

WEB API

Введение в Web API

Web API представляет способ построения приложения ASP.NET, который специально заточен для работы в стиле REST (Representation State Transfer или "передача состояния представления"). REST-архитектура предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером:

- GET
- POST
- PUT
- DELETE

Зачастую REST-стиль особенно удобен при создании всякого рода Single Page Application, которые нередко используют специальные javascript-фреймворки типа Angular, React или Vue.js. По сути Web API представляет собой веб-службу, к которой могут обращаться другие приложения. Причем эти приложения могут представлять любую технологию и платформу - это могут быть веб-приложения, мобильные или десктопные клиенты.

Создадим проект Web API. Для этого при создании проекта ASP.NET Core среди шаблонов выберем **API**:

Проект, который создается в Visual Studio, будет во многом напоминать проект для MVC за тем исключением, что в нем не будет представлений:

Кроме того, здесь есть модель WeatherForecast и типовой контроллер WeatherForecastController, который использует данную модель для обработки запросов:

```
1    using System;
2    using System.Collections.Generic;
3    using System.Linq;
4    using System.Threading.Tasks;
5    using Microsoft.AspNetCore.Mvc;
6    using Microsoft.Extensions.Logging;
7
8    namespace HelloWebApi.Controllers
9    {
10        [ApiController]
```

```

11     [Route("[controller]")]
12     public class WeatherForecastController : ControllerBase
13     {
14         private static readonly string[] Summaries = new[]
15         {
16             "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot"
17         };
18
19         private readonly ILogger<WeatherForecastController> _logger;
20
21         public WeatherForecastController(ILogger<WeatherForecastController> logger)
22         {
23             _logger = logger;
24         }
25
26         [HttpGet]
27         public IEnumerable<WeatherForecast> Get()
28         {
29             var rng = new Random();
30             return Enumerable.Range(1, 5).Select(index => new WeatherForecast
31             {
32                 Date = DateTime.Now.AddDays(index),
33                 TemperatureC = rng.Next(-20, 55),
34                 Summary = Summaries[rng.Next(Summaries.Length)]
35             })
36             .ToArray();
37         }
38     }
39 }

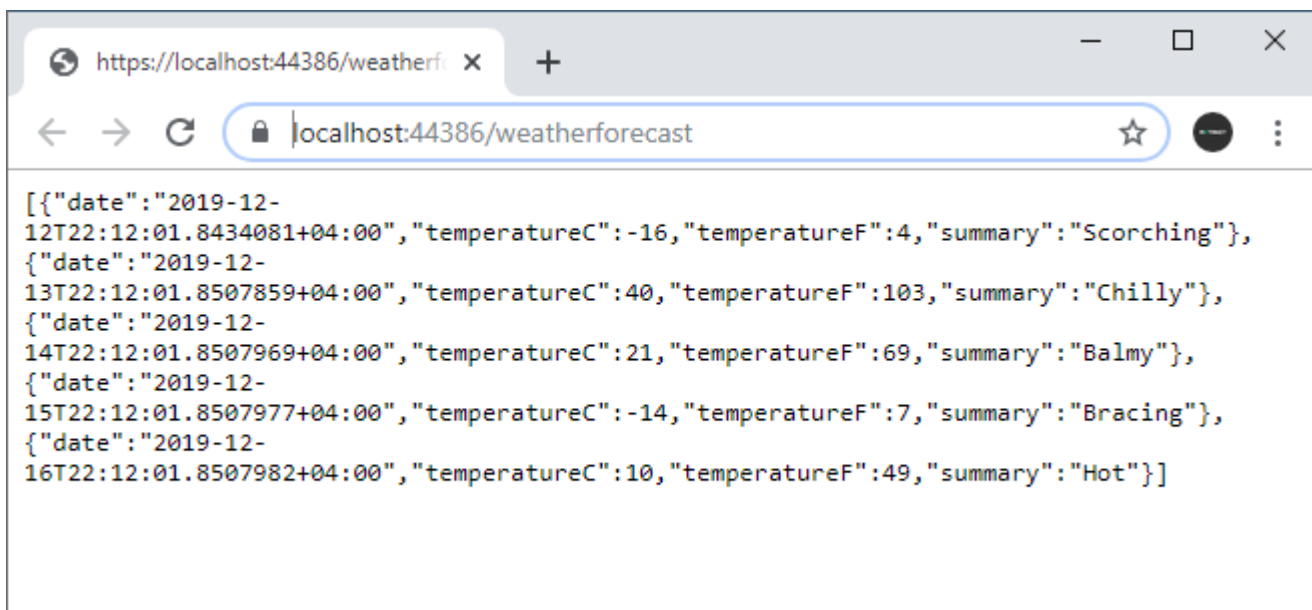
```

Определение контроллера начинается с атрибута **ApiController**, который позволяет добавить к контроллеру некоторую дополнительную функциональность. Но в реальности он необязателен для работы api-контроллера.

Для контроллера определен один общий маршрут с помощью атрибута `[Route("[controller]")]`. В итоге обращение по имени контроллера `/weatherforecast` будет соответствовать обращению к контроллеру `WeatherForecastController`, причем почти ко всем действиям сразу.

К единственному методу контроллера применяется специальный атрибут `[HttpGet]`, который указывает, какой именно тип запроса будет обрабатываться методом. Так, например, запрос `GET /weatherforecast` будет сопоставлен с методом `IEnumerable<WeatherForecast> Get` и вернет в ответ клиенту некоторый набор данных.

Так, в данном случае метод Get() эмулирует прогноз погоды. В реальности в этом контроллере нет большого смысла, тем не менее мы можем запустить проект на выполнение и увидеть в браузере возвращаемые методом данные:



Из других особенностей проекта Web API следует отметить содержимое класса Startup:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Builder;
6  using Microsoft.AspNetCore.Hosting;
7  using Microsoft.AspNetCore.HttpsPolicy;
8  using Microsoft.AspNetCore.Mvc;
9  using Microsoft.Extensions.Configuration;
10 using Microsoft.Extensions.DependencyInjection;
11 using Microsoft.Extensions.Hosting;
12 using Microsoft.Extensions.Logging;
13
14 namespace HelloWebApi
15 {
16     public class Startup
17     {
18         public Startup(IConfiguration configuration)
19         {
20             Configuration = configuration;
21         }
22     }
23 }
```

```

22
23     public IConfiguration Configuration { get; }
24
25     public void ConfigureServices(IServiceCollection services)
26     {
27         services.AddControllers();
28     }
29
30     public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
31     {
32         if (env.IsDevelopment())
33         {
34             app.UseDeveloperExceptionPage();
35         }
36
37         app.UseHttpsRedirection();
38
39         app.UseRouting();
40
41         app.UseAuthorization();
42
43         app.UseEndpoints(endpoints =>
44         {
45             endpoints.MapControllers();
46         });
47     }
48 }
49 }

```

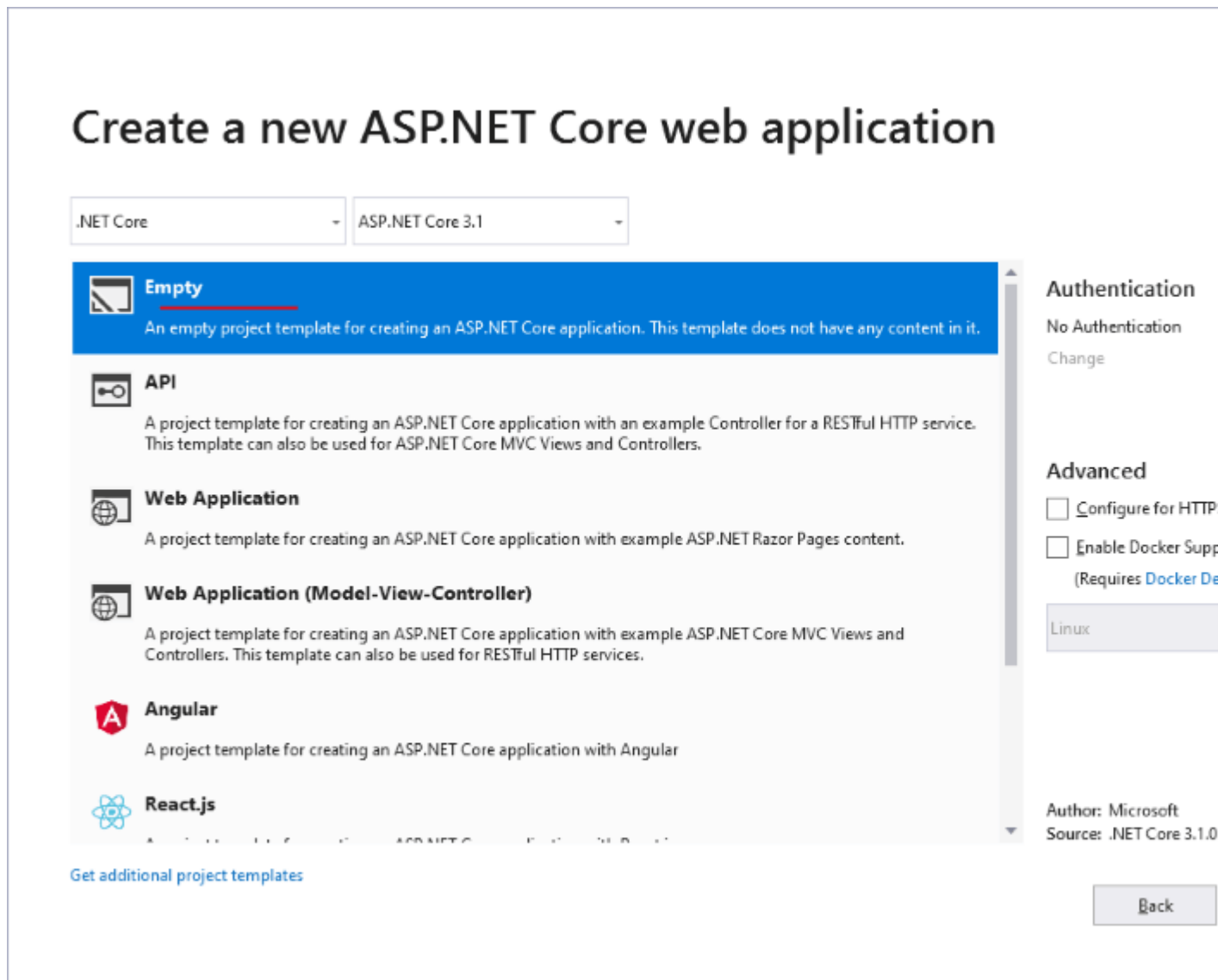
Прежде всего, поскольку в данном случае не используются представления, то подключение в методе `ConfigureServices()` сервисов MVC, необходимых для работы контроллеров Web API производится с помощью метода **`services.AddControllers()`**

Второй момент - при использовании маршрутизации в методе `Configure()` не определяется никаких маршрутов. Вместе этого просто вызывается метод `endpoints.MapControllers()`, который позволяет сопоставлять запросы с контроллерами. В итоге конкретные маршруты задаются локально с помощью атрибутов контроллера.

В итоге, как можно увидеть, большого смысла от данного типа проекта нет, равным образом мы могли бы взять пустой проект и добавить все необходимое сами.

Создание контроллера

Создадим простейшее приложение на Web API, которое будет выполнять все основные операции с данными. Для этого создадим проект по типу Empty:



Далее добавим в проект новую папку **Models**, а в нее поместим новый класс User:

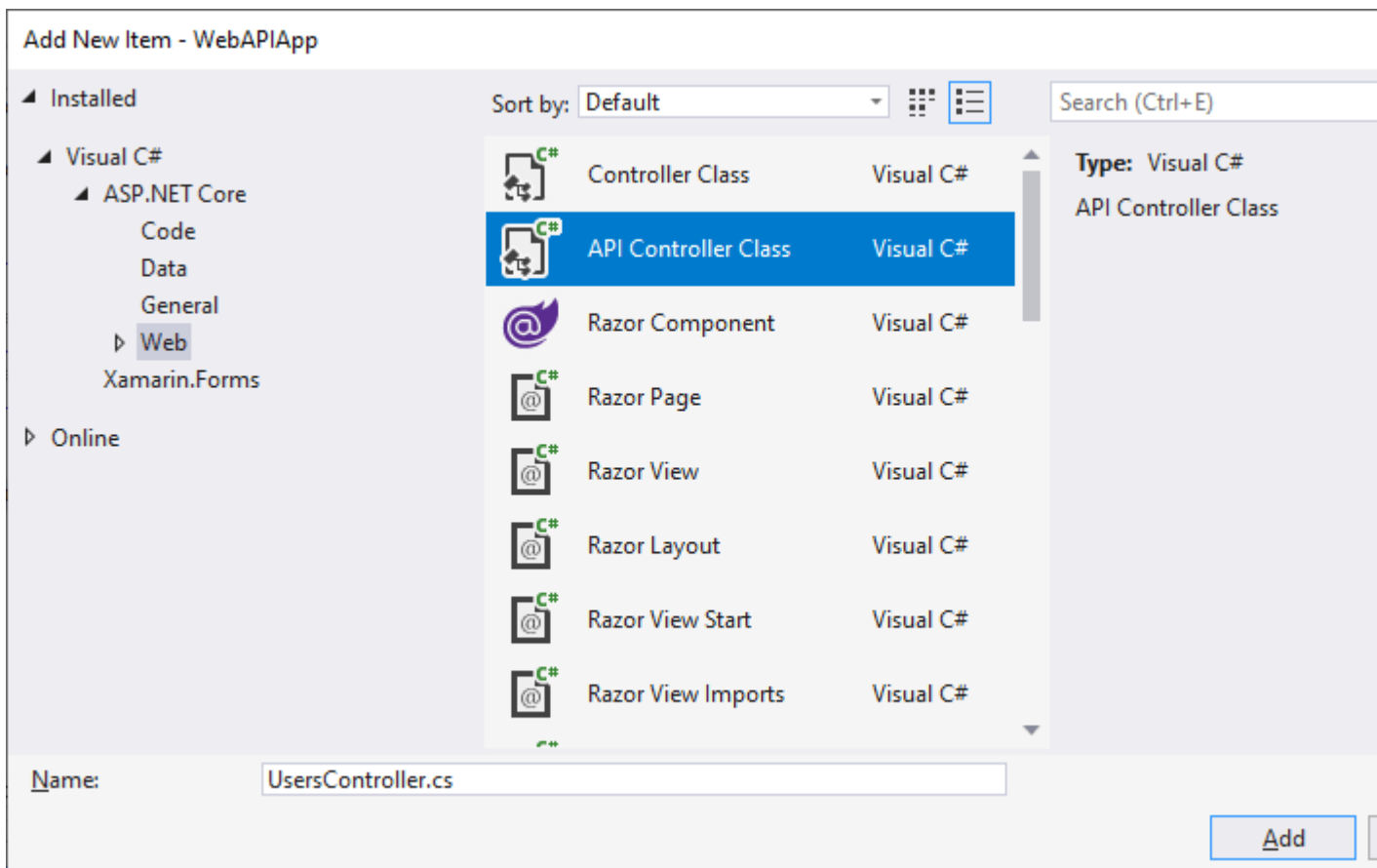
```
1 public class User
2 {
3     public int Id { get; set; }
4     public string Name { get; set; }
5     public int Age { get; set; }
6 }
```

Для взаимодействия с MS SQL Server через Entity Framework через пакетный менеджер Nuget добавим в проект пакет **Microsoft.EntityFrameworkCore.SqlServer**.

Также добавим в папку Models новый класс **UsersContext** для взаимодействия с базой данных:

```
1    using Microsoft.EntityFrameworkCore;
2
3    namespace WebAPIApp.Models
4    {
5        public class UsersContext : DbContext
6        {
7            public DbSet<User> Users { get; set; }
8            public UsersContext(DbContextOptions<UsersContext> options)
9                : base(options)
10            {
11                Database.EnsureCreated();
12            }
13        }
14    }
```

Далее добавим в проект новую папку **Controllers**, а в ней создадим новый api-контроллер. Для этого при добавлении нового элемента в проект можно использовать шаблон **API Controller Class**:



Назовем новый элемент **UserController**.

После его создания изменим его код следующим образом:

```
1  using System.Collections.Generic;
2  using System.Linq;
3  using Microsoft.EntityFrameworkCore;
4  using Microsoft.AspNetCore.Mvc;
5  using WebAPIApp.Models;
6  using System.Threading.Tasks;
7
8  namespace WebAPIApp.Controllers
9  {
10     [ApiController]
11     [Route("api/[controller]")]
12     public class UsersController : ControllerBase
13     {
14         UsersContext db;
15         public UsersController(UsersContext context)
16         {
17             db = context;
```

```

18         if (!db.Users.Any())
19         {
20             db.Users.Add(new User { Name = "Tom", Age = 26 });
21             db.Users.Add(new User { Name = "Alice", Age = 31 });
22             db.SaveChanges();
23         }
24     }
25
26     [HttpGet]
27     public async Task<ActionResult<IEnumerable<User>>> Get()
28     {
29         return await db.Users.ToListAsync();
30     }
31
32     // GET api/users/5
33     [HttpGet("{id}")]
34     public async Task<ActionResult<User>> Get(int id)
35     {
36         User user = await db.Users.FirstOrDefaultAsync(x => x.Id == id);
37         if (user == null)
38             return NotFound();
39         return new ObjectResult(user);
40     }
41
42     // POST api/users
43     [HttpPost]
44     public async Task<ActionResult<User>> Post(User user)
45     {
46         if (user == null)
47         {
48             return BadRequest();
49         }
50
51         db.Users.Add(user);
52         await db.SaveChangesAsync();
53         return Ok(user);
54     }
55
56     // PUT api/users/
57     [HttpPut]
58     public async Task<ActionResult<User>> Put(User user)
59     {
60         if (user == null)

```



```

61         {
62             return BadRequest();
63         }
64         if (!db.Users.Any(x => x.Id == user.Id))
65         {
66             return NotFound();
67         }
68
69         db.Update(user);
70         await db.SaveChangesAsync();
71         return Ok(user);
72     }
73
74     // DELETE api/users/5
75     [HttpDelete("{id}")]
76     public async Task<ActionResult<User>> Delete(int id)
77     {
78         User user = db.Users.FirstOrDefault(x => x.Id == id);
79         if (user == null)
80         {
81             return NotFound();
82         }
83         db.Users.Remove(user);
84         await db.SaveChangesAsync();
85         return Ok(user);
86     }
87 }
88 }

```

Прежде всего к контроллеру применяется атрибут **[ApiController]**, который позволяет использовать ряд дополнительных возможностей, в частности, в плане привязки модели и ряд других. Также к контроллеру применяется атрибут маршрутизации, который указывает, как контроллер будет сопоставляться с запросами.

В конструкторе контроллера получаем контекст данных и используем его для операций с данными. Также в конструкторе контроллера добавляем ряд начальных данных.

Контроллер API предназначен преимущественно для обработки запросов протокола HTTP: Get, Post, Put, Delete, Patch, Head, Options. В данном случае для каждого типа запросов в контроллере определен свои методы. Так, метод `Get()` обрабатывает запросы типа GET и возвращает коллекцию объектов из бд.

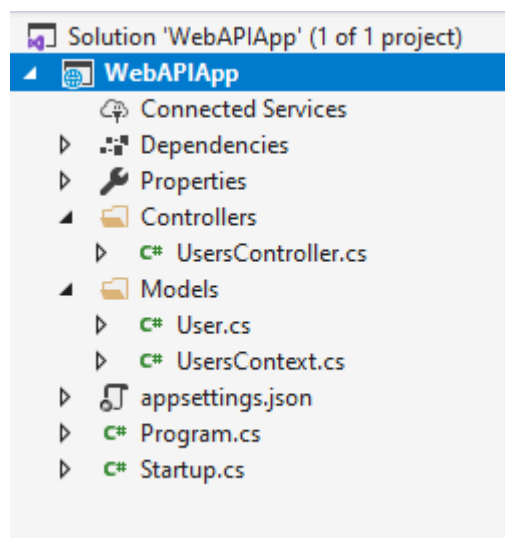
Если запрос Get содержит параметр id (идентификатор объекта), то он обрабатывается другим методом - `Get(int id)`, который возвращает объект по переданному id.

Запросы типа Post обрабатываются методом `Post(User user)`, который получает из тела запроса отправленные данные и добавляет их в базу данных.

Метод `Put(User user)` обрабатывает запросы типа Put - получает данные из запроса и изменяет ими объект в базе данных.

И метод `Delete(int id)` обрабатывает запросы типа Delete, то есть запросы на удаление - получает из запроса параметр id и по данному идентификатору удаляет объект из БД.

В итоге у нас получится следующий проект:



Теперь, чтобы это все использовать, изменим код класса Startup:

```
1    using Microsoft.AspNetCore.Builder;
2    using Microsoft.Extensions.DependencyInjection;
3    using Microsoft.EntityFrameworkCore;
4    using WebAPIApp.Models;
5
6    namespace WebAPIApp
7    {
8        public class Startup
9        {
10            public void ConfigureServices(IServiceCollection services)
11            {
12                string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trust
13                // устанавливаем контекст данных
```

