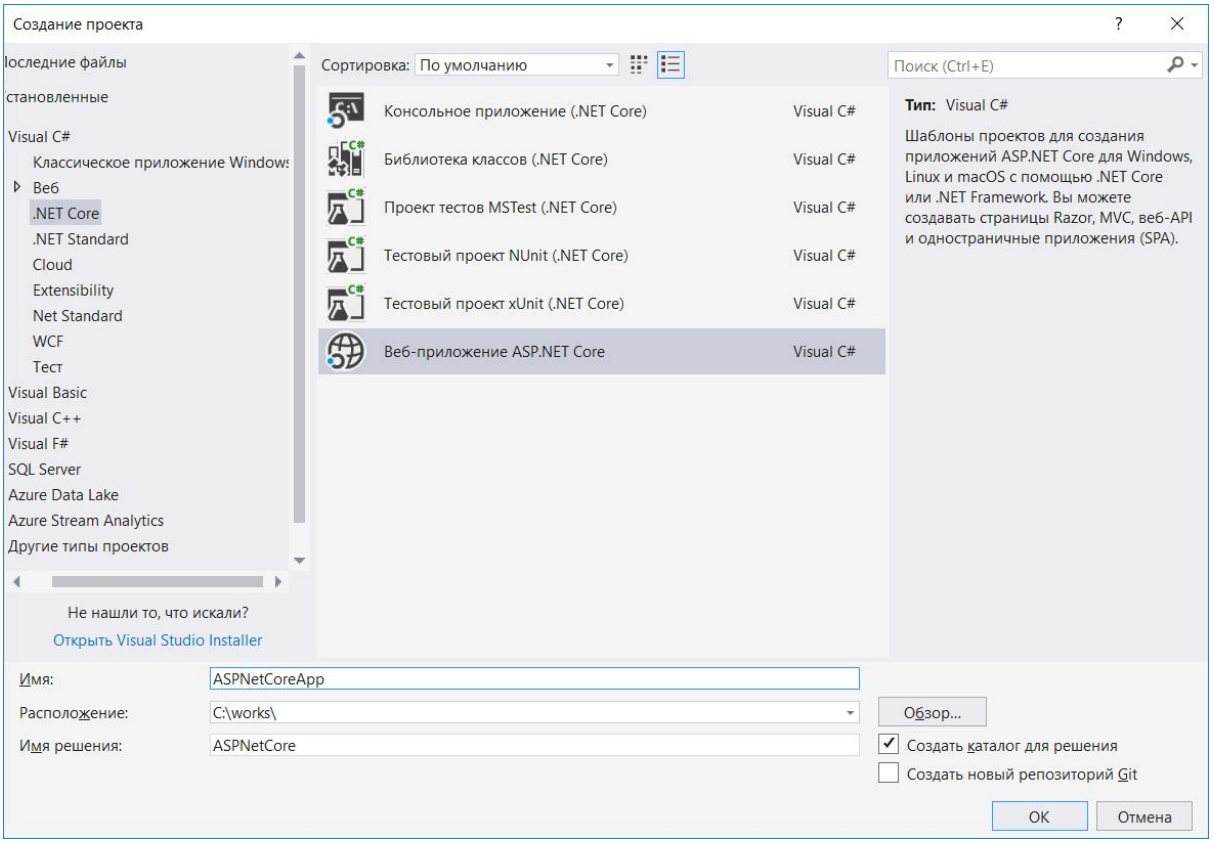
Для успешного прохождения процесса создания веб-приложения следует Visual Studio запускать с правами администратора.



