Udemy course: Ultimate ASP.NET pt. 7

Section 8: Securing Your API

#40: Overview

- Types of API Security
    - N
        - Network Level
          > IP Whitelisting
          > IP Blacklisting
        - for example for institutions or companies which have specific internet adresses and none else should be able to access it
        - not suitable for public access APIs
    - B
        - Basic Authentcation
          with each call to the API
        - with Username and Password
        - has lots of system overhead - on the system level
        - every time someone uses your API: extra database call to make sure that the person has rights to access the application
        - another database call to give them the resource
    - A
        - API Key Access
        - Issue an access key to client app
        - similar to basic authentication
        - every time a call is executed, and API key is provided with it
        - you still have to validate, before access of the resource is possible
    - J
        - current industry standard
        - use of JSON Web Tokens (JWT)
        - issue an encoded token with a limited lifetime and reissue a new one when needed
        - different token vs access key:
          > JWT is not mean to be super secure
          > API key has username and password, you would encrypt it
          > sensitive information should not be included in the JWT!
          > instead you include - what we call - claims
          > claim: basic bits of information, that every user needs to have
          so for us to know; that the user is who they say, they are
          > limited lifetime - perhaps e.g. 30 minutes; after 30 minutes the token is useless
        - we will be implementing this way in our application

#41: Setup User Identity Core

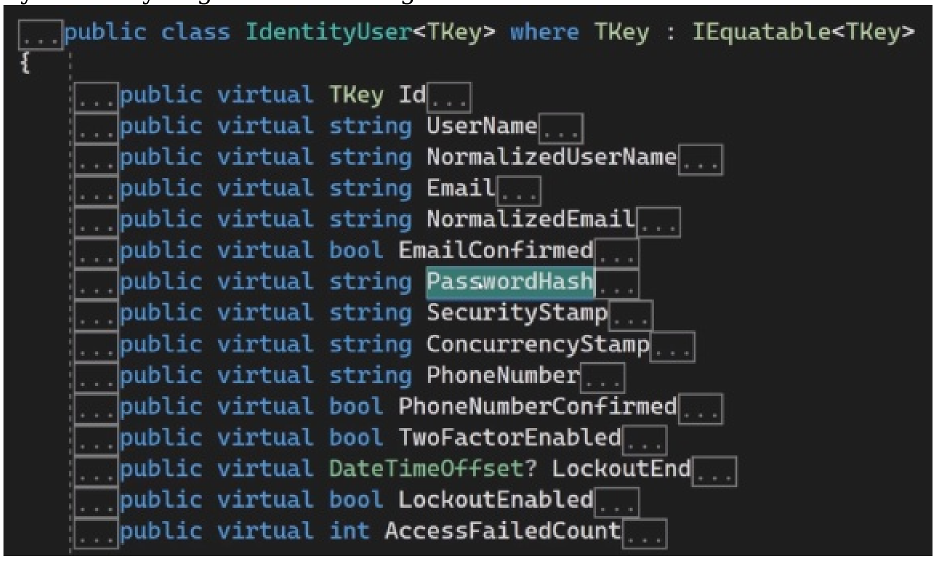
- Identity Core
    - flagship security library in any .NET project
    - Microsoft developed it, highly extensible
    - in this course we just look at more basic stuff
    - just enough to secure our API
- to use it
    - we add following line in our Program.cs after we add our db context:
      ```
      builder.Services.AddIdentityCore<IdentityUser>();
      ```
    - IdentityCore comes with everything out of the box, that we use as a user type, like
        - email adress, phone number, user name, password, encryption
    - all comes built-in
- usually makes sense to use the built-in encryption of that library; only use your own when you really know what you are doing!
- next to our above line, we also add roles next

    -
      ```
      .AddRoles<IdentityRole>()
      ```
    - roles are about what a user can do
    - next we also let it know, which data store it should use

        -
          ```
          ```

```
.AddEntityFrameworkStores<HotelListingDbContext>()
```

- this is about which database context we want to use;
  in our case we use HotelListingDbContext
  - for that we need to install Identity.EnityFrameworkCore library
  - so we install it with:
    $ dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore
- next we want to extend the default offerings of idenity user
  - idenity user: default user type, that comes with lots of properties like email etc.
  - by default you get the following:



```
public class IdentityUser<TKey> where TKey : IEquatable<TKey>
{
    public virtual TKey Id
    public virtual string UserName
    public virtual string NormalizedUserName
    public virtual string Email
    public virtual string NormalizedEmail
    public virtual bool EmailConfirmed
    public virtual string PasswordHash
    public virtual string SecurityStamp
    public virtual string ConcurrencyStamp
    public virtual string PhoneNumber
    public virtual bool PhoneNumberConfirmed
    public virtual bool TwoFactorEnabled
    public virtual DateTimeOffset? LockoutEnd
    public virtual bool LockoutEnabled
    public virtual int AccessFailedCount
```

  - there is nothing about first name and last name!
  - for that we add a new class in Data folder, we name it e.g. ApiUser
  - so we declare the class like this:

```
public class ApiUser : IdentityUser

{

public string FirstName { get; set; }

public string LastName { get; set; }

}
```

  - how to use that: so wherever we would have used IdentityUser we just use ApiUser - also in Program.cs we will use that!
- so we have the following lines in Program.cs now:
  -

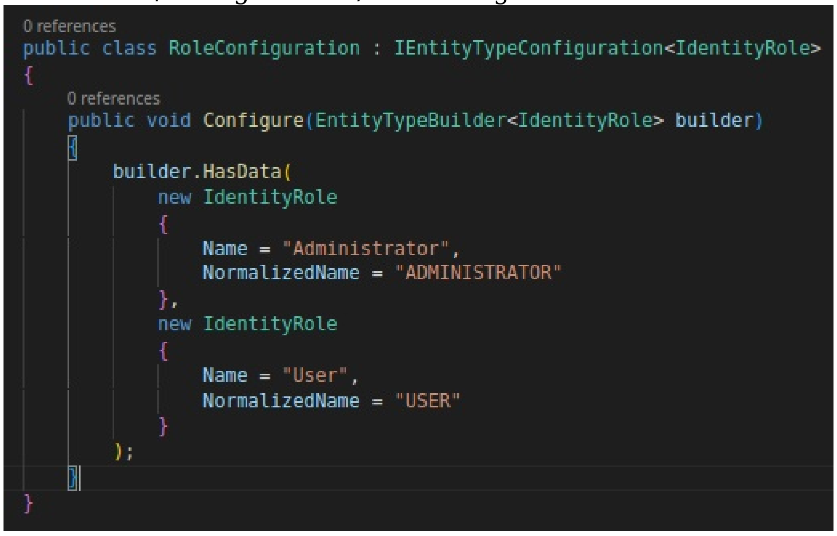
```
builder.Services.AddIdentityCore<ApiUser>()

.AddRoles<IdentityRole>()

.AddEntityFrameworkStores<HotelListingDbContext>()
```

