



Документирование API. Создание URL-адресов. Преобразование моделей.

Overview

Бывают ситуации, когда вы находитесь в роли разработчика клиентского приложения и вам, например, поставили следующую задачу: “Добавить возможность оповещать о появлении новых запчастей на производстве, для этого используйте следующий API по такому-то url-адресу.

Для того, чтобы работать с данным API - вам может потребоваться следующая информация:

1. Какие эндпоинты (url-адреса) вообще доступны?
2. Для чего каждый из эндпоинтов требуется и что вообще выполняет?
3. Какие данные требуются для корректной работы?
4. и так далее...

В ответ вам могут выдать документацию следующего содержания:

Method	URL	Payload	Result
POST	/api/InventoryItem	CreateInventoryItemCommand (input)	Creates a new inventory item
GET	/api/InventoryItem	InventoryItemListDataCollection (output)	Returns all items
PUT	/api/InventoryItem/{id}	RenameInventoryItemCommand (input)	Renames an item
DELETE	/api/InventoryItem/{id}	DeactivateInventoryItemCommand (input)	De-activates an item
POST	/api/InventoryItem/{id}	RemoveItemsFromInventoryCommand (input)	Removes a number of items from the stock
POST	/api/InventoryItem/{id}	CheckInItemsToInventoryCommand (input)	Adds a number of items to the stock
GET	/api/InventoryItem/{id}	InventoryItemDetail (output)	Returns item details

Достаточно ли информативно для человека, который ещё ни разу не работал с данным API?

Для того, чтобы разработчикам клиентских приложений было проще работать с серверными приложениями (API) - документацию потребуется сделать более информативной.

В нашем случае используется система автодокументирования Swagger, благодаря которому от разработчика API не потребуется много сил. Достаточно будет только создать метод для контроллера API и описать его комментарием.

Всё остальное сформируется самостоятельно в отдельном JSON-файле и при открытии документации, всё необходимое сформируется в виде красивого интерфейса с возможностью поиграться с API при помощи красивых разноцветных кнопок ;)

Описание контроллера API

В качестве примера задокументируем методы для работы с Пользователями Интернет-магазина.

```
UserController.cs  BackendApi-Test  Program.cs  IUserService.cs  IRepositoryWrapper.cs  IRepositoryBase.cs  UserService.cs
BackendApi-Test  BackendApi.Controllers.UserController

1 using Domain.Interfaces;
2 using Domain.Models;
3 using Microsoft.AspNetCore.Mvc;
4
5 namespace BackendApi.Controllers
6 {
7     [Route(template: "api/[controller]")]
8     [ApiController]
9     public class UserController : ControllerBase
10    {
11        private IUserService _userService;
12        public UserController(IUserService userService)
13        {
14            _userService = userService;
15        }
16
17        [HttpGet]
18        public async Task<IActionResult> GetAll()
19        {
20            return Ok(await _userService.GetAll());
21        }
22
23        [HttpGet(template: "{id}")]
24        public async Task<IActionResult> GetById(int id)
25        {
26            return Ok(await _userService.GetById(id));
27        }
28
29        [HttpPost]
```

Какие атрибуты существуют для описания API в стиле REST (Что такое REST)?

Для чего данные методы вообще предназначены?

- Атрибут `HttpGet` используется для получения данных. Запросы с использованием этого метода могут только извлекать данные.
- Атрибут `HttpPost` используется для отправки сущностей к определённому ресурсу. Обычно используется для создания записи.
- Атрибут `HttpPut` в основном применяется для изменения существующей записи.
- Атрибут `HttpDelete` используется для удаления записи.

Также существуют и другие Http-методы (Смотреть тут)

Внедрение автоматической генерации

Описание API

Перейдите в файл Program.cs (или файл Startup.cs) и приведите метод расширения AddSwaggerGen к следующему виду.

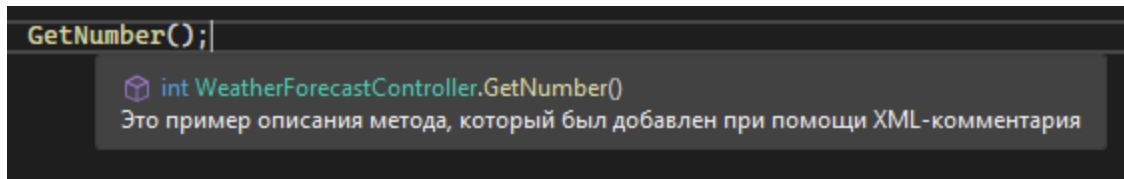
```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Интернет-магазин API",
        Description = "Краткое описание вашего API",
        Contact = new OpenApiContact
        {
            Name = "Пример контакта",
            Url = new Uri("https://example.com/contact")
        },
        License = new OpenApiLicense
        {
            Name = "Пример лицензии",
            Url = new Uri("https://example.com/license")
        }
    });
});
```

Теперь, при запуске проекта, на странице вы будете видеть информацию о вашем API (Наименование, описание, ссылки и т.д)



Описание методов API

Для того, чтобы получить более подробную информацию о методе или ресурсе (например, при наведении на метод), используются XML-комментарии.



