**Models and Controllers:**

* All database models should be organized in the "Models" namespace or folder.
* Controllers should be organized in the "Controllers" namespace or folder.
* Authentication-related functionality (Identity, JWT) should be encapsulated within the appropriate models and controllers.

Models:

The name of my web API is called ShopSite.CW.WebApp which is a simple eCommerce to sell anything such as beverages, and cloths.

created a new web API template in a folder called ShopSite.CW.WebApp by running this command in the terminal “dotnet new webapi”.

Created a Model folder to contain all database models. Inside this folder, I created these classes

This is to assign a user to a specific role ie Admin or Customer

namespace ShopSite.CW.WebApp.Models

{

public class AssignRoleModel

{

public string UserId { get; set; }

public string RoleName { get; set; }

}

}

This is for authentication when logging in.

namespace ShopSite.CW.WebApp.Models

{

public class AuthModel

{

public string Email { get; set; }

public string Password { get; set; }

}

}

This class represents a category entity typically used in an application such as an online shop.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.Json.Serialization;

using System.Threading.Tasks;

namespace ShopSite.CW.WebApp.Models

{

public class Category

{

public int CategoryId { get; set; }

public string Name { get; set; }

public string? Description { get; set; }

// Navigation property for Products

[JsonIgnore]

public List<Product>? Products { get; set; }

}

}

**Properties**:

-CategoryId: An integer property representing the unique identifier of the category.

-Name: A string property representing the name of the category.

-Description: An optional string property representing a brief description of the category. It is nullable (string?), indicating that it can be assigned a null value.

**Navigation Property**:

-Products: A navigation property representing the collection of products associated with this category. It is declared as a list of Product objects (List<Product>). This property is marked with the [JsonIgnore] attribute, indicating that it should be ignored during JSON serialization. This is useful to prevent circular reference issues when serializing objects to JSON format.

By defining this class, you can create instances of Category objects, each representing a distinct category in your application. These objects can be used to organize and manage products, and the navigation property Products allows you to access the list of products associated with a particular category.

This class represents an order entity

using Microsoft.AspNetCore.Identity;

namespace ShopSite.CW.WebApp.Models

{

public class Order

{

public int OrderId { get; set; }

public DateTime OrderDate { get; set; }

public string CustomerId {get; set;}

public IdentityUser? Customer {get; set;}

public string CustomerName { get; set; }

public string ShippingAddress { get; set; }

public decimal? TotalAmount { get; set; }

public bool? IsShipped { get; set; }

// Navigation property for OrderItems

public List<OrderItem>? OrderItems { get; set; }

public Order()

{

OrderDate = DateTime.Now;

}

}

}

**Properties**:

-OrderId: An integer property representing the unique identifier of the order.

-OrderDate: A DateTime property representing the date and time when the order was placed.

-CustomerId: A string property representing the identifier of the customer who placed the order.

-Customer: An optional property representing the customer who placed the order. It's of type IdentityUser, which suggests that it's using ASP.NET Core Identity for user management. The property is nullable (IdentityUser?), indicating that it can be assigned a null value.

-CustomerName: A string property representing the name of the customer who placed the order.

-ShippingAddress: A string property representing the shipping address for the order.

-TotalAmount: A nullable decimal property representing the total amount of the order.

-IsShipped: A nullable boolean property indicating whether the order has been shipped.

**Navigation Property**:

-OrderItems: A navigation property representing the collection of order items associated with this order. It's declared as a list of OrderItem objects (List<OrderItem>). Each OrderItem represents a product included in the order.

**Constructor**:

-The constructor initializes the OrderDate property with the current date and time using DateTime.Now.

By defining this class, you can create instances of Order objects, each representing a distinct order in your application. These objects can be used to manage and track orders, including details such as the customer, shipping address, order items, and total amount.

This is a class named OrderItem within the namespace ShopSite.CW.WebApp.Models. This class represents an item within an order.

namespace ShopSite.CW.WebApp.Models

{

public class OrderItem

{

public int OrderItemId { get; set; }

//public int OrderId { get; set; } // Identifier for the order this item belongs to

public int Quantity { get; set; } // Quantity of the product in this order item

// Foreign key for Product

public int ProductId { get; set; }

// Navigation property for Product

public Product? Product { get; set; }

}

}

* **Properties**:
  + OrderItemId: An integer property representing the unique identifier of the order item.
  + Quantity: An integer property representing the quantity of the product included in this order item.
  + ProductId: An integer property representing the foreign key for the product associated with this order item.
* **Navigation Property**:
  + Product: A navigation property representing the product associated with this order item. It's declared as an instance of the Product class. This allows you to access the details of the product related to this order item.

