Portierung von WCF

zu ASP.NET Core mit Web-API

# 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>

Ein Bild, das Screenshot enthält.

Automatisch generierte Beschreibung

Beispiel einer Chat Apps

Gesucht ist die Portierung zu ASP.NET Core mit Web-API

# Technologien

* Server: ASP.NET Core mit Web-API
* Client: WPF mit MVVM Light

# Source Code

Das Projekt liegt in GitHub:

<https://github.com/xdah031/Vergleich-WCF-Alternativen/tree/master/ASPNETCoreWebAPI>

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 **ASPNETCoreWebAPI.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

Web-API und WCF sind zwei verschiedene Konzepte von Microsoft und daher haben daher unterschiedliche Verhalten. In Web-API konzentriert sich man auf das Request-Response-Verfahren zur Abfrage von Daten von einem Server. Das bedeutet, dass die Code-Logik in Client gelegen wird. In Server werden nur die Controller und Datenbank konfiguriert.

## Datenbank

Im Unterschied zu WCF sollten die Daten in Web-API nicht direkt im Controller behandelt 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"));  services.AddControllers();  } |

## Controllers

Die Abfragen von Clients werden in Controller behandelt. Die Typen sind GET, POST, PUT und DELETE.

Client soll die gesamte Chat-Historie abfragen und Messages senden, also GET und POST Request. Diese werden in *MessagesController* definiert und Daten werden von der Datenbank zurückgegeben.

Außerdem sind Verhalten von Clients in *UsersController* zu betrachten:

1. Login: User wird zur Datenbank hinzugefügt – POST.
2. In Chat: User fragt die Informationen über andere User ab – GET.
3. Logout: User wird von Datenbank aus entfernt – DELETE.

Code:

|  |
| --- |
| // GET: api/Users  [HttpGet]  public List<User> Get()  {  return \_context.Users.ToList();  }  // POST: api/Users  [HttpPost]  public async Task Post([FromBody] string value)  {  var user = new User(value);  await \_context.Users.AddAsync(user);  await \_context.SaveChangesAsync();  } |

# Chat Client

Client können Web-APIs vom Server mit HttpClient aufrufen. Dazu brauchen sie die Server-Adresse und hier wird JSON verwendet.

Code:

|  |
| --- |
| static HttpClient client = new HttpClient();  …  // Setup Server Address  client.BaseAddress = new Uri("https://localhost:5001");  MediaTypeWithQualityHeaderValue contentType =  new MediaTypeWithQualityHeaderValue("application/json");  client.DefaultRequestHeaders.Accept.Add(contentType); |

Danach kann der Client die Abfragen senden, indem GetAsync, PostAsync, PutAsync und DeleteAsync aufgerufen werden, je nachdem, welche Daten der Client nehmen wollte.

Code:

|  |
| --- |
| public async Task GetMessagesAsync()  {  while (true)  {  var response = client.GetAsync("/api/Messages").Result;  if (response.IsSuccessStatusCode)  {  var data = response.Content.ReadAsStringAsync().Result;  var history = JsonConvert.DeserializeObject<List<Message>>(data);  Messages = new ObservableCollection<Message>(history);  }  // Refresh Chat List every 0.5 second  await Task.Delay(TimeSpan.FromSeconds(0.5), CancellationToken.None);  }  } |

Weil Web-API kein Duplex-Service unterstützt, müssen die Client regelmäßig durch *Task.Delay* die Daten vom Server abfragen und aktualisieren.

# Zusammenfassung

*Web-API* bietet die Lösung von One-Way-Services *WCF*. Obwohl die Migration aufwändig ist, ist die Code-Logik einfacher als z.B. gRPC. Aber wenn man Duplex-Services portieren wollte, ist das nicht die gute Lösung. Dafür hat Microsoft vorgeschlagen, dass sich *WebSockets* mit *SignalR* benutzen lässt.

## Referenz:

[1]: Tutorial: Erstellen einer Web-API mit ASP.NET Core,

<https://docs.microsoft.com/de-de/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio>