4 Задание 4. Разработка REST Веб-сервиса на базе ASP.NET Core

Цель: В этом задании мы познакомимся с основами разработки веб-API с помощью ASP.NET Core. Мы познакомимся с принципами создания проекта веб-API, методами добавления классов модели и контекста базы данных, формирование шаблонов контроллера с использованием методов CRUD, настройкой маршрутизации, URL-путей и возвращаемых значений, вызова веб-API с помощью Postman.

В итоге вы получите веб-API, позволяющий работать с элементами списка дел, хранимыми в базе данных.

4.1 Критерии оценивания

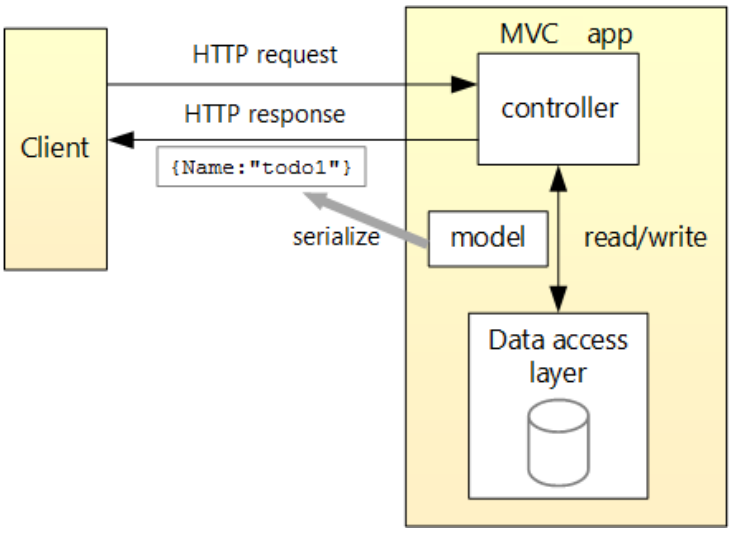
№	Задача
1.	Запустить базовое тестовое приложение WeatherForecast и показать его работу.
2.	Разработать и показать работу веб-API, позволяющего работать с элементами списка дел, хранимыми в базе данных
3.	Продемонстрировать работу объекта передачи данных (DTO), скрывающего информацию основного объекта
4.	Разработать и продемонстрировать работу клиентского приложения для созданного REST-сервиса

4.2 Методические рекомендации

1.1.1 Обзор и подготовка

В этом руководстве создается следующий API-интерфейс

API	Описание	Текст запроса	Текст ответа
GET /api/TodoItems	Получение всех элементов задач	Отсутствуют	Массив элементов задач
GET /api/TodoItems/{id}	Получение объекта по идентификатору	Отсутствуют	Элемент задачи
POST /api/TodoItems	Добавление нового элемента	Элемент задачи	Элемент задачи
PUT /api/TodoItems/{id}	Обновление существующего элемента	Элемент задачи	Отсутствуют
DELETE /api/TodoItems/{id}	Удаление элемента	Отсутствуют	Отсутствуют



Настройка системы будет дана на примере Visual Studio Code. Установить IDE можно по этой ссылке: https://code.visualstudio.com/

Настройка на базе Visual Studio 2019 реализуется аналогичным образом. Более подробно инструкцию по настройке среды разработки на базе Visual Studio 2019 можно найти на странице: https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio

Перед выполнением практической работы, необходимо установить:

1. C# для Visual Studio Code (последняя версия) https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp
2. Пакет SDK для .NET Core 3.1 или более поздней версии: https://dotnet.microsoft.com/download/dotnet-core/3.1

1.1.2 Создание веб-проекта

1. Откройте Интегрированный терминал. Для этого можно нажать клавиши Ctrl+`
2. Смените каталог (cd) на папку, в которой будет содержаться папка проекта

3. Выполните следующие команды:

```
dotnet new webapi -o TodoApi
cd TodoApi
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.InMemory
code -r ../TodoApi
```

Выполнение данных команд:

1. создает новый проект типа веб-API, после чего
2. вы переходите в папку с новым проектом.
3. Вы добавляете пакеты NuGet (распространяемые пакеты для платформы .NET), которые позволят обеспечить работу с базой данных в памяти
4. Открываете проект в новом окне Visual Studio Code

1.1.3 Тестирование API

Шаблон нового проекта типа веб-API не пустой. В нем уже реализуется простейший веб-API, который возвращает случайный прогноз погоды на некоторое число дней вперед.

Код этого метода можно найти в файле

`TodoApi\Controllers\WeatherForecastController.cs`

Рассмотрим его содержимое. Метод, отвечающий за генерацию «прогноза погоды» - это метод `Get()` со следующей реализацией:

```
[HttpGet]
public IEnumerable<WeatherForecast> Get()
{
    var rng = new Random();
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateTime.Now.AddDays(index),
        TemperatureC = rng.Next(-20, 55),
        Summary = Summaries[rng.Next(Summaries.Length)]
    })
    .ToArray();
}
```

Аннотация [HttpGet] перед методом означает, что он будет вызван при поступлении запроса GET на веб-сервер. Метод создает массив объектов класса WeatherForecast (который объявлен в файле `TodoApi\WeatherForecast.cs`), на случайное количество дней (от 1 до 5), со случайной температурой и погодными условиями. Возвращаемый массив объектов WeatherForecast будет сериализован в формат JSON, т.к. по умолчанию ASP.NET Core поддерживает именно серисализацию в JSON.