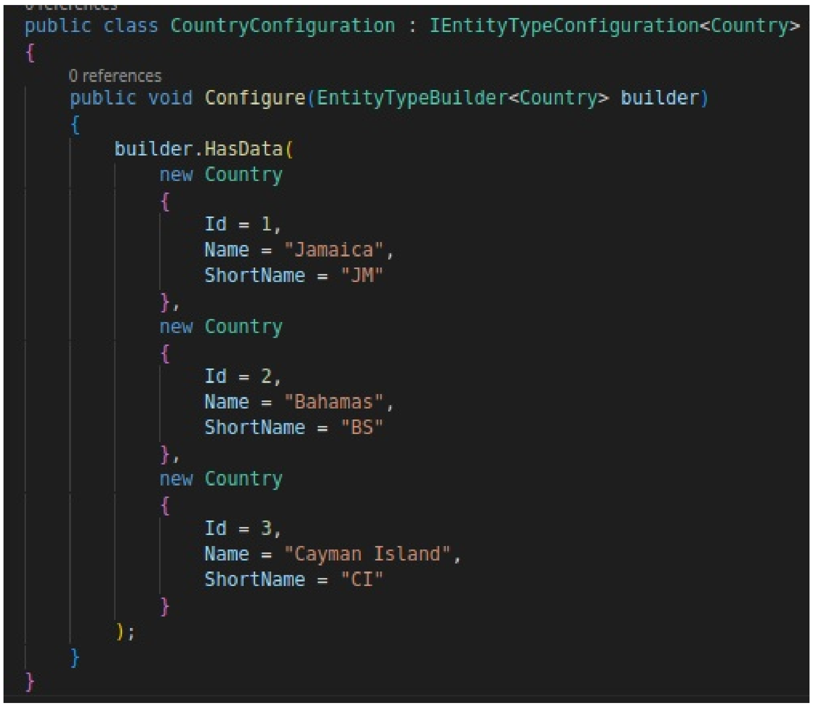
- Note: AddItentyCore needs either IdentityUser or a type that is extending it like our ApiUser
- next we have to let the Db context know, that it is responsible for storing our idenity stuff
- in the db context file (HotelListingDbContext), we also need to change the inheritance
  - we should inherit from IdentityDbContext<ApiUser>, not just from DbContext
  - so it is no longer just inherting from our regular DbContext, but from IdentityDbContext relative to our extended UserType
- now with these changes, we need to make another migration:
  - in Visual Studio it would be in the package manager:
    add-migration AddedIdentity
  - in Visual Studio Code:
    $ dotnet ef migrations add AddedIdentity
- the result of this command is a new huge migration file in Migrations
- Note:
  - the id columns in the migration are string, guid by default

        - it is mostly alright and no dealbreaker
        - AspNetRules, AspNetUsers entity are added
        - in AspNetUsers are our First and LastName as well as the builtin fields
        - also created AspNetUserLogins - for example used when you have 3rd party authentification
        - AspNetUserRoles entitys - to assign one user to one or several roles
          > a many-to-many relationsship
          > all it includes: UserId and RoleId
        - AspNetUserTokens
        - the table names could be overwritten - but that is not covered in this course
          - it is in either in Security or Enity Framework course
    - to actually run the migration
        - in Visual Studio: Update-Database
        - in Visual Studio Core: dotnet ef database update
- next we can add default user roles

#42: Add Default User Roles

- we look into roles that should be present, when the API gets deployed
    - on first build, these roles should already be present
    - it musn't be roles that the users is in control of
    -
- add folder Data/Configurations
    - add class RoleConfiguration there
    - it inherits from
      `IEntityTypeConfiguration<IdentityRole>`
- IdentityRole is the data type, that we wish to create our configuration for
    - in this case: IdentityRole
- it has a method

    -
      ```
      public void Configure(EntityTypeBuilder<IdentityRole> builder)
      ```
    - for the purpose to build our configurations
- NormalizedName
    - usually just as Name, but all CAPS
- our code in Data/Configurations/RoleConfiguration.cs:
    - 
      ```
      0 references
      public class RoleConfiguration : IEntityTypeConfiguration<IdentityRole>
      {
          0 references
          public void Configure(EntityTypeBuilder<IdentityRole> builder)
          {
              builder.HasData(
                  new IdentityRole
                  {
                      Name = "Administrator",
                      NormalizedName = "ADMINISTRATOR"
                  },
                  new IdentityRole
                  {
                      Name = "User",
                      NormalizedName = "USER"
                  }
              );
          }
      }
      ```
    - in this class, we could also rename the table, if we wanted to
    - we can setup all kinds of configuration, that is associated with the data type
- we update our HotelListingDbContext
    - currently it is quite big and messy and we need the roles information
    - expanded like it is, it has too much
    - we create another Data/Cofigurations class with name CountriesConfiguration
    - to start with that, we can just copy our RoleConfiguration
    - the class is now related to the Country - not to IdentityRole like RoleConfiguration
- so this class we then have like this:

- 

```
public class CountryConfiguration : IEntityTypeConfiguration<Country>
{
    0 references
    public void Configure(EntityTypeBuilder<Country> builder)
    {
        builder.HasData(
            new Country
            {
                Id = 1,
                Name = "Jamaica",
                ShortName = "JM"
            },
            new Country
            {
                Id = 2,
                Name = "Bahamas",
                ShortName = "BS"
            },
            new Country
            {
                Id = 3,
                Name = "Cayman Island",
                ShortName = "CI"
            }
        );
    }
}
```

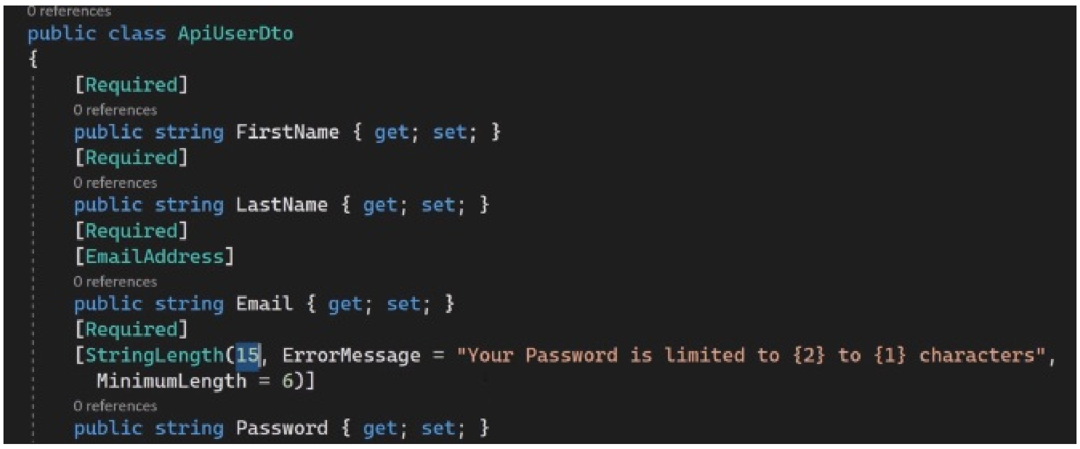
  - Note: we cut .HasData(...); from our HotelListingDbContext class
    - we also need to use the following:

    -
      ```
      using Microsoft.EntityFrameworkCore.Metadata.Builders;
      ```
    - the autocorrection functionality in Visual Studio Code, might not see that!
  - it is not needed in HotelListingDbContext, since it is applied by the Configure method in CountriesConfiguration()
  - so we just put the line in the HotelListingDbContext class:

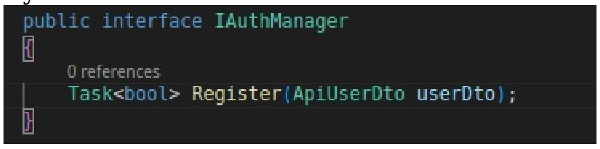
    -
      ```
      modelBuilder.ApplyConfiguration(new CountryConfiguration());
      ```
    - to make it leaner and simpler
  - next we do the same with the Hotel data
    - we create HotelConfiguration class
    - and extract the lines from the HotelListingDbContext to the HotelConfiguration just like before
- now HotelListingDbContent class looks much more neater
  - also for any configuration, we have a particular file we can make the changes to
  - so this is about separation of concerns
- we run another migration called AddedDefaultRoles
  - also run the update
- the dataentity dbo.AspNetUserRole should now have both roles
  - of Administrator and User
- so we are done with the roles and also did some refactoring

#43: Setup Auth Manager For Registration

- we want API functionality to register Users
- we create a new Dto for Users functionality
  - Models/Users/ApiUserDto
  - initial draft like this:

        - 

```
0 references
public class ApiUserDto
{
    [Required]
    0 references
    public string FirstName { get; set; }
    [Required]
    0 references
    public string LastName { get; set; }
    [Required]
    [EmailAddress]
    0 references
    public string Email { get; set; }
    [Required]
    [StringLength(15, ErrorMessage = "Your Password is limited to {2} to {1} characters",
      MinimumLength = 6)]
    0 references
    public string Password { get; set; }
```

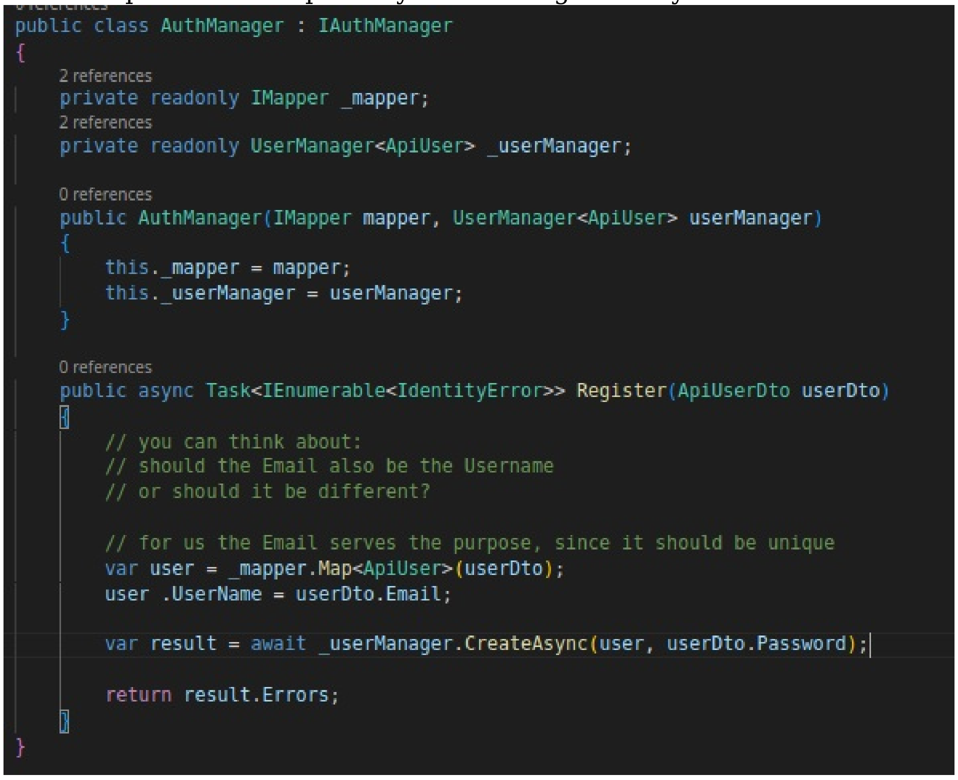
        - later on, we will refactor a bit with a LoginDto base class
    - cool about IdentityCore
        - it has its own validation or passwo
    - it has its own validation of passwords
    - a minimum length of password
        - and must be different character types etc.
    - Note: we can put validation already on this level; like you see with the StringLength example
- we implement a new contract, name IAuthManager
    - initially it can look like this:
        - 

```
public interface IAuthManager
{
    0 references
    Task<bool> Register(ApiUserDto userDto);
}
```

- in our MapperConfig we have to add a mapper configuration
    - create a map between the two data types:
    -

```
CreateMap<ApiUserDto, ApiUser>().ReverseMap();
```

- next we can continue with our Repository - the imlementation of the new contract
    - we revise our return type of Register, we are not satisfied with bool return type
        - e.g. we want to know why it failed, when an error happens
        - so we want to add an validation message
    - instead of an bool we now change it to an IEnumerable<IdentityError>>
- so our implemented Repository AuthManager finally looks like this:



```
public class AuthManager : IAuthManager
{
    2 references
    private readonly IMapper _mapper;
    2 references
    private readonly UserManager<ApiUser> _userManager;

    0 references
    public AuthManager(IMapper mapper, UserManager<ApiUser> userManager)
    {
        this._mapper = mapper;
        this._userManager = userManager;
    }

    0 references
    public async Task<IEnumerable<IdentityError>> Register(ApiUserDto userDto)
    {
        // you can think about:
        // should the Email also be the Username
        // or should it be different?

        // for us the Email serves the purpose, since it should be unique
        var user = _mapper.Map<ApiUser>(userDto);
        user .UserName = userDto.Email;

        var result = await _userManager.CreateAsync(user, userDto.Password);

        return result.Errors;
    }
}
```

    - that is it for setting up our AuthManager
    - later we will change/extend it more