By defining this class, you can create instances of OrderItem objects, each representing a distinct item within an order in your application. These objects can be used to manage and track individual items included in orders, including details such as the quantity and the associated product.

This is a class named Product within the namespace ShopSite.CW.WebApp.Models. This class represents a product entity .

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.Json.Serialization;

using System.Threading.Tasks;

namespace ShopSite.CW.WebApp.Models

{

public class Product

{

public int ProductId { get; set; }

public string Name { get; set; }

public string? Description { get; set; }

public decimal Price { get; set; }

public int StockQuantity { get; set; }

public DateTime CreatedOn { get; set; }

// Property for storing product image as binary data

public byte[]? ImageData { get; set; }

public string? ImageMimeType { get; set; }

// Navigation property for Suppliers

public int? SupplierId {get; set;}

public Supplier? Supplier { get; set; }

// Navigation property for Categories

public int? CategoryId { get; set; }

public Category? Category { get; set; }

public Product()

{

CreatedOn = DateTime.Now;

}

}

}

* **Properties**:
  + ProductId: An integer property representing the unique identifier of the product.
  + Name: A string property representing the name of the product.
  + Description: An optional string property representing a brief description of the product. It is nullable (string?), indicating that it can be assigned a null value.
  + Price: A decimal property representing the price of the product.
  + StockQuantity: An integer property representing the quantity of the product in stock.
  + CreatedOn: A DateTime property representing the date and time when the product was created. It is initialized to the current date and time using DateTime.Now in the constructor.
* **Image Properties**:
  + ImageData: A byte array property representing the binary data of the product image.
  + ImageMimeType: A string property representing the MIME type of the product image.
* **Navigation Properties**:
  + SupplierId: An integer property representing the foreign key for the supplier associated with this product.
  + Supplier: A navigation property representing the supplier associated with this product. It's declared as an instance of the Supplier class.
  + CategoryId: An integer property representing the foreign key for the category associated with this product.
  + Category: A navigation property representing the category associated with this product. It's declared as an instance of the Category class.
* **Constructor**:
  + The constructor initializes the CreatedOn property with the current date and time using DateTime.Now.

By defining this class, you can create instances of Product objects, each representing a distinct product in your application. These objects can be used to manage and track products, including details such as the name, price, quantity in stock, associated supplier, and category.

This code defines a class named ShopContext that inherits from IdentityDbContext<IdentityUser>. This class serves as the database context for your ASP.NET Core application, specifically for managing the entities related to the shop site.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using Microsoft.AspNetCore.Identity;

namespace ShopSite.CW.WebApp.Models

{

public class ShopContext : IdentityDbContext<IdentityUser>

{

public ShopContext(DbContextOptions<ShopContext> *options*) : base(options)

{

}

public DbSet<Category> Categories { get; set; }

public DbSet<Order> Orders { get; set; }

public DbSet<OrderItem> OrderItems { get; set; }

public DbSet<Product> products{get; set;}

public DbSet<Supplier> suppliers{get; set;}

public object ProductSuppliers { get; internal set; }

}

}

* **Constructor**:
  + The constructor initializes the ShopContext class with the provided DbContextOptions<ShopContext> using the base constructor.
* **DbSet Properties**:
  + Categories: A DbSet property representing the collection of categories in the database.
  + Orders: A DbSet property representing the collection of orders in the database.
  + OrderItems: A DbSet property representing the collection of order items in the database.
  + Products: A DbSet property representing the collection of products in the database.
  + Suppliers: A DbSet property representing the collection of suppliers in the database.
* **Identity User**:
  + The generic argument IdentityUser in IdentityDbContext<IdentityUser> indicates that the ShopContext is using ASP.NET Core Identity for user management, and IdentityUser is the default user entity provided by Identity.
* **Other Properties**:
  + ProductSuppliers: An additional property that seems to be mistakenly defined as an object and set as internal. It doesn't seem to be related to any DbSet and might not be necessary. You may consider removing it unless it serves a specific purpose.

