

APBD - Tutorial 3

pgago@pja.edu.pl

March 19, 2020

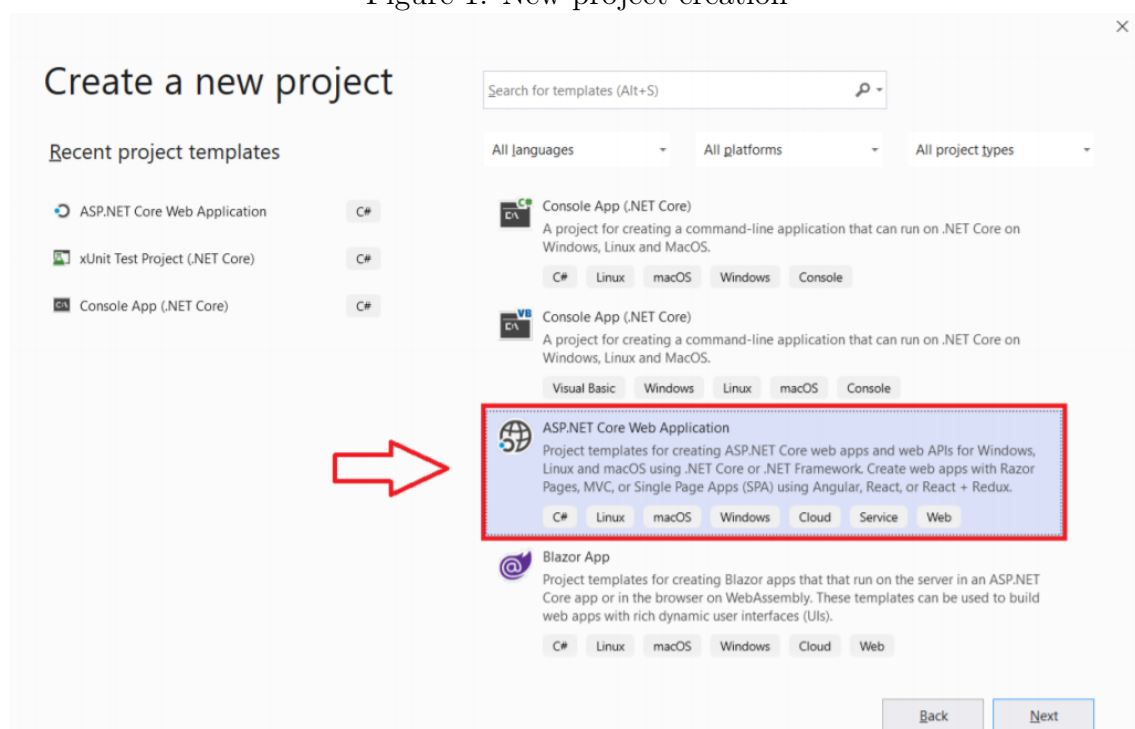
Task 1 - Postman installation

During this exercise, we will create a new Web API application. When testing a web application, we'll need an additional application called Postman. The application is free and exists both in the form of a plugin for the Chrome browser or a separate desktop application. Please install it. The most convenient way to use the desktop version. Address URL: <https://www.postman.com/downloads/>

Task 2 - New project creation

1. Create a new repository for tutorial 3.
2. Then, create a new project **ASP.NET Core Web Application**. From the available templates choose **API**.

Figure 1: New project creation

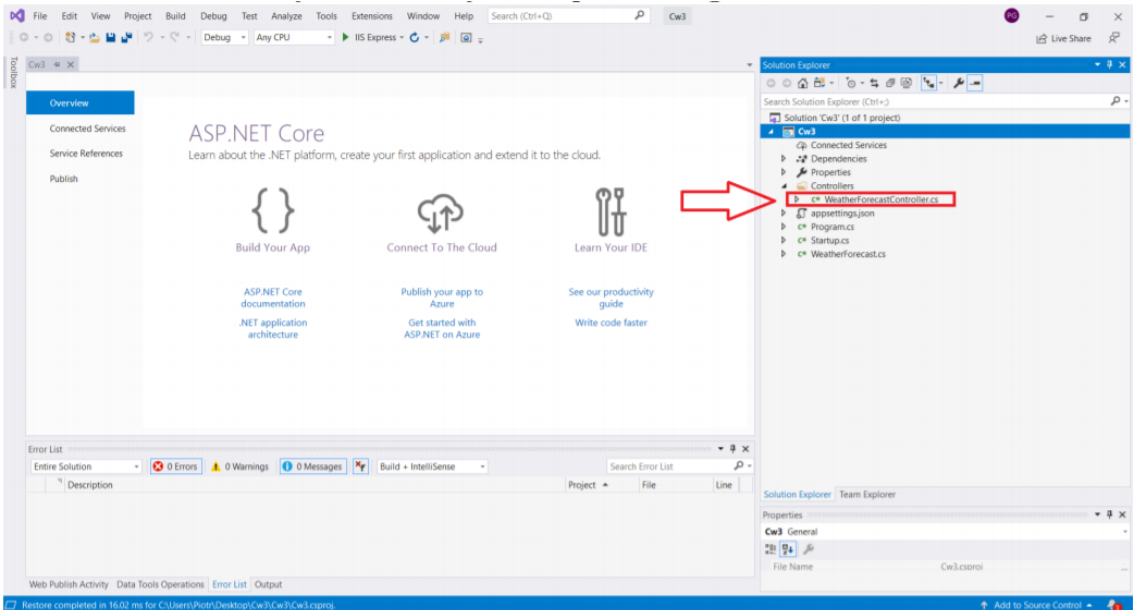
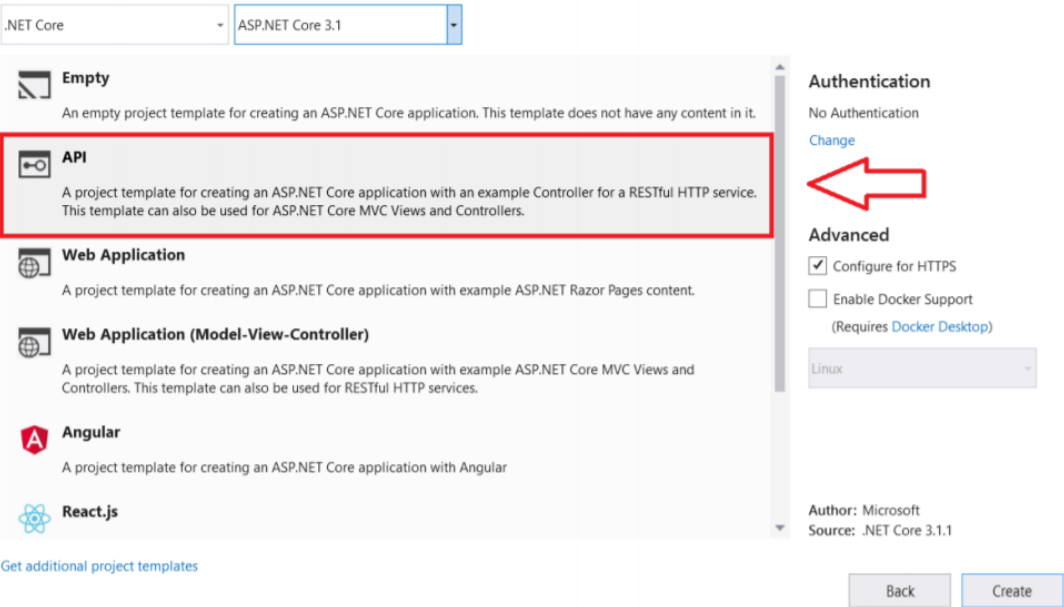


3. After creating a new project you should delete `Controllers/WeatherForecastController.cs`

Figure 2: Available templates

x

Create a new ASP.NET Core web application

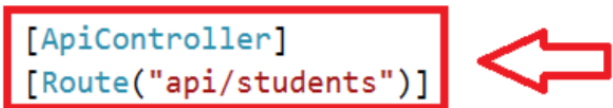


Task 3 - Adding new controller

1. In this task we will add our first controller. The controller processes user requests and returns the expected result.
2. We add a new controller called `StudentsController.cs` to the Controllers folder.
3. When we think of API and REST approach - we should think about the resources that our API provides via HTTP. Controller names usually reflect the resources being shared.
4. Then please modify the code so that our class is ready to handle HTTP REST requests.
5. We add two attributes above the class name. The first of these `[ApiController]` marks our controller as an API controller. Thanks to this, we will be able to use

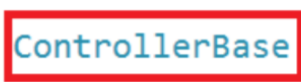
several built-in validation mechanisms that will be useful later. The `[Route]` attribute allows you to specify the address that our controller will identify. All requests sent to this address will be automatically redirected to this controller.

```
7 namespace Cw3.Controllers
8 {
9     [ApiController]
10    [Route("api/students")]
11    public class StudentsController : Controller
12    {
13        public IActionResult Index()
14        {
15            return View();
16        }
17    }
18 }
```



6. Then we change the base class - we should inherit from `ControllerBase`. The `ControllerBase` class contains many useful helper methods that we will use.

```
namespace Cw3.Controllers
{
    [ApiController]
    [Route("api/students")]
    public class StudentsController : ControllerBase
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```



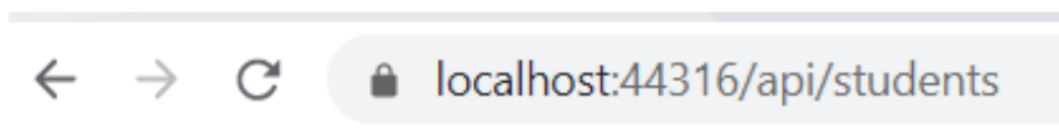
7. The `Index()` method has stopped compiling. At this point, we are adding the first method that will return data. We modify the `Index()` method as follows.

```

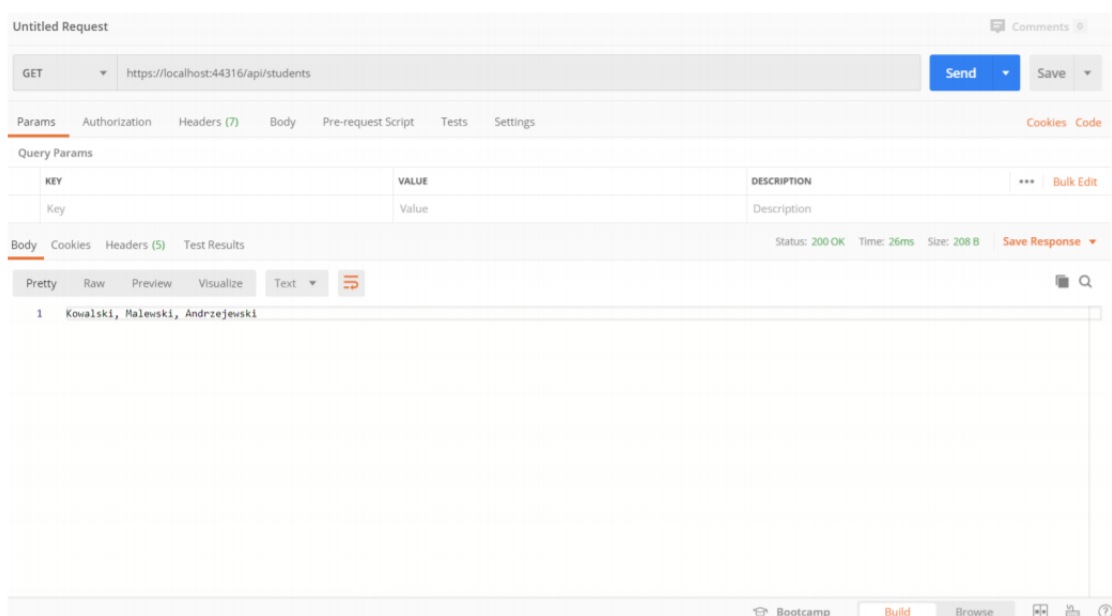
[ApiController]
[Route("api/students")]
0 references
public class StudentsController : ControllerBase
{
    [HttpGet]
    0 references
    public string GetStudent()
    {
        return "Kowalski, Malewski, Andrzejewski";
    }
}

```

8. Then let's try to run the application. Then try to enter `https://localhost:44316/api/students`. Please make the same HTTP request using Postman. Remember that in your case the address may differ (44316 is a port number and may be different).



Kowalski, Malewski, Andrzejewski



Task 4 - passing parameter as a URL segment

1. By selecting the appropriate URL and HTTP method (e.g. GET, PUT, POST, DELETE, PATCH) we can choose which method will be run. Usually, however, we need to send some additional data to the server. We have several ways to do this. The first is the use of the URL segment.
2. The URL segment is selected when the information sent is related to the identification of a resource (similar to the id in the database). I.e. it can be a student id from the database. It would be bad, however, to use the URL segment to send i.e. information on how we want to sort the data.
3. Please add a new method that will allow us to pass the parameter. As you can see in the `HttpGet` attribute, we've added the `id` parameter. In addition, the added method also has a parameter with the same name. Then, when making the request, the parameter from the corresponding part of the URL will be automatically passed to the appropriate method.

Figure 3: Running the application

```
[HttpGet("{id}")]
0 references
public IActionResult GetStudent(int id)
{
    if (id == 1)
    {
        return Ok("Kowalski");
    } else if (id == 2)
    {
        return Ok("Malewski");
    }

    return NotFound("Nie znalezione studenta");
}
```

4. In addition, we use the `IActionResult` interface here. It is an interface used to return various types of data - text, JSON files, XML and others. In our case, we use two additional methods `Ok()` and `NotFound()`. Both methods come from the `ControllerBase` superclass. They allow us to easily return data and "wrap" them in the appropriate response and HTTP code (eg 200 OK or 404 Not Found).
5. Then test the method with the help of the Postman app. Please try to transfer data in different ways and check what the effect will be.

Testing the endpoint

Untitled Request

Comments 0

GET

https://localhost:44316/api/students/2

Send

Save

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Cookies

Code

Query Params

KEY	VALUE	DESCRIPTION	***	Bulk Edit
Key	Value	Description		

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 4.61s

Size: 184 B

Save Response

Pretty

Raw

Preview

Visualize

Text

1

Malewski

Task 5 - passing the data using QueryString

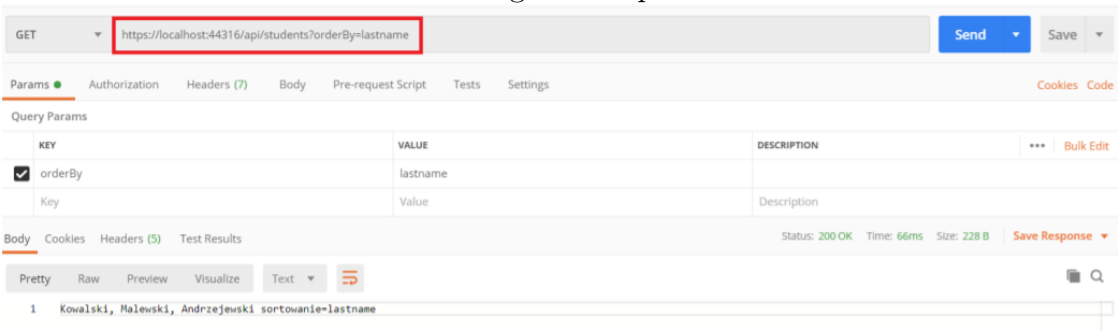
- 1. In this exercise, let's modify the method we wrote in the previous task.
- 2. We want to pass a parameter that would allow us to determine how data should be sorted. In this case, we are not downloading data from the database yet, but for now we are only looking at how to transfer the necessary data to the server.
- 3. Modify the first method added to `StudentsController` so that it looks like the one below.

Modification of the method to accept QueryString parameters

```
[HttpGet]
0 references
public string GetStudents(string orderBy)
{
    return $"Kowalski, Malewski, Andrzejewski sortowanie={orderBy}";
}
```

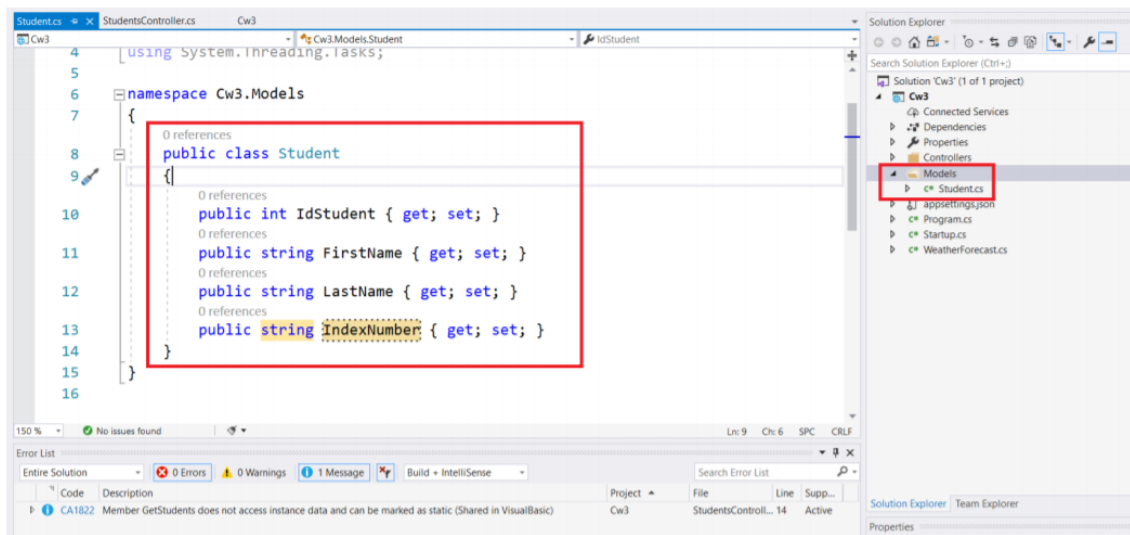
- 4. Then run the application and check if we are able to transfer data with the help of QueryString.

Testing the endpoint



Task 6 - passing the data in Body

1. In the next task, he will prepare a method for responding to HTTP POST requests. These types of methods are usually used to add new items to the database. We will prepare a method that will be used to add new students to the database. Of course, we do not currently use a real database.
2. First, let's add a folder called "Models" to the project. Inside it, add a file called "Student.cs". It will be a class that models our data. Using it, we will transfer data to the controller.



3. Then add another method to the controller.

