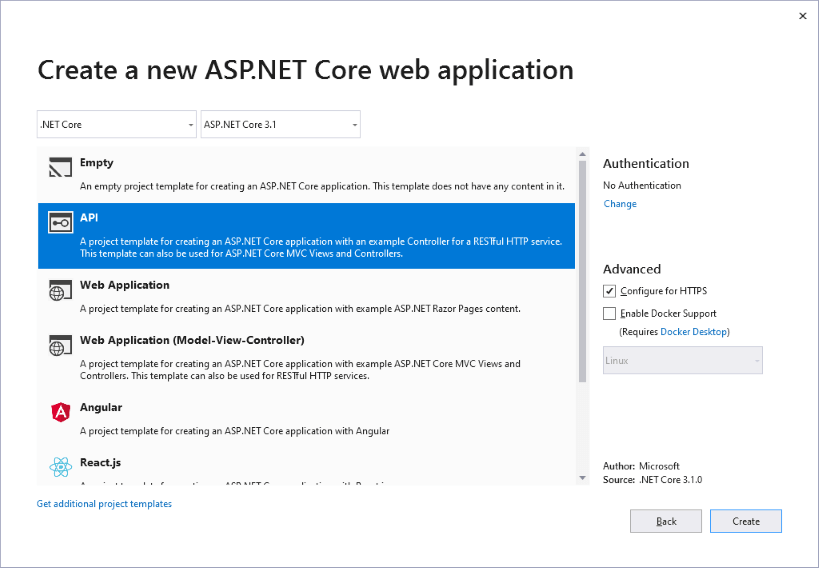
## Разработка Web API

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:

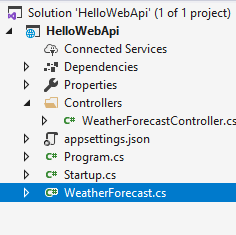
* GET
* POST
* PUT
* DELETE

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:

[?](https://metanit.com/sharp/aspnet5/23.1.php)

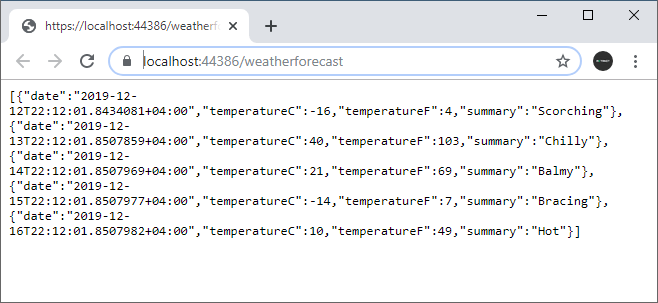
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Extensions.Logging;    namespace HelloWebApi.Controllers  {      [ApiController]      [Route("[controller]")]      public class WeatherForecastController : ControllerBase      {          private static readonly string[] Summaries = new[]          {              "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"          };            private readonly ILogger<WeatherForecastController> \_logger;            public WeatherForecastController(ILogger<WeatherForecastController> logger)          {              \_logger = logger;          }            [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();          }      }  } |

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:

[?](https://metanit.com/sharp/aspnet5/23.1.php)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Builder;  using Microsoft.AspNetCore.Hosting;  using Microsoft.AspNetCore.HttpsPolicy;  using Microsoft.AspNetCore.Mvc;  using Microsoft.Extensions.Configuration;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.Extensions.Hosting;  using Microsoft.Extensions.Logging;    namespace HelloWebApi  {      public class Startup      {          public Startup(IConfiguration configuration)          {              Configuration = configuration;          }            public IConfiguration Configuration { get; }            public void ConfigureServices(IServiceCollection services)          {              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();              });          }      }  } |

