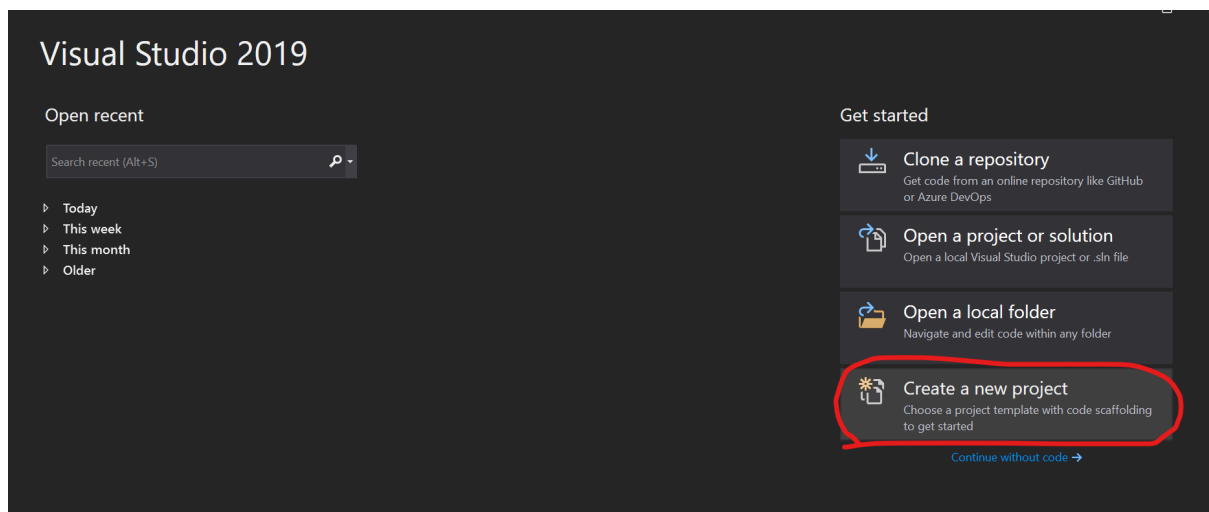


Laborator 1

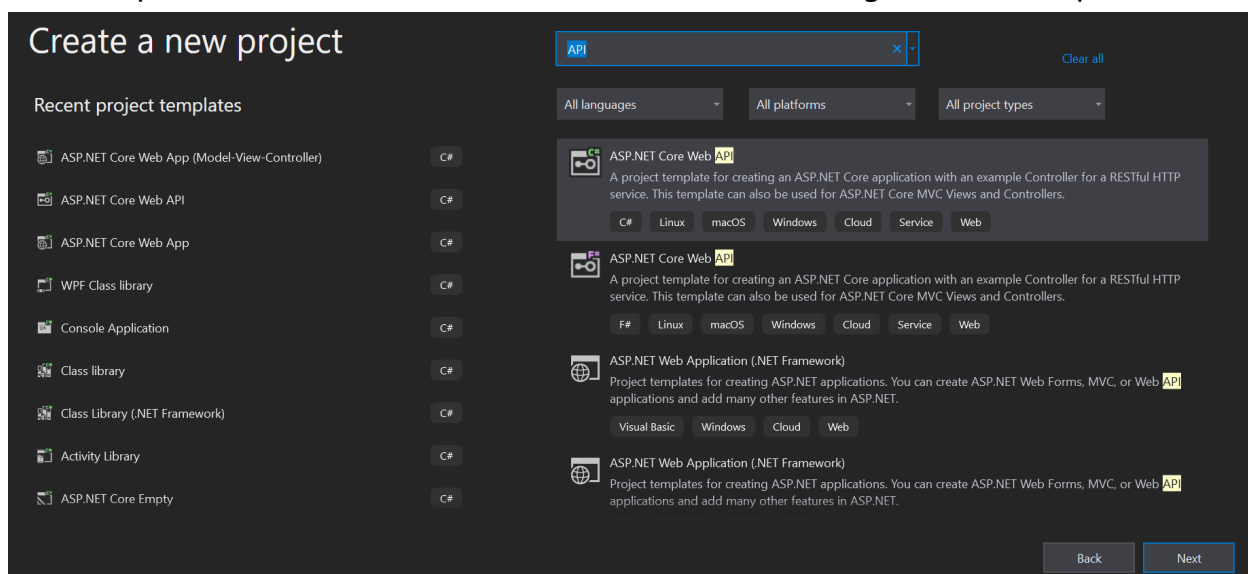
Intro in .Net Core, Web APIs

Hai sa ne jucam putin cu .NET.

