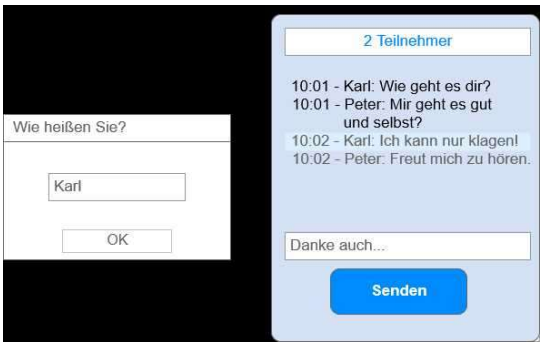
Ein Grpc Chat Programm

Aufgabenstellung

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.



Beispiel einer Chat Apps

Technologien

- Server: Grpc in .NET Core 3.1
- Client: WPF mit MVVM light

Source Code

Das Projekt liegt in GitHub:

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

Es enthält zwei Verzeichnisse: *WpfClient* für Client und *GrpcServer* für Server.

Anleitung:

- Führen Sie **GrpcServer.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

In Grpc werden die Services in Proto-Datei definiert. Server und Clients kommunizieren durch Nachrichten miteinander.

Proto

Code:

```
service Chat {
	rpc Join (User) returns (stream MessageModel);
	rpc Send (MessageModel) returns (google.protobuf.Empty);
	rpc LogOut (User) returns (google.protobuf.Empty);
	rpc GetUserlist (google.protobuf.Empty) returns (stream Userlist);
}

message User {
	string name = 1;
}

message Userlist {
	repeated User user = 1;
}

message MessageModel {
	string user = 1;
	string text = 2;
	google.protobuf.Timestamp time = 3;
}
```

Im Code gibt es insgesamt drei Datenstrukturen von Nachrichten und vier Methode, die in ChatService.cs erstellt sind.

ChatService

Code:

```
public class ChatService : Chat.ChatBase
    {
        private static List<MessageModel> history = new List<MessageModel>();
        private static Userlist users = new Userlist();
        private static readonly TimeSpan Interval = TimeSpan.FromSeconds(10);
        public ChatService()
        {
        }

        public override async Task Join(User request,
            IServerStreamWriter<MessageModel> responseStream,
            ServerCallContext context)
        {
            var token = context.CancellationToken;
            users.User.Add(request);
            Console.WriteLine("User {0} is connected!", request);
            int i = 0;
            while (!token.IsCancellationRequested)
            {
                if (i < history.Count)
                    await responseStream.WriteAsync(history[i++]);
            }
        }
```