Если вы хотите вернуть объекты в другом формате, вы можете подключить к вашему сервису и другие методы сериализации. Общие инструкции как это сделать доступны тут: *https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-3.1*

Запустим приложение в виртуальном веб-сервере. Нажмите клавиши CTRL+F5, чтобы запустить приложение. В браузере перейдите по следующему URL-адресу:

`https://localhost:5001/WeatherForecast`

Если всё настроено правильно, то возвращаемые данные JSON будут выглядеть примерно так:


```
    {
        "date": "2019-07-16T19:04:05.7257911-06:00",
        "temperatureC": 52,
        "temperatureF": 125,
        "summary": "Mild"
    },
    {
        "date": "2019-07-17T19:04:05.7258461-06:00",
        "temperatureC": 36,
        "temperatureF": 96,
        "summary": "Warm"
    },
    {
        "date": "2019-07-18T19:04:05.7258467-06:00",
        "temperatureC": 39,
        "temperatureF": 102,
        "summary": "Cool"
    },
    {
        "date": "2019-07-19T19:04:05.7258471-06:00",
        "temperatureC": 10,
        "temperatureF": 49,
        "summary": "Bracing"
    },
    {
        "date": "2019-07-20T19:04:05.7258474-06:00",
        "temperatureC": -1,
        "temperatureF": 31,
        "summary": "Chilly"
    }
]
```


1.1.4 Добавление класса модели

Модель прогноза погоды, определенная в файле `TodoApi\WeatherForecast.cs`, плохо подходит к задаче создания планировщика дел. В связи с этим, нам необходимо создать новый класс модели данных.

- Добавьте папку с именем *Models*.
- Добавьте класс TodoItem в папку *Models*, используя следующий код:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Свойство Id выступает в качестве уникального ключа реляционной базы данных.

Классы моделей можно размещать в любом месте проекта, но обычно для этого используется папка *Models*.

Далее нам необходимо добавить класс контекста базы данных. *Контекст базы данных* —это основной класс, который координирует функциональные возможности Entity Framework для модели данных. Этот класс является производным от класса `Microsoft.EntityFrameworkCore.DbContext`. Добавьте класс `TodoContext` в папку *Models*.

Введите следующий код:

```
using Microsoft.EntityFrameworkCore;

namespace TodoApi.Models
{
    public class TodoContext : DbContext
    {
        public TodoContext(DbContextOptions<TodoContext> options)
            : base(options)
        {
        }

        public DbSet<TodoItem> TodoItems { get; set; }
    }
}
```

Свойство `DbSet` представляет собой коллекцию объектов, которая сопоставляется с определенной таблицей в базе данных. При этом по умолчанию название свойства должно соответствовать множественному числу названию модели в соответствии с правилами английского языка. То есть `TodoItem` - название класса модели представляет единственное число, а `TodoItems` - множественное число.

Через параметр options в конструктор контекста данных будут передаваться настройки контекста.

В ASP.NET Core службы (такие как контекст базы данных) должны быть зарегистрированы с помощью контейнера внедрения зависимостей. Контейнер предоставляет службу контроллерам.

Обновите файл *Startup.cs*, используя следующий выделенный код:

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

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

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            // Этот вызов добавляет контекст базы данных
            //а также указывает, что будет использована база данных в памяти
            services.AddDbContext<TodoContext>(opt =>
               opt.UseInMemoryDatabase("TodoList"));
            services.AddControllers();
        }

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

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

Код выделенный желтым добавляет контекст базы данных в контейнер внедрения зависимостей, а также указывает, что контекст базы данных будет использовать базу данных в памяти.

1.1.5 Формирование контроллера на основе шаблона

Создадим контроллер API на основе шаблона TodoItem в виде класса модели и TodoContext в виде класса контекста данных. Для этого необходимо выполнить следующие команды:

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design
dotnet add package Microsoft.EntityFrameworkCore.Design
dotnet tool install --global dotnet-aspnet-codegenerator
dotnet aspnet-codegenerator controller -name TodoItemsController -async -api -m
TodoItem -dc TodoContext -outDir Controllers
```

Если последняя команда не выполняется корректно, вам необходимо внимательно изучить результаты выполнения предыдущей команды и исправить ошибки

Предыдущие команды:

- добавляют пакеты NuGet, необходимые для формирования шаблонов;
- устанавливают подсистему формирования шаблонов (`dotnet-aspnet-codegenerator`);
- формируют шаблоны для `TodoItemsController` с помощью команды `dotnet-aspnet-codegenerator`, которой в качестве параметров передаются имена классов, отвечающих за модель (`TodoItem`) и контекст базы данных (`TodoContext`).

Результатом генерации контроллера является файл
`\TodoApi\Controllers\TodoItemsController.cs`

Давайте изучим этот файл. В соответствии с представленной моделью, нам автоматически были сгенерированы базовые методы на

- получение значения набора TodoItem: `GET: api/TodoItems GetTodoItems()`
- получение отдельного TodoItem, `GET: api/TodoItems/5 GetTodoItem(long id)`
- на редактирование объекта по его индексу `PUT: api/TodoItems/5 PutTodoItem(long id, TodoItem todoItem)`
- на создание нового объекта TodoItem `POST: api/TodoItems PostTodoItem(TodoItem todoItem)`
- на удаление объекта по индексу `DELETE: api/TodoItems/5 DeleteTodoItem(long id)`

Отредактируем метод `PostTodoItem`, заменив «магическую строку» `"GetTodoItem"` на оператор nameof:

```
// POST: api/TodoItems
[HttpPost]
public async Task<ActionResult<TodoItem>> PostTodoItem(TodoItem todoItem)
{
    _context.TodoItems.Add(todoItem);
    await _context.SaveChangesAsync();

