**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:

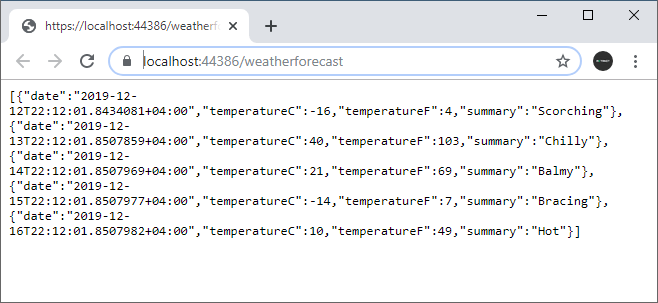
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Threading.Tasks;  using Microsoft.AspNetCore.Mvc;  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:

|  |  |
| --- | --- |
| 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();              });          }      }  } |

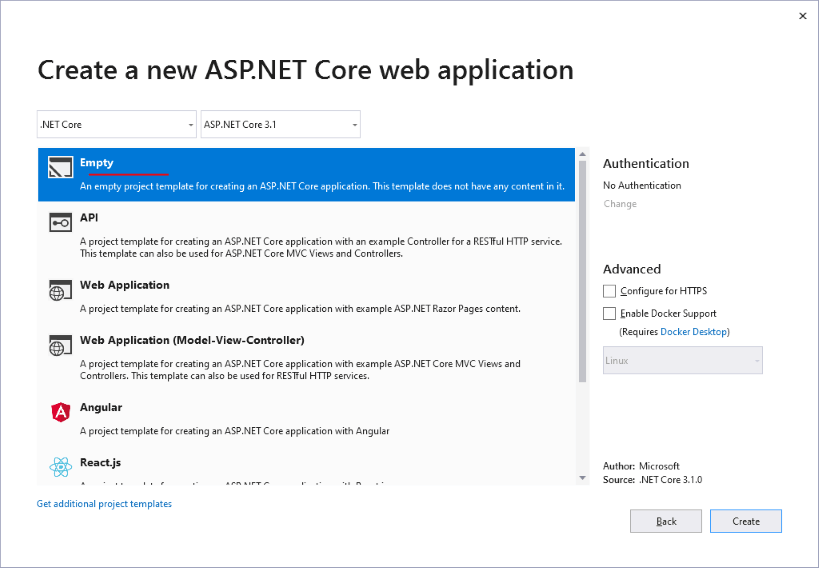
Прежде всего, поскольку в данном случае не используются представления, то подключение в методе ConfigureServices() сервисов MVC, необходимых для работы контроллеров Web API производится с помощью метода **services.AddControllers()**

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:

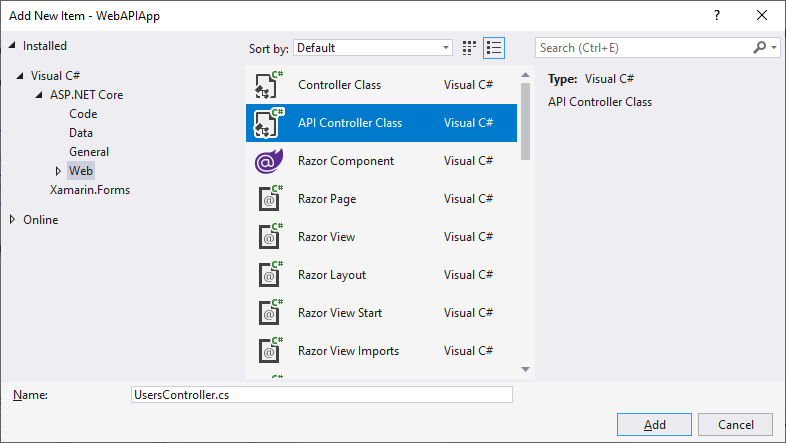
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class User  {      public int Id { get; set; }      public string Name { get; set; }      public int Age { get; set; }  } |

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:

|  |  |
| --- | --- |
| 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:

|  |  |
| --- | --- |
| 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.

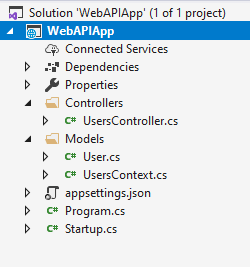
Если запрос Get содержит параметр id (идентификатор объекта), то он обрабатывается другим методом - Get(int id), который возвращает объект по переданному id.

