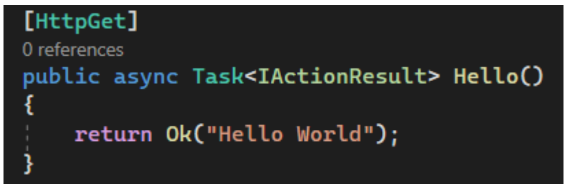
001_CWRK:Distributed Systems Assessed Coursework Report
Stefan Citiriga 2022/23

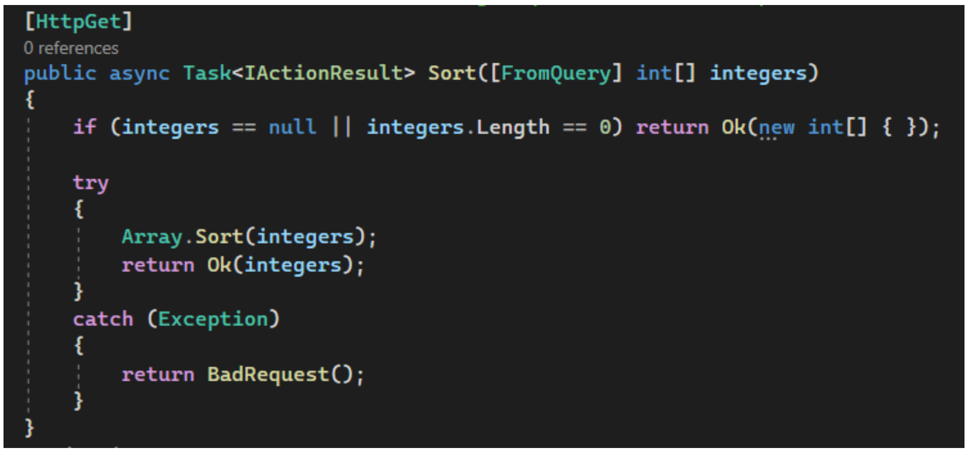
My API has multiple endpoints that can be accessed by sending requests to their respective URIs. It has 3 different controllers, and each controller responds to requests via action methods. The general URI route is "/api/[Controller]/[Action]", which is inherited by all the controllers. The server is processing requests asynchronously so it can get requests from different clients at the same time. This API is stateless because the server does not store any information specific to the client, all the necessary information is stored in the request which, at the same time, means that requests are processed independently, without the need of information from previous requests. A stateful API would have stored some piece of information about the client or about previous requests. One could argue that by changing the state of the database, there might be new resources created or removed, which technically influences further requests, which would make this server not purely stateless.

The Route Mapping has been implemented as stated before, in the BaseController, as "/api/[Controller]/[Action]". This is inherited in every other controller since every controller inherits BaseController. This has indeed made life easier. Other ways of setting up routing would have been to specify only some of the route in the BaseController (e.g. "api/[Controller]") and then have the rest of it specified as attribute before each action method.

A GET method retrieves a specified resource or piece of data from the server back to the client.

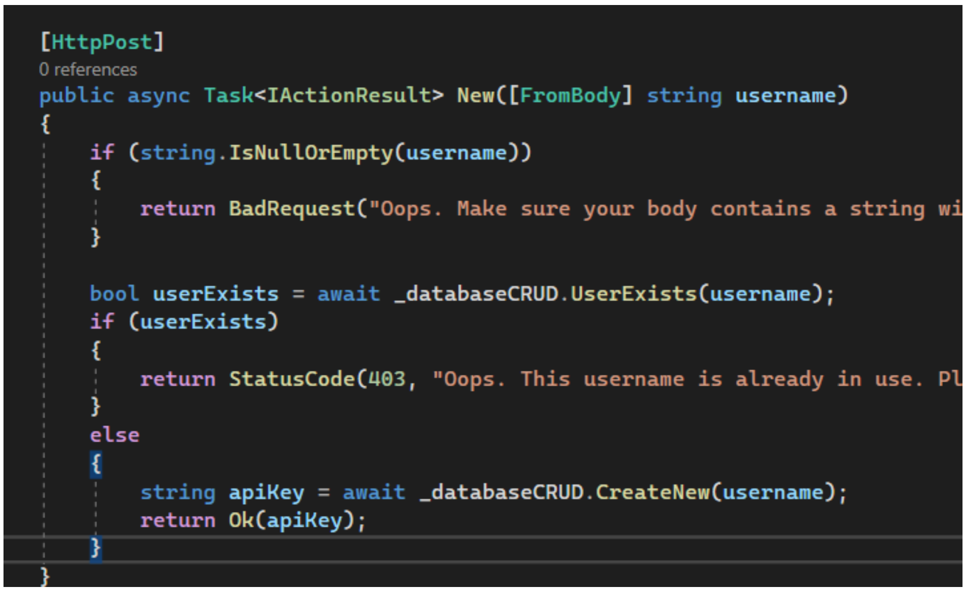


```
[HttpGet]
0 references
public async Task<IActionResult> Hello()
{
    return Ok("Hello World");
}
```

