DTOS

namespace ReportClaim.Models.DTO

{

public class ClaimDTO

{

public string ClaimType { get; set; }=string.Empty;

}

}

namespace ReportClaim.Models.DTO

{

public class ErrorReponseDTO

{

public int ErrorCode { get; set; }

public string ErrorMessage { get; set; }

}

}

namespace ReportClaim.Models.DTO

{

public class LoginRequestDTO

{

public string Username { get; set; }

public string Password { get; set; }

}

}

namespace ReportClaim.Models.DTO

{

public class LoginResponseDTO

{

public string Username { get; set; }=string.Empty;

public string Token { get; set; } = string.Empty;

}

}

namespace ReportClaim.Models.DTO

{

public class PolicyDTO

{

public string PolicyNumber { get; set; }=string.Empty;

}

}

namespace ReportClaim.Models.DTO

{

public class ReportDTO

{

public int PolicyId { get; set; }

public int ClaimId { get; set; }

public DateTime IncidentDate { get; set; }

public string ClaimaintName { get; set; } = string.Empty;

public string ClaimaintPhone { get; set; } = string.Empty;

public string ClaimaintEmail { get; set; } = string.Empty;

public IFormFile? PhotoId { get; set; }

public IFormFile? SettlementForm { get; set; }

public IFormFile? DeathCertificate { get; set; }

public IFormFile? PolicyCertificate { get; set; }

public IFormFile? AddressProof { get; set; }

public IFormFile? CancelledCheck { get; set; }

public IFormFile? Others { get; set; }

}

}

namespace ReportClaim.Models.DTO

{

public class UserCreateDTO

{

public string Username { get; set; } = string.Empty;

public string Password { get; set; } = string.Empty;

public Roles Role { get; set; }

}

}

namespace ReportClaim.Models.DTO

{

public class UserTokenDTO

{

public string Username { get; set; } = string.Empty;

public string Role { get; set; } = string.Empty;

}

}

MODELS

namespace ReportClaim.Models

{

public class Claim

{

public int ClaimId { get; set; }

public string ClaimType { get; set; } = string.Empty;

public IEnumerable<Report> Reports { get; set; }

public Claim()

{

Reports=new List<Report>();

}

}

}

using System.ComponentModel.DataAnnotations;

using ReportClaim.Validation;

namespace ReportClaim.Models

{

public class Policy

{

public int Id { get; set; }

[PolicyNumberValidator]

public string PolicyNumber { get; set; } = string.Empty;

public IEnumerable<Report> Reports { get; set; }

public Policy()

{

Reports=new List<Report>();

}

}

}

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ReportClaim.Models

{

public class Report

{

public int Id { get; set; }

[Required(ErrorMessage = "IncidentDate can not be blank")]

public DateTime IncidentDate { get; set; }

[Required(ErrorMessage = "ClaimName can not be blank")]

public string ClaimaintName { get; set; } = string.Empty;

[Required(ErrorMessage = "ClaimPhone can not be blank")]

public string ClaimaintPhone { get; set; } = string.Empty;

[Required(ErrorMessage = "ClaimEmail can not be blank")]

public string ClaimaintEmail { get; set; } = string.Empty;

public string SettlementForm { get; set; } = string.Empty;

public string DeathCertificate { get; set; } = string.Empty;

public string PolicyCertificate { get; set; } = string.Empty;

public string PhotoId { get; set; } = string.Empty;

public string AddressProof { get; set; } = string.Empty;

public string CancelledCheck { get; set; } = string.Empty;

public string Others { get; set; } = string.Empty;

// Foreign Keys

public int ClaimId { get; set; } // Foreign key for Claim

public Claim Claim { get; set; } // Navigation property

public int PolicyId { get; set; } // Foreign key for Policy

public Policy Policy { get; set; } // Navigation property

public Report()

{

Policy = new Policy();

Claim = new Claim();