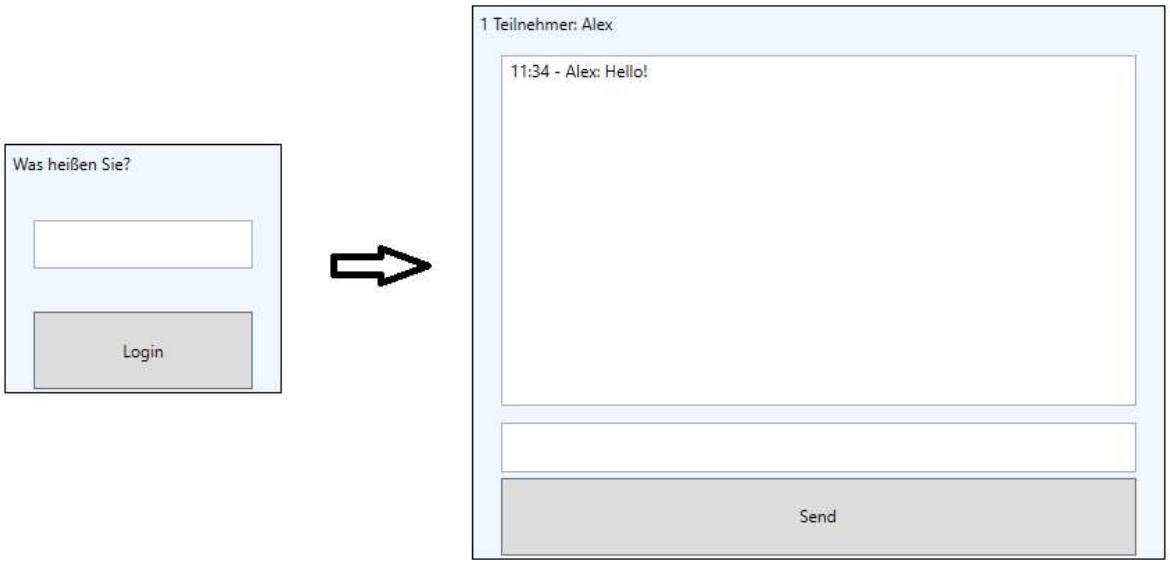
```
        public override Task<Empty> Send(MessageModel message, ServerCallContext
context)
        {
            message.Time = Timestamp.FromDateTimeOffset(DateTimeOffset.UtcNow);
            Console.WriteLine("Message {0} is sended!", message);
            history.Add(message);
            return Task.FromResult(new Empty());
        }

        public override Task<Empty> LogOut(User request, ServerCallContext context)
        {
            users.User.Remove(request);
            Console.WriteLine("User {0} is disconnected!", request);
            return Task.FromResult(new Empty());
        }
        public override async Task GetUserlist(Empty _,
            IServerStreamWriter<Userlist> responseStream,
            ServerCallContext context)
        {
            var token = context.CancellationToken;
            int i = -1;
            while (!token.IsCancellationRequested)
            {
                if (i != users.GetHashCode())
                {
                    await responseStream.WriteAsync(users);
                    i = users.GetHashCode();
                }
                // Userlist refresh every 10 seconds
                await Task.Delay(Interval, token);
            }
        }
    }
```

Hier definieren wir das Verhalten vom Service. Weil wir keine Datenbank verwenden wollten, werden zwei Liste von dem Chatverlauf und Clients erzeugt. Im Unterschied zum WCF-Server hat Grpc keine *Callback*-Funktion, durch die Nachrichten von Server an Clients gesendet werden können, sondern wird *Stream* angewendet, indem Server eine Reihenfolge von Nachrichten an Client senden wollte oder umgekehrt. Die Funktionen mit Stream werden nicht enden, solange das Stream noch offen ist.

Um die Nachrichten vom Server an Clients zu senden, wird ein Stream zu jedem Client geöffnet. Immer wenn der Server eine neue Nachricht bekommt, dann wird dies Stream auch „aktualisiert“, damit die Clients die neuen Nachrichten gelesen können.

Chat Client

Die Application-Oberfläche wird als MainWindow genannt. Sie besteht aus zwei UserControls, LoginUC und ChatViewUC. Das Chatfenster wird angezeigt, nach dem wir den Name-TextBox ausfüllen und „Login“-Button klicken.



User Control

ViewModel

Durch MVVM light werden zwei Klasse erzeugt. *MainViewModel* Klasse enthält alle Hauptfunktionen für die Oberfläche, und ViewModelLocator besitzt die Verbindung mit der Anwendung.

Verbindung mit Server

Code:

```
static Channel channel = new Channel("127.0.0.1:50051",ChannelCredentials.Insecure);
readonly Chat.ChatClient client = new Chat.ChatClient(channel);
```

Zuerst verwenden wir die generierte Clientklasse zum Aufrufen des Diensts. Wenn wir die Methoden vom Server aufrufen wollten, benutzen wir „client.‘*Methode_Name*‘(‚*Anfrage-Nachricht*‘)“

Initialisierung

Code:

```
        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);
        }
```

Amfang wird das Login-Fenster angezeigt und ChatView-Fenster hingegen versteckt. Dann werden die Funktionalitäten zu Login-, Send-Button und Closing-Event hinzugefügt. Der Client wird beim Fenster-schließen ausgeloggt.

