Alternativen zum Windows Communication Framework (WCF)

Erstellen einer einfachen Chat-Anwendung mit ASP.NET Core und SoapCore

Beschreibung der Zielanwendung

Die Clients sind WPF-Fenster (siehe Bild). Startet man das Programm wird man zunächst nach einem Namen gefragt. Danach öffnet sich das eigentliche Hauptfenster. In diesem kann man unten einen Text eingeben. Beim Klicken auf „Senden“ wird dieser Text und der Name des Nutzers an den Server geschickt. Der Server sendet ihn außerdem mit Zeitstempel an alle Clients und speichert ihn intern in einem Chatverlauf. Beim ersten Anmelden bekommt ein Client außerdem den kompletten Chatverlauf seit dem Neustart des Servers (wir brauchen keine Datenbanken oder irgendwas, nur eine Liste mit Chatbeiträgen). Weiterhin wird in einem Chatfenster auch immer angezeigt wie viele Personen aktuell für den Chat angemeldet sind. Schließt jemand das Programm wird er automatisch abgemeldet und verschwindet aus der Liste.

Potenzielles Aussehen der Zielanwendung

Eingesetzte Technologien

- Plattform: Microsoft ASP.NET Core 3.0 auf .NET Core 3.1
- Client: WPF-Anwendung mit MVVM Light
- Server: ASP.NET-Konsolenanwendung mit SoapCore

Quellcode

Der gesamte Quellcode ist auf GitHub öffentlich verfügbar: https://github.com/benjamin-hempel/Vergleich-WCF-Alternativen/tree/master/SoapCoreChatApplication

Die Solution besteht aus zwei Verzeichnissen: WPFClient für den Client und SoapCoreServer für den Server. Genaueres zum Quellcode wird auf den folgenden Seiten erläutert. Leider konnte die Anwendung mit SoapCore nicht lauffähig gemacht werden. Die Begründung hierfür ist im Abschnitt „Begründung für das Scheitern der Umsetzung“ zu finden.

Das Service Model im Detail

Das Service Model setzt sich aus drei Komponenten zusammen:

1. Das Interface IChatService, welches alle Methoden enthält, bei denen der Client Daten an den Server übermittelt und ggf. Daten vom Server anfragt.
2. Das Interface IChatCallback, welches alle Methoden enthält, bei denen der Server unabhängig Daten an alle Clients übermittelt.
3. Die Klasse Message, welche eine einzelne Chat-Nachricht mit ihren Attributen modelliert.

Alle befinden sich im zwischen Client und Server geteilten Namespace SoapCoreChatApplication.Contract.

Das Interface IChatService

Das Interface IChatService deklariert die folgenden Methoden, welche auf der Server-Seite implementiert werden müssen und im Client aufgerufen werden:

- void SendChat(string name, string text): Übermittelt die Nachricht vom User „name“ mit dem Inhalt „text“ an den Server und hat keinen Rückgabewert.
- void Join(string name): Teilt dem Server mit, dass der User „name“ dem Chat beitritt und hat keinen Rückgabewert.
- void LogOut(string name): Teilt dem Server mit , dass der User „name“ den Chat verlässt und hat keinen Rückgabewert.
- List<Message> GetChats(): Der Client fragt beim Server eine Liste aller gesendeter Nachrichten (seit dem letzten Neustart des Servers) an. Gibt dem Client diese Liste zurück.
- void Refresh(): Löst aus, dass der Server allen Clients die aktuelle User-Liste übermittelt und hat keinen Rückgabewert.

```
[ServiceContract(CallbackContract = typeof(IChatCallback), SessionMode =
       SessionMode.Required)]
```

Außerdem wird hier der Service Contract definiert. Die Callbacks sind dabei im Interface IChatCallback deklariert, außerdem werden Sessions für die Erfüllung des Vertrages vorausgesetzt.

```
[OperationContract(IsOneWay = true)]
```

Jede im Interface deklarierte Methode wird außerdem als (ggf. einseitige) Operation definiert. Einseitig bedeutet dabei, dass die Methode vom Server keine Daten zurückerwartet (void).

Das Interface IChatCallback

Das Interface IChatCallback deklariert die folgenden Methoden, welche auf der Client-Seite implementiert werden müssen und im Server aufgerufen werden:

- void ReceiveChat(Message message): Sendet die Nachricht „message“ an den Client und hat keinen Rückgabewert.
- void RefreshUserList(List<string> userList): Sendet die aktuelle User-Liste (userList) an den Client und hat keinen Rückgabewert.

Auch hier werden die einzelnen Methoden, wie in IChatService, als (ggf. einseitige) Operation definiert.

Die Klasse Message

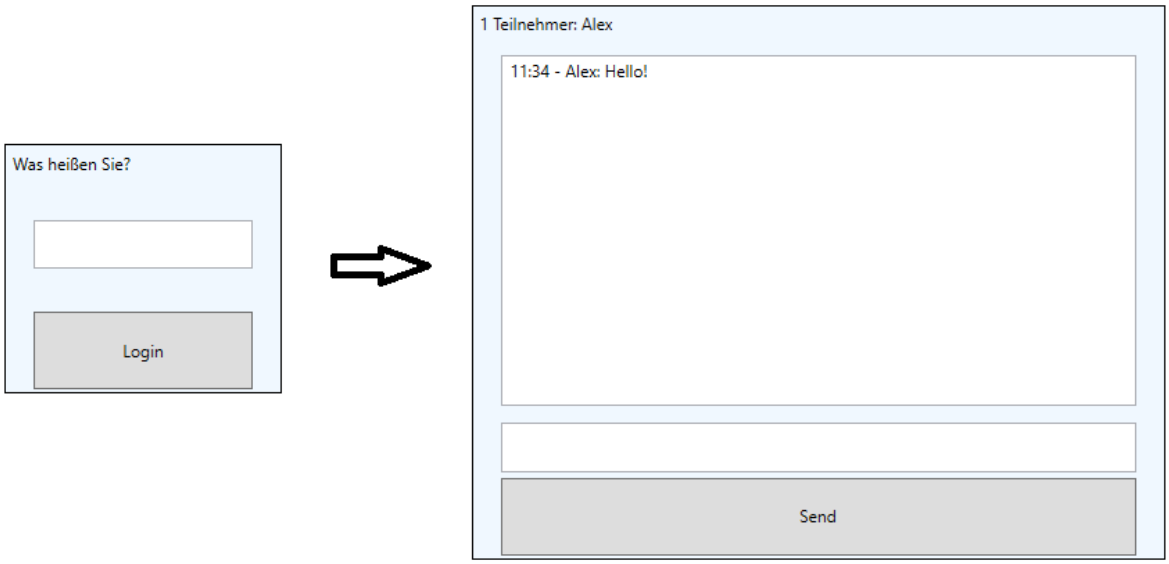
Die Klasse Message modelliert eine einzelne Chatnachricht und besitzt folgende Attribute, für welche jeweils einfache get- und set-Methoden definiert werden:

- string user: Der User, von welchem die Nachricht gesendet wurde. Standardwert „Anonymous“.
- string text: Der Inhalt der Nachricht. Kein Standardwert.
- DateTime time: Der Zeitpunkt, zu welchem die Nachricht gesendet wurde. Standardwert ist der minimal definierbare Zeitpunkt (DateTime.MinValue).

Die Klasse Message wird außerdem als DataContract definiert. Das bedeutet, dass sowohl Client als auch Server diese Klasse implementiert haben und mit dieser gleich umgehen können. Die einzelnen Attribute der Klasse werden weiterhin als DataMember definiert.

Der Client im Detail

Das Hauptfenster der Anwendung, auch MainWindow genannt, setzt sich aus zwei User Controls zusammen: LoginUC und ChatViewUC. Zunächst wird nur die Eingabe für den Namen geöffnet, das eigentliche Chatfenster bleibt verborgen. Sobald ein Name eingetragen und auf „Login“ geklickt wurde, wird das Eingabefeld verborgen und das Chatfenster angezeigt.



Reihenfolge und Aussehen der User Controls

MVVMLight integriert das View-Model-Prinzip für WPF, welches Benutzeroberfläche und Programmlogik trennt. Es existieren zwei Klassen: MainViewModel enthält die Interaktionslogik für die gesamte Benutzeroberfläche, während ViewModelLocator die Verbindung mit der Oberfläche unterhält.

Anlegen der Verbindung zum Server

```
static readonly DuplexChannelFactory<IChatService> factory =
       new DuplexChannelFactory<IChatService>(new InstanceContext(new
       ChatCallback()), new NetHttpBinding(), new
       EndpointAddress("http://localhost:8000/nethttp"));
readonly IChatService server = factory.CreateChannel();
```

Unter Nutzung der DuplexChannelFactory, welche von System.ServiceModel bereitgestellt wird, wird die Verbindung mit folgenden Parametern konfiguriert:

- die Verbindung nutzt das Interface IChatService,
- die Implementation für das Callback-Interface befindet sich in der Klasse ChatCallback,
- als Binding für die Verbindung wird NetHttpBinding verwendet,
- der Server ist unter der Adresse http://localhost:8000/nethttp zu finden.

Anschließend wird der eigentliche Kommunikationskanal erstellt, welcher die eben beschriebene Konfiguration nutzt.

Konstruktor MainViewModel

```
public MainViewModel()
        {
            LoginVisibility = Visibility.Visible;
            ChatViewVisibility = Visibility.Collapsed;

            LoginCommand = new RelayCommand(LoginMethod);
            SendCommand = new RelayCommand(SendMethod);

            if (IsInDesignMode)
            {
                WindowTitle = "Chat Application (Design)";
            }
            else
            {
                WindowTitle = "Chat Application";
            }

            Application.Current.MainWindow.Closing += new
                CancelEventHandler(MainWindow_Closing);
        }
```

Zunächst wird das Login-Fenster mit der Namenseingabe als sichtbar gesetzt und das Chatfenster verborgen. Anschließend wird die Funktionalität für Login- und Senden-Button sowie das Closing-Event hinzugefügt. Sobald der User das Chatfenster schließt, wird er automatisch ausgeloggt.

Der Server im Detail

Der Server setzt sich, neben dem oben beschriebenen Service Model, aus drei Bestandteilen zusammen:

1. Der Klasse Program, welche die die Main-Funktion enthält.
2. Der Klasse Startup, welche die Parameter und das Verhalten für das Starten des Servers beinhaltet.
3. Der Klasse ChatService, welche das Interface IChatService implementiert.

Die Klasse Program

```
public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseUrls("http://localhost:8000")
                .UseStartup<Startup>()
                .Build();
            host.Run();
        }
```

In der Main-Funktion wird der Server mit folgenden Parametern erstellt und ausgeführt:

- es wird der Kestrel-Webserver verwendet,
- der Server ist unter der Adresse http://localhost:8000 erreichbar,
- die Klasse Startup enthält die Konfiguration zum Starten des Servers.

Die Klasse Startup

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddSoapCore();
            services.TryAddSingleton<IChatService, ChatService>();
            services.AddMvc();
        }
```

Zunächst werden die vom Server verwendeten Services gesetzt. Als erstes wird hier SoapCore hinzugefügt. Anschließend werden der im Interface IChatService definierte und in der Klasse ChatService implementierte Chat-Service und MVC hinzugefügt.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
       ILoggerFactory loggerFactory)
       {
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.UseSoapEndpoint<IChatService>("/nethttp", new
                NetHttpBinding(), SoapSerializer.DataContractSerializer);
            });

            app.UseWebSockets(new WebSocketOptions
            {
                KeepAliveInterval = TimeSpan.FromSeconds(120),
                ReceiveBufferSize = 4 * 1024
            });
        }
```

In dieser Funktion wird der Server konfiguriert. Er verwendet das von ASP.NET bereitgestellte Routing und es wird ein SoapCore-Endpunkt auf der Adresse „/nethttp“ hinzugefügt, welcher (theoretisch) das NetHttpBinding und den DataContractSerializer zum Serialisieren der Daten verwendet.

Die Klasse ChatService

Die Implementation der Klasse ChatService ist weitestgehend selbsterklärend. Es werden zwei statische Listen für alle Nachrichten seit dem letzten Neustart sowie für die aktuell angemeldeten User gehalten. Es gibt jedoch eine Besonderheit:

```
Dictionary<IChatCallback, string> users = new Dictionary<IChatCallback, string>();
```

Es wird ein Dictionary angelegt, in welchem die Callback-Kanäle der Clients den Usernamen zugeordnet werden.

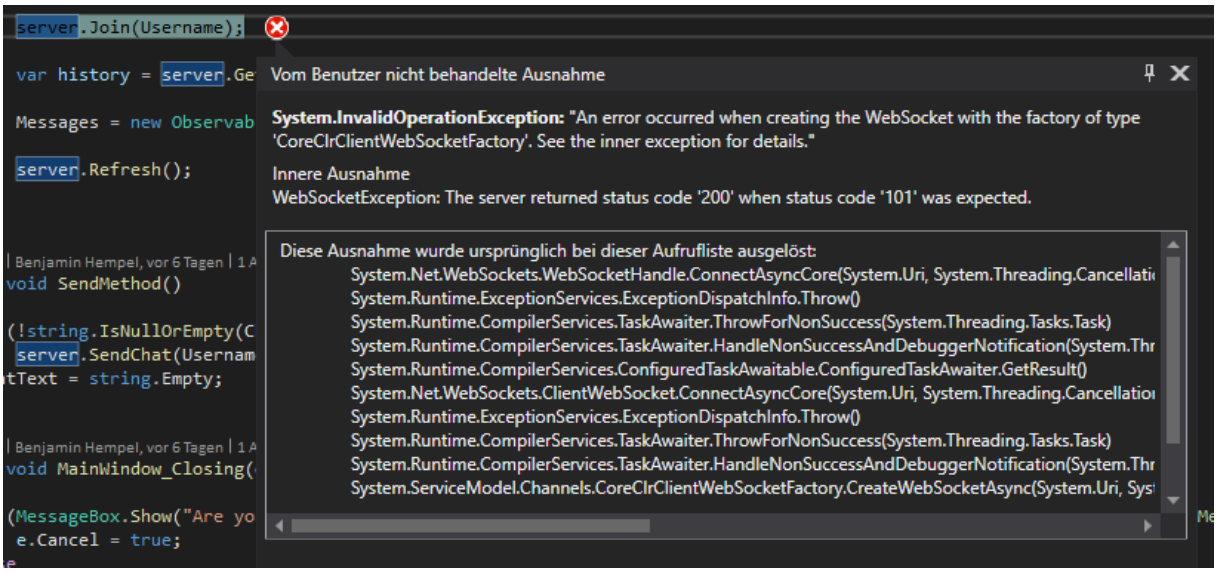
```
var connection = OperationContext.Current.GetCallbackChannel<IChatCallback>();
```

