Portierung von WCF zu ASP.NET Core mit SignalR

Aufgabenstellung:

Gegeben ist eine Chat-Anwendung, die mit *WCF* in .NET Framework 4.8 erstellt wurde. Diese Anwendung liegt in GitHub:

https://github.com/xdah031/Vergleich-WCF-Alternativen/tree/master/WcfChatApplication

Beispiel einer Chat Apps

Gesucht ist die Portierung zu ASP.NET Core mit SignalR

Technologien

- Server: ASP.NET Core mit SignalR
- Client: WPF mit MVVM Light

Source Code

Das Projekt liegt in GitHub:

https://github.com/xdah031/Vergleich-WCF-Alternativen/tree/master/ASPNETCoreSignalR

Es enthält drei Verzeichnisse: *SharedLibrary* für die Modelle, die von den Clients und Server benutzt werden, *WPFClient* für Client und *Server* für Server.

Anleitung:

- Führen Sie **ASPNETCoreSignalR.sln** Solution aus und dann starten Sie das gesamte Projekt oder
- Starten Sie den Server und die Clients manuell im Hauptverzeichnis, wenn Sie mehrere Clients parallel simulieren wollten.

Chat Server

SignalR und WCF sind zwei verschiedene Konzepte von Microsoft und haben daher unterschiedliche Verhalten. In SignalR konzentriert sich man auf die Echt-Zeit-Webfunktionen zur Kommunikation zwischen Server und Clients. Dafür verwendet SignalR Hub. Ein Hub ist eine Pipeline, durch die Server und Clients die Methoden untereinander aufrufen können.

Man kann auch Nachrichten außerhalb eines Hubs, z.B. in einen Controller oder Middleware, senden.

Datenbank

Im Unterschied zum WCF-Beispiel sollten die Daten in SignalR nicht direkt im Hub gespeichert werden. Deshalb wird eine Datenbank erstellt und am einfachsten ist In-Memory Datenbank durch EF Core.

Diese Datenbank referenziert die beiden Modelle Message und User.

Code:

```
public class DataContext : DbContext
    {
        public DataContext(DbContextOptions<DataContext> options) : base(options)
        { }

        public DbSet<Message> Messages { get; set; }
        public DbSet<User> Users { get; set; }
    }
```

Danach muss man die in die *Startup*-Klasse einbinden.

Code:

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<DataContext>(options =>
                     options.UseInMemoryDatabase("Messages"));
            ...
        }
```

Hubs

Zuerst wird die ChatHub.cs, die von der Hub-Klasse vererbt, erstellt. Hier werden die Kommunikation-Methoden definiert:

1. SendMessage: Wenn ein Client eine Nachricht an den Server geschickt hat, wird der Server diese an alle Clients weiterleiten.
2. Join: Beim Login wird neu Client zur Datenbank hinzugefügt. Die Userliste wird auch in Client-Seite aktualisiert.
3. OnConnectedAsync: Das Verhalten, wenn neu Client teilgenommen hat. Hier lädt der Client die Chat-Historie unter.
4. OnDisconnectedAsync: Das Verhalten, wenn neu Client ausgeloggt hat. Dabei wird der Client aus der Datenbank entfernt.

Der Server sendet Nachrichten wieder an Clients durch das Client-Objekt. Die meisten benutzten Methoden davon sind *All*, *Caller*, *Other* und *Group*. Der Unterschied zwischen diesen Methoden liegt daran, an welchen Client die Nachricht gesendet wird. Im Vergleich zu gRPC unterstützt SignalR alle komplexen Objekte und Datenstrukturen in den Parametern und Rückgabewerten.

Darüber hinaus sollte async/await verwendet werden.

Code:

```
        public async Task SendMessage(string user, string message)
        {
            var newMessage = new Message(user, message, DateTime.Now);
            await _context.Messages.AddAsync(newMessage);
            await _context.SaveChangesAsync();
            await Clients.All.SendAsync("ReceiveMessage", newMessage);
        }
```

Zum Beispiel in der SendMessage-Methode, wenn der Client eine Nachricht senden wollte, ruft er die Methode mit seinem Usernamen und die Nachricht auf. Alle Clients müssen auf eine Invoke-Methode warten, die „ReceiveMessage“ hören kann, um neue Nachricht zu bekommen.

Danach wird der Hub in der Startup-Klasse konfiguriert. Zum einen ist SignalR-Service durch den Aufruf *services.AddSignalR()* in ConfigureServices hinzugefügt, zum anderen wird ein neuer Endpunkt durch endpoints.*MapHub<ChatHub>("/chatHub")* für diesen Dienst eingestellt. Die Clients werden diese Adresse nutzen.

Chat Client

In der Client-Seite werden die Verbindung mit dem Server und die Invoke-Methoden erstellt. Der Client benutzt das NuGet-Paket „Microsoft.AspNetCore.SignalR.Client“. Die meisten Methoden in SignalR werden zum Zweck der Echtzeit-Verarbeitung asynchron mit *await* verwendet.

Verbindung

Der Client stellt zuerst eine HubConnection mit dem Server-Dienst „ChatHub“ ein. Berücksichtigt werden muss ist *AutomaticReconnect*. Wenn wir das einstellen wollten, müssen wir entweder die erweiterte Methode „WithAutomaticReconnect“ nutzen oder selbst in einem *Closed*-Handler handeln.

Beim manuellen Wiederherstellen können wir Code-Logik hinzufügen. Am einfachsten wird der Fehler durch *MessageBox* angezeigt. Außerdem sollte dich eine zufällige Verzögerung einstellen lassen, bevor neue Verbindung beginnt, damit der Server nicht überladen würde.

Danach könnte der Client *await connection.StartAsync()* bzw. *connection.StopAsync()* aufrufen, um die Verbindung zu starten bzw. zu beenden.

Code:

```
HubConnection connection;
...
// Setup Server Address
connection = new HubConnectionBuilder()
    .WithUrl("https://localhost:5001/ChatHub")
    .WithAutomaticReconnect()
    .Build();

// Reconnect if the connection undesirable closed
connection.Closed += async (error) =>
{
    MessageBox.Show(error.Message);
    await Task.Delay(new Random().Next(0, 5) * 1000);
    await connection.StartAsync();
};
```

Invoke-Methoden

Für die Kommunikation muss man zuerst den Handler, der beim Aufruf einer festgelegten Methode aufgerufen wird, erstellt. In WPF muss der CollectionView-Typ *ListBox* von einem Dispatcher-Thread aus aufgerufen werden. Für die Typen wie *Label* oder *TextBox* ist hingegen unnötig.

Danach kann man die in Hub definierten Methoden durch *connection.InvokeAsync* aufrufen. In dieser Methode kann man zusätzlich ein CancellationToken mit try/catch hinzufügen und Fehler manuell behandeln.

Außerdem unterstützt SignalR auch Streaming (C# 8.0), das in diesem Beispiel nicht verwendet wird.

Code:

```
connection.On<Message>("ReceiveMessage", (message) =>
{
    Application.Current.Dispatcher.InvokeAsync(() =>
    {
        Messages.Add(message);
    });
});
...
await connection.InvokeAsync("SendMessage", Username, ChatText);
```

Zusammenfassung

Mit SignalR kann man viel Codeaufwand sparen, den beim RAW-WebSockets geschrieben werden muss, deshalb ist es einfach zu erlernen und implementieren. Es bietet zusätzlich Transportfallback (ausgenommen ist Java-Client), das WebSockets nicht unterstützt. In meisten Fälle sollte SignalR anstatt WebSockets verwendet werden.

Der Migrationsaufwand von WCF zu ASP.Net Core mit SignalR ist natürlich sehr hoch, obwohl es die WCF-Features gut realisiert. Im Vergleich mit gRPC finde ich, dass der Migrationsaufwand des gRPC nicht geringer als der des SignalR ist, obwohl gRPC ein ähnliches Konzept wie WCF besitzt. Das liegt daran, dass SignalR eine Technologie von Microsoft ist und gRPC dagegen Google. Die Kompatibilität des SignalR ist sicherlich besser.

gRPC konzentriert auf die Performance, so dass SignalR keinen Leistungsvorteil hat. Für One-Way-Service ist Basic Web-APIs auch effizienter. Trotzdem ist die Leistung des SignalR nicht gering.

Zusammengefasst ist zu sagen, dass SignalR eine gute Alternative von WCF ist, wenn wir neue Anwendungen bauen wollten, genauso wie gRPC. Wenn man SignalR für Duplex-Services mit Basic Web-APIs für One-Way-Services kombinieren würde, dann könnte man die Leistung der Anwendung deutlich verbessert.

Referenz:

[1]: Einführung in ASP.NET Core SignalR,

https://docs.microsoft.com/de-de/aspnet/core/signalr/introduction?view=aspnetcore-3.1

[2]: Life after WCF, Mark Rendle,

https://unwcf.com/posts/life-after-wcf/