```

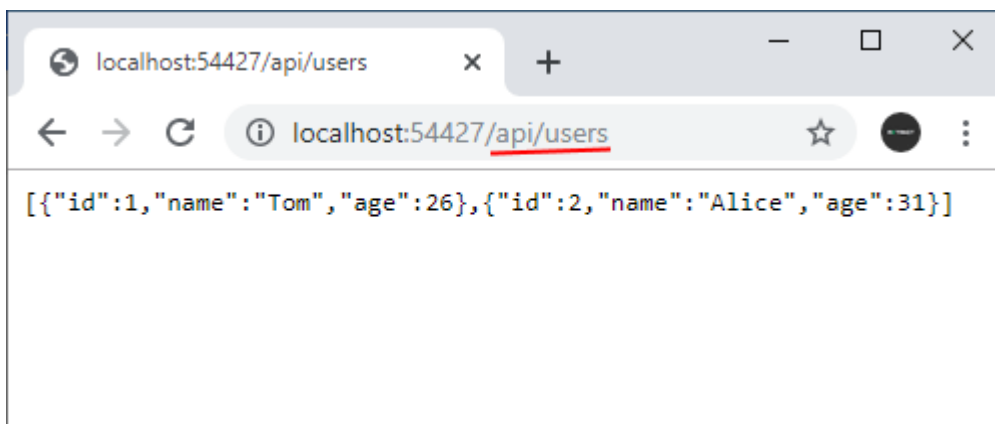
14         services.AddDbContext<UsersContext>(options => options.UseSqlServer(con
15
16         services.AddControllers(); // используем контроллеры без представлений
17     }
18
19     public void Configure(IApplicationBuilder app)
20     {
21         app.UseDeveloperExceptionPage();
22
23         app.UseRouting();
24
25         app.UseEndpoints(endpoints =>
26         {
27             endpoints.MapControllers(); // подключаем маршрутизацию на контролл
28         });
29     }
30 }
31 }

```

Чтобы задействовать контроллеры, в методе `ConfigureServices()` вызывается метод **`services.AddControllers()`**.

Чтобы подключить маршрутизацию контроллеров на основе атрибутов, в методе `Configure()` вызывается метод **`endpoints.MapControllers()`**. После этого мы сможем обращаться к контроллеру через запрос `api/users`, поскольку к контроллеру применяется атрибут маршрутизации `[Route("api/[controller]")]`, где параметр "controller" указывает на название контроллера.

Запустим приложение и обратимся по пути `api/users`:



Поскольку запрос из адресной строки браузера представляет GET-запрос, то его будет обрабатывать метод

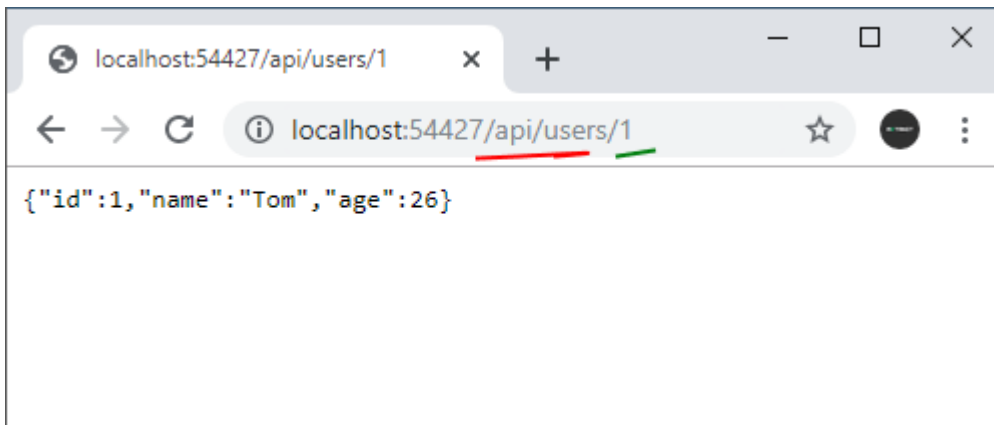
```

1  [HttpGet]
2  public async Task<ActionResult<IEnumerable<User>>> Get()
3  {
4      return await db.Users.ToListAsync();
5  }

```

Этот метод возвратит всех пользователей из базы данных. Поэтому в браузере мы увидим все те данные, которые были добавлены в конструкторе.

Передадим параметр id:



Поскольку это также запрос типа Get, но теперь также передается параметр id, то сработает следующий метод:

```

1  [HttpGet("{id}")]
2  public IActionResult Get(int id)
3  {
4      User user = db.Users.FirstOrDefault(x => x.Id == id);
5      if (user == null)
6          return NotFound();
7      return new ObjectResult(user);
8  }

```

Тестирование контроллера

В прошлой теме был создан контроллер Web API, и протестирована работа метода GET. Однако напрямую из строки браузера кроме запросов GET другие типы запросов мы протестировать не можем. Конечно, мы можем создать клиент в виде веб-страницы, мобильного приложения под какую-нибудь платформу или даже графического или консольного десктопного приложения, но создание клиента может занять довольно много времени, тогда как нам просто надо протестировать обработку запросов.

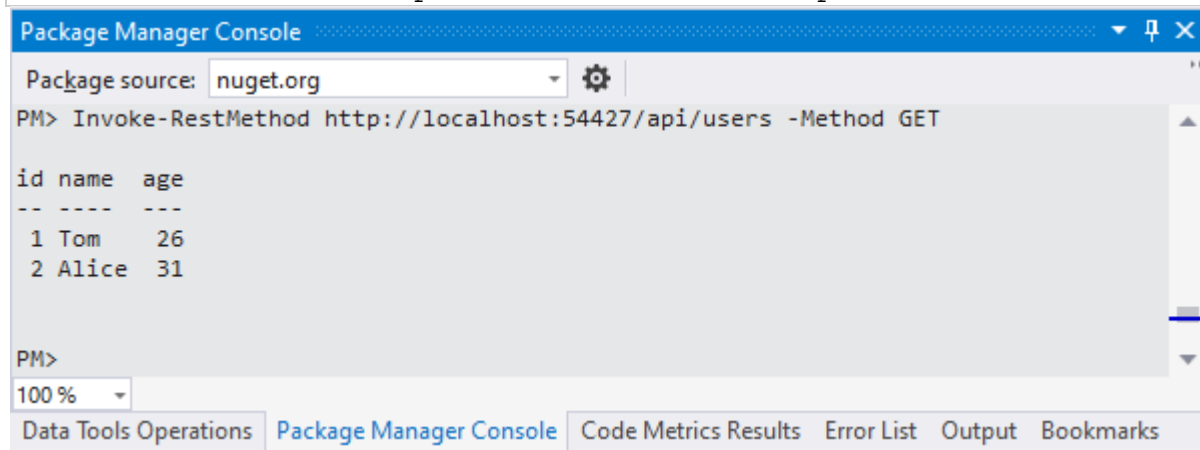
Для тестирования контроллера Web API можно применять специальные инструменты, которые устанавливаются в виде отдельных приложений, либо в виде расширений для браузеров, например, **Fiddler** или **Postman**. Однако Visual Studio предоставляет еще один способ - ввод команд в окне **Package Manager Console**,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

Использование PowerShell

Тестирование GET-запросов

Например, в моем случае приложение запускается по адресу `http://localhost:54427`. Поэтому для тестирования GET-запроса на получение списка объектов от контроллера `UserController` я должен ввести в Package Manager Console следующую команду:

```
Invoke-RestMethod http://localhost:54427/api/users -Method GET
```



Для тестирования другого GET-метода, который возвращает пользователя по id, введем другой запрос:

```
Invoke-RestMethod http://localhost:54427/api/users/1 -Method GET
```

И Package Manager Console выведет:

```
id name age
```

```
-- ---- ---  
1 Tom    26
```

Тестирование POST-запросов

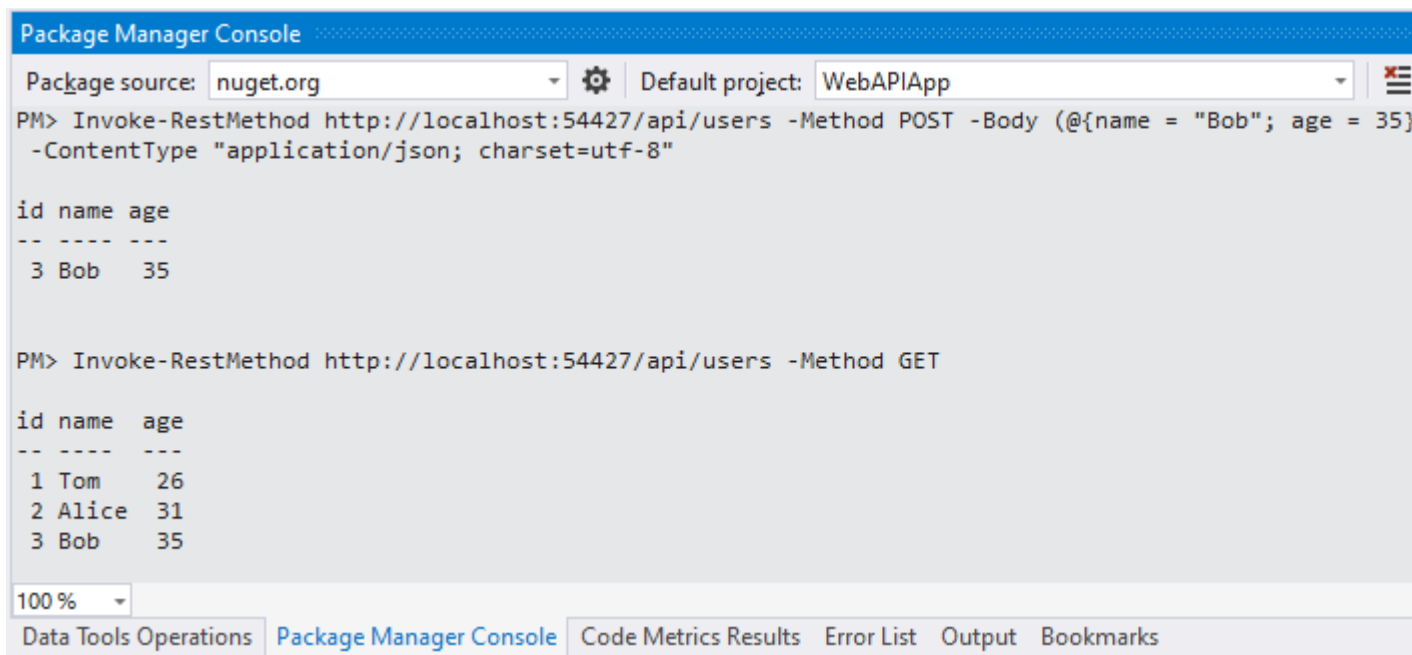
Для тестирования POST-запроса необходимо передать объект в формате json с указанием некоторой дополнительной информацией. В моем случае команда выглядела бы так:

```
Invoke-RestMethod http://localhost:54427/api/users -Method POST -  
Body (@{name = "Bob"; age = 35} | ConvertTo-Json) -ContentType  
"application/json; charset=utf-8"
```

Аргумент `-Body` в этой команде указывает на тело запроса - тот объект, который будет отправляться на сервер. В моем случае это объект класса `User`, поэтому все его значения имеют ключи, которые соответствуют названиям свойств класса `User`. Кроме того, указываем с помощью флага `ConvertTo-Json`, что объект будет отправляться в формате JSON. А дополнительный аргумент `-ContentType` устанавливает в запросе заголовок `Content-Type`. В итоге эта команда возвратит следующий результат:

```
id name age  
-- ---- ---  
3 Bob    35
```

После этого мы можем повторно получить список пользователей и увидеть в нем добавленный объект:



The screenshot shows the Package Manager Console with the following content:

```
Package Manager Console  
Package source: nuget.org Default project: WebAPIApp  
PM> Invoke-RestMethod http://localhost:54427/api/users -Method POST -Body (@{name = "Bob"; age = 35}  
-ContentType "application/json; charset=utf-8"  
  
id name age  
-- ---- ---  
3 Bob    35  
  
PM> Invoke-RestMethod http://localhost:54427/api/users -Method GET  
  
id name age  
-- ---- ---  
1 Tom    26  
2 Alice   31  
3 Bob     35
```

The console window includes a zoom level of 100% and a taskbar at the bottom with tabs for Data Tools, Operations, Package Manager Console (active), Code Metrics Results, Error List, Output, and Bookmarks.

Тестирование PUT-запросов

Для тестирования PUT-запроса передается так же объект, только в данном случае уже надо указать id редактируемого объекта:

```
Invoke-RestMethod http://localhost:54427/api/users -Method PUT -  
Body (@{id = 3; name = "Bob Marley"; age = 44} | ConvertTo-Json) -  
ContentType "application/json"
```

The screenshot shows the Package Manager Console with the following content:

Package source: nuget.org | Default project: WebAPIApp

id	name	age
1	Tom	26
2	Alice	31
3	Bob	35

```
PM> Invoke-RestMethod http://localhost:54427/api/users -Method PUT -Body (@{id = 3; name = "Bob Marley"; age = 44} | ConvertTo-Json) -ContentType "application/json"
```

id	name	age
3	Bob Marley	44