By defining this class, you are configuring the database context for your ASP.NET Core application, specifying the entities (DbSet properties) that will be used to interact with the database. These entities represent tables in the database, and the ShopContext class provides methods for querying and manipulating data in these tables.

This is a class named Supplier within the namespace ShopSite.CW.WebApp.Models. This class represents a supplier entity.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.Json.Serialization;

using System.Threading.Tasks;

namespace ShopSite.CW.WebApp.Models

{

public class Supplier

{

public int SupplierId { get; set; }

public string Name { get; set; }

public string? ContactName { get; set; }

public string? ContactEmail { get; set; }

public string? Phone { get; set; }

public string? Address { get; set; }

// Navigation property for Products

public List<Product>? Products { get; set; }

}

}

* **Properties**:
  + SupplierId: An integer property representing the unique identifier of the supplier.
  + Name: A string property representing the name of the supplier.
  + ContactName: An optional string property representing the name of the contact person at the supplier company. It is nullable (string?), indicating that it can be assigned a null value.
  + ContactEmail: An optional string property representing the email address of the contact person at the supplier company.
  + Phone: An optional string property representing the phone number of the supplier.
  + Address: An optional string property representing the address of the supplier.
* **Navigation Property**:
  + Products: A navigation property representing the collection of products supplied by this supplier. It's declared as a list of Product objects (List<Product>). This property allows you to access the list of products associated with this supplier.

By defining this class, you can create instances of Supplier objects, each representing a distinct supplier in your application. These objects can be used to manage and track supplier information, including details such as the name, contact information, and associated products.

This is a class named UpdateRoleModel within the namespace ShopSite.CW.WebApp.Models. This class represents a model used for updating roles in your application.

namespace ShopSite.CW.WebApp.Models

{

public class UpdateRoleModel

{

public string RoleId { get; set; }

public string NewRoleName { get; set; }

}

}

* + **Properties**:
  + RoleId: A string property representing the identifier of the role to be updated.
  + NewRoleName: A string property representing the new name for the role after the update.

By defining this class, you can create instances of UpdateRoleModel objects, each representing the data required to update a role in your application. These objects can be used to pass information between different parts of your application, such as controllers and views, to perform role updates based on user actions or system requirements.

……………

Before I generated the controllers for each model class,

I Installed required packages using the terminal:

dotnet add package Microsoft.EntityFrameworkCore

dotnet add package Microsoft.EntityFrameworkCore.Sqlite

dotnet add package Microsoft.EntityFrameworkCore.Tools

Updated the appsettings.json with connection string:

"ConnectionStrings": {

"Connection": "Data Source=shop.db;"

},

now, in program.cs, I register the context with dependency injection:

//code

builder.Services.AddControllers();

builder.Services.AddDbContext<SchoolContext>(options =>

options.UseSqlite(builder.Configuration.GetConnectionString("Connection")));

and then added app.MapControllers();

just before the app.UseHttpsRedirection();

Now, to add the migrations and update the DB (terminal):

as we are using vs code, we need to install a few things in order first:

dotnet new tool-manifest

dotnet tool install --local dotnet-ef

dotnet ef migrations add InitialCreate

dotnet ef database update

Also install these packages to generate the code for controllers:

dotnet tool install --local dotnet-aspnet-codegenerator

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Now run in the terminal these commands to generate the code for controllers:

dotnet aspnet-codegenerator controller -name ProductsController -async -api -m Product -dc ShopContext -outDir Controllers

above command explained:

This terminal command utilizes the ASP.NET Core scaffolding tool (aspnet-codegenerator) to generate a controller for handling CRUD operations related to products in your ASP.NET Core Web API project.

