gRPC

Para podermos usar o gRPC no nosso projeto, e gerar os respetivos protos, precisamos de instalar o NuGet package Grpc.AspNetCore no lado do server e o Grpc.Tools e o Grpc.Net.Client do lado do cliente.

Para o Grpc.Tools (lado do cliente) podemos ter uma flag PrivateAssets pois o tooling package não é requerido em runtime.

```
<PackageReference Include="Grpc.Tools" Version="2.23.0" PrivateAssets="All" />
```

Proto file

O gRPC usa uma abordagem contract-first para o desenvolvimento de APIs através da utilização de Protocol Buffers. Os Protocol Buffers (protobuf) são usados como Interface Design Language (IDL) por default. Os ficheiros .proto contêm:

- A definição dos serviços gRPC
- As mensagens enviadas entre clientes e servers

Abaixo podemos ver o exemplo de um proto.

- A keyword "message" serve para definir estruturas de dados
- A keyword "service" serve para definir serviços.
- A keyword "rpc" service para definir a função de um service

```
syntax = "proto3";

import "google/protobuf/empty.proto";

option csharp_namespace = "Primavera.Lithium.Sample.WebApi.CustomCode";

package alive;

service Alive {
    rpc Sum (SumRequest) returns (SumReply);
}

message SumRequest {
    int32 number1 = 1;
    int32 number2 = 2;
}

message SumReply {
    int32 message = 1;
}
```

Neste caso estamos a definir um serviço chamado Alive que invoca o método Sum. Já o Sum envia um request do tipo SumRequest e recebe um reply to tipo SumReply.

Server

O ficheiro .proto é incluido no projeto ao adiciona-lo ao `<Protobuf>` no `<ItemGroup>` que se enconta no ficheiro .csproj do projeto. Por default os assets do server e do cliente são gerados para cada ficheiro *.proto incluído neste ItemGroup. Para nos assegurarmos que apenas os assets do server são gerados num projeto server, temos que definir que a flag GrpcServices é igual a Server.

```
<ItemGroup>
  <Protobuf Include="CustomCode\Protos\alive.proto" GrpcServices="Server" />
</ItemGroup>
```

Os ficheiros:

- São gerados numa base "as-needed" sempre que damos build ao projeto
- Não são adicionados ao projeto ou checked no source control
- São um build artifact contidos na diretoria obj

O tooling package gera os tipos do C# que representam as mensagens definidas nos ficheiros *.proto incluidos no ficheiro .csproj do projeto. Nos assets do lado do server, é gerado um serviço abastrato do tipo base (neste caso AliveBase) e este contém a definição de todas as calls gRPC contidas num ficheiro .proto. Para podermos utilizar este serviço primeiro temos que criar um serviço concreto (criar uma nova classe) que derive do tipo base e implemente a lógica para as calls gRPC. Podemos ver um exemplo da implementação do serviço Alive e da respetiva lógica da função Sum abaixo.

```
/// <summary>
/// Defines the class for the gRPC Alive service.
/// </summary>
public class AliveService : Alive.AliveBase
{
    private readonly ILogger<AliveService> logger;

    /// <summary>
    /// Initializes a new instance of the <see cref="AliveService" /> class.
    /// </summary>
    /// <param name="loggerService">LoggerService</param>
    public AliveService(ILogger<AliveService> loggerService)
    {
        this.logger = loggerService;
    }

    /// <summary>
    /// Implements the Sum function
    /// </summary>
    /// <param name="request">gRPC Sum Request</param>
    /// /// <param name="context">gRPC Context</param>
    /// <returns>
```

```
    /// The sum of 2 numbers
    /// </returns>
    ////[Authorize]
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1062:Validate
arguments of public methods", Justification = "<Pending>")]
    public override Task<SumReply> Sum(SumRequest request, ServerCallContext
context)
    {
        var user = context.GetHttpContext().User;

        SumReply reply = new SumReply();
        reply.Message = request.Number1 + request.Number2;
        return Task.FromResult(reply);
    }
}
```

Cliente

Para implementarmos a lógica do lado do cliente o processo inicial é similar ao do server. Primeiro copiamos os ficheiros *.proto que definimos do lado do server para o cliente (é expectavel que a versão mais recente esteja sempre presente em ambos os lados para existir uma "concordância" de contratos entre eles. Caso hajam funções, ou serviços, no proto do lado do server que não estejam no proto do lado do cliente este não as conseguirá gerar, e consequentemente invocar, apesar de estes existirem).

