

COMPUTING SUBJECT: Restful ASP.Net Core-services

TYPE: Assignment

IDENTIFICATION: RestService#2

COPYRIGHT: *Peter Levinsky & Michael Claudius*

LEVEL: Medium

TIME CONSUMPTION: 1hour + approx. 1hour for the advanced part

EXTENT: 100 lines

OBJECTIVE: Restful services with more advanced URI's

PRECONDITIONS: Rest service theory. Http-concepts
Computer Networks Ch. 2.2

COMMANDS:

IDENTIFICATION: RestService#2 / PELE with kindly respect and inspiration from MICL

Overall Purpose

The overall purpose for the group of 'RestService' assignments is to be able to provide and consume restful ASP.Net Core web services, to prepare the 'RestService' to be published in Azure, including testing the service and finally to setup the 'RestService' to be consumed from a browser (e.g. using Typescript) i.e. support CORS.

The whole group of assignments consist of 7 steps:

1. [A simple REST Service with CRUD.](#)
- 2. More advanced and complex URI's. (this assignment)**
3. Adding help-pages to the REST Service (Swagger)
4. Testing a REST Service and publish in Azure.
5. Consuming a REST service from a C# Console application.
6. Adding Support for CORS to the REST Service
7. A REST Service using a database

Background Material:

The HTTP protocol: See Computer Network chap 2 pp. 111-136

Note of REST (Peter Levinsky): See [NetHttpNote.pdf](#)

Oswago Universitet: RESTful Service Best Practices: Recommendations for Creating Web Services: See <http://cs.oswego.edu/~alex/teaching/csc435/RESTful.pdf>

Usefull tools (Postman & Fiddler): See [Tools.htm](#) (tool #3 & tool #4)

This Assignment: RestService#2

Purpose

The purpose of this assignment is bring you a step forward to more advanced URI's and REST Services.

Mission

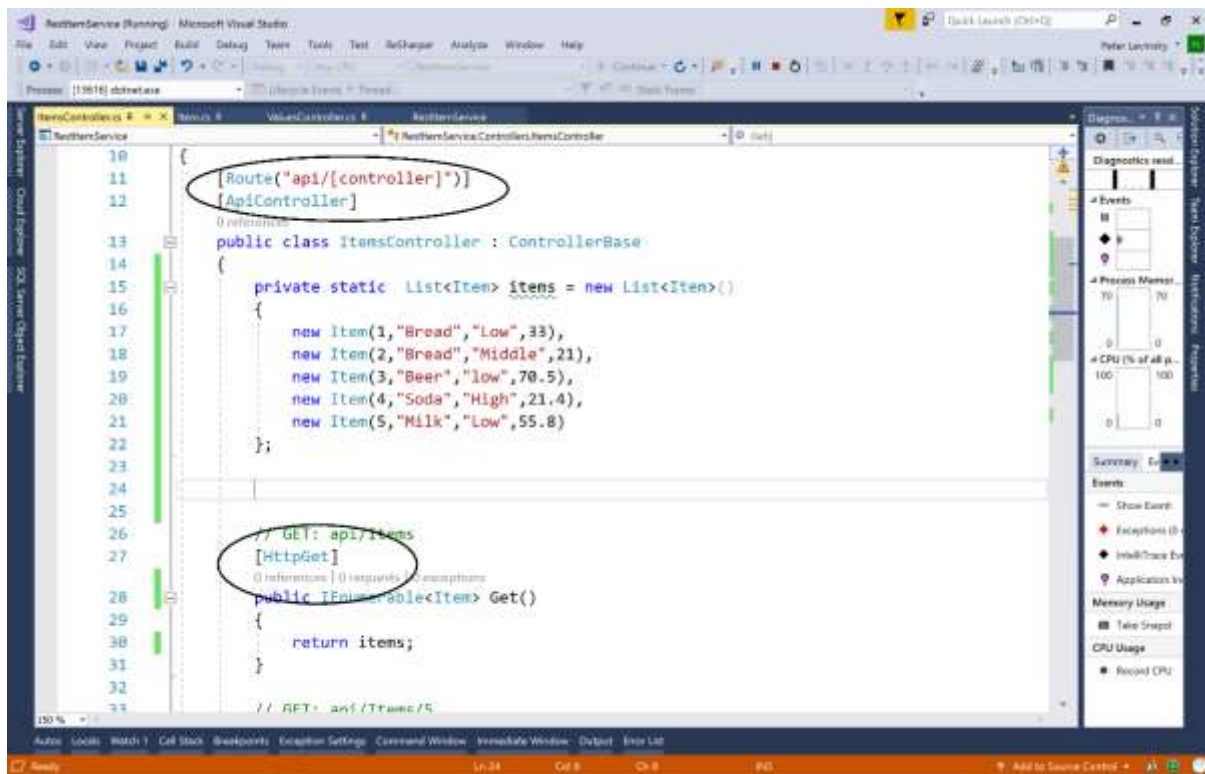
You are to improve your former Restful web services based on the ASP.Net Core services. You will better understand URI's, how to design and maintain them. This we shall do in three steps:

1. Design and control your own URI's for your REST Service
2. Create new URI's for your REST Service
3. Creating searching through URI's to your REST Service
4. Creating status codes

Assignment 1: Design and control your own URI's

You are to refactor your REST Service, so you can design and control your URI's.

Have a look at your '**ItemsController**':



```
10 {
11     [Route("api/{controller}")]
12     ApiController
13 }
14 public class ItemsController : ControllerBase
15 {
16     private static List<Item> items = new List<Item>()
17     {
18         new Item(1, "Bread", "Low", 33),
19         new Item(2, "Bread", "Middle", 21),
20         new Item(3, "Beer", "low", 70.5),
21         new Item(4, "Soda", "High", 21.4),
22         new Item(5, "Milk", "Low", 55.8)
23     };
24
25
26     // GET: api/items
27     [HttpGet]
28     public IEnumerable<Item> Get()
29     {
30         return items;
31     }
32
33     // GET: api/items/5
```

What does “[Route(“api/[controller]”)]” mean ? and “[HttpGet]” ?
and further down in the code you have “[HttpPut(“{id}”)]” ?

These are part of the default template API controller.

You have properly guess that the Route has something to do with the URI you are using. Last assignment you were using <http://localhost:44343/api/Items> i.e. Route(“api/[controller]”) is mapped to .../api/Items because the name of the controller is ‘**ItemsController**’, the [controller] tell to use the part of the class name before the Controller as the resource name here Items.

[HttpGet], [HttpPut] ... are all the HTTP Method in the HTTP Request Header, for short Get, Put, Post and Delete

For HttpPut (like for get and delete) you have an additional part (“{id}”) to the URI, so the last part of the URI e.g. “.../api/Items/6” is in fact the id which is parsed to the method as the parameter ‘id’.

This assignment is to change these default settings into some values you self can control.

- a. Change the Route annotation to [Route(“api/localItems”)]
Run your REST Service – What happen now? What is your URI to the REST Service?
- b. Change the “[HttpGet(“{id}”, Name = “Get”)]” annotation into two lines:
[HttpGet]
[Route(“{id}”)]
- c. Do the same for HttpPut and HttpDelete

Try this refactored REST Service using Postman or Fiddler. – Does it work?

Assignment 2: Create new URI’s for your REST Service

In the template controller you have five methods, which in some cases could be insufficient, then you need to create new method(s) and thereby new URI’s to your controller to meet your needs.

- a. Create a new method to get Items where the name contains a requested string.
 - a. Create a new method ‘**GetFromSubstring**’ in your Controller class like:
public IEnumerable<Item> GetFromSubstring(String substring)

Hint for code: *items.FindAll (i => i.Name.Contains(substring));*

- b. Design the http method – in this case Get
use the annotation [HttpGet]
 - c. Design the URI – in this case

E.g. use the annotation `[Route("Name/{substring}")]`

Note the parameter `{substring}` must be exact the name of the parameter of the method.

The URI will then be `'.../api/localItems/Name/theRequestedSubstring'`

- d. Try this new REST Service through a browser.
- b. Create another method and design http method and URI to get all Items with quality 'Low', 'Middle' or 'High'.
- c. Consider (not implement) how to design that you can get Items of either over some low quantity value OR below under some high quantity value OR both over a low quantity value and below a high quantity value.

Assignment 3 (Advanced): Creating searching through URI's to your REST Service

If you should design URI's for searching you should use the question-mark (?) in the URI.

E.g. `.../api/localItems/search?LowQuantity=22`

Or `.../api/localItems/search?HighQuantity=40`

Or `.../api/localItems/search?LowQuantity=22&HighQuantity=40`

This assignment is addressing this issue.

- a. To begin with you need a model class to contain information listed after the question-mark. You go back to your '**ModelLib**' and create a new model class e.g. '**FilterItem**' with the properties '**LowQuantity**' and '**HighQuantity**' – remember to compile (build).
- b. In your controller class create a new method e.g. '**GetWithFilter**' it should look like:

```
public IEnumerable<Item> GetWithFilter([FromQuery] FilterItem filter)
```

– Note the annotation `[FromQuery]` meaning it takes the information from the URI / URL as shown above.
- c. Design the Http method and the URI
- d. Make the code in the method to pick all items matching the FilterItem, be aware if e.g. LowQuantity is not part of the search string the property in the FilterItem-object is default (here 0 (zero)).
- e. Try your refactored REST-Service from a Browser (if it's a Get-method) or from Fiddler / Postman.

Assignment 4 (Advanced): Creating status codes

The next step is to send back appropriated status code. Look here to see the possible status codes https://en.wikipedia.org/wiki/List_of_HTTP_status_codes .

You are to change your method by

- a. first declare possible return statuscode
- b. change the return type into IActionResult
- c. In the code return different values

Let's take the Get by Id as an example:

- a. Declare return values

```
[HttpGet] // Http request Method
[Route("{id}")] // Http request URI
[ProducesResponseType(StatusCodes.Status200OK)] // http response statuscode
[ProducesResponseType(StatusCodes.Status404NotFound)] // http response statuscode
```

- b. Change the return type

```
public IActionResult Get(int id)
```

- c. Return different results

```
if (items.Exists(i => i.Id == id))
{
    return Ok(items.Find(i => i.Id == id));
}
return NotFound($"item id {id} not found");
```

Run your REST service and try to get an id that exists and one that do not exists – any changes from former requests?

You can now handle more advanced URI and are ready for adding help information to your REST service