    //return CreatedAtAction("GetTodoItem", new { id = todoItem.Id }, todoItem);
    return CreatedAtAction(nameof(GetTodoItem), new { id = todoItem.Id }, todoItem);
 }
```

Этот код является методом `HTTP POST`, обозначенным атрибутом [HttpPost]. Этот метод получает значение элемента списка дел из текста HTTP-запроса.

Метод CreatedAtAction:

- В случае успеха возвращает код состояния HTTP 201. HTTP 201 представляет собой стандартный ответ для метода HTTP POST, создающий ресурс на сервере.
- Добавляет в ответ заголовок Location. Заголовок `Location` указывает URI новой созданной задачи. Дополнительные сведения см. в статье 10.2.2 201 "Создан ресурс".
- Указывает действие `GetTodoItem` для создания URI заголовка `Location`. Ключевое слово `nameof` C# используется для предотвращения жесткого программирования имени действия в вызове `CreatedAtAction`.

Для того, чтобы осуществлять вызовы к этому веб-сервису, нам необходимо понимать, по какому адресу будут располагаться наши ресурсы. Путь к ним формируется по следующим правилам:

1) Адрес и порт сервера, где располагается ресурс (в нашем случае, это https://localhost:5001)
2) Далее путь задается атрибутом `[Route("api/[controller]")]`, который располагается перед описанием контроллера (в нашем случае, в файле `TodoItemsController`), где `[controller]` – это имя контроллера. По соглашению, имя контроллера – это имя класса без суффикса «Controller», то есть в нашем случае имя контроллера будет `TodoItems`
   Таким образом, корневой путь к нашему API будет следующим: https://localhost:5001/api/TodoItems
3) Если в атрибуте есть дополнительный шаблон маршрута (например `[HttpGet("products")])`, то его необходимо добавить к пути (в нашем примере нет таких шаблонов). Дополнительную информацию по шаблонам маршрутов можно найти по ссылке: https://docs.microsoft.com/ru-ru/aspnet/core/mvc/controllers/routing?view=aspnetcore-3.1#attribute-routing-with-httpverb-attributes
4) Если в шаблоне указывается переменная-заполнитель (например [HttpGet("{id}")]), то к запросу необходимо добавить идентификатор через "/". Таким образом, если вы хотите прочитать информацию об объекте №1 из нашей коллекции, необходимо выполнить запрос по следующему URL: https://localhost:5001/api/TodoItems/1

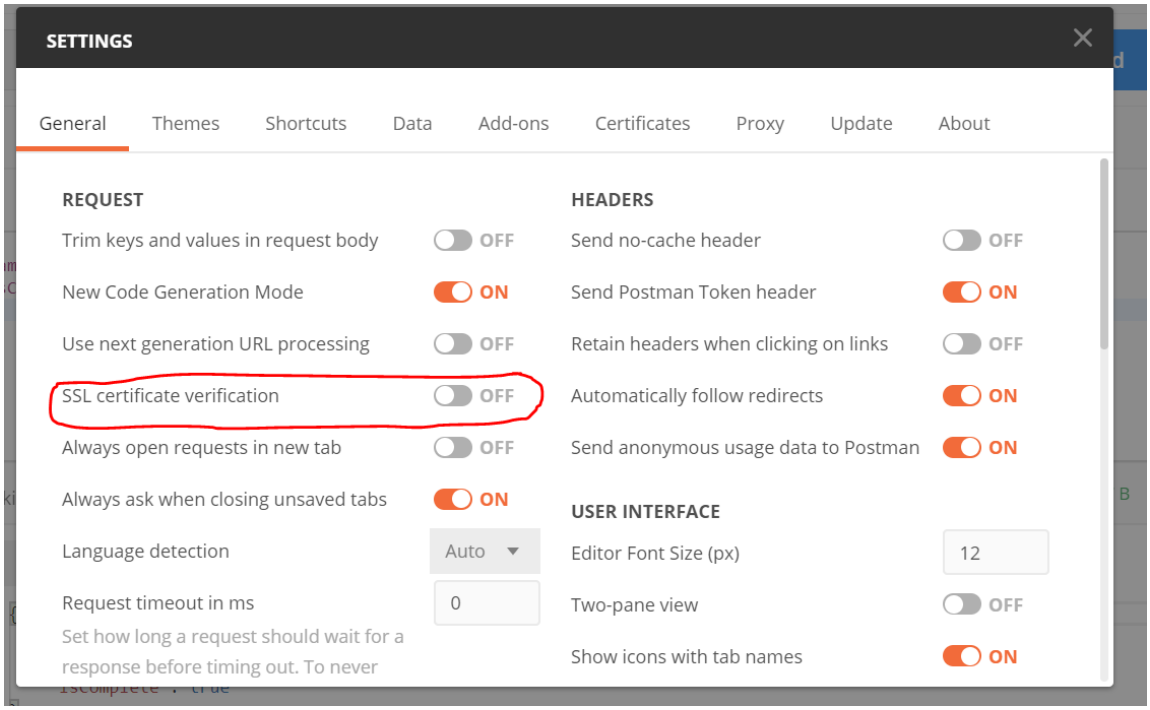
1.1.6 Тестирование REST-сервиса с помощью Postman

Postman — это система, которая позволяет создавать и выполнять запросы к веб-сервисам, документировать и мониторить ваши сервисы. Эта утилита является незаменимой при разработке и тестировании веб-сервисов. Для начала, вам необходимо будет установить Postman с официального сайта: https://www.postman.com/downloads/

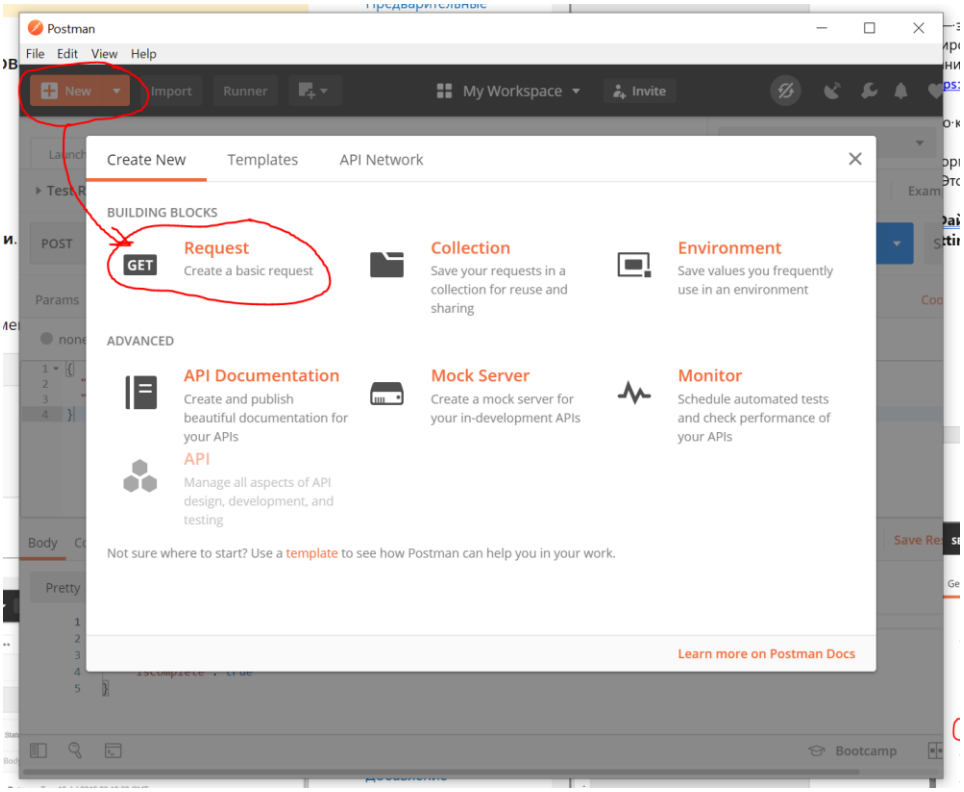
После того как запустите ваш веб-сервис (Ctrl+F5), запустите Postman.

Закрыв форму с регистрацией, необходимо предварительно отключить проверку SSL-сертификата в Postman. Это делается следующим образом:

В меню **Файл** > **Параметры** (вкладка **Общие**), отключите параметр **Проверка SSL-сертификата**. (**File** – **Settings** – **General** – **SSL certificate verification**)



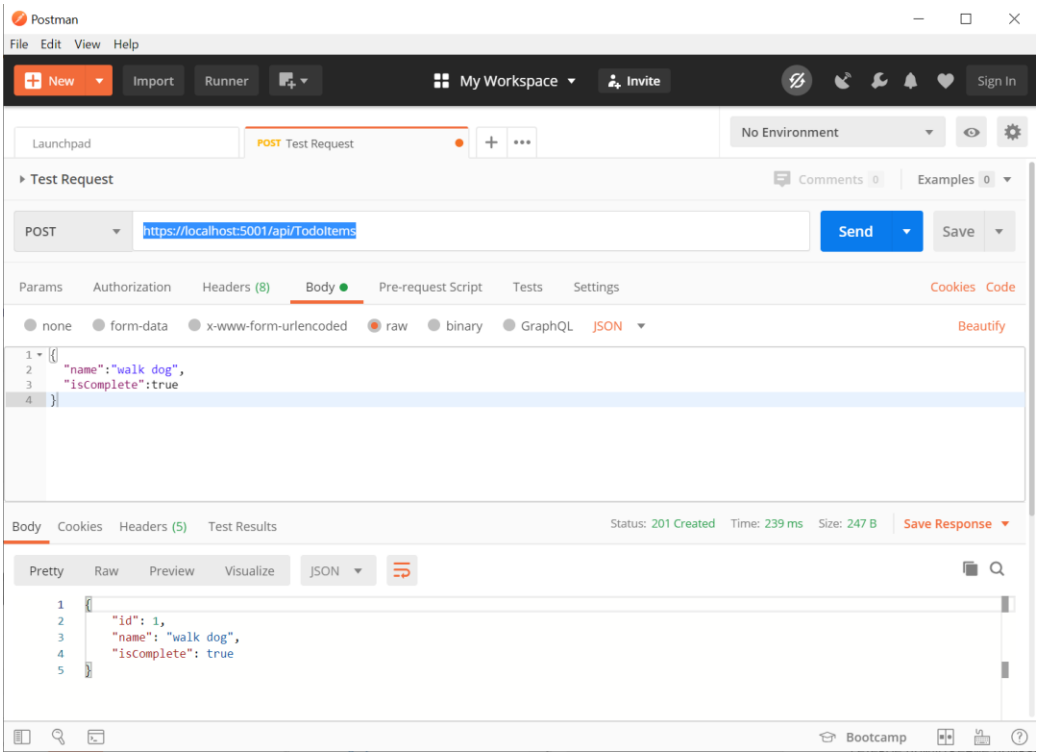
После этого вам необходимо создать новый запрос



Введите название запроса (по вашему вкусу) и создайте новую коллекцию (папку) для сохранения этого запроса. Для начала, мы протестируем метод POST для создания новых объектов в нашей коллекции.

- В поле URL введите адрес нашего API: https://localhost:5001/api/TodoItems
- Установите HTTP-метод POST.
- Откройте вкладку **Тело (Body)**.
- Установите переключатель **без обработки (raw)**.
- Задайте тип **JSON** .
- В теле запроса введите код JSON для элемента списка дел:

```
{
  "name":"walk dog",
  "isComplete":true
}
```



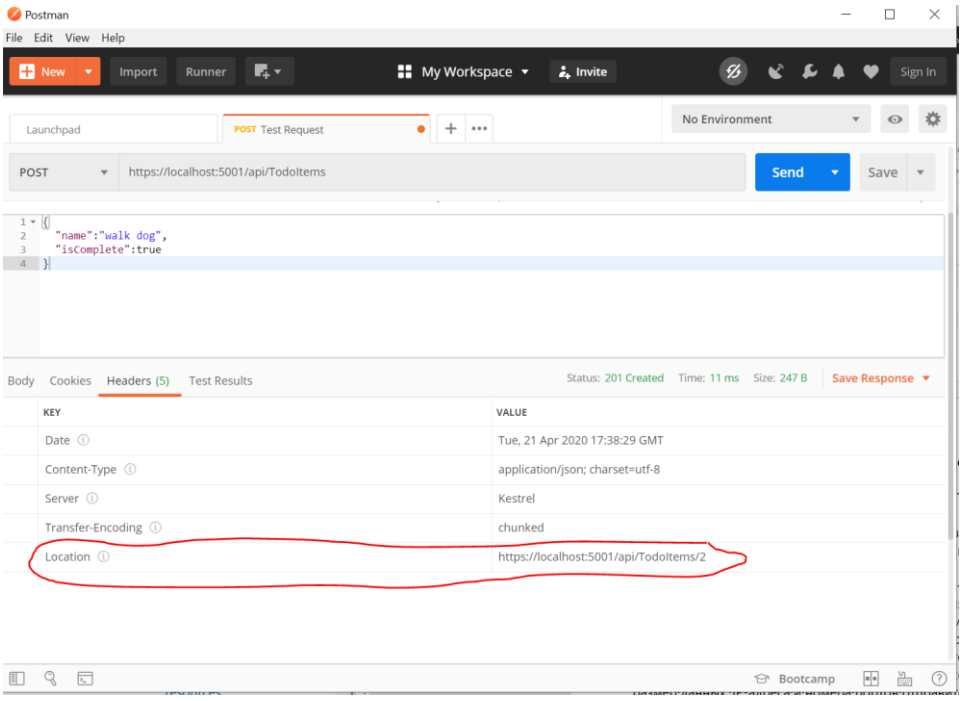
Нажмите кнопку **Отправить (Send)**

При успешном выполнении запроса в поле ответа, должен быть отображен статус 201 Created.

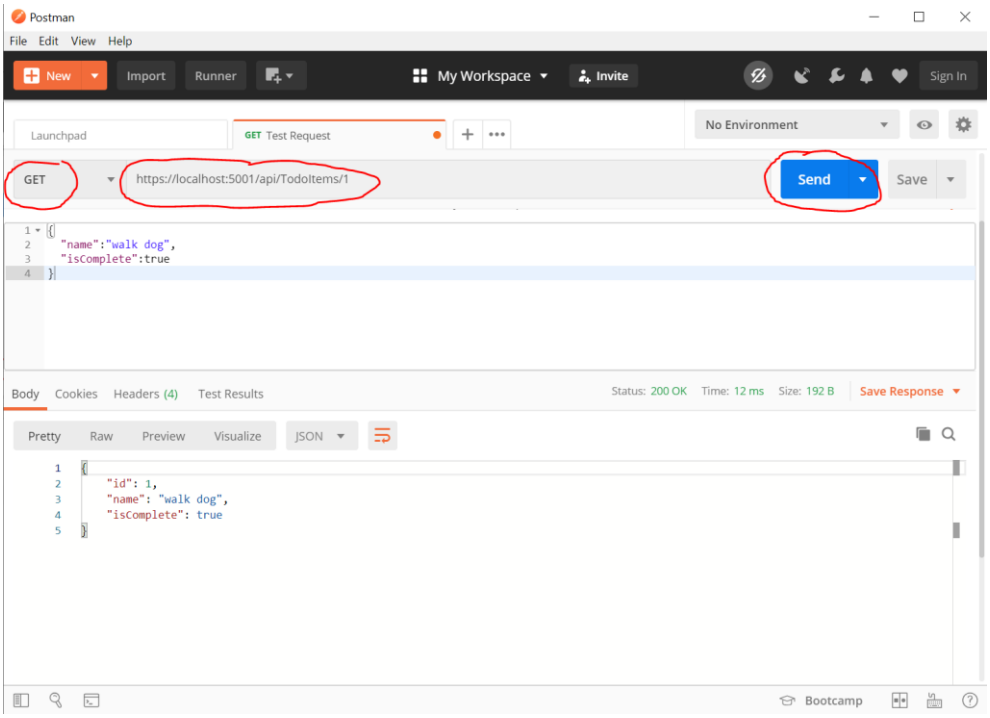
1.1.7 Тестирование метода GET

Теперь мы можем протестировать работу метода `GET`.

Перейдите в раздел заголовков (Headers) ответа на ваш запрос POST. В заголовке «Location» будет содержаться URL вновь-созданного элемента коллекции (например, https://localhost:5001/api/TodoItems/1)



Скопируйте этот URL и вставьте его в поле URL запроса. Измените тип запроса на GET и нажмите SEND.



Выполнив этот запрос, в поле Body ответа мы можем наблюдать информацию о существующем объекте, который был создан предыдущим запросом.

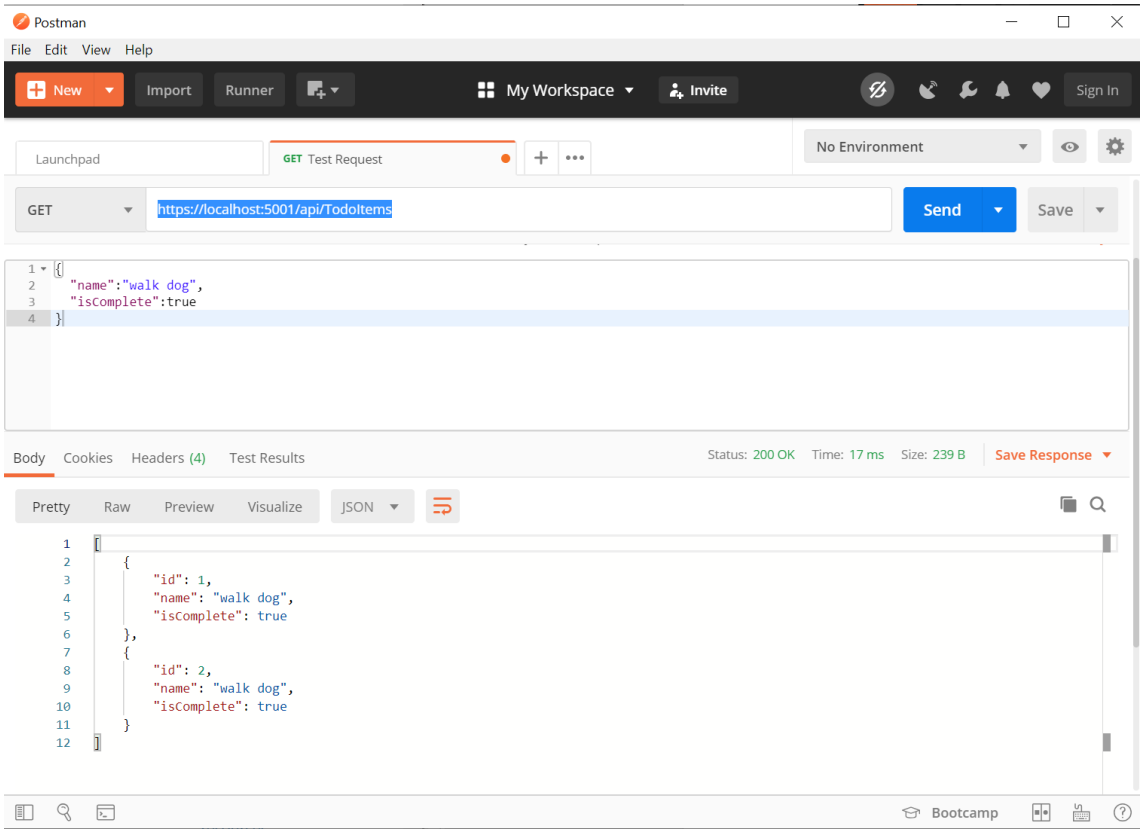
Как вы помните, наше приложение обеспечивает два типа конечных точек на запрос GET:

- получение значения набора TodoItem: `GET: api/TodoItems GetTodoItems()`
- получение отдельного TodoItem, `GET: api/TodoItems/5 GetTodoItem(long id)`

Сейчас мы с вами протестировали второй метод – получение отдельного TodoItem. Для тестирования получения набора элементов, давайте добавим еще один элемент в коллекцию, вызвав еще раз метод POST (можно с теми же параметрами, или же поменять содержимое поля «name» на то, что вы пожелаете). Вызов этого метода добавит еще один элемент в коллекцию.