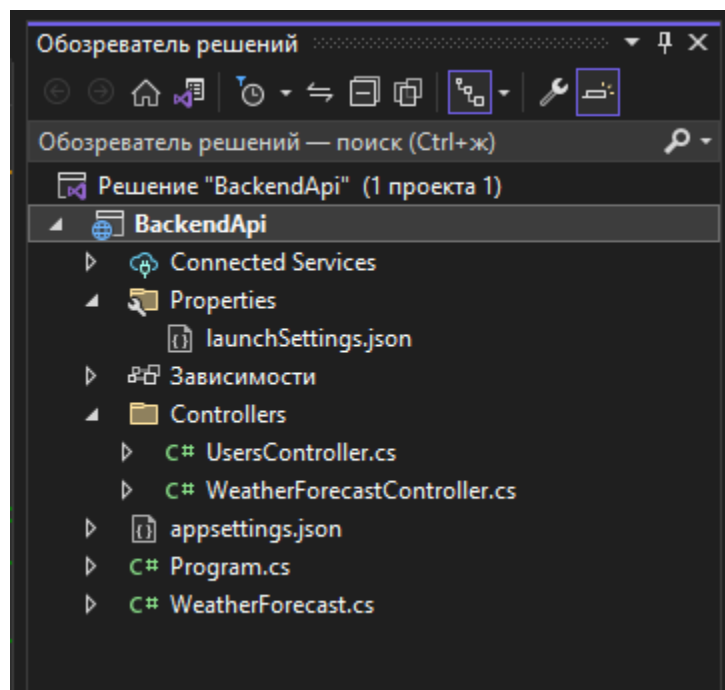
Данные комментарии также используются для описания эндпоинтов (url-адресов) к API.

Например, опишите в таком же стиле один из методов вашего контроллера. В моём случае это метод Add в контроллере UserController:

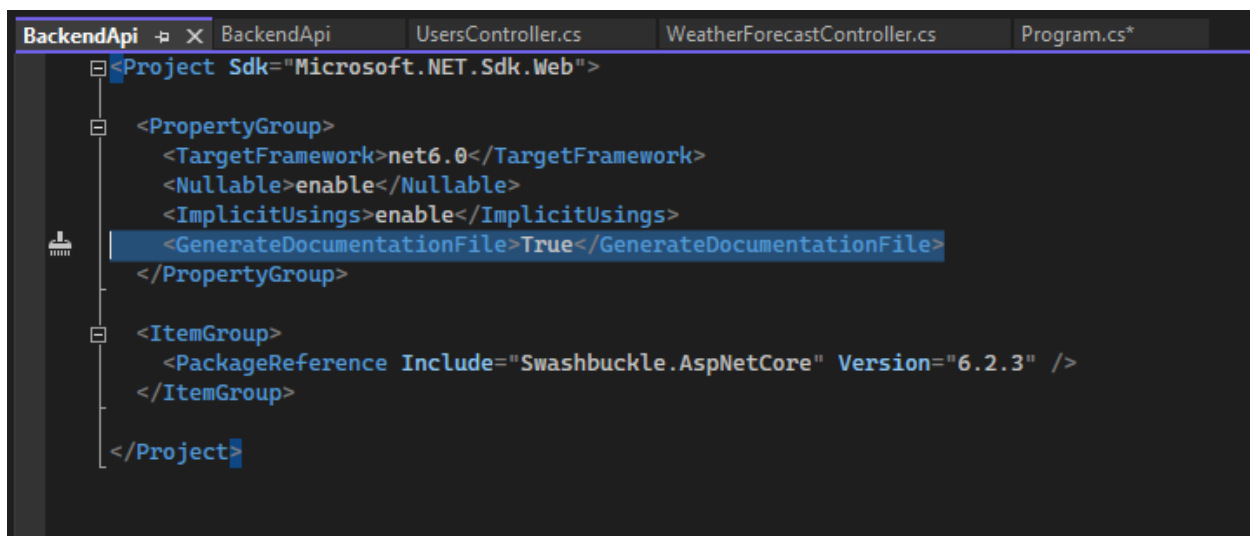
```
/// <summary>
/// Создание нового пользователя
/// </summary>
/// <remarks>
/// Пример запроса:
///
///     POST /Todo
///     {
///         "login" : "A4Tech Bloody B188",
///         "password" : "!Pa$$word123@",
///         "firstname" : "Иван",
///         "lastname" : "Иванов",
///         "middlename" : "Иванович"
///     }
///
/// </remarks>
/// <param name="model">Пользователь</param>
/// <returns></returns>

// POST api/<UsersController>
[HttpPost]
public async Task<IActionResult> Add(User user)
{
    await _userService.Create(user);
    return Ok();
}
```

Для того, чтобы описание метода было видно на странице - нажмите дважды левой кнопкой мыши по вашему проекту.



Укажите свойство проекта `GenerateDocumentationFile` со значением `True`.