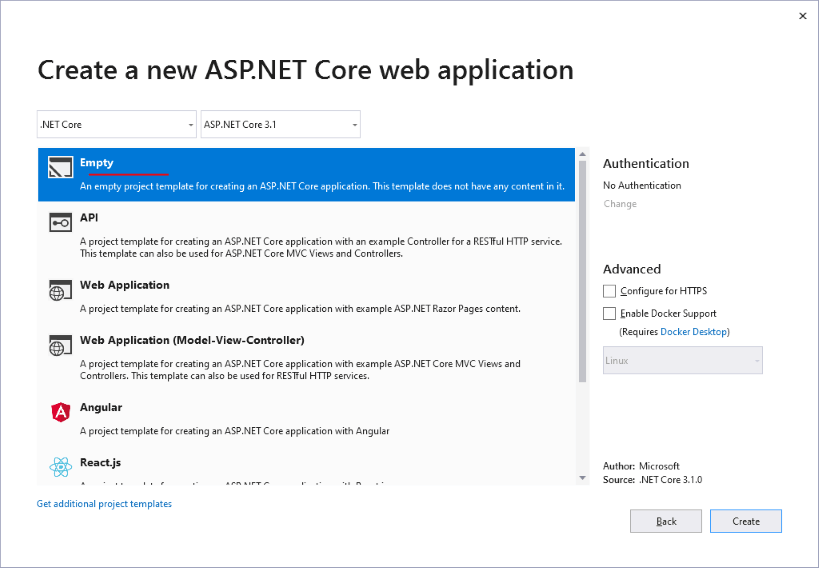
Прежде всего, поскольку в данном случае не используются представления, то подключение в методе ConfigureServices() сервисов MVC, необходимых для работы контроллеров Web API производится с помощью метода services.AddControllers()

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 конкректные 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:

[?](https://metanit.com/sharp/aspnet5/23.2.php)

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

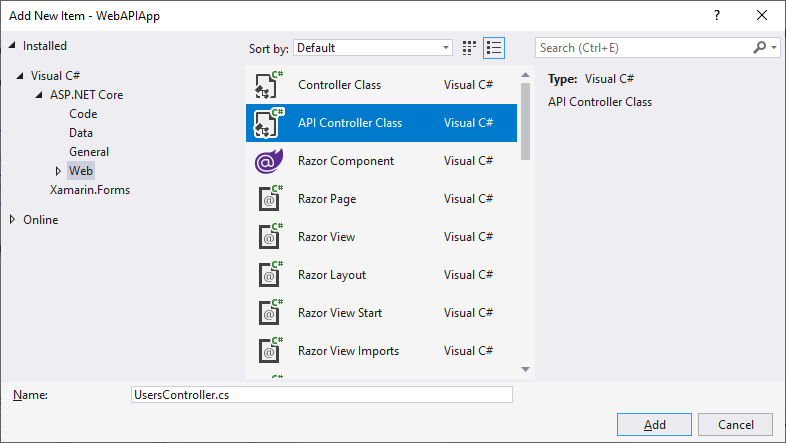
Для взаимодействия с MS SQL Server через Entity Framework через пакетный менеджер Nuget добавим в проект пакет Microsoft.EntityFrameworkCore.SqlServer.

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

[?](https://metanit.com/sharp/aspnet5/23.2.php)

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

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 UsersController.

После его создания изменим его код следующим образом:

[?](https://metanit.com/sharp/aspnet5/23.2.php)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88 | using System.Collections.Generic;  using System.Linq;  using Microsoft.EntityFrameworkCore;  using Microsoft.AspNetCore.Mvc;  using WebAPIApp.Models;  using System.Threading.Tasks;    namespace WebAPIApp.Controllers  {      [ApiController]      [Route("api/[controller]")]      public class UsersController : ControllerBase      {          UsersContext db;          public UsersController(UsersContext context)          {              db = context;              if (!db.Users.Any())              {                  db.Users.Add(new User { Name = "Tom", Age = 26 });                  db.Users.Add(new User { Name = "Alice", Age = 31 });                  db.SaveChanges();              }          }            [HttpGet]          public async Task<ActionResult<IEnumerable<User>>> Get()          {              return await db.Users.ToListAsync();          }            // GET api/users/5          [HttpGet("{id}")]          public async Task<ActionResult<User>> Get(int id)          {              User user = await db.Users.FirstOrDefaultAsync(x => x.Id == id);              if (user == null)                  return NotFound();              return new ObjectResult(user);          }            // POST api/users          [HttpPost]          public async Task<ActionResult<User>> Post(User user)          {              if (user == null)              {                  return BadRequest();              }                db.Users.Add(user);              await db.SaveChangesAsync();              return Ok(user);          }            // PUT api/users/          [HttpPut]          public async Task<ActionResult<User>> Put(User user)          {              if (user == null)              {                  return BadRequest();              }              if (!db.Users.Any(x => x.Id ==user.Id))              {                  return NotFound();              }                db.Update(user);              await db.SaveChangesAsync();              return Ok(user);          }            // DELETE api/users/5          [HttpDelete("{id}")]          public async Task<ActionResult<User>> Delete(int id)          {              User user = db.Users.FirstOrDefault(x => x.Id == id);              if (user == null)              {                  return NotFound();              }              db.Users.Remove(user);              await db.SaveChangesAsync();              return Ok(user);          }      }  } |

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.

