Table of Contents

[Basics 3](#_Toc118581765)

[Current Resources with Changes 3](#_Toc118581766)

[New Resources 3](#_Toc118581767)

[/Core/Entities 3](#_Toc118581768)

[Message.cs 3](#_Toc118581769)

[SignalRConnection.cs 3](#_Toc118581770)

[SignalRGroup.cs 4](#_Toc118581771)

[/Core/DB 4](#_Toc118581772)

[DataContext.cs 4](#_Toc118581773)

[Migrations and Update Database 4](#_Toc118581774)

[Migrations 4](#_Toc118581775)

[Update DB 4](#_Toc118581776)

[/Core/Dto/AutoMapper 4](#_Toc118581777)

[AutoMapperProfiles.cs 4](#_Toc118581778)

[/Core/Repositories 5](#_Toc118581779)

[MessageRepository 5](#_Toc118581780)

[MessageRespository.cs 5](#_Toc118581781)

[SignalRRepository 5](#_Toc118581782)

[ISignalRRepository.cs 5](#_Toc118581783)

[SignalRRepository.cs 5](#_Toc118581784)

[/Core/BusinessLogic 6](#_Toc118581785)

[MessageBusinessLogic 6](#_Toc118581786)

[IMessageBusinessLogic.cs 6](#_Toc118581787)

[MessageBusinessLogic.cs 6](#_Toc118581788)

[SignalRBusinessLogic 7](#_Toc118581789)

[ISignalRBusinessLogic.cs 7](#_Toc118581790)

[SignalRBusinessLogic.cs 7](#_Toc118581791)

[/Core/Extensions 8](#_Toc118581792)

[ServiceExtensions.cs 8](#_Toc118581793)

[/Core/SignalR 8](#_Toc118581794)

[PresenceTracker.cs 8](#_Toc118581795)

[Method GetConnectionsForUser 8](#_Toc118581796)

[Method UserConnected 8](#_Toc118581797)

[Method UserDisconnected 8](#_Toc118581798)

[PresenceHub.cs 9](#_Toc118581799)

[Method OnConnectedAsync 9](#_Toc118581800)

[Method OnDisconnectedAsync 9](#_Toc118581801)

[MessageHub.cs 10](#_Toc118581802)

# Basics

|  |  |
| --- | --- |
| WorkingFolder | This is still working with “Site-06-SignalR” |
| 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/DB/DataContext
2. /Core/Extensions/ServiceExtensions
3. /core/Entities/Message
4. /Core/Respositories/MessageRepository
5. /Core/Dto/AutoMapper/AutoMapperProfiles
6. /Core/SignalR/PresenceTracker
7. /Core/SignalR/MessageHub
8. /Core/Signal/PresenceHub

# New Resources

1. /Core/Entities/SignalRConnection
2. /Core/Entities/SignalRGroup
3. /Core/Repositories/ISignalRRepository
4. /Core/Repositories/SignRRepository
5. /Core/BusinessLogic/ISignalRBusinessLogic
6. /Core/BusinessLogic/SignalRBusinessLogic

# /Core/Entities

## Message.cs

Update the DateMessageSent default to DateTime.UtcNow

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

## SignalRConnection.cs

using System.ComponentModel.DataAnnotations;

namespace MSC.Api.Core.Entities;

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.Api.Core.Entities;

public class SignalRGroup

{

    /// <summary>

    /// Empty constructor is needed for the EF

    /// </summary>

    public SignalRGroup()

    {

    }

    public SignalRGroup(string groupName)

    {

        GroupName = groupName;

    }

    /// <summary>

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

    /// </summary>

    [Key]

    public string GroupName { get; set; }

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

}

# /Core/DB

## DataContext.cs

Add DBSet for the above entities we created

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

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

## Migrations and Update Database

### Migrations

Add to migrations

* dotnet ef migrations add SignalREntitiesAdded -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/Dto/AutoMapper

## AutoMapperProfiles.cs

Add a new private method to return Z with the datetime to show as UTC

    private void Map\_DateTime\_Add\_Z\_ForUtc()

    {

        CreateMap<DateTime, DateTime>().ConvertUsing(d => DateTime.SpecifyKind(d, DateTimeKind.Utc));

    }

And then add it to the constructor

        //We'll have the Z at the end of the dates meaning utc

        Map\_DateTime\_Add\_Z\_ForUtc();