}

}

}

using System.ComponentModel.DataAnnotations;

namespace ReportClaim.Models

{

public enum Roles

{

Admin,

Claimaint

}

public class User

{

[Key]

public string Username { get; set; } = string.Empty;

public byte[] Password { get; set; }

public byte[] HashKey { get; set; }

public Roles Role { get; set; }

}

}

CONTROLLERS

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using ReportClaim.Interfaces;

using ReportClaim.Models.DTO;

using ReportClaim.Services;

namespace ReportClaim.Controllers

{

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

[ApiController]

public class ClaimController : ControllerBase

{

private readonly IClaimService \_claimService;

public ClaimController(IClaimService claimService)

{

\_claimService = claimService;

}

[HttpPost]

public async Task<ActionResult> CreateClaim(ClaimDTO claimDTO)

{

try

{

if (ModelState.IsValid)

{

var claim = await \_claimService.CreateClaim(claimDTO);

return Ok(claim);

}

else

{

return BadRequest(new ErrorReponseDTO

{

ErrorCode = 400,

ErrorMessage = "one or more fields validate error"

});

}

}

catch (Exception ex)

{

return StatusCode(500, ex);

}

}

}

}

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using ReportClaim.Interfaces;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Controllers

{

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

[ApiController]

public class PolicyController : ControllerBase

{

private readonly IPolicyService \_policyService;

public PolicyController(IPolicyService policyService)

{

\_policyService = policyService;

}

[HttpPost]

public async Task<ActionResult> InputPolicy(PolicyDTO policyDTO)

{

try

{

if (ModelState.IsValid)

{

var policy = await \_policyService.CreatePolicy(policyDTO);

return Ok(policy);

}

else

{

return BadRequest(new ErrorReponseDTO

{

ErrorCode = 400,

ErrorMessage = "one or more fields validate error"

});

}

}

catch (Exception ex)

{

return StatusCode(500, ex);

}

}

}

}

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using ReportClaim.Interfaces;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

using System.Net;

using System.Security.Claims;

namespace ReportClaim.Controllers

{

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

[ApiController]

public class ReportController : ControllerBase

{

private readonly IPolicyService \_policyService;

private readonly IClaimService \_claimService;

private readonly IReportService \_reportService;

public ReportController(IPolicyService policyService, IClaimService claimService, IReportService reportService)

{

\_policyService = policyService;

\_claimService = claimService;

\_reportService = reportService;

}

[HttpGet]

[Route("getPolicy")]

public async Task<ActionResult<IEnumerable<PolicyDTO>>> GetPolicies()

{

List<string> policyNumbers = new List<string>();

try

{

var policies = await \_policyService.GetAllPolicies();

foreach (var policy in policies) { policyNumbers.Add(policy.PolicyNumber); }

if (policies == null || !policies.Any())

{

return NotFound("No policies found.");

}

return Ok(policyNumbers);

}

catch (Exception ex)

{

return StatusCode((int)HttpStatusCode.InternalServerError, $"Error retrieving policies: {ex.Message}");

}

}

[HttpGet]

[Route("getClaim")]

public async Task<ActionResult<IEnumerable<ClaimDTO>>> GetClaims()

{

List<string> claimTypes = new List<string>();

try

{

var claims= await \_claimService.GetAllClaims();

foreach (var claimType in claims) { claimTypes.Add(claimType.ClaimType); }

if (claimTypes == null || !claimTypes.Any())

{

return NotFound("No claims found.");

}

return Ok(claimTypes);

}

catch (Exception ex)

{

return StatusCode((int)HttpStatusCode.InternalServerError, $"Error retrieving claims: {ex.Message}");

}

}

[HttpPost]

public async Task<ActionResult> CreateReport([FromForm] ReportDTO reportDTO)

{

try

{

if (ModelState.IsValid)

{

var report = await \_reportService.CreateReport(reportDTO);

return Ok(report.Id);

}

else

{

return BadRequest(new ErrorReponseDTO

{

ErrorCode = 400,

ErrorMessage = "one or more fields validate error"

});

}

}

catch (Exception ex)

{

return StatusCode(500,ex.Message);

}

}

}

}

using Microsoft.AspNetCore.Mvc;

using ReportClaim.Interfaces;

using ReportClaim.Models.DTO;

namespace ReportClaim.Controllers

{

public class UserController : ControllerBase

{

private readonly IUserService \_userService;

private readonly ILogger<UserController> \_logger;

public UserController(ILogger<UserController> logger, IUserService userService)

{

\_userService = userService;

\_logger = logger;

}

[HttpPost("Register")]

public async Task<ActionResult<LoginResponseDTO>> Register(UserCreateDTO createDTO)

{

try

{

if (ModelState.IsValid)

{

var user = await \_userService.Register(createDTO);

return Ok(user);

}

else

{

return BadRequest(new ErrorReponseDTO

{

ErrorMessage = "one or more validation errors",

ErrorCode = 400

});

}

}

catch (Exception e)

{

\_logger.LogError(e, "Could not register user");

return BadRequest(new ErrorReponseDTO

{

ErrorMessage = e.Message,

ErrorCode = 500

});

}

}

[HttpPost("Login")]

public async Task<ActionResult<LoginResponseDTO>> Login(LoginRequestDTO requestDTO)

{

try

{

var user = await \_userService.Authenticate(requestDTO);

return Ok(user);

}

catch (Exception e)

{

\_logger.LogError(e, "Invalid username or password");

return Unauthorized(new ErrorReponseDTO

{

ErrorMessage = e.Message,

ErrorCode = 401

});

}

}

}

}

CONTEXT

using Microsoft.EntityFrameworkCore;

using ReportClaim.Models;

namespace ReportClaim.Contexts

{

public class ReportClaimContext:DbContext

{

public ReportClaimContext(DbContextOptions options):base(options)

{

}

public DbSet<Policy> Policies { get; set; }

public DbSet<Claim> Claims { get; set; }

public DbSet<Report> Reports { get; set; }

public DbSet<User> Users { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//Report to Policy

modelBuilder.Entity<Report>()

.HasOne(p => p.Policy)

.WithMany(p => p.Reports)

.HasForeignKey(p => p.PolicyId)

.HasConstraintName("FK\_PolicyNumber");

//Report to ClaimType

modelBuilder.Entity<Report>()

.HasOne(c => c.Claim)

.WithMany(c => c.Reports)

.HasForeignKey(c => c.ClaimId)

.HasConstraintName("FK\_ClaimType");

}

}

}

INTERFACES

using ReportClaim.Models.DTO;

using ReportClaim.Models;

namespace ReportClaim.Interfaces

{

public interface IClaimService

{

public Task<Claim> CreateClaim(ClaimDTO claimDTO);

public Task<IEnumerable<Claim>> GetAllClaims();

}

}

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Interfaces

{

public interface IPolicyService

{

public Task<Policy> CreatePolicy(PolicyDTO policyDTO);

public Task<IEnumerable<Policy>> GetAllPolicies();

}

}

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Interfaces

{

public interface IReportService

{

public Task<Report> CreateReport(ReportDTO reportDTO);

public Task<IEnumerable<Report>> GetAllReports();

}

}

namespace ReportClaim.Interfaces

{

public interface IRepository<T, K> where T : class

{

public Task<T> Create(T entity);

public Task<T> Update(T entity);

public Task<T> Delete(K key);

public Task<T> GetById(K id);

public Task<IEnumerable<T>> GetAll();

}

}

using ReportClaim.Models.DTO;

namespace ReportClaim.Interfaces

{

public interface ITokenService

{

public Task<string> GenerateToken(UserTokenDTO user);

}

}

using ReportClaim.Models.DTO;

namespace ReportClaim.Interfaces

{

public interface IUserService

{

public Task<LoginResponseDTO> Authenticate(LoginRequestDTO loginRequestDTO);

public Task<LoginResponseDTO> Register(UserCreateDTO registerDTO);

}

}

SERVICES

using AutoMapper;

using ReportClaim.Interfaces;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

using ReportClaim.Repositories;

namespace ReportClaim.Services

{

public class ClaimService : IClaimService

{

private readonly IRepository<Claim, int> \_claimRepository;

private readonly IMapper \_mapper;

public ClaimService(IRepository<Claim, int> repository, IMapper mapper)

