# Project

* Site-16-Api-Ng-sinalR
  + Copied from: Site-15-Api-Ng-identity-role-management
* 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/SignalR/PresenceTrackerMemory.cs | MSC.WebApi/SignalR/PresenceHub.cs |
| MSC.Core/DB/Entities/SignalR/SignalRGoup.cs | MSC.WebApi/SignalR/MessageHub.cs |
| MSC.Core/DB/Entities/SignalR/SignalConnection.cs |  |
| MSC.Core/Repositories/SignalRRepository |  |
| MSC.Core/BusinessLogic/SignalRBusinessLogic |  |
| MSC.Core/BusinessLogiv/MessageBusinessLogic |  |
|  |  |
|  |  |
|  |  |

# Resources updated

|  |  |
| --- | --- |
| MSC.Core | MSC.WebApi |
| MSC.Core/Extensions/AppServiceExtensions.cs | Programs.cs |
| MSC.Core/DB/Data/DataContext.cs |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Clear Database

* New migrations created
* Clear the users
* dotnet ef database drop
* dotnet ef database update

# Intro

No package to install for SignalR

# MSC.Core/BusinessLogic

## UserBusiessLogic

### IUSerBusinessLogic.cs

Update current method to get includePhotos param and add new methid by id

    Task<AppUser> GetUserRawAsync(string userName, bool includePhotos = false);

    Task<AppUser> GetUserRawAsync(int userName, bool includePhotos = false);

### UserBusinessLogic.cs

Update current method and create new method

    public async Task<AppUser> GetUserRawAsync(string userName, bool includePhotos = false)

    {

        if(string.IsNullOrWhiteSpace(userName))

            throw new ValidationException("User name missing");

        var user = await \_userRepo.GetUserRawAsync(userName, includePhotos);

        return user;

    }

    public async Task<AppUser> GetUserRawAsync(int id, bool includePhotos = false)

    {

        if(id <= 0)

            throw new ValidationException("User id missing");

        var user = await \_userRepo.GetUserAsync(id, includePhotos);

        return user;

    }

## MessageBusinessLogic

### IMessageBusinessLogic.cs

Add a new method signature just like the other AddMessage

Task<BusinessResponse> AddMessageWithReadRecipt(MessageCreateDto msg, int senderId, bool markMsgAsRead);

### MessageBusinessLogic.cs

Move the content of the AddMessage into a private function.

Create the new AddMessageWithReceipt and update the current AddMessage method

If the Boolean markMsgAsRead is true then make DateMessageRead = utc now

    public async Task<BusinessResponse> AddMessageWithReadRecipt(MessageCreateDto msg, int senderId, bool markMsgAsRead)

    {

        var result = await AddMessageHandle(msg, senderId, markMsgAsRead);

        return result;

    }

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

    {

        var result = await AddMessageHandle(msg, senderId, markMsgAsRead: false);

        return result;

    }

    private async Task<BusinessResponse> AddMessageHandle(MessageCreateDto msg, int senderId, bool markMsgAsRead = false)

    {

        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(recipient == 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

        };

        if(markMsgAsRead){

            message.DateMessageRead = DateTime.UtcNow;

        }

        \_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");

    }

# PresenceHub

## MSC.Core/SignalR

### PresenceTrackerMemory.cs

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace MSC.WebApi.SignalR;

/// <summary>

/// starting with local presence tracker. The elaborate one would be with redis or in database

/// add as a singleton in services

/// </summary>

public class PresenceTrackerMemory

{

    //dictionary to track the users when logging in and logging out

    //key will be the user name

    //value will be the list of connection ids for that user, like from phone and desktop

    private static readonly Dictionary<string, List<string>> \_onlineUsers = new Dictionary<string, List<string>>();

    public Task<bool> UserConnected(string userName, string connectionId)

    {

        var isOnline = false;

        lock (\_onlineUsers)

        {

            if(\_onlineUsers.ContainsKey(userName))

                \_onlineUsers[userName].Add(connectionId);

            else{

                \_onlineUsers.Add(userName, new List<string>{ connectionId});

                isOnline = true;

            }

        }

        //return Task.CompletedTask;

        return Task.FromResult(isOnline);

    }

    public Task<bool> UserDisconnected(string userName, string connectionId)

    {

        var isOffLine = false;

        lock(\_onlineUsers){

            if(!\_onlineUsers.ContainsKey(userName)){

                //return Task.CompletedTask;

                return Task.FromResult(isOffLine);

            }

            //remove the connections

            \_onlineUsers[userName].Remove(connectionId);

            if(\_onlineUsers[userName].Count == 0){

                //remove the user

                \_onlineUsers.Remove(userName);

                isOffLine = true;

            }

        }

        //return Task.CompletedTask;

        return Task.FromResult(isOffLine);

    }

    public Task<string[]> GetOnlineUsers(){

        string[] onlineUsers;

        lock (\_onlineUsers){

            onlineUsers = \_onlineUsers.OrderBy(k => k.Key).Select(k => k.Key).ToArray();

        }

        return Task.FromResult(onlineUsers);

    }

    public Task<string[]> GetConnectionsForUser(string userName){

        string[] connectionsIds;

        lock (\_onlineUsers){

            connectionsIds = \_onlineUsers.GetValueOrDefault(userName).ToArray();

        }

        return Task.FromResult(connectionsIds);

    }

}

## MSC.WebApi/SignalR

### PresenceHub.cs

* It is a C# class
* Displays the presence of users like user login/logout and online users
* Starting with local presence tracker. The elaborate one would be with redis or in database
* PresenceHub derives from Hub and then override the virtual methods

using System;

using System;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.SignalR;

using MSC.Core.Extensions;

using MSC.Core.SignalR;

namespace MSC.WebApi.SignalR;

[Authorize]

/// <summary>

/// Displays the presence of users like user login and online users

/// Starting with local presence tracker. The elaborate one would be with redis or in database

/// PresenceHub, it derives from Hub and then override the virtual methods

/// No package to install for SignalR

/// Has methods when the user connects to the Hub and again when the user Disconnects from the Hub

/// </summary>

public class PresenceHub : Hub

{

    //NOTE: NewMessageReceived event is being sent by the MessageBub

    private const string \_userIsOnline = "UserIsOnline";

    private const string \_userIsOffline = "UserIsOffline";

    private const string \_getOnlineUsers = "GetOnlineUsers";

    private readonly PresenceTrackerMemory \_tracker;

    public PresenceHub(PresenceTrackerMemory tracker)

    {

        \_tracker = tracker;

    }

    /// <summary>

    /// Implement OnConnectedAsync to tell other users when the current user goes online

    /// </summary>

    /// <returns></returns>

    public override async Task OnConnectedAsync()

    {

        var userName =  Context.User.GetUserName();

        var connectionId = Context.ConnectionId;

        var isOnline = await \_tracker.UserConnected(userName, connectionId);

        if(isOnline){

            //invoke messages to clients other than logging in user that a user is logging in

            await Clients.Others.SendAsync(\_userIsOnline, userName);

        }

        var onlineUsers = await \_tracker.GetOnlineUsers();

        //send to all the users

        //await Clients.All.SendAsync(\_getOnlineUsers, onlineUsers);

        //only send to the current user

        await Clients.Caller.SendAsync(\_getOnlineUsers, onlineUsers);

    }

    /// <summary>

    /// Implmenent OnDisconnectedAsync to tell other users when the current user goes offline

    /// </summary>

    /// <param name="exception"></param>

    /// <returns></returns>

    public override async  Task OnDisconnectedAsync(Exception exception)