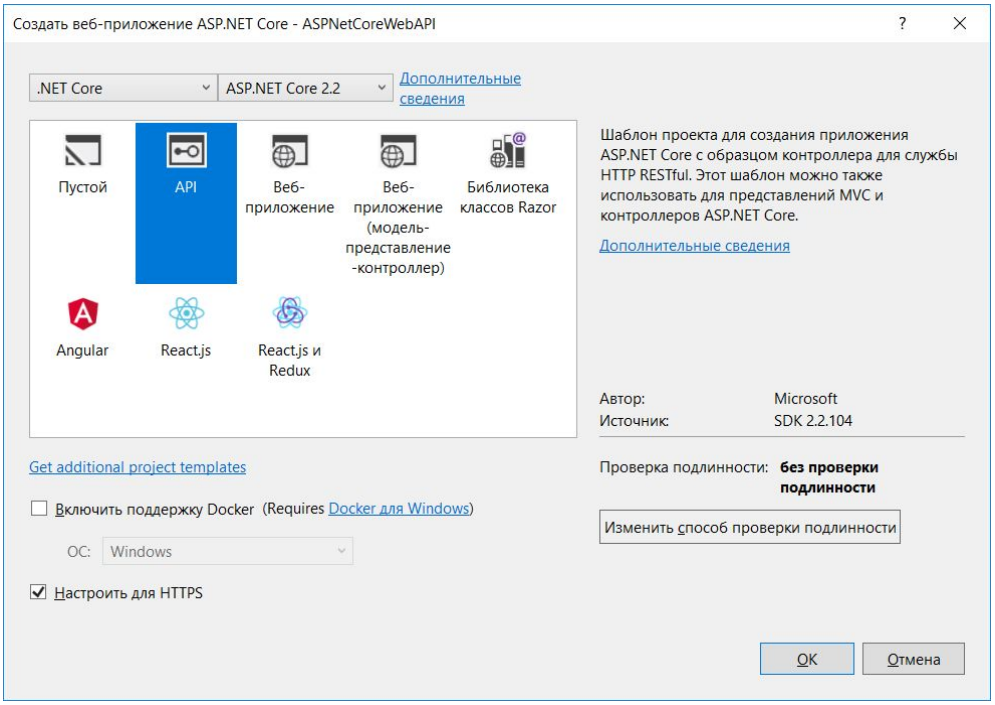
create, read, update, and delete (CRUD).

1. Создать проект веб-API

Указать имя решения ASPNetCore и имя проекта ASPNetCoreWebAPI



Используя ASP.NET Core 2.2 создать шаблон API



2. Создать модели данных сущностей

В соответствии с определенными сущностями выбранной темы создать модели данных.

Ведение блогов выбрано предметной областью, на которой будет рассматриваться процесс разработки.

1) Создать сущность блог
Создать класс блога Blog.cs в папке Models.

```
using System.Collections.Generic;

namespace ASPNetCoreApp.Models
{
    public partial class Blog
    {
        public Blog()
        {
            Post = new HashSet<Post>();
        }

        public int BlogId { get; set; }
        public string Url { get; set; }

        public virtual ICollection<Post> Post { get; set; }
    }
}
```

1) Создать сущности поста
Создать класс поста в Post.cs в папке Models.

```
namespace ASPNetCoreApp.Models
{
    public class Post
    {
        public int PostId { get; set; }
        public int BlogId { get; set; }
        public string Content { get; set; }
        public string Title { get; set; }

        public virtual Blog Blog { get; set; }
    }
}
```

Пост представляет собой публикацию и связан с блогом. Для демонстрации принципов и простоты реализации, связь с блогом нежесткая, т.е. пост может существовать сам по себе.

Для свойств можно задавать атрибуты:
[Key]
[Required]
[Column(TypeName ="nvarchar(100)")]

3. Создать контекст базы данных

Контекст базы данных — это основной класс, который координирует функциональные возможности Entity Framework для заданной модели данных о обеспечивает взаимодействие с хранилищем данных.

Создать класс поста в BloggingContext.cs в папке Models.

```
using Microsoft.EntityFrameworkCore;

namespace ASPNetCoreApp.Models
{
   public partial class BloggingContext : DbContext
   {
       #region Constructor
       public BloggingContext(DbContextOptions<BloggingContext>
options)
           : base(options)
       { }
       #endregion

       public virtual DbSet<Blog> Blog { get; set; }
       public virtual DbSet<Post> Post { get; set; }

       protected override void OnModelCreating(ModelBuilder
modelBuilder)
       {
           modelBuilder.Entity<Blog>(entity =>
           {
               entity.Property(e => e.Url).IsRequired();
           });

           modelBuilder.Entity<Post>(entity =>
           {
               entity.HasOne(d => d.Blog)
                   .WithMany(p => p.Post)
                   .HasForeignKey(d => d.BlogId);
           });
       }
   }
}
```