Deschidem Visual Studio Community Edition si creem un proiect nou:



Noi o sa lucram cu aplicatii ASP .NET Core Web Application in *C#*, deci acesta este template-ul de care vom avea nevoie, dăm ce nume sugestiv ne vine pe loc:



Configure your new project

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web

Project name

WebApplication12

Location

C:\Users\mihae\source\repos

Solution name ⓘ

WebApplication12

☐ Place solution and project in the same directory

Additional information

ASP.NET Core Web API C# Linux macOS Windows Cloud Service Web

Target Framework ⓘ

.NET 5.0 (Current)

Authentication Type ⓘ

None

☒ Configure for HTTPS ⓘ

☐ Enable Docker ⓘ

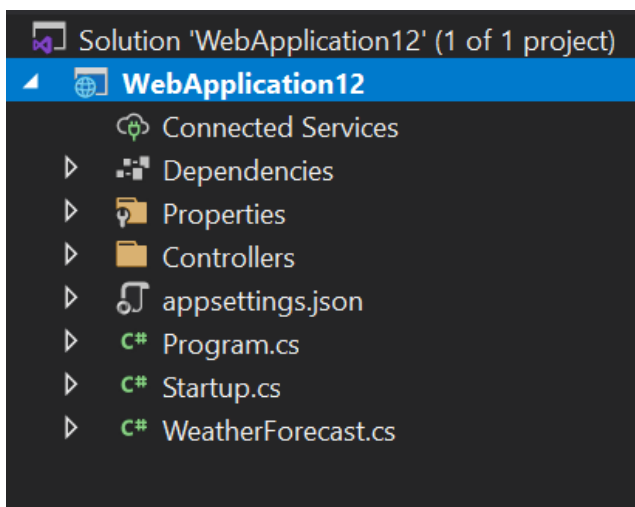
Docker OS ⓘ

Linux

☒ Enable OpenAPI support ⓘ

Am creat proiectul, ne putem juca.

Cam așa ar trebui sa arate solutia noastra:



Noi vom lucra pe baza acestui template, vom pleca de la el, il vom dezvolta, vom adăuga chestii smechele si îl vom face sa lucreze ca o aplicație calumea, completa.

Cate ceva despre .NET Core, utile de-a lungul laboratorului

Funcționează ca o aplicație de consola; motiv pentru care are o funcție `Main` (`public static void Main(args)`) care este apelata la pornire, gasita in fisierul **Program.cs**.

Ea nu face decat sa ridice un `Host` si sa specifice clasa **Startup** pentru configurari; isi face treaba out-of-the-box, nu avem treaba cu ea.

Clasa **Startup** este mai interesanta putin; aici se găsesc toate configurările aplicației. Este inima soluției; de exemplu, cand vom avea nevoie sa adaugam o librărie nouă, aici vom intra; cum vom vrea sa evoluam aplicatia si sa-i mai aduagam o chestie, doua, aici vom mai avea treaba.

Core, fata de inechitul Framework, este mai lightweight, si portabil pe Linux / Mac. Asta datorită faptului ca toate dependintele (inclusiv cele din librariile de Windows) au fost puse intr-un dll, toate la un loc, si apoi acel dll spart in mai multe dll-uri mai mici. Astfel, aplicația vin din cutie cu minimul necesar pentru a rula, iar fiecare nevoie individuala se gaseste in alt dll, il alta librerie, pe care ti-o poti trage cand e necesar. Astfel, apar **NuGet**-urile, si **NuGet Package Manager**.