Let's break down the components of this command:

* dotnet aspnet-codegenerator controller: This is the command-line interface for invoking the ASP.NET Core scaffolding tool to generate a controller.
* -name ProductsController: Specifies the name of the controller to be generated. In this case, the controller will be named ProductsController.
* -async: Indicates that asynchronous action methods should be generated in the controller. Asynchronous methods can improve the scalability and responsiveness of your application by allowing it to handle more concurrent requests.
* -api: Specifies that the controller should be generated as an API controller. API controllers in ASP.NET Core are specifically designed for building RESTful APIs and typically return data in JSON format.
* -m Product: Specifies the model class (Product) that the controller will operate on. The controller will include actions for CRUD operations (Create, Read, Update, Delete) for the Product model.
* -dc ShopContext: Specifies the database context (ShopContext) that the controller will use for accessing data. The controller will interact with the database through this context to perform CRUD operations on the Product model.
* -outDir Controllers: Specifies the output directory (Controllers) where the generated controller file will be placed. In this case, the generated ProductsController file will be placed in the Controllers directory.

Overall, this command generates a controller named ProductsController with asynchronous API action methods for performing CRUD operations on the Product model, using the ShopContext database context for data access, and placing the generated controller file in the Controllers directory of your project.

ProductController:

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using ShopSite.CW.WebApp.Models;

namespace ShopSite.CW.WebApp.Controllers

{

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

[ApiController]

public class ProductsController : ControllerBase

{

private readonly ShopContext \_context;

public ProductsController(ShopContext *context*)

{

\_context = context;

}

// GET: api/Products

[HttpGet]

public async Task<ActionResult<IEnumerable<Product>>> Getproducts()

{

return await \_context.products.Include(*p* => p.Supplier).Include(*p* => p.Category).ToListAsync();

}

// GET: api/Products/5

[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int *id*)

{

var product = await \_context.products.Include(*p* => p.Supplier).Include(*p* => p.Category).FirstOrDefaultAsync(*p* => p.ProductId == id);

if (product == null)

{

return NotFound();

}

return product;

}

// PUT: api/Products/5

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPut("{id}")]

[Authorize(Roles ="Admin")]

public async Task<IActionResult> PutProduct(int *id*, Product *product*)

{

if (id != product.ProductId)

{

return BadRequest();

}

\_context.Entry(product).State = EntityState.Modified;

try

{

await \_context.SaveChangesAsync();

}

catch (DbUpdateConcurrencyException)

{

if (!ProductExists(id))

{

return NotFound();

}

else

{

throw;

}

}

return NoContent();

}

// POST: api/Products

// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754

[HttpPost]

[Authorize(Roles ="Admin")]

public async Task<ActionResult<Product>> PostProduct(Product *product*)

{

\_context.products.Add(product);

await \_context.SaveChangesAsync();

return CreatedAtAction("GetProduct", new { id = product.ProductId }, product);

}

// DELETE: api/Products/5

[HttpDelete("{id}")]

[Authorize(Roles ="Admin")]

public async Task<IActionResult> DeleteProduct(int *id*)

{

var product = await \_context.products.FindAsync(id);

if (product == null)

{

return NotFound();

}

\_context.products.Remove(product);

await \_context.SaveChangesAsync();

return NoContent();

}

private bool ProductExists(int *id*)

{

return \_context.products.Any(*e* => e.ProductId == id);

}

}

}

Let's break down the components of this controller:

1. \*\*Controller Attributes\*\*:

- `[Route("api/[controller]")]`: Specifies the route template for the controller. In this case, the `[controller]` token will be replaced with the controller name (`ProductsController`). So, the route for this controller will be `/api/Products`.

- `[ApiController]`: Indicates that the controller is an API controller. This attribute enables various API-specific behaviors, such as automatic model validation and response formatting.

2. \*\*Constructor\*\*:

- The constructor injects an instance of `ShopContext` into the controller. This allows the controller to interact with the database through the `ShopContext` instance.

3. \*\*Action Methods\*\*:

- `Getproducts()`: Handles HTTP GET requests to retrieve all products. It returns a list of products, including their associated supplier and category information.

- `GetProduct(int id)`: Handles HTTP GET requests to retrieve a specific product by its ID. It returns the product object, including its associated supplier and category information.

- `PutProduct(int id, Product product)`: Handles HTTP PUT requests to update an existing product. It requires the user to have the "Admin" role. It updates the product information based on the provided ID.

- `PostProduct(Product product)`: Handles HTTP POST requests to create a new product. It requires the user to have the "Admin" role. It adds the new product to the database.

- `DeleteProduct(int id)`: Handles HTTP DELETE requests to delete a product by its ID. It requires the user to have the "Admin" role. It removes the product from the database.

4. \*\*Authorization\*\*:

- `[Authorize(Roles ="Admin")]`: Specifies that certain action methods (`PutProduct`, `PostProduct`, `DeleteProduct`) require the user to have the "Admin" role in order to access them.

5. \*\*Helper Method\*\*:

- `ProductExists(int id)`: A private helper method that checks whether a product with the specified ID exists in the database.

Overall, this controller provides a RESTful API for managing product data, ensuring that certain operations are restricted to users with the appropriate role ("Admin").

……………………………..

\*Creating email service for sign up.

In shopcontext, we changed this line (inheriting class basically):

public class ShopContext : DbContext

to

public class ShopContext : IdentityDbContext<IdentityUser>

and then I installed this identity because I saw some errors:

dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore

and

using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

using Microsoft.AspNetCore.Identity;

at the top of the ShopContext.cs file.

Now in the program.cs file and I added another service dependency injection:

builder.Services.AddIdentity<IdentityUser, IdentityRole>()

.AddEntityFrameworkStores<ShopContext>().AddDefaultTokenProviders();

To create tables in database, I installed these packages by running these commands in the terminal:

dotnet ef migrations add IdentityAdded

dotnet ef database update

Now the new tables have been added to the database.

I installed this dependency “MailKit (by Jeffrey Stedfast)” using nuget gallery extension

add the registration with email verification and sign in functionalities to our existing application.

In the Models folder, add a new class called

AuthModel.cs and put this code in there:

//code

public class AuthModel

{

public string Email { get; set; }

public string Password { get; set; }

}

Now in the project directory create a new folder named

Services (at the same level as Models and Controllers)

and inside this folder create two new classes named

EmailService.cs and EmailSettings.cs

Inside the EmailService.cs file put this code:

//code

public class EmailService

{

private readonly EmailSettings \_emailSettings;

public EmailService(IOptions<EmailSettings> emailSettings)

{

\_emailSettings = emailSettings.Value;

}

public void SendEmail(string toEmail, string subject, string body)

{

var message = new MimeMessage();

message.From.Add(new MailboxAddress("Support CareApp", \_emailSettings.SmtpUsername));

message.To.Add(new MailboxAddress("Reciever Name", toEmail));

message.Subject = subject;

var textPart = new TextPart("plain")

{

Text = body

};

message.Body = textPart;

using (var client = new SmtpClient())

{

client.Connect(\_emailSettings.SmtpServer, \_emailSettings.SmtpPort,

SecureSocketOptions.StartTls);

client.Authenticate(\_emailSettings.SmtpUsername, \_emailSettings.SmtpPassword);

client.Send(message);

client.Disconnect(true);

}

}

}

In the EmailSettings.cs put this code inside it:

//code

public class EmailSettings

{

public string SmtpServer { get; set; }

public int SmtpPort { get; set; }

public string SmtpUsername { get; set; }

public string SmtpPassword { get; set; }

}

dd a new empty

controller (by right clicking the controller folder

and add new file -> AccountController.cs and put this

code inside this controller:

namespace IdentityPractice.Controllers

{

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

[ApiController]

public class AccountController : ControllerBase

{

private readonly UserManager<IdentityUser> \_userManager;

private readonly SignInManager<IdentityUser> \_signInManager;

private readonly EmailService \_emailService;

public AccountController(UserManager<IdentityUser> userManager,

SignInManager<IdentityUser> signInManager, EmailService emailService)

{

\_userManager = userManager;

\_signInManager = signInManager;

\_emailService = emailService;

}

[HttpPost("register")]

public async Task<IActionResult> Register(AuthModel model)

{

var user = new IdentityUser { UserName = model.Email, Email = model.Email };

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

if (result.Succeeded)

{

// Generate an email verification token

var token = await \_userManager.GenerateEmailConfirmationTokenAsync(user);

// Create the verification link

var verificationLink = Url.Action("VerifyEmail", "Account", new { userId = user.Id, token =

token }, Request.Scheme);

// Send the verification email

var emailSubject = "Email Verification";

var emailBody = $"Please verify your email by clicking the following link: {verificationLink}";

\_emailService.SendEmail(user.Email, emailSubject, emailBody);

return Ok("User registered successfully. An email verification link has been sent.");

}

return BadRequest(result.Errors);

}

// Add an action to handle email verification

[HttpGet("verify-email")]

public async Task<IActionResult> VerifyEmail(string userId, string token)

{

var user = await \_userManager.FindByIdAsync(userId);

if (user == null)

{

return NotFound("User not found.");

}

var result = await \_userManager.ConfirmEmailAsync(user, token);

if (result.Succeeded)

{

return Ok("Email verification successful.");

}

return BadRequest("Email verification failed.");

}

[HttpPost("login")]

public async Task<IActionResult> Login(AuthModel model)

{

var result = await \_signInManager.PasswordSignInAsync(model.Email, model.Password,

isPersistent: false, lockoutOnFailure: false);

if (result.Succeeded)

{

return Ok("Login successful.");

}

return Unauthorized("Invalid login attempt.");

}

}

}

In the program.cs, I registered my email service in our program.cs file:

builder.Services.Configure<EmailSettings>(builder.Configuration.GetSection("EmailSettings"));

builder.Services.AddScoped<EmailService>();

As we have mentioned configuration section, we need to add this in the appsettings.json file after the connection string:

"EmailSettings": {

"SmtpServer": "smtp.gmail.com",

"SmtpPort": 587,

"SmtpUsername": "kobejoshhtc10@gmail.com",

"SmtpPassword": "rzps kqmb ceex ffxm"

},

Now we will add JWT token and for that we need to install this dependency:

Microsoft.AspNetCore.Authentication.JwtBearer

in program.cs, add this

builder.Services.AddAuthentication(*options* =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(*options* =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = true,

ValidateAudience = true,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

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

ValidAudience = builder.Configuration["Jwt:Issuer"],

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

};

});

in appsettings.json file, add this:

"Jwt": {

"Key": "ThisIsAVeryLongRandomStringUsedAsJWTKey",

"Issuer": "http://localhost:5186",

"ExpireHours": 87000

},

updated AccountController.cs

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Mvc;

using ShopSite.CW.WebApp.Models;

using ShopSite.CW.WebApp.Services;

using Microsoft.IdentityModel.Tokens;

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

//using ShopSite.CW.WebApp.Services;

namespace ShopSite.CW.WebApp.Controllers

{

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

[ApiController]

public class AccountController : ControllerBase

{

private readonly UserManager<IdentityUser> \_userManager;

private readonly SignInManager<IdentityUser> \_signInManager;

private readonly EmailService \_emailService;

private readonly IConfiguration \_configuration;

public AccountController(UserManager<IdentityUser> *userManager*, SignInManager<IdentityUser> *signInManager*,

EmailService *emailService*, IConfiguration *configuration*)

{

\_userManager = userManager;

\_signInManager = signInManager;

\_emailService = emailService;

\_configuration = configuration;

}

[HttpPost("register")]

public async Task<IActionResult> Register(AuthModel *model*)

{

var user = new IdentityUser { UserName = model.Email, Email = model.Email };

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

if (result.Succeeded)

{

// Generate an email verification token

var token = await \_userManager.GenerateEmailConfirmationTokenAsync(user);

// Create the verification link

var verificationLink = Url.Action("VerifyEmail", "Account", new { userId = user.Id, token = token }, Request.Scheme);

// Send the verification email

var emailSubject = "Email Verification";

var emailBody = $"Please verify your email by clicking the following link: {verificationLink}";

\_emailService.SendEmail(user.Email, emailSubject, emailBody);

return Ok("User registered successfully. An email verification link has been sent.");

}

return BadRequest(result.Errors);

}

// Add an action to handle email verification

[HttpGet("verify-email")]

public async Task<IActionResult> VerifyEmail(string *userId*, string *token*)

{

var user = await \_userManager.FindByIdAsync(userId);

if (user == null)

{

return NotFound("User not found.");

}

var result = await \_userManager.ConfirmEmailAsync(user, token);

if (result.Succeeded)

{

return Ok("Email verification successful.");

}

return BadRequest("Email verification failed.");

}

[HttpPost("login")]

public async Task<IActionResult> Login(AuthModel *model*)

{

var result = await \_signInManager.PasswordSignInAsync(model.Email, model.Password, *isPersistent*: false, *lockoutOnFailure*: false);

if (result.Succeeded)

{

var user = await \_userManager.FindByEmailAsync(model.Email);

var roles = await \_userManager.GetRolesAsync(user);

var token = GenerateJwtToken(user,roles);

return Ok(new { Token = token });

}

return Unauthorized("Invalid login attempt.");

}

[HttpPost("logout")]

public async Task<IActionResult> Logout()

{

await \_signInManager.SignOutAsync();

return Ok("Logged out");

}

private string GenerateJwtToken(IdentityUser *user*, IList<string> *roles*)

{

var claims = new List<Claim>

{

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

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

};

// Add roles as claims

foreach (var role in roles)

{

claims.Add(new Claim(ClaimTypes.Role, role));

}

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(\_configuration["Jwt:Key"]));

var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var expires = DateTime.Now.AddHours(Convert.ToDouble(\_configuration["Jwt:ExpireHours"]));

var token = new JwtSecurityToken(

\_configuration["Jwt:Issuer"],

\_configuration["Jwt:Issuer"],

claims,

*expires*: expires,

*signingCredentials*: creds

);

return new JwtSecurityTokenHandler().WriteToken(token);

}

}

}

Now we will add RolesController.cs

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Identity;

using ShopSite.CW.WebApp.Models;

using Microsoft.AspNetCore.Authorization;

namespace ShopSite.CW.WebApp.Controllers

{

[Authorize(Roles ="Admin")]

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

[ApiController]

public class RolesController : ControllerBase

{

private readonly RoleManager<IdentityRole> \_roleManager;

private readonly UserManager<IdentityUser> \_userManager;

public RolesController(RoleManager<IdentityRole> *roleManager*, UserManager<IdentityUser> *userManager*)

{

\_roleManager = roleManager;

\_userManager = userManager;

}

[HttpGet]

public IActionResult GetRoles()

{

var roles = \_roleManager.Roles.ToList();

return Ok(roles);

}

[HttpGet("{roleId}")]

public async Task<IActionResult> GetRole(string *roleId*)

{

var role = await \_roleManager.FindByIdAsync(roleId);

if (role == null)

{

return NotFound("Role not found.");

}

return Ok(role);

}

[HttpPost]

public async Task<IActionResult> CreateRole([FromBody] string *roleName*)

{

var role = new IdentityRole(roleName);

var result = await \_roleManager.CreateAsync(role);

if (result.Succeeded)

{

return Ok("Role created successfully.");

}

return BadRequest(result.Errors);

}

[HttpPut]

public async Task<IActionResult> UpdateRole([FromBody] UpdateRoleModel *model*)

{

var role = await \_roleManager.FindByIdAsync(model.RoleId);

if (role == null)

{

return NotFound("Role not found.");

}

role.Name = model.NewRoleName;

var result = await \_roleManager.UpdateAsync(role);

if (result.Succeeded)

{

return Ok("Role updated successfully.");

}

return BadRequest(result.Errors);

}

[HttpDelete]

public async Task<IActionResult> DeleteRole(string *roleId*)

{

var role = await \_roleManager.FindByIdAsync(roleId);

if (role == null)

{

return NotFound("Role not found.");

}

var result = await \_roleManager.DeleteAsync(role);

if (result.Succeeded)

{

return Ok("Role deleted successfully.");

}

return BadRequest(result.Errors);

}

[HttpPost("assign-role-to-user")]

public async Task<IActionResult> AssignRoleToUser([FromBody] AssignRoleModel *model*)

{

var user = await \_userManager.FindByIdAsync(model.UserId);

if (user == null)

{

return NotFound("User not found.");

}

var roleExists = await \_roleManager.RoleExistsAsync(model.RoleName);

if (!roleExists)

{

return NotFound("Role not found.");

}

var result = await \_userManager.AddToRoleAsync(user, model.RoleName);

if (result.Succeeded)

{

return Ok("Role assigned to user successfully.");

}

return BadRequest(result.Errors);

}

}

}

then we will add these classes in the models:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace ShopSite.CW.WebApp.Models

{

public class AssignRoleModel

{

public string UserId { get; set; }

public string RoleName { get; set; }

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace ShopSite.CW.WebApp.Models

{

public class UpdateRoleModel

{

public string RoleId { get; set; }

public string NewRoleName { get; set; }

}

}