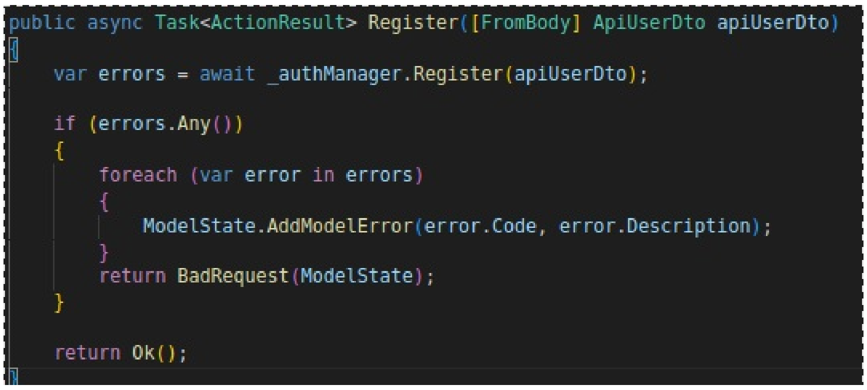
- like we will setup the controller to meet the call and return the positive or negative feedback to our client
- One more thing before we move on:
  - we can add the role here actually
  - so another line to add in above Register method:

    ```
    if (result.Succeeded){

    await _userManager.AddToRoleAsync(user, "User");

    }
    ```
- before we return the Errorcode

#44: . Setup Registration Endpoint

- we are creating an empty API controller
  - not much scaffolding functionality for any logging/registration functionality
- our Base class is again ControllerBase
- when you want to modify data, you would use
  - POST, PUT, or PATCH
  - we did not look at PATCH yet
  - basically it is used for targeted update operations
  - we will use POST, since we want to hide the data
- there are/should be multiple requests in our controller
  - for register
  - for login
  - to refresh token
- for the Register action we would use these attributes and method declaration:

  -
    ```
    // POST:
    api/Account/register

    [HttpPost]

    [Route("register")]
    ```

  -
    ```
    ProducesResponseType(StatusCodes.Status400BadRequest)]

    [ProducesResponseType(StatusCodes.Status500InternalServerError)]

    [ProducesResponseType(StatusCodes.Status200OK)]

    public async Task<ActionResult> Register([FromBody] ApiUserDto apiUserDto){[...]}
    ```
  - if validation fails we would want to get a BadRequest - we have seen that before
    - so that is that line for
    - `ProducesResponseType(StatusCodes.Status400BadRequest)]`
    - if the request returns status code 500, we can return the Internal Status Error
    - also we can produce a Status 200 - which is OK
- we implement the action like that: 

```
public async Task<ActionResult> Register([FromBody] ApiUserDto apiUserDto)
{
    var errors = await _authManager.Register(apiUserDto);

    if (errors.Any())
    {
        foreach (var error in errors)
        {
            ModelState.AddModelError(error.Code, error.Description);
        }
        return BadRequest(ModelState);
    }

    return Ok();
}
```

  - if (errors.Any()):
    - we want to iterate through the errors
    - and add them to the model state
    - the model state handles the errors
    - it checks whether the data type is accepted for the request
    - and the model state gets set
    - the BAD Request was actually returned by the model state object
    - AddModelError requires a key and a error message
    - see the foreach loop above
  - otherwise we can just return Ok()
    - Note: don't return Ok(apiUserDto) here; it would send your password!!!
- in Program.cs we have to add our AddScoped Line

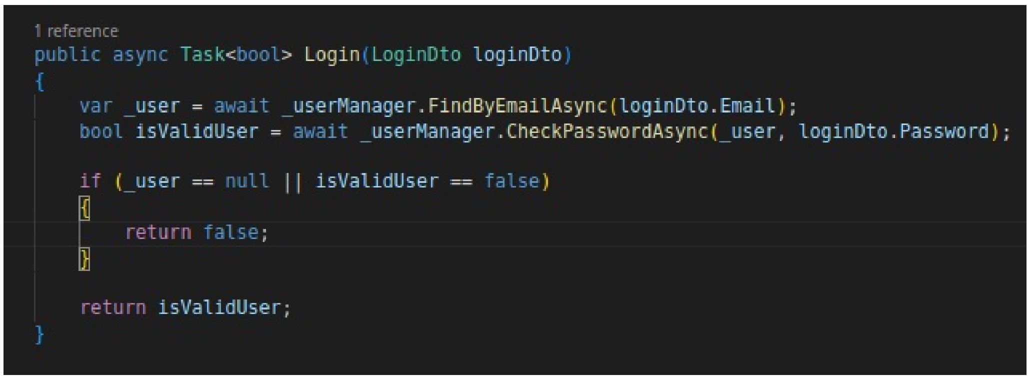
  -
    ```
    builder.Services.AddScoped<IAuthManager, AuthManager>();
    ```
- we can try our Register API via Swagger (or Postman)
- we should get a success response with 200 status, with following example
  - {
  - "firstName": "Max",
  - "lastName": "Maier",
  - "email": "max@example.com",
  - "password": "P@assword123"
  - }
- Note: with too simple passwords it would return Status 400
  - Bad Error Request
    - it would also show in Response body:

      ```
        "PasswordRequiresNonAlphanumeric": [
          "Passwords must have at least one non alphanumeric character."
      ```
  - also when email already exists
    - it would appear Username ... is already taken.
-

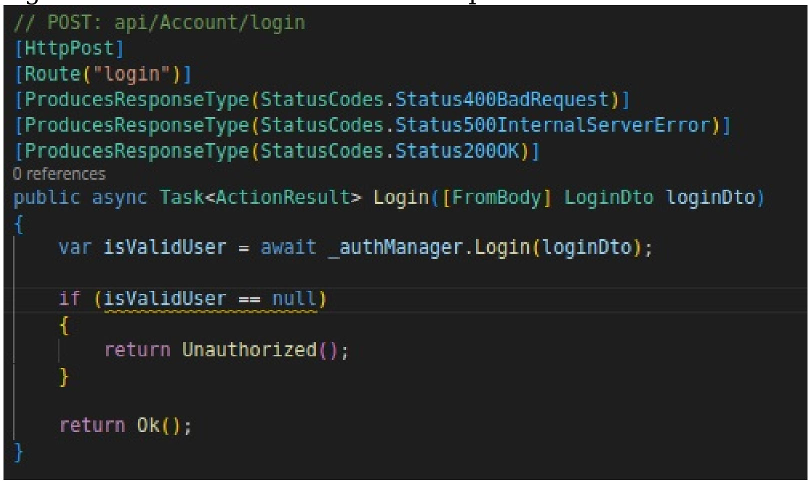
#45: Setup Login Endpoint

- login operations tend to just verify that the user exists in the system;
  - this differs what an API and a regular web application does
  - for a regular app: you create a cookie, and it keeps the cookie throughout the session
  - an API is stateless however; so no Cookies;
    - APIs don't know when they are beeing access or by whom at any point in time
  - later more on
    - how we secure and faciliate that kind of authentication
- the login operation
  - usually not negotiable, but it can be managed
  - e.g. with JSON Web tokens
- in our contract we add the login method, as first step with this return type

  -
    ```
    Task<bool> Login(LoginDto loginDto);
    ```
- how do we validate the login?
  - we validate, we not really login a person in!
  - there is a method, FindByEmailAsync in UserManager
    - or FindByLoginAsync
    - or FindByNameAsync
  - select the appropiate method, based on the data, you are feeding it
- in our Login Action in our Repository we can take the UserManager methods
  - FindByEmail and CheckPasswordAsync as next step
  - of course asynchronous methods again
- we implement our Login in Authmanager repository like this:

- 

```
1 reference
public async Task<bool> Login(LoginDto loginDto)
{
    var _user = await _userManager.FindByEmailAsync(loginDto.Email);
    bool isValidUser = await _userManager.CheckPasswordAsync(_user, loginDto.Password);

    if (_user == null || isValidUser == false)
    {
        return false;
    }

    return isValidUser;
}
```

- our Login action in our controller ends up like this:
  - 

```
// POST: api/Account/login
[HttpPost]
[Route("login")]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
[ProducesResponseType(StatusCodes.Status200OK)]
0 references
public async Task<ActionResult> Login([FromBody] LoginDto loginDto)
{
    var isValidUser = await _authManager.Login(loginDto);

    if (isValidUser == null)
    {
        return Unauthorized();
    }

    return Ok();
}
```

  - error 403 stand for:
    - you are trying to do something, you`re not authorized to do
  - e.g. you try to access a page, you are not allowed to access
  - Unauthorized():
    - we want to use, when we are unauthorized
    - and Forbidden() we would use, when you are actually not authorized
- so that it for the login endpoint, we test later
  - after we have validated, that person is in the database, then what happens next?
  - so we will see how to equip our API to issue a JWT token

#46: Implement JWT Authentication Part 1

- we want our application to use JWT to validate the users presence
- install the package
  - Microsoft.AspNetCore.Authentification.JwtBearer
- in our Program.cs file we add following line - below the AddScoped Instructions
  -

```
builder.Services.AddAuthentication(options => {

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme; // "Bearer"

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
```

- Note:
  - magic string with value "Bearer"
  - JwtBearerDefault is just a class with constants in there
  - next we will also append option to our JwtBearer!
- so we append following options:
  -

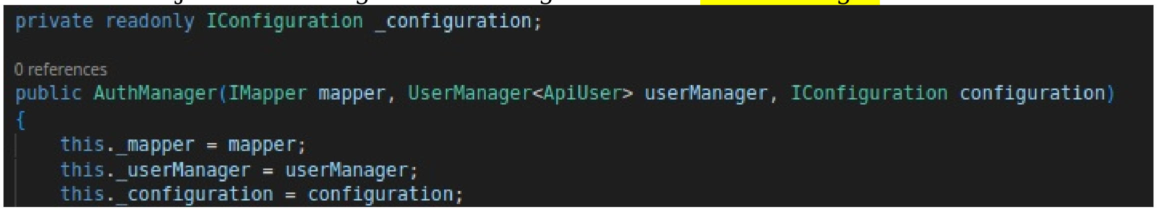
```
}).AddJwtBearer(options => {

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuerSigningKey
= true,
ValidateIssuer =
true,

ValidateAudience =
true,

ValidateLifetime =
true,

ClockSkew = TimeSpan.Zero,

ValidIssuer =
builder.Configuration["JwtSettings:Issuer"],

ValidAudience =
builder.Configuration["JwtSettings:Audience"],

IssuerSigningKey =
new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["JwtSettings:Key"]))

};

});
```

- Notes about these options:
    - e.g. ValidateIssuerSigningKey -> if anyone tries to spoof the Token, they cannot replicate our secret key encoded; they cannot replicate the token
      > even if they could, the API would reject it
    - ValidateIssuer: make sure it came from someone specific - from us
    - ValidateAudience: someone we recognise
    - ValidateLifetime: the token should be expired after some time
        - these above options are default
    - ClockSkew: should it look minutes ahead/past etc. - we don't want that in our App
    - ValidIssuer: a string
    - ValidAudience: a string
    - IssuerSigningKey: the new key we want to validate

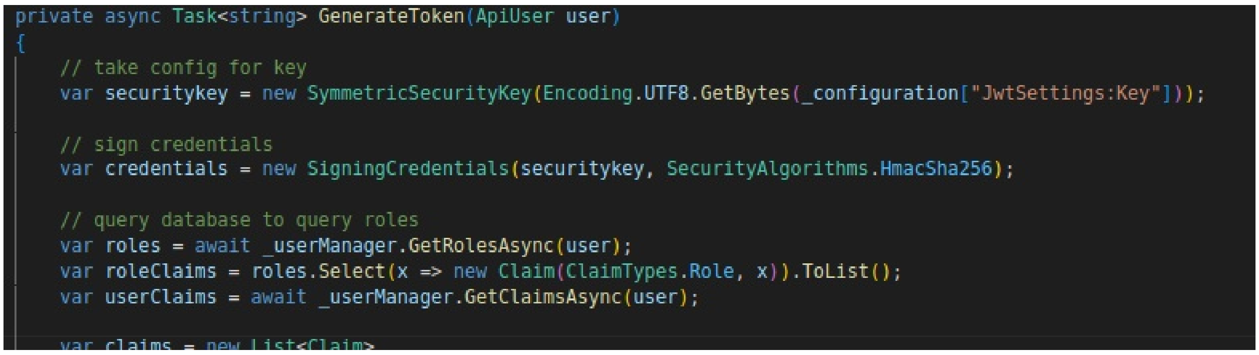
- the configuration of JwtSettings:Issuer and Audience we put in our appsettings.json!
    -
```
"JwtSettings": {

"Issuer": "HotelListingAPI",

"Audience": "HotelListingAPIClient",

"DurationInMinutes":
10,

"Key": "YourSuperSecretKey"

},
```
    - issuer: could be name of app, company or whatever
        - duration depends on your risk appetite: e.g. 10 minutes might be too short for some users
        - it can also put unneeded load on your system
        - perhaps 30 minutes, or 2 hours - whether you prefer security or comfortable use
    - Key:
        - you should usually not put anything that someone could guess
- so what did we do just here:
    - 1) we told our app, that we want to use the bearer as our default authetification scheme (see options line)
    - 2) we extend our configuration for it
    - our Program.cs then knows how to setup all the application, what to validate - when requests come in
    - next: we come back to our AuthManager Login operation and extend it
- Note about the Key:

    - you usually would rather put it in the user secrets
    - in Visual Studio: rightclick on Project in Explorer -> Manage User Secrets
    - new config file; things you usually don't want to put in version control

#47: Implement JWT Authentication Part 2

- what does it take to actually generate our tokens
- in our (Repository-)AuthManager implementation, we will create some methods
- in Visual Studio we can press Ctrl+M+O to collapse all methods and properties (Ctrl+M+P unfold)
    - in Visual Studio Code:
        - Ctrl+K+0
        - collapses all
        - Ctrl+K+J uncollapses
- we have to inject our ConfigurationManager into our AuthManager
- 

```
private readonly IConfiguration _configuration;

0 references
public AuthManager(IMapper mapper, UserManager<ApiUser> userManager, IConfiguration configuration)
{
    this._mapper = mapper;
    this._userManager = userManager;
    this._configuration = configuration;
```



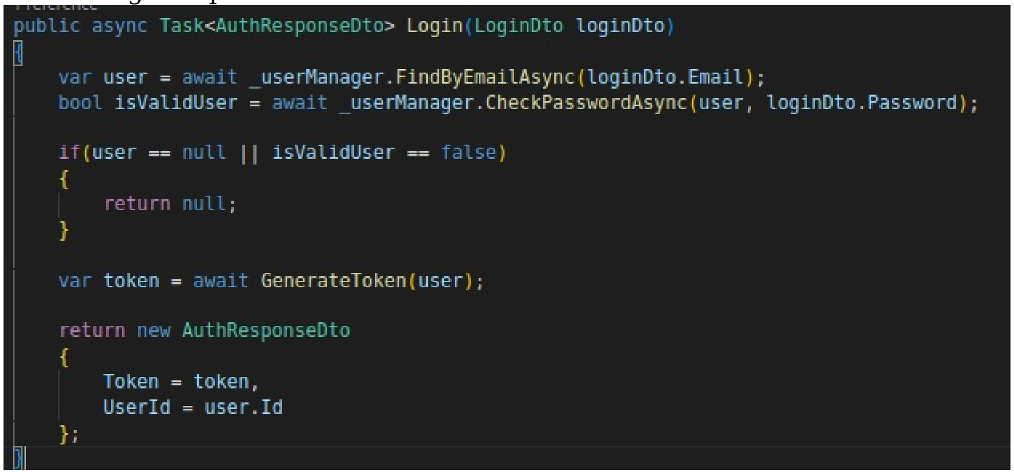
```
private async Task<string> GenerateToken(ApiUser user)
{
    // take config for key
    var securitykey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration["JwtSettings:Key"]));

    // sign credentials
    var credentials = new SigningCredentials(securitykey, SecurityAlgorithms.HmacSha256);

    // query database to query roles
    var roles = await _userManager.GetRolesAsync(user);
    var roleClaims = roles.Select(x => new Claim(ClaimTypes.Role, x)).ToList();
    var userClaims = await _userManager.GetClaimsAsync(user);

    var claims = new List<Claim>
```

    - we can do that by using IConfiguration as AuthManger Constructor argument
    - to generate the token we implement GenerateToken method
    - in SigningCredentials method we define the security algorithm to use
    - Claim: is a kind of Security Type
        - in our system we don't have any claims; but this line is just for demonstration, how it would be like with claims from databases
    - next we create the list of actual claims for our token
        - by default there are supposed to be four different claims present per user
        - whenever we are doing a JWT

        -
          ```
          var claims = new List<Claim>

          {

          new Claim(JwtRegisteredClaimNames.Sub, user.Email),

          new Claim(JwtRegisteredClaimNames.Jti, Guid.NewGuid().ToString()),

          new Claim(JwtRegisteredClaimNames.Email, user.Email),

          new Claim("uid", user.Id),

          }

          .Union(userClaims).Union(roleClaims);
          ```
        - that is the four different claims
        - course teacher Jti not sure; but probably to prevent Play Bug (kinda builtin controller); NewGuid() to make it a random GUID every each time
        - you might be working on a system, where you want to record of the user in the database - for that the last claim with uid
    - next the end of this method JwtSecurityToken Method:

        -
          ```
          // create the actual token with
          ```

```
configuration including claims

var token = new JwtSecurityToken(

issuer: _configuration["JwtSettings:Issuer"],

audience: _configuration["JwtSettings:Audience"],

claims: claims,

expires: DateTime.Now.AddMinutes(Convert.ToInt32(_configuration["JwtSettings:DurationInMinu
tes"])),

signingCredentials:
credentials

);



return new JwtSecurityTokenHandler().WriteToken(token);
```

        - we create finally the token using the configuration
        - and then return JwtSecurityTokenHandler
        - issuer: make sure it is the supposed issuer like defined - for us in config
        - claims: about the user
        - expires: important to set an expiration time,
        - signingCredentials: the credentials when we started
        - in the return statement we just return a string
- now we still need to modify our login operation
    - true/false not good enough as response!
    - we create a new Dto in Users/LoginDto
    - our revised Login implementation:
        - 

```
public async Task<AuthResponseDto> Login(LoginDto loginDto)
{
    var user = await _userManager.FindByEmailAsync(loginDto.Email);
    bool isValidUser = await _userManager.CheckPasswordAsync(user, loginDto.Password);

    if(user == null || isValidUser == false)
    {
        return null;
    }

    var token = await GenerateToken(user);

    return new AuthResponseDto
    {
        Token = token,
        UserId = user.Id
    };
}
```

    - explained at around minute 17-20
    - we also slightly modify the AccountController of Controller - the Login action
        - nothing dramatic, just about the return type - we rename isValidUser to authResponse
        - and we also return the authResponse!
- then we can test it
    - it should return 200 with right password - for my example P@assword123
    - 400 Unauthorized when wrong password or wrong user (email)
- so far it looks alright on the surface, but
    - it is not that secure!
    - jwt.io allows to watch into our token
    - it has name, and an id - which is about the issuance date!
    - there are claims which you can see,
- with the return token we see all the information!
    -