Posteriormente, e tal como do lado do server, definimos todos os nossos ficheiros .proto no ficheiro .csproj para os assets destes poderem ser gerados. Neste caso a flag GrpcServices é igual a Client para só serem gerados os assets para o cliente.

```
<ItemGroup>
    <Protobuf Include="Protos\alive.proto" GrpcServices="Client" />
</ItemGroup>
```

Nos assets do lado do cliente, é gerado um asset concreto do tipo cliente. As calls do gRPC no ficheiro .proto são traduzidas em métodos no tipo concreto, que podem ser chamadas. Abaixo podemos ver um exemplo da implementação do serviço Alive e da respetiva lógica da função Sum.

```
static async Task Main(string[] args)
{
    // The port number(20001) must match the port of the gRPC server.
    var channel = GrpcChannel.ForAddress("https://localhost:20001");
    var client =  new Alive.AliveClient(channel);

    Console.WriteLine();
    Console.WriteLine("Unary Call Test");
    Console.WriteLine("Write the first number");
    var number1 = Console.ReadLine();

    Console.WriteLine("Write the second number");
```

```
    var number2 = Console.ReadLine();
    Console.WriteLine();

    var aliveRequest = new SumRequest { Number1 = Convert.ToInt32(number1),
Number2 = Convert.ToInt32(number2) };

    var alive = await client.SumAsync(aliveRequest);
    Console.WriteLine($"{number1} + {number2} = {alive.Message}");
}
```

Notas:

- Para invocarmos diferentes serviços precisamos de criar diferentes clientes
- Para mais informações técnicas consultar o powerpoint

Serviços gRPC com ASP.NET Core

Nota: Para ser possível utilizar o gRPC é necessário possuir o SDK do .NET Core 3.0, ou superior, no nosso projeto.

Configurar o gRPC

No startup.cs:

- O gRPC é ativo no middleware com o método `AddGrpc`
- Cada serviço gRPC é adicionado ao pipepline de routing pelo método `MapGrpcService`

Abaixo podemos observar um exemplo da implementação para o serviço Alive.

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add services to
the container.
    // For more information on how to configure your application, visit
https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();
    }

    // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();
```

```
        app.UseEndpoints(endpoints =>
        {
            // Communication with gRPC endpoints must be made through a gRPC
client.
            // To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909
            endpoints.MapGrpcService<AliveService>();
        });
    }
}
```

Os middlewares e features do ASP.NET Core partilham a pipeline de routing, portanto uma app pode ser configurada para servir request handlers adicionais. Os request handlers adicionais, tais como controllers MVC, funcionam em paralelo com os serviços gRPC configurados.

Configurar o Kestrel

Os endpoints Kestrel gRPC:

- Requerem o HTTP/2
- Devem estar seguros com o Transport Layer Security (TLS)

HTTP/2

O gRPC requer o HTTP/2 para funcionar. O kestrel suporta HTTP/2 em maior partes dos sistemas operativos modernos. Os endpoints Kestrel são configurados para suportar conexões HTTP/1.1 e HTTP/2 por default. Deste forma, podemos manter a nossa API REST e implementar o gRPC em paralelo, o que é bom em caso de migração de uma tecnologia ou se quisermos no geral disponibilizar as duas opções.

Para ativarmos o HTTP/2 e mantermos suporte ao HTTP/1 temos que definir o protocolo do EndpointDefaults para o Kestrel como "Http1AndHttp2" no ficheiro appSetting.json. Se só quisermos suportar HTTP/2 é só meter "Http2" exclusivamente.

```
"Kestrel": {
    "EndpointDefaults": {
      "Protocols": "Http1AndHttp2"
    }
}
```

TLS

Os endpoints Kestrel usados para o gRPC devem ser seguros com o TLS. Em desenvolvimento, um endpoint seguro com TLS é automaticamente criado em `https://localhost:5001` quando o certificado de desenvolvimento do ASP.NET Core está presente. Não é necessário configurações. Um prefixo `https` verifica que o endpoint Kestrel está a usar TLS.

Em produtivo, o TLS deve ser explicitamente configurado. No seguinte exemplo do appsettings.json, é fornecido um endpoint HTTP/2 com segurança TLS.