Теперь выполним метод GET, указав в качестве URL https://localhost:5001/api/TodoItems

При успешном выполнении запроса должен вернуться массив JSON-объектов, содержащий результаты всех наших предыдущих запросов POST.



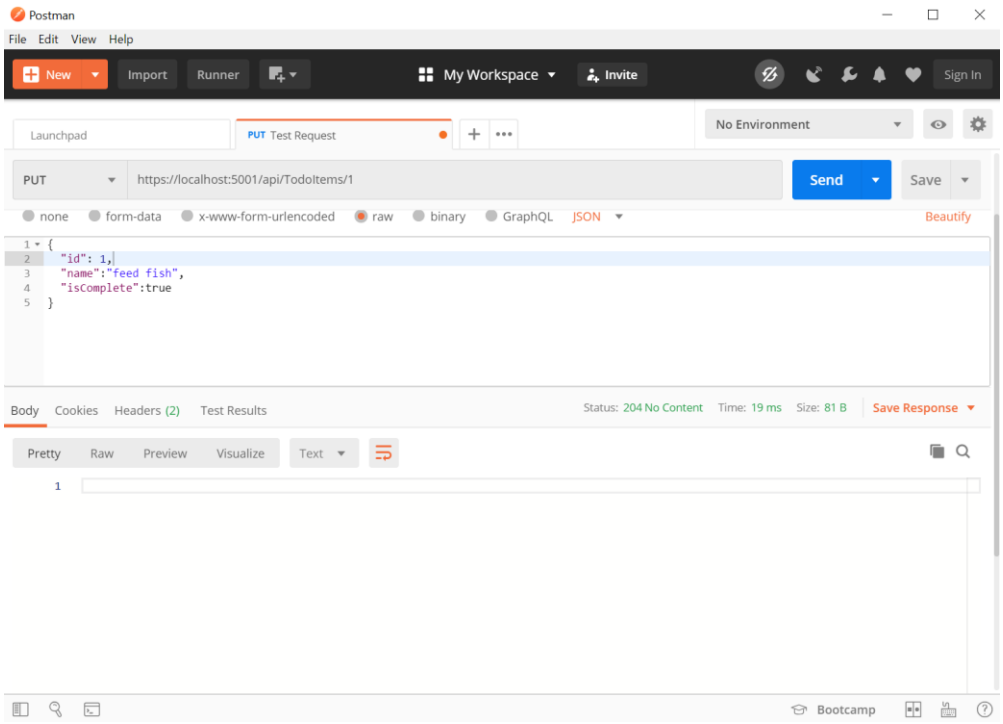
1.1.8 Тестирование метода PUT

Метод PutTodoItem отвечает за замену содержимого существующего на сервере объекта на другой. Ответом на успешное выполнение запроса PUT будет — 204 (No Content). Согласно спецификации HTTP, запрос PUT требует, чтобы клиент отправлял всю обновленную сущность, а не только изменения. Чтобы обеспечить поддержку частичных обновлений, используйте HTTP PATCH.

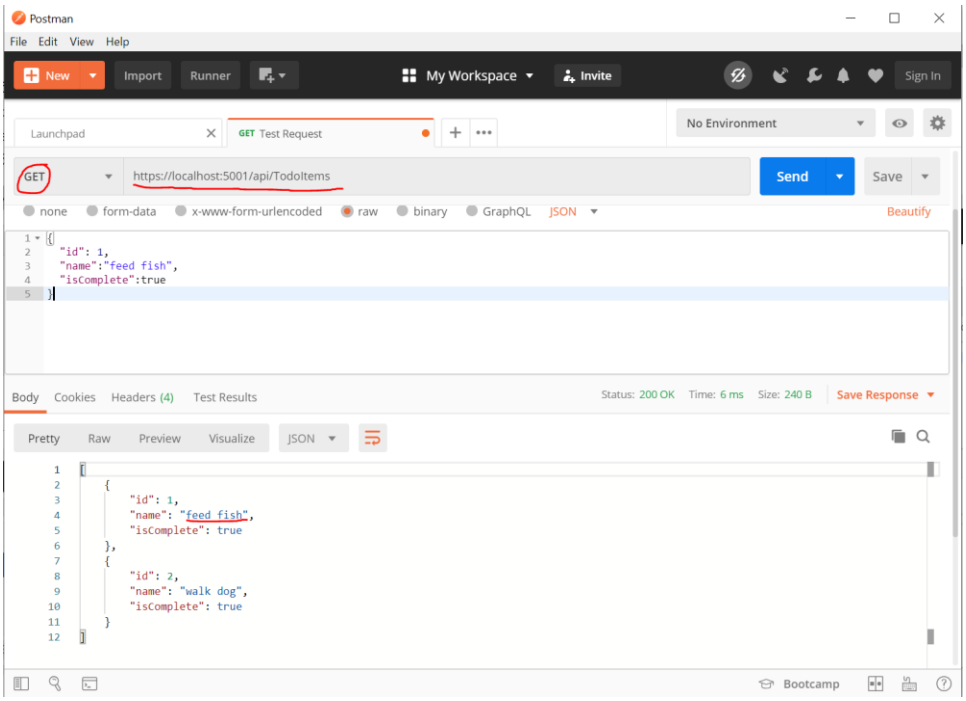
Если возникнет ошибка вызова PutTodoItem, вызовите GET, чтобы в базе данных был хотя бы один элемент.

В этом примере используется база данных в памяти, которая должна быть инициирована при каждом запуске приложения. При выполнении вызова PUT в базе данных уже должен существовать какой-либо элемент. Для этого перед вызовом PUT выполните вызов GET, чтобы убедиться в наличии такого элемента в базе данных.

Обновите элемент списка дел с идентификатором 1 и присвойте ему имя "feed fish":