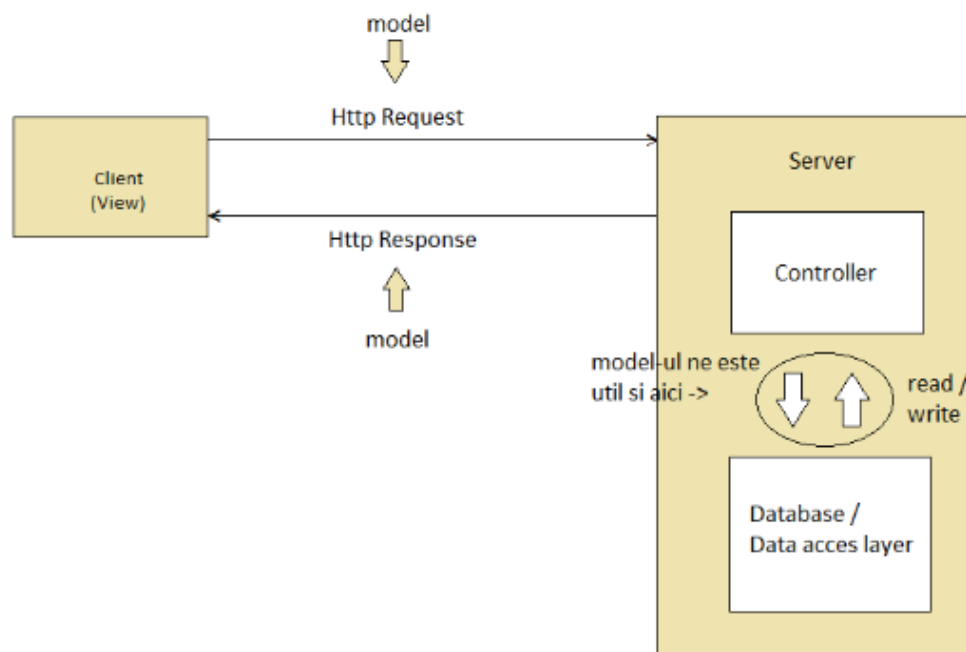
NuGet = librerie (basically); continue mai multe DLL-uri (Dynamic Link Library), care au și ele niște dependinte în spate, etc. Iar NuGet Package Manager este sistemul prin care se gestionează librării în ecosistemul .NET; sort of npm (prin NuGet Package Manager vom lua librării pe care le vom adauga in Startup-ul de care vorbeam mai sus). Pentru a accesa, avem in bara de sus Tools -> NuGet Package Manager.

MVC si aplicatiile noastre de toate zilele

Design pattern-ul de Model View Controller, popular mai peste tot într-o forma sau alta (MVVM in aplicatii complexe, MVP - android, etc). Impartirea pe date

(model), cum sunt ele afisate (view) si logica lor de manipulare (controller) este destul de intuitiva si aplicabila peste tot.

Noi vom face o mica adaptare: cum vom folosi un client-server architecture, controller-ul va fi pe server, modelul pe amandoua (fiind „contractul” intre cele doua), iar view-ul pe client. Dar mai usor, tineti minte doar schema asta:



Schema de mai sus o sa ne fie de folos mai tot laboratorul
Pare mai complicata decat este, va asigur.

Astfel, ajungem in **Web API Controllers**.