    {

        var userName =  Context.User.GetUserName();

        var connectionId = Context.ConnectionId;

        var isOffLine = await \_tracker.UserDisconnected(userName, connectionId);

        if(isOffLine){

            //invoke messages to clients other than logging out user that a user is logging out

            await Clients.Others.SendAsync(\_userIsOffline, userName);

        }

        //original: get the users online and send to every one who is connected

        //update: not sending the list

        var onlineUsers = await \_tracker.GetOnlineUsers();

        //await Clients.All.SendAsync(\_getOnlineUsers, onlineUsers);

        await Clients.Caller.SendAsync(\_getOnlineUsers, onlineUsers);

        await base.OnDisconnectedAsync(exception);

    }

}

# MessageHub

## Important

Will be tracking the group and connections via the DB so created the entities, repo and BL.

## MSC.Core/DB

### Entities/SignalR

#### *SignalRConnection*.cs

using System.ComponentModel.DataAnnotations;

namespace MSC.Core.Entities.SignalR;

public class SignalRConnection

{

    /// <summary>

    /// Empty constructory is needed for the EF

    /// </summary>

    public SignalRConnection()

    {

    }

    public SignalRConnection(string connectionId, string userName, int userId)

    {

        ConnectionId = connectionId;

        UserName = userName;

        UserId = userId;

    }

    [Key]

    public string ConnectionId { get; set; }

    public string UserName { get; set; }

    public int UserId { get; set; }

}

#### SignalRGroup.cs

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace MSC.Core.Entities.SignalR;

public class SignalRGroup

{

    /// <summary>

    /// Empty constructor is needed for the EF

    /// </summary>

    public SignalRGroup()

    {

    }

    public SignalRGroup(string groupName)

    {

        GroupName = groupName;

    }

    /// <summary>

    /// GroupName is the unique key and it will be indexed as well

    /// </summary>

    [Key]

    public string GroupName { get; set; }

    public ICollection<SignalRConnection> Connections { get; set; } = new List<SignalRConnection>();

}

### Data

#### DataContext.cs

Add the new entities to the dbset

    public DbSet<SignalRGroup> SignalRGroups { get; set; }

    public DbSet<SignalRConnection> SignalRConnections { get; set; }

### Create Migrations

|  |  |
| --- | --- |
| > dotnet ef migrations add SignalRGroupsAdded  Done. To undo this action, use 'ef migrations remove'  > dotnet ef database update |  |

## MSC.Core/Repositories

### ISignalRRepository.cs

using System.Threading.Tasks;

using MSC.Core.DB.Entities.SignalR;

namespace MSC.Core.Repositories;

public interface ISignalRRepository

{

    Task<bool> SaveAllSync();

    void AddGroup(SignalRGroup group);

    void RemoveConnection(SignalRConnection connection);

    Task<SignalRGroup> GetMessageGroup(string groupName);

    Task<SignalRGroup> GetGroupByConnection(string connectionId);

    Task<SignalRConnection> GetConnection(string connectionId);

}

### SignalRRepository.cs

using System.Linq;

using System.Threading.Tasks;

using AutoMapper;

using Microsoft.EntityFrameworkCore;

using MSC.Core.DB.Data;

using MSC.Core.DB.Entities.SignalR;

namespace MSC.Core.Repositories;

public class SignalRRepository : ISignalRRepository

{

    private readonly DataContext \_context;

    private readonly IMapper \_mapper;

    public SignalRRepository(DataContext context, IMapper mapper)

    {

        \_context = context;

        \_mapper = mapper;

    }

    public async Task<bool> SaveAllSync()

    {

        return await \_context.SaveChangesAsync() > 0;

    }

    public void AddGroup(SignalRGroup group)

    {

        \_context.SignalRGroups.Add(group);

    }

    public void RemoveConnection(SignalRConnection connection)

    {

        \_context.SignalRConnections.Remove(connection);

    }

    public async Task<SignalRGroup> GetMessageGroup(string groupName)

    {

        //also fill in the connections for the group

        var group = await \_context.SignalRGroups

                                .Include(x => x.Connections)