Если мы просмотрим список всех объектов, то мы заметим, что значение первого элемента действительно изменилось



1.1.9 Тестирование метода DELETE

Аналогично, мы можем проверить работу метода DELETE.

- Укажите метод `DELETE`.
- Укажите URI удаляемого объекта (например, `https://localhost:5001/api/TodoItems/1`).
- Нажмите кнопку **Отправить**.

1.1.10 Создание объектов передачи данных (Data Transfer Object - DTO)

В настоящее время пример приложения предоставляет весь объект `TodoItem`. Рабочие приложения обычно ограничивают вводимые данные и возвращают их с помощью подмножества модели. Это связано с несколькими причинами, и безопасность является основной. Подмножество модели обычно называется объектом передачи данных (Data Transfer Object - DTO), моделью ввода или моделью представления. В этом документе будем использовать аббревиатуру **DTO**.

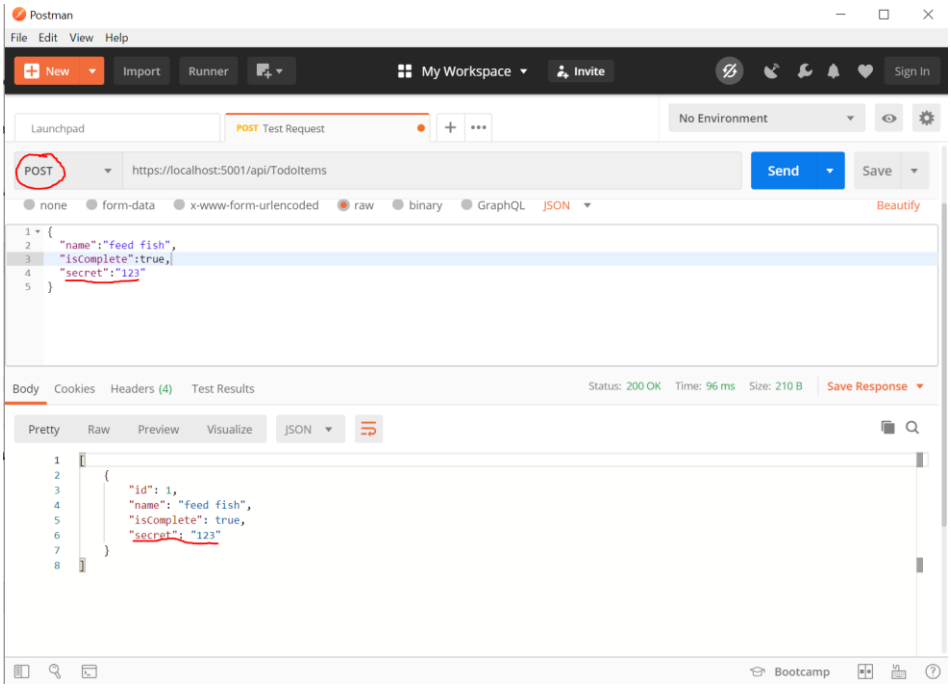
DTO можно использовать для следующего:

- Предотвращение избыточной публикации.
- Скрытие свойств, которые не предназначены для просмотра клиентами.
- Пропуск некоторых свойств, чтобы уменьшить размер полезной нагрузки.
- Сведение графов объектов, содержащих вложенные объекты. Сведенные графы объектов могут быть удобнее для клиентов.

Чтобы продемонстрировать подход с применением DTO, обновите класс `TodoItem`, включив в него поле секрета:

```
public class TodoItem
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
    public string Secret { get; set; }
}
```

Перекомпилируйте приложение и проверьте в Postman, что вы теперь можете создавать объекты, отправляя и получая секретное поле с помощью методов POST и GET



Для предотвращения доступа к полю Secret, создадим DTO-класс, повторяющий поля нашего основного класса но без поля Secret. Создайте в структуре проекта новый класс:

```
public class TodoItemDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public bool IsComplete { get; set; }
}
```

Теперь нам необходимо значительно переработать контроллер, `TodoItemsController` для использования `TodoItemDTO` вместо `TodoItem`.

В конец класса контроллера добавим метод

```
    private static TodoItemDTO ItemToDTO(TodoItem todoItem) =>
            new TodoItemDTO
            {
                Id = todoItem.Id,
                Name = todoItem.Name,
                IsComplete = todoItem.IsComplete
            };
```

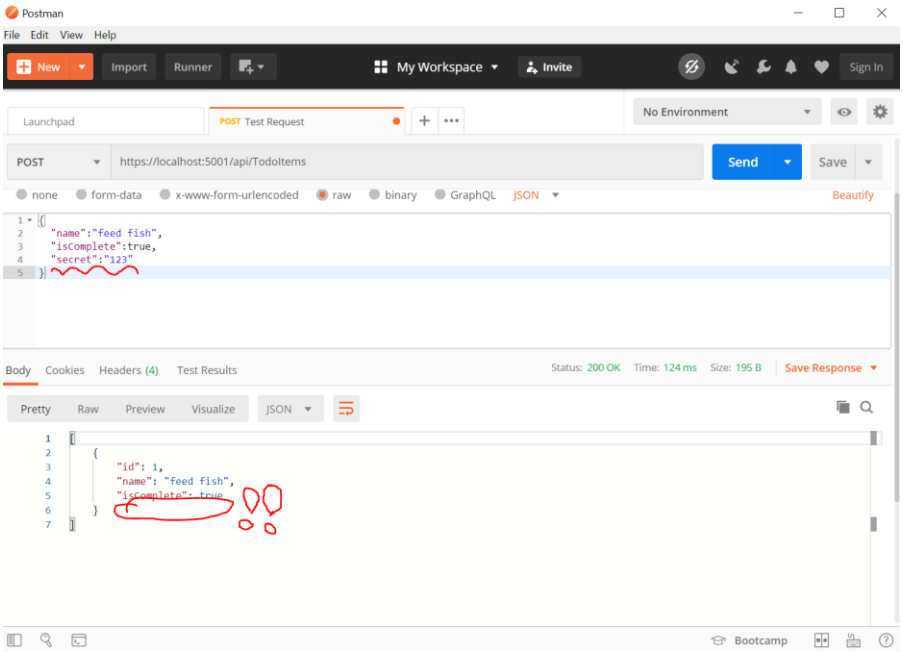
Он позволит преобразовать наш объект в DTO

Далее **заменим** определения и реализацию методов обработки HTTP запросов в классе `TodoItemsController` :