```
PM> Invoke-RestMethod http://localhost:54427/api/users -Method GET
```

id	name	age
1	Tom	26
2	Alice	31
3	Bob Marley	44

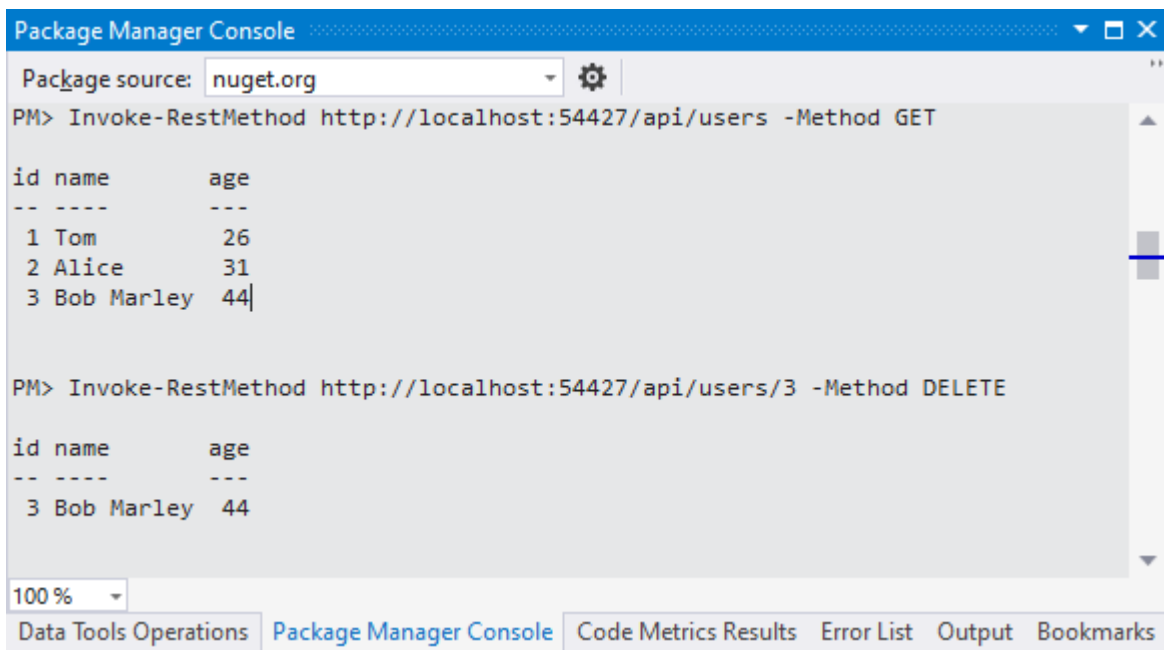
100 %

Data Tools Operations | **Package Manager Console** | Code Metrics Results | Error List | Output | Bookmarks

Тестирование запросов DELETE

В запросе DELETE необходимо передать id удаляемого объекта:

```
Invoke-RestMethod http://localhost:54427/api/users/3 -Method DELETE
```



```
Package Manager Console
Package source: nuget.org
PM> Invoke-RestMethod http://localhost:54427/api/users -Method GET

id name      age
---
1 Tom        26
2 Alice      31
3 Bob Marley 44

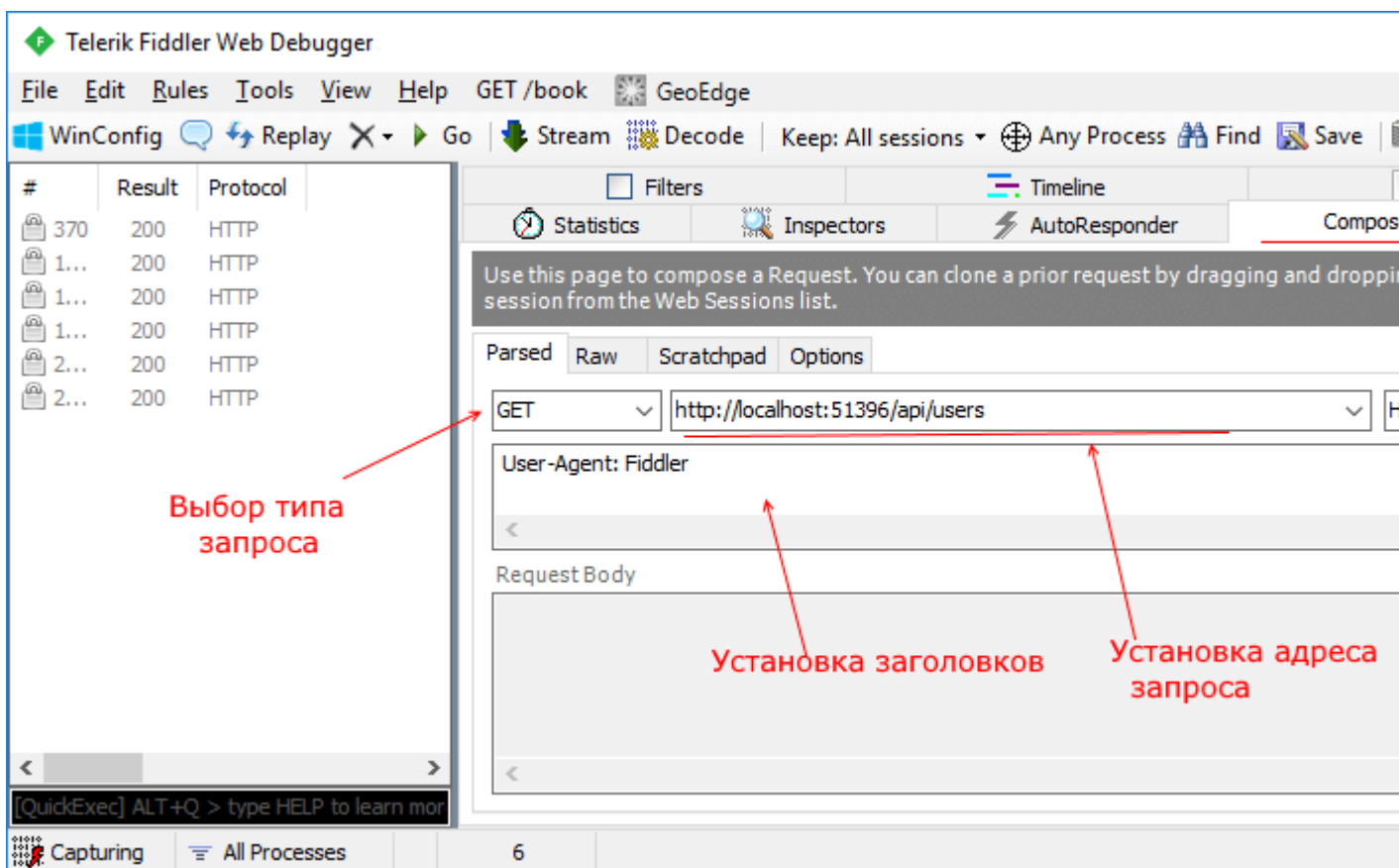
PM> Invoke-RestMethod http://localhost:54427/api/users/3 -Method DELETE

id name      age
---
3 Bob Marley 44
```

Fiddler

Теперь рассмотрим, как использовать [Fiddler](#). Загрузим установочный пакет по ссылке и установим.

После установки запустим проект в Visual Studio на выполнение. Параллельно откроем программу Fiddler и перейдем на вкладку **Composer**. На этой вкладке мы можем выбрать тип запроса (GET/POST/PUT/DELETE) и установить адрес. Так как я буду обращаться к методу, возвращающему список пользователей, то в моем случае это будет запрос GET и адрес *http://localhost:54427/api/users/*.



После ввода адреса нажмем на кнопку Execute. После осуществления запроса в левом поле-списке запросов выберем сделанный только что запрос, и на вкладке **Inspectors** можно будет увидеть результат запроса - список пользователей, который Fiddler получает в сериализованном виде:

Telerik Fiddler Web Debugger

File Edit Rules Tools View Help GET /book GeoEdge

WinConfig Replay Go Stream Decode Keep: All sessions Any Process Find Save

#	Result	Pro...	Host	URL
17721	200	HTTP	Tunnel to	vortex.data
18030	200	HTTP	Tunnel to	microsoft.st
18083	200	HTTP	Tunnel to	portal.mail.r
18120	200	HTTP	Tunnel to	lc61.dsr.live
18229	200	HTTP	Tunnel to	az700632.v
18638	200	HTTP	Tunnel to	az667904.v
18883	200	HTTP	Tunnel to	microsoft.st
19700	200	HTTP	Tunnel to	microsoft.st
20217	200	HTTP	localhost:51396	/api/users
20422	200	HTTP	Tunnel to	portal.mail.r
20471	200	HTTP	Tunnel to	lc61.dsr.live
20472	200	HTTP	Tunnel to	bar.love.ma
20473	200	HTTP	Tunnel to	ok.ru:443
20490	200	HTTP	Tunnel to	microsoft.st
21367	200	HTTP	Tunnel to	microsoft.st
21580	200	HTTP	Tunnel to	ct210.dsr.liv
22248	200	HTTP	Tunnel to	microsoft.st
22678	200	HTTP	Tunnel to	portal.mail.r

выделим запрос

Request Headers [Raw]

GET /api/users HTTP/1.1

Client

User-Agent: Fiddler

Transport

Host: localhost:51396

Get SyntaxView Transformer Headers **TextView** ImageView

Auth Caching Cookies Raw JSON XML

[{"id":1,"name":"Tom","age":27},{id":2,"name":"Alice","age":31},{id":6,"na

полученные данные

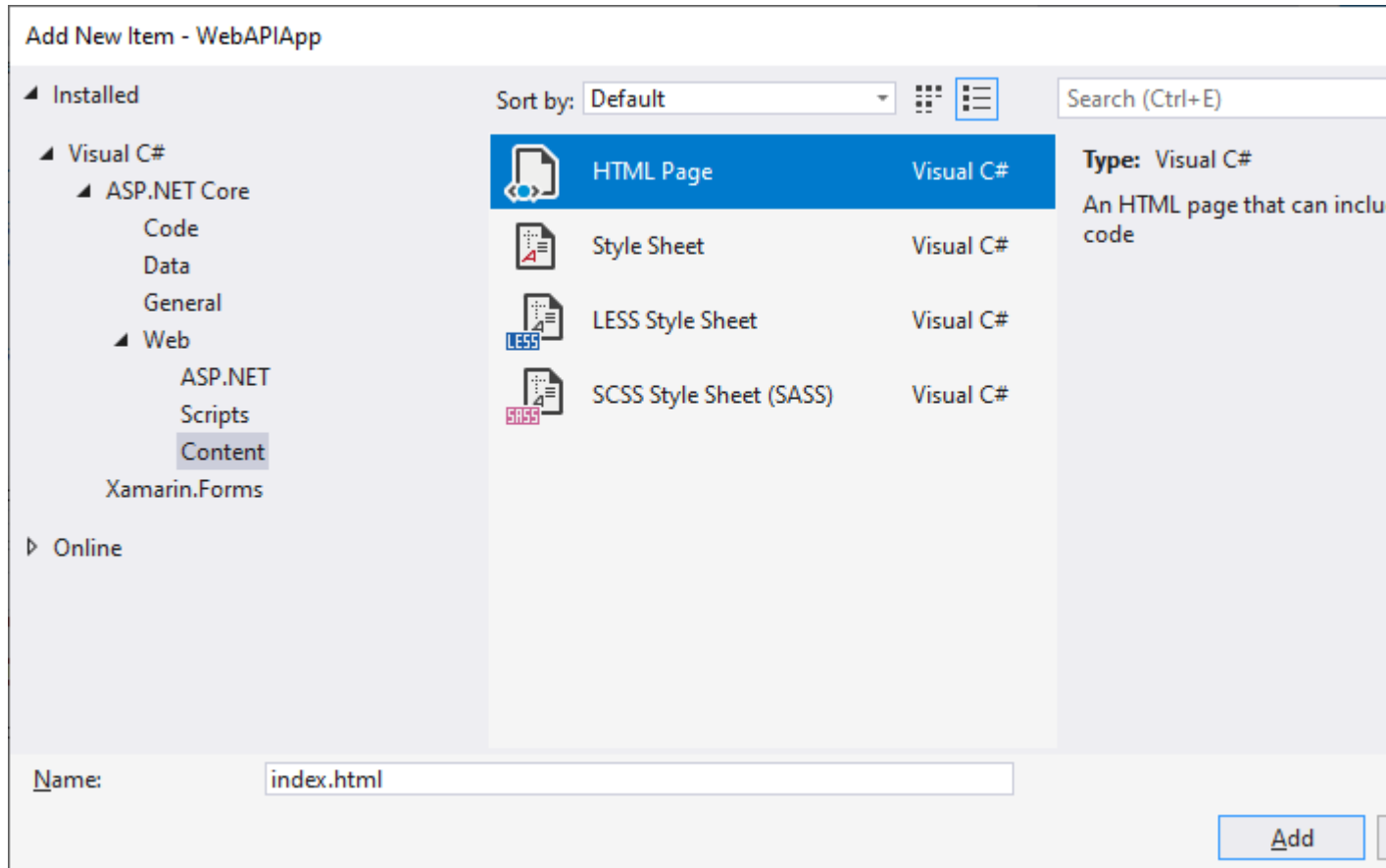
0:55 55/97 Find... (press Ctrl+Enter to highli)

Capturing All Processes 1 / 106 Download Progress: 0 bytes. Hit F5 to refresh.

Создание клиента для WEB API

Продолжим работу с проектом из прошлой теме. В ней был создан и протестирован контроллер UsersController. Теперь создадим для него визуальную часть, которая будет представлять веб-страницу. То есть из веб-страницы мы будем отправлять запросы к контроллеру и обрабатывать ответ от контроллера.

Для создания веб-клиента добавим в проект папку **wwwroot** и затем в ней определим новый элемент **HTML Page**, который назовем "index.html":



Затем изменим класс Startup:

```
1 using Microsoft.AspNetCore.Builder;
2 using Microsoft.Extensions.DependencyInjection;
3 using Microsoft.EntityFrameworkCore;
4 using WebAPIApp.Models;
5
6 namespace WebAPIApp
7 {
8     public class Startup
9     {
10         public void ConfigureServices(IServiceCollection services)
```

```

11         {
12             string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trust
13             // устанавливаем контекст данных
14             services.AddDbContext<UsersContext>(options => options.UseSqlServer(con
15             services.AddControllers(); // используем контроллеры без представлений
16         }
17
18     public void Configure(IApplicationBuilder app)
19     {
20         app.UseDeveloperExceptionPage();
21
22         app.UseDefaultFiles();
23         app.UseStaticFiles();
24
25         app.UseRouting();
26
27         app.UseEndpoints(endpoints =>
28         {
29             endpoints.MapControllers();
30         });
31     }
32 }
33 }

```

Здесь в метод `Configure()` были добавлены два вызова для работы со статическими файлами:

```

1 app.UseDefaultFiles();
2 app.UseStaticFiles();

```

Благодаря этому мы сможем обратиться напрямую к веб-странице, например, по пути **`http://localhost:xxxx/index.html`**. Для этого изменим файл **`index.html`**:

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta charset="utf-8" />
5     <meta name="viewport" content="width=device-width" />
6     <title>Список пользователей</title>
7     <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css" />
8 </head>
9 <body>
10     <h2>Список пользователей</h2>

```

```
11 <form name="userForm">
12     <input type="hidden" name="id" value="0" />
13     <div class="form-group col-md-5">
14         <label for="name">Имя:</label>
15         <input class="form-control" name="name" />
16     </div>
17     <div class="form-group col-md-5">
18         <label for="age">Возраст:</label>
19         <input class="form-control" name="age" type="number" />
20     </div>
21     <div class="panel-body">
22         <button type="submit" id="submit" class="btn btn-primary">Сохранить</button>
23         <a id="reset" class="btn btn-primary">Сбросить</a>
24     </div>
25 </form>
26 <table class="table table-condensed table-striped col-md-6">
27     <thead><tr><th>Id</th><th>Имя</th><th>Возраст</th><th></th></tr></thead>
28     <tbody>
29     </tbody>
30 </table>
31 <div>2019 © Metanit.com</div>
32 <script>
33     // Получение всех пользователей
34     async function GetUsers() {
35         // отправляет запрос и получаем ответ
36         const response = await fetch("/api/users", {
37             method: "GET",
38             headers: { "Accept": "application/json" }
39         });
40         // если запрос прошел нормально
41         if (response.ok === true) {
42             // получаем данные
43             const users = await response.json();
44             let rows = document.querySelector("tbody");
45             users.forEach(user => {
46                 // добавляем полученные элементы в таблицу
47                 rows.append(row(user));
48             });
49         }
50     }
51     // Получение одного пользователя
52     async function GetUser(id) {
53         const response = await fetch("/api/users/" + id, {
```

```

54         method: "GET",
55         headers: { "Accept": "application/json" }
56     });
57     if (response.ok === true) {
58         const user = await response.json();
59         const form = document.forms["userForm"];
60         form.elements["id"].value = user.id;
61         form.elements["name"].value = user.name;
62         form.elements["age"].value = user.age;
63     }
64 }
65 // Добавление пользователя
66 async function CreateUser(userName, userAge) {
67
68     const response = await fetch("api/users", {
69         method: "POST",
70         headers: { "Accept": "application/json", "Content-Type": "applicat
71         body: JSON.stringify({
72             name: userName,
73             age: parseInt(userAge, 10)
74         })
75     });
76     if (response.ok === true) {
77         const user = await response.json();
78         reset();
79         document.querySelector("tbody").append(row(user));
80     }
81 }
82 // Изменение пользователя
83 async function EditUser(userId, userName, userAge) {
84     const response = await fetch("api/users", {
85         method: "PUT",
86         headers: { "Accept": "application/json", "Content-Type": "applicat
87         body: JSON.stringify({
88             id: parseInt(userId, 10),
89             name: userName,
90             age: parseInt(userAge, 10)
91         })
92     });
93     if (response.ok === true) {
94         const user = await response.json();
95         reset();
96         document.querySelector("tr[data-rowid='" + user.id + "']").replace

```

```

97         }
98     }
99     // Удаление пользователя
100    async function DeleteUser(id) {
101        const response = await fetch("/api/users/" + id, {
102            method: "DELETE",
103            headers: { "Accept": "application/json" }
104        });
105        if (response.ok === true) {
106            const user = await response.json();
107            document.querySelector("tr[data-rowid='" + user.id + "']").remove();
108        }
109    }
110
111    // сброс формы
112    function reset() {
113        const form = document.forms["userForm"];
114        form.reset();
115        form.elements["id"].value = 0;
116    }
117    // создание строки для таблицы
118    function row(user) {
119
120        const tr = document.createElement("tr");
121        tr.setAttribute("data-rowid", user.id);
122
123        const idTd = document.createElement("td");
124        idTd.append(user.id);
125        tr.append(idTd);
126
127        const nameTd = document.createElement("td");
128        nameTd.append(user.name);
129        tr.append(nameTd);
130
131        const ageTd = document.createElement("td");
132        ageTd.append(user.age);
133        tr.append(ageTd);
134
135        const linksTd = document.createElement("td");
136
137        const editLink = document.createElement("a");
138        editLink.setAttribute("data-id", user.id);
139        editLink.setAttribute("style", "cursor:pointer;padding:15px;");

```

```
140         editLink.append("Изменить");
141         editLink.addEventListener("click", e => {
142
143             e.preventDefault();
144             GetUser(user.id);
145         });
146         linksTd.append(editLink);
147
148         const removeLink = document.createElement("a");
149         removeLink.setAttribute("data-id", user.id);
150         removeLink.setAttribute("style", "cursor:pointer;padding:15px;");
151         removeLink.append("Удалить");
152         removeLink.addEventListener("click", e => {
153
154             e.preventDefault();
155             DeleteUser(user.id);
156         });
157
158         linksTd.append(removeLink);
159         tr.appendChild(linksTd);
160
161         return tr;
162     }
163     // сброс значений формы
164     document.getElementById("reset").click(function (e) {
165
166         e.preventDefault();
167         reset();
168     })
169
170     // отправка формы
171     document.forms["userForm"].addEventListener("submit", e => {
172         e.preventDefault();
173         const form = document.forms["userForm"];
174         const id = form.elements["id"].value;
175         const name = form.elements["name"].value;
176         const age = form.elements["age"].value;
177         if (id == 0)
178             CreateUser(name, age);
179         else
180             EditUser(id, name, age);
181     });
182
```



```

183         // загрузка пользователей
184         GetUsers();
185
186     </script>
187 </body>
188 </html>

```

Основная логика здесь заключена в коде javascript. При загрузке страницы в браузере получаем все объекты из БД с помощью функции `GetUsers`:

```

1  async function GetUsers() {
2      const response = await fetch("/api/users", {
3          method: "GET",
4          headers: { "Accept": "application/json" }
5      });
6      if (response.ok === true) {
7          const users = await response.json();
8          let rows = document.querySelector("tbody");
9          users.forEach(user => {
10             rows.append(row(user));
11         });
12     }
13 }

```

Для добавления строк в таблицу используется функция `row()`, которая возвращает строку. В этой строке будут определены ссылки для изменения и удаления пользователя.

Ссылка для изменения пользователя с помощью функции `GetUser()` получает с сервера выделенного пользователя:

```

1  async function GetUser(id) {
2      const response = await fetch("/api/users/" + id, {
3          method: "GET",
4          headers: { "Accept": "application/json" }
5      });
6      if (response.ok === true) {
7          const user = await response.json();
8          const form = document.forms["userForm"];
9          form.elements["id"].value = user.id;
10         form.elements["name"].value = user.name;
11         form.elements["age"].value = user.age;
12     }
13 }

```

И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит id пользователя, мы можем узнать, какое действие выполняется - добавление или редактирование. Если id равен 0, то выполняется функция CreateUser, которая отправляет данные в POST-запросе:

```
1  async function CreateUser(userName, userAge) {
2
3      const response = await fetch("api/users", {
4          method: "POST",
5          headers: { "Accept": "application/json", "Content-Type": "application/json" },
6          body: JSON.stringify({
7              name: userName,
8              age: parseInt(userAge, 10)
9          })
10     });
11     if (response.ok === true) {
12         const user = await response.json();
13         reset();
14         document.querySelector("tbody").append(row(user));
15     }
16 }
```

Если же ранее пользователь был загружен на форму, и в скрытом поле сохранился его id, то выполняется функция EditUser, которая отправляет PUT-запрос:

```
1  async function EditUser(userId, userName, userAge) {
2      const response = await fetch("api/users", {
3          method: "PUT",
4          headers: { "Accept": "application/json", "Content-Type": "application/json" },
5          body: JSON.stringify({
6              id: parseInt(userId, 10),
7              name: userName,
8              age: parseInt(userAge, 10)
9          })
10     });
11     if (response.ok === true) {
12         const user = await response.json();
13         reset();
14         document.querySelector("tr[data-rowid='" + user.id + "']").replaceWith(row(user));
15     }
16 }
```

При нажатии на ссылку "Удалить" выполняется DELETE-запрос, который по id удаляет пользователя.