                                .FirstOrDefaultAsync(x => x.GroupName == groupName);

        return group;

    }

    public async Task<SignalRGroup> GetGroupByConnection(string connectionId)

    {

        var group = await \_context.SignalRGroups

                                .Include(x => x.Connections)

                                .Where(x => x.Connections.Any(x => x.ConnectionId == connectionId))

                                .FirstOrDefaultAsync();

        return group;

    }

    public async Task<SignalRConnection> GetConnection(string connectionId)

    {

       var connection = await \_context.SignalRConnections.FindAsync(connectionId);

       return connection;

    }

}

## MSC.Core/BusinessLogic

### ISignalRBusinessLogic.cs

using System.Threading.Tasks;

using MSC.Core.DB.Entities.SignalR;

namespace MSC.Core.BusinessLogic;

public interface ISignalRBusinessLogic

{

    Task<bool> SaveAllSync();

    void AddGroup(SignalRGroup group);

    void RemoveConnection(SignalRConnection connection);

    Task<SignalRGroup> GetMessageGroup(string groupName);

    Task<SignalRGroup> GetGroupByConnection(string connectionId);

    Task<SignalRConnection> GetConnection(string connectionId);

}

### SignalRBusinessLogic.cs

using System.Threading.Tasks;

using MSC.Core.DB.Entities.SignalR;

using MSC.Core.Repositories;

namespace MSC.Core.BusinessLogic;

public class SignalRBusinessLogic : ISignalRBusinessLogic

{

    private readonly ISignalRRepository \_srRepo;

    public SignalRBusinessLogic(ISignalRRepository srRepo)

    {

        \_srRepo = srRepo;

    }

    public async Task<bool> SaveAllSync()

    {

        return await \_srRepo.SaveAllSync();

    }

    public void AddGroup(SignalRGroup group)

    {

        \_srRepo.AddGroup(group);

    }

    public void RemoveConnection(SignalRConnection connection)

    {

        \_srRepo.RemoveConnection(connection);

    }

    public async Task<SignalRGroup> GetMessageGroup(string groupName)

    {

       var group = await \_srRepo.GetMessageGroup(groupName);

       return group;

    }

    public async Task<SignalRGroup> GetGroupByConnection(string connectionId)

    {

        var group = await \_srRepo.GetGroupByConnection(connectionId);

        return group;

    }

    public Task<SignalRConnection> GetConnection(string connectionId)

    {

        var connection = \_srRepo.GetConnection(connectionId);

        return connection;

    }

}

## MSC.WebApi/SignalR

### MessageHub.cs – Without Groups Tracking – Simple

This is simple implementation. Follow the implementation [below](#_MessageHub.cs_–_With) with groups tracking

using System;

using System.Net;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.SignalR;

using MSC.Core.BusinessLogic;

using MSC.Core.Dtos;

using MSC.Core.Extensions;

namespace MSC.WebApi.SignalR;

[Authorize]

/// <summary>

/// MessageHub, it derives from Hub and then override the virtual methods

/// No package to install for SignalR

/// Messages will be sent using the MessageHub and not MessageController.CreateMessage

/// </summary>

public class MessageHub : Hub

{

    private const string \_keyReceiveMessageThread = "ReceiveMessageThread";

    private const string \_keyNewMessage = "NewMessage";

    private readonly IMessageBusinessLogic \_msgBl;

    private readonly IUserBusinessLogic \_userBl;

    public MessageHub(IMessageBusinessLogic msgBl,

                    IUserBusinessLogic userBl)

    {

        \_msgBl = msgBl;

        \_userBl = userBl;

    }

    /// <summary>

    /// Implement OnConnectedAsync

    /// </summary>

    /// <returns></returns>

    public override async Task OnConnectedAsync()

    {

        var httpContext = Context.GetHttpContext();

        var connectionId = Context.ConnectionId;