# /Core/Repositories

## MessageRepository

### MessageRespository.cs

#### Method GetMessageThread

Update the unreadMessage to DateTime.UtcNow in method **GetMessageThread**

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

## SignalRRepository

### ISignalRRepository.cs

using System.Threading.Tasks;

using MSC.Api.Core.Entities;

namespace MSC.Api.Core.Repositories;

public interface ISignalRRepository

{

    void AddGroup(SignalRGroup group);

    void RemoveConnection(SignalRConnection connection);

    Task<SignalRConnection> GetConnection(string connectionId);

    Task<SignalRGroup> GetMessageGroup(string groupName);

    Task<SignalRGroup> GetGroupForConnection(string connectionId);

    Task<bool> SaveAllAsync();

}

### SignalRRepository.cs

using System.Threading.Tasks;

using AutoMapper;

using Microsoft.EntityFrameworkCore;

using MSC.Api.Core.DB;

using MSC.Api.Core.Entities;

namespace MSC.Api.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 void AddGroup(SignalRGroup group)

    {

        \_context.SignalRGroups.Add(group);

    }

    public async Task<SignalRConnection> GetConnection(string connectionId)

    {

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

        return 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> GetGroupForConnection(string connectionId)

    {

        return await \_context.SignalRGroups

                            .Include(x => x.Connections)

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

                            .FirstOrDefaultAsync();

    }

    public void RemoveConnection(SignalRConnection connection)

    {

        \_context.SignalRConnections.Remove(connection);

    }

    public async Task<bool> SaveAllAsync()

    {

        return await \_context.SaveChangesAsync() > 0;

    }

}

# /Core/BusinessLogic

## MessageBusinessLogic

### IMessageBusinessLogic.cs

Add a new overload method

    Task<BusinessResponse> AddMessage(Message message);

### MessageBusinessLogic.cs

Implement the new overload method and update the current AddMessage method

    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

        };

        var result = await AddMessage(message);

        return result;

    }

    public async Task<BusinessResponse> AddMessage(Message message)

    {

        \_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");

    }

## SignalRBusinessLogic

### ISignalRBusinessLogic.cs

using System.Threading.Tasks;

using MSC.Api.Core.Entities;

namespace MSC.Api.Core.BusinessLogic;

public interface ISignalRBusinessLogic

{

    void AddGroup(SignalRGroup group);

    void RemoveConnection(SignalRConnection connection);

    Task<SignalRConnection> GetConnection(string connectionId);

    Task<SignalRGroup> GetMessageGroup(string groupName);

    Task<SignalRGroup> GetGroupForConnection(string connectionId);

    Task<bool> SaveAllAsync();

}

### SignalRBusinessLogic.cs

using System.Threading.Tasks;

using MSC.Api.Core.Entities;

using MSC.Api.Core.Repositories;

namespace MSC.Api.Core.BusinessLogic;

public class SignalRBusinessLogic : ISignalRBusinessLogic

{

    private readonly ISignalRRepository \_srRepo;

    public SignalRBusinessLogic(ISignalRRepository srRepo)

    {

        \_srRepo = srRepo;

    }

    public void AddGroup(SignalRGroup group)

    {

        \_srRepo.AddGroup(group);

    }

    public async Task<SignalRConnection> GetConnection(string connectionId)

    {

        return await \_srRepo.GetConnection(connectionId);

    }

    public async Task<SignalRGroup> GetMessageGroup(string groupName)

    {

        return await \_srRepo.GetMessageGroup(groupName);

    }

    public async Task<SignalRGroup> GetGroupForConnection(string connectionId)

    {

        return await \_srRepo.GetGroupForConnection(connectionId);

    }

    public void RemoveConnection(SignalRConnection connection)

    {

        \_srRepo.RemoveConnection(connection);

    }

    public async Task<bool> SaveAllAsync()

    {

        return await \_srRepo.SaveAllAsync();

    }

}

# /Core/Extensions

## ServiceExtensions.cs

Add the SignalRRepository and SignalRBusinessLogic for IoC

        services.AddScoped<ISignalRRepository, SignalRRepository>();

        services.AddScoped<ISignalRBusinessLogic, SignalRBusinessLogic>();

# /Core/SignalR

## PresenceTracker.cs

### Method GetConnectionsForUser

Add a method to get all the connections for the user

    /// <summary>

    /// Get all connections for the user user in MessageHub

    /// </summary>

    /// <returns></returns>

    public Task<List<string>> GetConnectionsForUser(string userName)