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:



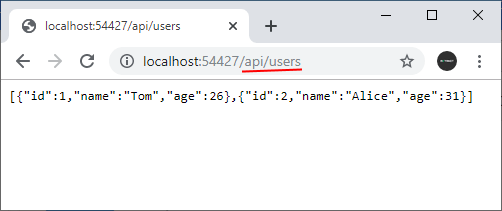
Теперь, чтобы это все использовать, изменим код класса Startup:

|  |  |
| --- | --- |
| 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*:

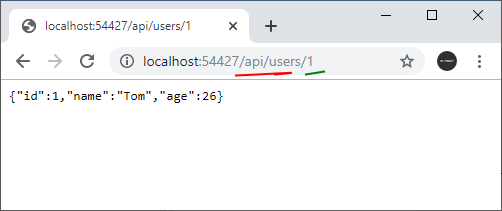


Поскольку запрос из адресной строки браузера представляет GET-запрос, то его будет обрабатывать метод

|  |  |
| --- | --- |
| 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:

|  |  |
| --- | --- |
| 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.

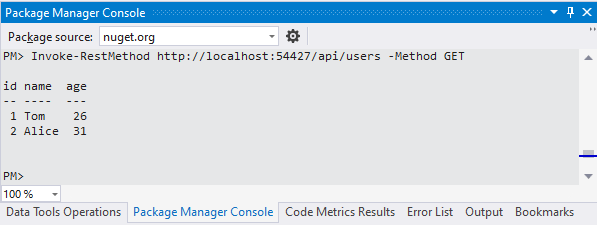
Для тестирования контроллера Web API можно применять специальные инструменты, которые устанавливаются в виде отдельных приложений, либо в виде расширений для браузеров, например, **Fiddler** или **Postman**. Однако Visual Studio предоставляет еще один способ - ввод команд в окне **Package Manager Console**,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

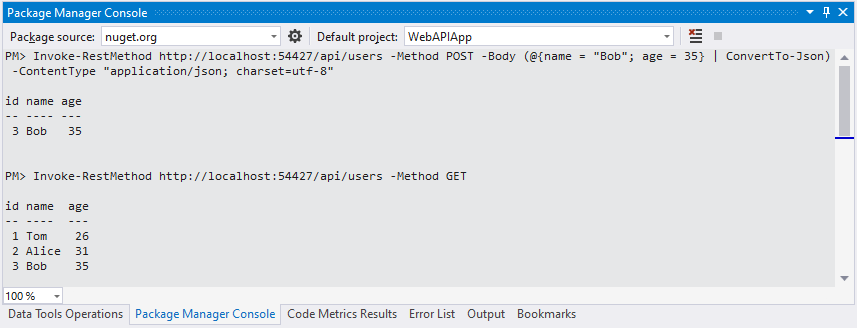
Аргумент -Body в этой команде указывает на тело запроса - тот объект, который будет отправляться на сервер. В моем случае это объект класса User, поэтому все его значения имеют ключи, которые соответствуют названиям свойств класса User. Кроме того, указываем с помощью флага ConvertTo-Json, что объект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

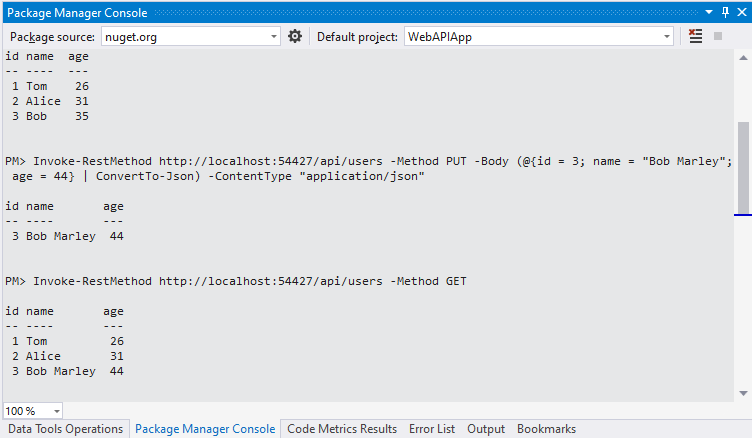
После этого мы можем повторно получить список пользователей и увидеть в нем добавленный объект:



#### Тестирование PUT-запросов

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

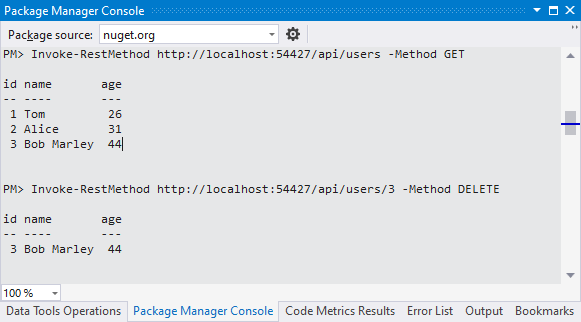
Invoke-RestMethod http://localhost:54427/api/users -Method PUT -Body (@{id = 3; name = "Bob Marley"; age = 44} | ConvertTo-Json) -ContentType "application/json"



#### Тестирование запросов DELETE

В запросе DELETE необходимо передать id удаляемого объекта:

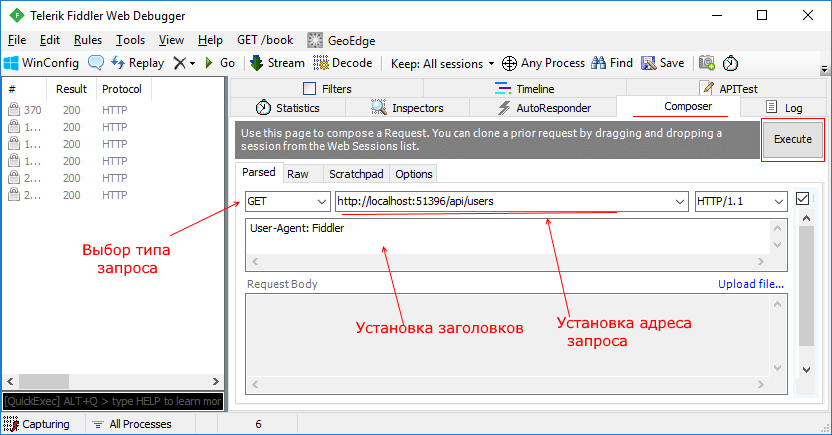
Invoke-RestMethod http://localhost:54427/api/users/3 -Method DELETE



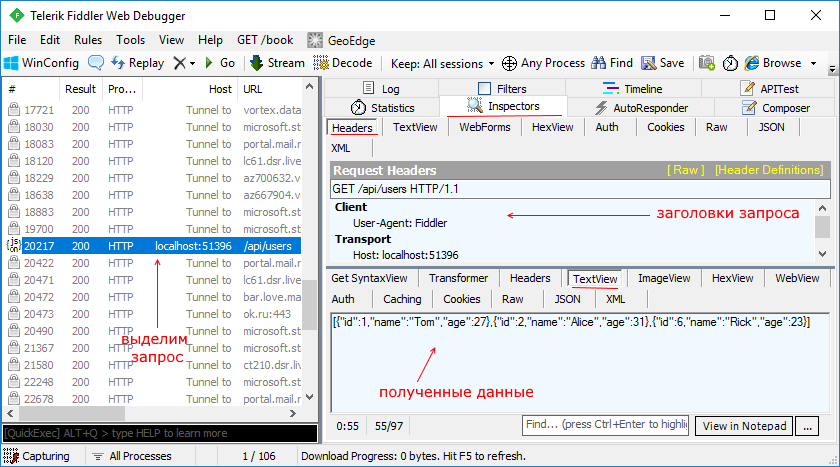
### Fiddler

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/*.