In jeder Methode wird dann zu Beginn der Callback-Kanal des aufrufenden Clients geholt. Dadurch wird z.B. der Broadcast neuer Nachrichten an alle anderen User realisiert.

Begründung für das Scheitern der Umsetzung

Für die Umsetzung einer brauchbaren Chat-Anwendung werden Sessions und Duplex-Verträge benötigt. Das einzige derzeit hierfür nutzbare Binding ist NetHttpBinding, welches WebSockets zur Realisierung dieser Funktionen verwendet.

Aufgrund der quasi nicht vorhandenen Dokumentation von SoapCore, welche nur aus der Schnellstart-Anleitung auf GitHub (https://github.com/DigDes/SoapCore) und einer Beispiel-Anwendung besteht, konnte die Anwendung nicht lauffähig gemacht werden. Es wird stets nur das BasicHttpBinding, welches ausschließlich Anfrage-Antwort-Szenarien realisieren kann, gezeigt bzw. verwendet. Deshalb wird vermutet, dass SoapCore als Middleware WebSockets nicht unterstützt und deshalb eine Code-200- („OK“) statt einer Code-101-Antwort („Changing protocols“) an den Client zurückgibt.



Die Exception, die vom Client geworfen wird, da SoapCore offensichtlich WebSockets nicht unterstützt, obwohl diese explizit in der Server-Anwendung aktiviert wurden.

Es wäre zwar möglich gewesen, die Chat-Anwendung entsprechend anzupassen, dies bietet jedoch aus folgenden Gründen keinen Mehrwert für die Beurteilung der verschiedenen WCF-Alternativen:

- Die konzeptionelle Umsetzung der Chat-Anwendung mittels Anfrage-Antwort-Schema wurde bereits unter Verwendung von IPC Service Framework gezeigt.
- SoapCore bietet damit, außer dem WCF-ähnlichen Konzept und der Nutzung von SOAP, keinen nennenswerten Mehrwert gegenüber IPC Service Framework.

Bewertung und Vergleich

Vorteile

- SoapCore nutzt das Service-Model, welches auch von WCF verwendet wird. Deshalb können alle nicht-WCF-spezifischen Implementierungen, sofern sie einfache Anfrage-Antwort-Szenarien umsetzen, vermutlich ohne jegliche Änderungen weiterverwendet werden.
- Durch die Nähe der Funktionsweise zu WCF fällt der Migrations- und Einarbeitungsaufwand vergleichsweise gering aus.

Nachteile

- Mit SoapCore lassen sich (höchstwahrscheinlich) keine Anwendungen realisieren, welche Duplex-Verträge oder Sessions benötigen.
- Das Projekt ist so gut wie gar nicht dokumentiert. Es existieren lediglich die Schnellstart-Anleitung und ein Beispielprogramm im GitHub-Repository des Entwicklers (https://github.com/DigDes/SoapCore)
- Der langfristige Einsatz von SoapCore ist mit großer Unsicherheit verbunden, da es nur von einer Person entwickelt wird. Es werden zwar regelmäßig Pull Requests von anderen Entwicklern, die sich am Code beteiligen, integriert, allerdings steht hinter SoapCore keine große Community, welche das Projekt gegebenenfalls weiterentwickeln könnte.

Fazit

Für einfache Anwendungen, welche das Anfrage-Antwort-Schema verwenden, ist SoapCore aufgrund seiner konzeptionellen Nähe zu WCF technisch gut als Alternative geeignet. Der Migrations- und Einarbeitungsaufwand fällt dadurch sehr gering aus.

Für komplexere Szenarien kann es jedoch nicht verwendet werden, da offensichtlich die Unterstützung für die benötigten Technologien, wie Duplex oder Sessions, fehlt. **Die quasi nicht vorhandene Dokumentation und ungewisse Zukunft des Projekts machen SoapCore daher letztendlich ungeeignet für den Einsatz in Produktivumgebungen**.