Table of Contents

[Create Base API Controller 3](#_Toc107275459)

[Apply the BaseApiController to UsersController 3](#_Toc107275460)

[AppUser Entity Update 3](#_Toc107275461)

[DTOs 3](#_Toc107275462)

[HashKeyDto 4](#_Toc107275463)

[UserRegisterDto 4](#_Toc107275464)

[LoginDto 4](#_Toc107275465)

[UserTokenDto 4](#_Toc107275466)

[UserDto 4](#_Toc107275467)

[Extensions 4](#_Toc107275468)

[CryptoExtensions.cs 4](#_Toc107275469)

[ComputeHashHmacSha512 4](#_Toc107275470)

[ComputeHashHmacSha512 with saltKey passed 4](#_Toc107275471)

[ByteExtensions.cs 4](#_Toc107275472)

[AreEqual 4](#_Toc107275473)

[Repository 5](#_Toc107275474)

[Repository 5](#_Toc107275475)

[IUserRepository.cs 5](#_Toc107275476)

[UserRepository.cs 5](#_Toc107275477)

[BusinessLogic 5](#_Toc107275478)

[IUserBusinessLogic.cs 5](#_Toc107275479)

[UserBusinessLogic.cs 5](#_Toc107275480)

[Account Controller 5](#_Toc107275481)

[Register 5](#_Toc107275482)

[Login 6](#_Toc107275483)

[Testing 6](#_Toc107275484)

[Remove Garbage Data from Database 6](#_Toc107275485)

[Use Postman to Test 6](#_Toc107275486)

[JWT Token 7](#_Toc107275487)

[Info 7](#_Toc107275488)

[Request/Response 8](#_Toc107275489)

[Token Service 8](#_Toc107275490)

[App Settings Json 8](#_Toc107275491)

[ConfigKeyConstants 8](#_Toc107275492)

[ConfigExtensions 8](#_Toc107275493)

[TokenService 8](#_Toc107275494)

[Register the TokenService with Dependency Injection 9](#_Toc107275495)

[Applying JWT Token 9](#_Toc107275496)

[UserBusinessLogic.Register 9](#_Toc107275497)

[UserBusinessLogic.Login 10](#_Toc107275498)

[AccountController.Register 10](#_Toc107275499)

[AccountController.Login 11](#_Toc107275500)

[Testing 11](#_Toc107275501)

[Checking Token 12](#_Toc107275502)

[GetUsers and GetUser to return UserDto 12](#_Toc107275503)

[Adding Authentication Middleware 13](#_Toc107275504)

[UserController 13](#_Toc107275505)

[Package 13](#_Toc107275506)

[AddAuthentication Extension 13](#_Toc107275507)

[Program.cs 13](#_Toc107275508)

[Testing Authentication PostMan 14](#_Toc107275509)

[Login 14](#_Toc107275510)

[GetUser 14](#_Toc107275511)

# Create Base API Controller

In Controllers folder create a c# class with name BaseApiController.cs and add the attributes to it.

using Microsoft.AspNetCore.Mvc;

namespace MSC.Api.Controllers;

[ApiController]

[Route("api/[controller]")]

public class BaseApiController : ControllerBase

{

}

## Apply the BaseApiController to UsersController

* UsersController will inherit from BaseApiController
* Remove the two attributes applied since these are not needed any more.

# AppUser Entity Update

1. Add properties to the AppUser Entity

 public byte[] PasswordHash { get; set; }

public byte[] PasswordSalt {get; set;}

1. Push to the DB by issuing the following command
   1. dotnet ef migrations add UserPasswordAdded -o Core/DB/Migrations
   2. dotnet ef database update

# DTOs

Create a new folder /Core/Dto

|  |  |
| --- | --- |
| HashKeyDto  * Create a new class /Core/Dto/HashKeyDto * It will have 2 properties   + byte[] Salt   + byte[] Hash | UserRegisterDto  * Create a new class /Core/Dto/UserRegisterDto * It will have 2 properties   + string UserName : add required and minLength attributes   + string Password: add required attribute |
| LoginDto  * Create a new class /Core/Dto/LoginDto * It will have 2 properties   + string UserName : add required attribute   + string Password: add required attribute | UserTokenDto  * Create a new class /Core/Dto/UserTokenDto * It will have 2 properties   + string UserName   + string Token |
| UserDto  * Create a new class /Core/Dto/UserDto * It will have 2 properties   + int Id   + string UserName |  |

# Extensions

## CryptoExtensions.cs

Create a new extension in Core/Extensions folder. It will get the following methods to compute hash

### ComputeHashHmacSha512

Extension to compute hash. This will return a DTO HashKeyDto

### ComputeHashHmacSha512 with saltKey passed

Extension to compute hash. This will return a DTO HashKeyDto

## ByteExtensions.cs

Create a new extension in Core/Extensions folder.

### AreEqual

Return bool true when sequence are equal for two byte arrays.

# Repository

## Repository

|  |  |
| --- | --- |
| IUserRepository.cs Create 3 signatures   1. Task<AppUser> GetUser(string userName); 2. Task<AppUser> Register(AppUser user); 3. Task<bool> UserExists(string userName); | UserRepository.cs Implement the 3 methods   1. GetUser(string userName): gets user from db 2. Register(AppUser user): will register the user 3. UserExists(string userName): checks user if exists |

# BusinessLogic

|  |  |
| --- | --- |
| IUserBusinessLogic.cs Create 2 signatures   1. Task<AppUser> Register(UserRegisterDto registerUser); 2. Task<AppUser> Login(LoginDto login); | UserBusinessLogic.cs Implement the 2 methods   1. Register(UserRegisterDto registerUser): will check and convert the info to AppUser to save 2. Task<AppUser> Login(LoginDto login): login the user |

# Account Controller

1. Add to /Controllers/AccountController and derive from BaseApiController
2. Constructor: inject the UserBusinessLogic
3. Add two actions to register and login

## Register

A POST action to register the user

public async Task<ActionResult<AppUser>> Register([FromBody] UserRegisterDto registerUser)

    {

        try

        {

            var user = await \_userBl.Register(registerUser);

            if(user == null || user.Id <= 0)

                return BadRequest("Unable to create registration");

            return Ok(user);

        }

        catch(ValidationException vEx)

        {

            return BadRequest(vEx.Message);

        }

        catch

        {

            throw;

        }

    }

## Login

A POST action to do login. Will implement the JWT token below.

    public async Task<ActionResult<AppUser>>Login([FromBody] LoginDto login)

    {

         try

        {

            var user = await \_userBl.Login(login);

            if(user == null)

                return Unauthorized("Unable to login user");

            return Ok(user);

        }

        catch(ValidationException vEx)

        {

            return Unauthorized(vEx.Message);

        }

        catch(UnauthorizedAccessException aEx)

        {

            return Unauthorized(aEx.Message);

        }

        catch

        {

            throw;

        }

    }

## Testing

### Remove Garbage Data from Database

Check 0004 Basic Setup - EntityFrameWork Setup Code First - DBContext - Sqlite.docx, section “Drop database and recreate”

### Use Postman to Test

* Run the API.
* Then using postman check functionality

#### Register Test

POST: <https://localhost:5000/api/account/register>

|  |  |  |
| --- | --- | --- |
| **Payload Test 1**  {  "username": "dave1",  "password": "password"  } | **Payload Test 2**  {  "username": "bob12",  "password": "password"  } | **Payload Test 3**  {  "username": "tom12",  "password": "password"  } |

#### Get Users Test

GET: <https://localhost:5000/api/users>

#### Get User by Id : 1 Test

GET: <https://localhost:5000/api/users/1>

#### Login Test

POST: <https://localhost:5000/api/account/login>

|  |  |  |
| --- | --- | --- |
| **Payload Test 1**  {  "username": "dave1",  "password": "password"  } | **Payload Test 2**  {  "username": "bob12",  "password": "password"  } | **Payload Test 3**  {  "username": "tom12",  "password": "password"  } |

# JWT Token

## Info

JWT stands for JSON Web Token.

* These are self-contained and can contain credentials, claims, and other information.
  + Self-contained means no session to maintain.
  + Portable – a single token can be used with multiple back ends.
  + No cookies required – mobile friendly
  + Performance – once a token is issued there is no need to make a database request to verify a user authentication.
* Contains three parts, separated by a period
  + First part: It is the header of the token and contains algorithm and type (JWT) of token. Algorithm is used to encrypt the signature in third part of the token.
  + Second part:
    - this is the payload where we can contain claims (roles etc) and credentials.
    - Will have three time stamps
      * nbf: token cannot be used before certain time
      * exp: expiry date time
      * iat: issue date
  + Third part: this is where the token signature is encrypted.

## Request/Response

Graphical user interface, text, application

Description automatically generated with medium confidence

## Token Service

### App Settings Json

Add new item to appSettings.json and appSettings.Development.json

"TokenKey": "7NCyQkWBsqV3bZsT4qShUN6qzpWUjmRs"

Token is 16 to 32 character.

### ConfigKeyConstants

Go to /Core/Constants/ConfigKeyConstants.cs and add the following key

public const string TokenKey = "TokenKey";

### ConfigExtensions

Go to /Core/Constants/ConfigConstants.cs and add the following extension method to extract the tokenKey

Note:

1. ConfigExtensions has new methods to pull by IConfiguration. Check latest, old methods are there as well
2. ServiceExtensions also received an update, check
3. Program.cs is now using IConfiguration to pick the configuration and then pass to the ServiceExtensions

public static string GetTokenKey(this IConfiguration config)

{

var tokenKey = config.GetSectionValue<string>(ConfigKeyConstants.TokenKey, string.Empty);

return tokenKey;

}

### TokenService

1. Create the token server in /Core/Services folder.
2. Create interface ITokenService with one method signature CreateToken(AppUser user).
3. Create a concreate class TokenService and implement the interface. Here will implement the CreateToken method.

#### CreateToken Method

For this the following package must be installed. Check 0001 Project - Basic Info - Setup.docx for details.

* System.IdentityModel.Tokens.Jwt v6.20.0 by Microsoft

Check the TokenService for the code. It is well commented. The steps involve

1. Creating symmetric security key
2. Creating claims
3. Creating credentials
4. Creating token descriptor
5. Creating token handler
6. Creating token
7. And finally getting the written token

### Register the TokenService with Dependency Injection

1. Go to /Core/Extensions/ServiceExtensions
2. Add the TokenService as AddScoped to extension method RegisterRepos

//services

collection.AddScoped<ITokenService, TokenService>();

# Applying JWT Token

## UserBusinessLogic.Register

Change the register method to return UserTokenDto. This will also create token using the TokenService created above.

    public async Task<UserTokenDto> Register(UserRegisterDto registerUser)

    {

        if (registerUser == null)

            throw new ValidationException("Invalid user");

        var user = await RegisterUser(registerUser);

        if(user == null || user.Id <= 0)

            throw new ValidationException("Unable to create registration");

        var userToken = new UserTokenDto

        {

            UserName = user.UserName,

            Token = \_tokenService.CreateToken(user)

        };

        return userToken;

    }

## UserBusinessLogic.Login

Change the login method to return UserTokenDto. This will also create token using the TokenService created above.

    public async Task<UserTokenDto> Login(LoginDto login)

    {

        if (login == null)

            throw new ValidationException("Login info missing");

        var user = await \_usersRepo.GetUser(login.UserName);

        if (user == null || user.PasswordSalt == null || user.PasswordHash == null)

            throw new UnauthorizedAccessException("Either username or password is wrong");

        //password is hashed in db. Hash login password and check against the DB one

        var hashKeyLogin = login.Password.ComputeHashHmacSha512(user.PasswordSalt);

        if (hashKeyLogin == null)

            throw new UnauthorizedAccessException("Either username or password is wrong");

        //both are byte[]

        if (!hashKeyLogin.Hash.AreEqual(user.PasswordHash))

            throw new UnauthorizedAccessException("Either username or password is wrong");

        var userToken = new UserTokenDto

        {

            UserName = user.UserName,

            Token = \_tokenService.CreateToken(user)

        };

        return userToken;

    }

## AccountController.Register

Change register action to return UserTokenDto. Also update the if statement

    [HttpPost("register")]

    public async Task<ActionResult<UserTokenDto>> Register([FromBody] UserRegisterDto registerUser)

    {

        try

        {

            var user = await \_userBl.Register(registerUser);

            if (user == null || string.IsNullOrWhiteSpace(user.UserName) || string.IsNullOrWhiteSpace(user.Token))

                return BadRequest("Unable to create registration");

            return Ok(user);

        }

        catch(ValidationException vEx)

        {

            return BadRequest(vEx.Message);

        }

        catch

        {

            throw;

        }

    }

## AccountController.Login

Change login action to return UserTokenDto.

    [HttpPost("login")]

    public async Task<ActionResult<UserTokenDto>>Login([FromBody] LoginDto login)

    {

         try

        {

            var user = await \_userBl.Login(login);

            if (user == null || string.IsNullOrWhiteSpace(user.UserName) || string.IsNullOrWhiteSpace(user.Token))

                return Unauthorized("Unable to login user");

            return Ok(user);

        }

        catch(ValidationException vEx)

        {

            return Unauthorized(vEx.Message);

        }

        catch(UnauthorizedAccessException aEx)

        {

            return Unauthorized(aEx.Message);

        }

        catch

        {

            throw;

        }

    }

## Testing

Follow [above](#_Testing) testing instructions. No need to remove the data from the database.

Now when you will login you’ll see UserTokenDto in postman.

{

    "userName": "bob12",

    "token": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiJib2IxMiIsIm5iZiI6MTY1NjM5MTgxMywiZXhwIjoxNjU2OTk2NjEyLCJpYXQiOjE2NTYzOTE4MTN9.Lh0AbtXZpYPN1YBIyB5pnHXmjp0RRrwxVwBXdvWndbutsWlxM87Nw1PfYv8wtdU7RFV9QV3txBlU\_vhzcO3A2g"

}

### Checking Token

Copy the token and go to <https://jwt.io/> and paste the token in there to look at

Graphical user interface, text, application

Description automatically generated

#### Verify Signature

Copy the token from the app settings.json file and put it in the text box under Verify Signature.

The token on the left should be un changed at this point. You can test with any other string and the token will change.

# GetUsers and GetUser to return UserDto

* The change is needed in both UsersBusinessLogic and UsersController.
* Test the change as well

# Adding Authentication Middleware

## UserController

1. Apply AllowAnonymous attribute to GetUsers action
2. Apply Authorize attribute to GetUser action
3. And test again. GetUsers should pass where as GetUser should fail

## Package

Following package must be installed for this. [Check 0001 Project - Basic Info - Setup.docx](0001%20Project%20-%20Basic%20Info%20-%20Setup.docx) for details.

* Microsoft.AspNetCore.Authentication.JwtBearer v6.0.6 by Microsoft

## ServiceIdentityExtensions.cs

Create a new extension class under /Core/Extensions/ServiceIdentityExtensions.cs and add the following method to it

    public static void AddIdentityServices(this IServiceCollection collection, IConfiguration config)

    {

        var tokenKey = Encoding.UTF8.GetBytes(config.GetTokenKey());

        collection.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

                .AddJwtBearer(options =>

                {

                    options.TokenValidationParameters = new TokenValidationParameters

                    {

                        ValidateIssuerSigningKey = true,

                        IssuerSigningKey = new SymmetricSecurityKey(tokenKey),

                        ValidateIssuer = false,

                        ValidateAudience = false

                    };

                });

    }

## Program.cs

Two changes are need here.

AddAuthentication by calling the RegisterAuthentication extension method. This will go in the CUSTOM section which is at the top part of the file

builder.Services. AddIdentityServices(configuration);

Next we will use Authentication. This is in the CUSTOM section towards the bottom of the file.

Important: ordering is important here.

//CUSTOM: Start

//ordering is important here. UseCors before UseAuthentication and UseAuthentication before UseAuthorization

app.UseCors(myAllowSpecificOrigins);

app.UseAuthentication();

//CUSTOM: End

# Testing Authentication PostMan

## Login

POST: <https://localhost:5000/api/account/login>

Payload:

{

    "username": "bob12",

    "password": "password"

}

This will give back following result

{

    "userName": "bob12",

    "token": "eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiJib2IxMiIsIm5iZiI6MTY1NjM5NTk5NSwiZXhwIjoxNjU3MDAwNzk1LCJpYXQiOjE2NTYzOTU5OTV9.\_HLCu8WJsmEfoA\_4YZfBJc3z\_saG0BLlpqreBn1ZhXGIibIoHwe7n5aWA40ol\_4c204W-IQfsXfeimKVLZ7n4g"

}

## GetUser

GET: <https://localhost:5000/api/users/1>

Copy the [above token](#_Login) returned under login.

Then click Header tab and add “Authorization” header

Header Key: Authorization

Header Value: Bearer eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.eyJuYW1laWQiOiJib2IxMiIsIm5iZiI6MTY1NjM5MTgxMywiZXhwIjoxNjU2OTk2NjEyLCJpYXQiOjE2NTYzOTE4MTN9.Lh0AbtXZpYPN1YBIyB5pnHXmjp0RRrwxVwBXdvWndbutsWlxM87Nw1PfYv8wtdU7RFV9QV3txBlU\_vhzcO3A2g

Type “Bearer” then put space and then paste the token that is copied

Send the request and this time you should get the result back.