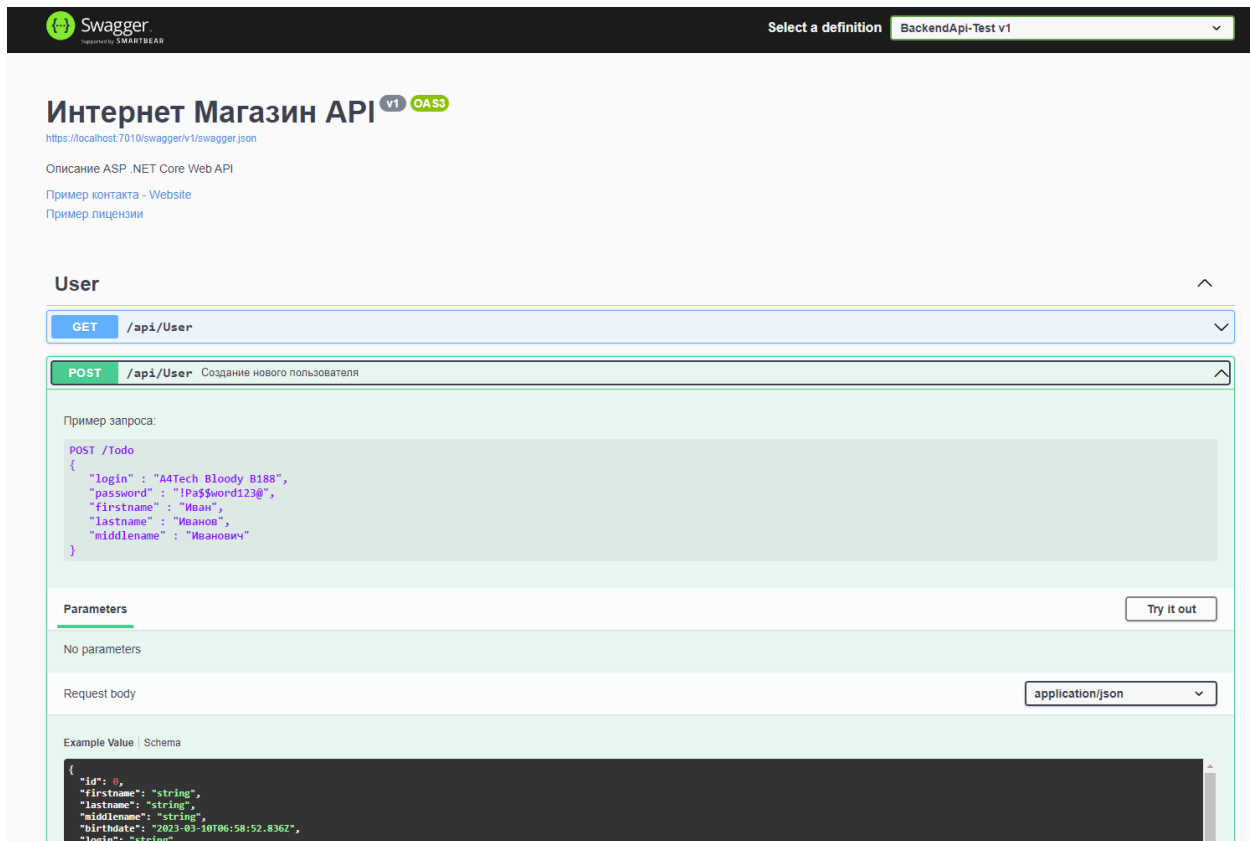
Вернитесь к файлу `Program.cs` и добавьте в метод расширения `AddSwaggerGen` следующий код:

```
var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));
```

В результате, метод расширения должен быть описан следующим образом:

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc(name: "v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "Интернет Магазин API",
        Description = "Описание ASP .NET Core Web API",
        Contact = new OpenApiContact
        {
            Name = "Пример контакта",
            Url = new Uri("https://example.com/contact")
        },
        License = new OpenApiLicense
        {
            Name = "Пример лицензии",
            Url = new Uri("https://example.com/license")
        },
    },
    // using System.Reflection;
    var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";
    options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));
});
```

Запустите проект и проверьте, что методу для регистрации было добавлено описание:



Более подробно про Swagger можно прочитать ([ТУТ](#))

Проблема при добавлении или изменении записи. Применение маппинга.

Когда вы раскрываете метод в котором в качестве параметра используется модель БД - вы могли заметить, что пример для запроса (Example Value) формируется не совсем корректный. Формируется модель со всеми возможными данными.

Parameters

Try it out

No parameters

Request body

application/json

Example Value | Schema

```
{
  "id": 0,
  "firstname": "string",
  "lastname": "string",
  "middlename": "string",
  "birthdate": "2023-03-10T06:58:52.836Z",
  "login": "string",
  "email": "string",
  "password": "string",
  "userOrders": [
    {
      "userId": 0,
      "orderId": 0,
      "order": {
        "id": 0,
        "userId": 0,
        "time": {
          "ticks": 0,
          "days": 0,
          "hours": 0,
          "milliseconds": 0,
          "minutes": 0,
          "seconds": 0
        },
        "date": "2023-03-10T06:58:52.836Z",
        "status": "string",
        "userOrders": [
          "string"
        ]
      }
    }
  ]
}
```

Связано это с тем, что мы используем модель у которой есть свойство, которое принимает внутри себя описание некоторой другой модели и та, в свою очередь, может иметь внутри себя описание других моделей и так далее.