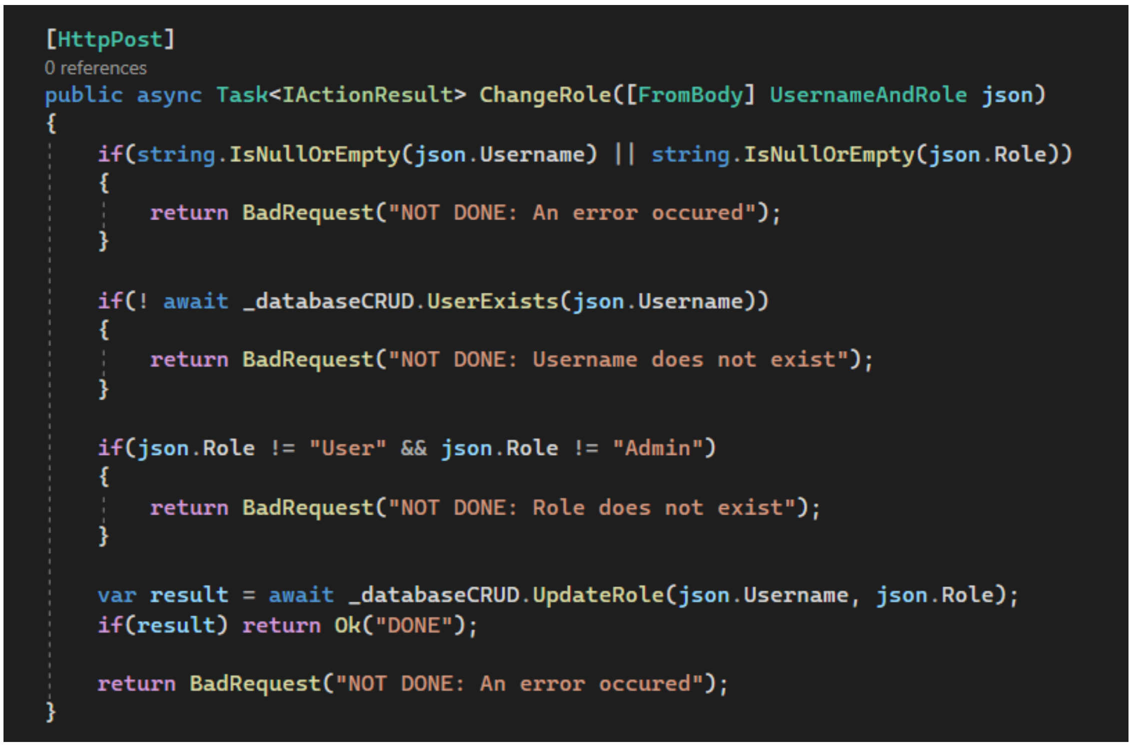


```
[HttpGet]
0 references
public async Task<IActionResult> Sort([FromQuery] int[] integers)
{
    if (integers == null || integers.Length == 0) return Ok(new int[] { });

    try
    {
        Array.Sort(integers);
        return Ok(integers);
    }
    catch (Exception)
    {
        return BadRequest();
    }
}
```

A POST method sends/submits a piece of data (in various formats) to the server, which often changes or creates resources.

