**Razor Pages Authentication and authorization using ASP.NET Core Identity**

Razor Pages manage security using existing Asp.net Core supports for security. With Razor Pages application, you can manage Authentication, Authorization, Data protection, app secrets and many other features to build secure app and avoid security vulnerabilities. For this purpose, Asp.net Core provides **built-in Identity** **providers** that can help you build secure Razor Pages App. You have also the possibility to use other identity services like Facebook, Twitter, or LinkedIn.

Anyway, securing web applications is a large topic, with many considerations that are out of the scope of this chapter. In this chapter, we are going to look at how to implement security at its most basic security features such as **Authentication** and **Authorization**, using Razor Pages application. These 2 key concepts involve that the users should be authenticated (ensuring that you know who a given request is coming from) and their access to certain resources should be restricted so that they have access only to resources they are allowed to and with different privileges. So what are Authentication and Authorization?

## *Authentication*

Authentication is the process to figure out whether a user trying to log in is the right person. This is achieved by comparing credentials provided within a request to those stored in a trusted data store.

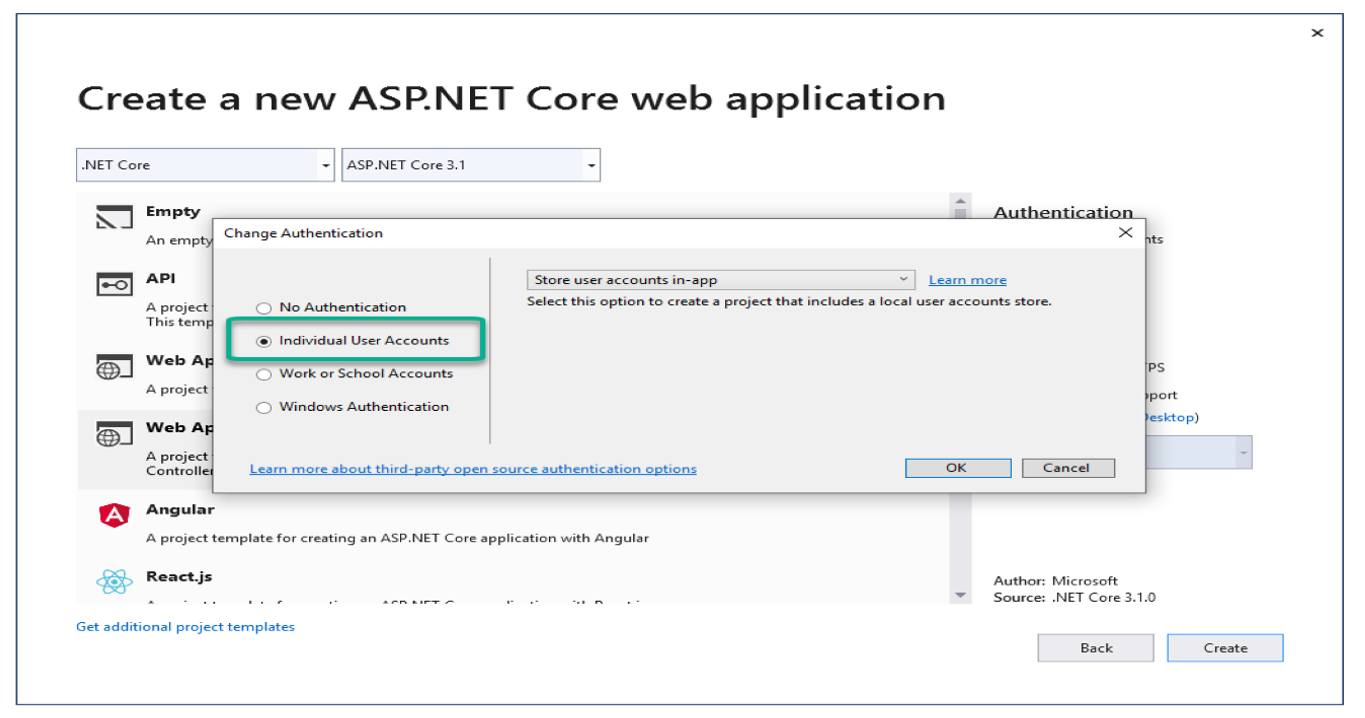
## *Authorization*

Authorization is the process that determines the area of the application the authenticated user is allowed to access to. It encapsulates rules that should be applied to control what the user can access.

## *ASP.NET Core Identity*

ASP.NET Core Identity is a membership system you can use to support Authentication and Authorization as well. As mentioned earlier, it has support for local user accounts as well as external login providers like Microsoft Account, Twitter, Facebook, Google, and more. In addition to ASP.NET Core Identity, your application can use windows authentication, or a third-party identity provider like Identity Server.

ASP.NET Core Identity is included in new project templates if the **Individual User Accounts** option is selected as shown below.



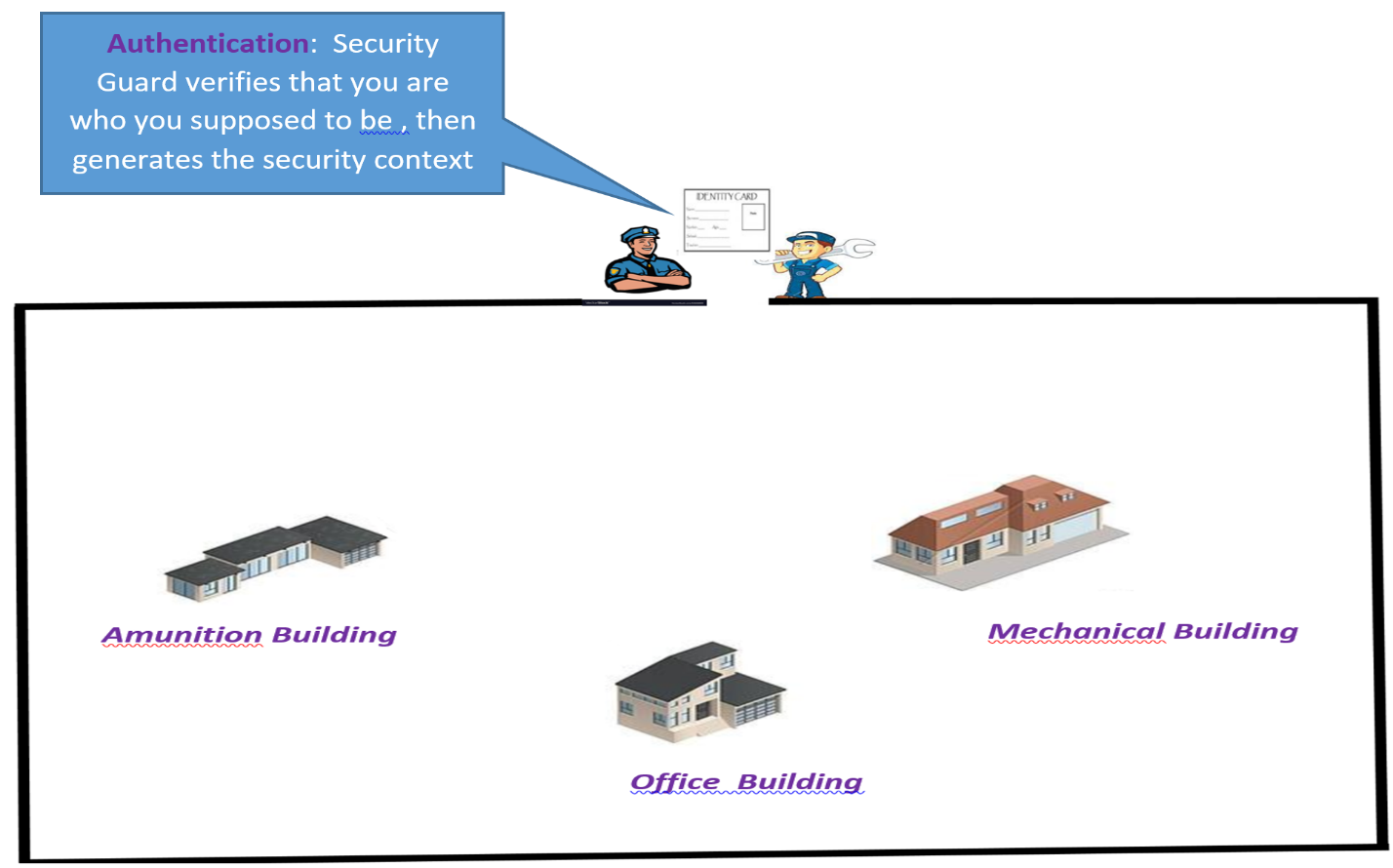
This template includes support for registration, login, external logins, forgotten passwords, and additional functionality.

However, in this chapter, you are going to learn about ASP.Net Core Identity aspects from scratch to understand how the Identity security mechanism is built. For that and before coding, I am going to take you through bunch of illustrations that will provide you with the big picture and help you understand the code.

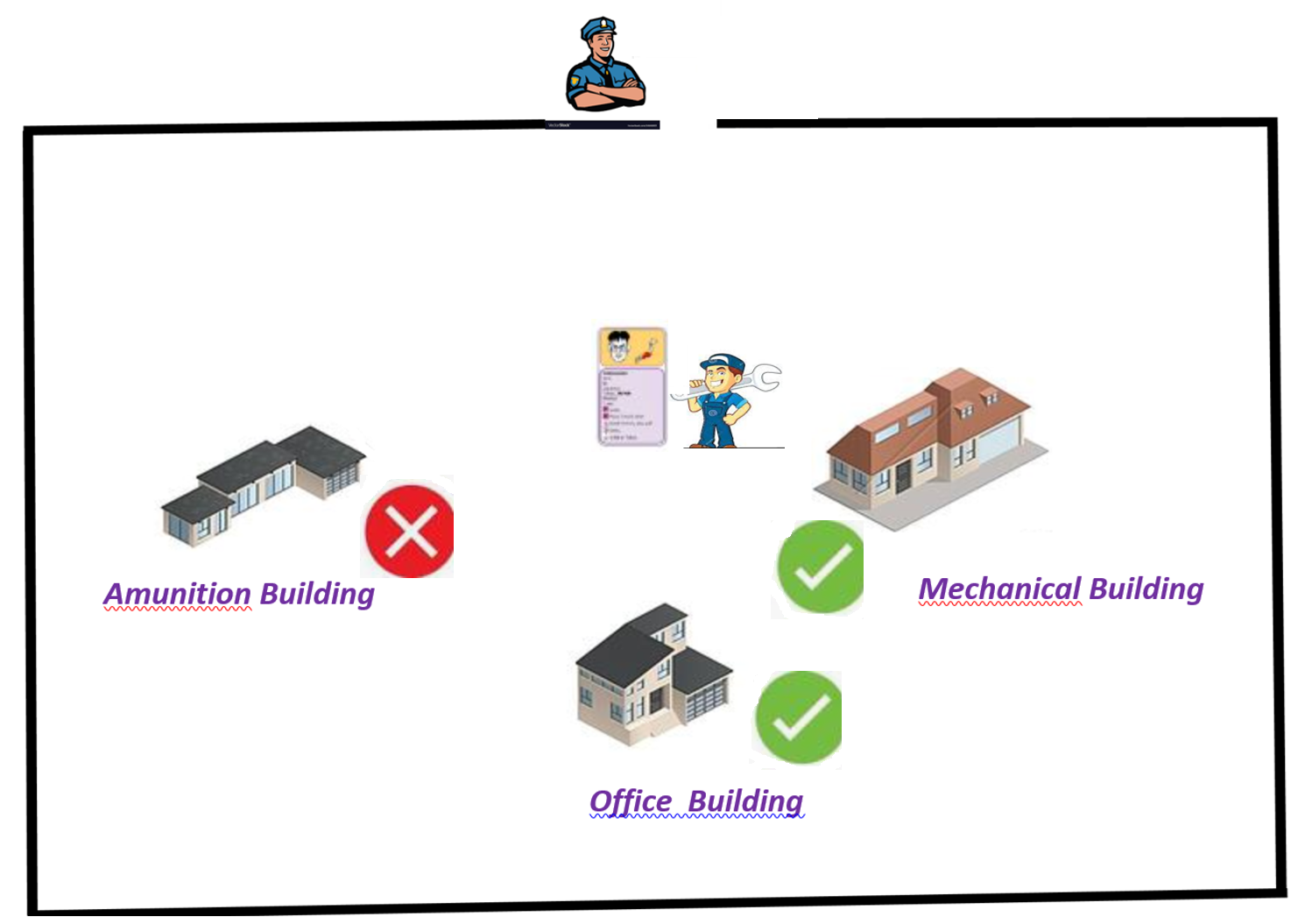
## *Authentication and Authorization*

Before digging into implementing these two security features, let us illustrate these two concepts using a real life example:

Let us suppose that you are working at a military base as a Mechanical Engineer. Your job is restricted to build and maintain large machines like generators, refrigerators and air-conditioning units. Every Morning, when you show up at the entrance of the base, you need to show your identity card ( provided by the military base) to the security Guard. This is illustrated in the figure below.



Once the security guard has verified that you are the person you supposed to be , the security guard will likely provide you with a kind of access card that will restrict your access to the ammunition building ( you have nothing to do with ammunitions) . This access card contains your identity information as well as other info about buildings you can access to and maybe how long you can access these buildings. This is what I refer to as the security context in the figure below



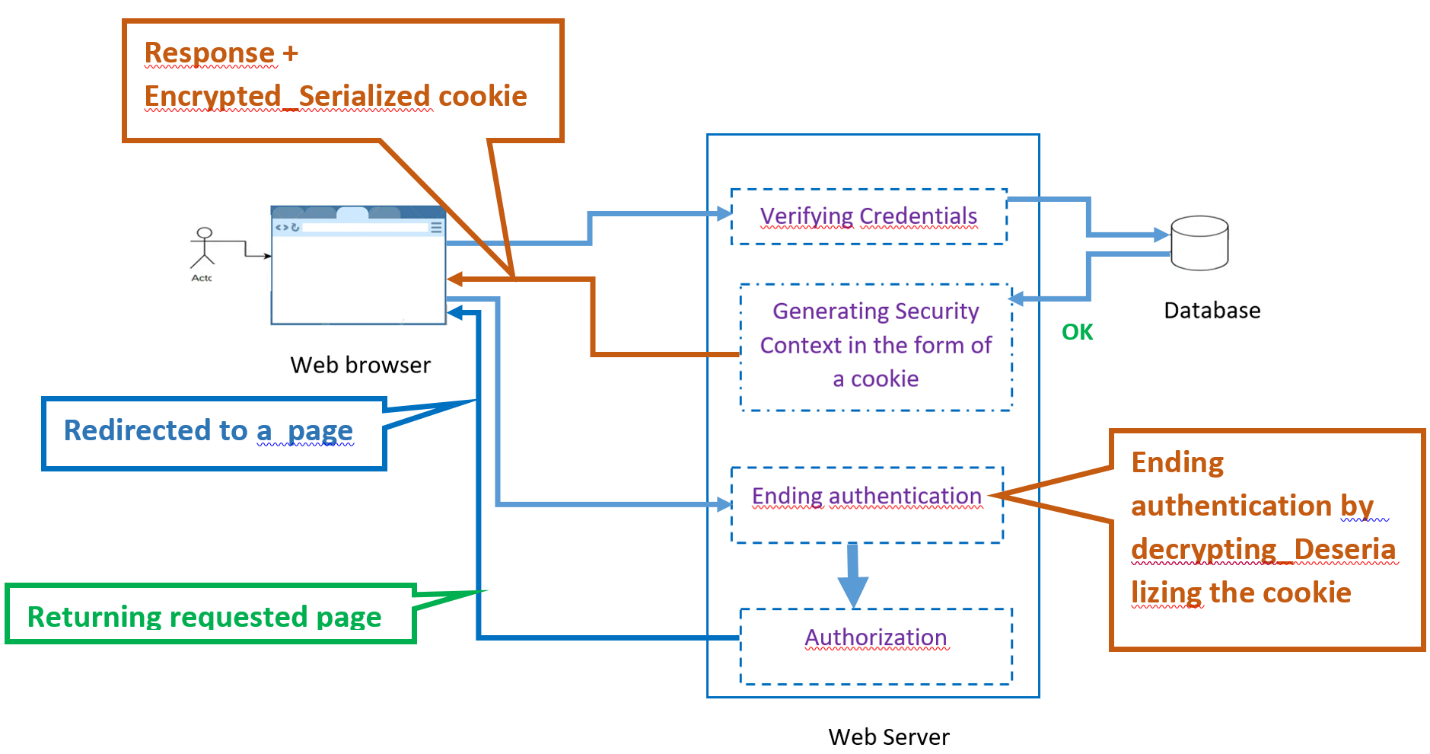
**Security Context:** All the info relevant to the military base

Now if you want to access any building, you cannot just walk-in. This is because each building has its requirements (for example, you should be the high commander to be able to access the ammunition building) and Authorization is the process of verifying that the security context satisfies the building access requirements.

***What has this to do with programming ?***

The process of being logged-in by providing the **username** and the **password** corresponds exactly to the security guard verifying that you are who you supposed to be. Then, you get what we call a security context, in the form of either a cookie-based authentication, token-based authentication or any other authentication type. In this chapter, we are adopting ***cookies based authentication***. So, once logged-in, a cookie (that contains your security context) is generated. Then any subsequent requests to access different pages will carry the cookie to verify the authorization required by each page. This is illustrated below showing the flow of sending http requests and getting http responses when accessing different pages/resources.

**What is a cookie**? It is a piece of information stored in the header of the http request and the http response that is carried back and forth between the browser and the web server.



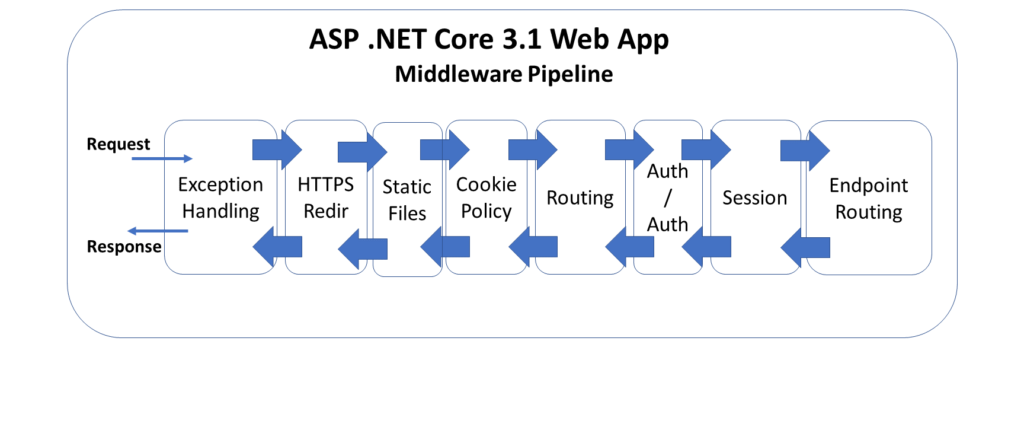
As illustrated above, the user is presented with the login page. Once the username and the password are provided, the request is sent to a server that will verify that the username and the password are valid (they exist in the database). Then the identities are pulled into the server and stored in a cookie-based security context. The cookie is then **encrypted and serialized** before appended to the header of the http response. Then upon receiving the http response you may be redirected to a page. Then, on the server side, the cookie is **decrypted and de-serialized** to get the security context. Verifying the security context will make sure that the logged-in person is who is supposed to be. This ends the process of authentication. Now , we can verify whether the user is authorized to access requested pages or not. The authorization happens right after authentication. In case, the user is authorized, the server will return the page that the user is requesting. Otherwise, the user will be denied access to the page.

**Introduction to the Middleware pipeline**

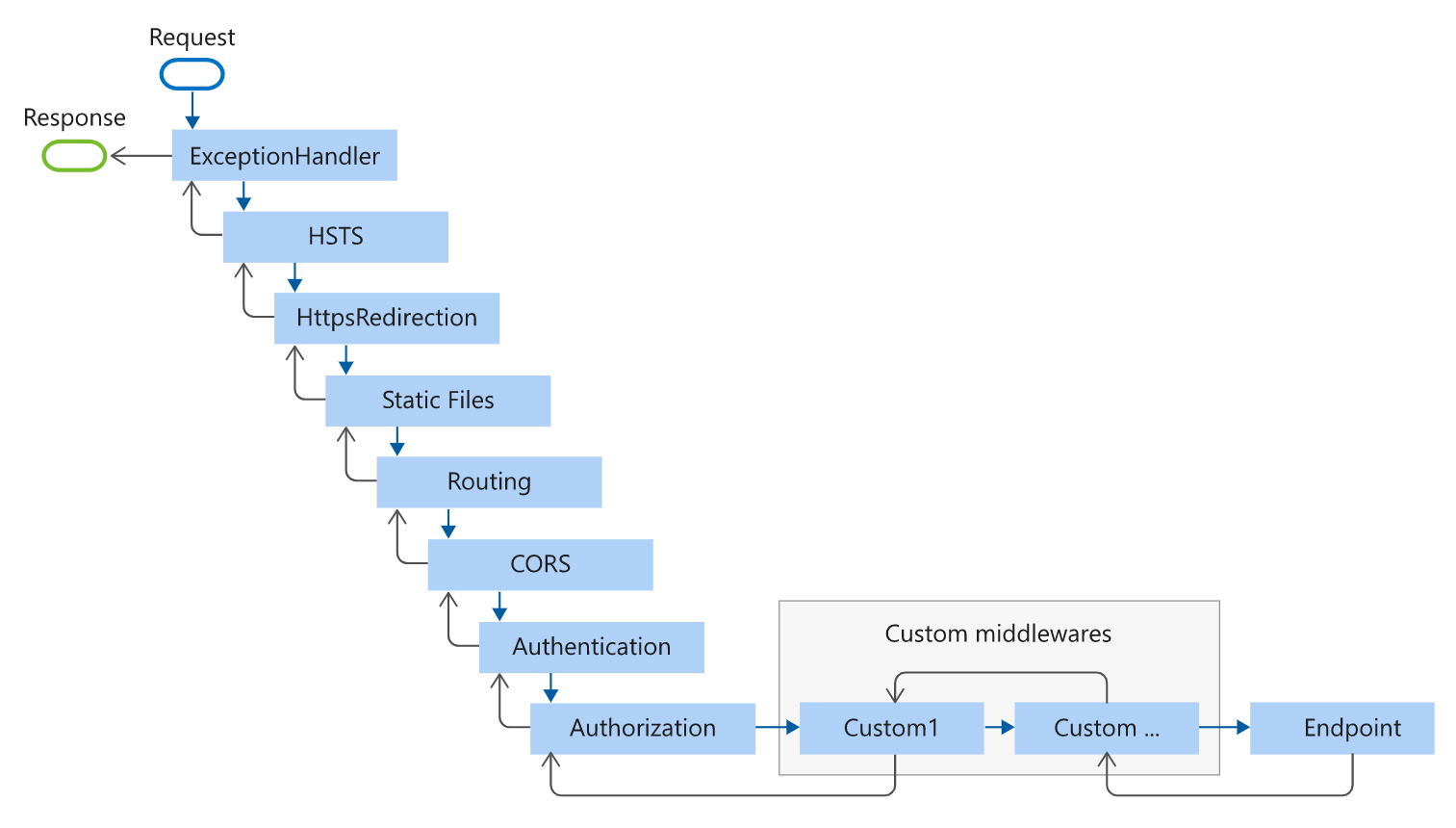
When requests are sent to a web server, they need to be processed in some way. A number of considerations need to be taken into account:

* Where should the request be directed or routed to?
* Should details of the request be logged?
* Should the application simply return the content of a file?
* Should it compress the response?
* What should happen if an exception is encountered while the request is being processed?
* Is the person making the request actually authenticated? and allowed to access the resource they have requested?
* How should cookies or other request-related data be handled?

Each of these processing actions are performed by separate components. The term used to describe these components is **Middleware**. Together, they form the request pipeline. The figure below shows the complete Asp.Net Core pipeline. The order is very important, so it is necessary to understand the placement of each request delegate in the pipeline.

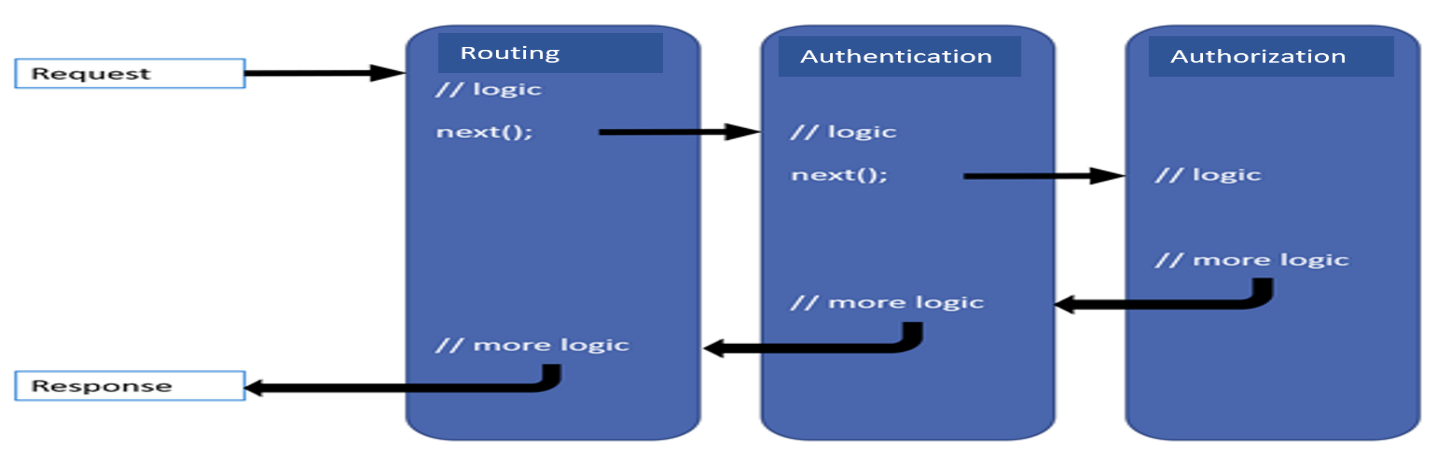


**The figure below shows the ASP.Net Core 5.0 web App Middleware Pipeline**

****

**How Middleware in ASP.Net Core Works**

When an HTTP request comes in, the first request delegate handles that request. It can either pass the request down to the next in line or short-circuit the pipeline by preventing the request from propagating further, e.g. serving static files without the need for authentication; handling exceptions before anything else, ….etc. The returned response travels back in the reverse direction back through the pipeline. As mentioned before, Middlewares are executed in the order they are added. Routing, authentication and authorization are the relevant ones in the context of securing Razor Pages app using the ASP.Net Core identity. In the following simple illustration, I only showed those middlewares.



***Routing middleware***: It is about where the request should be directed or routed to.

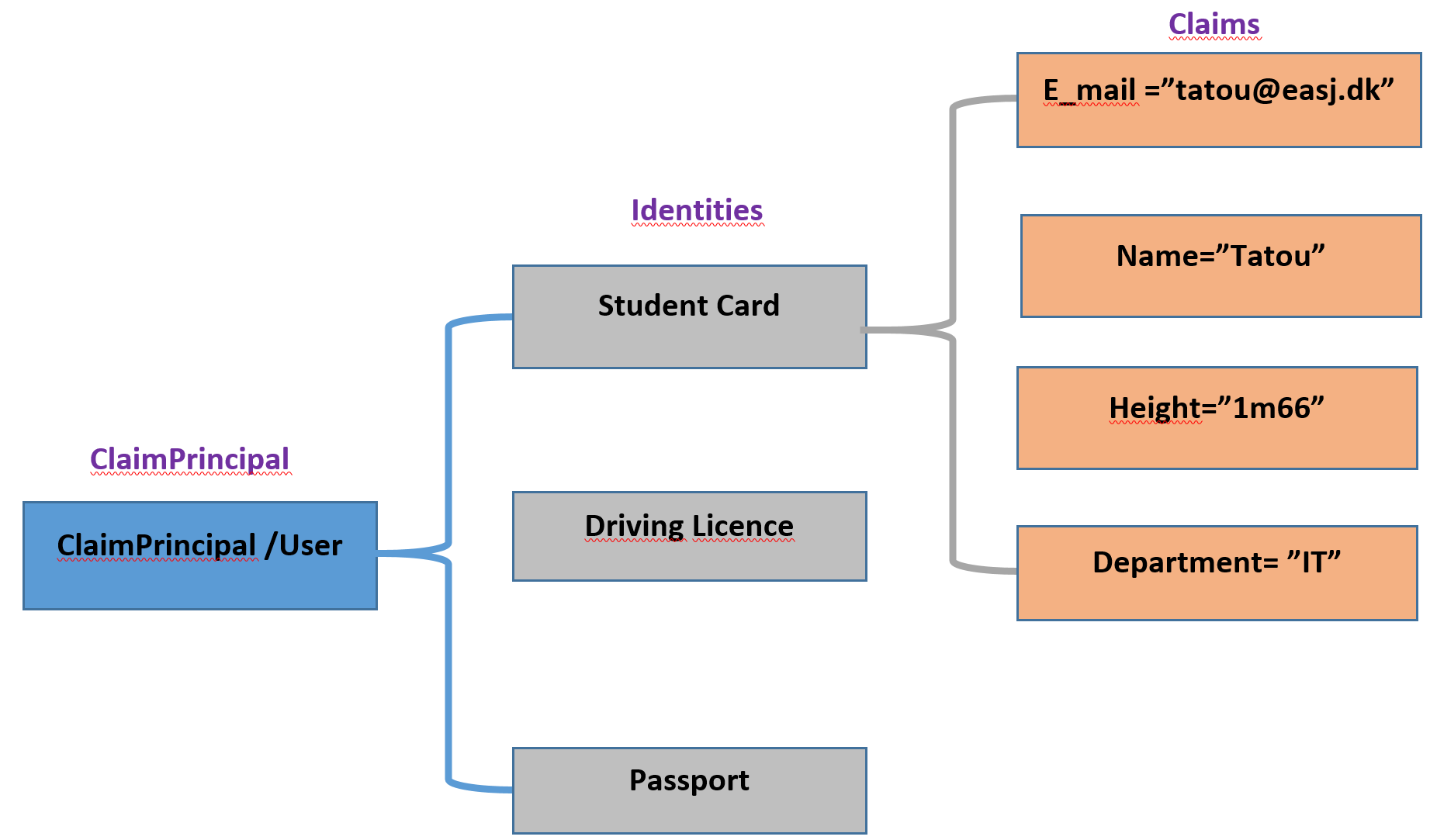
***Authentication middleware***: It is about authenticating the logged-in user.

***Authorization middleware***: It is about allowing authorized users to access the resource they have requested.

**What is the ASP.NET Core Security Context?**

Before starting the code , it is very important to understand the Security Context concept we were talking about. As mentioned before, the security context represents all the info about the logged-in user for security purposes. It could be your name ( that you have on your driving license) , your Email (that the school assigned to you) …….etc. . As illustrated below, all this info will be encapsulated in a top level object , called “**ClaimPrincipal**” in ASP.Net Core. You can basically consider the ClaimPrincipal object as the user.

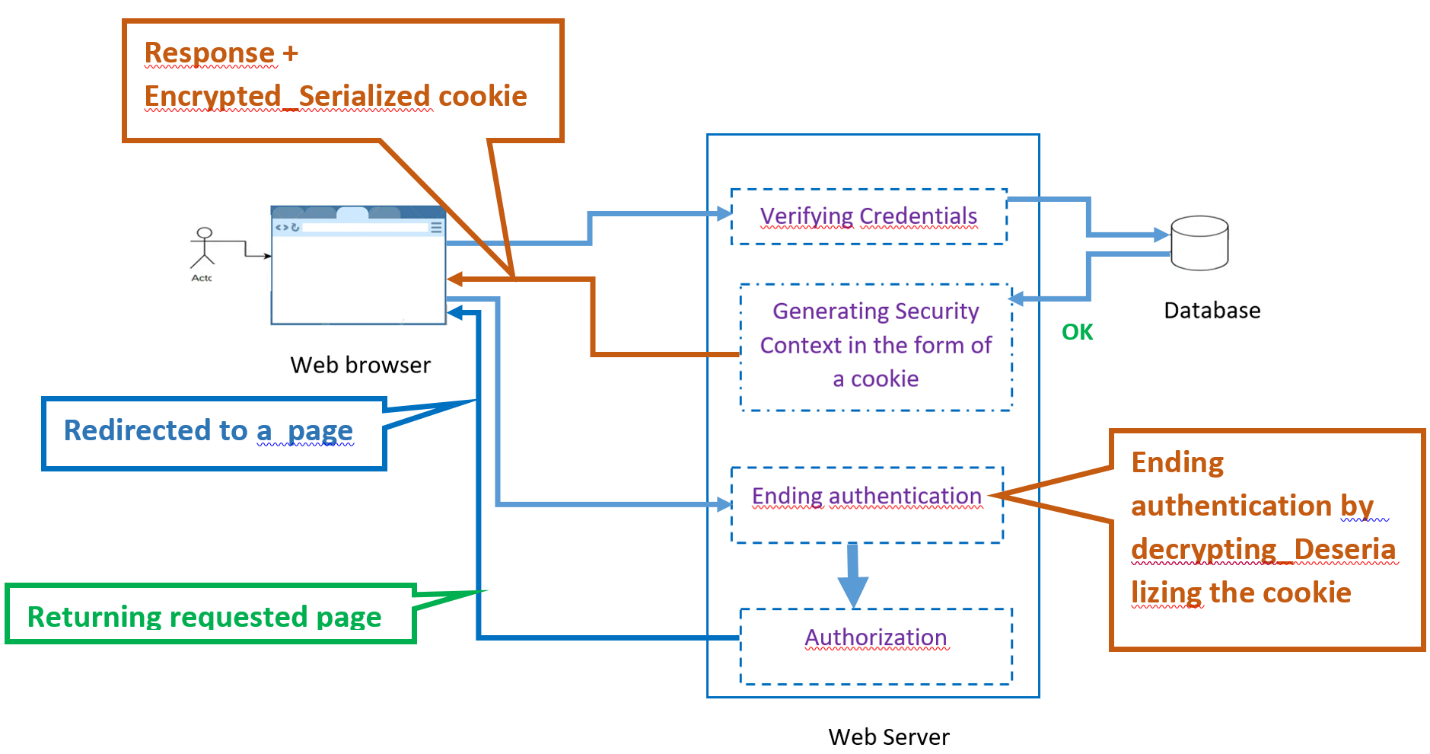
As a user can have many types of **identities** (driving license, Employee Access Card, Student Card , Passport…etc.), **a ClaimPrincipal** object represents one or many identities. Then an identity may have many **claims**. **What are claims** ? Claims are name value pairs that represent **properties** the user. For example, if you are using your Student Card as your identity, you claim that the **E\_mail** on the student card has a value “**tatou@easj.dk**” for example. This is clearly illustrated in the figure below.



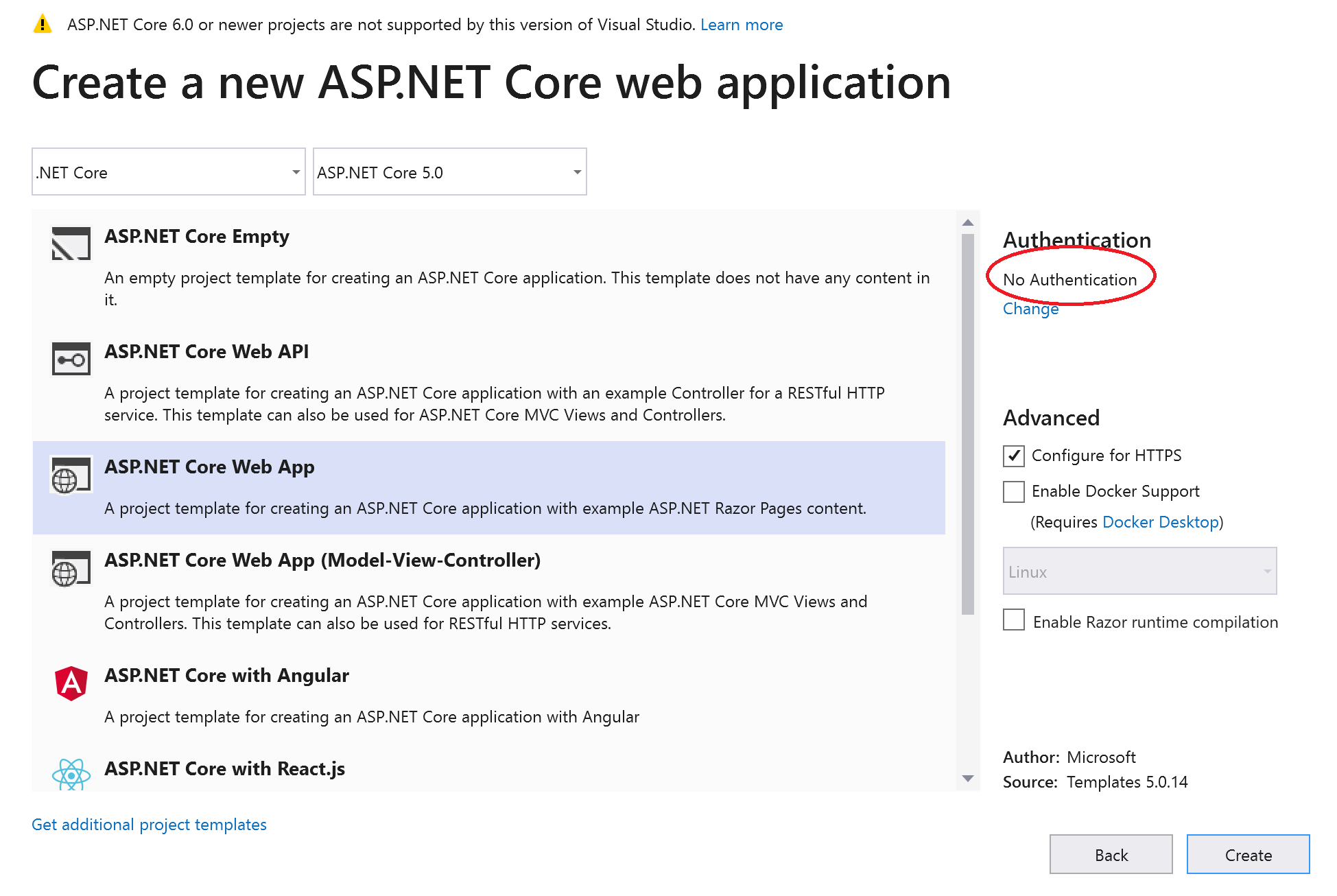
As mentioned before, claims are properties that represent the authenticated user. So , they can be used as part of authorization policies. You could create a policy called “**StudentOnly**” that requires the existence of a claim called “**E\_mail**” having the value **“tatou@easj.dk”.** Then, we apply the claim to pages that has this claim as requirement. If the claim is fulfilled, you get access to the page , otherwise you will be denied access to it. Do not worry if it is not clear yet. More details about this in the upcoming sections when implementing.

**ASP.NET Core Identity Implementation**

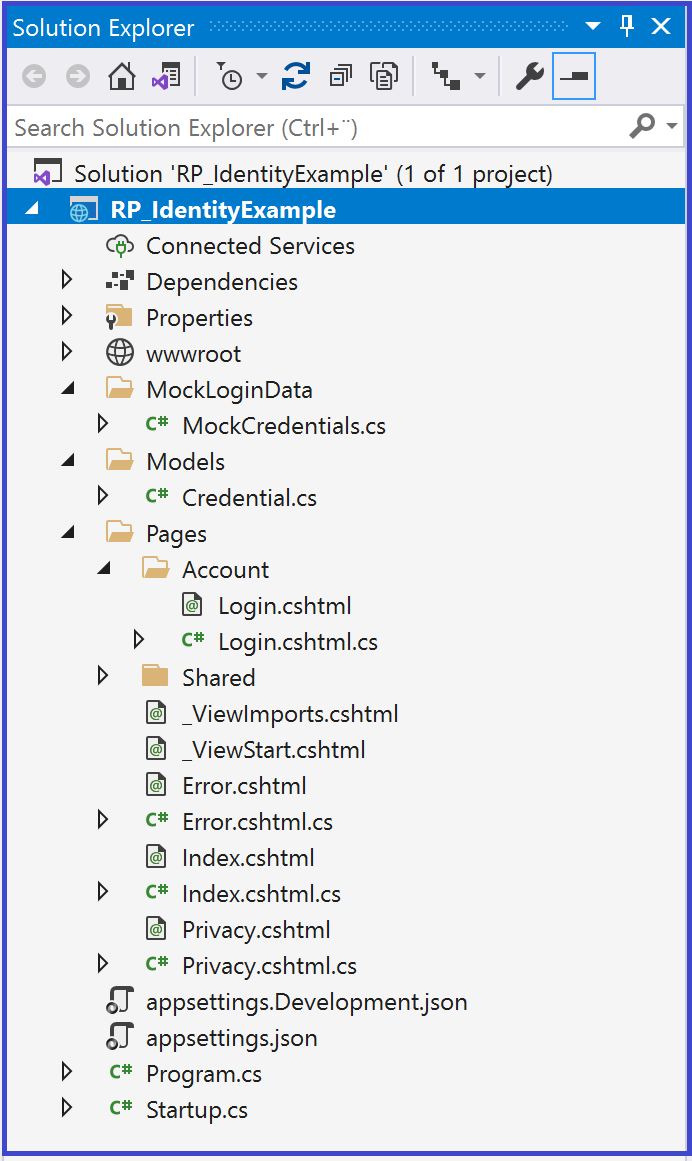
In this section, I am going to implement the following scenario: The user enters its credentials in the Login page . Once logged in, some claims are defined and the user will try to access some specific pages to which the user can/cannot access.



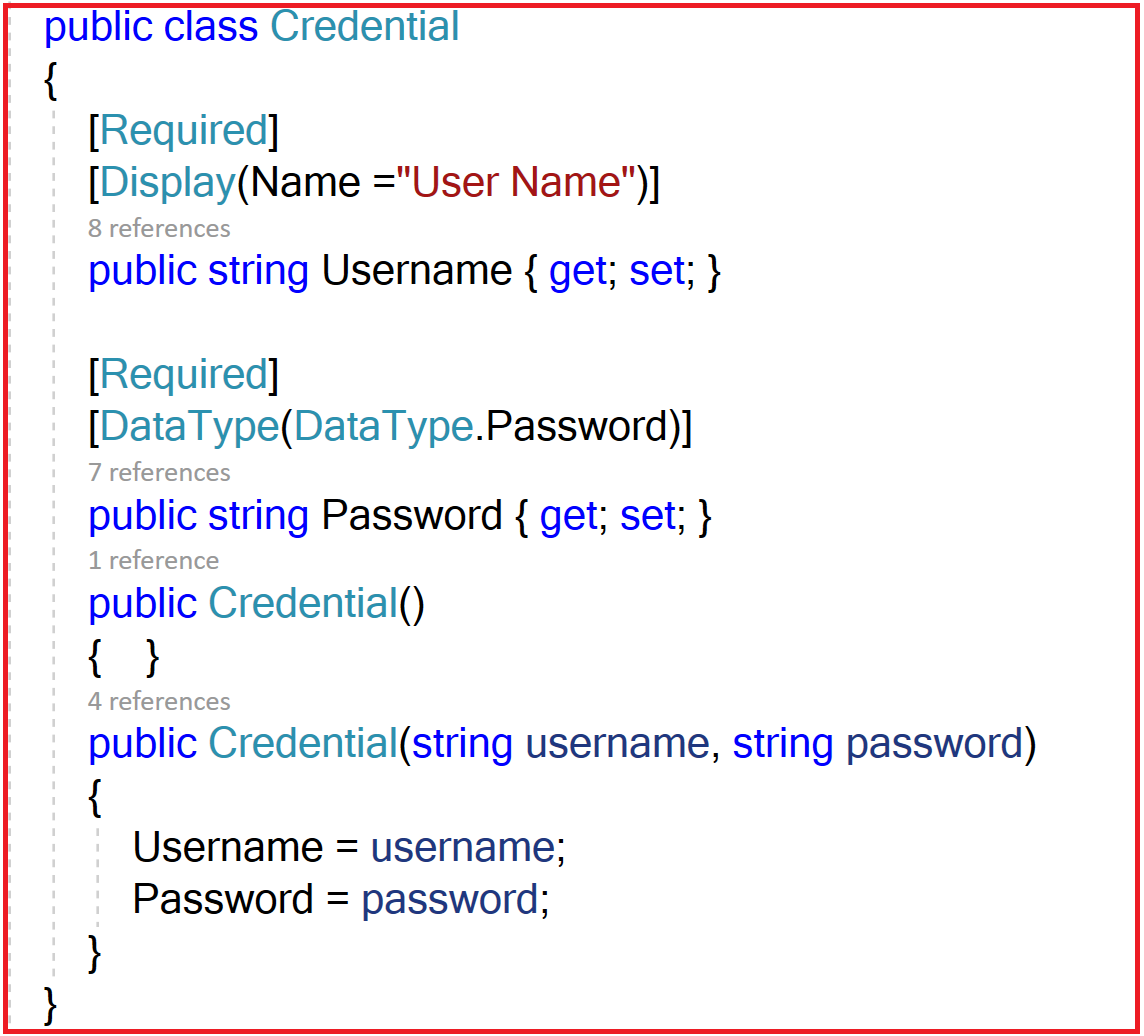
In Visual Studio 2019, I created a Razor Page application, named R***P\_IdentityExample***”



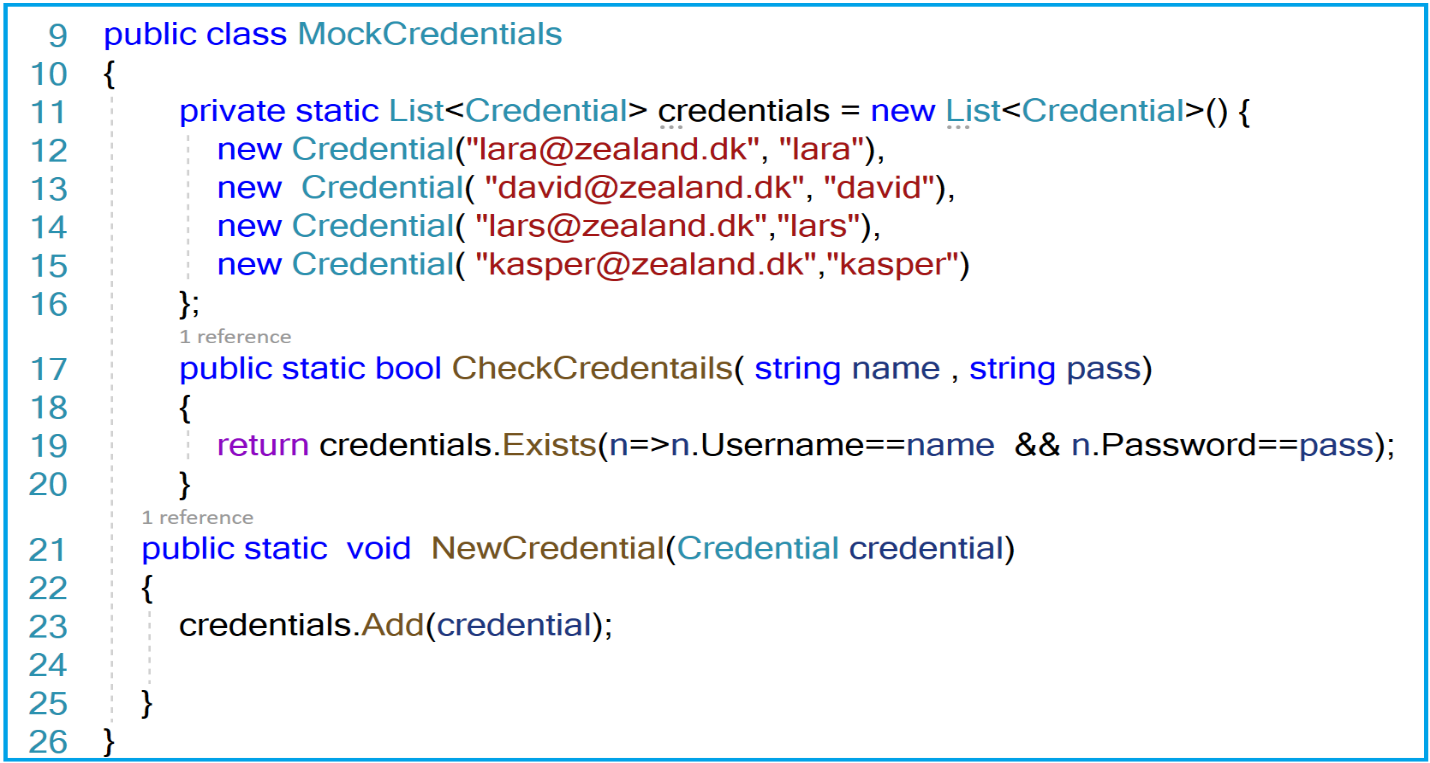
The solution structure is shown below :



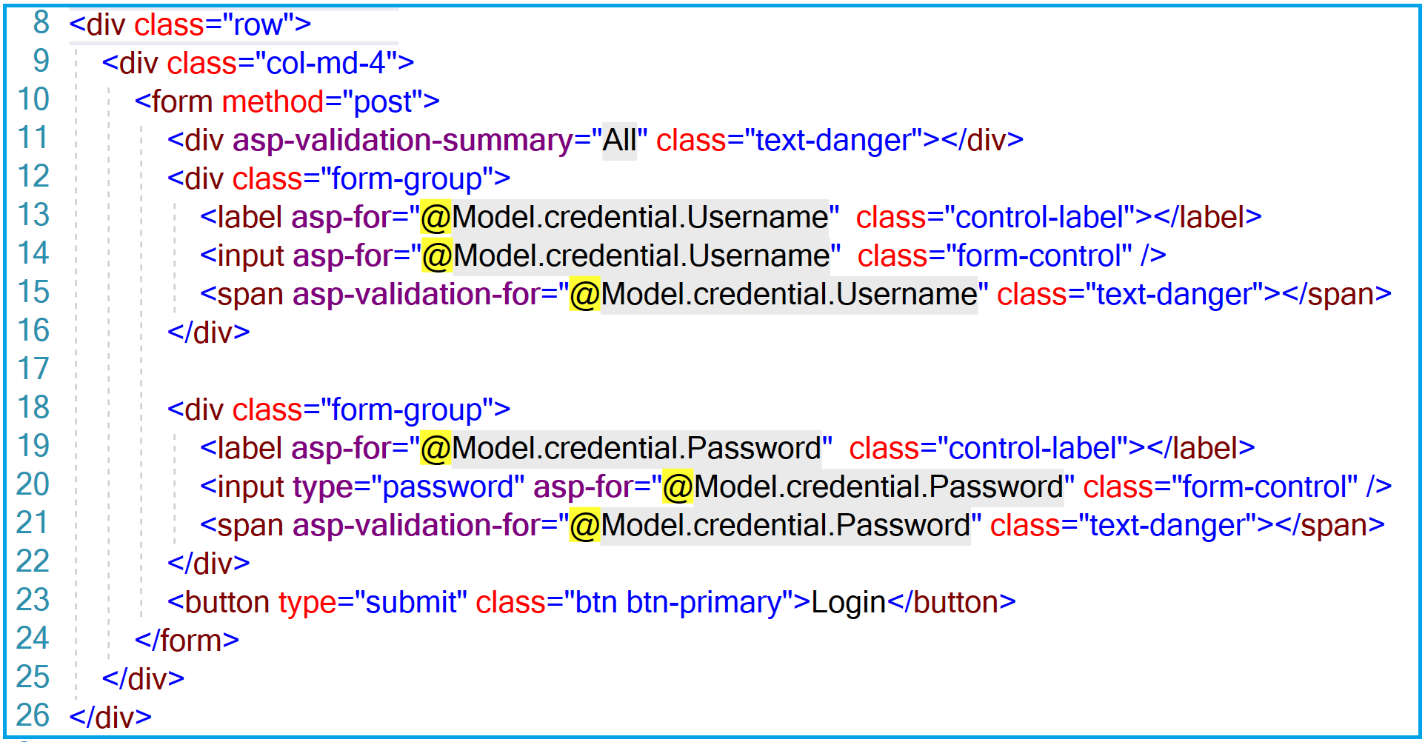
* In the **Models folder**, I created the **Credential class** that encapsulates the Username and the Password properties. As you can note, the Password is going to be shown as dots and not text. This is achieved by adding the following decoration[**DataType(DataType.**Password)] to the Password property.



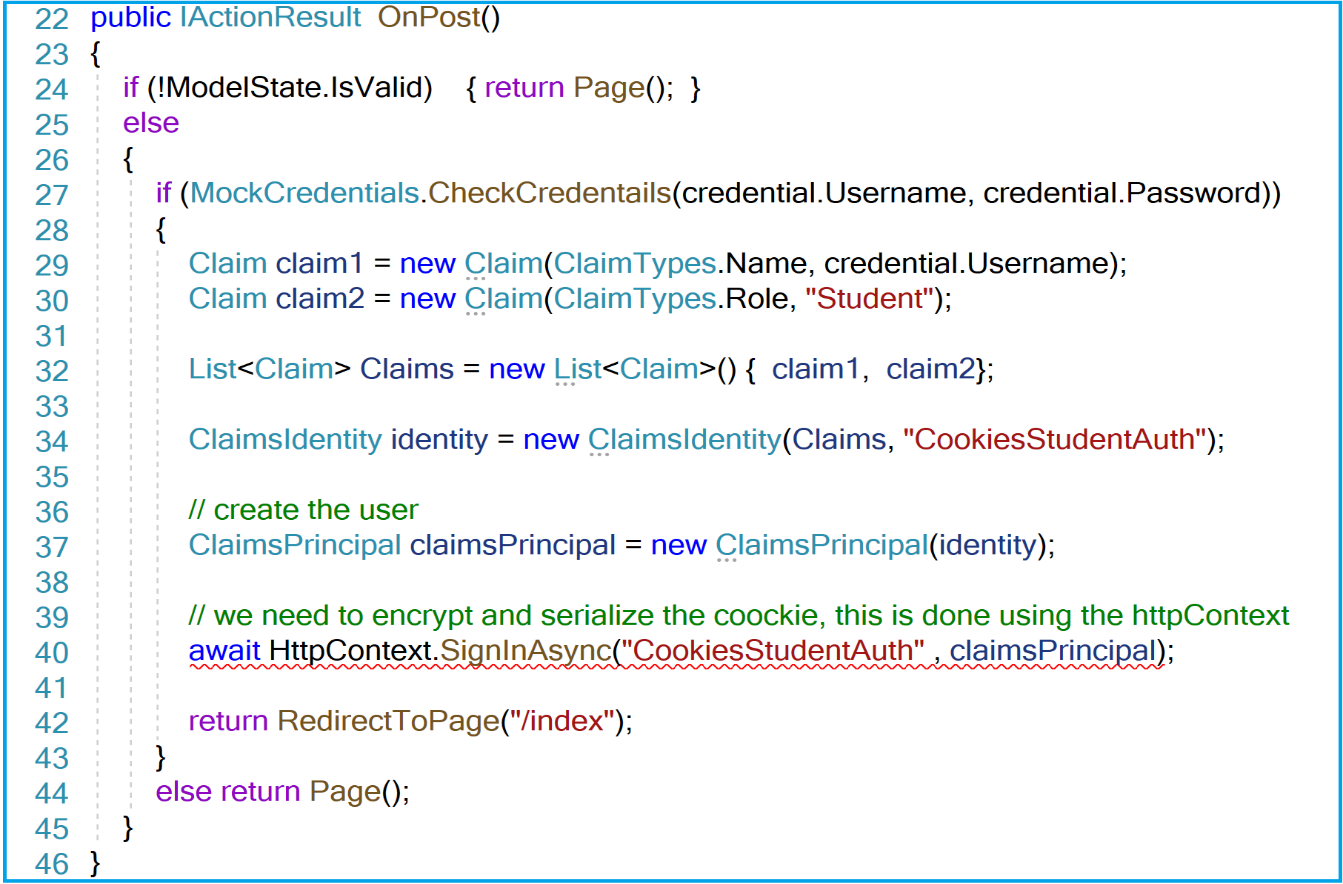
* In the **MockLoginData**, I created the MockCredentials class that contains a list of the Credential objects and a method ***CheckCredentials(string username, string password)*** that checks whether the username and the password are valid, then returns a Boolean type. I also implemented ***NewCredential (Credential credential)*** method to register a new user.



* In the **Account folder**, I created a “**Login**” Razor Page. Below is the form using the “**post**” method. So when submitting the form, the **OnPost** method is called.



Let us implement the OnPost() method. Below is the code



The code is very simple. If the **Modelstate** is valid (that is we pass the data annotations validations), we call the **CheckCredentials()** method to verify that the credentials exist. If the credentials are valid ( exist in the database), we then create the security context for the logged in user. **How**?

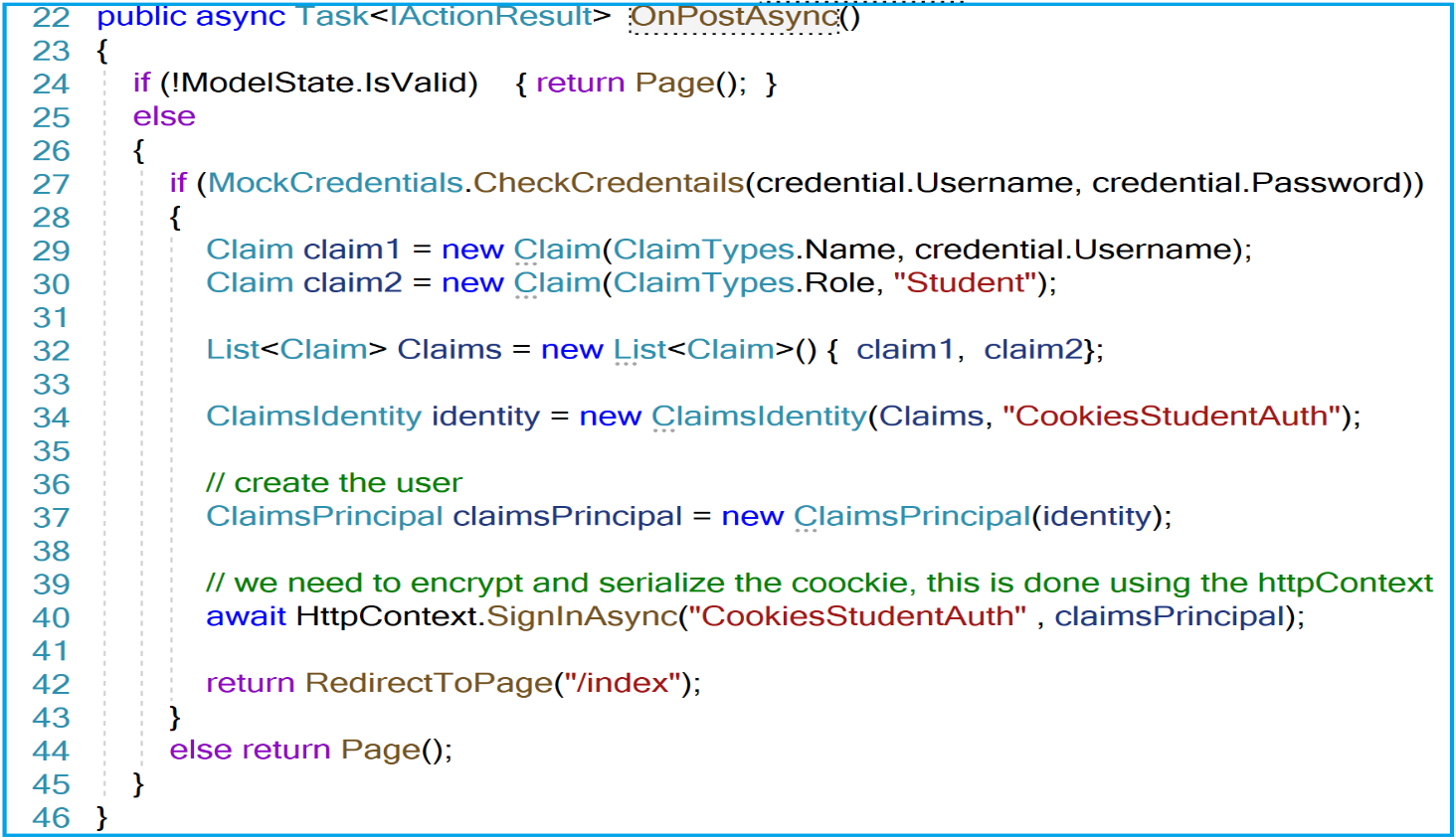
**Line 29-30** : We created 2 claims that describes some properties of the student. We claim that the name is going to be the username and the role is Student.

**Line 34**: We then created the Identity object by specifying the authentication type “***CookiesStudentAuth***”. This type will be configured in the StartUp.cs class .

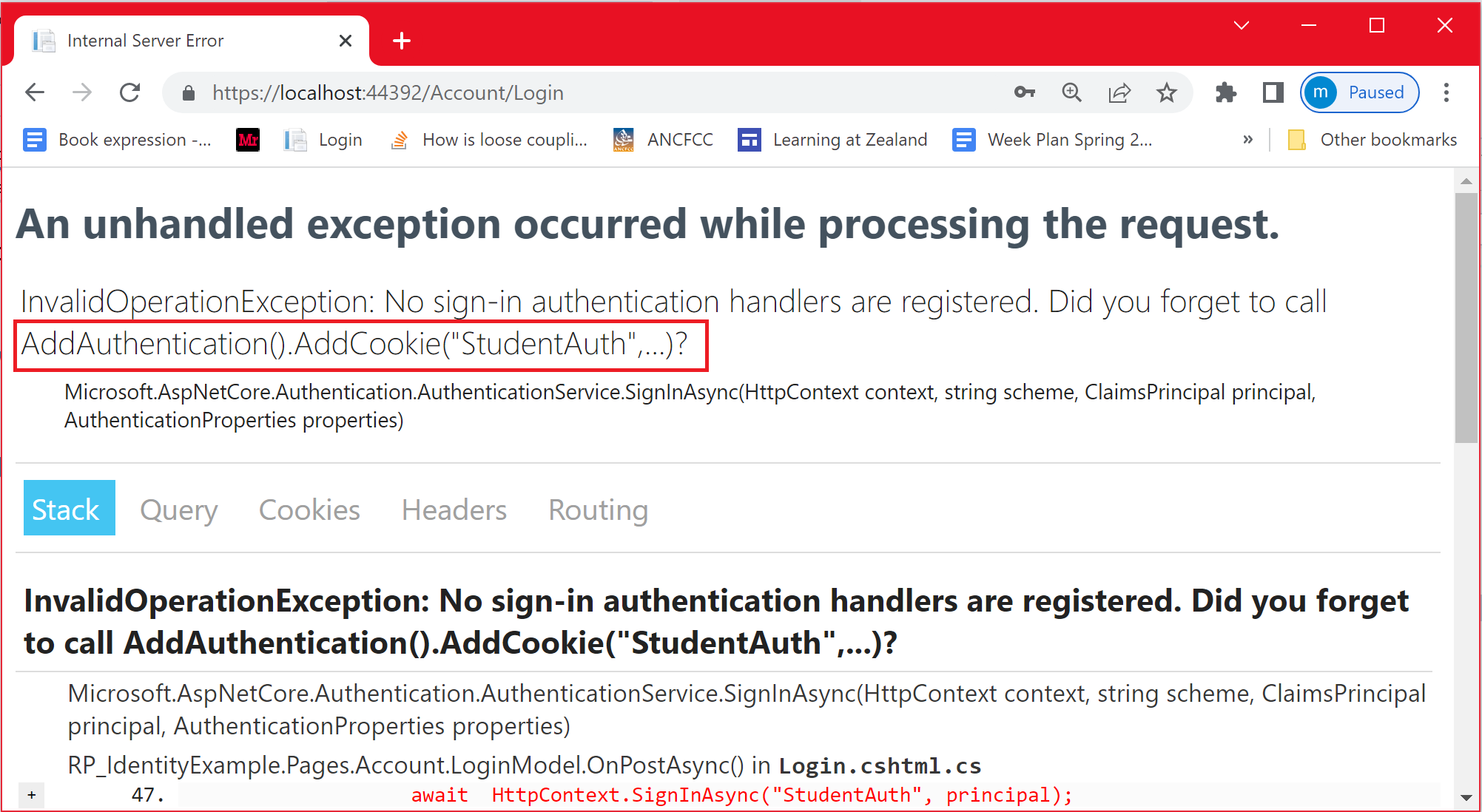
**Line 37**: Then we created the ClaimPrincipal that represents the user.

**Line 40**: The method SignInAsync() is used to sign in the user for the specified authentication scheme. This method takes care of encrypting and serializing the cookie. It is an extension method for the HttpContext object. Note that this extension method is implemented as **asynchronous** method. That is why we need to **await** its call. If you remember the class about asynchronous programming, the await keyword can exist only in an asynchronous method and this is why we have a compile error. So we need to make our OnPost() method asynchronous.

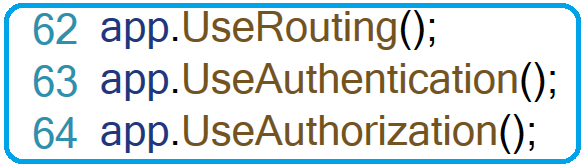
The error free implementation is shown below



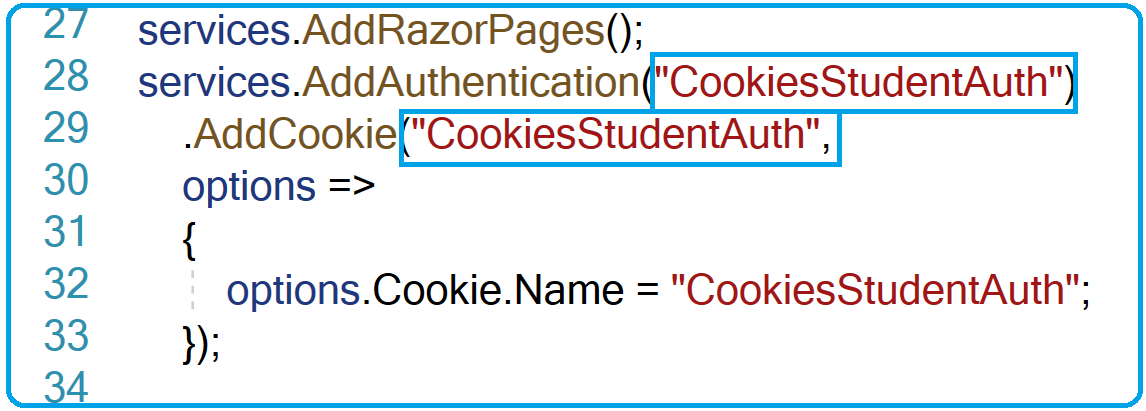
Let us run the application. The following exception is thrown



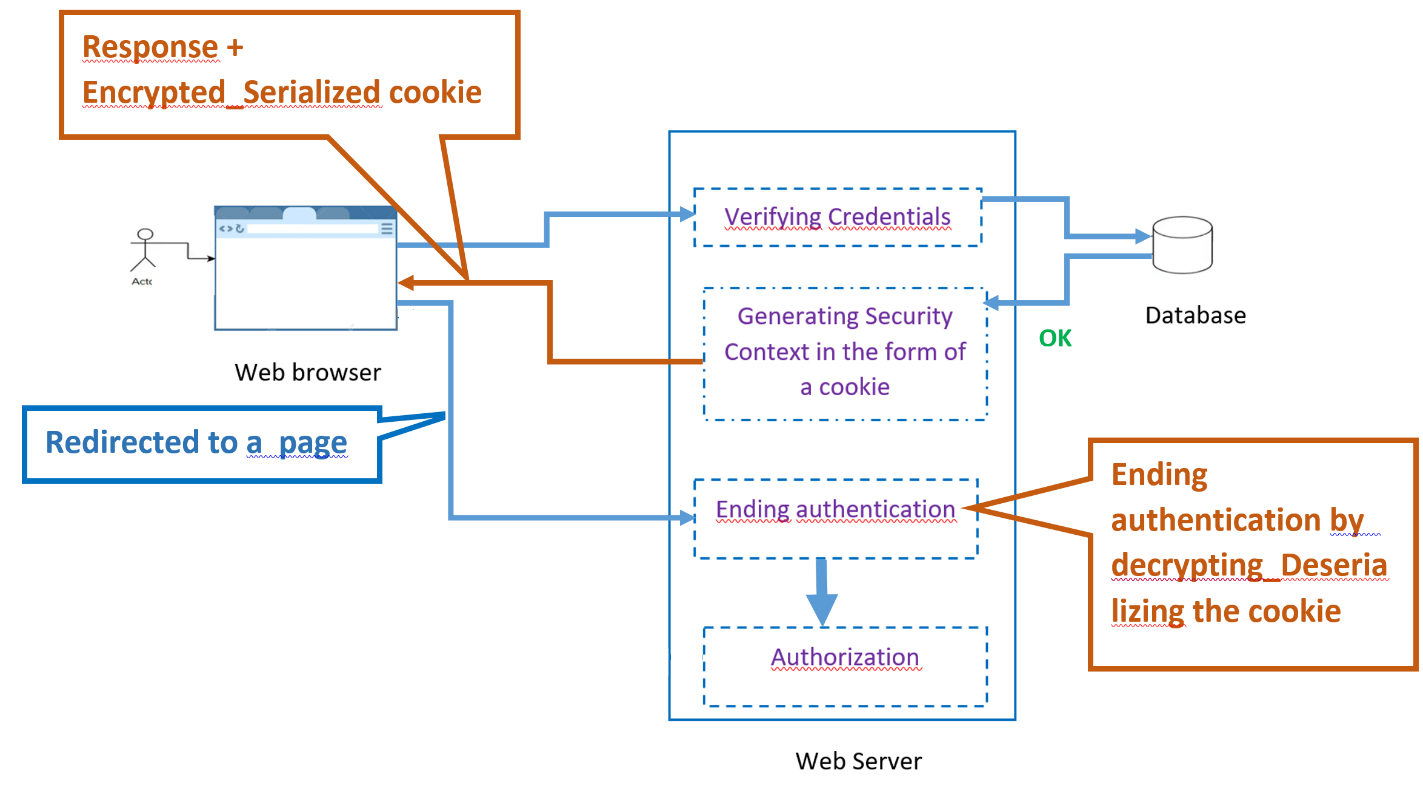
As clearly mentioned on the error message, “**no Sign\_in authentication handlers are registered**”. Before assigning an authentication handler, we should make sure that the authentication middleware is added. This is because, the authentication middleware is the one that process the request for authentication purpose. So we need to add this middleware in the Configure() method in the Startup.cs class just after the Routing middleware, as shown below:



Now, we need to specify an authentication handler (a cookie-based or a token-based or…etc). To use a cookie based authentication, we need to assign the authentication handler by calling the ***AddCookie()*** method as shown below. We provide the Cookie Authentication scheme “**CookiesStudentAuth**” to AddCookie() handler. We also indicate to the Authentication middleware to use the authentication scheme “**CookiesStudentAuth**” through the AddAuthentication() service. Otherwise, a default authentication scheme will be used.

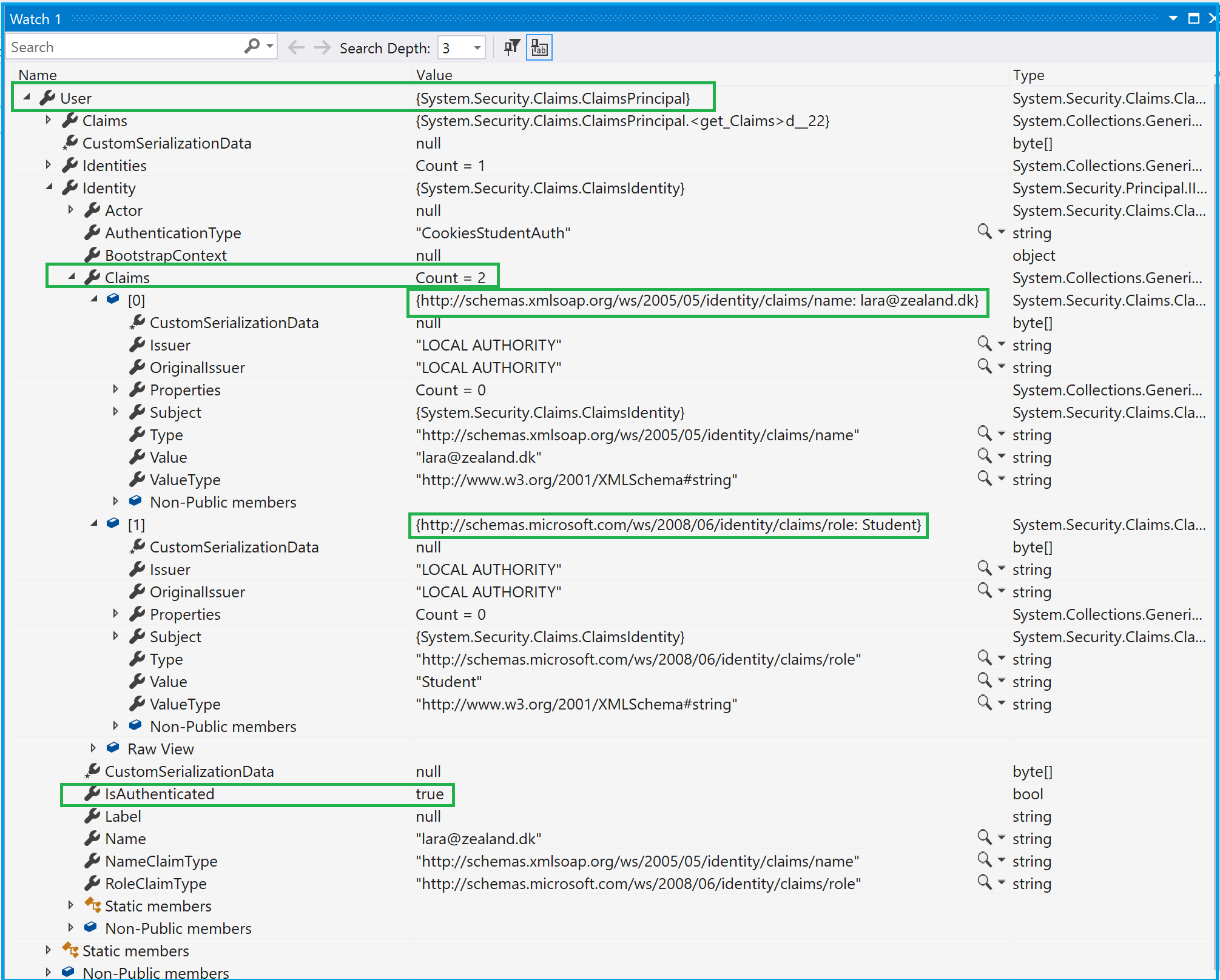


Now that all the settings are in place, let us verify that the logged-in user can be authenticated by the server according to the following scenario



Let us first remind you how things work : Once the user credentials are validated, a security context is created in the form of a cookie. This Cookie will be carried forth and back between the browser and the server. On its way to the browser, the cookie is **encrypted**, **serialized** by the authentication middleware. On receiving any subsequent request, the authentication middleware will then **decrypt** and **de-serialize** the cookie. This allows the server check whether the info (claims , identity, ..etc) encapsulated into the cookie belongs to the logged-in user. We are going to have a look at the content of the Cookie after decryption and de-serialization.

Let us run the application in Debug mode (Place a break point on the OnGet method of the IndexModel class). Get to the Login page, then enter valid credentials (for example [lara@zealand.dk](mailto:lara@zealand.dk) as username and lara as the password). Once logged in and stopping at the break point, let us watch the User object.



As you can see, the user (User) is the Claim principal. As expected, from the info extracted from the cookie, the user has two claims:

* + - name:lara@zealand.dk
    - role : Student

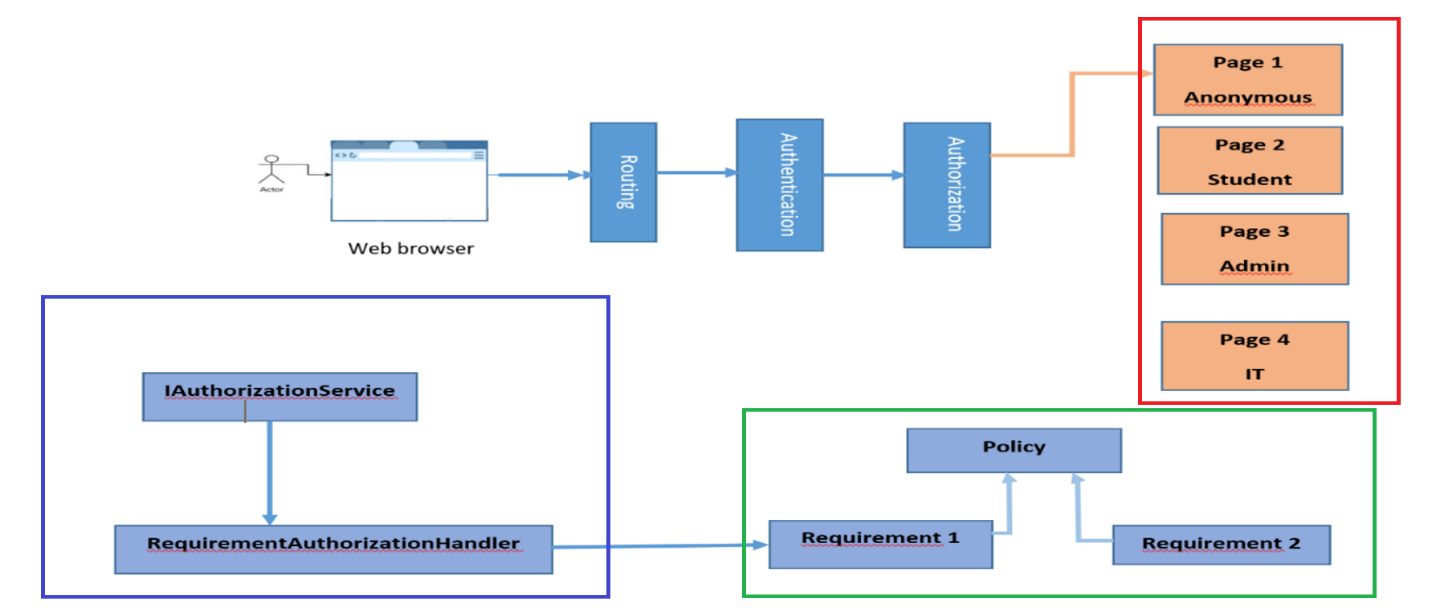
Another and a very important observation is that the property “**isAuthenticated**” set to true.

That means , once the user is logged in , you can figure out whether the user is authenticated by checking whether **User.Identity.IsAuthenticated** is true or false.

Now that the user is authenticated, the authorization process starts right away. The authorization process is about figuring out whether the authenticated user has right to access the requested page or not.

**Authorization**

Now let us move to the authorization, to figure out whether a user is restricted access to parts of your application.

****

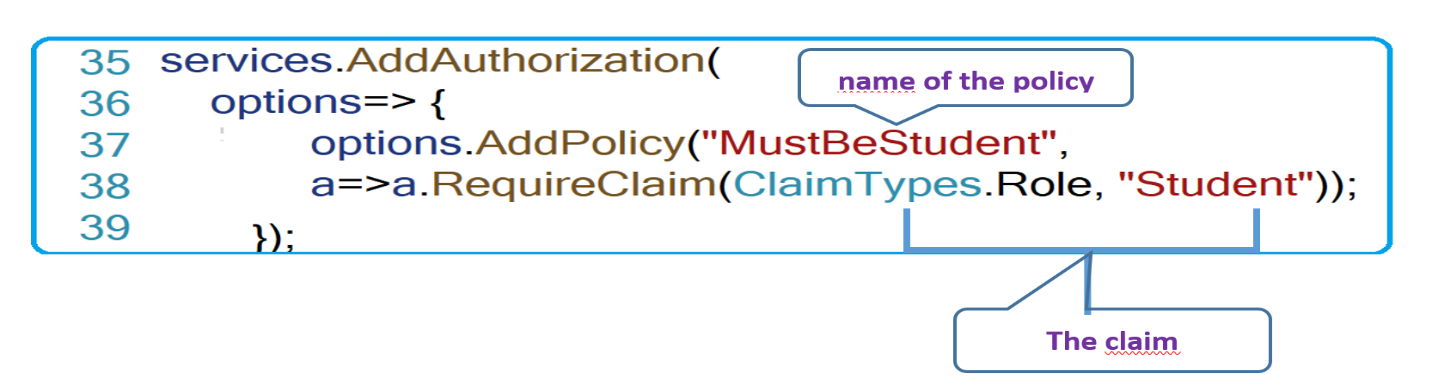
As shown above, [claims-based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/claims?view=aspnetcore-6.0) and [role-based authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles?view=aspnetcore-6.0) use three building blocks (this happens underneath the covers):

* + - Preconfigured policy
    - Requirement
    - Requirement handler

Let us look closer at every block and the orchestration of the authorization process occurs.

**Preconfigured policy**

An authorization policy consists of one or more requirements. The policy should be registered as part of the authorization service configuration in the Startup.cs class as shown below:



In the example above, a "***MustBeStudent***" policy is created. It has a requirement—that requires that the user has a Student role. Note that we could chain many requirements for the same policy

**Requirement**

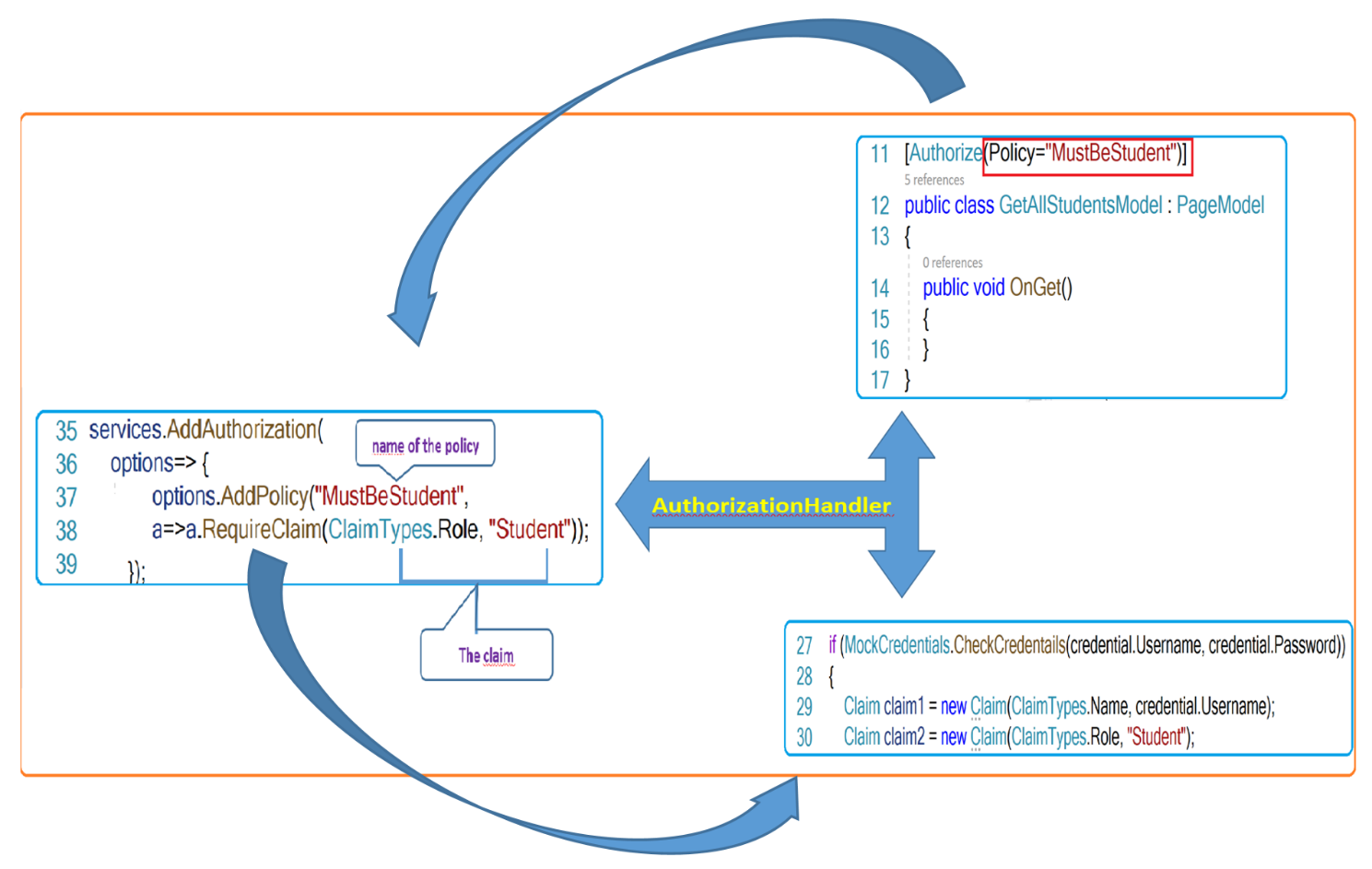
Razor Pages may have requirements for accessing them. These requirements are applied to the Razor Pages as policies by using the [**Authorize]** attribute with the policy name. The figure below shows an example of applying the previous policy to the page “***GetAllStudents***”.



**Requirement handler**

Each preconfigured requirement has a corresponding authorization handler. The main function of this handler is to look at the presence of the claims for the authenticated user . Underneath the covers, the authorization middleware uses the IAuthorizationService, which uses the RequirementAuthorizationHandler. Then, handler checks whether the authenticated user fulfills the requirements imposed by the page by comparing the claims against the page requirements.

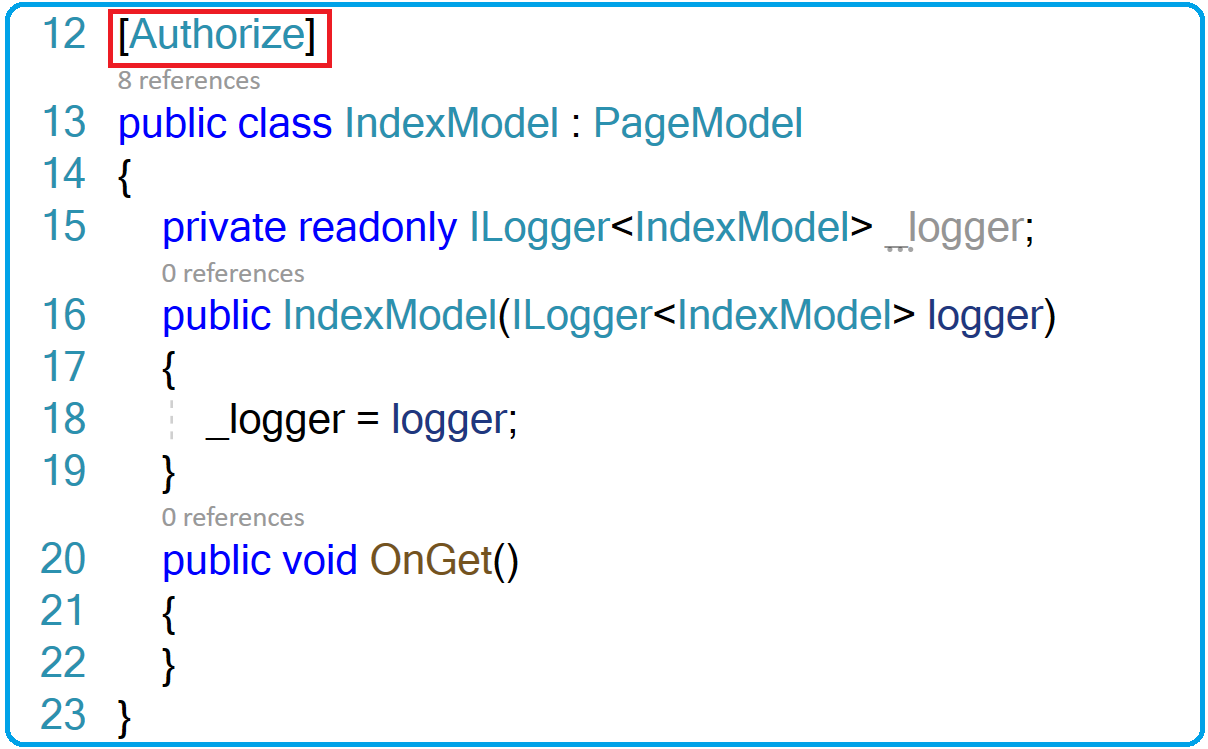
The big picture about the Authorization process, when orchestrating the 3 blocks may look as follows:

****

**Implementation of authorization**

The simplest way to restrict access to **anonymous users**, is by applying requirement to a page using the [**Authorize**] attribute. Anonymous users are those who do not have an account. By applying the [**Authorize**] attribute, only logged-in user will access the page.

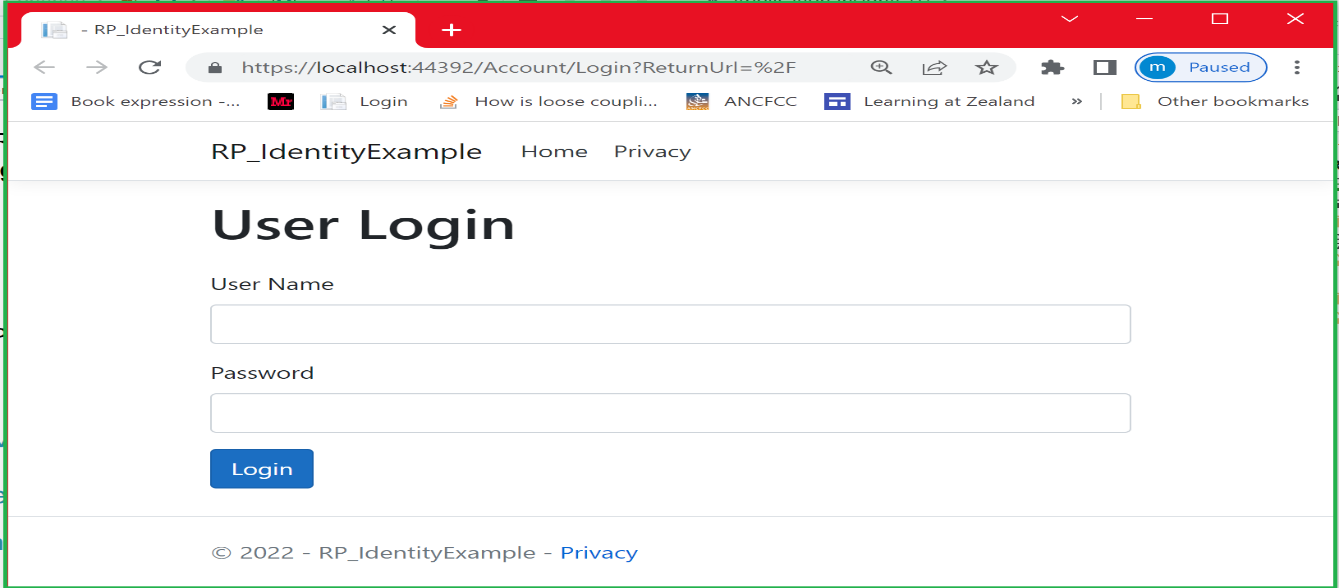
You already know that when running the application, the page that is displayed by defaul , is the index page ( this is the default one ). Let us make it inaccessible to anonymous users by adding the [**Authorize**] attribute.



Let us run the application. As you can see, you are automatically redirected to the login page because the index page is protected. Note that the ***returnUrl*** is empty.

**Be careful**: if you did not clear the cookie or you did not close your browser, you will still be redirected to the index page.

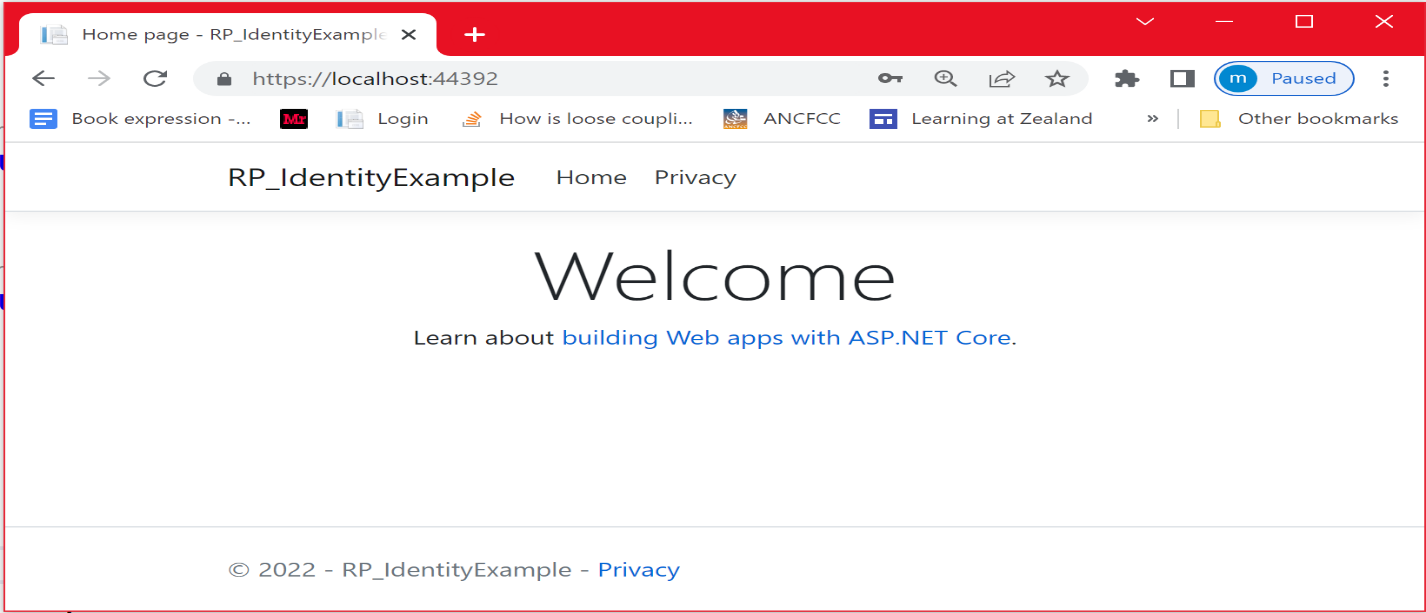
The question is how the system knows where the login page is located?



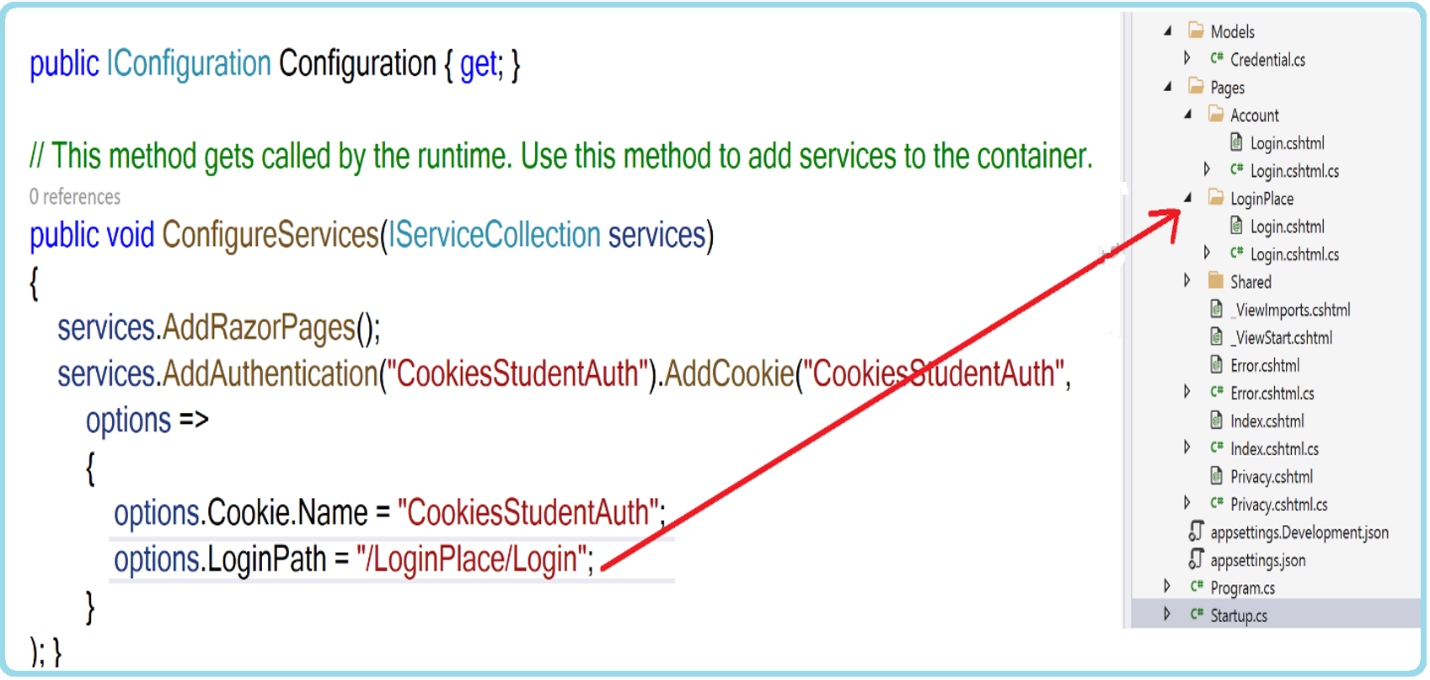
***How the system knows where the Login is located* ?**

The system figures out where to find the Login page. This is because the Login page is supposed to be by default in the “**Account**”folder. However, you can place the Login page anywhere and explicitly specify its path in the Startup.cs when configuring authentication. I will show you how to do that in a moment. For the moment, let us try to log in.

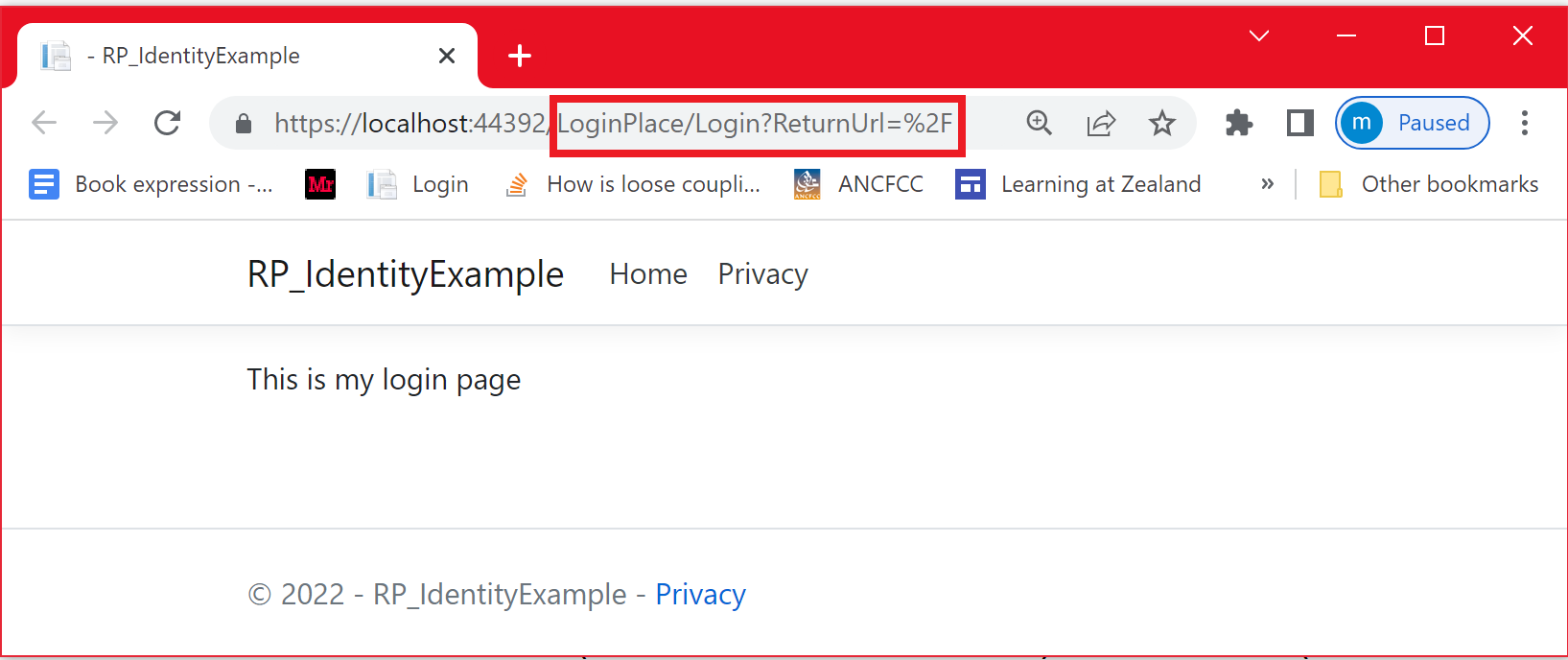
As you can see below, the logged user gets access to the index page which is protected from the anonymous users.



As mentioned before, you can place the Login page somewhere else and **explicitly** specify its path in the Startup.cs when configuring authentication. As you can see below, I added the LoginPlace folder where I placed the **Login** page. I did specify its path using the **LoginPath** property that informs the middleware where to find the Login page.

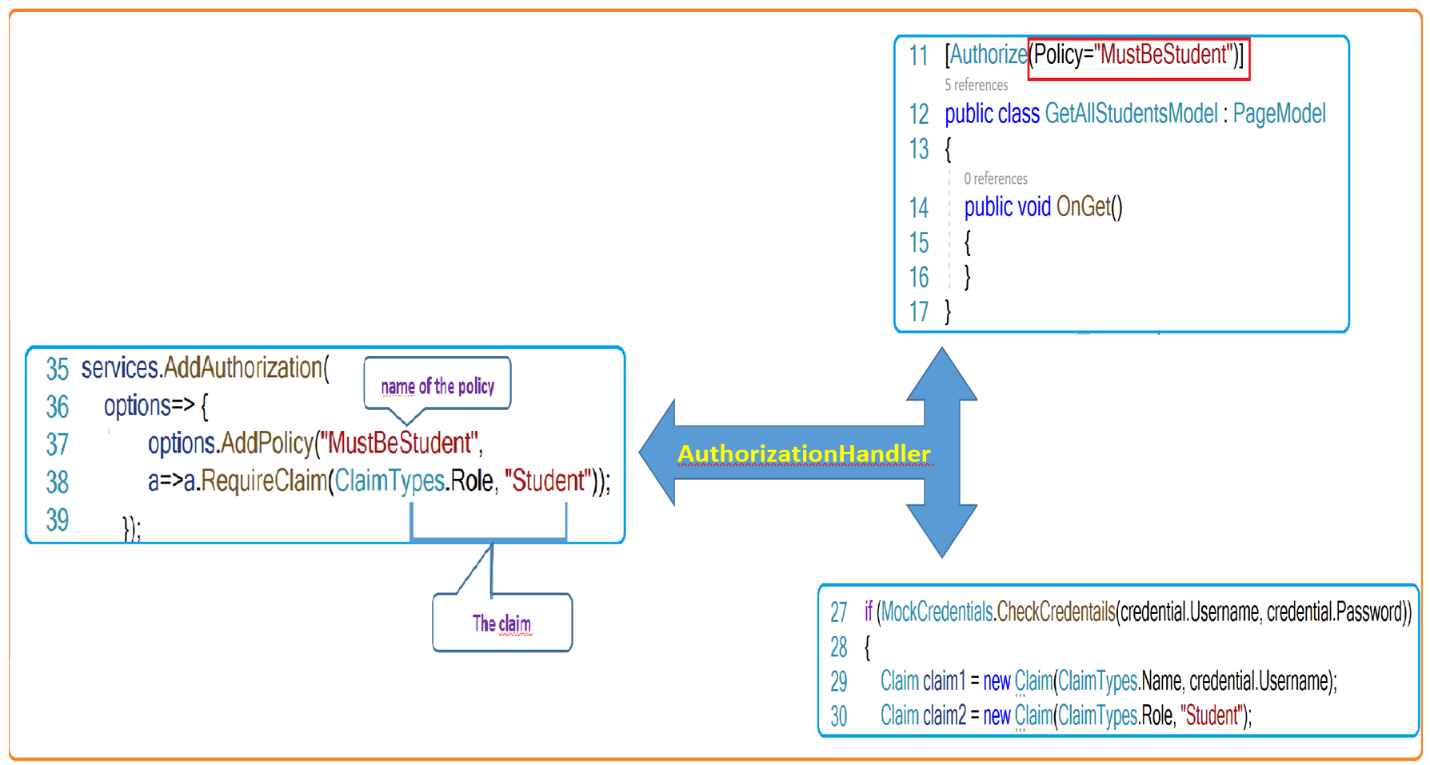


As you can see, we successfully located the Login Page.



Let us keep the default setting with the Login page placed in the Account folder and remove the “***LoginPlace***” folder .

Now let us logged-in and try to get access to the **GetAllStudents** page . As you can see below, this page requires the “***MustBeStudent***” policy.

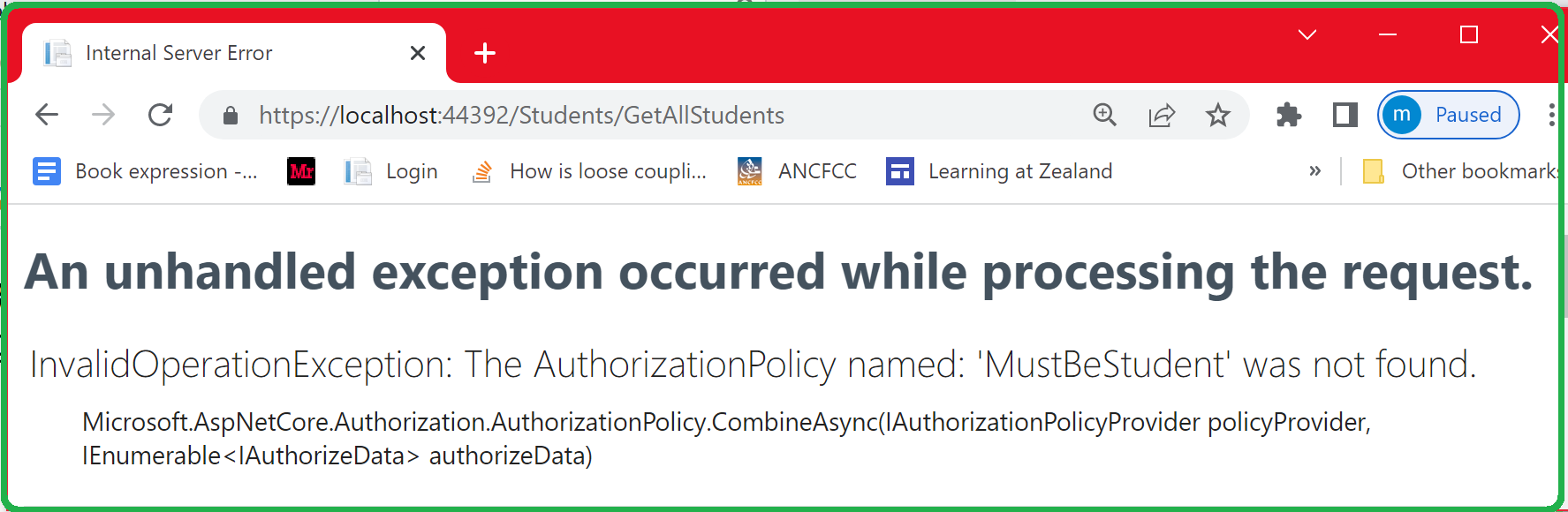
****

As you can see:

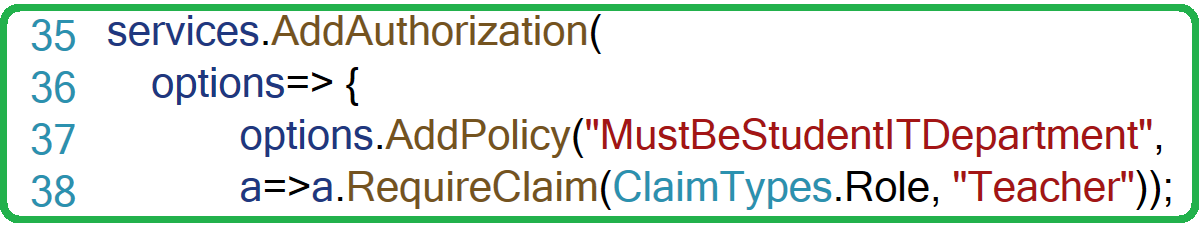
* + A predefined policy named “**MustBeStudent**” is configured with AddAuthorization service with the claim (***ClaimTypes.Role,”Student”)***
  + The page defined a policy named “**MustBeStudent**”.
  + The logged-in user will acquire the claim (***ClaimTypes.Role,”Student”)***

So we will be able to access this page.

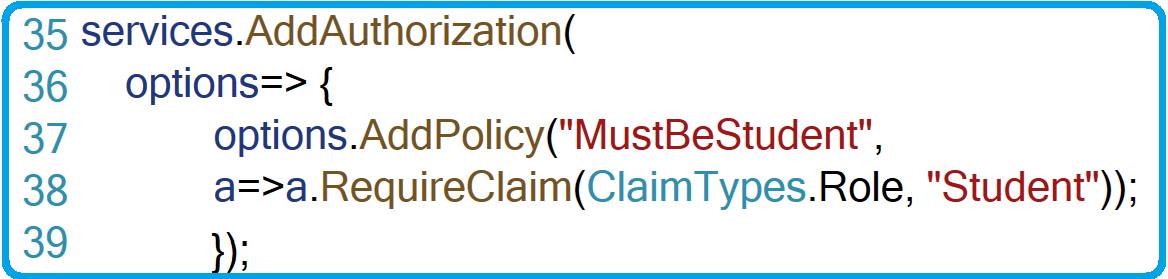
Let us **log in** and access the GetAllStudents page.



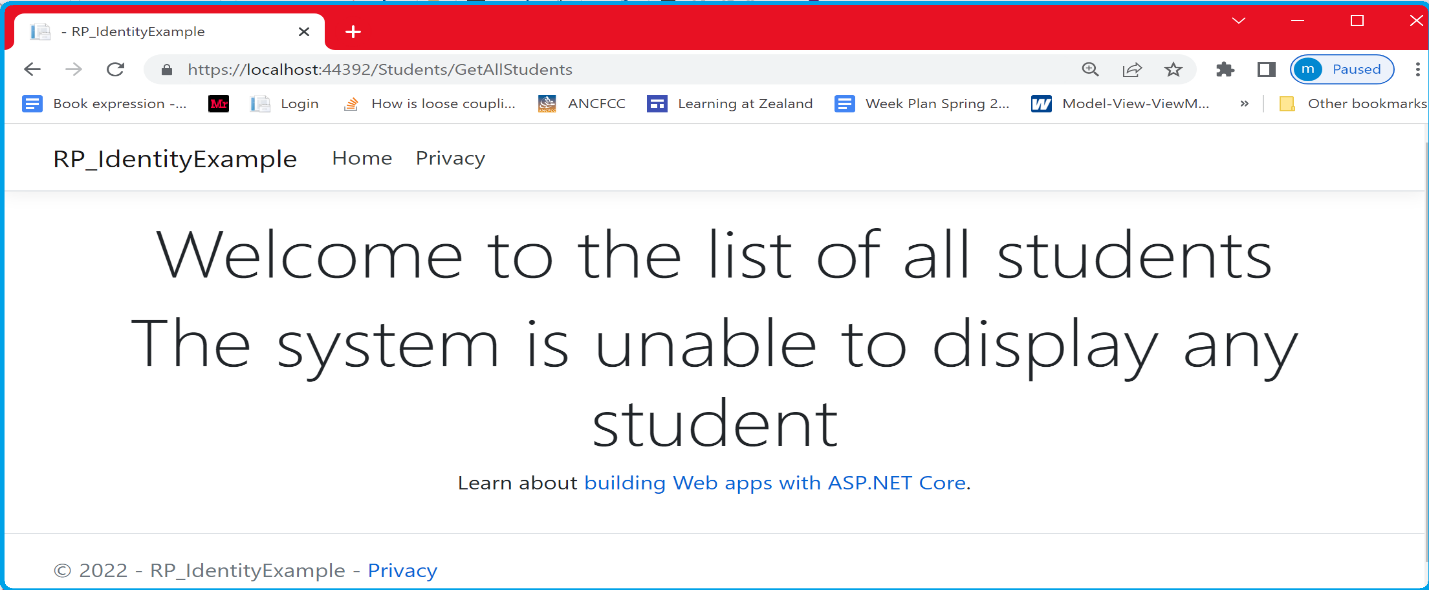
This error is due to the fact that the “**MustBeStudent**” policy is not configured. When looking at the authentication service configuration, the policy name and the claim value were wrong, as shown below.



Let us put the right name “**MustBeStudent”** and the right claim (**ClaimTypes.Role, “Student”**) into the service configuration

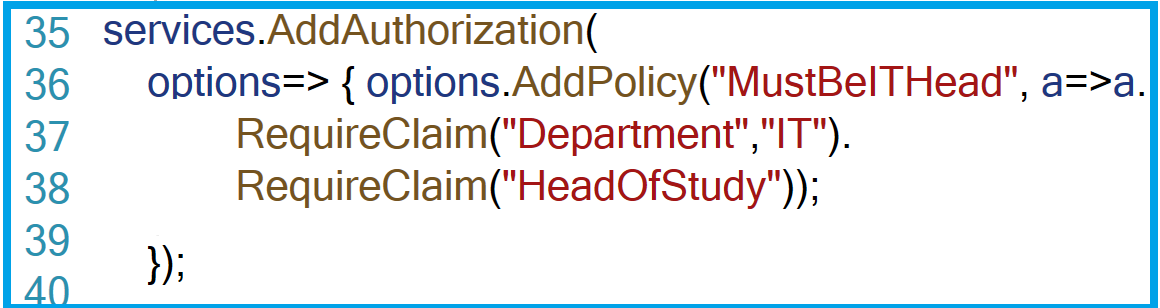


Let us run the application again. As you can see, we access the page without any problem.



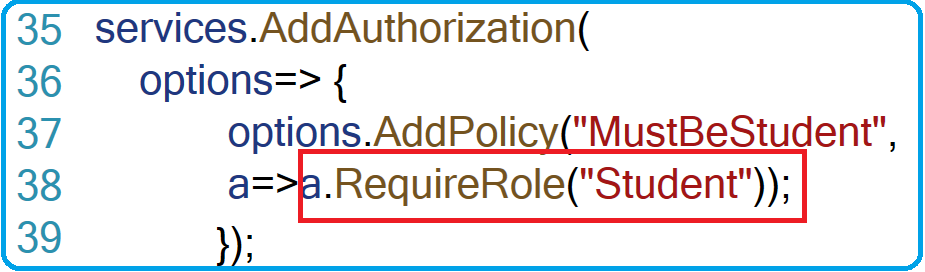
**Important**

We can also chain claims. As shown below , this policy states that to access the page, you should be the **Head of Studies** belonging to the **IT department**.

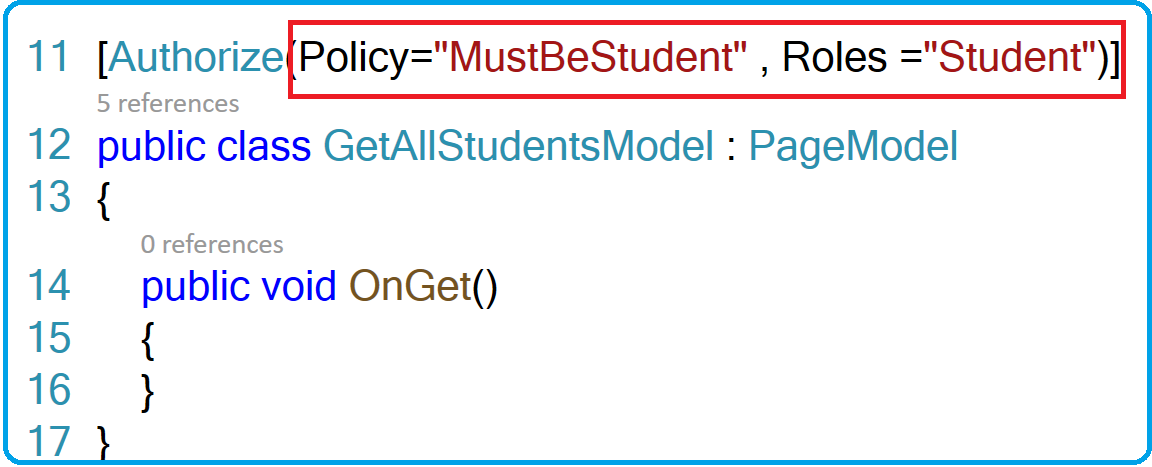


**Role v/s Claims Based Authorization**

In the previous section, we configured the “**MustBeStudent**” policy as a claim. We can also configure it as a Role, as shown below.

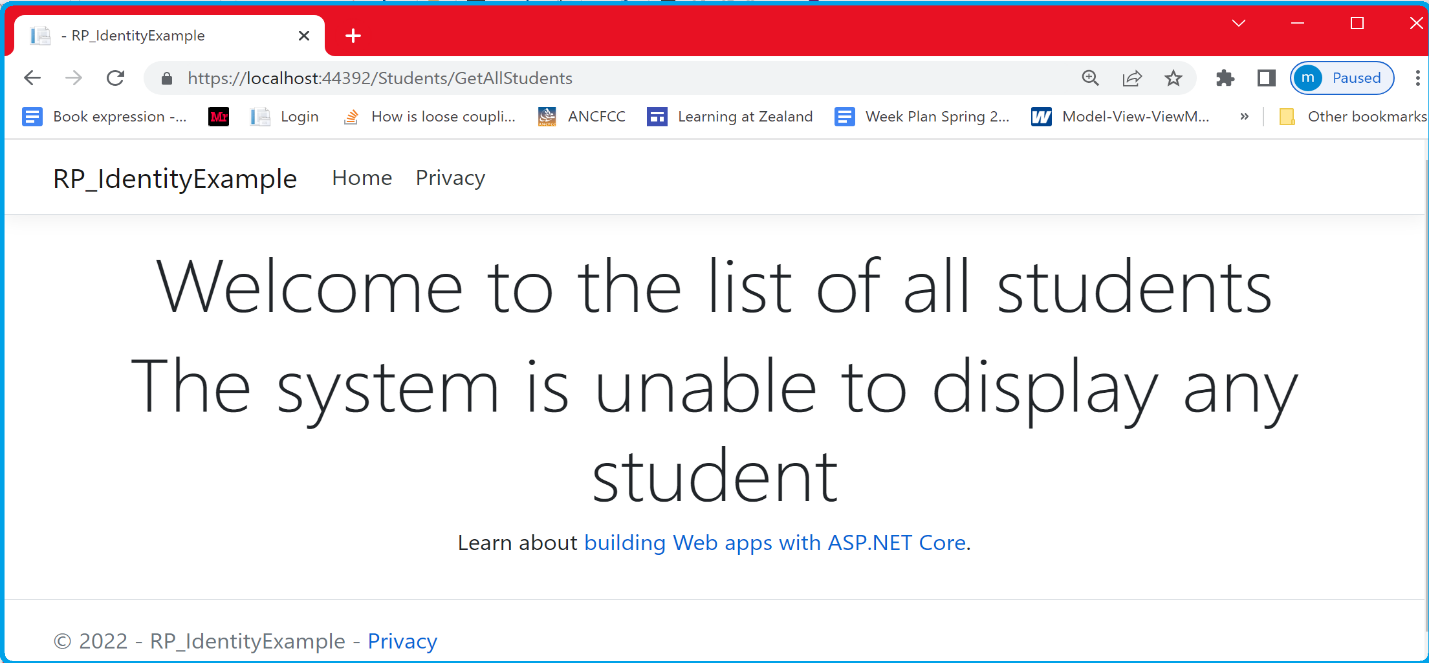


Then the attribute on the page will look as follows:

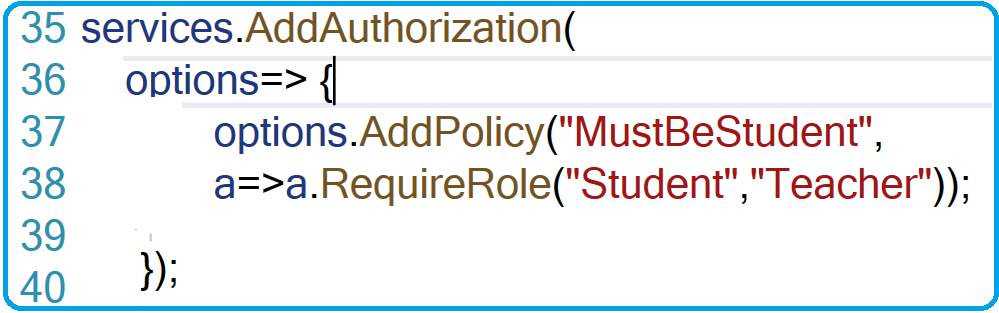


**In general** , if the requirements need complex Scenarios , use claims , otherwise use role

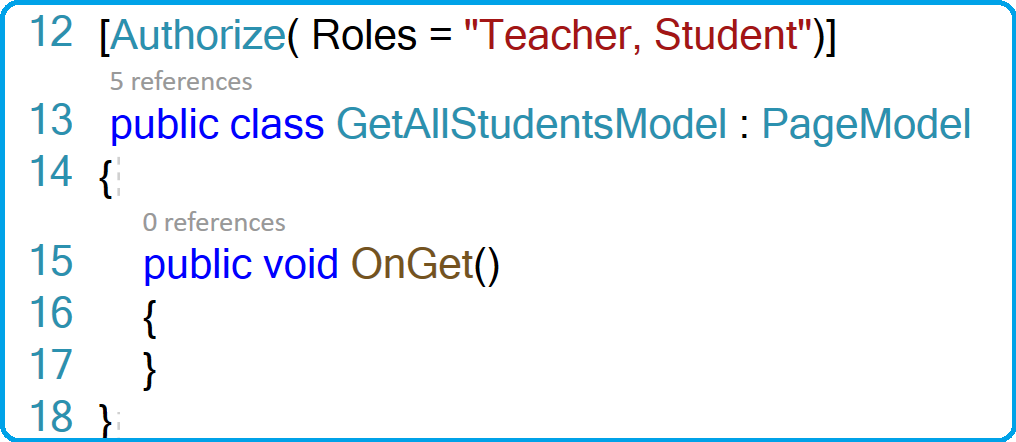
Let us run the application and get access to the GetAllStudents page.



**Note** that we can even specify a list of roles as follows:

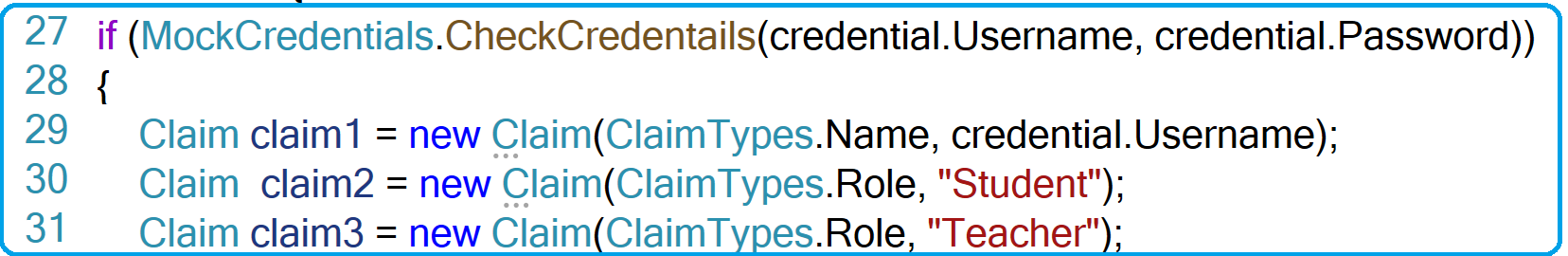


In this case, users belonging to either the Student or Teacher roles (or both) would have access to the GetAllStudents page. Below, I show how to use the [**Authorize**] attribute.