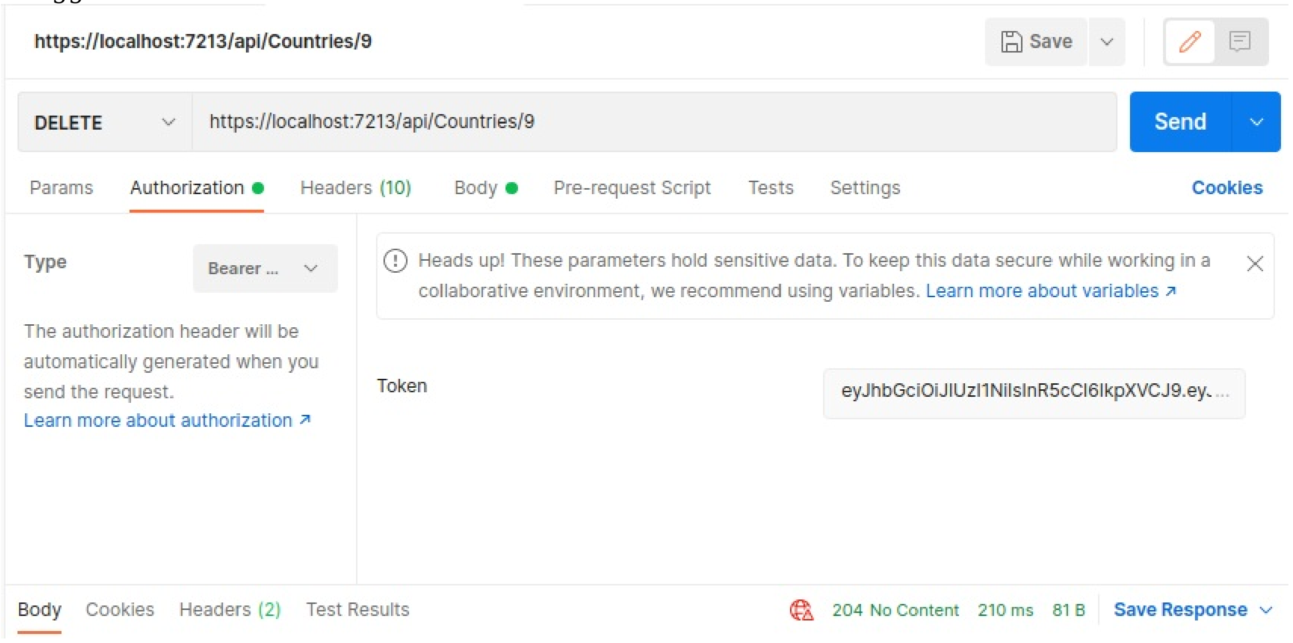
```

return Ok(authResponse);
```

#48: Protecting Endpoints

- we still have to actually implement the security - we just did preparation for it
- we can still create/change countries etc. and query almost everything
- so we restrict our API to only registered users
    - we don't want our list of countries to be available to anyone
    - e.g. we don't want random persons e.g. bots, to access and potentially attack our API (with DDoS attack)
- first step to add Authorization to our API
    - add following annotation tag to method in controller

        -
          ```
          [Authorize]
          ```
    - Note: difference if you place above the whole class or just above each action
        - above class: you have to authorize for every action of the controller
        - above single action - only those actions are protected from access
- when we do at that tag above our class and call our API of get countries we now get following status code
    - status 401 not authorized
    - but that is not necessarily the behaviour we want
    - we don't want to authorize every access to the API
    - instead we just want to authorize specific actions e.g. modify, add, delete (POST, PUT, DELETE)
- how to get authorized to access
    - use login ACTION with right user and password data
    - first we test it in Postman
    - we select Authorization and select Bearer Token here
        - we paste our token which we got via login action - on the right side
        - it is still unauthorized!
- we have to put following in our Program.cs

    -
      ```
      app.UseAuthentication(); // new

      app.UseAuthorization(); // we had
      already
      ```
    - with this we should be able to get the authorization correctly, status code should be 200-204 depending on Action
    - return code 204: mean no content came back; in Delete Action
    - swagger test:



https://localhost:7213/api/Countries/9

DELETE https://localhost:7213/api/Countries/9

Type: Bearer ...

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Token eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey...

204 No Content 210 ms 81 B

- just authorize what is need to be authorized
- let's take it to a different level when it comes to authorization
    - we have different user levels/roles
    - Admin and User
- we can specify roles in the Tags like this:

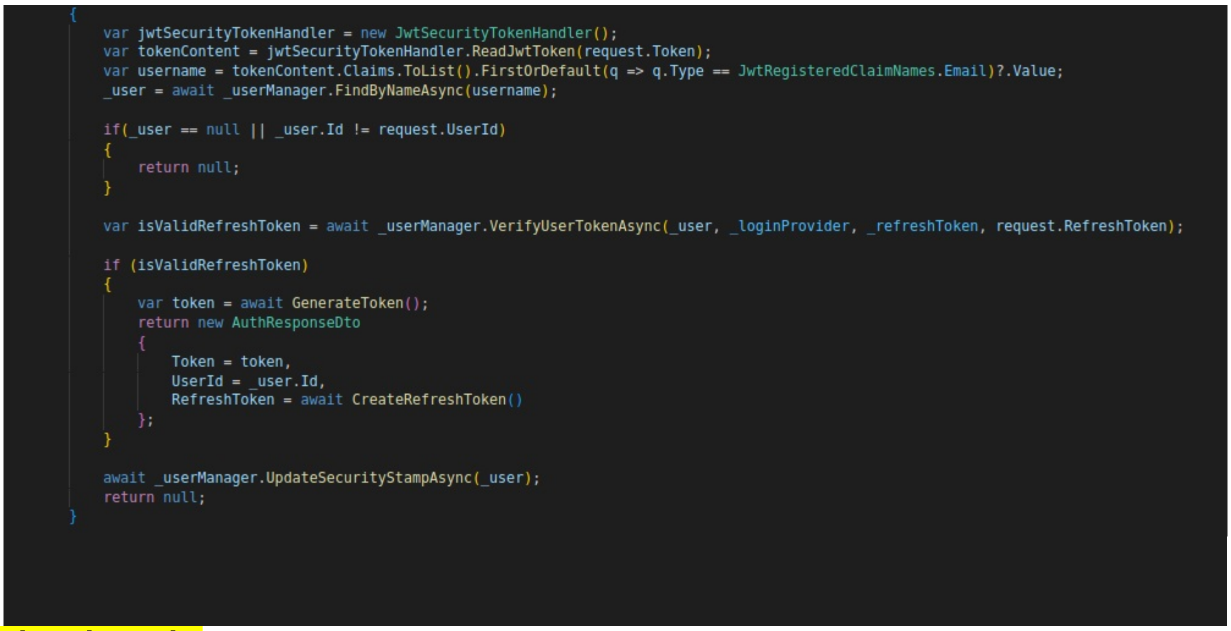
    -
      ```
      [Authorize(Roles ="Administrator,User")]
      ```
- to find the role from that token you are using

    - you can put your token into jwt.io app
- we want the functionality to set specific roles to users
    - to set admin on existing users
    - we will do that just directly via the database - for security it shouldn't be possible via the API as normal user
- then creation of admin users would be a different operation; only should only possible for Admin role
- so as a challange:
    - 1) extend your IAuthManager and AccountController to create admin users, with a logged in Admin
    - 2) secure your hotels controller as you see fit
- so that is all to secure your API
- Note for testing and status messages:
    - when you try to login with GET instead of POST:
        - you will get status 405: Method not allowed!
    -