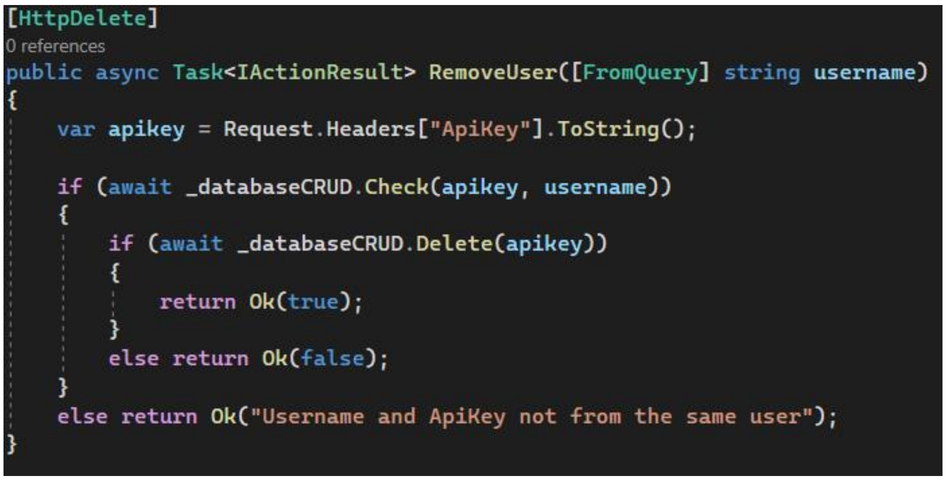


```
[HttpPost]
0 references
public async Task<IActionResult> New([FromBody] string username)
{
    if (string.IsNullOrEmpty(username))
    {
        return BadRequest("Oops. Make sure your body contains a string wi
    }

    bool userExists = await _databaseCRUD.UserExists(username);
    if (userExists)
    {
        return StatusCode(403, "Oops. This username is already in use. Pl
    }
    else
    {
        string apiKey = await _databaseCRUD.CreateNew(username);
        return Ok(apiKey);
    }
}
```



```
[HttpPost]
0 references
public async Task<IActionResult> ChangeRole([FromBody] UsernameAndRole json)
{
    if(string.IsNullOrEmpty(json.Username) || string.IsNullOrEmpty(json.Role))
    {
        return BadRequest("NOT DONE: An error occured");
    }

    if(! await _databaseCRUD.UserExists(json.Username))
    {
        return BadRequest("NOT DONE: Username does not exist");
    }

    if(json.Role != "User" && json.Role != "Admin")
    {
        return BadRequest("NOT DONE: Role does not exist");
    }

    var result = await _databaseCRUD.UpdateRole(json.Username, json.Role);
    if(result) return Ok("DONE");

    return BadRequest("NOT DONE: An error occured");
}
```

A DELETE method deletes/removes something from the server.



```
[HttpDelete]
0 references
public async Task<IActionResult> RemoveUser([FromQuery] string username)
{
    var apikey = Request.Headers["ApiKey"].ToString();

    if (await _databaseCRUD.Check(apikey, username))
    {
        if (await _databaseCRUD.Delete(apikey))
        {
            return Ok(true);
        }
        else return Ok(false);
    }
    else return Ok("Username and ApiKey not from the same user");
}
```

The Server and Client use the API key for Authentication and Authorization. Authentication consists in checking whether the API key sent in the header exists in the database, while Authorization consists of using the API key to determine the user's role ("Admin" or "User"). In the context of this project, API keys are good since they allow for building custom auth without using more robust auth mechanisms like OAuth 2.0. In the real world, I am fairly certain that authentication and authorization based on API keys is not secure enough. Two ways that I can think of in which it could be exploited are man-in-the-middle or eavesdropping attacks or just having the key compromised by the client not having it stored securely.

Database management and data access has been done as in the EF Lab but using a Transient lifetime service instead of the Singleton, for thread safety. All the methods that access the database context are in this DatabaseCRUD class. By using this class to handle all the interactions with the database and injecting it as a dependency into the controllers, data access and business logic are separated. This loose coupling allows for easy changes in the data access part of the project, without the need to change any of the other files.

Reflection:

I did not pay a lot of attention about making sure all the asynchronous code runs safely, the user/new get method would work, but I could not figure how to make both the post and the get methods be on action name "New" or how to route it otherwise. The client was easier to write than the server or at least there were less spots where I got stuck for a while.