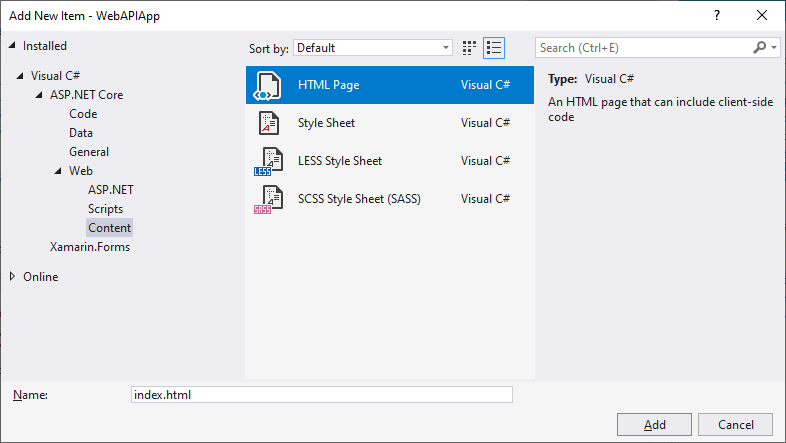
После ввода адреса нажмем на кнопку Execute. После осуществления запроса в левом поле-списке запросов выберем сделанный только что запрос, и на вкладке **Inspectors** можно будет увидеть результат запроса - список пользователей, который Fiddler получает в сериализованном виде:



## Создание клиента для WEB API

Продолжим работу с проектом из прошлой темы. 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  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 | 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.UseDefaultFiles();              app.UseStaticFiles();                app.UseRouting();                app.UseEndpoints(endpoints =>              {                  endpoints.MapControllers();              });          }      }  } |

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

|  |  |
| --- | --- |
| 1  2 | app.UseDefaultFiles();  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  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  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>Список пользователей</title>      <link href="<https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css>" rel="stylesheet" />  </head>  <body>      <h2>Список пользователей</h2>      <form name="userForm">          <input type="hidden" name="id" value="0" />          <div class="form-group col-md-5">              <label for="name">Имя:</label>              <input class="form-control" name="name" />          </div>          <div class="form-group col-md-5">              <label for="age">Возраст:</label>              <input class="form-control" name="age" type="number" />          </div>          <div class="panel-body">              <button type="submit" id="submit" class="btn btn-primary">Сохранить</button>              <a id="reset" class="btn btn-primary">Сбросить</a>          </div>      </form>      <table class="table table-condensed table-striped  col-md-6">          <thead><tr><th>Id</th><th>Имя</th><th>возраст</th><th></th></tr></thead>          <tbody>          </tbody>      </table>      <div>2019 © Metanit.com</div>      <script>          // Получение всех пользователей          async function GetUsers() {              // отправляет запрос и получаем ответ              const response = await fetch("/api/users", {                  method: "GET",                  headers: { "Accept": "application/json" }              });              // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const users = await response.json();                  let rows = document.querySelector("tbody");                  users.forEach(user => {                      // добавляем полученные элементы в таблицу                      rows.append(row(user));                  });              }          }          // Получение одного пользователя          async function GetUser(id) {              const response = await fetch("/api/users/" + id, {                  method: "GET",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  const form = document.forms["userForm"];                  form.elements["id"].value = user.id;                  form.elements["name"].value = user.name;                  form.elements["age"].value = user.age;              }          }          // Добавление пользователя          async function CreateUser(userName, userAge) {                const response = await fetch("api/users", {                  method: "POST",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  reset();                  document.querySelector("tbody").append(row(user));              }          }          // Изменение пользователя          async function EditUser(userId, userName, userAge) {              const response = await fetch("api/users", {                  method: "PUT",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      id: parseInt(userId, 10),                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  reset();                  document.querySelector("tr[data-rowid='" + user.id + "']").replaceWith(row(user));              }          }          // Удаление пользователя          async function DeleteUser(id) {              const response = await fetch("/api/users/" + id, {                  method: "DELETE",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector("tr[data-rowid='" + user.id + "']").remove();              }          }            // сброс формы          function reset() {              const form = document.forms["userForm"];              form.reset();              form.elements["id"].value = 0;          }          // создание строки для таблицы          function row(user) {                const tr = document.createElement("tr");              tr.setAttribute("data-rowid", user.id);                const idTd = document.createElement("td");              idTd.append(user.id);              tr.append(idTd);                const nameTd = document.createElement("td");              nameTd.append(user.name);              tr.append(nameTd);                const ageTd = document.createElement("td");              ageTd.append(user.age);              tr.append(ageTd);                const linksTd = document.createElement("td");                const editLink = document.createElement("a");              editLink.setAttribute("data-id", user.id);              editLink.setAttribute("style", "cursor:pointer;padding:15px;");              editLink.append("Изменить");              editLink.addEventListener("click", e => {                    e.preventDefault();                  GetUser(user.id);              });              linksTd.append(editLink);                const removeLink = document.createElement("a");              removeLink.setAttribute("data-id", user.id);              removeLink.setAttribute("style", "cursor:pointer;padding:15px;");              removeLink.append("Удалить");              removeLink.addEventListener("click", e => {                    e.preventDefault();                  DeleteUser(user.id);              });                linksTd.append(removeLink);              tr.appendChild(linksTd);                return tr;          }          // сброс значений формы          document.getElementById("reset").click(function (e) {                e.preventDefault();              reset();          })            // отправка формы          document.forms["userForm"].addEventListener("submit", e => {              e.preventDefault();              const form = document.forms["userForm"];              const id = form.elements["id"].value;              const name = form.elements["name"].value;              const age = form.elements["age"].value;              if (id == 0)                  CreateUser(name, age);              else                  EditUser(id, name, age);          });            // загрузка пользователей          GetUsers();        </script>  </body>  </html> |