        //current user

        var callerUserName = Context.User.GetUserName();

        var callerUserId = Context.User.GetId();

        //will be passing in the other users info via query string

        var otherUserName = httpContext.Request.Query["otherUserName"].ToString();

        var otherUserId = int.Parse(httpContext.Request.Query["otherUserId"].ToString());

        //build the group name, create a group of two users.

        var groupName = GetGroupName(callerUserName, otherUserName);

        //add to SignalR groups

        await Groups.AddToGroupAsync(connectionId, groupName);

        //get the message thread from the message business logic just like MessgeController

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

        //send the messages to the caller

        await Clients.Group(groupName).SendAsync(\_keyReceiveMessageThread, messages);

    }

    /// <summary>

    /// Implmenent OnDisconnectedAsync

    /// </summary>

    /// <param name="exception"></param>

    /// <returns></returns>

    public override Task OnDisconnectedAsync(Exception exception)

    {

        return base.OnDisconnectedAsync(exception);

    }

    /// <summary>

    /// Moved here from MessageController

    /// </summary>

    /// <param name="msg"></param>

    /// <returns></returns>

    public async Task CreateMessage(MessageCreateDto msg)

    {

        //get the claims

        var claims = Context.User.GetUserClaims();

        if(claims == null || !claims.HasUserName || !claims.HasId || !claims.HasGuid)

            throw new HubException("User issue");

        //check message

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

            throw new HubException("Message info invalid");

        //add message

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

        if(result == null)

            throw new HubException("Unable to send message");

        if(result.HttpStatusCode != HttpStatusCode.OK)

            throw new HubException(result.Message ?? "Unable to send message");

        //the message that got added

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

        //build group name

        var groupName = GetGroupName(messagedAdded.SenderUsername, messagedAdded.RecipientUsername);

        await Clients.Group(groupName).SendAsync(\_keyNewMessage, messagedAdded);

    }

    //sort in alphabatical order and build group name

    private string GetGroupName(string caller, string other)

    {

        //Less than zero –strA is less than strB.

        //Zero –strA and strB are equal.

        //Greater than zero –strA is greater than strB

        var stringCompare = string.CompareOrdinal(caller, other) < 0;

        return stringCompare ? $"{caller}-{other}" : $"{other}-{caller}";

    }

}

### MessageHub.cs – With Groups Tracking

**Constructor**

Inject the SignalRBusinessLogic in constructor

    private readonly ISignalRBusinessLogic \_srBl;

    public MessageHub(IMessageBusinessLogic msgBl,

                    IUserBusinessLogic userBl,

                    ISignalRBusinessLogic srBl)

    {

        \_msgBl = msgBl;

        \_userBl = userBl;

        \_srBl = srBl;

    }

Here is the full thing

using System;

using System.Linq;

using System.Net;

using System.Threading.Tasks;

using AutoMapper;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.SignalR;

using MSC.Core.BusinessLogic;

using MSC.Core.DB.Entities.SignalR;

using MSC.Core.Dtos;

using MSC.Core.Extensions;

using MSC.Core.SignalR;

namespace MSC.WebApi.SignalR;

[Authorize]

/// <summary>

/// MessageHub, it derives from Hub and then override the virtual methods

/// No package to install for SignalR

/// Messages will be sent using the MessageHub and not MessageController.CreateMessage

/// </summary>

public class MessageHub : Hub

{

    private const string \_keyReceiveMessageThread = "ReceiveMessageThread";

    private const string \_keyNewMessage = "NewMessage";

    private const string \_keyUpdatedGroup = "UpdatedGroup";

    //when the recipient is not on the message page, is online then send the new message received event

    private const string \_keyNewMessageReceived = "NewMessageReceived";

    private readonly IMessageBusinessLogic \_msgBl;

    private readonly IUserBusinessLogic \_userBl;

    private readonly ISignalRBusinessLogic \_srBl;

    private readonly PresenceTrackerMemory \_presenceTracker;

    private readonly IHubContext<PresenceHub> \_presenceHub;

    private readonly IMapper \_mapper;

    public MessageHub(IMessageBusinessLogic msgBl,

                    IUserBusinessLogic userBl,

                    ISignalRBusinessLogic srBl,

                    PresenceTrackerMemory presenceTracker,

                    IHubContext<PresenceHub> presenceHub,

                    IMapper mapper)