{

\_claimRepository = repository;

\_mapper = mapper;

}

public async Task<Claim> CreateClaim(ClaimDTO claimDTO)

{

try

{

Claim claim = \_mapper.Map<Claim>(claimDTO);

var result=await \_claimRepository.Create(claim);

return result;

}

catch

{

throw new Exception("Cannot add claim");

}

}

public async Task<IEnumerable<Claim>> GetAllClaims()

{

try

{

var claims = await \_claimRepository.GetAll();

if (claims.Count() <= 0)

throw new Exception("Cannot provide all policies");

return claims;

}

catch (Exception ex)

{

throw new Exception(ex.Message);

}

}

}

}

using AutoMapper;

using Microsoft.AspNetCore.Server.IIS.Core;

using ReportClaim.Interfaces;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Services

{

public class PolicyService : IPolicyService

{

private readonly IRepository<Policy,int> \_policyRepository;

private readonly IMapper \_mapper;

public PolicyService(IRepository<Policy,int> repository,IMapper mapper )

{

\_policyRepository = repository;

\_mapper = mapper;

}

public async Task<Policy> CreatePolicy(PolicyDTO policyDTO)

{

try

{

var policy = \_mapper.Map<Policy>(policyDTO);

var result=await \_policyRepository.Create(policy);

return result;

}

catch

{

throw new Exception("Cannot add policy");

}

}

public async Task<IEnumerable<Policy>> GetAllPolicies()

{

try

{

var policies = await \_policyRepository.GetAll();

if (policies.Count() <= 0)

throw new Exception("Cannot provide all policies");

return policies;

}

catch (Exception ex) {

throw new Exception(ex.Message);

}

}

}

}

using AutoMapper;

using ReportClaim.Interfaces;

using ReportClaim.Models.DTO;

using ReportClaim.Models;

public class ReportService : IReportService

{

private readonly IRepository<Report, int> \_repository;

private readonly IMapper \_mapper;

private readonly IWebHostEnvironment \_environment; // To get the web root path for file saving

public ReportService(IRepository<Report, int> repository, IMapper mapper, IWebHostEnvironment environment)

{

\_repository = repository;

\_mapper = mapper;

\_environment = environment;

}

public async Task<Report> CreateReport(ReportDTO entity)

{

try

{

var report = new Report

{

PolicyId = entity.PolicyId,

ClaimId = entity.ClaimId,

ClaimaintName = entity.ClaimaintName,

ClaimaintPhone = entity.ClaimaintPhone,

ClaimaintEmail = entity.ClaimaintEmail,

IncidentDate = entity.IncidentDate

};

// Handle file uploads and conversions

report.PhotoId = await SaveFileAndGetBase64(entity.PhotoId, "PhotoId");

report.SettlementForm = await SaveFileAndGetBase64(entity.SettlementForm, "SettlementForm");

report.DeathCertificate = await SaveFileAndGetBase64(entity.DeathCertificate, "DeathCertificate");

report.PolicyCertificate = await SaveFileAndGetBase64(entity.PolicyCertificate, "PolicyCertificate");

report.AddressProof = await SaveFileAndGetBase64(entity.AddressProof, "AddressProof");

report.CancelledCheck = await SaveFileAndGetBase64(entity.CancelledCheck, "CancelledCheck");

report.Others = await SaveFileAndGetBase64(entity.Others, "Others");

var newReport = await \_repository.Create(report);

return newReport;

}

catch

{

throw new Exception("Cannot create report");

}

}

private async Task<string> SaveFileAndGetBase64(IFormFile file, string fileNamePrefix)

{

if (file == null || file.Length == 0)

return string.Empty;

var uniqueFileName = $"{fileNamePrefix}\_{DateTime.Now.Ticks}{Path.GetExtension(file.FileName)}";

var uploadsFolderPath = Path.Combine(\_environment.WebRootPath, "uploads");

if (!Directory.Exists(uploadsFolderPath))