Login-Funktion

Code:

```
        public void LoginMethod()
        {
            if (!string.IsNullOrEmpty(Username))
            {
                LoginVisibility = Visibility.Collapsed;
                ChatViewVisibility = Visibility.Visible;

                var request = new User { Name = Username };
                Task[] tasks = new Task[2];

                var replies = client.Join(request);
                tasks[0] = ListenAsync(replies.ResponseStream, tokenSource.Token);

                var list = client.GetUserlist(
                     new Google.Protobuf.WellKnownTypes.Empty());
                tasks[1] = ListenUserlistAsync(
                     list.ResponseStream, tokenSource.Token);
            }
        }

        public void SendMethod()
        {
            if (!string.IsNullOrEmpty(ChatText))
            {
                var message = new MessageModel { User = Username, Text = ChatText };
                _ = client.Send(message);
            }
            ChatText = string.Empty;
        }
```

In dieser Login-Funktion wird der *Username* in Server angemeldet und der Client bekommt dadurch zwei Streams für den Chatverlauf und User-Liste. Um diese Streams zu lesen, werden zwei Hilfsfunktionen „ListenAsync“ und „ListenUserlistAsync“ erstellt.

In *ListenAsync* werden das Stream Schritt für Schritt mit *foreach* gelesen, danach werden die Nachrichten in der Variable *Messages* gespeichert, die mit View verbunden ist. Diese Funktion ist auch nicht beendet. Um das Stream zu enden, benutzen wir ein *CancellationToken* bzw. *try-catch*-Methode. Die Funktion *ListenUserlistAsync* ist ähnlich.

Dazu braucht wir eine externe Funktion *AsAsyncEnumerable* für *IAsyncStreamReader*, um await foreach verwenden zu können.

Mit *SendMethod()* sendet der Client die Nachricht am Server. Server wird danach diese Nachricht in Stream schreiben.

Code:

```
        private async Task ListenAsync(IAsyncStreamReader<MessageModel> stream,
                CancellationToken token)
        {
            try
            {
                await foreach (var m in stream.AsAsyncEnumerable(token))
                {
                    Messages.Add(new Message(m.User, m.Text, m.Time.ToDateTime()));
                }
            }
            catch (RpcException e) when (e.StatusCode == StatusCode.Cancelled)
            {
                return;
            }
            catch (OperationCanceledException)
            {
                return;
            }
        }
```

PropertyChanged.Fody

Fody ist ein *Nuget*-Paket, mit dem wir *Event* für *PropertyChanged* nicht selbst definieren müssen. Die Event-Handler werden automatisch in Backend erzeugt, damit der Code sauber und sichtbar wird.

Code:

```
        public string WindowTitle { get; set; }
```

Anstatt:

```
        string windowTitle;
        public string WindowTitle
        {
            get { return windowTitle; }
            set
            {
                windowTitle = value;
                RaisePropertyChanged("WindowTitle");
            }

        }
```

IDataErrorInfo

IDataErrorInfo ist eine Schnittstelle, mit der die Errors behandelt werden können. In diesem Beispiel wird eine einfache Check-Methode, indem der Username geprüft wird, ob er leer ist.

Code:

```
        public string Error => string.Empty;

        public string this[string propertyName]
        {
            get
            {
                switch (propertyName)
                {
                    case nameof(Username):
                        if (Username == "")
                            return "Username is required!";
                        break;
                }
                return string.Empty;
            }
        }
```

Zusammenfassung

Diese einfache Chat Anwendung ist nur ein Beispiel mit dem Ziel, dass man die Verwendung von Grpc in C# besser verstehen kann. Deshalb konzentriere ich nicht so stark auf die Ausnahmebehandlung und Optimierung. Im Gegenstand zum WCF-Server ist zu erkennen, dass die Funktionen umgebaut werden müssen, besonders für Duplex-Service, obwohl die Funktionalität gleich ist. Deshalb ist die Portierung von WCF zu Grpc möglich, aber aufwändig. Für die großen WCF-Anwendungen in .NET Framework ist andere Lösung zu finden.

Weil die Apps auf meiner Lernerfahrung basieren, freue ich mich auf ihre Vorschläge, Korrekturen und Kommentare, wofür die Anwendung verbessert wird.