    {

        \_msgBl = msgBl;

        \_userBl = userBl;

        \_srBl = srBl;

        \_presenceTracker = presenceTracker;

        \_presenceHub = presenceHub;

        \_mapper = mapper;

    }

    /// <summary>

    /// Implement OnConnectedAsync

    /// </summary>

    /// <returns></returns>

    public override async Task OnConnectedAsync()

    {

        var httpContext = Context.GetHttpContext();

        var connectionId = Context.ConnectionId;

        //current user

        var callerUserName = Context.User.GetUserName();

        var callerUserId = Context.User.GetId();

        //will be passing in the other users info via query string

        var otherUserName = httpContext.Request.Query["otherUserName"].ToString();

        var otherUserId = int.Parse(httpContext.Request.Query["otherUserId"].ToString());

        //build the group name, create a group of two users.

        var groupName = GetGroupName(callerUserName, otherUserName);

        //add to SignalR groups

        await Groups.AddToGroupAsync(connectionId, groupName);

        //add to database

        var group = AddToGroup(groupName, connectionId, callerUserName, callerUserId);

        //send the message to the group

        await Clients.Group(groupName).SendAsync(\_keyUpdatedGroup, group);

        //get the message thread from the message business logic just like MessgeController

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

        //send the messages to the caller

        //await Clients.Group(groupName).SendAsync(\_keyReceiveMessageThread, messages);

        await Clients.Caller.SendAsync(\_keyReceiveMessageThread, messages);

    }

    /// <summary>

    /// Implmenent OnDisconnectedAsync

    /// </summary>

    /// <param name="exception"></param>

    /// <returns></returns>

    public override async Task OnDisconnectedAsync(Exception exception)

    {

        var group = await RemoveFromMessageGroup(Context.ConnectionId);

        await Clients.Group(group.GroupName).SendAsync(\_keyUpdatedGroup, group);

        //users will be automatically removed from the group

        await base.OnDisconnectedAsync(exception);

    }

    /// <summary>

    /// Moved here from MessageController

    /// </summary>

    /// <param name="msg"></param>

    /// <returns></returns>

    public async Task CreateMessage(MessageCreateDto msg)

    {

        //get the claims

        var claims = Context.User.GetUserClaims();

        if(claims == null || !claims.HasUserName || !claims.HasId || !claims.HasGuid)

            throw new HubException("User issue");

        //check message

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

            throw new HubException("Message info invalid");

        //check that we have a connection for the recipient. If we have then the recipient is on message page

        bool isRecipientOnMessagePage = false;

        bool markMessageAsRead = false;

        var recipient = await \_userBl.GetUserRawAsync(msg.RecipientId, includePhotos: true);

        if(recipient == null)

            throw new HubException("Recipient not found");

        var groupName = GetGroupName(claims.UserName, recipient.UserName);

        SignalRGroup group = await \_srBl.GetMessageGroup(groupName);

        if(group != null && group.Connections.Any(x => x.UserName == recipient.UserName)){

            isRecipientOnMessagePage = true;

            markMessageAsRead = true;

        }

        //add message , mark the message read if the recipient is in message group with sender

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

        if(result == null)

            throw new HubException("Unable to send message");

        if(result.HttpStatusCode != HttpStatusCode.OK)

            throw new HubException(result.Message ?? "Unable to send message");

