Table of Contents

[Basics 3](#_Toc117116156)

[Current Resources with Changes 3](#_Toc117116157)

[New Resources 3](#_Toc117116158)

[/Core/Entities 3](#_Toc117116159)

[Message.cs 3](#_Toc117116160)

[AppUser.cs 3](#_Toc117116161)

[/Core/DB 4](#_Toc117116162)

[DataContext.cs 4](#_Toc117116163)

[Migrations 5](#_Toc117116164)

[Update DB 5](#_Toc117116165)

[/Core/Enums 5](#_Toc117116166)

[MessageType.cs 5](#_Toc117116167)

[/Core/Dto 5](#_Toc117116168)

[MessageDto.cs 5](#_Toc117116169)

[MessageCreateDto.cs 6](#_Toc117116170)

[BusinessResponse.cs 6](#_Toc117116171)

[/Core/Dto/AutoMapper 6](#_Toc117116172)

[AutoMapperProfiles.cs 6](#_Toc117116173)

[/Core/Dto/Helpers 7](#_Toc117116174)

[MessageParams.cs 7](#_Toc117116175)

[/Core/Repositories 7](#_Toc117116176)

[Message Repository 7](#_Toc117116177)

[IMessageRepository 7](#_Toc117116178)

[MessageRepository 7](#_Toc117116179)

[/Core/BusinessLogic 9](#_Toc117116180)

[Message Business Logic 9](#_Toc117116181)

[IMessageBusinessLogic 9](#_Toc117116182)

[MessageBusinessLogic 9](#_Toc117116183)

[Register Message Repository and Business Logic for IoC 🡺 /Core/Extensions 11](#_Toc117116184)

[ServiceExtensions 11](#_Toc117116185)

[Controllers 11](#_Toc117116186)

[MessagesController.cs 11](#_Toc117116187)

# Basics

|  |  |
| --- | --- |
| WorkingFolder | Copy the content of “Site-03-Like Feature” in “Site-04-Message Feature” and issue   * dotnet build |
| Cloudinary Setting | appsetting.json is not checked in so make sure to bring in this from “Site-01-Basics” |

# Current Resources with Changes

1. /Core/Entities/AppUser
2. /Core/DB/DataContext
3. /Core/Extensions/ServiceExtensions
4. /Core/Dto/AutoMapper/AutoMapperProfiles
5. /Core/Dto/BusinessResponse

# New Resources

1. /Core/Entities/Message
2. /Core/Dto/MessageDto
3. /Core/Dto/MessageCreateDto
4. /Core/Enums/MessageType
5. /Core/Dto/Helpers/MessageParams
6. /Core/Repositories/IMessageRepository
7. /Core/Repositories/MessageRepository
8. /Core/BusinessLogic/IMessageBusinessLogic
9. /Core/BusinessLogic/MessageBusinessLogic
10. /Controllers/MessagesController

# /Core/Entities

## Message.cs

using System;

namespace MSC.Api.Core.Entities;

public class Message

{

    public int Id { get; set; }

    public int SenderId { get; set; }

    public string SenderUsername { get; set; }

    public AppUser Sender { get; set; }

    public bool SenderDeleted { get; set; }

    public int ReceipientId { get; set; }

    public string ReceipientUsername { get; set; }

    public AppUser Receipient { get; set; }

    public bool ReceipientDeleted { get; set; }

    public string MessageContent { get; set; }

    public DateTime? DateMessageRead { get; set; }

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

}

## AppUser.cs

Add the two new properties to AppUser

    public ICollection<Message> MessagesSent { get; set; }

    public ICollection<Message> MessagesReceived { get; set; }

# /Core/DB

## DataContext.cs

Add the DbSet and configure the Messages. Full DataContext class has been posted here.

using Microsoft.EntityFrameworkCore;

using MSC.Api.Core.Entities;

namespace MSC.Api.Core.DB

{

    public class DataContext : DbContext

    {

        public DataContext(DbContextOptions options) : base(options)

        {

        }

        //AppUser will have a table name of Users

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

        //UserLike will have a table name of Likes

        public DbSet<UserLike> Likes { get; set; }

        //UserMessage will have a table name of Messages

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

        //give entities some configuration

        protected override void OnModelCreating(ModelBuilder builder)

        {

            base.OnModelCreating(builder);

            CreateUserLike(builder);

            CreateMessage(builder);

        }

        /// <summary>

        /// Configure User Likes

        /// </summary>

        /// <param name="builder"></param>

        private void CreateUserLike(ModelBuilder builder)

        {

            //user like configuration

            //key is combination of sourceUserId and LikedUserId

            builder.Entity<UserLike>()

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

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

            builder.Entity<UserLike>()

                    .HasOne(s => s.SourceUser)

                    .WithMany(l => l.UsersILiked)

                    .HasForeignKey(s => s.SourceUserId)

                    .OnDelete(DeleteBehavior.Cascade) //when the user is deleted then delete the related entities. For sql server use DeleteBehavior.NoAction

            ;

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

            builder.Entity<UserLike>()

                    .HasOne(s => s.LikedUser)

                    .WithMany(l => l.UsersLikedMe)

                    .HasForeignKey(s => s.LikedUserId)

                    .OnDelete(DeleteBehavior.Cascade) //when the user is deleted then delete the related entities. For sql server use DeleteBehavior.NoAction

            ;

        }

        /// <summary>

        /// Confiure User Messages

        /// </summary>

        /// <param name="builder"></param>

        private void CreateMessage(ModelBuilder builder)

        {

            //user message configuration

            //receiver

            builder.Entity<Message>()

                    .HasOne(r => r.Receipient)

                    .WithMany(m => m.MessagesReceived)

                    .OnDelete(DeleteBehavior.Restrict) //both the parties need to delete the message to be removed from the database

            ;

            //sender

            builder.Entity<Message>()

                    .HasOne(s => s.Sender)

                    .WithMany(m => m.MessagesSent)

                    .OnDelete(DeleteBehavior.Restrict)

            ;

        }

    }

}

### Migrations

Add to migrations

* dotnet ef migrations add MessageEntityAdded -o Core/DB/Migrations

To remove the migration use

* dotnet ef migrations remove

### Update DB

1. either start the application with dotnet watch run or
2. issue command dotnet ef database update

# /Core/Enums

## MessageType.cs

namespace MSC.Api.Core.Enums;

public enum MessageType

{

    Inbox, //received

    InboxUnread, //received not read

    Outbox, //send

}

# /Core/Dto

## MessageDto.cs

using System;

namespace MSC.Api.Core.Dto;

public class MessageDto

{

    public int Id { get; set; }

    public int SenderId { get; set; }

    public Guid SenderGuid { get; set; }

    public string SenderUsername { get; set; }

    public string SenderPhotoUrl { get; set; }

    public int ReceipientId { get; set; }

    public Guid ReceipientGuid { get; set; }

    public string ReceipientUsername { get; set; }

    public string ReceipientPhotoUrl { get; set; }

    public string MessageContent { get; set; }

    public DateTime? DateMessageRead { get; set; }

    public DateTime DateMessageSent { get; set; }

}

## MessageCreateDto.cs

namespace MSC.Api.Core.Dto;

public class MessageCreateDto

{

    public int ReceipientUserId { get; set; }

    public string Content { get; set; }

}

## BusinessResponse.cs

In some cases we may need to return result via this helper class. So create a “Data” property and also a function to convert it back to its type.

using System.Net;

namespace MSC.Api.Core.Dto;

public class BusinessResponse

{

    public BusinessResponse()

    {

    }

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

    {

        HttpStatusCode = httpStatusCode;

        Message = message;

        Data = data;

    }

    public System.Net.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;

    }

}

# /Core/Dto/AutoMapper

## AutoMapperProfiles.cs

Renamed the following private function and all its usage. New name

    private static string PickMainUrl(ICollection<Photo> photos)

and then added message profile

    private void Map\_Message\_to\_MessageDto()

    {

        CreateMap<Message, MessageDto>()

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

        .ForMember(dest => dest.ReceipientPhotoUrl, opt => opt.MapFrom(src => PickMainUrl(src.Receipient.Photos)))

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

        .ForMember(dest => dest.ReceipientGuid, opt => opt.MapFrom(src => src.Receipient.GuId))

        ;

    }

And finally added it to the AutoMapperProfiles

Map\_Message\_to\_MessageDto();

# /Core/Dto/Helpers

## MessageParams.cs

using MSC.Api.Core.Enums;

namespace MSC.Api.Core.Dto.Helpers;

public class MessageParams : PaginationParams

{

    public int UserId { get; set; }

    public MessageType MessageType { get; set; } = MessageType.Unread;

}

# /Core/Repositories

## Message Repository

### IMessageRepository

using System.Collections.Generic;

using System.Threading.Tasks;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Dto.Helpers;

using MSC.Api.Core.Entities;

namespace MSC.Api.Core.Repositories;

public interface IMessageRepository

{

    void AddMessage(Message message);

    void DeleteMessage(Message message);

    Task<Message> GetMessage(int id);

    Task<PageList<MessageDto>> GetMessagesForUser(MessageParams msgParams);

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

    Task<bool> SaveAllSync();

}

### MessageRepository

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using AutoMapper;

using AutoMapper.QueryableExtensions;

using Microsoft.EntityFrameworkCore;

using MSC.Api.Core.DB;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Dto.Helpers;

using MSC.Api.Core.Entities;

using MSC.Api.Core.Enums;

namespace MSC.Api.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(Message message)

    {

        \_context.Messages.Add(message);

    }

    public void DeleteMessage(Message message)

    {

        \_context.Messages.Remove(message);

    }

    public async Task<Message> GetMessage(int id)

    {

        //following will not pull the receipent or sender information

        //var message = await \_context.Messages.FindAsync(id);

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

        var message = await \_context.Messages

                                    .Include(u => u.Receipient)

                                    .Include(u => u.Sender)

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

        return message;

    }

    public async Task<PageList<MessageDto>> GetMessagesForUser(MessageParams msgParams)

    {

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

        query = msgParams.MessageType switch

        {

            //receipent of the message

            MessageType.Inbox => query.Where(u => u.Receipient.Id == msgParams.UserId && !u.ReceipientDeleted),

            //receipent of the message and not read it

            MessageType.InboxUnread => query.Where(u => u.Receipient.Id == msgParams.UserId && u.DateMessageRead == null && !u.ReceipientDeleted),

            //defult sender outbox

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

        };

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

        var pageList = await PageList<MessageDto>.CreateAsync(messages, msgParams.PageNumber, msgParams.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<Message>> GetMessageThread(int currentUserId, int receipientId)

    {

        //message conversations between two users

        var messages = await \_context.Messages

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

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

                                    .Where(m =>

                                            (m.Receipient.Id == currentUserId && m.Sender.Id == receipientId && !m.ReceipientDeleted) ||

                                            (m.Receipient.Id == receipientId && m.Sender.Id == currentUserId && !m.SenderDeleted)

                                        )

                                    .OrderBy(m => m.DateMessageSent)

                                    .ToListAsync();

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

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

        {

            //update the date

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

            await \_context.SaveChangesAsync();

        }

        return messages;

    }

    public async Task<bool> SaveAllSync()

    {

        return await \_context.SaveChangesAsync() > 0;

    }

}

# /Core/BusinessLogic

## Message Business Logic

### IMessageBusinessLogic

using System.Collections.Generic;

using System.Threading.Tasks;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Dto.Helpers;

using MSC.Api.Core.Entities;

namespace MSC.Api.Core.BusinessLogic;

public interface IMessageBusinessLogic

{

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

    void DeleteMessage(Message message);

    Task<Message> GetMessage(int id);

    Task<PageList<MessageDto>> GetMessagesForUser(MessageParams msgParams);

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

    Task<BusinessResponse> DeleteMessage(int currentUserId, int msgId);

}

### MessageBusinessLogic

using System.Collections.Generic;

using System.Net;

using System.Threading.Tasks;

using AutoMapper;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Dto.Helpers;

using MSC.Api.Core.Entities;

using MSC.Api.Core.Repositories;

namespace MSC.Api.Core.BusinessLogic;

public class MessageBusinessLogic : IMessageBusinessLogic

{

    private readonly IMessageRepository \_msgRepo;

    private readonly IUsersRepository \_usersRepo;

    private readonly IMapper \_mapper;

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

    {

        \_msgRepo = msgRepo;

        \_usersRepo = userRepo;

        \_mapper = mapper;

    }

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

    {

        if (msg == null || msg.ReceipientUserId <= 0 || string.IsNullOrWhiteSpace(msg.Content))

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

        //get source user

        var sender = await \_usersRepo.GetAppUserAsync(senderId, includePhotos: true);

        if (sender == null)

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

        if (sender.Id == msg.ReceipientUserId)

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

        var receipient = await \_usersRepo.GetAppUserAsync(msg.ReceipientUserId, includePhotos: true);

        if (receipient == null)

            return new BusinessResponse(HttpStatusCode.NotFound, "Receipient not found");

        var message = new Message

        {

            Sender = sender,

            Receipient = receipient,

            SenderUsername = sender.UserName,

            ReceipientUsername = receipient.UserName,

            MessageContent = msg.Content

        };

        \_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 void DeleteMessage(Message message)

