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. Testing a REST Service and publish in Azure.
4. Adding Support for CORS to the REST Service
5. Consuming a REST service from a C# Console application.
6. 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.

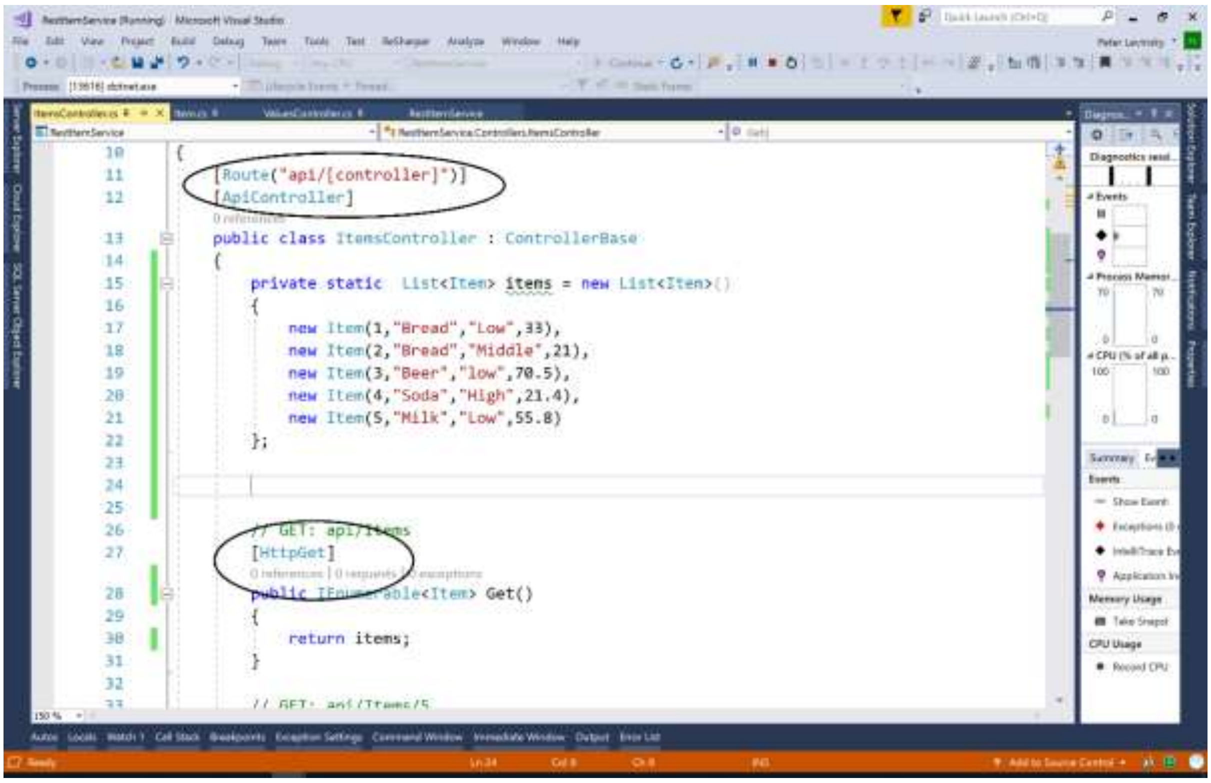
Mission
You are to improve your former Restful web services based on the ASP.Net Core services. You will better understand URI's, how to design and maintain them. This we shall do in three steps:

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**':



```
{
    [Route("api/[controller]")]
    [ApiController]
    public class ItemsController : ControllerBase
    {
        private static  List<Item> items = new List<Item>()
        {
            new Item(1,"Bread","Low",33),
            new Item(2,"Bread","Middle",21),
            new Item(3,"Beer","low",70.5),
            new Item(4,"Soda","High",21.4),
            new Item(5,"Milk","Low",55.8)
        };


        // GET: api/Items
        [HttpGet]
        public IEnumerable<Item> Get()
        {
            return items;
        }

        // 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 guessed 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?
   You could redo you changes to come back to ‘api/[controller]’

b. Change the “*[HttpGet("{id}")]*”annotation into two lines:
   [HttpGet]
   [Route(“{id}”)]

c. Do the same for HttpPut and HttpDelete

Try this refactored REST Service using Postman, Fiddler or Swagger. – 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. Though when adding methods to the controller you also need to add methods to your manager.

a. Create a new method to get Items where the name contains a requested string.
   a. Define a new methods ‘**GetFromSubstring’** in your interface like:

      IEnumerable<Item> GetFromSubstring(String substring);

      Then implement this new method ‘**GetFromSubstring’** in your manager class.
      Hint for code: *items.where(i => i.Name.Contains(substring));*

b. In the controller class; design the http method – in this case Get
   use the annotation *[HttpGet]* – you also need to make the to make the method here – like:
   ```
   public IEnumerable<Item> GetFromSubstring(String substring)
   ```

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/Items/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 an itemQuality either '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 interface and your manager class create a new method e.g. '**GetWithFilter**' it should look like:

   ```
   public IEnumerable<Item> GetWithFilter(FilterItem filter)
   ```

c. In your controller class create the REST api like:

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

   – Note the annotation [FromQuery] meaning it takes the information from the URI / URL as shown above.

d. Design the Http method and the URI

e. 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)).

f. 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 from your REST API i.e. Your controller. 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 us 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
```

Ignore *the two redlines for producesResponseType*

b. Change the return type

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

c. Return different results:
1) In the manager, you could refactor the code to

```
if (items.Exists(i => i.Id == id))
{
   return items.Find(i => i.Id == id);
}
throw new KeyNotFoundException();
```

2) In the controller, you could refactor the code to

```
try{
   return ok(mgr.Get(id));
}catch(KeyNotFoundException knfe){
   notFound(knfe.Message);
}
```

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

For more information of typical Status Codes responses see https://www.c-sharpcorner.com/article/frequently-used-status-code-and-how-to-return-them-from-asp-net-core-web-api/

You can use a more explicit form for returning status codes, instead of using the predefined like ok(…) or notFound(…).
This is simply `return StatusCode(HttpStatusCode.<anycode>);`

Assignment 5 (extra): using ranging

Look in **http://cs.oswego.edu/~alex/teaching/csc435/RESTful.pdf** pages **26**

Make support of pagination in your rest service e.g. only show 5 items at the time

You are to implement the header reading – see:

- Get Value https://stackoverflow.com/questions/3530041/getting-a-request-headers-value
- Set Value https://stackoverflow.com/questions/50630398/how-to-set-response-headers-in-net-core-2-0-service/53618771

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