```
Domain
Domain.Models.User

1 namespace Domain.Models
2 {
3     Ссылка: 16
4     public class User
5     {
6         Ссылка: 1
7         public User()
8         {
9             UserOrders = new HashSet<UserOrder>();
10        }
11
12        Ссылка: 2
13        public int Id { get; set; }
14        Ссылка: 2
15        public string Firstname { get; set; } = null!;
16        Ссылка: 2
17        public string Lastname { get; set; } = null!;
18        Ссылка: 2
19        public string Middlename { get; set; } = null!;
20        Ссылка: 2
21        public DateTime Birthdate { get; set; }
22        Ссылка: 2
23        public string Login { get; set; } = null!;
24        Ссылка: 2
25        public string Email { get; set; } = null!;
26        Ссылка: 2
27        public string Password { get; set; } = null!;
28
29        Ссылка: 2
30        public virtual ICollection<UserOrder> UserOrders { get; set; }
31    }
32 }
```

Рассмотрим на примере контроллера UserController

Обычно, для запросов или ответов API специально формируются модели (Контракты), которые описывают внутри себя требуемые данные для корректной работы определенного метода API.

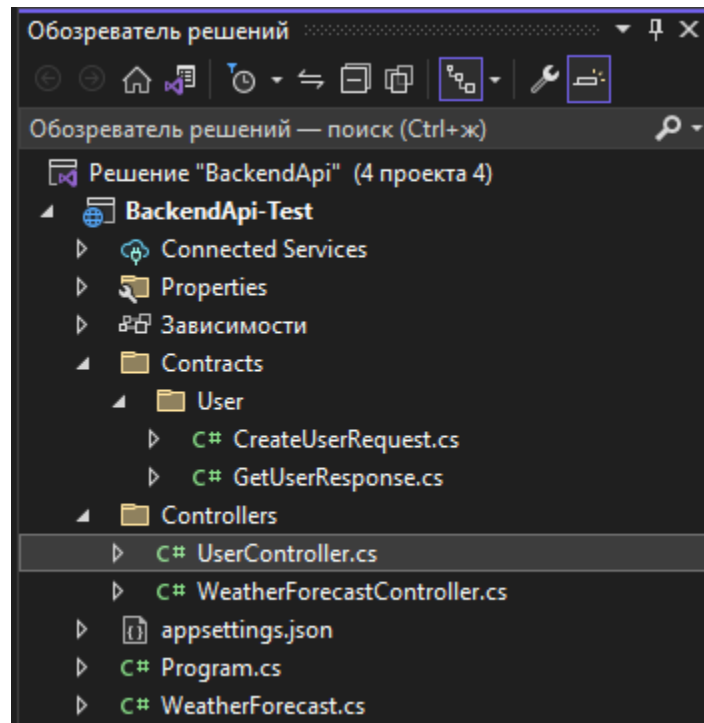
Таким образом, мы избавляемся от лишних данных для запросов или ответов, которые не будут играть никакой роли при работе. А также, можем требовать дополнительно от клиента некоторые другие данные или наоборот - возвращать более информативный ответ, чем сама запись из таблицы.

Как раз, для решения проблемы слишком информативного Example Value, потребуется сформировать отдельные контракты для Запросов и Ответов, в которых будут описаны только те свойства класса, которые мы хотим получить или отправить.

Приступим к практике, на примере ресурса User (Контроллер UserController)

Создайте в проекте API папку Contracts, а внутри него папку с названием ресурса. Внутри папки ресурса создайте класс в следующем виде:

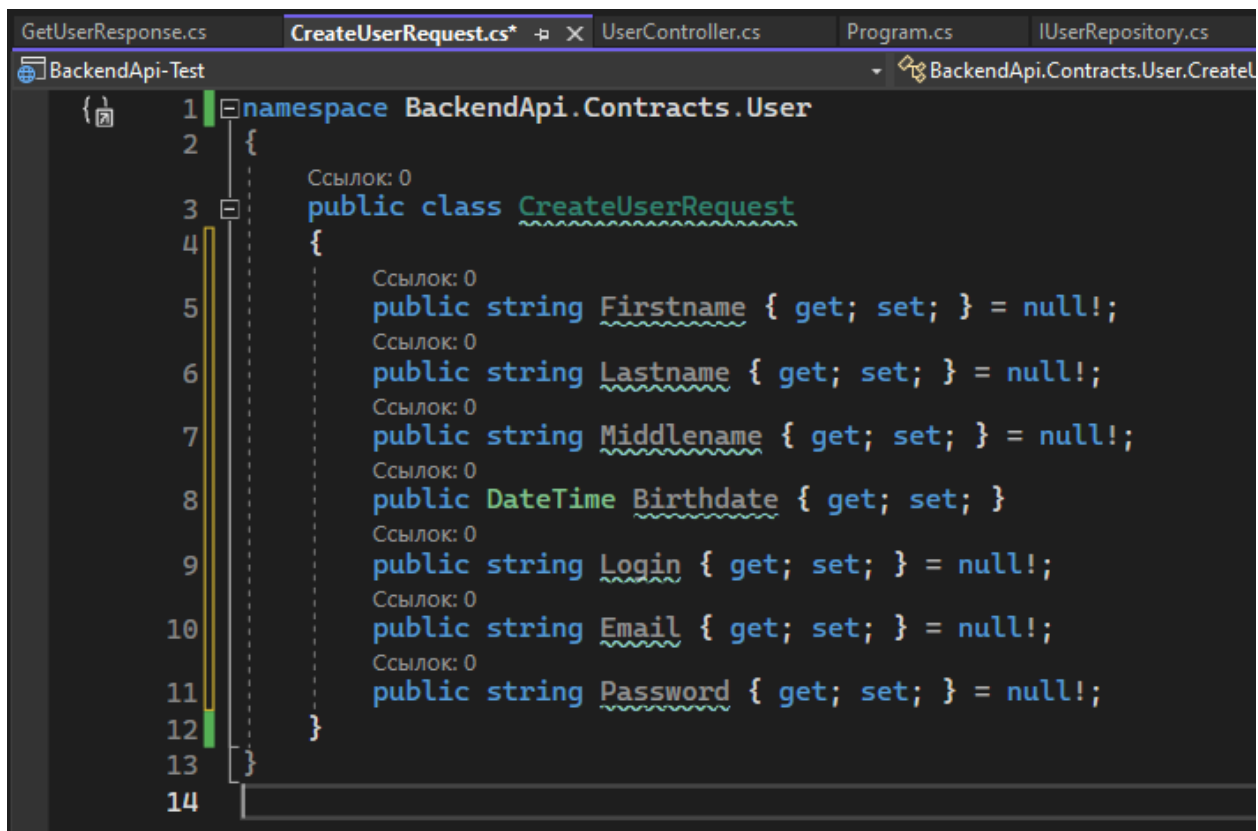
- Для запроса к API - ДействиеРесурсRequest (Например, CreateUserRequest)
- Для ответа от API - ДействиеРесурсResponse (Например, GetUserResponse)



Опишите ваш контракт для создания записи. Для наших проектов будет достаточным описать контракт всеми атрибутами таблицы для которой пишется контракт.

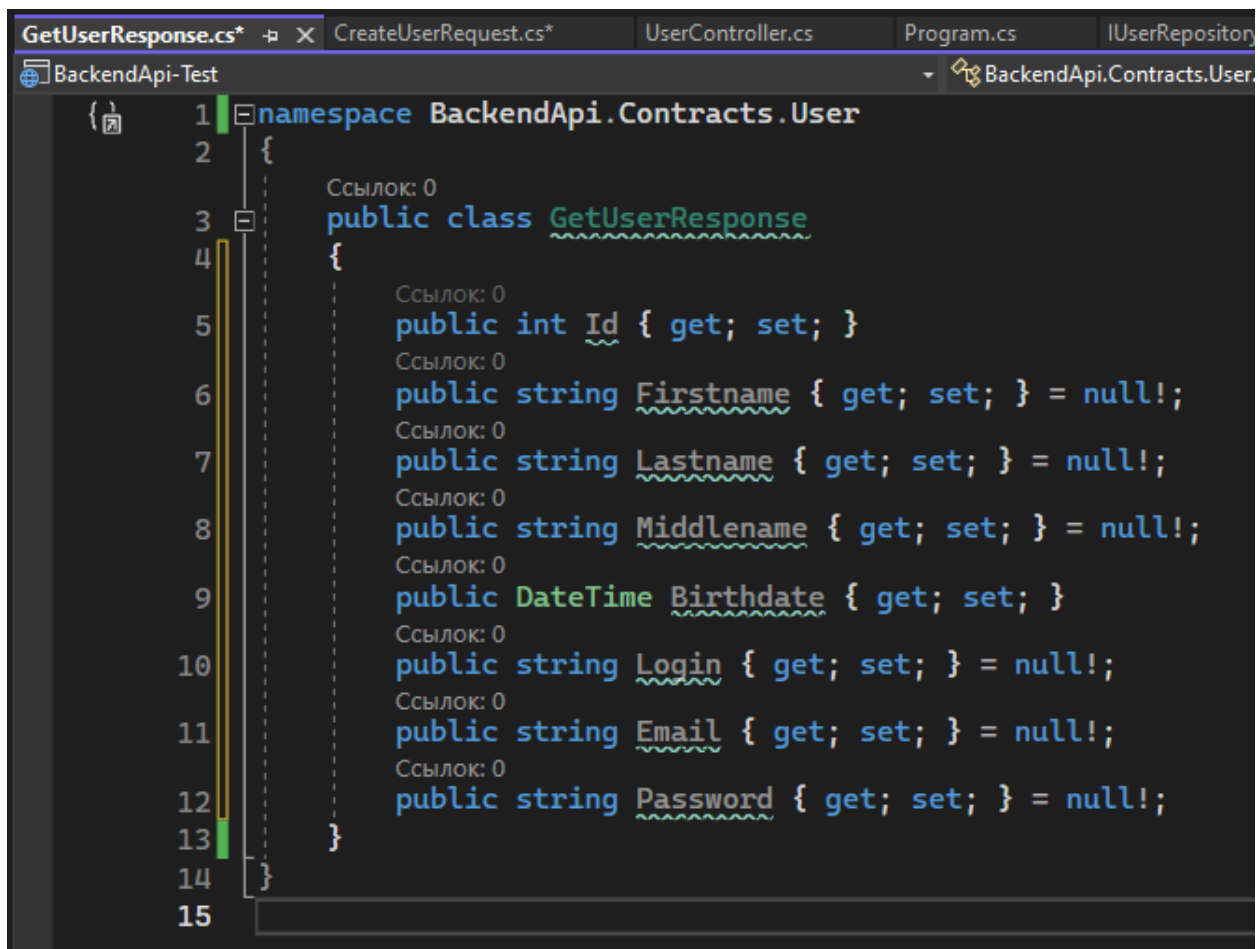


Так как при создании нового пользователя Id генерируется со стороны Базы данных, для данного контракта свойство Id можно убрать.