```
[HttpPost]
0 references
public IActionResult CreateStudent(Student student)
{
    //... add to database
    //... generating index number
    student.IndexNumber = $"s{new Random().Next(1, 20000)}";
    return Ok(student);
}
```

4. As you can see the method does not do too much. Note the [HttpPost] attribute and the "Student student" parameter. Each time a complex type is used as a parameter (e.g. Student class), the data is deserialized and retrieved from the body of the HTTP request.
5. Then run the application and test it with the help of the Postman application.

Untitled Request

Comments 0

POST

https://localhost:44316/api/students

Send

Save

Params

Authorization

Headers (9)

Body

Pre-request Script

Tests

Settings

Cookies

Code

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautify

1 {

2 "idStudent": 1,

3 "firstName": "Jan",

4 "lastName": "Kowalski"

5 }

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 69ms

Size: 259 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

1 {

2 "idStudent": 1,

3 "firstName": "Jan",

4 "lastName": "Kowalski",

5 "indexNumber": "s9785"

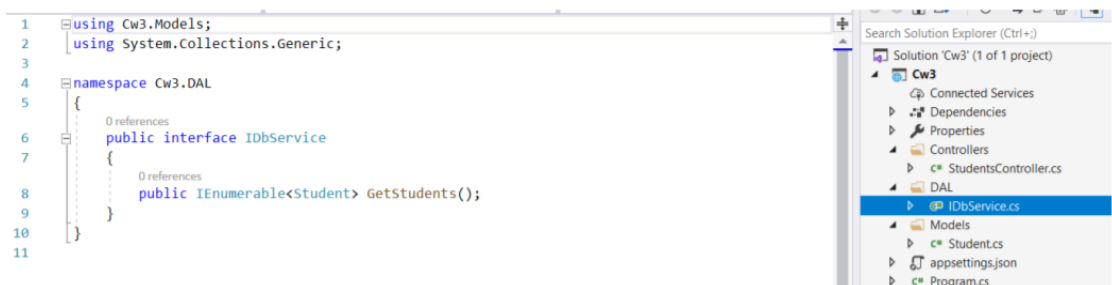
6 }

Task 7 - adding additional methods

1. Please add methods that respond to PUT and DELETE requests yourself. each of these methods, please return the code 200 and the message "Update complete" or "Delete completed".
2. Both methods should take as the id parameter passed as the URL segment.
3. Test the operation of the added methods in the Postman application.

Task 8 - adding "artificial" database

1. In the last exercise we will try to add an artificial database. Often, while working on the application, we would like to deal with the implementation of subsequent elements without waiting for the database administrator.
2. In this case, it's worth "blocking" the database, which we will later replace with a "real" one without major problems using the dependency container.
3. To begin with, we add a DAL (Data Access Layer) folder to the project.
4. We put two files inside. The first one will represent an interface with one method returning a collection of students. Note that I have chosen the `IEnumerable<T>` interface as the return type.



5. Then in the same folder we add the `MockDbService.cs` class. Inside, we add a collection containing several students.



6. As you can see, the `MockDbService` class returns previously prepared collections. The interface will serve us as a "glue" or abstraction layer, which allows us to use the `MockDbService` class in any controller. Using the interface will also allow us to easily replace the interface implementation with another one in the future.
7. We could now use the `MockDbService` class directly in the `StudentsController`. However, we do not want to create a strong relationship between the `StudentsController` and `MockDbService` classes.
8. First, let's modify the `StudentsController` class as follows.

```

[ApiController]
[Route("api/students")]
1 reference
public class StudentsController : ControllerBase
{
    private readonly IDbService _dbService;

    0 references
    public StudentsController(IDbService dbService)
    {
        _dbService = dbService;
    }
}

[HttpGet]
0 references
public IActionResult GetStudents(string orderBy)
{
    return Ok(_dbService.GetStudents());
}

```

9. As you can see we pass the parameter in the constructor. The parameter type is IDbService.
10. Then we need to register our service. We will describe this mechanism during the next lecture. We modify the ConfigureServices method in the Startup.cs class.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDbService, MockDbService>();
    services.AddControllers();
}

```

11. Then start the application and use Postman to make HTTP GET requests to the address api/students.

GEThttps://localhost:44316/api/students

Send

Save

Params

Authorization

Headers (7)

Body

Pre-request Script

Tests

Settings

Cookies

Code

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

This request does not have a body

Body

Cookies

Headers (5)

Test Results

Status: 200 OK

Time: 7ms

Size: 417 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

[

{

"idStudent": 1,

"firstName": "Jan",

"lastName": "Kowalski",

"indexNumber": null

}

,

{

"idStudent": 2,

"firstName": "Anna",

"lastName": "Maliewski",

"indexNumber": null

}

,

{

"idStudent": 3,

"firstName": "Andrzej",

"lastName": "Andrzejewicz",

"indexNumber": null

}

]