Список пользователей

localhost:54427/index.html

Список пользователей

Имя:

Возраст:

Сохранить

Сбросить

Id	Имя	возраст	
1	Tom	26	Изменить Удалить
2	Alice	31	Изменить Удалить
4	Sam	28	Изменить Удалить

2019 © Metanit.com

Валидация в Web API

В прошлой теме было рассмотрено создание представления - визуальной части для работы с Web API. В частности, мы могли создать или отредактировать модель и отправить ее на сервер. Но при этом не учитывалась валидация данных. Более того не учитывался вывод ошибок валидации, чтобы пользователь смог увидеть, что не так, изменить данные и повторить отправку.

Если бы мы работали в ASP.NET Core MVC, то там с валидацией все проще - с помощью значения **ModelState.IsValid** проверяем корректность модели. Если модель проходит валидацию, то перенаправляем на определенное действие, если не проходит валидацию, то возвращаем представление с ошибками. Однако Web API использует в целом иную модель обработки запросов, а взаимодействие между сервером и клиентом происходит главным образом через Ajax, что накладывает свои ограничения на валидацию данных.

При использовании Web API состояние обработки запроса на сервере мы можем контролировать с помощью статусных кодов:

- 200: статус Ok. Указывает на удачное выполнение запроса
- 201: статус Created. Указывает на успешное создание объекта, как правило, используется в запросах POST
- 204: статус NoContent - запрос прошел успешно, например, после удаления
- 400: статус BadRequest - ошибка при выполнении запроса
- 401: статус Unauthorized - пользователь не авторизован
- 403: статус Forbidden - доступ запрещен
- 404: статус NotFound - ресурс не найден

Отправляя определенный статусный код, мы уже даем клиенту знать о характере возникшей ошибки или статусе запросе.

Но мы не ограничены статусными кодами и, как и в MVC, можем использовать для валидации объект **ModelState**.

В прошлых темах мы работали с моделью User. Теперь добавим в нее атрибуты валидации:

```
1 using System;
2 using System.ComponentModel.DataAnnotations;
3
4 namespace WebAPIApp.Models
5 {
6     public class User
7     {
8         public int Id { get; set; }
9         [Required(ErrorMessage = "Укажите имя пользователя")]
```

```

10         public string Name { get; set; }
11         [Range(1, 100, ErrorMessage = "Возраст должен быть в промежутке от 1 до 100
12         [Required(ErrorMessage = "Укажите возраст пользователя")]
13         public int Age { get; set; }
14     }
15 }

```

Поскольку изменилось определение модели, выполним миграцию базы данных.

Далее добавим в код контроллера валидацию. Для этого изменим метод, обрабатывающий запросы POST:

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using Microsoft.EntityFrameworkCore;
4  using Microsoft.AspNetCore.Mvc;
5  using WebAPIApp.Models;
6  using System.Threading.Tasks;
7
8  namespace WebAPIApp.Controllers
9  {
10     [ApiController]
11     [Route("api/[controller]")]
12     public class UsersController : ControllerBase
13     {
14         UsersContext db;
15         public UsersController(UsersContext context)
16         {
17             db = context;
18             if (!db.Users.Any())
19             {
20                 db.Users.Add(new User { Name = "Tom", Age = 26 });
21                 db.Users.Add(new User { Name = "Alice", Age = 31 });
22                 db.SaveChanges();
23             }
24         }
25
26         [HttpGet]
27         public async Task<ActionResult<IEnumerable<User>>> Get()
28         {
29             return await db.Users.ToListAsync();
30         }
31

```

```

32         // GET api/users/5
33         [HttpGet("{id}")]
34         public async Task<ActionResult<User>> Get(int id)
35         {
36             User user = await db.Users.FirstOrDefaultAsync(x => x.Id == id);
37             if (user == null)
38                 return NotFound();
39             return new ObjectResult(user);
40         }
41
42         // POST api/users
43         [HttpPost]
44         public async Task<ActionResult<User>> Post(User user)
45         {
46             // обработка частных случаев валидации
47             if (user.Age == 99)
48                 ModelState.AddModelError("Age", "Возраст не должен быть равен 99");
49
50             if (user.Name == "admin")
51             {
52                 ModelState.AddModelError("Name", "Недопустимое имя пользователя - а");
53             }
54             // если есть лшибки - возвращаем ошибку 400
55             if (!ModelState.IsValid)
56                 return BadRequest(ModelState);
57
58             // если ошибок нет, сохраняем в базу данных
59             db.Users.Add(user);
60             await db.SaveChangesAsync();
61             return Ok(user);
62         }
63         // остальные методы
64     }
65 }

```

С помощью объекта `ModelState` здесь валидируется полученная модель `User`. Но кроме проверки свойства **`ModelState.IsValid`** мы также можем добавить и еще дополнительные проверки. Например:

```

1     if (user.Name == "admin")
2     {
3         ModelState.AddModelError("Name", "Недопустимое имя пользователя - admin");
4     }

```

Для добавления дополнительной ошибки используется метод `ModelState.AddModelError`, первый параметр которого - ключ ошибки, а второй - сообщение об ошибке. В качестве ключа мы можем использовать любое значение, но по умолчанию система сохраняет все ошибки свойств модели по ключу "Название_свойства". Поэтому все ошибки, связанные со свойством `Name`, сохраняются по ключу "Name". Причем по одному ключу мы можем указать множество ошибок.

Все ошибки валидации сохраняются в объекте `ModelState`, который передается в метод `BadRequest` и, таким образом, отправляется клиенту вместе с ошибкой 400.

Теперь рассмотрим, как мы можем получить эти ошибки на стороне клиента. Изменим код веб-страницы `index.html` следующим образом:

```
1      <!DOCTYPE html>
2      <html>
3      <head>
4          <meta charset="utf-8" />
5          <meta name="viewport" content="width=device-width" />
6          <title>Список пользователей</title>
7          <link href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css" rel="stylesheet">
8      </head>
9      <body>
10         <h2>Список пользователей</h2>
11         <div id="errors" class="alert alert-danger" style="display:none;"></div>
12         <form name="userForm">
13             <input type="hidden" name="id" value="0" />
14             <div class="form-group col-md-5">
15                 <label for="name">Имя:</label>
16                 <input class="form-control" name="name" />
17             </div>
18             <div class="form-group col-md-5">
19                 <label for="age">Возраст:</label>
20                 <input class="form-control" name="age" type="number" />
21             </div>
22             <div class="panel-body">
23                 <button type="submit" id="submit" class="btn btn-primary">Сохранить</button>
24                 <a id="reset" class="btn btn-primary">Сбросить</a>
25             </div>
26         </form>
27         <table class="table table-condensed table-striped" col-md-6">
28             <thead><tr><th>Id</th><th>Имя</th><th>Возраст</th><th></th></tr></thead>
29             <tbody>
30             </tbody>
```

```

31     </table>
32     <div>2019 © Metanit.com</div>
33     <script>
34         // Получение всех пользователей
35         async function GetUsers() {
36             // отправляет запрос и получаем ответ
37             const response = await fetch("/api/users", {
38                 method: "GET",
39                 headers: { "Accept": "application/json" }
40             });
41             // если запрос прошел нормально
42             if (response.ok === true) {
43                 // получаем данные
44                 const users = await response.json();
45                 let rows = document.querySelector("tbody");
46                 users.forEach(user => {
47                     // добавляем полученные элементы в таблицу
48                     rows.append(row(user));
49                 });
50             }
51         }
52         // Получение одного пользователя
53         async function GetUser(id) {
54             const response = await fetch("/api/users/" + id, {
55                 method: "GET",
56                 headers: { "Accept": "application/json" }
57             });
58             if (response.ok === true) {
59                 const user = await response.json();
60                 const form = document.forms["userForm"];
61                 form.elements["id"].value = user.id;
62                 form.elements["name"].value = user.name;
63                 form.elements["age"].value = user.age;
64             }
65         }
66         // Добавление пользователя
67         async function CreateUser(userName, userAge) {
68
69             const response = await fetch("api/users", {
70                 method: "POST",
71                 headers: { "Accept": "application/json", "Content-Type": "applicat
72                 body: JSON.stringify({
73                     name: userName,

```



```

74         age: parseInt(userAge, 10)
75     })
76 });
77 if (response.ok === true) {
78     const user = await response.json();
79     reset();
80     document.querySelector("tbody").append(row(user));
81 }
82 else {
83     const errorData = await response.json();
84     console.log("errors", errorData);
85     if (errorData) {
86         // ошибки вследствие валидации по атрибутам
87         if (errorData.errors) {
88             if (errorData.errors["Name"]) {
89                 addError(errorData.errors["Name"]);
90             }
91             if (errorData.errors["Age"]) {
92                 addError(errorData.errors["Age"]);
93             }
94         }
95         // кастомные ошибки, определенные в контроллере
96         // добавляем ошибки свойства Name
97         if (errorData["Name"]) {
98             addError(errorData["Name"]);
99         }
100
101         // добавляем ошибки свойства Age
102         if (errorData["Age"]) {
103             addError(errorData["Age"]);
104         }
105     }
106
107     document.getElementById("errors").style.display = "block";
108 }
109 }
110 // Изменение пользователя
111 async function EditUser(userId, userName, userAge) {
112     const response = await fetch("api/users", {
113         method: "PUT",
114         headers: { "Accept": "application/json", "Content-Type": "applicat
115         body: JSON.stringify({
116             id: parseInt(userId, 10),

```

```
117         name: userName,
118         age: parseInt(userAge, 10)
119     })
120 });
121 if (response.ok === true) {
122     const user = await response.json();
123     reset();
124     document.querySelector("tr[data-rowid='" + user.id + "']").replace
125 }
126 }
127 // Удаление пользователя
128 async function DeleteUser(id) {
129     const response = await fetch("/api/users/" + id, {
130         method: "DELETE",
131         headers: { "Accept": "application/json" }
132     });
133     if (response.ok === true) {
134         const user = await response.json();
135         document.querySelector("tr[data-rowid='" + user.id + "']").remove()
136     }
137 }
138
139 // сброс формы
140 function reset() {
141     const form = document.forms["userForm"];
142     form.reset();
143     form.elements["id"].value = 0;
144 }
145 function addError(errors) {
146     errors.forEach(error => {
147         const p = document.createElement("p");
148         p.append(error);
149         document.getElementById("errors").append(p);
150     });
151 }
152 // создание строки для таблицы
153 function row(user) {
154
155     const tr = document.createElement("tr");
156     tr.setAttribute("data-rowid", user.id);
157
158     const idTd = document.createElement("td");
159     idTd.append(user.id);
```

```
160         tr.append(idTd);
161
162         const nameTd = document.createElement("td");
163         nameTd.append(user.name);
164         tr.append(nameTd);
165
166         const ageTd = document.createElement("td");
167         ageTd.append(user.age);
168         tr.append(ageTd);
169
170         const linksTd = document.createElement("td");
171
172         const editLink = document.createElement("a");
173         editLink.setAttribute("data-id", user.id);
174         editLink.setAttribute("style", "cursor:pointer;padding:15px;");
175         editLink.append("Изменить");
176         editLink.addEventListener("click", e => {
177
178             e.preventDefault();
179             GetUser(user.id);
180         });
181         linksTd.append(editLink);
182
183         const removeLink = document.createElement("a");
184         removeLink.setAttribute("data-id", user.id);
185         removeLink.setAttribute("style", "cursor:pointer;padding:15px;");
186         removeLink.append("Удалить");
187         removeLink.addEventListener("click", e => {
188
189             e.preventDefault();
190             DeleteUser(user.id);
191         });
192
193         linksTd.append(removeLink);
194         tr.appendChild(linksTd);
195
196         return tr;
197     }
198     // сброс значений формы
199     document.getElementById("reset").addEventListener("click", function (e) {
200
201         e.preventDefault();
202         reset();
```

```

203         })
204
205         // отправка формы
206         document.forms["userForm"].addEventListener("submit", e => {
207             e.preventDefault();
208             document.getElementById("errors").innerHTML="";
209             document.getElementById("errors").style.display = "none";
210
211             const form = document.forms["userForm"];
212             const id = form.elements["id"].value;
213             const name = form.elements["name"].value;
214             const age = form.elements["age"].value;
215             if (id == 0)
216                 CreateUser(name, age);
217             else
218                 EditUser(id, name, age);
219         });
220
221         // загрузка пользователей
222         GetUsers();
223
224     </script>
225 </body>
226 </html>

```

Для вывода ошибок здесь определен специальный блок с `id="errors"`. При получении ошибки в функции `CreateUser()` мы получаем данные, посланные через объект `ModelState`.