    {

        throw new System.NotImplementedException();

    }

    public Task<Message> GetMessage(int id)

    {

        throw new System.NotImplementedException();

    }

    public async Task<PageList<MessageDto>> GetMessagesForUser(MessageParams msgParams)

    {

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

        return messages;

    }

    //message between two users. Also marks receipients un read messages as read

    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;

    }

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

    {

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

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

            return new BusinessResponse(HttpStatusCode.Unauthorized);

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

        if (message.Sender.Id == currentUserId)

            message.SenderDeleted = true;

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

        if (message.Receipient.Id == currentUserId)

            message.ReceipientDeleted = true;

        //when both have deleted it then delete it altogether

        if (message.SenderDeleted && message.ReceipientDeleted)

            \_msgRepo.DeleteMessage(message);

        if (await \_msgRepo.SaveAllSync())

            return new BusinessResponse(HttpStatusCode.OK);

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

    }

}

# Register Message Repository and Business Logic for IoC 🡺 /Core/Extensions

## ServiceExtensions

#### RegisterRepos

        services.AddScoped<IMessageRepository, MessageRepository>();

        services.AddScoped<IMessageBusinessLogic, MessageBusinessLogic>();

# Controllers

## MessagesController.cs

using System.Collections.Generic;

using System.Net;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.Mvc;

using MSC.Api.Core.BusinessLogic;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Dto.Helpers;

using MSC.Api.Core.Extensions;

namespace MSC.Api.Controllers;

[Authorize]

public class MessagesController : BaseApiController

{

    private readonly IMessageBusinessLogic \_msgBl;

    public MessagesController(IMessageBusinessLogic msgBl)

    {

        \_msgBl = msgBl;

    }

    [HttpPost("send/message")]

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

    {

        //get the claims

        var userClaims = User.GetUserClaims();

        if (userClaims == null || (!userClaims.HasGuid || !userClaims.HasUserName))

            return BadRequest("User issue");

        var result = await \_msgBl.AddMessage(msg, userClaims.UserId);

        if (result == null)

            return BadRequest("Unable to send message");

        ActionResult actionResult = BadRequest("Unable to send message");

        switch (result.HttpStatusCode)

        {

            case HttpStatusCode.OK:

                actionResult = Ok(result.ConvertDataToType<MessageDto>());

                break;

            case HttpStatusCode.BadRequest:

                actionResult = BadRequest(result.Message);

                break;

            case HttpStatusCode.NotFound:

                actionResult = NotFound(result.Message);

                break;

            default:

                actionResult = BadRequest("Unable to send message");

                break;

        }

        return actionResult;

    }

    [HttpGet("user/get/messages")]

    public async Task<ActionResult<IEnumerable<MessageDto>>> GetMessagesForUser([FromQuery] MessageParams msgParams)

    {

        //get the claims

        var userClaims = User.GetUserClaims();

        if (userClaims == null || (!userClaims.HasGuid || !userClaims.HasUserName))

            return BadRequest("User issue");

        msgParams.UserId = userClaims.UserId;

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

        if (messages == null)

            return NotFound("No messages found");

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

        return Ok(messages);

    }

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

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

    {

        //get the claims

        var userClaims = User.GetUserClaims();

        if (userClaims == null || (!userClaims.HasGuid || !userClaims.HasUserName))

            return BadRequest("User issue");

        var messages = await \_msgBl.GetMessageThread(userClaims.UserId, otherUserId);

        if (messages == null)

            return NotFound("No messages found");

        return Ok(messages);

    }

    [HttpDelete("delete/message/{id:int}")]

    public async Task<ActionResult> DeleteMessage([FromRoute] int id)

    {

        //get the claims

        var userClaims = User.GetUserClaims();

        if (userClaims == null || (!userClaims.HasGuid || !userClaims.HasUserName))

            return BadRequest("User issue");

        var result = await \_msgBl.DeleteMessage(userClaims.UserId, id);

        if (result == null)

            return BadRequest("Unable to delete message");

        ActionResult actionResult = BadRequest("Unable to delete message");

        switch (result.HttpStatusCode)

        {

            case HttpStatusCode.OK:

                actionResult = Ok(result.ConvertDataToType<MessageDto>());

                break;

            case HttpStatusCode.BadRequest:

                actionResult = BadRequest(result.Message);

                break;

            case HttpStatusCode.NotFound:

                actionResult = NotFound(result.Message);

                break;

            default:

                actionResult = BadRequest("Unable to delete message");

                break;

        }

        return actionResult;

    }

}