```
{
  "Kestrel": {
    "Endpoints": {
      "HttpsInlineCertFile": {
        "Url": "https://localhost:5001",
        "Protocols": "Http2",
        "Certificate": {
          "Path": "<path to .pfx file>",
          "Password": "<certificate password>"
        }
      }
    }
  }
}
```

Alternativamente, os endpoints Kestrel podem ser configurados no Program.cs

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.ConfigureKestrel(options =>
            {
                options.Listen(IPAddress.Any, 5001, listenOptions =>
                {
                    listenOptions.Protocols = HttpProtocols.Http2;
                    listenOptions.UseHttps("<path to .pfx file>",
                        "<certificate password>");
                });
            });
            webBuilder.UseStartup<Startup>();
        });
```

Negociação de protocolos

O TLS é usado para mais do que proteger comunicações. O handshake do TLS, Application-Layer Protocol Negotiation (ALPN), é usado para negociar o protocolo de conexão entre o cliente e o server quando um endpoint suporta multiplos protocolos. Este negociação determina se a conexão usa HTTP/1.1 ou HTTP/2.

Se um endpoint HTTP/2 for configurado sem TLS, o ListenOptions.Protocols do endpoint deverá ser alterado para `HttpProtocols.Http2`. Um endpoint com multiplos protocolos (por exemplo, `HttpProtocols.Http1AndHttp2`) não pode ser utilizado sem o TLS porque sem ele não há negociação de protocolos. Todas as conexões a enpoints inseguros devem ser, por default, HTTP/1.1, e desta forma a call gRPC falha.

Nota: O macOS não suporta ASP.NET Core com TLS e são necessárias configurações adicionais para correr com sucesso serviços gRPC no macOS.

Exemplo mais completo

Outras vertentes adicionais informativas estão aqui implementadas de forma prática. Este exemplo possui:

- Método de autenticação por JWTSecurityToken
- Repository
- 2 funções para o serviço Cinema onde podemos ver os filmes disponiveis e a quantidade de bilhetes disponíveis para um certo filme. Uma das funções é Unary call (call normal cliente-servidor, é feito um request e retornada uma resposta) e a outra é Server Streaming (a stream fica aberta até que o Server acabe de mandar tudo o que tem para enviar ao cliente. De forma a simular este caso no código foi implementado um timeout entre envios de filmes disponiveis)
- Altera o tipo de logging do gRPC para debug level, de forma a obtermos informação de logging mais completa
- Integração com APIs do ASP.NET Core como Dependency Injection para injeção do contexto do repository e logging. Por default, a implementação do serviço gRPC pode usar serviços de DI com qualquer lifetime (Singleton, Scoped, or Transient).
- Acesso a alguns dados de mensagem HTTP/2 que a API do gRPC disponibiliza, tais como, o metodo, host, header e trailers. O acesso a estes é feito através do argumento `ServerCallContext` passado em cada método gRPC. Este não fornece acesso total ao `HttpContext` em todas as APIs do ASP.NET. O método da extensão `GetHttpContext` fornece acesso total ao `HttpContext` que representa a mensagem subjacente nas APIs ASP.NET.

Do lado do server

Ficheiro proto

```
syntax = "proto3";

import "google/protobuf/empty.proto";

option csharp_namespace = "MyLiMsRPC";

package cinema;

// The greeting service definition.
service Cinema {
  // Get available movies
  rpc GetAvailableMovies (google.protobuf.Empty) returns (stream
GetAvailableMoviesResponse);
  // Get available ticket count
  rpc GetAvailableTicketsForMovie (GetAvailableTicketsForMovieRequest) returns
(GetAvailableTicketsForMovieResponse);
}

// The response message to get available movies
message GetAvailableMoviesResponse {
  int32 movieId = 1;
  string name = 2;
  string genre = 3;
  int32 duration = 4;
  double classification = 5;
  int32 numberOfAvailableTickets = 6;
```

```
}

// The request message to get the available ticket count
message GetAvailableTicketsForMovieRequest {
  int32 movieId = 1;
}

// The response message containing the available ticket count
message GetAvailableTicketsForMovieResponse {
  int32 availableTickets = 1;
}
```

Implementação do serviço

```
public class CinemaService : Cinema.CinemaBase
{
    private readonly ILogger<CinemaService> _logger;
    private readonly CinemaRepository _cinemaRepository;

    public CinemaService(ILogger<CinemaService> logger, CinemaRepository
cinemaRepository)
    {
        _logger = logger;
        _cinemaRepository = cinemaRepository;
    }

    public override async Task GetAvailableMovies(Empty request,
IServerStreamWriter<GetAvailableMoviesResponse> responseStream, ServerCallContext
context)
    {
        Console.WriteLine("List of movies - " + _cinemaRepository.ListOfMovies());
        foreach (var movie in _cinemaRepository.ListOfMovies())
        {
            Console.WriteLine("Movie - " + movie);
            //await Task.Delay(1000);
            await responseStream.WriteAsync(movie);
        }
    }

    public override Task<GetAvailableTicketsForMovieResponse>
GetAvailableTicketsForMovie(GetAvailableTicketsForMovieRequest request,
ServerCallContext context)
    {
        GetAvailableTicketsForMovieResponse reponse = new
GetAvailableTicketsForMovieResponse();
        reponse.AvailableTickets =
_cinemaRepository.GetAvailableTickets(request.MovieId);
        return Task.FromResult(reponse);
    }
}
```

Implementação do CinemaRepository

```
public class CinemaRepository
{
    private readonly ILogger<CinemaRepository> _logger;

    public CinemaRepository(ILoggerFactory loggerFactory)
    {
        _logger = loggerFactory.CreateLogger<CinemaRepository>();
    }

    public List<GetAvailableMoviesResponse> ListOfMovies()
    {
        return new List<GetAvailableMoviesResponse>
        {
            new GetAvailableMoviesResponse
            {
                MovieId = 1,
                Name = "Avengers",
                Genre = "Superheros",
                Duration = 159,
                Classification = 8.9,
                NumberOfAvailableTickets = 8
            },
            new GetAvailableMoviesResponse
            {
                MovieId = 2,
                Name = "Titanic",
                Genre = "Documentary",
                Duration = 125,
                Classification = 7.2,
                NumberOfAvailableTickets = 27
            },
            new GetAvailableMoviesResponse
            {
                MovieId = 3,
                Name = "IT",
                Genre = "Terror",
                Duration = 172,
                Classification = 6.4,
                NumberOfAvailableTickets = 5
            }
        };
    }

    public int GetAvailableTickets(int movieId)
    {
        return ListOfMovies().First(movie => movie.MovieId ==
movieId).NumberOfAvailableTickets;
    }
}
```

Startup

```
public class Startup
{
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.AddFilter("Grpc", LogLevel.Debug);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
    });

    // This method gets called by the runtime. Use this method to add services to
the container.
    // For more information on how to configure your application, visit
https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();

        services.AddSingleton<CinemaRepository>();

        services.AddAuthorization(options =>
        {
            options.AddPolicy(JwtBearerDefaults.AuthenticationScheme, policy =>
            {

policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
                policy.RequireClaim(ClaimTypes.Name);
            });
        });
        services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
            .AddJwtBearer(options =>
            {
                options.TokenValidationParameters =
                    new TokenValidationParameters
                    {
                        ValidateAudience = false,
                        ValidateIssuer = false,
                        ValidateActor = false,
                        ValidateLifetime = true,
                        IssuerSigningKey = SecurityKey
                    };
            });
    }

    // This method gets called by the runtime. Use this method to configure the
HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
```

```
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseRouting();

        app.UseAuthentication();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            // Communication with gRPC endpoints must be made through a gRPC
client.
            // To learn how to create a client, visit:
https://go.microsoft.com/fwlink/?linkid=2086909
            endpoints.MapGrpcService<AliveService>();
            endpoints.MapGrpcService<CinemaService>();

            endpoints.MapGet("/generateJwtToken", context =>
            {
                return
context.Response.WriteAsync(GenerateJwtToken(context.Request.Query["name"]));
            });
        });
    }

    private string GenerateJwtToken(string name)
    {
        if (string.IsNullOrEmpty(name))
        {
            throw new InvalidOperationException("Name is not specified.");
        }

        var claims = new[] { new Claim(ClaimTypes.Name, name) };
        var credentials = new SigningCredentials(SecurityKey,
SecurityAlgorithms.HmacSha256);
        var token = new JwtSecurityToken("ExampleServer", "ExampleClients",
claims, expires: DateTime.Now.AddSeconds(60), signingCredentials: credentials);
        return JwtTokenHandler.WriteToken(token);
    }

    private readonly JwtSecurityTokenHandler JwtTokenHandler = new
JwtSecurityTokenHandler();
    private readonly SymmetricSecurityKey SecurityKey = new
SymmetricSecurityKey(Guid.NewGuid().ToByteArray());
}
```