```
1 namespace BackendApi.Contracts.User
2 {
3     Ссылка: 0
4     public class CreateUserRequest
5     {
6         Ссылка: 0
7         public string Firstname { get; set; } = null!;
8         Ссылка: 0
9         public string Lastname { get; set; } = null!;
10        Ссылка: 0
11        public string Middlename { get; set; } = null!;
12        Ссылка: 0
13        public DateTime Birthdate { get; set; }
14        Ссылка: 0
15        public string Login { get; set; } = null!;
16        Ссылка: 0
17        public string Email { get; set; } = null!;
18        Ссылка: 0
19        public string Password { get; set; } = null!;
20    }
21 }
```

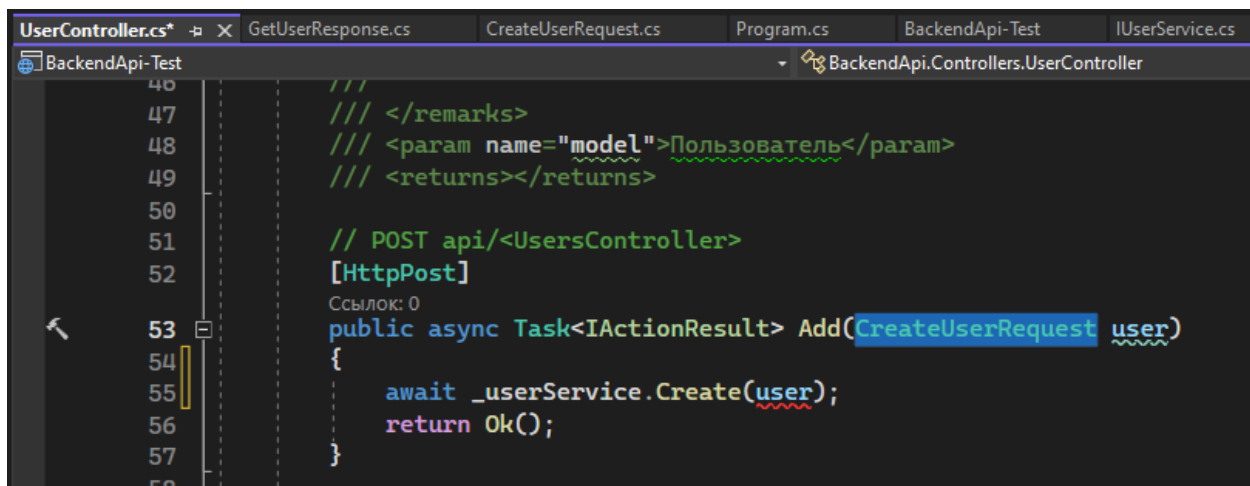
Контракт для получения данных может выглядеть следующим образом.