{

Directory.CreateDirectory(uploadsFolderPath);

}

var filePath = Path.Combine(uploadsFolderPath, uniqueFileName);

// Save the file to the folder

using (var fileStream = new FileStream(filePath, FileMode.Create))

{

await file.CopyToAsync(fileStream);

}

using (var memoryStream = new MemoryStream())

{

await file.CopyToAsync(memoryStream);

var fileBytes = memoryStream.ToArray();

return Convert.ToBase64String(fileBytes);

}

}

public async Task<IEnumerable<Report>> GetAllReports()

{

try

{

var reports = await \_repository.GetAll();

if (reports.Count() <= 0)

throw new Exception("Cannot provide all policies");

return reports;

}

catch

{

throw new Exception("Cannot get all reports");

}

}

}

using System.IdentityModel.Tokens.Jwt;

using System.Security.Claims;

using System.Text;

using Microsoft.AspNetCore.Mvc;

using Microsoft.IdentityModel.Tokens;

using ReportClaim.Interfaces;

using ReportClaim.Models.DTO;

namespace ReportClaim.Services

{

public class TokenService : ITokenService

{

private readonly string secretKey;

public TokenService(IConfiguration configuration)

{

secretKey = configuration["JWT:SecretKey"];

}

public async virtual Task<string> GenerateToken(UserTokenDTO user)

{

string \_token = string.Empty;

var \_claims = new[]

{

new Claim(JwtRegisteredClaimNames.GivenName, user.Username),

new Claim("Role", user.Role),

};

var \_key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey));

var \_credentials = new SigningCredentials(\_key, SecurityAlgorithms.HmacSha256);

var \_tokenDescriptor = new JwtSecurityToken(

claims: \_claims,

expires: DateTime.Now.AddDays(1),

signingCredentials: \_credentials

);

\_token = new JwtSecurityTokenHandler().WriteToken(\_tokenDescriptor);

return \_token;

}

}

}

using System.Security.Cryptography;

using System.Text;

using ReportClaim.Interfaces;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

using ReportClaim.Repositories;

namespace ReportClaim.Services

{

public class UserService : IUserService

{

private readonly IRepository<User, string> \_userRepository;

private readonly ILogger \_logger;

private readonly ITokenService \_tokenService;

public UserService(IRepository<User,string> repository, ILogger logger, ITokenService tokenService) {

\_userRepository = repository;

\_logger = logger;

\_tokenService = tokenService;

}

public async Task<LoginResponseDTO> Authenticate(LoginRequestDTO loginUser)

{

var user = await \_userRepository.GetById(loginUser.Username);

if (user == null)

{

throw new Exception("User not found");

}

HMACSHA256 hmac = new HMACSHA256(user.HashKey);

byte[] passwordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(loginUser.Password));

for (int i = 0; i < passwordHash.Length; i++)

{

if (passwordHash[i] != user.Password[i])

{

throw new Exception("Invalid username or password");

}

}

return new LoginResponseDTO()

{

Username = user.Username,

Token = await \_tokenService.GenerateToken(new UserTokenDTO()

{

Username = user.Username,

Role = user.Role.ToString()

})

};

}

public async Task<LoginResponseDTO> Register(UserCreateDTO registerUser)

{

HMACSHA256 hmac = new HMACSHA256();

byte[] passwordHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(registerUser.Password));

User user = new User()

{

Username = registerUser.Username,

Password = passwordHash,

HashKey = hmac.Key,

Role = registerUser.Role

};

try

{

var addesUser = await \_userRepository.Create(user);

LoginResponseDTO response = new LoginResponseDTO()

{

Username = user.Username

};

return response;

}

catch (Exception e)

{

\_logger.LogError(e, "Could not register user");

throw new Exception("Could not register user");

}

}

}

}

MAPPERS

using AutoMapper;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Mapper

{

public class ClaimProfile:Profile

{

public ClaimProfile()

{

CreateMap<ClaimDTO, Claim>();

CreateMap<Claim,ClaimDTO>();

}

}

}

using AutoMapper;

using ReportClaim.Models;

using ReportClaim.Models.DTO;

namespace ReportClaim.Mapper

{

public class PolicyProfile:Profile

{

public PolicyProfile()

{

CreateMap<PolicyDTO, Policy>();

CreateMap<Policy, PolicyDTO>();

}

}

}

REPOSITORIES

using Microsoft.EntityFrameworkCore;

using ReportClaim.Contexts;

using ReportClaim.Exceptions;

using ReportClaim.Interfaces;

using ReportClaim.Models;

namespace ReportClaim.Repositories

{

public class ClaimRepository : IRepository<Claim, int>

{

private readonly ReportClaimContext \_context;

public ClaimRepository(ReportClaimContext reportClaimContext)

{

\_context = reportClaimContext;

}

public async Task<Claim> Create(Claim entity)

{

try

{

\_context.Claims.Add(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch

{

throw new CannotCreateException("Claim");

}

}

public async Task<Claim> Delete(int key)

{

var entity = await GetById(key);

try

{

\_context.Claims.Remove(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch {

throw new Exception("Claim does not exsists");

}

}

public async Task<IEnumerable<Claim>> GetAll()

{

var claims = await \_context.Claims.ToListAsync();

return claims;

}

public async Task<Claim> GetById(int id)

{

var claim = await \_context.Claims.FirstOrDefaultAsync(c => c.ClaimId == id);

if (claim == null)

{

throw new CannotFindException("Claim");

}

return claim;

}

public async Task<Claim> Update(Claim entity)

{

try

{

var oldClaim = await GetById(entity.ClaimId);

oldClaim.ClaimType = entity.ClaimType;

await \_context.SaveChangesAsync();

return oldClaim;

}

catch

{

throw new CannotUpdateException("Claim");

}

}

}

}

using System.Security.Claims;

using Microsoft.EntityFrameworkCore;

using ReportClaim.Contexts;

using ReportClaim.Exceptions;

using ReportClaim.Interfaces;

using ReportClaim.Models;

namespace ReportClaim.Repositories

{

public class PolicyRepository : IRepository<Policy, int>

{

private readonly ReportClaimContext \_context;

public PolicyRepository(ReportClaimContext reportClaimContext)

{

\_context = reportClaimContext;

}

public async Task<Policy> Create(Policy entity)

{

try

{

\_context.Policies.Add(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch

{

throw new CannotCreateException("Policy");

}

}

public async Task<Policy> Delete(int key)

{

var entity= await GetById(key);

try

{

\_context.Policies.Remove(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch {

throw new Exception("Cannot delete");

}

}

public async Task<IEnumerable<Policy>> GetAll()

{

var policies= await \_context.Policies.ToListAsync();

return policies;

}

public async Task<Policy> GetById(int id)

{

var policy = await \_context.Policies.FirstOrDefaultAsync(policy => policy.Id == id);

if (policy == null)

{

throw new CannotFindException("Policy");

}

return policy;

}

public async Task<Policy> Update(Policy entity)

{

try

{

var oldPolicy=await GetById(entity.Id);

oldPolicy.PolicyNumber = entity.PolicyNumber;

await \_context.SaveChangesAsync();

return oldPolicy;

}

catch {

throw new CannotUpdateException("Policy");

}

}

}

}

using Microsoft.EntityFrameworkCore;

using ReportClaim.Contexts;

using ReportClaim.Exceptions;

using ReportClaim.Interfaces;

using ReportClaim.Models;

namespace ReportClaim.Repositories

{

public class ReportRepository : IRepository<Report, int>

{

private readonly ReportClaimContext reportClaim;

public ReportRepository(ReportClaimContext reportClaimContext)

{

reportClaim = reportClaimContext;

}

public async Task<Report> Create(Report entity)

{

try

{

reportClaim.Reports.Add(entity);

await reportClaim.SaveChangesAsync();

return entity;

}

catch

{

throw new CannotCreateException("Report");

}

}

public async Task<Report> Delete(int key)

{

var entity=await GetById(key);

try

{

reportClaim.Reports.Remove(entity);

await reportClaim.SaveChangesAsync();

return entity;

}

catch

{

throw new Exception("Cannot delete");

}

}

public async Task<IEnumerable<Report>> GetAll()

{

var reports = await reportClaim.Reports.ToListAsync();

return reports;

}

public async Task<Report> GetById(int id)

{

var report = await reportClaim.Reports.FirstOrDefaultAsync(policy => policy.Id == id);

if (report == null)

{

throw new CannotFindException("Report");

}

return report;

}

public async Task<Report> Update(Report entity)

{

try

{

var oldReport = await GetById(entity.Id);

oldReport.ClaimaintName = entity.ClaimaintName;

oldReport.ClaimaintPhone = entity.ClaimaintPhone;

oldReport.CancelledCheck = entity.CancelledCheck;

oldReport.AddressProof = entity.AddressProof;

oldReport.DeathCertificate = entity.DeathCertificate;

oldReport.PolicyCertificate = entity.PolicyCertificate;

oldReport.SettlementForm = entity.SettlementForm;

await reportClaim.SaveChangesAsync();

return oldReport;

}

catch

{

throw new CannotUpdateException("Policy");

}

}

}

}

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata.Internal;

using ReportClaim.Contexts;

using ReportClaim.Exceptions;

using ReportClaim.Interfaces;

using ReportClaim.Models;

namespace ReportClaim.Repositories

{

public class UserRepository : IRepository<User, string>

{

ReportClaimContext \_context;

ILogger<UserRepository> \_logger;

public UserRepository(ReportClaimContext reportClaimContext, ILogger<UserRepository> logger)

{

\_context = reportClaimContext;

\_logger = logger;

}

public async Task<User> Create(User entity)

{

try

{

\_context.Users.Add(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch (Exception e)

{

\_logger.LogError(e, "Could not add user");

throw new CouldNotAddException("User");

}

}

public async Task<User> Delete(string key)

{

var user = await GetById(key);

try

{

\_context.Users.Remove(user);

await \_context.SaveChangesAsync();

return user;

}

catch (Exception e)

{

\_logger.LogError(e, "Could not delete user");

throw new Exception("Unable to delete");

}

}

public async Task<IEnumerable<User>> GetAll()

{

var users = await \_context.Users.ToListAsync();

if (users.Count == 0)

{

throw new CollectionEmptyException("Users");

}

return users;

}

public async Task<User> GetById(string id)

{

try

{

var user = await \_context.Users.FirstOrDefaultAsync(u => u.Username == id);

return user;

}

catch (Exception e)

{

\_logger.LogError(e, "Could not get user");

throw new CannotFindException("User");

}

}

public async Task<User> Update(User entity)

{

var user = await GetById(entity.Username);

if (user == null)

{

throw new CannotFindException("User");

}

try

{

\_context.Users.Update(entity);

await \_context.SaveChangesAsync();

return entity;

}

catch (Exception e)

{

\_logger.LogError(e, "Could not update user details");

throw new Exception("Unable to modify user object");

}

}

}

}

MISC

using System.ComponentModel.DataAnnotations;

namespace ReportClaim.Validation

{

public class PolicyNumberValidator:ValidationAttribute

{

protected override ValidationResult IsValid(object value, ValidationContext validationContext)

{

if (value == null)

{

return new ValidationResult("Policy Number cannot be empty");

}

if (value.ToString().Length < 6)

{

return new ValidationResult("Policy Number should be length of 6");

}

return ValidationResult.Success;

}

}

}

using Microsoft.AspNetCore.Mvc.Filters;

using Microsoft.AspNetCore.Mvc;

using ReportClaim.Models.DTO;

namespace ReportClaim.Misc

{

public class ReportExceptionFilter : ExceptionFilterAttribute

{

public override void OnException(ExceptionContext context)

{

context.Result = new BadRequestObjectResult(new ErrorReponseDTO

{

ErrorCode = 500,

ErrorMessage = context.Exception.Message

});

}

}

}

EXCEPTIONS

using System.Runtime.Serialization;

namespace ReportClaim.Exceptions

{

[Serializable]

public class CannotCreateException : Exception

{

string mssg;

public CannotCreateException(string mssg)

{

mssg = $"Cannot create {mssg}";

}

public override string Message => mssg;

}

}

using System.Runtime.Serialization;

namespace ReportClaim.Exceptions

{

[Serializable]

public class CannotFindException : Exception

{

string mssg;

public CannotFindException(string str)

{

mssg = $"Cannot find {str}";

}

public override string Message => mssg;

}

}

using System.Runtime.Serialization;

namespace ReportClaim.Exceptions

{

[Serializable]

public class CannotUpdateException : Exception

{

string mssg;

public CannotUpdateException(string str)

{

mssg = "CannotUpdate"+str;

}

override public string Message=>mssg;

}

}

using System.Runtime.Serialization;

namespace ReportClaim.Exceptions

{

class CollectionEmptyException : Exception

{

string mssg = string.Empty;

public CollectionEmptyException()

{

mssg = "Collection is empty";

}

public override string Message => mssg;

}

}

PROGRAM. CS

using System.Text;

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Http.Features;

using Microsoft.EntityFrameworkCore;

using Microsoft.IdentityModel.Tokens;

using ReportClaim.Contexts;

using ReportClaim.Interfaces;

using ReportClaim.Misc;

using ReportClaim.Models;

using ReportClaim.Repositories;

using ReportClaim.Services;

namespace ReportClaim

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

#region Context

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

{

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));

});

#endregion

#region Filters

builder.Services.AddScoped<ReportExceptionFilter>();

#endregion

#region Authentication

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = false,

ValidateAudience = false,

ValidateLifetime = true,

ValidateIssuerSigningKey = true,

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

};

options.Events = new JwtBearerEvents

{

OnMessageReceived = context =>

{

// Check if the Authorization header is present

var authorizationHeader = context.Request.Headers["Authorization"].ToString();

if (string.IsNullOrEmpty(authorizationHeader))

{

Console.WriteLine("No Authorization header found.");

}

else if (!authorizationHeader.StartsWith("Bearer "))

{

Console.WriteLine("Authorization header is not a Bearer token.");

}

else

{

// Token is present and valid

Console.WriteLine("Authorization header found: " + authorizationHeader);

}

return Task.CompletedTask;

},

OnAuthenticationFailed = context =>

{

Console.WriteLine("Authentication failed: " + context.Exception.Message);

return Task.CompletedTask;

},

OnTokenValidated = context =>

{

Console.WriteLine("Token validated.");

return Task.CompletedTask;

},

OnChallenge = context =>

{

context.Response.Headers.Add("WWW-Authenticate", "Bearer");

context.Response.StatusCode = 401; // Unauthorized

return Task.CompletedTask;

}

};

});

#endregion

#region Repositories

builder.Services.AddScoped<IRepository<Policy, int>, PolicyRepository>();

builder.Services.AddScoped<IRepository<Claim, int>, ClaimRepository>();

builder.Services.AddScoped<IRepository<Report, int>, ReportRepository>();

#endregion

#region Services

builder.Services.AddScoped<IPolicyService, PolicyService>();

builder.Services.AddScoped<IClaimService, ClaimService>();

builder.Services.AddScoped<IReportService, ReportService>();

#endregion

#region Mapper

builder.Services.AddAutoMapper(typeof(Policy));

builder.Services.AddAutoMapper(typeof(Claim));

builder.Services.AddAutoMapper(typeof(Report));

#endregion

// Add services to the container.

builder.Services.AddControllers();

// Enable file uploads with specific limits (optional)

builder.Services.Configure<FormOptions>(options =>

{

options.MultipartBodyLengthLimit = 10 \* 1024 \* 1024; // Limit file size to 10 MB (adjust as needed)

});

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseHttpsRedirection();

// Serve static files from wwwroot

app.UseStaticFiles();

app.UseAuthorization();

app.MapControllers();

app.Run();

}

}

}