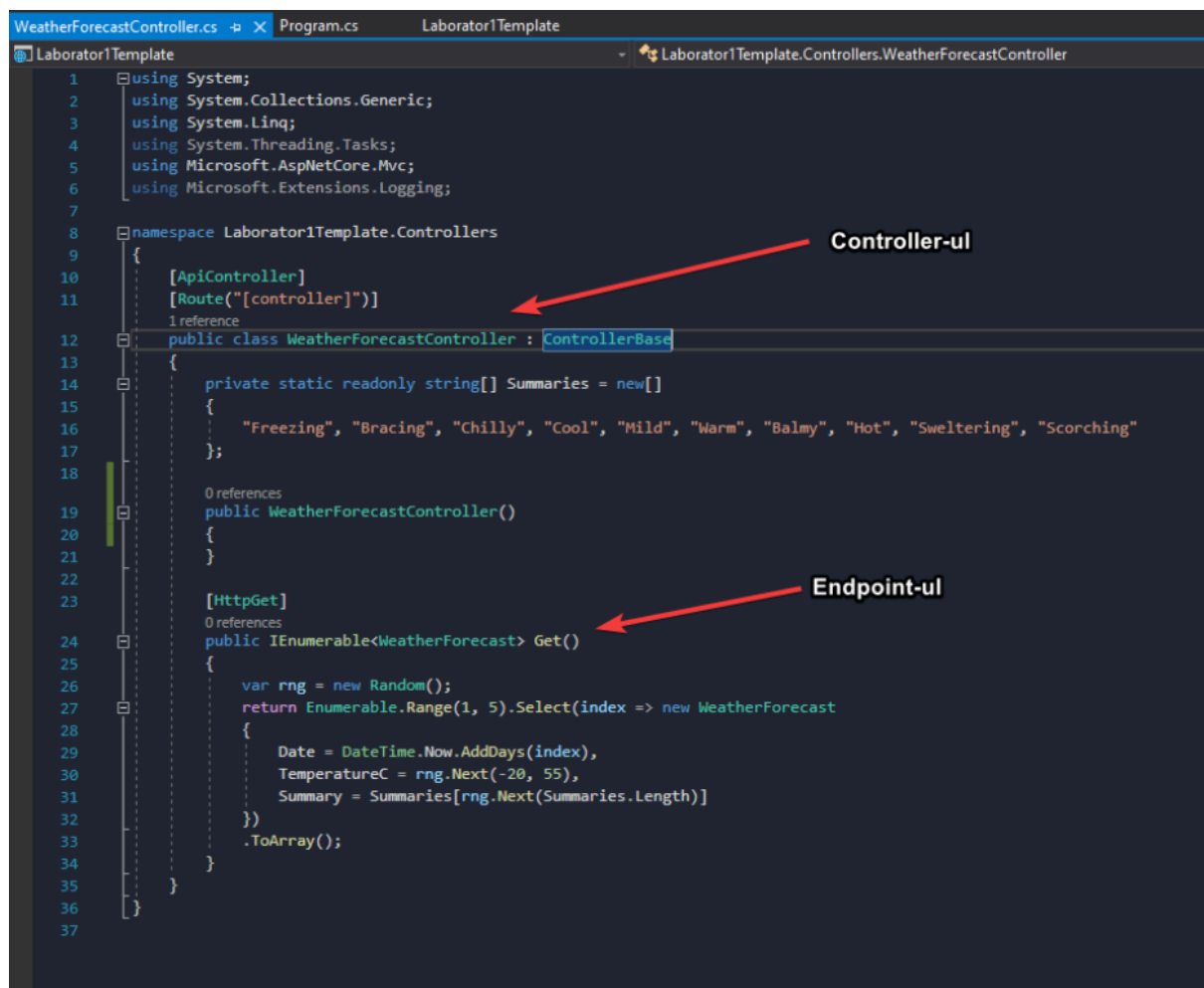
Poveste / Teorie: In .NET MVC clasic, Controller-ul returna direct View-ul, randat pe server in functie de model-ul respectiv (returna un cod html care se punea pe pagina de client).

Cum, in a noastra utilizare, avem codul de client separat de cel de server, aceasta abordare nu mai functioneaza, motiv pentru care Controller-ul va returna direct Modelul, datele, sub format JSON; el va avea rolul unui Web API.

Ce trebuie sa stiti despre ideea de API: este o interfata (din punctul de vedere a clientului) ce ofera un set de operatii catre alte programatori.

Controllere noastre, vor avea un rol de Web API, pentru ca ele vor oferi anumite operatii, de care partea de client se va folosi. Termenul pentru aceste operatii in cazul unui API Controller este de API endpoint. Endpoint, cu sensul de punct de capat al unei comunicari (pentru client, este capatul de final unde poate merge pentru date, pentru server, este capatul de inceput pentru a oferi date).

Un exemplu ar fi chiar Controller-ul implicit pe care ni-l da Template-ul:



```
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 Laborator1Template.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", "Sweltering", "Scorching"
17         };
18
19         public WeatherForecastController()
20         {
21         }
22
23         [HttpGet]
24         public IEnumerable<WeatherForecast> Get()
25         {
26             var rng = new Random();
27             return Enumerable.Range(1, 5).Select(index => new WeatherForecast
28             {
29                 Date = DateTime.Now.AddDays(index),
30                 TemperatureC = rng.Next(-20, 55),
31                 Summary = Summaries[rng.Next(Summaries.Length)]
32             })
33                 .ToArray();
34         }
35     }
36 }
37
```