```

1     if (errorData) {
2         const errorData = await response.json();
3         console.log("errors", errorData);
4         (errorData) {
5             // ошибки вследствие валидации по атрибутам
6             if (errorData.errors) {
7                 if (errorData.errors["Name"]) {
8                     addError(errorData.errors["Name"]);
9                 }
10                if (errorData.errors["Age"]) {
11                    addError(errorData.errors["Age"]);
12                }
13            }
14            // кастомные ошибки, определенные в контроллере

```

```
15         // добавляем ошибки свойства Name
16         if (errorData["Name"]) {
17             addError(errorData["Name"]);
18         }
19         // добавляем ошибки свойства Age
20         if (errorData["Age"]) {
21             addError(errorData["Age"]);
22         }
23     }
24     document.getElementById("errors").style.display = "block";
25 }
```

Но чтобы обратиться к ошибкам, надо пройти несколько уровней вложенности. Ошибки, которые добавляются в результате применения правил атрибутов валидации, можно получить из объекта `errorData.errors`. Например, чтобы получить ошибки свойства `Age`, придется использовать вызов `errorData.errors["Age"]`. Получение сообщения об ошибках, которые были определены в контроллере, производится непосредственно из посланного объекта `errorData["Age"]`. Причем каждый из таких вызовов представляет собой массив.

И теперь если мы введем некорректные данные, мы получим сообщения об ошибках.

Список пользователей

localhost:54427/index.html

Список пользователей

Недопустимое имя пользователя - admin

Возраст не должен быть равен 99

Имя:

admin

Возраст:

99

Сохранить

Сбросить

Id	Имя	возраст	
1	Tom	26	Изменить Удалить
2	Alice	31	Изменить Удалить

Content negotiation

Когда метод контроллера возвращает ответ, то инфраструктура MVC определяет, в каком формате этот ответ лучше отправить клиенту. Формат контента зависит от ряда факторов: какой формат принимает клиент, какой формат может генерировать MVC, политика форматирования, возвращаемый методом тип.

Иногда метод контроллера возвращает результат в конкретном формате. Например, с помощью метода `Content` мы можем вернуть объект `string`, а с помощью метода `Json()` можно вернуть объект, сериализованный в формат `json`. Однако контроллеры в Web API для возвращения результат а используют самые различные методы, а не только `Json` и `Content`. Например:

```
1    [HttpGet]
2    public IEnumerable<User> Get()
3    {
4        return db.Users.ToList();
5    }
6
7    [HttpGet("{id}")]
8    public IActionResult Get(int id)
9    {
10        User user = db.Users.FirstOrDefault(x => x.Id == id);
11        if (user == null)
12            return NotFound();
13        return new ObjectResult(user);
14    }
15
16    [HttpDelete("{id}")]
17    public IActionResult Delete(int id)
18    {
19        User user = db.Users.FirstOrDefault(x => x.Id == id);
20        if (user==null)
```

```

21      {
22          return NotFound();
23      }
24      db.Users.Remove(user);
25      db.SaveChanges();
26      return Ok(user);
27  }

```

Первый метод `Get` возвращает коллекция элементов `IEnumerable<User>`. Однако здесь следует правило, если метод возвращает стандартный РОСО-объект, то на самом деле неявно возвращается объект **ObjectResult**, в который оборачивается РОСО-объект. То есть в случае с методом `Get` мы могли бы написать:

```

1  [HttpGet]
2  public IActionResult Get()
3  {
4      return new ObjectResult(db.Users.ToList());
5  }

```

Когда же мы передаем в метод, который отправляет определенный статусный код, например, в метод `Ok()`, то он также возвращает объект класса `ObjectResult`, точнее один из наследников этого класса. Например, выражение `return Ok(user);` возвращает объект `OkObjectResult`.

Использование типа `ObjectResult` во всех случаях имеет важное значение, так как он реализует такую функциональность, как **Content negotiation**.

Content negotiation предполагает процесс согласования между сервером и клиентом по поводу типа контента, который отправляется клиенту.

Если метод возвращает ответ в виде строки, то есть объекта `string`, эта строка отправляется клиенту как есть, а для заголовка `Content-Type` устанавливается значение `text/plain`. Данные простейших типов, как `int` или `DateTime`, при отправке также форматируются в строку.

А для объектов классов отправляемые данные в `ObjectResult` по умолчанию форматируются в формат JSON, а для заголовка `Content-Type` устанавливается значение `application/json`.

Заголовок Accept

Большинство браузеров включают в запрос заголовок `Accept`, который указывает набор форматов, предпочтительных для получения ответа. В частности, браузер Google Chrome в запросе отправляет следующий заголовок `Accept`:

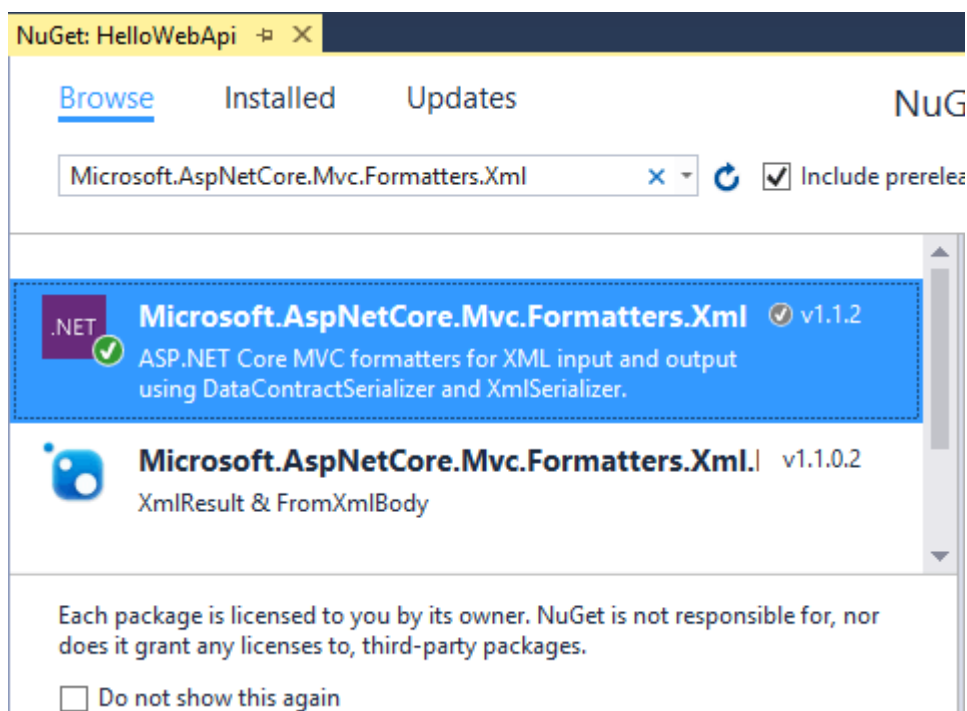
```
1 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=
```

Этот заголовок указывает, что предпочтительными форматами для браузера являются HTML, XHTML, XML и WEBP (для изображений). Значение `q`, которое может указываться после каждого формата, обозначает вес данного формата. По умолчанию, если `q` не указан, то каждый формат имеет вес 1.0. А, к примеру, значение `q = 0.9` для формата `application/xml` указывает, что браузер принимает данные в XML, но предпочитает форматы HTML и XHTML (для которых по умолчанию вес 1.0). Две звездочки в конце `/*/*` говорят о том, что браузер может принимать любой формат, но в соответствии с его весом `q = 0.8` он будет менее предпочтительным по сравнению с ранее указанными форматами.

То есть данным заголовком Google Chrome говорит, что он предпочитает данные в формате HTML или XHTML, а изображения - в формате WEBP. Если форматы HTML и XHTML недоступны, то можно отправить данные в XML. Если же ни один из этих форматов недоступен для отправки ответа, тогда браузер может принять ответ в любом формате.

Форматирование ответа в xml

Добавим через NuGet в проект пакет **Microsoft.AspNetCore.Mvc.Formatters.Xml**:



Добавим форматировщик `XmlSerializerFormatters` к сервисам MVC в классе `Startup.cs`:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddMvc().AddXmlDataContractSerializerFormatters();
4
5     // остальной код метода
6 }
```

В качестве альтернативы можно также использовать другой способ:

```
1 services.AddMvc(options =>
2 {
3     options.OutputFormatters.Add(new XmlDataContractSerializerOutputFormatter());
4 });
```

Вне зависимости от того, какой из этих двух способов будет применяться, для сериализации ответа будет использоваться класс **`System.Runtime.Serialization.DataContractSerializer`**.

В качестве альтернативы для сериализации данных в xml можно использовать класс **`System.Xml.Serialization.XmlSerializer`**:

```
1 public void ConfigureServices(IServiceCollection services)
2 {
3     services.AddMvc().AddXmlSerializerFormatters();
4
5     // альтернативный способ
6     // services.AddMvc(options =>
7     // {
8     //     options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
9     // });
10
```

```
11         // остальной код метода
12     }
```

В данном случае разница

между `XmlDataContractSerializerOutputFormatter` и `XmlSerializerOutputFormatter` будет небольшая за тем исключением, что `XmlSerializerOutputFormatter` на клиентские приложения на .NET, которые используют старые версии фреймворка при работе с XML.