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

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

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

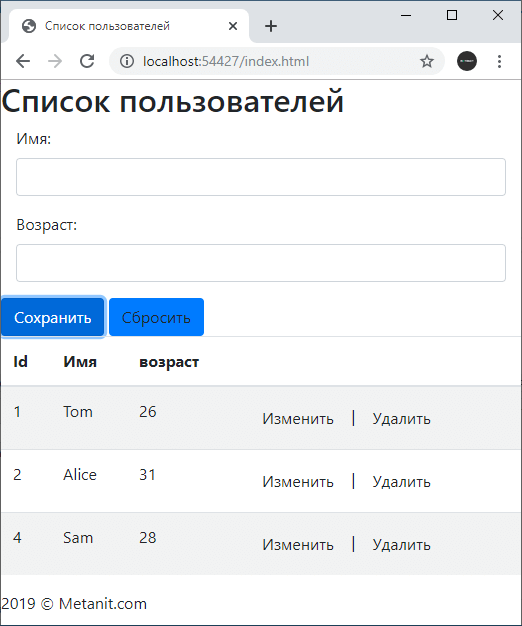
И выделенный пользователь добавляется в форму над таблицей. Эта же форма применяется и для добавления объекта. С помощью скрытого поля, которое хранит id пользователя, мы можем узнать, какое действие выполняется - добавление или редактирование. Если id равен 0, то выполняется функция CreateUser, которая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

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

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

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

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



## 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 Unathorized - 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | using System;  using System.ComponentModel.DataAnnotations;    namespace WebAPIApp.Models  {      public class User      {          public int Id { get; set; }          [Required(ErrorMessage = "Укажите имя пользователя")]          public string Name { get; set; }          [Range(1, 100, ErrorMessage = "Возраст должен быть в промежутке от 1 до 100")]          [Required(ErrorMessage = "Укажите возраст пользователя")]          public int Age { get; set; }      }  } |

Поскольку изменилось определение модели, выполним миграцию базы данных.

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

|  |  |
| --- | --- |
| 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 | 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)          {              // обработка частных случаев валидации              if (user.Age == 99)                  ModelState.AddModelError("Age", "Возраст не должен быть равен 99");                if (user.Name == "admin")              {                  ModelState.AddModelError("Name", "Недопустимое имя пользователя - admin");              }              // если есть лшибки - возвращаем ошибку 400              if (!ModelState.IsValid)                  return BadRequest(ModelState);                // если ошибок нет, сохраняем в базу данных              db.Users.Add(user);              await db.SaveChangesAsync();              return Ok(user);          }          // остальные методы      }  } |

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

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 валидаци 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:

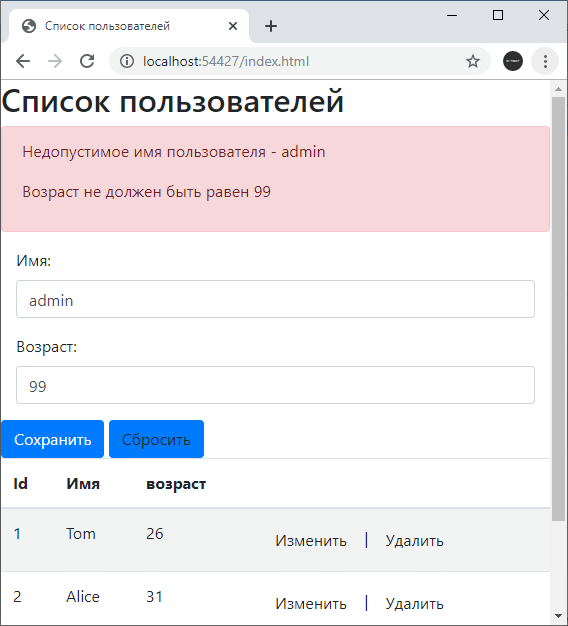
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>Список пользователей</title>      <link href="<https://stackpath.bootstrapcdn.com/bootstrap/4.4.0/css/bootstrap.min.css>" rel="stylesheet" />  </head>  <body>      <h2>Список пользователей</h2>      <div id="errors" class="alert alert-danger" style="display:none;"></div>      <form name="userForm">          <input type="hidden" name="id" value="0" />          <div class="form-group col-md-5">              <label for="name">Имя:</label>              <input class="form-control" name="name" />          </div>          <div class="form-group col-md-5">              <label for="age">Возраст:</label>              <input class="form-control" name="age" type="number" />          </div>          <div class="panel-body">              <button type="submit" id="submit" class="btn btn-primary">Сохранить</button>              <a id="reset" class="btn btn-primary">Сбросить</a>          </div>      </form>      <table class="table table-condensed table-striped  col-md-6">          <thead><tr><th>Id</th><th>Имя</th><th>возраст</th><th></th></tr></thead>          <tbody>          </tbody>      </table>      <div>2019 © Metanit.com</div>      <script>          // Получение всех пользователей          async function GetUsers() {              // отправляет запрос и получаем ответ              const response = await fetch("/api/users", {                  method: "GET",                  headers: { "Accept": "application/json" }              });              // если запрос прошел нормально              if (response.ok === true) {                  // получаем данные                  const users = await response.json();                  let rows = document.querySelector("tbody");                  users.forEach(user => {                      // добавляем полученные элементы в таблицу                      rows.append(row(user));                  });              }          }          // Получение одного пользователя          async function GetUser(id) {              const response = await fetch("/api/users/" + id, {                  method: "GET",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  const form = document.forms["userForm"];                  form.elements["id"].value = user.id;                  form.elements["name"].value = user.name;                  form.elements["age"].value = user.age;              }          }          // Добавление пользователя          async function CreateUser(userName, userAge) {                const response = await fetch("api/users", {                  method: "POST",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  reset();                  document.querySelector("tbody").append(row(user));              }              else {                  const errorData = await response.json();                  console.log("errors", errorData);                  if (errorData) {                      // ошибки вследствие валидации по атрибутам                      if (errorData.errors) {                           if (errorData.errors["Name"]) {                              addError(errorData.errors["Name"]);                          }                          if (errorData.errors["Age"]) {                              addError(errorData.errors["Age"]);                          }                      }                      // кастомные ошибки, определенные в контроллере                      // добавляем ошибки свойства Name                      if (errorData["Name"]) {                          addError(errorData["Name"]);                      }                        // добавляем ошибки свойства Age                      if (errorData["Age"]) {                          addError(errorData["Age"]);                      }                  }                    document.getElementById("errors").style.display = "block";              }          }          // Изменение пользователя          async function EditUser(userId, userName, userAge) {              const response = await fetch("api/users", {                  method: "PUT",                  headers: { "Accept": "application/json", "Content-Type": "application/json" },                  body: JSON.stringify({                      id: parseInt(userId, 10),                      name: userName,                      age: parseInt(userAge, 10)                  })              });              if (response.ok === true) {                  const user = await response.json();                  reset();                  document.querySelector("tr[data-rowid='" + user.id + "']").replaceWith(row(user));              }          }          // Удаление пользователя          async function DeleteUser(id) {              const response = await fetch("/api/users/" + id, {                  method: "DELETE",                  headers: { "Accept": "application/json" }              });              if (response.ok === true) {                  const user = await response.json();                  document.querySelector("tr[data-rowid='" + user.id + "']").remove();              }          }            // сброс формы          function reset() {              const form = document.forms["userForm"];              form.reset();              form.elements["id"].value = 0;          }          function addError(errors) {              errors.forEach(error => {                  const p = document.createElement("p");                  p.append(error);                  document.getElementById("errors").append(p);              });          }          // создание строки для таблицы          function row(user) {                const tr = document.createElement("tr");              tr.setAttribute("data-rowid", user.id);                const idTd = document.createElement("td");              idTd.append(user.id);              tr.append(idTd);                const nameTd = document.createElement("td");              nameTd.append(user.name);              tr.append(nameTd);                const ageTd = document.createElement("td");              ageTd.append(user.age);              tr.append(ageTd);                const linksTd = document.createElement("td");                const editLink = document.createElement("a");              editLink.setAttribute("data-id", user.id);              editLink.setAttribute("style", "cursor:pointer;padding:15px;");              editLink.append("Изменить");              editLink.addEventListener("click", e => {                    e.preventDefault();                  GetUser(user.id);              });              linksTd.append(editLink);                const removeLink = document.createElement("a");              removeLink.setAttribute("data-id", user.id);              removeLink.setAttribute("style", "cursor:pointer;padding:15px;");              removeLink.append("Удалить");              removeLink.addEventListener("click", e => {                    e.preventDefault();                  DeleteUser(user.id);              });                linksTd.append(removeLink);              tr.appendChild(linksTd);                return tr;          }          // сброс значений формы          document.getElementById("reset").addEventListener("click", function (e) {                e.preventDefault();              reset();          })            // отправка формы          document.forms["userForm"].addEventListener("submit", e => {              e.preventDefault();              document.getElementById("errors").innerHTML="";              document.getElementById("errors").style.display = "none";                const form = document.forms["userForm"];              const id = form.elements["id"].value;              const name = form.elements["name"].value;              const age = form.elements["age"].value;              if (id == 0)                  CreateUser(name, age);              else                  EditUser(id, name, age);          });            // загрузка пользователей          GetUsers();        </script>  </body>  </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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | if (errorData) {      const errorData = await response.json();      console.log("errors", errorData);      (errorData) {          // ошибки вследствие валидации по атрибутам          if (errorData.errors) {              if (errorData.errors["Name"]) {                  addError(errorData.errors["Name"]);              }              if (errorData.errors["Age"]) {                  addError(errorData.errors["Age"]);              }          }          // кастомные ошибки, определенные в контроллере          // добавляем ошибки свойства Name          if (errorData["Name"]) {              addError(errorData["Name"]);          }          // добавляем ошибки свойства Age          if (errorData["Age"]) {              addError(errorData["Age"]);          }      }      document.getElementById("errors").style.display = "block";  } |