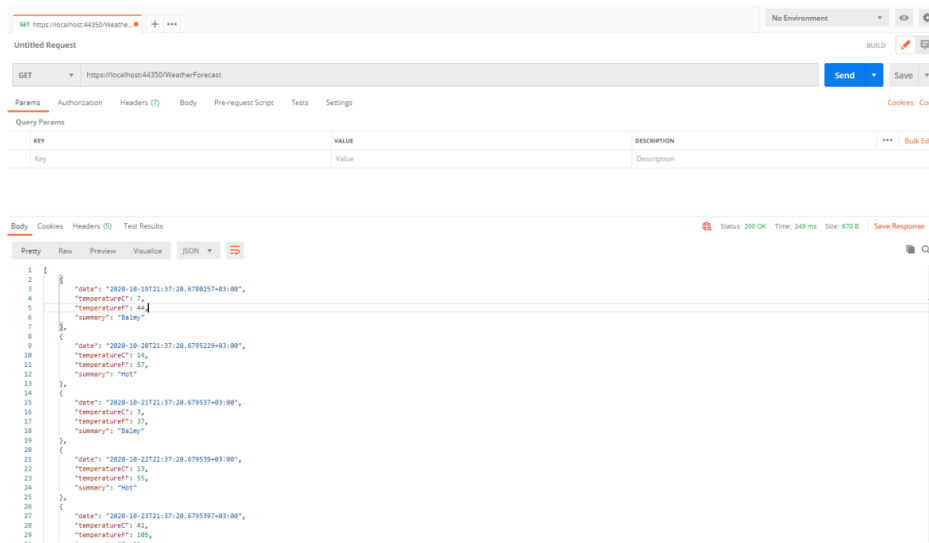
Controller-ul

Endpoint-ul

Acesta poate fi accesat din afara, și este calea de comunicare cu server-ul. Asa se pot deschide căi de comunicare (date de endpoint-uri) din partea aplicației.

Pentru a test API-uri, recomand Postman. Cand rulăm proiectul, putem lua url-ul din browser, il punem in postman, completam cu ruta catre endpoint-ul nostru

(/controller/actionName de obicei - actionName este dat in paranteze langa verb-ul endpoint-ului, de ex `HttpGet(„weatherForecast“)`). Un exemplu de request si response-ul venit din aplicatia noastra ar fi:



Acesta poate fi accesat din afara, si este calea de comunicare cu serverul. Aşa se pot deschide căi de comunicare (date de endpoint-uri) din partea aplicaţiei.

Ce ar mai fi de specificat:

- **CRUD:** Create, read, update, delete. Similar, exista Http Verbs; cele mai populare sunt POST, GET, PUT, PATCH, DELETE, care s-ar traduce ca Create, Read, Update, Partial update, Delete(surprise); fiecarui endpoint ii este atasat un Verb, iar requestul din client va trebui sa corespunda.

1. GET:

- Atributul necesar creării unui endpoint de tipul get este `[HttpGet]`. În interiorul sau se declara ruta acţiunii (endpointul).

A. Get simplu (fără "numele endpointului în atribut"; fara parametri):

```
[HttpGet]
0 references
public IEnumerable<Student> Get()
{
    return Students;
}
```

B. Get cu parametri in rute:

- Atunci cand se face un req aplicația redirectioneaza cererea către endpointul/controllerul din link împreuna cu parametrii transmiși.
- Pentru a trimite id ca parametru în link vom folosi următoarele url-uri în postman:
 - ❖ <https://localhost:44367/api/students/1> (1 este id-ul)
 - ❖ <https://localhost:44367/api/students/withFilters/Maria/23> (Maria este parametru name si 23 age)
 - ❖ <https://localhost:44367/api/students/fromRouteWithId/2>
- [FromRoute] - Specifică parametrul ce va fi transmis in ruta

```
[HttpGet("{id}")]
0 references
public IEnumerable<Student> GetWithIdInEndpoint(int id)
{
    return Students.Where(s => s.Id == id);
}

[HttpGet("withFilters/{name}/{age}")]
0 references
public IEnumerable<Student> GetWithFilters(string name, int age)
{
    return Students.Where(s => s.Name.Equals(name) && s.Age <= age);
}

[HttpGet("fromRouteWithId/{id}")]
0 references
public IEnumerable<Student> GetFromRouteWithId([FromRoute] int id)
{
    return Students.Where(s => s.Id == id);
}
```

C. Get cu parametri Header:

- Un Header (Http header) este un field din req sau răspunsul primit în urma unui req în care se transmit date aditionale.
- [FromHeader] - atributul anunța ca id-ul va fi transmis in header

```
[HttpGet("fromHeader")]
0 references
public IEnumerable<Student> GetFromHeader([FromHeader] int id)
{
    return Students.Where(s => s.Id == id);
}
```