Do lado do cliente

```
```

```
class Program
{
    private const string Address = "localhost:5001";

    private static string _token;

    static async Task Main(string[] args)
    {
        var channel = CreateAuthenticatedChannel($"https://{Address}");
        var client = new Alive.AliveClient(channel);
        var cinemaClient = new Cinema.CinemaClient(channel);

        Console.WriteLine("gRPC Ticketer");
        Console.WriteLine();
        Console.WriteLine("Press a key:");
        Console.WriteLine("1: Make a sum");
        Console.WriteLine("2: Get available movies");
        Console.WriteLine("3: Get available Tickets for movie");
        Console.WriteLine("4: Authenticate");
        Console.WriteLine("5: Exit");
        Console.WriteLine();

        var exiting = false;
        while (!exiting)
        {
            var consoleKeyInfo = Console.ReadKey(intercept: true);
            switch (consoleKeyInfo.KeyChar)
            {
                case '1':
                    await PurchaseTicket(client);
                    break;
                case '2':
                    await GetAvailableMovies(cinemaClient);
                    break;
                case '3':
                    await GetAvailableTicketsForMovie(cinemaClient);
                    break;
                case '4':
                    _token = await Authenticate();
                    break;
                case '5':
                    exiting = true;
                    break;
            }
        }

        Console.WriteLine("Exiting");
    }

    private static GrpcChannel CreateAuthenticatedChannel(string address)
    {
        var credentials = CallCredentials.FromInterceptor((context, metadata) =>
        {
```

```
            if (!string.IsNullOrEmpty(_token))
            {
                metadata.Add("Authorization", $"Bearer {_token}");
            }
            return Task.CompletedTask;
        });

        // SslCredentials is used here because this channel is using TLS.
        // Channels that aren't using TLS should use ChannelCredentials.Insecure
instead.
        var channel = GrpcChannel.ForAddress(address, new GrpcChannelOptions
        {
            Credentials = ChannelCredentials.Create(new SslCredentials(),
credentials)
        });
        return channel;
    }

    private static async Task<string> Authenticate()
    {
        Console.WriteLine($"Authenticating as {Environment.UserName}...");
        var httpClient = new HttpClient();
        var request = new HttpRequestMessage
        {
            RequestUri = new Uri($"https://{Address}/generateJwtToken?name=
{HttpUtility.UrlEncode(Environment.UserName)}"),
            Method = HttpMethod.Get,
            Version = new Version(2, 0)
        };
        var tokenResponse = await httpClient.SendAsync(request);
        tokenResponse.EnsureSuccessStatusCode();

        var token = await tokenResponse.Content.ReadAsStringAsync();
        Console.WriteLine("Successfully authenticated.");
        Console.WriteLine("You token is : Bearer " + token);
        Console.WriteLine();

        return token;
    }

    private static async Task GetAvailableMovies(Cinema.CinemaClient client)
    {
        Console.WriteLine();
        Console.WriteLine("Server Call Test");
        Console.WriteLine("Getting available movies...");
        using (var call = client.GetAvailableMovies(new Empty()))
        {
            while (await call.ResponseStream.MoveNext())
            {
                var currentMovie = call.ResponseStream.Current;

                Console.WriteLine();
                Console.WriteLine($"Movie ID : {currentMovie.MovieId}");
                Console.WriteLine($"Movie Name : {currentMovie.Name}");
```

```
                Console.WriteLine($"Movie Genre : {currentMovie.Genre}");
                Console.WriteLine($"Movie Duration (minutes) :
{currentMovie.Duration}");
                Console.WriteLine($"Movie Classification :
{currentMovie.Classification}");
                Console.WriteLine($"Number of Tickets For Movie :
{currentMovie.NumberOfAvailableTickets}");
            }
        }
    }

    private static async Task GetAvailableTicketsForMovie(Cinema.CinemaClient
client)
    {
        Console.WriteLine();
        Console.WriteLine("Choose a movieID to know how many tickets are
available");
        var movieID = Console.ReadLine();
        var request = new GetAvailableTicketsForMovieRequest();
        request.MovieId = Convert.ToInt32(movieID);
        var response = await client.GetAvailableTicketsForMovieAsync(request);
        Console.WriteLine("There are " + response.AvailableTickets + " tickets
available");
    }

    private static async Task PurchaseTicket(Ticketer.TicketerClient client)
    {
       Console.WriteLine("Purchasing ticket...");
       try
       {
           var response = await client.BuyTicketsAsync(new BuyTicketsRequest {
Count = 1 });
           if (response.Success)
           {
               Console.WriteLine("Purchase successful.");
           }
           else
           {
               Console.WriteLine("Purchase failed. No tickets available.");
           }
       }
       catch (Exception ex)
       {
           Console.WriteLine("Error purchasing ticket." + Environment.NewLine +
ex.ToString());
       }
    }

    private static async Task GetAvailableTickets(Ticketer.TicketerClient client)
    {
       Console.WriteLine("Getting available ticket count...");
       var response = await client.GetAvailableTicketsAsync(new Empty());
       Console.WriteLine("Available ticket count: " + response.Count);
```

```
    }
}
```