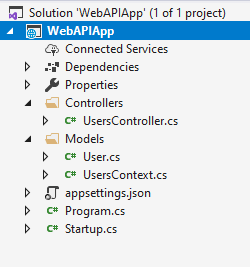
Если запрос Get содержит параметр id (идентификатор объекта), то он обрабатывается другим методом - Get(int id), который возвращает объект по переданному id.

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:

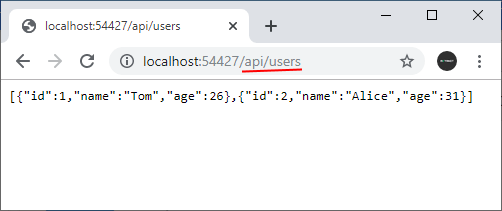
[?](https://metanit.com/sharp/aspnet5/23.2.php)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using Microsoft.AspNetCore.Builder;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.EntityFrameworkCore;  using WebAPIApp.Models;    namespace WebAPIApp  {      public class Startup      {          public void ConfigureServices(IServiceCollection services)          {              string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted\_Connection=True;";              // устанавливаем контекст данных              services.AddDbContext<UsersContext>(options => options.UseSqlServer(con));                services.AddControllers(); // используем контроллеры без представлений          }            public void Configure(IApplicationBuilder app)          {              app.UseDeveloperExceptionPage();                app.UseRouting();                app.UseEndpoints(endpoints =>              {                  endpoints.MapControllers(); // подключаем маршрутизацию на контроллеры              });          }      }  } |

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:



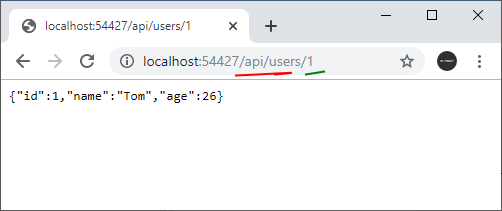
Поскольку запрос из адресной строки браузера представляет GET-запрос, то его будет обрабатывать метод

[?](https://metanit.com/sharp/aspnet5/23.2.php)

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

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 запос типа Get, но теперь также передается параметр id, то сработает следующий метод:

[?](https://metanit.com/sharp/aspnet5/23.2.php)

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

## 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.

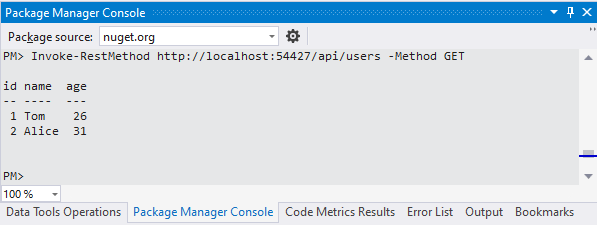
Для тестирования контроллера Web API можно применять специальные инструменты, которые устанавливаются в виде отдельных приложений, либо в виде расширений для браузеров, например, Fiddler или Postman. Однако Visual Studio предоставляет еще один способ - ввод команд в окне Package Manager Console, которое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