4. Выполнить регистрацию контекста базы данных

Контекст базы данных регистрируется с помощью контейнера внедрения зависимостей, который позволяет сделать объекты слабосвязанными. Выполнить регистрацию контекста базы данных в startup.cs.

```
using ASPNetCoreApp.Models;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace ASPNetCoreApp
{
   public class Startup
   {
       public Startup(IConfiguration configuration)
       {
           Configuration = configuration;
       }

       public IConfiguration Configuration { get; }

       public void ConfigureServices(IServiceCollection services)
       {
           services.AddDbContext<BloggingContext>(opt =>
           opt.UseInMemoryDatabase("Blogging"));

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_
2_2);
       }

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

           app.UseHttpsRedirection();
           app.UseMvc();
       }
   }
}
```

В приведенном выше коде срока с UseInMemoryDatabase определяет поставщика базы данных и позволяет использовать Entity Framework Core с выполняющейся в памяти базой данных. Такой подход используется при тестировании.

5. Создать контроллер блогов

5.1. Добавить новый контроллер блогов

Контроллер представляет собой центральное звено, которому маршрутизация передает данные запроса и который занимается обработкой ответом.

В папке Controllers создать BlogsController.cs и заменить код на следующий:

```
using ASPNetCoreApp.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace ASPNetCoreApp.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BlogsController : ControllerBase
    {
        private readonly BloggingContext _context;

        public BlogsController(BloggingContext context)
        {
            _context = context;
            if (_context.Blog.Count() == 0)
            {
                _context.Blog.Add(new Blog { Url = "http:\\blogs.net"
});
                _context.SaveChanges();
            }
        }
    }
}
```

В приведенном выше фрагменте кода создается класс контроллера API. В случае если блогов еще нет, то создается блог с Url = "http:\\blogs.net".
Контроллер является классом, а методы контроллера называются методы действия. Шаблон маршрута api/[controller] определяет путь, где api сегмент пути, а [controller] переменная. В приведенном выше коде, контроллер доступен по относительному пути /api/blogs.

[Route("api/[controller]")] маршрутизация для контроллера

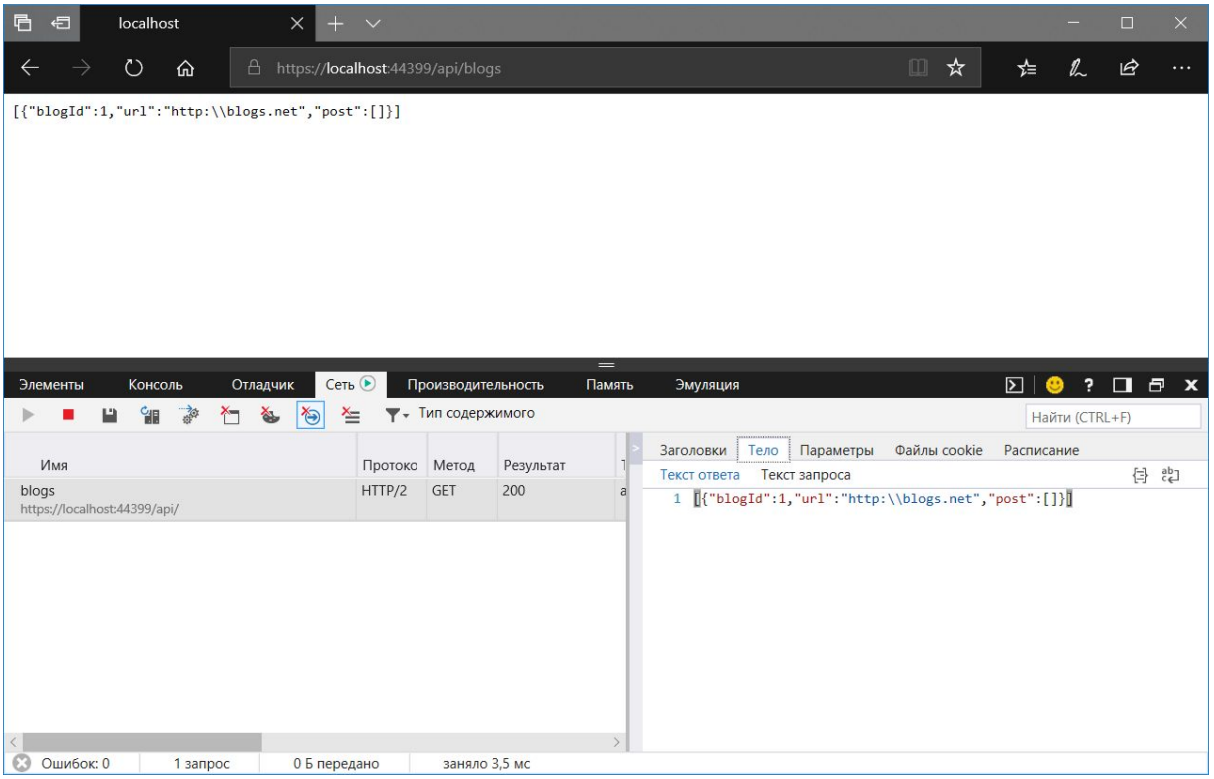
В launchSettings.json настроить вызов созданного контроллера

```
"profiles": {
   "IIS Express": {
     "commandName": "IISExpress",
     "launchBrowser": true,
     "launchUrl": "api/blogs",
...
```