        //the message that got added

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

        //when the recipient is not on the same message page and have connection then notify the recipient

        if(!isRecipientOnMessagePage){

            var connections = await \_presenceTracker.GetConnectionsForUser(recipient.UserName);

            if(connections != null){

                var sender = await \_userBl.GetUserAsync(Context.User.GetId());

                if(sender != null){

                    //to display to the logged in user the sender info

                    //since presenceHub us bing used to send the message, implement this on the presenceHub client

                    await \_presenceHub.Clients.Clients(connections).SendAsync(\_keyNewMessageReceived, sender);

                }

            }

        }

        //send the event for new message created

        await Clients.Group(groupName).SendAsync(\_keyNewMessage, messagedAdded);

    }

    //sort in alphabatical order and build group name

    private string GetGroupName(string caller, string other)

    {

        //Less than zero –strA is less than strB.

        //Zero –strA and strB are equal.

        //Greater than zero –strA is greater than strB

        var stringCompare = string.CompareOrdinal(caller, other) < 0;

        return stringCompare ? $"{caller}-{other}" : $"{other}-{caller}";

    }

    private async Task<SignalRGroup> AddToGroup(string groupName, string connectionId, string callerUserName, int callerUSerId)

    {

        //get the group from the DB and save it

        SignalRGroup group = await \_srBl.GetMessageGroup(groupName);

        if(group == null){

            group = new SignalRGroup(groupName);

            //only saving the group when not found.

            \_srBl.AddGroup(group);

        }

        //create connection

        SignalRConnection connection = new SignalRConnection(connectionId, callerUserName, callerUSerId);

        //add connection to group and call save method

        group.Connections.Add(connection);

        //save

        if(await \_srBl.SaveAllSync())

            return group;

        throw new HubException("Failed to join group");

    }

    private async Task<SignalRGroup> RemoveFromMessageGroup(string connectionId)

    {

        SignalRGroup group = await \_srBl.GetGroupByConnection(connectionId);

        if(group == null)

            throw new HubException("Failed to get group for connection");

        SignalRConnection connection = group.Connections.FirstOrDefault(x => x.ConnectionId == connectionId);

        if (connection == null)

            throw new HubException("Failed to get connection");

        \_srBl.RemoveConnection(connection);

        if(await \_srBl.SaveAllSync()){

            //group.Connections.Remove(connection);

            return group;

        }

        throw new HubException("Failed to remove from group");

    }

}

# Setup PresenceHub and MessageHub

## MSC.Core/Extensions

### AppServiceExtensions.cs

#### AddService Method

* Add signalR as a service
* Add PresenceTrackerMemory.cs as singleton
* Add SignalRRepository and SignalRBusinessLogic

    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>();

        services.AddScoped<ITokenService, TokenService>();

        services.AddScoped<IPhotoService, PhotoService>();

        //services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies()); //when have single project/assembly

        services.AddAutoMapper(typeof(AutoMapperProfiles).Assembly);

        //adding the Cloudinary to read data from

        //check programs.cs for ref: builder.Services.Configure<EnvConfig>(configuration);

        services.Configure<CloudinaryConfig>(config.GetSection(ConfigKeyConstants.CloudinarySettingsKey));

        //add the action filter as a service, it wil get applied to the abse controller

        services.AddScoped<LogUserActivityFilter>();

        services.AddSignalR();

        services.AddSingleton<PresenceTrackerMemory>();

        services.AddScoped<ISignalRRepository, SignalRRepository>();

        services.AddScoped<ISignalRBusiessLogic, SignalRBusinessLogic>();

        return services;

    }

#### AddAuthenticationService Method

Handle authentication for Signalr since it is not using http/ May be using web sockets or any other method

        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

        .AddJwtBearer(options => {

            //http authentication

            options.TokenValidationParameters = new TokenValidationParameters

            {

                ValidateIssuerSigningKey = true,

                IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(config.GetTokenKey())),

                ValidateIssuer = false,

                ValidateAudience = false

            };

            //signalR or websockets cannot send authentication header. Have to use query string with SignalR