GET ▾		https://localhost:44367/api/students/fromHeader	
Authorization	Headers (1)	Body	Pre-request Script
	Key	Value	
<input checked="" type="checkbox"/>	Id	6	
	New key	Value	
Response			

D. Get cu query params:

- Un Query param reprezinta o informatie transmisa in url sub forma:
api/controller/endpoint?title=Query_string&action=edit.
- [FromQuery] - atributul anunța ca id-ul va fi transmis ca query param.

```
[HttpGet("fromQuery")]
0 references
public IEnumerable<Student> GetFromQuery([FromQuery] int id)
{
    return Students.Where(s => s.Id == id);
}
```

2. POST:

- Atributul necesar creării unui endpoint de tipul post este [HttpPost]. În interiorul sau se declara ruta acțiunii (endpointul).

A. Post cu body:

- Un body este un obiect din req în care se transmit date aditionale.
- [FromBody] - atributul anunța ca datele vor fi transmise in body

```
[HttpPost("FromBody")]
0 references
public IEnumerable<Student> AddWithFromBody([FromBody] Student student)
{
    Students.Add(student);

    return Students;
}
```

POST ▼ https://localhost:44367/api/students/ Params Send ▼

Authorization Headers (1) **Body** ● Pre-request Script Tests

☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary JSON (application/json) ▼

```
1 {
2   "Name": "Maria",
3   "Age": 20
4 }
```

B. Post cu form:

- Un form este un obiect din req în care se transmit date aditionale.
- [FromForm] - atributul anunța ca datele vor fi transmise in body

```
[HttpPost("FromForm")]
0 references
public IActionResult AddWithFromForm([FromForm] Student student)
{
    Students.Add(student);

    return Ok(Students);
}
```

POST ▼ https://localhost:44367/api/students/fromForm Params

Authorization Headers (1) **Body** ● Pre-request Script Tests

☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary

	Key	Value	Description
<input checked="" type="checkbox"/>	Id	21	
<input checked="" type="checkbox"/>	Name	Miha	
<input checked="" type="checkbox"/>	Age	26	
	New key	Value	Description

Response

3. Put:

- Atributul necesar creării unui endpoint de tipul put este [HttpPut]. În interiorul sau se declara ruta acțiunii (endpointul).
- Este folosit pentru update total.

```
[HttpPut]
0 references
public async Task<IActionResult> Update ([FromBody] Student student)
{
    var studentIndex = Students.FindIndex((Student _student) => _student.Id.Equals(student.Id));
    Students[studentIndex] = student;

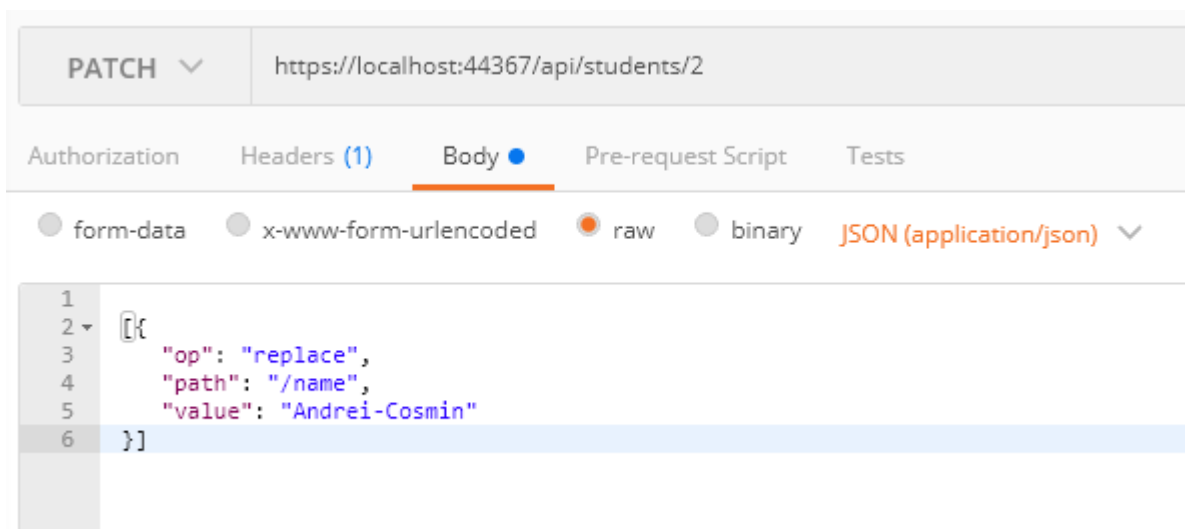
    return Ok(Students);
}
```