```
1 namespace BackendApi.Contracts.User
2 {
3     Ссылка: 0
4     public class GetUserResponse
5     {
6         Ссылка: 0
7         public int Id { get; set; }
8         Ссылка: 0
9         public string Firstname { get; set; } = null!;
10        Ссылка: 0
11        public string Lastname { get; set; } = null!;
12        Ссылка: 0
13        public string Middlename { get; set; } = null!;
14        Ссылка: 0
15        public DateTime Birthdate { get; set; }
16        Ссылка: 0
17        public string Login { get; set; } = null!;
18        Ссылка: 0
19        public string Email { get; set; } = null!;
20        Ссылка: 0
21        public string Password { get; set; } = null!;
22    }
23 }
```

Теперь, когда мы сформировали контракт для создания нового пользователя, остается только изменить метод добавления. В качестве параметра я указал контракт для создания нового пользователя.

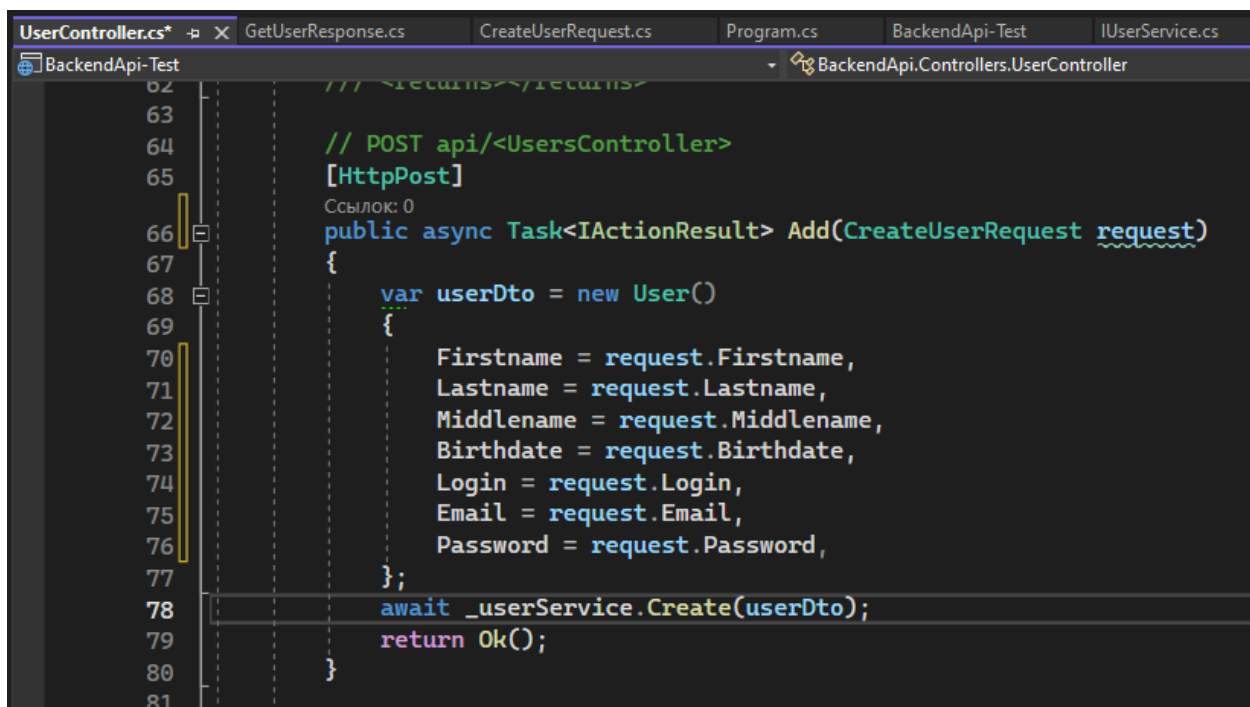
Так как метод Create у сервиса `_userService` в качестве аргумента принимает исключительно модель класса `User`.