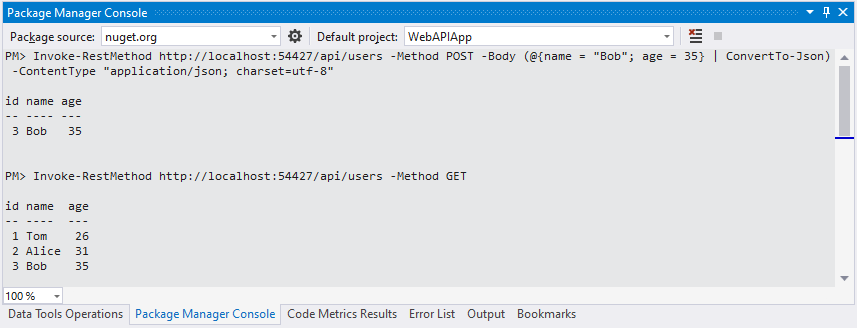
Аргумент -Body в этой команде указывает на тело запроса - тот объект, который будет отправляться на сервер. В моем случае это объект класса User, поэтому все его значения имеют ключи, которые соответствуют названиям свойств класса User. Кроме того, указываем с помощью флага ConvertTo-Json, что объект 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

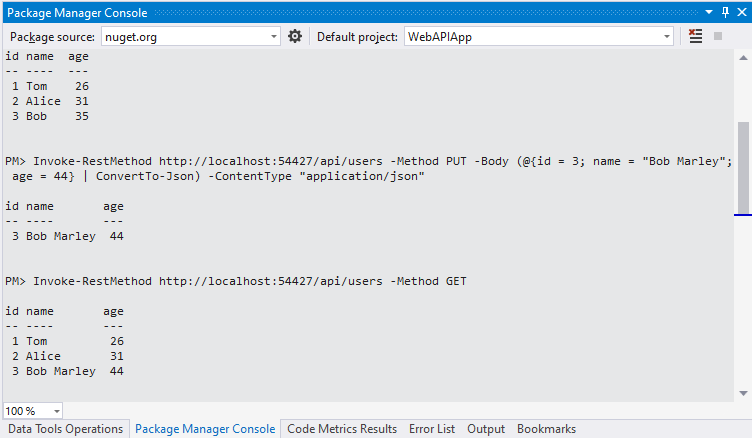
После этого мы можем повторно получить список пользователей и увидеть в нем добавленный объект:



#### Тестирование PUT-запросов

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

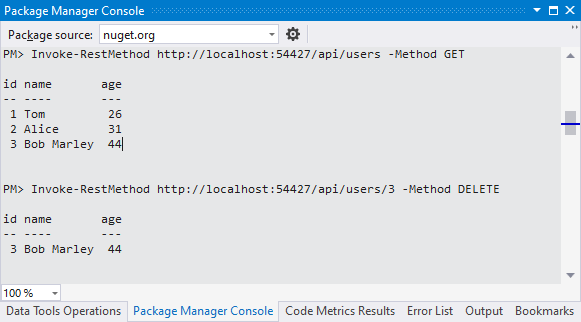
Invoke-RestMethod http://localhost:54427/api/users -Method PUT -Body (@{id = 3; name = "Bob Marley"; age = 44} | ConvertTo-Json) -ContentType "application/json"



#### Тестирование запросов DELETE

В запросе DELETE необходимо передать id удаляемого объекта:

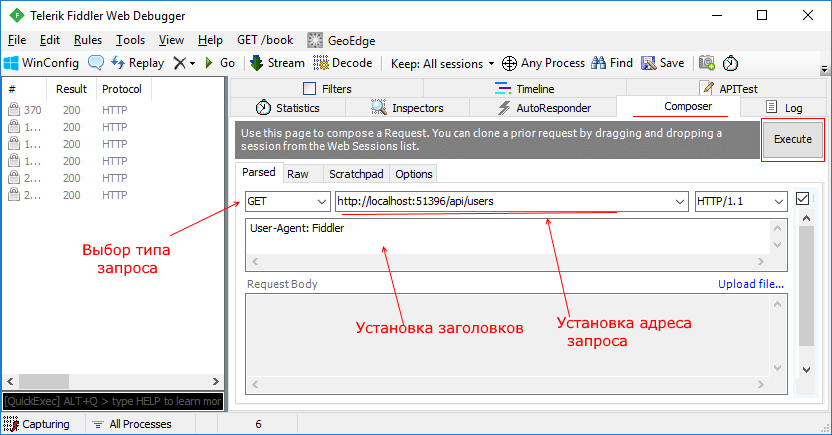
Invoke-RestMethod http://localhost:54427/api/users/3 -Method DELETE



### Fiddler

Теперь рассмотрим, как использовать [Fiddler](http://www.telerik.com/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:

