# Project

* Site-14-Api-Ng-messaging-queryparams-routeresolvers
  + Copied from: Site-13-Api-Ng-like
* For the “MySocialConnect-API”
  + dotnet restore
  + dotnet build
  + Go to project : MSC.WebApi
    - dotnet build : to build
    - dotnet run : to run the api

# New Resources

|  |  |
| --- | --- |
| MSC.Core | MSC.WebApi |
| MSC.Core/DB/Entities/UserMessage.cs | Controllers/MessageController.cs |
| MSC.Core/dtos/MessageCreateDto |  |
| MSC.Core/dtos/MessageDto.cs |  |
| MSC.Core/enums/zMessageType.cs |  |
| MSC.Core/Dtos/MessageSearchParamDto.cs |  |
| MSC.Core/Repositories/MessageRepository |  |
| MSC.Core/BusinessLogic/MessageBusinessLogic |  |
|  |  |
|  |  |

# Resources updated

|  |  |
| --- | --- |
| MSC.Core | MSC.WebApi |
| MSC.Core/DB/Entities/AppUser.cs |  |
| MSC.Core/DB/Data/DataContext.cs |  |
| MSC.Core/Mappers/AutoMapperProfile.cs |  |
| MSC.Core/Extensions/AppServiceExtensions.cs |  |
| MSC.Core.Dtos/BusinessResponse.cs |  |
|  |  |

# Clear Database

Clear the users

dotnet ef database drop

dotnet ef database update

New migration add in this project [here](#_Migrations_and_database).

# Intro

* We are setting up a Messages table that will have properties to hold the messages between two users.
* This is many to many relationship

# Database Update

## MSC.Core/DB/Entities

### UserMessage.cs

using System;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using Microsoft.EntityFrameworkCore;

using MSC.Core.DB.Entities;

namespace MSC.Core.Entities;

//table name comes from datacontext as "Messages"

//primary key will be cretaed as part of the convention

//check db context for more details

[Index(nameof(Guid))]

public class UserMessage

{

    [Column(Order = 1)]

    public int Id { get; set; }

    [Column(Order = 2)]

    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]

    [Required]

    public Guid Guid { get; set; } = Guid.NewGuid();

    [Column(Order = 3)]

    [Required]

    public int SenderId { get; set; }

    [Column(Order = 4)]

    [Required]

    public string SenderUserName { get; set; }

    public AppUser Sender { get; set; }

    [Column(Order = 5)]

    public bool SenderDeleted { get; set; } = false;

    [Column(Order = 6)]

    [Required]

    public int RecipientId { get; set; }

    [Column(Order = 7)]

    [Required]

    public string RecipientUserName { get; set; }

    public AppUser Recipient { get; set; }

    [Column(Order = 8)]

    public bool RecipientDeleted { get; set; } = false;

    [Column(Order = 9)]

    public string MessageContent { get; set; }

     [Column(Order = 10)]

    public DateTime DateMessageSent { get; set; } = DateTime.UtcNow;

    [Column(Order = 11)]

    public DateTime? DateMessageRead { get; set; }

}

### AppUser.cs

    //for messages, check DB Context

    public List<UserMessage> MessagesSent { get; set; }

    public List<UserMessage> MessagesReceived { get; set; }

## MSC.Core/DB/Data

### DataContext.cs

Add the data set

    //table name will be created as "Messages"

    //creating the db set so that we can directly query

    public DbSet<UserMessage> Messages {get; set;}

Add the configuration

    protected override void OnModelCreating(ModelBuilder builder)

    {

        base.OnModelCreating(builder);

        UserLikeSetup(builder);

        UserMessageSetup(builder);

    }

    private void UserLikeSetup(ModelBuilder builder)

    {

        //key is combination of SourceUserId and TargetUserId

        builder.Entity<UserLike>()

                .HasKey(k => new {k.SourceUserId, k.TargetUserId});

        //build relationships between AppUser and UserLike. Here the users liked by the logged in user

        builder.Entity<UserLike>()

            .HasOne(s => s.SourceUser) //UserLike field

            .WithMany(l => l.LikedUsers) //AppUser field

            .HasForeignKey(s => s.SourceUserId)

            //for sql server the same entity cannot have to cascades so one needs to DeleteBehavior.NoAction

            .OnDelete(DeleteBehavior.Cascade) //when the user is deleted then delete the related entities.

        ;

        //build relationships between AppUser and UserLike. Here the logged in user liked by other users

        builder.Entity<UserLike>()

            .HasOne(t => t.TargetUser) //UserLike field

            .WithMany(l => l.LikedByUsers) //AppUser field

            .HasForeignKey(t => t.TargetUserId)

            //for sql server the same entity cannot have to cascades so one needs to DeleteBehavior.NoAction

            .OnDelete(DeleteBehavior.Cascade) //when the user is deleted then delete the related entities

        ;

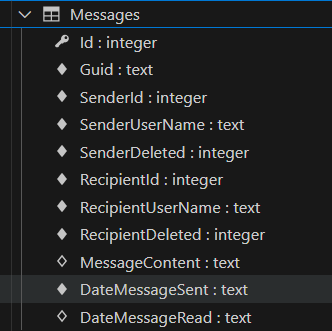
    }

## Migrations and database Update

> dotnet ef migrations add MessageEntityAdded

To undo above action, use 'ef migrations remove'

> dotnet ef database update



> dotnet watch --no-hot-reload

# MSC.Core/Enums

## zMessageType.cs

namespace MSC.Core.Enums;

public enum zMessageType

{

    Inbox, //received

    InboxUnread, //received not read

    Outbox, //send

}

# MSC.Core/Dtos

## MessageCreateDto.cs

namespace MSC.Core.Dtos;

public class MessageCreateDto

{

    public int RecipientId { get; set; }

    public string MessageContent { get; set; }

}

## MessageDto.cs

using System;

namespace MSC.Core.Dtos;

public class MessageDto

{

    public int Id { get; set; }

    public Guid Guid { get; set; }

    public int SenderId { get; set; }

    public Guid SenderGuid { get; set; }

    public string SenderUsername { get; set; }

    public string SenderPhotoUrl { get; set; }

    public int RecipientId { get; set; }

    public Guid RecipientGuid { get; set; }

    public string RecipientUsername { get; set; }

    public string RecipientPhotoUrl { get; set; }

    public string MessageContent { get; set; }

    public DateTime? DateMessageRead { get; set; }

    public DateTime DateMessageSent { get; set; }

}

## MessageSearchParamDto.cs

using System;

using MSC.Core.Dtos.Pagination;

using MSC.Core.Enums;

namespace MSC.Core.Dtos;

public class MessageSearchParamDto: PaginationParams

{

    public int UserId {get; set;}

    public zMessageType MessageType {get; set;} = zMessageType.InboxUnread;

}

## BusinessResponse.cs

using System.Net;

namespace MSC.Core.Dtos;

public class BusinessResponse

{

    public BusinessResponse()

    {

    }

    public BusinessResponse(HttpStatusCode httpStatusCode, string message = "", object data = null)

    {

        HttpStatusCode = httpStatusCode;

        Message = message;

        Data = data;

    }

    public HttpStatusCode HttpStatusCode { get; set; }

    public string Message { get; set; }

    public object Data {get; set;}

    public T ConvertDataToType<T>()

    {

        if(Data == null) return default(T);

        var newData = (T)Data;

        return newData;

    }

}

# MSC.Core/Mappers

## AutoMapperProfiles.cs

    public AutoMapperProfiles()

    {

        Map\_AppUser\_To\_UserDto();

        Map\_Photo\_To\_PhotoDto();

        Map\_AppUser\_To\_LoggedInUserDto();

        Map\_UserRegisterDto\_To\_AppUser();

        Map\_MemberUpdateDto\_To\_AppUser();

        Map\_Message\_to\_MessageDto();

    }

    private void Map\_Message\_to\_MessageDto()

    {

        CreateMap<UserMessage, MessageDto>()

        .ForMember(dest => dest.SenderPhotoUrl, opt => opt.MapFrom(src => PickMainUrl(src.Sender.Photos)))

        .ForMember(dest => dest.SenderGuid, opt => opt.MapFrom(src => src.Sender.Guid))

        .ForMember(dest => dest.RecipientPhotoUrl, opt => opt.MapFrom(src => PickMainUrl(src.Recipient.Photos)))

        .ForMember(dest => dest.RecipientGuid, opt => opt.MapFrom(src => src.Recipient.Guid))

        ;

    }

# MSC.Core/Repositories

## IMessageRepository.cs

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using MSC.Core.DB.Entities;

using MSC.Core.Dtos;

using MSC.Core.Dtos.Pagination;

namespace MSC.Core.Repositories;

public interface IMessageRepository

{

     void AddMessage(UserMessage message);

    void DeleteMessage(UserMessage message);

    Task<UserMessage> GetMessage(int id);

    Task<UserMessage> GetMessage(Guid guid);

    Task<PagedList<MessageDto>> GetMessagesForUser(MessageSearchParamDto search);

    Task<IEnumerable<UserMessage>> GetMessageThread(int currentUserId, int receipientId);

    Task<bool> SaveAllSync();

}

## MessageRepository.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using AutoMapper;

using AutoMapper.QueryableExtensions;

using Microsoft.EntityFrameworkCore;

using MSC.Core.DB.Data;

using MSC.Core.DB.Entities;

using MSC.Core.Dtos;

using MSC.Core.Dtos.Pagination;

using MSC.Core.Enums;

namespace MSC.Core.Repositories;

public class MessageRepository : IMessageRepository

{

    private readonly DataContext \_context;

    private readonly IMapper \_mapper;

    public MessageRepository(DataContext context, IMapper mapper)

    {

        \_context = context;

        \_mapper = mapper;

    }

    public void AddMessage(UserMessage message)

    {

        \_context.Messages.Add(message);

    }

    public void DeleteMessage(UserMessage message)

    {

       \_context.Messages.Remove(message);

    }

    public async Task<UserMessage> GetMessage(int id)

    {

        //we can do projectto to fill the entities or like following

        var message = await \_context.Messages

                            .Include(u => u.Recipient)

                            .Include(u => u.Sender)

                            .SingleOrDefaultAsync(x => x.Id == id);

        return message;

    }

    public async Task<UserMessage> GetMessage(Guid guid)

    {

        //we can do projectto to fill the entities or like following

        var message = await \_context.Messages

                            .Include(u => u.Recipient)

                            .Include(u => u.Sender)

                            .SingleOrDefaultAsync(x => x.Guid == guid);

        return message;

    }

    public async Task<PagedList<MessageDto>> GetMessagesForUser(MessageSearchParamDto search)

    {

        var query = \_context.Messages.OrderByDescending(m => m.DateMessageSent).AsQueryable();

        query = search.MessageType switch

        {

            //recipient of the message

            zMessageType.Inbox => query.Where(u => u.RecipientId == search.UserId && !u.RecipientDeleted),

            //recipient of the message and not read it

            zMessageType.InboxUnread => query.Where(u => u.RecipientId == search.UserId && u.DateMessageRead == null && !u.RecipientDeleted),

            //default sender outbox

            \_ => query.Where(u => u.Sender.Id == search.UserId && !u.SenderDeleted)

        };

        var messages = query.ProjectTo<MessageDto>(\_mapper.ConfigurationProvider);

        var pageList = await PagedList<MessageDto>.CreateAsync(messages, search.PageNumber, search.PageSize);

        return pageList;

    }

    //message thread between two users so check for both ways. Also eagily load photos for both receipent and sender

    public async Task<IEnumerable<UserMessage>> GetMessageThread(int currentUserId, int receipientId)

    {

        var messages = await \_context.Messages

                            .Include(u => u.Recipient).ThenInclude(p => p.Photos)

                            .Include(u => u.Sender).ThenInclude(p => p.Photos)

                            .Where(m =>

                                (m.RecipientId == currentUserId && m.SenderId == receipientId && !m.RecipientDeleted) ||

                                (m.RecipientId == receipientId && m.SenderId == currentUserId && !m.SenderDeleted)

                            )

                            .OrderBy(m => m.DateMessageSent)

                            .ToListAsync();

        var unreadMessages = messages.Where(m => m.DateMessageRead == null && m.Recipient.Id == currentUserId).ToList();

        if(unreadMessages != null && unreadMessages.Any())

        {

            //update the date

            unreadMessages.ForEach(x => {x.DateMessageRead = DateTime.UtcNow;});

            await \_context.SaveChangesAsync();

        }

        return messages;

    }

    public async Task<bool> SaveAllSync()

    {

        return await \_context.SaveChangesAsync() > 0;

    }

}

## MSC.Core/Extensions

### AppServiceExtensions.cs

Add to the services

    public static IServiceCollection AddServices(this IServiceCollection services, IConfiguration config)

    {

        services.AddScoped<IUserRepository, UserRepository>();

        services.AddScoped<IUserBusinessLogic, UserBusinessLogic>();

        services.AddScoped<ILikesRepository, LikesRepository>();

        services.AddScoped<ILikesBusinessLogic, LikesBusinessLogic>();

        services.AddScoped<IMessageRepository, MessageRepository>();

# MSC.Core/BusinessLogic

## IMessageBusinessLogic.cs

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using MSC.Core.DB.Entities;

using MSC.Core.Dtos;

using MSC.Core.Dtos.Pagination;

namespace MSC.Core.BusinessLogic;

public interface IMessageBusinessLogic

{

    Task<BusinessResponse> AddMessage(MessageCreateDto msg, int senderId);

    Task<BusinessResponse> DeleteMessage(int currentUserId, Guid msgGuid);

    void DeleteMessage(UserMessage message);

    Task<UserMessage> GetMessage(int id);

    Task<UserMessage> GetMessage(Guid guid);

    Task<PagedList<MessageDto>> GetMessagesForUser(MessageSearchParamDto search);

    Task<IEnumerable<MessageDto>> GetMessageThread(int currentUserId, int receipientId);

}

## MessageBusinessLogc.cs

using System;

using System.Collections.Generic;

using System.Net;

using System.Threading.Tasks;

using AutoMapper;

using MSC.Core.DB.Entities;

using MSC.Core.Dtos;

using MSC.Core.Dtos.Pagination;

using MSC.Core.Repositories;

namespace MSC.Core.BusinessLogic;

public class MessageBusinessLogic : IMessageBusinessLogic

{

    private readonly IMessageRepository \_msgRepo;

    private readonly IUserRepository \_userRepo;

    private readonly IMapper \_mapper;

    public MessageBusinessLogic(IMessageRepository msgRepo, IUserRepository userRepo, IMapper mapper)

    {

        \_msgRepo = msgRepo;

        \_userRepo = userRepo;

        \_mapper = mapper;

    }

    public async Task<BusinessResponse> AddMessage(MessageCreateDto msg, int senderId)

    {

        if(msg == null || msg.RecipientId <= 0 || string.IsNullOrWhiteSpace(msg.MessageContent))

            return new BusinessResponse(HttpStatusCode.BadRequest, "Message not good");

        //get source user

        var sender = await \_userRepo.GetUserAsync(senderId, includePhotos: true);

        if(sender == null)

            return new BusinessResponse(HttpStatusCode.BadRequest, "Logged in user not found");

        if(sender.Id == msg.RecipientId)

            return new BusinessResponse(HttpStatusCode.BadRequest, "You cannot send message to yourself");

        var recipient = await \_userRepo.GetUserAsync(msg.RecipientId, includePhotos: true);

        if(sender == null)

            return new BusinessResponse(HttpStatusCode.BadRequest, "Recipient not found");

        var message = new UserMessage{

            Sender = sender,

            SenderUserName = sender.UserName,

            Recipient = recipient,

            RecipientUserName = recipient.UserName,

            MessageContent = msg.MessageContent

        };

        \_msgRepo.AddMessage(message);

        if(await \_msgRepo.SaveAllSync())

        {

            var msgDto = \_mapper.Map<MessageDto>(message);

            return new BusinessResponse(HttpStatusCode.OK, "", msgDto);

        }

        return new BusinessResponse(HttpStatusCode.BadRequest, "Unable to send message");

    }

    public async Task<BusinessResponse> DeleteMessage(int currentUserId, Guid msgGuid)

    {

        var message = await \_msgRepo.GetMessage(msgGuid);

        if(message == null)

            return new BusinessResponse(HttpStatusCode.BadRequest, "No message found");

        if(message.Sender.Id != currentUserId && message.Recipient.Id != currentUserId)

            return new BusinessResponse(HttpStatusCode.Unauthorized);

        //due to EF only the sender will marked as deleted

        if(message.Sender.Id == currentUserId)

            message.SenderDeleted = true;

        //due to EF only the receipent will be marked as deleted

        if(message.Recipient.Id == currentUserId)

            message.RecipientDeleted = true;

        //when both have deleted it then delete it altogether

        if(message.SenderDeleted && message.RecipientDeleted)

            \_msgRepo.DeleteMessage(message);

        //update

        if(await \_msgRepo.SaveAllSync())

            return new BusinessResponse(HttpStatusCode.OK);

        return new BusinessResponse(HttpStatusCode.BadRequest, "Unable to delete message");

    }

    public void DeleteMessage(UserMessage message)

    {

        throw new NotImplementedException();

    }

    public async Task<UserMessage> GetMessage(int id)

    {

        var message = await \_msgRepo.GetMessage(id);

        return message;

    }

    public async Task<UserMessage> GetMessage(Guid guid)

    {

        var message = await \_msgRepo.GetMessage(guid);

        return message;

    }

    public async Task<PagedList<MessageDto>> GetMessagesForUser(MessageSearchParamDto search)

    {

        var messages = await \_msgRepo.GetMessagesForUser(search);

        return messages;

    }

    public async Task<IEnumerable<MessageDto>> GetMessageThread(int currentUserId, int receipientId)

    {

        var messages = await \_msgRepo.GetMessageThread(currentUserId, receipientId);

        if(messages == null)

            return null;

        var messagesDto = \_mapper.Map<IEnumerable<MessageDto>>(messages);

        return messagesDto;

    }

}

## MSC.Core/Extensions

### AppServiceExtensions.cs

Add to the services

    public static IServiceCollection AddServices(this IServiceCollection services, IConfiguration config)

    {

        services.AddScoped<IUserRepository, UserRepository>();

        services.AddScoped<IUserBusinessLogic, UserBusinessLogic>();

        services.AddScoped<ILikesRepository, LikesRepository>();

        services.AddScoped<ILikesBusinessLogic, LikesBusinessLogic>();

        services.AddScoped<IMessageRepository, MessageRepository>();

        services.AddScoped<IMessageBusinessLogic, MessageBusinessLogic>();

# MessageController.cs

using System;

using System.Collections.Generic;

using System.Net;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using MSC.Core.BusinessLogic;

using MSC.Core.Dtos;

using MSC.Core.Dtos.Pagination;

using MSC.Core.Extensions;

using MSC.WebApi.Controller;

namespace MSC.WebApi;

[Authorize]

public class MessageController : BaseApiController

{

    private readonly IMessageBusinessLogic \_msgBl;

    public MessageController(IMessageBusinessLogic msgBl)

    {

        \_msgBl = msgBl;

    }

    [HttpPost("create")]

    public async Task<ActionResult<MessageDto>> CreateMessage([FromBody] MessageCreateDto msg)

    {

        //get the claims

        var claims = User.GetUserClaims();

        if(claims == null)

            return BadRequest("User issue");

        var result = await \_msgBl.AddMessage(msg, claims.Id);

        if(result == null)

            return BadRequest("Unable to send message");

        var message = result.ConvertDataToType<MessageDto>();

        ActionResult actionResult = result.HttpStatusCode switch

        {

            HttpStatusCode.OK => Ok(message),

            HttpStatusCode.BadRequest => BadRequest(result.Message),

            HttpStatusCode.NotFound => NotFound(result.Message),

            \_ => BadRequest("Unable to send message")

        };

        return actionResult;

    }

    [HttpGet("user/get/messages")]

    public async Task<ActionResult<PagedList<MessageDto>>> GetMessagesForUser([FromQuery] MessageSearchParamDto search)

    {

        //get the claims

        var claims = User.GetUserClaims();

        if(claims == null)

            return BadRequest("User issue");

        search.UserId = claims.Id;

        var messages = await \_msgBl.GetMessagesForUser(search);

        if(messages == null)

            return NotFound("No messages found");

        Response.AddPaginationHeader(messages.CurrentPage, messages.PageSize, messages.TotalCount, messages.TotalPages);

        return Ok(messages);

    }

    [HttpGet("thread/{recipientId:int}")]

    public async Task<ActionResult<IEnumerable<MessageDto>>> GetMessageThread([FromRoute] int recipientId)

    {

        //get the claims

        var claims = User.GetUserClaims();

        if(claims == null)

            return BadRequest("User issue");

        var messages = await \_msgBl.GetMessageThread(claims.Id, recipientId);

        if(messages == null)

            NotFound("No messages found");

        return Ok(messages);

    }

    [HttpDelete("delete/{msgGuid:Guid}")]

    public async Task<ActionResult> DeleteMessage([FromRoute] Guid msgGuid)

    {

        //get the claims

        var claims = User.GetUserClaims();

        if(claims == null)

            return BadRequest("User issue");

        var result = await \_msgBl.DeleteMessage(claims.Id, msgGuid);

        if(result == null)

            BadRequest("Unable to delete message");

        ActionResult actionResult = result.HttpStatusCode switch

        {

            HttpStatusCode.OK => Ok(),

            HttpStatusCode.BadRequest => BadRequest(result.Message),

            HttpStatusCode.NotFound => NotFound(result.Message),

            \_ => BadRequest("Unale to delete message")

        };

        return actionResult;

    }

}

# Postman

The items are covered under Site-14