```
UserController.cs* x GetUserResponse.cs CreateUserRequest.cs Program.cs BackendApi-Test IUserService.cs
BackendApi-Test BackendApi.Controllers.UserController
46 ///
47 /// </remarks>
48 /// <param name="model">Пользователь</param>
49 /// <returns></returns>
50
51 // POST api/<UsersController>
52 [HttpPost]
53 Ссылка: 0
54 public async Task<IActionResult> Add(CreateUserRequest user)
55 {
56     await _userService.Create(user);
57     return Ok();
58 }
```

Для этого, потребуется преобразовать данные в подходящий тип (произвести маппинг) и передать в качестве аргумента преобразованные данные.

Процесс маппинга может выглядеть следующим образом:



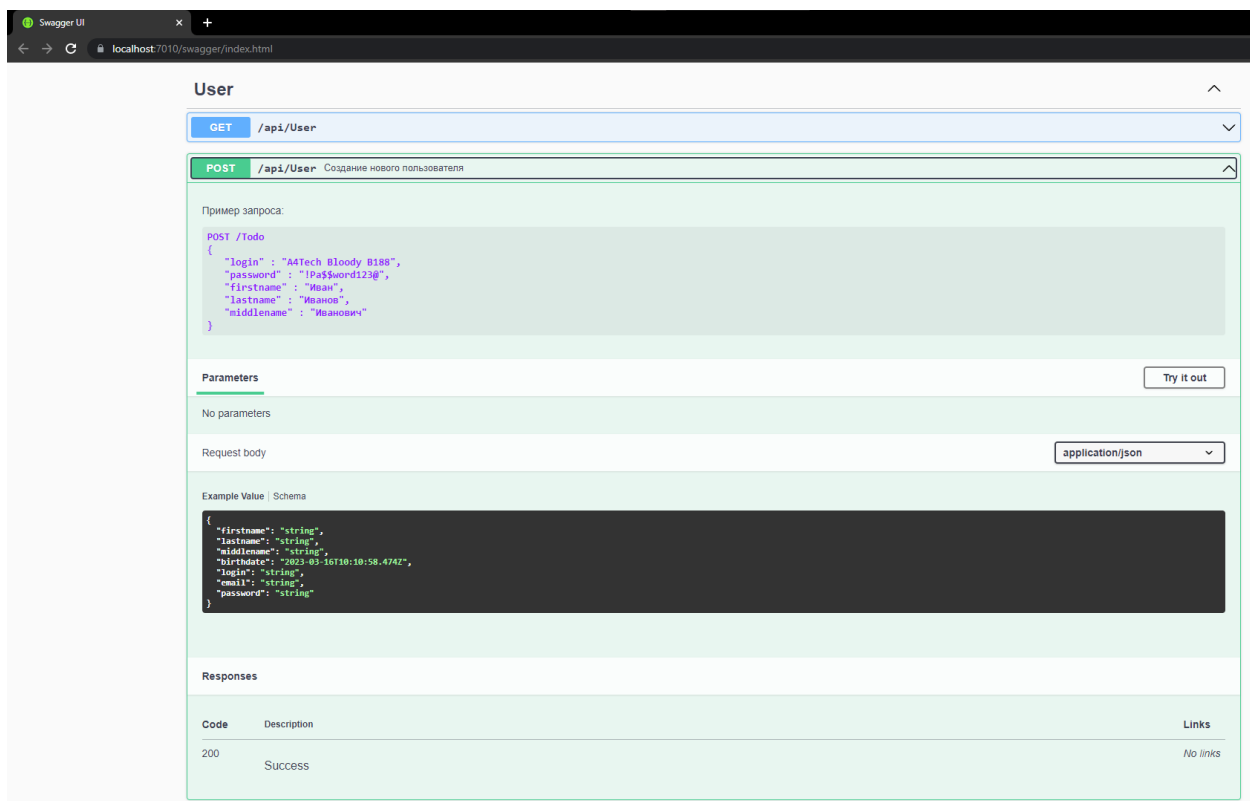
```
UserController.cs* x GetUserResponse.cs CreateUserRequest.cs Program.cs BackendApi-Test IUserService.cs
BackendApi-Test BackendApi.Controllers.UserController
62 /// </returns></returns>
63
64 // POST api/<UsersController>
65 [HttpPost]
66 Ссылка: 0
67 public async Task<IActionResult> Add(CreateUserRequest request)
68 {
69     var userDto = new User()
70     {
71         Firstname = request.Firstname,
72         Lastname = request.Lastname,
73         Middlename = request.Middlename,
74         Birthdate = request.Birthdate,
75         Login = request.Login,
76         Email = request.Email,
77         Password = request.Password,
78     };
79     await _userService.Create(userDto);
80     return Ok();
81 }
```

Такую операцию потребуется сделать и для метода, который возвращает данные.