```
    [HttpGet]
    public async Task<ActionResult<IEnumerable<TodoItemDTO>>> GetTodoItems()
    {
        return await _context.TodoItems
            .Select(x => ItemToDTO(x))
            .ToListAsync();
    }

    [HttpGet("{id}")]
    public async Task<ActionResult<TodoItemDTO>> GetTodoItem(long id)
    {
        var todoItem = await _context.TodoItems.FindAsync(id);

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

        return ItemToDTO(todoItem);
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTodoItem(long id, TodoItemDTO todoItemDTO)
    {
        if (id != todoItemDTO.Id)
        {
            return BadRequest();
        }

        var todoItem = await _context.TodoItems.FindAsync(id);
```

```
        if (todoItem == null)
        {
            return NotFound();
        }

        todoItem.Name = todoItemDTO.Name;
        todoItem.IsComplete = todoItemDTO.IsComplete;

        try
        {
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) when (!TodoItemExists(id))
        {
            return NotFound();
        }

        return NoContent();
    }

    [HttpPost]
    public async Task<ActionResult<TodoItemDTO>> CreateTodoItem(TodoItemDTO todoItemDTO)
    {
        var todoItem = new TodoItem
        {
            IsComplete = todoItemDTO.IsComplete,
            Name = todoItemDTO.Name
        };

        _context.TodoItems.Add(todoItem);
        await _context.SaveChangesAsync();

        return CreatedAtAction(
            nameof(GetTodoItem),
            new { id = todoItem.Id },
            ItemToDTO(todoItem));
    }
```

Изменения методов приводят к тому, что обработка запросов теперь меняет только доступные поля в классе `TodoItemDTO`, оставляя поле Secret незадействованным. После компиляции и запуска, вы можете проверить, что клиентские запросы более не имеют доступа на создание и чтение поля Secret.



Источник: https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio-code

4.3 Задание

Разработать клиентское приложение для созданного REST-сервиса. В качестве примера, можно использовать руководство по вызову веб-API с помощью JavaScript, доступное по адресу: https://docs.microsoft.com/ru-ru/aspnet/core/tutorials/web-api-javascript?view=aspnetcore-3.1