            //signalR, this allows the client to send the token as query string

            options.Events = new JwtBearerEvents

            {

                //pass it as a query string

                OnMessageReceived  = context =>

                {

                    //get access\_token from the query

                    var accessToken = context.Request.Query["access\_token"];

                    //check the path of the request and only do for signalr. Our paths starts with hubs/ check programs.cs for details

                    var path = context.HttpContext.Request.Path;

                    if(!string.IsNullOrEmpty(accessToken) && path.StartsWithSegments("/hubs"))

                    {

                        context.Token = accessToken;

                    }

                    return Task.CompletedTask;

                }

            };

        });

#### AddCorsServicePolicyBased Method

AddCredentials for SignalR

    public static string AddCorsServicePolicyBased(this IServiceCollection services, IConfiguration config)

    {

        var myAllowSpecificOrigins = "\_myAllowSpecificOrigins";

        //https://stackoverflow.com/questions/42858335/how-to-hardcode-and-read-a-string-array-in-appsettings-json

        var allowedSpecificOrigins = config.GetAllowSpecificOrigins();

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

        {

            services.AddCors(options =>

            {

                options.AddPolicy(name: myAllowSpecificOrigins,

                                policy =>

                                {

                                    policy.WithOrigins(allowedSpecificOrigins.ToArray())

                                    .AllowAnyHeader()

                                    .AllowAnyMethod()

                                    //signalR

                                    .AllowCredentials()

                                    ;

                                });

            });

        }

        return myAllowSpecificOrigins;

    }

## MSC.WebApi/Programs.cs

### Hub EndPoints

Create an end point for the above two created hubs for presence and message

app.MapControllers();

/\*\*\*Custom SignalR EndPoint - to put after app.MapControllers() Start\*\*\*/

app.MapHub<PresenceHub>("hubs/presence");

app.MapHub<MessageHub>("hubs/message");

/\*\*\*Custom SignalR EndPoint End\*\*\*/

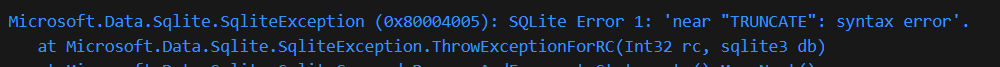
### Clearing the old Message Hub Connections

We may end up with old connections when the application starts etc.

In this case the MessageHub “OnDisconnectedAsync” will not get called.

In Programs.cs where we have seed data, also add code for deleting old connections

Keep in mind TRUNCATE doesn’t work for SQLite



/\*\*\*Custom Section Seed Data Start\*\*\*/

//IR\_REFACTOR

using var scope = app.Services.CreateScope();

var services = scope.ServiceProvider;

var logger = services.GetService<ILogger<Program>>();

try{

    var context = services.GetRequiredService<DataContext>();

    var userManager = services.GetRequiredService<UserManager<AppUser>>();

    var roleManager = services.GetRequiredService<RoleManager<AppRole>>();

    //Asynchronously applies any pending migrations for the context to the database. Will create the database if it does not already exist.

    await context.Database.MigrateAsync();

    try{

        //Remove old MessageHub Connection app start. TRUNCATE doesn't work with TRUNCATE

        //way #1

        //context.SignalRConnections.RemoveRange(context.SignalRConnections);

        //way #2

        //var remove = @"TRUNCATE TABLE [SignalRConnections]";

        var remove = @"DELETE FROM [SignalRConnections]";

        //await context.Database.ExecuteSqlRawAsync(remove);

        await context.Database.ExecuteSqlRawAsync(remove);

    }

    catch (Exception ex){

        logger.LogError(ex, ex.Message);

    }

    //await SeedData.SeedUsers(context);

    await SeedData.SeedUsers(userManager, roleManager);

}

catch(Exception ex)

{

    logger.LogError(ex, "An error occured during seeding data");

}

/\*\*\*Custom Section Seed Data End\*\*\*/

/\*\*\*Custom Section Seed Data End\*\*\*/