В этом случае, если клиент будет отправлять в запросе, например, к методу:

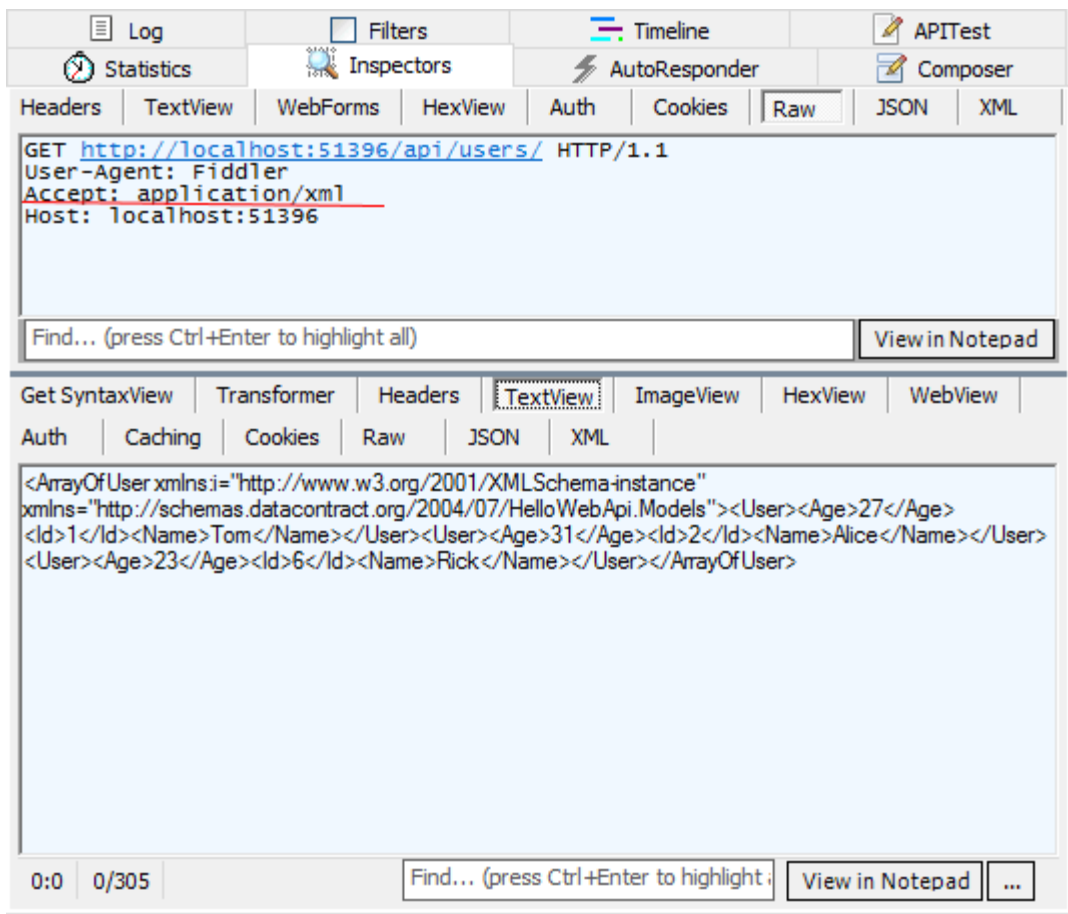
```
1     [HttpGet]
2     public IEnumerable<User> Get()
3     {
4         return db.Users.ToList();
5     }
```

следующий заголовок

```
1     Accept: application/xml
```

То сервер будет отправлять данные в формате xml.

Например, обратимся из Fiddlera к методу `Get`, отправив в запросе заголовок "Accept: application/xml":



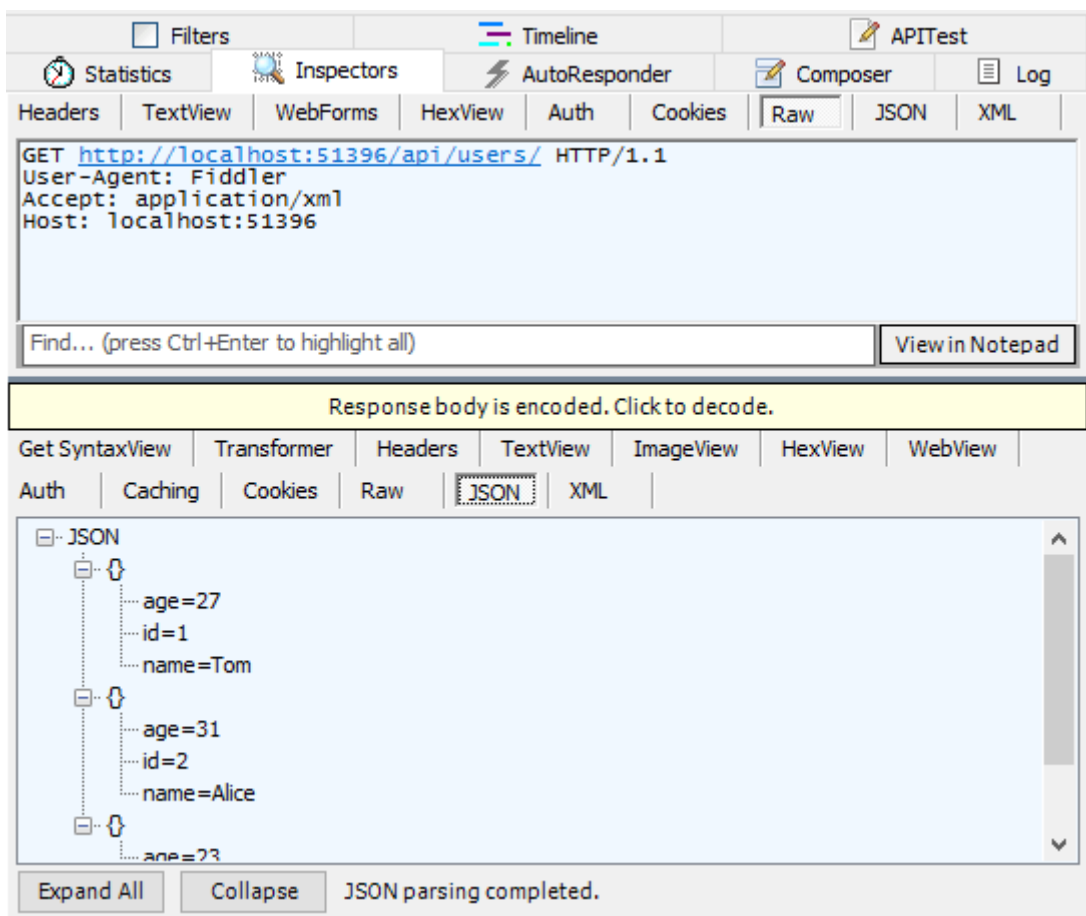
Если бы мы не указали заголовок Ассерпт, то данные по прежнему передавались бы в формате json.

Атрибут Produces

Мы можем переопределить систему согласования типа контента с помощью атрибута **Produces**. Этот атрибут выступает своего рода фильтром, который изменяет тип контента для объекта `ObjectResult`. В качестве значения в этот атрибут передается тип содержимого:

```
1 [HttpGet]
2 [Produces("application/json")]
3 public IEnumerable<User> Get()
4 {
5     return db.Users.ToList();
6 }
```

В данном случае, даже если приложение использует сериализацию ответа в формат XML, а клиент в запросе указывает заголовок "Ассерп: application/xml", данные все равно будут отправляться в формате json.



Получение типа контента из строки запроса

Заголовок Ассерп - не единственный способ указать серверу, в каком формате надо отправлять данные клиенту. Еще один способ представляет использование строки запроса. Так, изменим в классе Startup добавление сервисов MVC:

```
1 using Microsoft.AspNetCore.Builder;
2 using Microsoft.Extensions.DependencyInjection;
3 using Microsoft.EntityFrameworkCore;
4 using HelloWebApi.Models;
5 using Microsoft.Net.Http.Headers;
6
7 namespace HelloWebApi
```

```

8      {
9          public class Startup
10         {
11             public void ConfigureServices(IServiceCollection services)
12             {
13                 string con =
14 "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted_Connection=True;Multi
15                 services.AddDbContext<UsersContext>(options => options.UseSqlServer(con
16
17                 services.AddMvc()
18                     .AddXmlDataContractSerializerFormatters()
19                     .AddMvcOptions(opts => {
20                         opts.FormatterMappings.SetMediaTypeMappingForFormat("xml", new M
21                     });
22             }
23
24             public void Configure(IApplicationBuilder app)
25             {
26                 app.UseDefaultFiles();
27                 app.UseStaticFiles();
28                 app.UseMvc();
29             }
30         }
31     }

```

В методе `AddMvcOptions()` устанавливается сопоставление формата из строки запроса с определенным медиа-типом:

```

1     opts.FormatterMappings.SetMediaTypeMappingForFormat("xml", new MediaTypeHeaderValue("xml"));

```

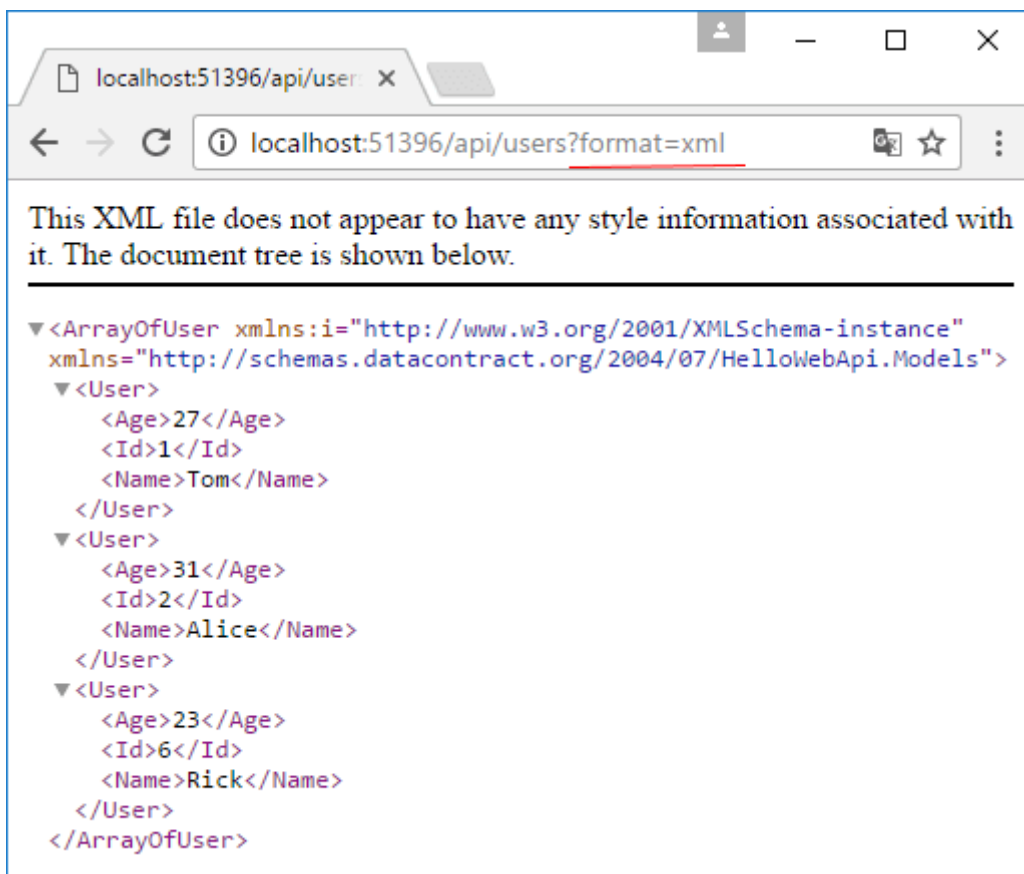
Теперь к методу контроллера надо применить атрибут **FormatFilter**:

```

1  [HttpGet]
2  [FormatFilter]
3  public IEnumerable<User> Get()
4  {
5      return db.Users.ToList();
6  }

```

При запросе мы можем указать формат xml, передав в строке запроса параметр `format=xml`:



Атрибут `FormatFilter` может применяться как к отдельным методам, так и ко всему контроллеру.

При этом серверу уже не важно, какой формат контента передается в запросе в заголовке `Ассерп` - он все равно не будет учитываться.