    {

        List<string> connectionIds;

        lock (\_onlineUsers)

        {

            connectionIds = \_onlineUsers.GetValueOrDefault(userName);

        }

        return Task.FromResult(connectionIds);

    }

### Method UserConnected

Return a Boolean flag to return true when user goes online

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

    {

        var isOnline = false;

        lock (\_onlineUsers)

        {

            if (\_onlineUsers.ContainsKey(userName))

            {

                //add the connectionId to the second part of the dictionary

                \_onlineUsers[userName].Add(connectionId);

            }

            else

            {

                //add the the user name with the connection id

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

                //user has come online so update the flag

                isOnline = true;

            }

        }

        //return Task.CompletedTask;

        return Task.FromResult(isOnline);

    }

### Method UserDisconnected

Only return true when the user has gone offline, all connections closed

    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 connection

            \_onlineUsers[userName].Remove(connectionId);

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

            {

                //remove the user

                \_onlineUsers.Remove(userName);

                //when the user is completely removed then make isoffline true

                isOffline = true;

            }

        }

        //return Task.CompletedTask;

        return Task.FromResult(isOffline);

    }

## PresenceHub.cs

### Method OnConnectedAsync

* Rather than sending the online users to “all”, send to the “caller” only
* Implement isOnline and when true only then send UserIsOnline

    public override async Task OnConnectedAsync()

    {

        //pick the current user from Token ==> claims

        //add the user to presenceTracker

        //other than the current user tell all others that the user is online

        var userName = Context.User.GetUserName();

        var connectionId = Context.ConnectionId;

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

        if (isOnline)

            await Clients.Others.SendAsync("UserIsOnline", userName);

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

        //update: get the users online and send to the caller only

        var currentUsers = await \_tracker.GetOnlineUsers();

        //await Clients.All.SendAsync("GetOnlineUsers", currentUsers);

        await Clients.Caller.SendAsync("GetOnlineUsers", currentUsers);

    }

### Method OnDisconnectedAsync

* No need to send the list of online users
* Implement isOffline and only send when true

    public override async Task OnDisconnectedAsync(Exception exception)

    {

        //pick the current user from Token ==> claims

        //remove the user from presenceTracker

        //tell all other users when the use goes offline

        var userName = Context.User.GetUserName();

        var connectionId = Context.ConnectionId;

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

        if (isOffline)

            await Clients.Others.SendAsync("UserIsOffline", Context.User.GetUserName());

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

        //update: not sending the list

        //var currentUsers = await \_tracker.GetOnlineUsers();

        //await Clients.All.SendAsync("GetOnlineUsers", currentUsers);

        await base.OnDisconnectedAsync(exception);

    }

## MessageHub.cs

We have a quite a few updates here. Full class is provided here

using System;

using System.Linq;

using System.Net;

using System.Threading.Tasks;

using AutoMapper;

using Microsoft.AspNetCore.Authorization;

using Microsoft.AspNetCore.SignalR;

using MSC.Api.Core.BusinessLogic;

using MSC.Api.Core.Dto;

using MSC.Api.Core.Entities;

using MSC.Api.Core.Extensions;

using MSC.Api.Core.Repositories;

//no package to install for SignalR

namespace MSC.Api.Core.SignalR;

[Authorize]

/// <summary>

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

/// </summary>

public class MessageHub : Hub

{

    private readonly IMessageBusinessLogic \_msgBl;

    private readonly ISignalRBusinessLogic \_signalrBl;

    private readonly IUsersRepository \_usersRepo;

    private readonly IHubContext<PresenceHub> \_presenceHub;

    private readonly PresenceTracker \_presenceTracker;

    private readonly IMapper \_mapper;

    public MessageHub(IUsersRepository userRepo,

        IMessageBusinessLogic msgBl,

        ISignalRBusinessLogic signalrBl,

        IHubContext<PresenceHub> presenceHub,

        PresenceTracker presenceTracker,

        IMapper mapper)

    {

        \_usersRepo = userRepo;

        \_msgBl = msgBl;

        \_signalrBl = signalrBl;

        \_presenceHub = presenceHub;

        \_presenceTracker = presenceTracker;

        \_mapper = mapper;

    }

    /// <summary>

    /// Implement OnConnectedAsync

    /// </summary>

    /// <returns></returns>

    public override async Task OnConnectedAsync()

    {

        //create a group of two users. will be passing in the other users name

        var httpContext = Context.GetHttpContext();