5.2. Добавить методы действий получения элементов блогов

```
      [HttpGet]
       public IEnumerable<Blog> GetAll()
       {
           return _context.Blog;
       }

       [HttpGet("{id}")]
       public async Task<IActionResult> GetBlog([FromRoute] int id)
       {
           if (!ModelState.IsValid)
           {
               return BadRequest(ModelState);
           }

           var blog = await _context.Blog.SingleOrDefaultAsync(m =>
m.BlogId == id);

           if (blog == null)
           {
               return NotFound();
           }

           return Ok(blog);
       }
```

Атрибут HttpGet обрабатывает http-запросы GET. Атрибут HttpGet("{id}") также обрабатывает запросы GET, но кроме этого, обрабатывает переменную {id}.



Если необходимо, чтобы свойства возвращались в том виде (регистре), как они объявлены в модели, то нужно добавить следующий код в startup.cs

```
using Newtonsoft.Json.Serialization;

public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc()
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2)
        .AddJsonOptions(options => {
            var resolver = options.SerializerSettings.ContractResolver;
            if (resolver != null)
                (resolver as DefaultContractResolver).NamingStrategy =
null;
            });
```

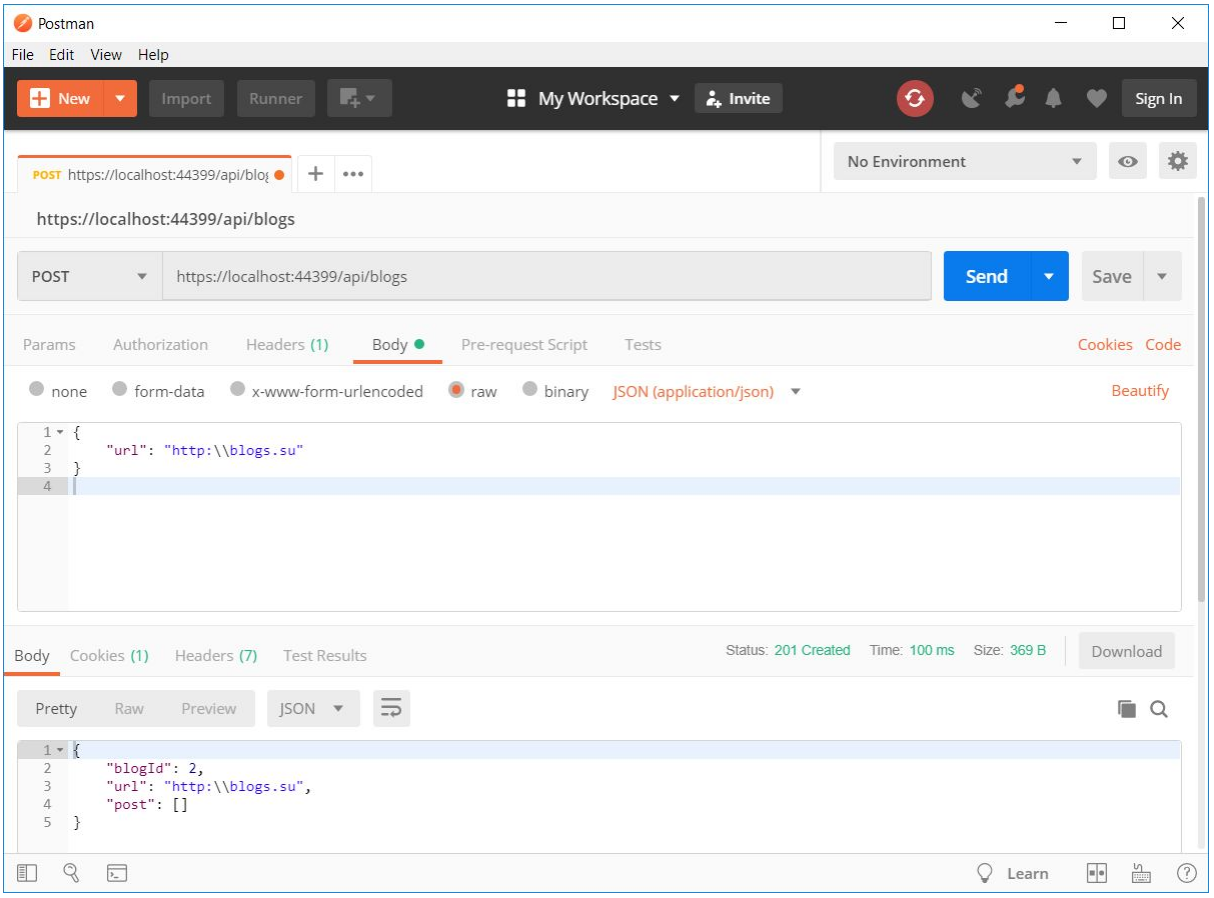
5.3. Добавить метод действия создания блога

```
        [HttpPost]
        public async Task<IActionResult> Create([FromBody] Blog blog)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            _context.Blog.Add(blog);
            await _context.SaveChangesAsync();

            return CreatedAtAction("GetBlog", new { id = blog.BlogId },
blog);
        }
```

Атрибут HttpPost обрабатывает http-запросы POST. Ключевым отличием метода POST от GET является то, что информация передается в теле запроса.

BadRequest один из типов возвращаемых значений, он формирует код состояния HTTP 400, если происходит сбой проверки модели.

Проверить работу метода в Postman:
В заголовке запроса указать тип json.



HTTP-код ответа 201 Created говорит о успешном создании данных на стороне сервера. В теле ответа возвращена информации о созданном объекте. Также в заголовке ответа получен новый путь ресурса URI.

Помимо атрибута метода [HttpPost] можно добавить атрибут маршрутизации например,

```
[HttGet]
[Route("api/blogs/{blogId:int}/posts/{postId:int}")]
public async Task<IActionResult> Get([FromBody] Blog blog, int blogId,
int postId){
```