#49: Refresh Token With Login

- the point of a refresh token
    - whenever a request comes in, and the token might have expired
    - based on the lifetime of the token
    - your user might have a poor user experience!
- so the refresh token:
    - if the request fails, we would then try to make the request again, passing over the refresh token
    - which gives them a fresh token
- to create a refresh token:
    - a token has been issued to a user
    - so we use the default identity table
    - instead of that: some persons might tell you, you have to extend the user table and so on
- steps in Repository AuthManager class
    - 1) call my user manger
        - 2) remove tokens that might have been issued already
          > we use following method of UserManager for that:
          ```
          RemoveAuthenticationTokenAsync(_user,
          _loginProvider, _refreshToken);
          ```
        - 3) next we create a new token:
          ```
          GenerateUserTokenAsync(_user,
          _loginProvider,
          _refreshToken);
          ```
        - 4) Set the new Authentication token
          ```
          SetAuthenticationTokenAsync(_user, _loginProvider, _refreshToken, newRefreshToken);
          ```
        - 5) and return the new Refresh token
    - we also need a method to actually verify the refresh token

        -
          ```
          public async Task<AuthResponseDto> VerifyRefreshToken(AuthResponseDto request)
          ```

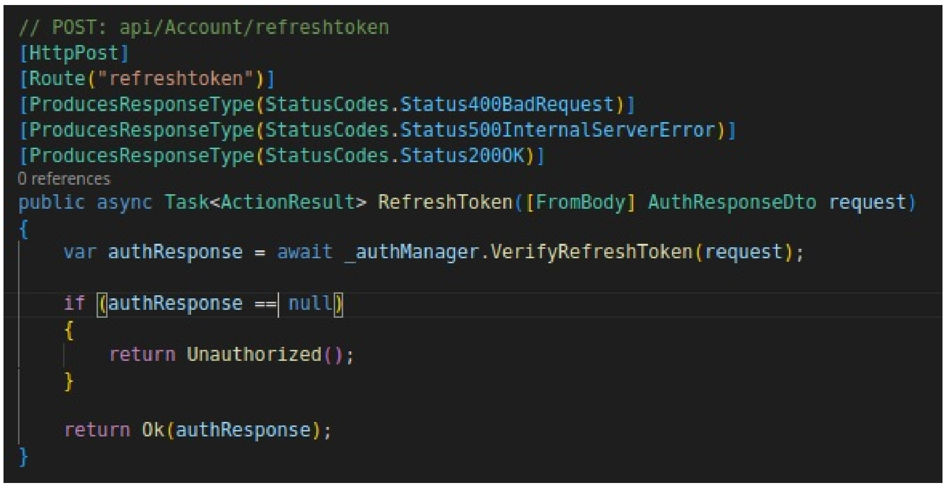


```
{
    var jwtSecurityTokenHandler = new JwtSecurityTokenHandler();
    var tokenContent = jwtSecurityTokenHandler.ReadJwtToken(request.Token);
    var username = tokenContent.Claims.ToList().FirstOrDefault(q => q.Type == JwtRegisteredClaimNames.Email)?.Value;
    _user = await _userManager.FindByNameAsync(username);

    if(_user == null || _user.Id != request.UserId)
    {
        return null;
    }

    var isValidRefreshToken = await _userManager.VerifyUserTokenAsync(_user, _loginProvider, _refreshToken, request.RefreshToken);

    if (isValidRefreshToken)
    {
        var token = await GenerateToken();
        return new AuthResponseDto
        {
            Token = token,
            UserId = _user.Id,
            RefreshToken = await CreateRefreshToken()
        };
    }

    await _userManager.UpdateSecurityStampAsync(_user);
    return null;
}
```

- to explain this code:
    - we read the content of the current token
    - and can read that result back into a strong type (process from compressed data to exctracted data)
    - so we want to read the information from that token
    - to know which user it belongs to
    - for the claims part, also see how we defined the claims in the GenerateToken method
    - this line seems a bit tricky:

```
var username = tokenContent.Claims.ToList().FirstOrDefault(q => q.Type == JwtRegisteredClai
mNames.Email)?.Value;
```

    - here we look which user matches with the token and which is also available in the database
    - the ? operator here: in case it is null: don't throw an exception
    - then the check if the user actually exists;
    - the other check about the user.id makes sure noone is succeeding with spoofing our request with an invalid user id
    - check if the RefreshToken is valid with method `VerifyUserTokenAsync()`
    - in case it is not valid, the Method UpdateSecurityStampAsync is called
    - which makes sure a saved login from that user is logged off
- we notice we can do some refactoring here
    - we created a user object in every single method
    - so we make a class variable of type ApiUser instead
    -
- we need an method for refresh token in our API Controller (Account)
    - code:



```
// POST: api/Account/refreshtoken
[HttpPost]
[Route("refreshtoken")]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
[ProducesResponseType(StatusCodes.Status500InternalServerError)]
[ProducesResponseType(StatusCodes.Status200OK)]
0 references
public async Task<ActionResult> RefreshToken([FromBody] AuthResponseDto request)
{
    var authResponse = await _authManager.VerifyRefreshToken(request);

    if (authResponse == null)
    {
        return Unauthorized();
    }

    return Ok(authResponse);
}
```

    -
    - we use use the VerifyRefreshToken method in here
    - if it is null, we return unauthorized, other wise ok(authResponse)
- another thing we have to do - add some two lines to our Program.cs
    -

```
.AddTokenProvider<DataProtectorTokenProvider<ApiUser>>("HotelListingApi")


.AddDefaultTokenProviders();
```

  - below

```
builder.Services.AddIdentityCore<ApiUser>()

.AddRoles<IdentityRole>()
```

  - this is important to make it work with our identity tables
- Note:
  - we also should define the CreateRefreshToken and VerifyRefreshToken in our interface IAuthManager
  - in our case like this:

    -
      ```
      Task<string> CreateRefreshToken();

      Task<AuthResponseDto> VerifyRefreshToken(AuthResponseDto request);
      ```
- a Note on an user question about using frontend code to store a token:
  - I was unclear at the summary of 49. It sounded like an external front-end application would discover (how?) that the JWT token had expired and would then taken specific action in the dotnet core backend to apply the refresh token?
  - Since the controllers have the "RefreshToken" POST that can be called to Verify (and create a new one if needed), this - it seems - would require the front-end to make frequent calls to "RefreshToken" since the front-end wouldn't know for sure when it had expired until it tried to call a back-end function and it failed authentication.
  - That is essentially correct. The front-end app would have stored the JWT and would be able to check if it is expired or not. So, it can do that check before making the API call and if it is within reasonable thresholds (let's say a minute to...15 minutes expired...up to your really) then you call the refresh endpoint first, get the new token, and then make the request.

Summery:

- introduced Identity Core
  - looked on all the configurations
  - seed new roles
  - how to facilicate user registration and login
  - how they can authenticate
  - how to secure the API
  - how to configure refresh tokens
  - abstract heavy operations away from our controllers through the use of repository like constructs