```
UserController.cs*  X GetUserResponse.cs  CreateUserRequest.cs  Program.cs  BackendApi-Test  IUs
BackendApi-Test  BackendApi.Controllers.UserController

26  [HttpGet(template: "{id}")]
27  Ссылки: 0
28  public async Task<IActionResult> GetById(int id)
29  {
30      var result :User = await _userService.GetById(id);
31      var response = new GetUserResponse()
32      {
33          Id = result.Id,
34          Firstname = result.Firstname,
35          Lastname = result.Lastname,
36          Middlename = result.Middlename,
37          Birthdate = result.Birthdate,
38          Login = result.Login,
39          Email = result.Email,
40          Password = result.Password,
41      };
42      return Ok(response);
43  }
```

Теперь, если запустить проект и проверить отображение Example Value - всё будет в порядке:



Скорее всего, вы задитесь следующим вопросом:

- “Погодите, неужели я должен страдать и для маппинга объектов постоянно расписывать столько строчек кода?”

```
var userDto = new User()
{
    Firstname = request.Firstname,
    Lastname = request.Lastname,
    Middlename = request.Middlename,
    Birthdate = request.Birthdate,
    Login = request.Login,
    Email = request.Email,
    Password = request.Password,
};
```

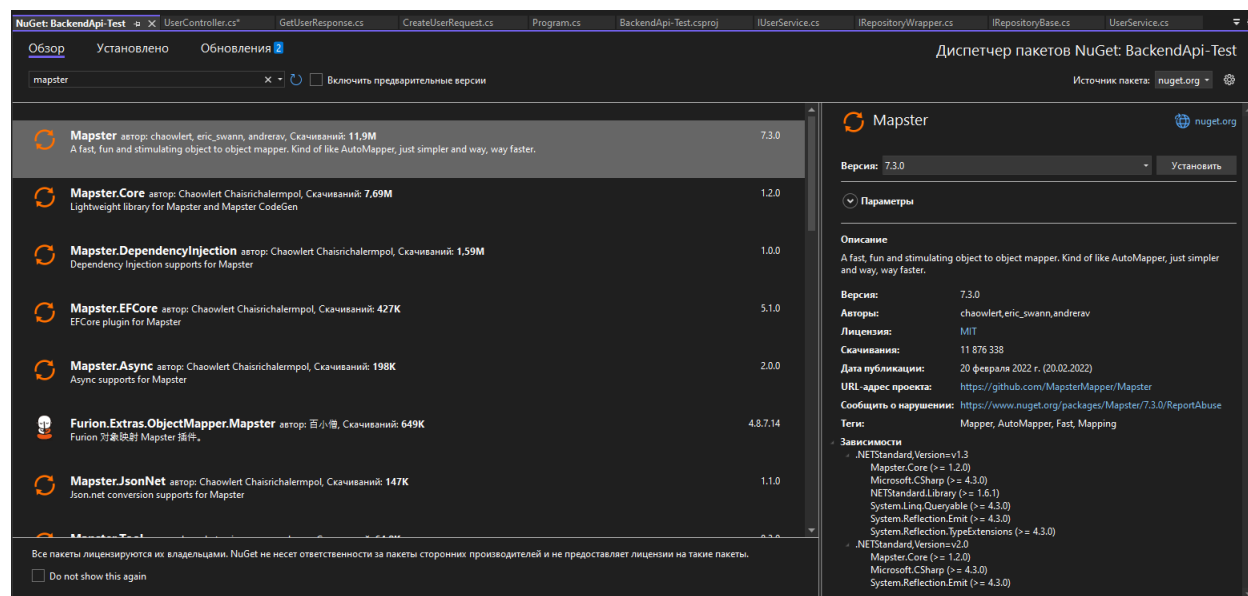
Ответом будет:

- “Нет, есть решения и попроще”

Одним из таких решений будет библиотека Mapster.

Работа с Mapster

Для начала потребуется установить для проекта API следующую библиотеку:





Чтобы преобразовать данные при помощи данной библиотеки, используйте следующую конструкцию.

переменная.Adapt<Требуемый_класс>();

Теперь, достаточно только дописать метод расширения `Adapt<T>()` для какого-нибудь объекта и вы получите объект в требуемом типе. По сути, точно такая же операция по преобразованию объектов, которую мы сделали ранее, но только в одну строчку кода.

```
UserController.cs  x  GetUserResponse.cs  CreateUserRequest.cs  Program.cs  BackendApi-Test.csproj  IUserService.cs
BackendApi-Test
[HttpPost]
Ссылка: 0
public async Task<IActionResult> Add(CreateUserRequest request)
{
    var userDto = request.Adapt<User>();
    await _userService.Create(userDto);
    return Ok();
}
```

Задачи:

1. Описать все методы вашего API (Достаточно будет текста, который кратко опишет действие для описываемого метода API)

Например:

GET	/api/User	Получение списка всех пользователей БД
Parameters		
No parameters		
Responses		
Code	Description	Links
200	Success	No links

2. Исправить проблему с отображением Example Value, где это необходимо