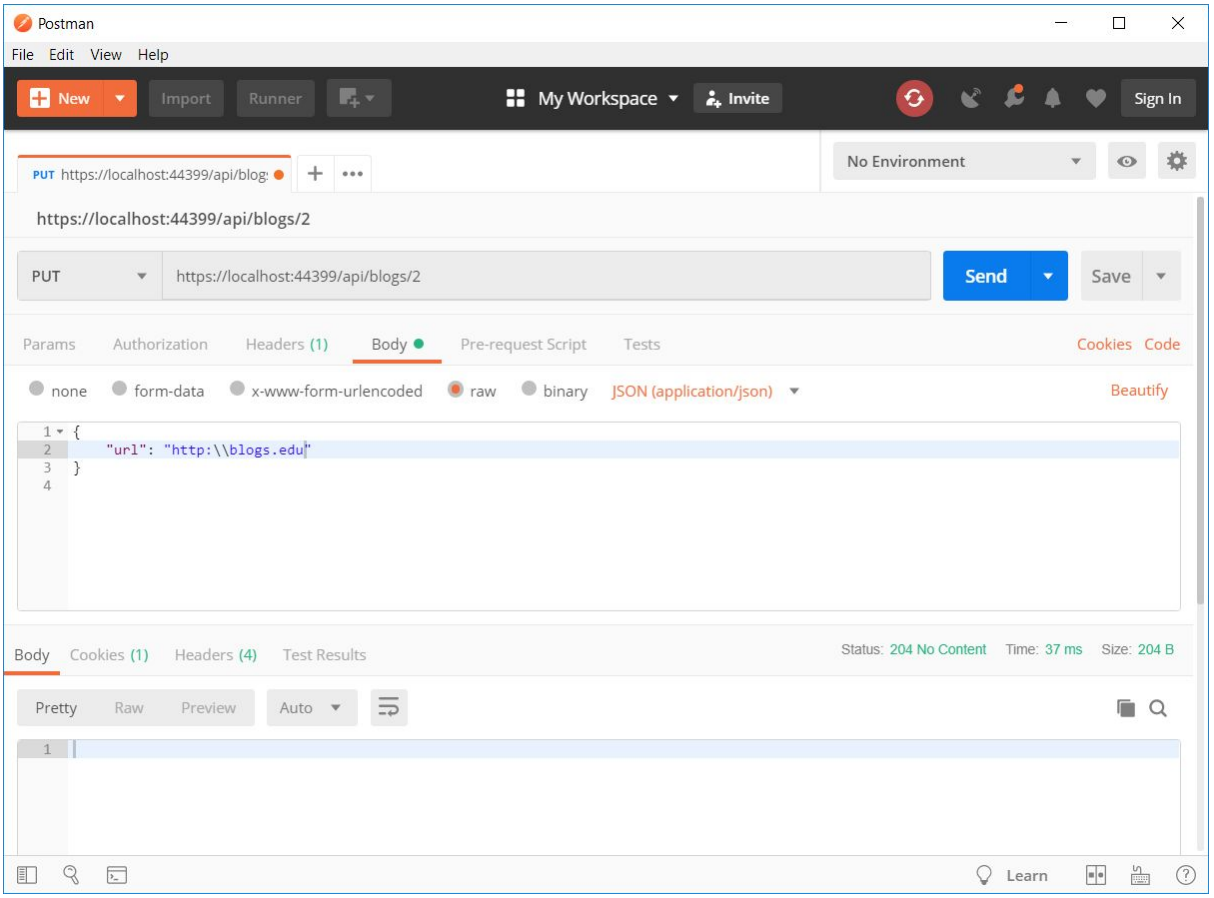
более подробно в документации
Маршрутизация в ASP.NET Core | Microsoft Docs
https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing?view=aspnetcore-2.2

5.4. Добавить метод действия обновления блога

```
        [HttpPut("{id}")]
        public async Task<IActionResult> Update([FromRoute] int id,
[FromBody] Blog blog)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            var item = _context.Blog.Find(id);
            if (item == null)
            {
                return NotFound();
            }
            item.Url = blog.Url;
            _context.Blog.Update(item);
            await _context.SaveChangesAsync();
            return NoContent();
        }
```

Обновление объекта отличается от создания тем, что сначала ищется существующий объект по идентификатору и только потом происходит обновление. При успешном обновлении объекта сервер возвращает код 204 No Content.

Проверка в Postman



5.5. Добавить метод действия удаления блога

```
[HttpDelete("{id}")]
public async Task<IActionResult> Delete([FromRoute] int id)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    var item = _context.Blog.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    _context.Blog.Remove(item);
    await _context.SaveChangesAsync();
    return NoContent();
}
```

Источники:

Учебник. Создание веб-API с помощью MVC ASP.NET Core
https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/first-web-api?view=aspnetcore-2.2&tabs=visual-studio

Связи — EF Core | Microsoft Docs
https://docs.microsoft.com/ru-ru/ef/core/modeling/relationships

Маршрутизация в ASP.NET Core | Microsoft Docs
https://docs.microsoft.com/ru-ru/aspnet/core/fundamentals/routing?view=aspnetcore-2.2