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 некорретные данные, мы получим сообщения об ошибках.



## 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 возвратить объект string, а с помощью метода Json() можно возвратить объект, сериализованый в формат json. Однако контроллеры в Web API для возвращения результат а используют самые различные методы, а не только Json и Content. Например:

|  |  |
| --- | --- |
| 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 | [HttpGet]  public IEnumerable<User> Get()  {      return db.Users.ToList();  }    [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);  }    [HttpDelete("{id}")]  public IActionResult Delete(int id)  {      User user = db.Users.FirstOrDefault(x => x.Id == id);      if(user==null)      {          return NotFound();      }      db.Users.Remove(user);      db.SaveChanges();      return Ok(user);  } |

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

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

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 оправке 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:

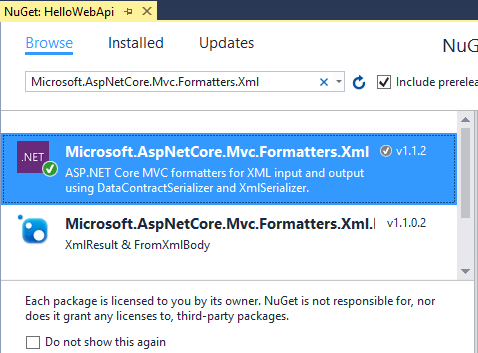
|  |  |
| --- | --- |
| 1 | Accept: text/html,application/xhtml + xml,application/xml;q = 0.9,image/webp,\*/\*;q = 0.8 |

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

В качестве альтернативы можно также использовать другой способ:

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

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  2  3  4  5  6  7  8  9  10  11  12 | public void ConfigureServices(IServiceCollection services)  {      services.AddMvc().AddXmlSerializerFormatters();        // альтернативный способ      // services.AddMvc(options =>      // {      //      options.OutputFormatters.Add(new XmlSerializerOutputFormatter());      // });        // остальной код метода  } |

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:

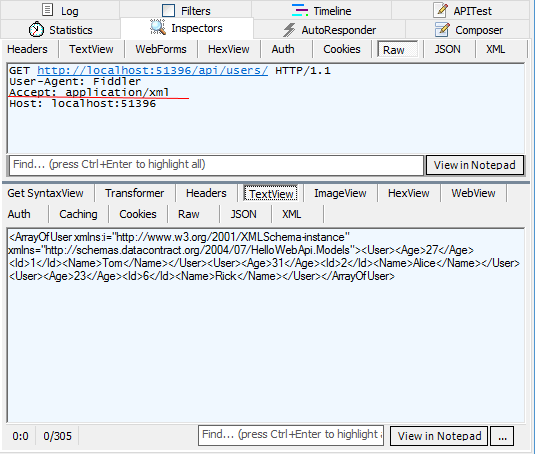
|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpGet]  public IEnumerable<User> Get()  {      return db.Users.ToList();  } |

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":



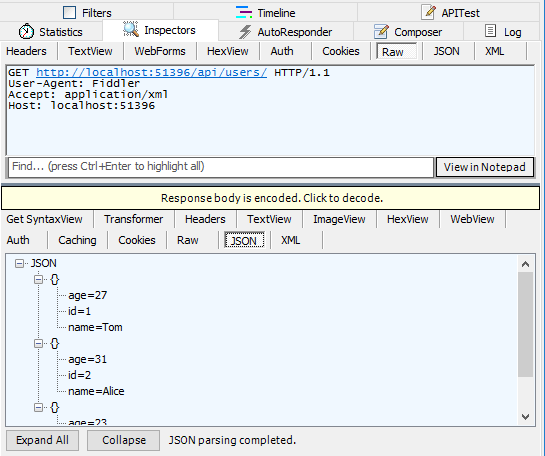
Если бы мы не указали заголовок Accept, то данные по прежнему передавались бы в формате json.

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

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



### Получение типа контента из строки запроса

Заголовок Accept - 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  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 | using Microsoft.AspNetCore.Builder;  using Microsoft.Extensions.DependencyInjection;  using Microsoft.EntityFrameworkCore;  using HelloWebApi.Models;  using Microsoft.Net.Http.Headers;    namespace HelloWebApi  {      public class Startup      {          public void ConfigureServices(IServiceCollection services)          {              string con = "Server=(localdb)\\mssqllocaldb;Database=usersdbstore;Trusted\_Connection=True;MultipleActiveResultSets=true";              services.AddDbContext<UsersContext>(options => options.UseSqlServer(con));                services.AddMvc()                  .AddXmlDataContractSerializerFormatters()                  .AddMvcOptions(opts => {                      opts.FormatterMappings.SetMediaTypeMappingForFormat("xml", new MediaTypeHeaderValue("application/xml"));               });          }            public void Configure(IApplicationBuilder app)          {              app.UseDefaultFiles();              app.UseStaticFiles();              app.UseMvc();          }      }  } |

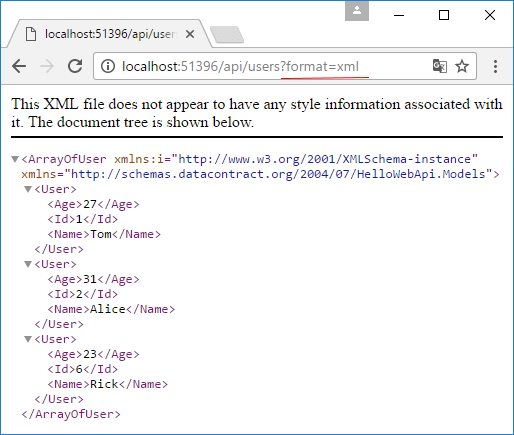
В методе AddMvcOptions() устанавливается сопоставление формата из строки запроса с определенным медиа-типом:

|  |  |
| --- | --- |
| 1 | opts.FormatterMappings.SetMediaTypeMappingForFormat("xml", new MediaTypeHeaderValue("application/xml")); |

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

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

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 Accept - он все равно не будет учитываться.