4. Patch:

- Atributul necesar creării unui endpoint de tipul patch este [HttpPatch]. În interiorul sau se declara ruta acțiunii (endpointul).
- Este folosit pentru update parțial.

```
[HttpPatch("{id}")]
0 references
public IActionResult Patch ([FromRoute] int id, [FromBody] JsonPatchDocument<Student> student)
{
    if(student != null)
    {
        var studentToUpdate = Students.FirstOrDefault(_student => _student.Id.Equals(id));
        student.ApplyTo(studentToUpdate, ModelState);

        if (!ModelState.IsValid)
        {
            return BadRequest();
        }
        return Ok(Students);
    }
    else
    {
        return BadRequest();
    }
}
```



5. Delete:

- Atributul necesar creării unui endpoint de tipul delete este [HttpDelete].
În interiorul sau se declara ruta acțiunii (endpointul).

```
[HttpGet]
0 references
public IActionResult Delete (Student student)
{
    Students.Remove(student);
    return Ok(Students);
}
```

- **Endpoint**-urile vor returna un status code; pe langa obiectul / mesajul de care este nevoie, un endpoint va returna si un status code, ce va indica ce s-a intamplat cu respectivul request; cele mai intalnite sunt: 200 (Ok), 404(Not Found), 400(Bad Request), 401(Unauthorized), 405 (Method not allowed - asta este legat de HttpVerb pus gresit) si 500 (Internal Server Error); pentru simplitate, in .NET cand dam return, ne putem folosi de niște metode venite din ControllerBase, intuitive (ex: return Ok(model), return NotFound(model) etc).

```
return BadRequest(); //Status400BadRequest
return NoContent(); // Status204NoContent
return NotFound(); // Status404NotFound
return Forbid(); // Status403Forbidden
return Ok(Students); // Status200OK
```

O sugestie: cand nu o sa mearga ceva bine, si sigur se va intampla sa fie un null unde nu va asteptati, use debugging. Duce-ti-va pe linia care cauzează probleme, apasati F9 (sau click la inceput de rand) pentru a pune un breakpoint, aplicatia se oprește acolo și dati hover pe ce variabile sunt, vedeti ce este null; de acolo va mai puteti juca cu debugger-ul pana gasiti solutia (F5- run pana la final sau urmatorul breakpoint; F10 - run doar o operatie, dupa asteapta input again; F11 - debugger-ul intra in metoda apelata la acest pas).

Extra:

Pentru a crea o clasa in c# dăm click dreapta pe proiect/folder -> Add -> New Item apoi selectam Class din lista. La fel facem și pentru Controlleri însă din lista alegem Controller apoi din API -> Api Controller Empty.

// Declararea unei clase

```
public class Student
{
    [Required] // Atribut folosit pentru a genera o err în cazul în care Id ul
este transmis ca null;
    public int Id { get; set; }

    public string Nume { get { return nume; } set { nume = value; } }

    [Range(18, 100)] // Atribut pentru a genera err în cazul transmiterii unui
număr mai mic ca 18 si mai mare ca 100
    public int Age { get; set; }
}
```

// Declararea unei liste

```
public static List<Student> Students = new List<Student>
{
    new Student {Id = 1, Name = "Ana", Age = 21},
    new Student {Id = 2, Name = "Andrei", Age = 22},
    new Student {Id = 3, Name = "Vlad", Age = 23},
    new Student {Id = 4, Name = "Marius", Age = 24},
    new Student {Id = 5, Name = "Daniela", Age = 31},
    new Student {Id = 6, Name = "Ana", Age = 25},
    new Student {Id = 7, Name = "Ana", Age = 30},
    new Student {Id = 8, Name = "Ana", Age = 20},
};
// List = lista; List<Student> = o lista ale căror obiecte sunt de tipul Student
List<Student> Students
```

Scrierea de mai sus este similar cu:

```
public static List<Student> Students = new List<Student>();  
Students.Add( new Student {Id = 7, Name = "Ana", Age = 30});
```

```
Students.Where(s => s.Id == id);
```

Funcția Where are în interior o lambda expression și filtrează lista de studenți; noi o să folosim de obicei forma cea mai simplă (obiect primit) => obiect.Ceva == x;