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

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

        //build the group name

        var groupName = GetGroupName(Context.User.GetUserName(), otherUserName);

        //add to group

        await Groups.AddToGroupAsync(Context.ConnectionId, groupName);

        //persist in DB

        SignalRGroup group = await AddToGroup(groupName);

        //pass the group back

        await Clients.Group(groupName).SendAsync("UpdatedGroup", group);

        //when the user joins the group get the messages

        var messages = await \_msgBl.GetMessageThread(Context.User.GetUserId(), otherUserId);

        //send the message to the caller

        //await Clients.Group(groupName).SendAsync("ReceiveMessageThread", messages);

        await Clients.Caller.SendAsync("ReceiveMessageThread", messages);

    }

    /// <summary>

    /// Implmenent OnDisconnectedAsync

    /// </summary>

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

    /// <returns></returns>

    public override async Task OnDisconnectedAsync(Exception exception)

    {

        //remove from the DB

        SignalRGroup group = await RemoveFromMessageGroup();

        //pass the group back

        await Clients.Group(group.GroupName).SendAsync("UpdatedGroup", group);

        //users will be automatically removed from the group

        await base.OnDisconnectedAsync(exception);

    }

    //copied the code from the MessageController CreateMessage action and then updated it

    public async Task SendMessage(MessageCreateDto msg)

    {

        //get the claims

        var userClaims = Context.User.GetUserClaims();

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

            throw new HubException("User issue");

        //build message

        Message message = await BuildMessage(msg, Context, userClaims);

        if (message == null)

            throw new HubException("Message not built");

        //group info and group

        var groupName = GetGroupName(message.SenderUsername, message.ReceipientUsername);

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

        //mark the message as read

        if (group.Connections.Any(x => x.UserName == message.ReceipientUsername))

        {

            message.DateMessageRead = DateTime.UtcNow;

        }

        else

        {

            //target receipient of the message is online but not on the messages page. User is on any other page so show notification

            var connections = await \_presenceTracker.GetConnectionsForUser(message.ReceipientUsername);

            if (connections != null)

            {

                //send notification

                var user = \_mapper.Map<UserDto>(message.Sender);

                await \_presenceHub.Clients.Clients(connections).SendAsync("NewMessageReceived", user);

            }

        }

        //add message to db

        var result = await \_msgBl.AddMessage(message);

        if (result == null)

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

        if (result.HttpStatusCode != HttpStatusCode.OK)

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

        //we have the message that got added to the database in result so pick it

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

        //return the added message

        await Clients.Group(groupName).SendAsync("NewMessage", messageAdded);

    }

    private async Task<Message> BuildMessage(MessageCreateDto msg, HubCallerContext context, UserClaimGetDto userClaims)

    {

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

            throw new HubException("User issue");

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

            throw new HubException("Message info invalid");

        //sender user

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

        if (sender == null)

            throw new HubException("Logged in user not found");

        //receipient

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

        if (receipient == null)

            throw new HubException("Receipient not found");

        var message = new Message

        {

            Sender = sender,

            Receipient = receipient,

            SenderUsername = sender.UserName,

            ReceipientUsername = receipient.UserName,

            MessageContent = msg.Content

        };

        return message;

    }

    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}";

    }

    /// <summary>

    /// Add to group

    /// </summary>

    /// <param name="groupName"></param>

    /// <returns></returns>

    private async Task<SignalRGroup> AddToGroup(string groupName)

    {

        //logged in user name and id

        var userName = Context.User.GetUserName();

        var userId = Context.User.GetUserId();

        //get the group from the db

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

        //create a connection

        SignalRConnection connection = new SignalRConnection(Context.ConnectionId, userName, userId);

        //logic

        if (group == null)

        {

            group = new SignalRGroup(groupName);

            \_signalrBl.AddGroup(group);

        }

        group.Connections.Add(connection);

        //save

        if (await \_signalrBl.SaveAllAsync())

            return group;

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

    }

    /// <summary>

    /// Remove connection from Message group

    /// </summary>

    /// <param name="connectionId"></param>

    /// <returns></returns>

    private async Task<SignalRGroup> RemoveFromMessageGroup()

    {

        SignalRGroup group = await \_signalrBl.GetGroupForConnection(Context.ConnectionId);

        if (group == null)

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

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

        if (connection == null)

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

        \_signalrBl.RemoveConnection(connection);

        if (await \_signalrBl.SaveAllAsync())

            return group;

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

    }

}