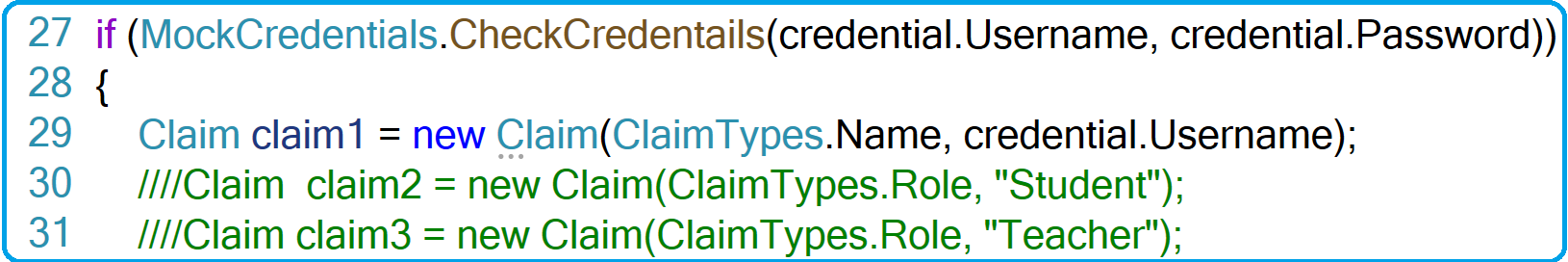
That means anyone having one or both of these roles can access the page

Below is how to specify the Role as a claim.

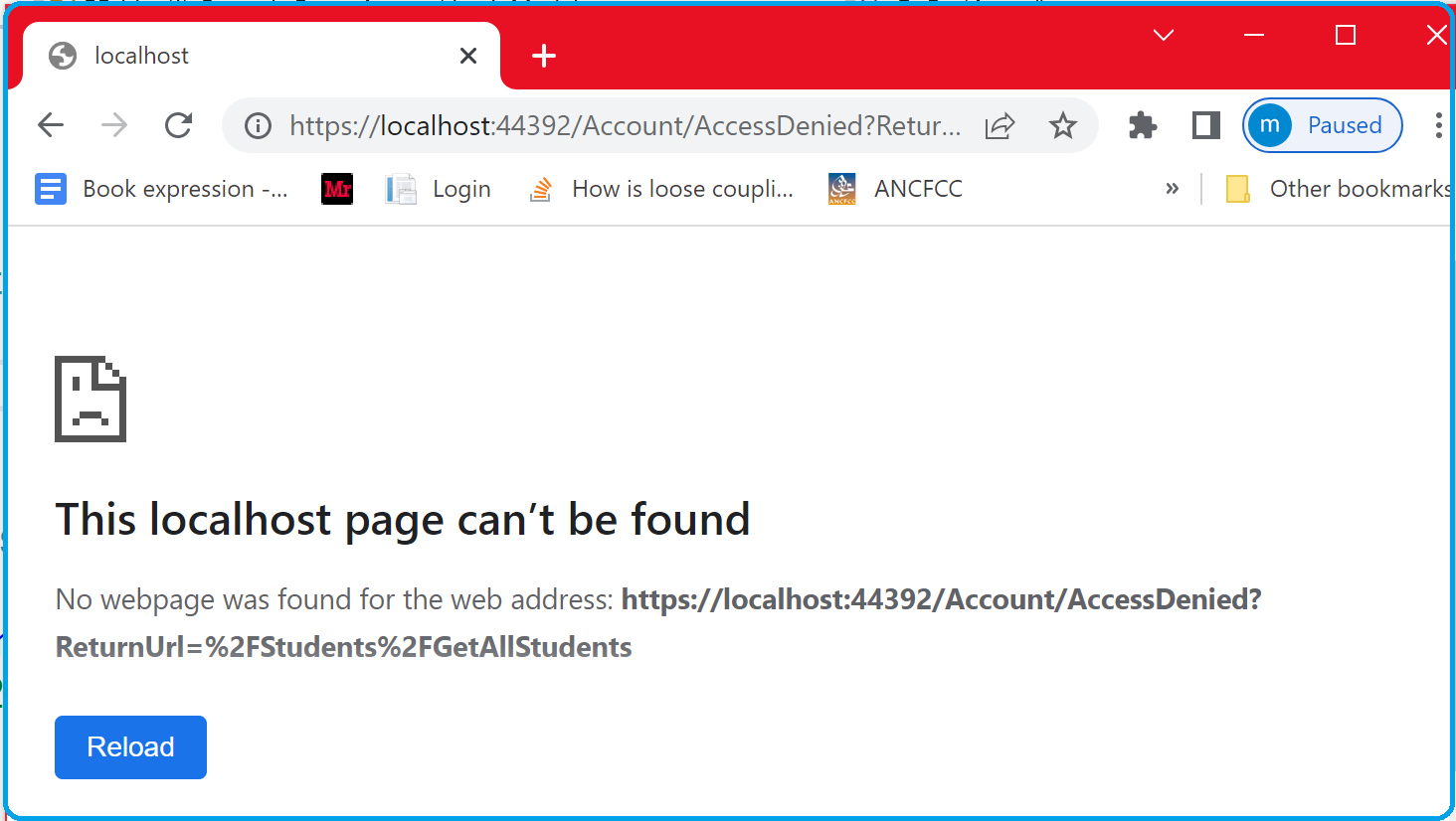


**Access Denied**

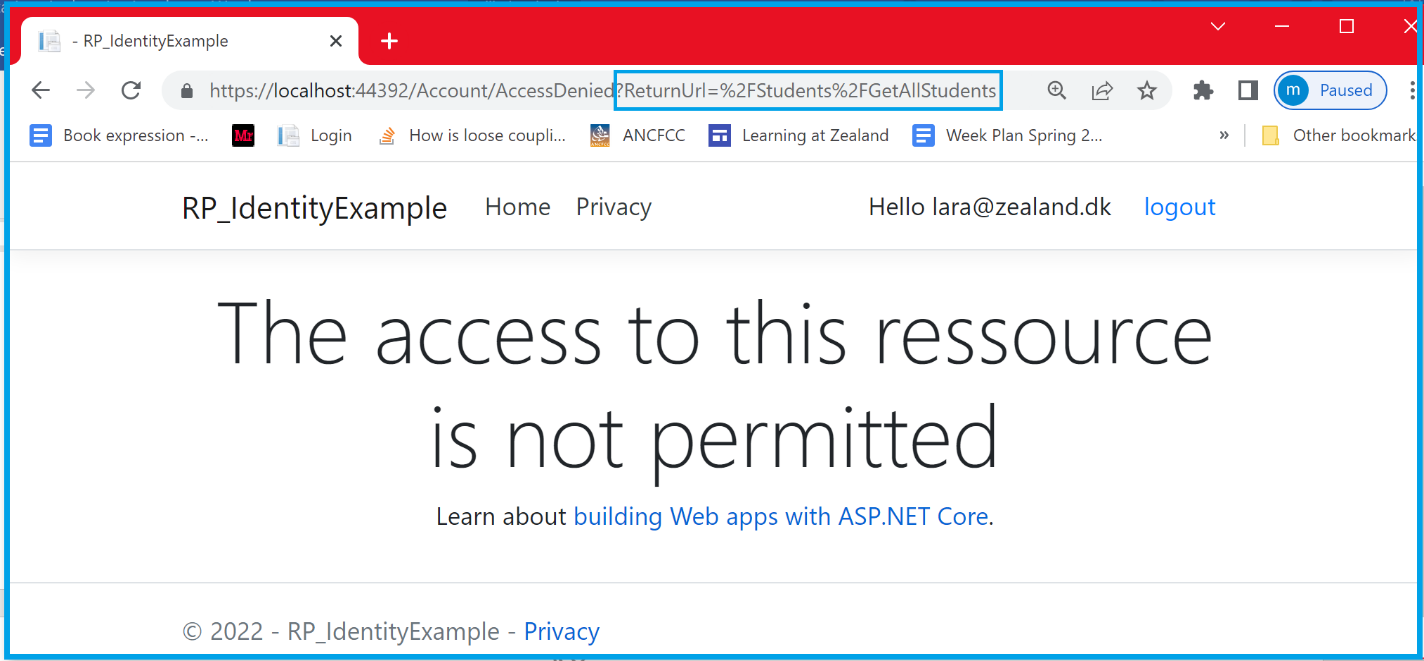
Now let us remove claim2 and claim3



Let us run the application and navigate to the ***GetAllStudents*** page.As you can see, we get access denied.



What is interesting here, is that the system tries to find an “***AccessDenied***” page in the Account folder. Let us create an AccessDenied page in the Account folder , run the application again and try to navigate to the GetAllStudents page. As you can see, the new created “**AccessDenied**”page is displayed. Note the ReturnUrl on the URL , which is the page to which the access is denied



**Note** : As done with the Login page earlier , we can also specify where the AccessDenied page is located:

**Implementing Logout**

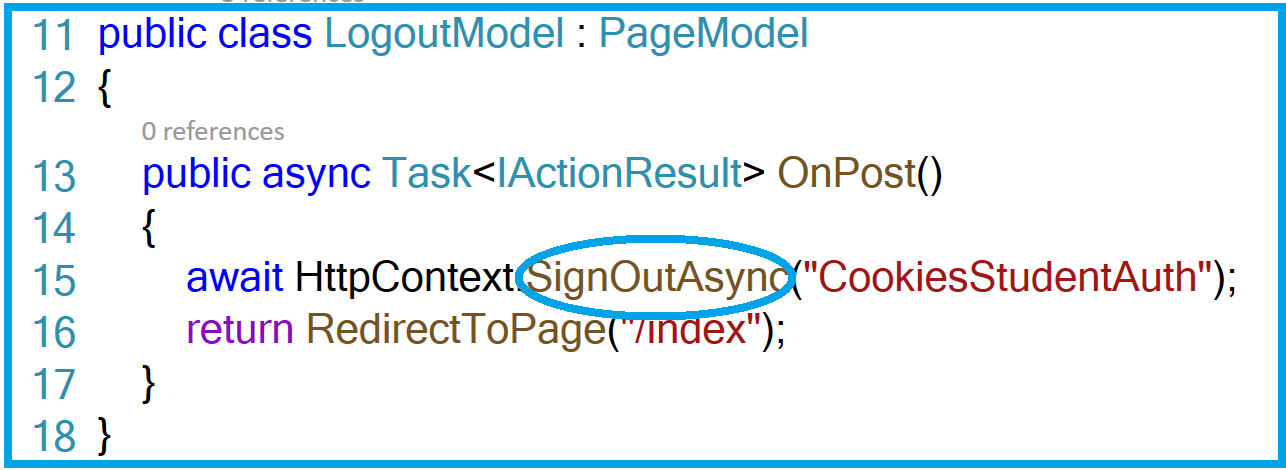
As long as you keep the browser open, you still have the cookie. Let us implement the **LogOut** functionality that make sure that the cookie is removed when logging out.

I will just remind you the httpContext extension method ***SingInAsync()*** method that allows encrypt and serialize the security context into a cookie ( that is sent to the browser). The same way, we have the httpContext extension method **SingOutAsync()** method that allows kill and remove the cookie.

Before implementing the Logout, let us create the **Logout** page. No need for the page part ( HTML part). We are going to use **a button** encapsulated into **a form** that uses the “**post**” method

**Note:** we can also use a link and use the OnGet() method for the logout code.

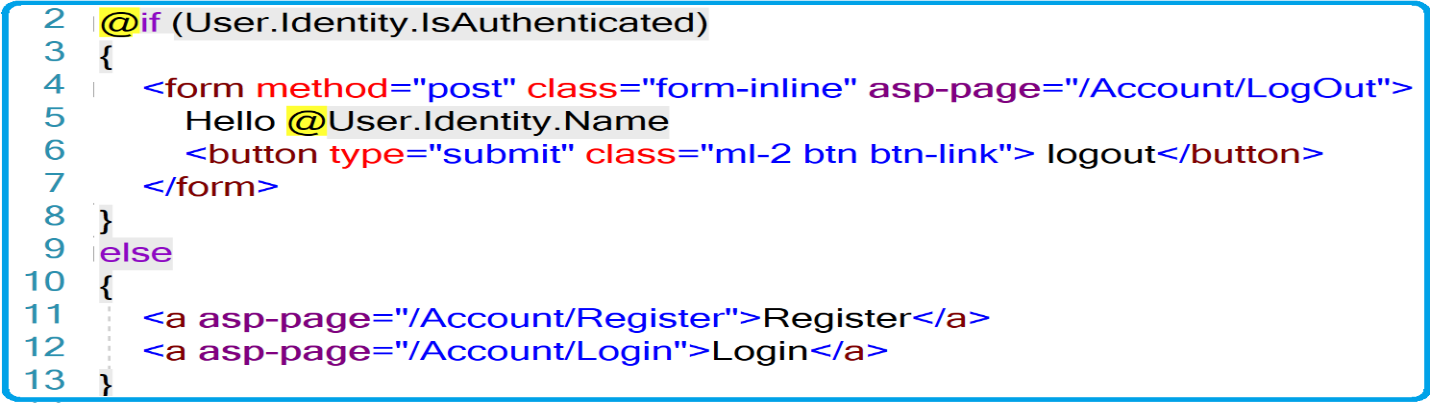
When clicking on the button, the OnPost() is called. The implementation of the OnPost() method is shown below. We call the **SignOutAsync()** specifying the authentication scheme that we want to remove.



One last thing and we are done . At the right hand side of the top menu , let us create a menu describing the login status . We would like to show “**Regsiter**” and “**Login**” buttons when the user is not authenticated , while showing a “**Hello Mr X**” message and the **Logout** button when the user is authenticated.

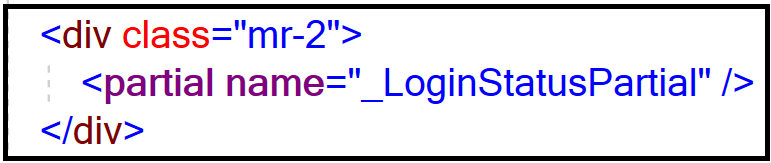
In the shared folder, let us create a Partial page called **“\_LoginStatusPartial.cshtml**”.

Add the following code :

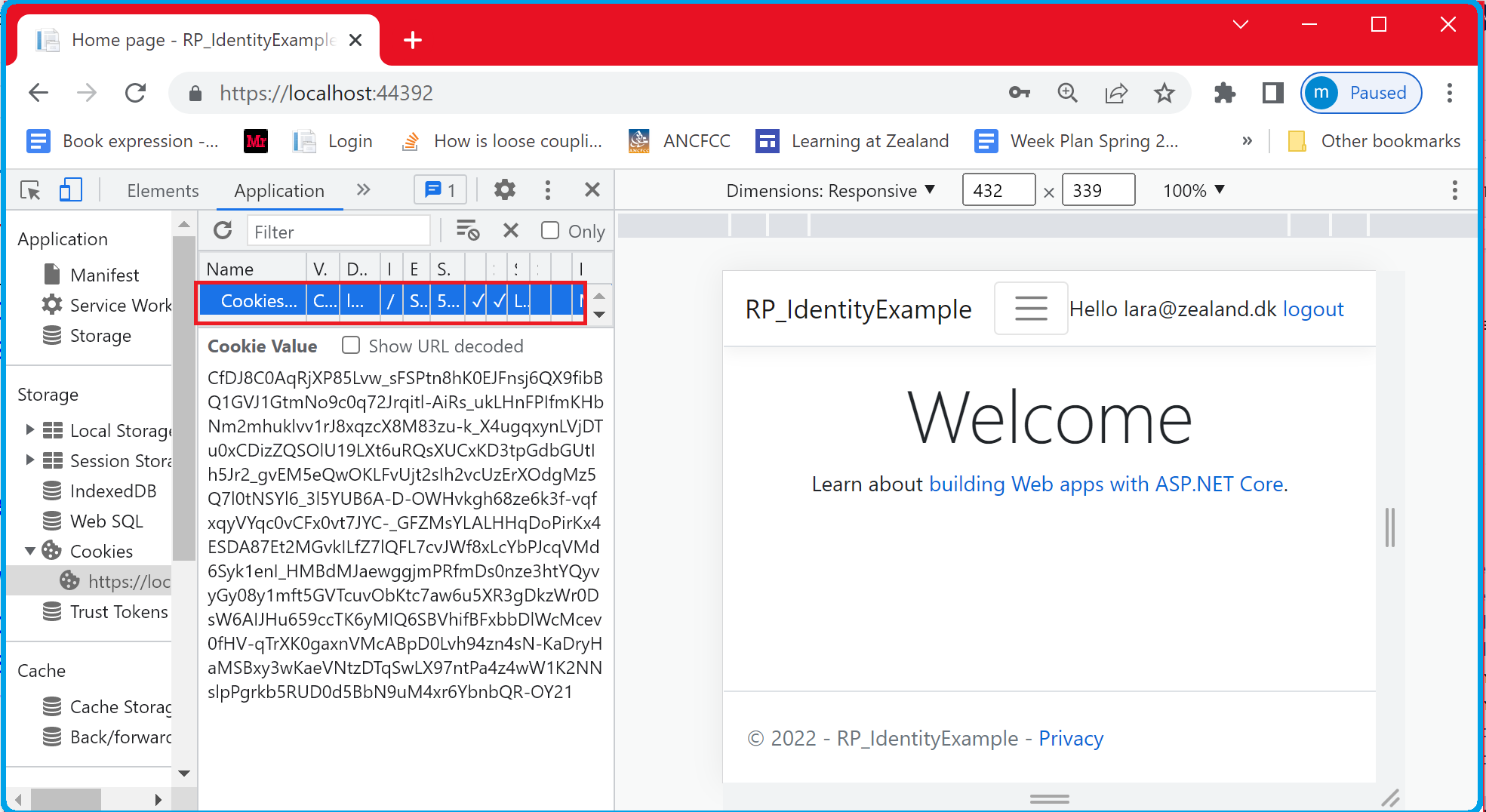


**@if (User.Identity.IsAuthenticated)** states whether the user gets authenticated at the server side. In this case, the Hello message along with the identity name are shown , Otherwise the two **Register** and **Login** links are shown.

Bfore running the application , integrate this Partial page into the \_Layout page by placing the following code on the right place



|  |  |
| --- | --- |
| |  | | --- | | Let us run the application.  **After authentication**    Let us have a look at the cookie. The encrypted cookie is shown on the browser | |



|  |
| --- |
| **After Logout**    Let us have a look at the cookie. The cookie is removed |

