Write a Grpc Service that does following

(1)single product by ID and a

(2)nother method that can return a list of Products back to the client.

The first step in implementing our own Product Service in a gRPC Application is to add a new proto. Open up the Server Project and add a new proto file under the Protos folder. Name it products.proto.

grpc-in-aspnet-core-getting-started

This will give you a new proto file with the syntax and namespace already ready. We will just have to add in the Service Definition and Message Objects here.

Before continuing make sure that yo u change the properties of the proto file similar to the below screenshot. Remember to do this step everytime you create a new proto file.

The first step in implementing our own Product Service in a gRPC Application is to add a new proto.

Step:1

Open up the Server Project and add a new proto file under the Protos folder. Name it products.proto.

Step:2

Let’s add in our required service definition and the associated Message objects to the products.proto file.

**syntax = "proto3";**

**option csharp\_namespace = "gRPCServer.Protos";**

**service Product{**

**rpc GetProductById (GetProductByIdModel) returns (ProductModel);**

**rpc GetAllProducts (GetAllProductsRequest) returns (stream ProductModel);**

**}**

**message GetProductByIdModel{**

**int32 productId = 1;**

**}**

**message GetAllProductsRequest{**

**}**

**message ProductModel{**

**int32 productId = 1;**

**string name = 2;**

**string description = 3;**

**float price =4;**

**}**

**Line 3 to 6** is where you define the Service and it’s method. Here we have 2 Methods under the Product Service. The first one takes in a Message object (productId) and returns a single product.

* **The other method takes a blank Message object and returns a stream of ProductModel objects.**

What is a stream? gRPC usually is used for streaming in and out data. In our context we use it to stream the data one product at a time from the server to the client. This is quite similar to IEnumerable<ProductModel> but with lot more flexibilty.

Line 13, we define the actual model of the product. Note that you cannot use int, decimal like you would do back in C#. Remember that proto is a completely different language and has nothing to do with C# or even Microsoft. You can find the allowed object types

* **https://protobuf.dev/programming-guides/proto3/#scalar**

**step:4 -We need a list of Product data to work with**

create a new class with a list of hardcoded value for the Product model

we can use databse instead of list, You can install the Entity Framework Core package, establish a connection and use the dbContext to pull in product data from your database as well.

public static class ProductData

{

**public static List<ProductModel> ProductModels = new List<ProductModel>**

**{**

**new ProductModel**

**{**

**ProductId = 1,**

**Name = "Pepsi",**

**Description = "Soft Drink",**

**Price = 10**

**},**

**new ProductModel**

**{**

**ProductId = 2,**

**Name = "Fanta",**

**Description = "Soft Drink",**

**Price = 13**

**},**

**new ProductModel**

**{**

**ProductId = 3,**

**Name = "Pizza",**

**Description = "Fast Food",**

**Price = 25**

**},**

**new ProductModel**

**{**

**ProductId = 4,**

**Name = "French Fries",**

**Description = "Fast Food",**

**Price = 20**

**}**

**};**

**}**

With the Product data ready,

* **Step5:**
* **let’s create a service class that could serve data from the ProductData Class to the client. In the server project add a new class named ProductService under the Services folder.**

public class ProductService : Product.ProductBase

{

private readonly ILogger<ProductService> \_logger;

public ProductService(ILogger<ProductService> logger)

{

\_logger = logger;

}

public override Task<ProductModel> GetProductById(GetProductByIdModel request, ServerCallContext context)

{

var product = ProductData.ProductModels.Where(p => p.ProductId == request.ProductId).FirstOrDefault();

if(product!=null)

{

return Task.FromResult(product);

}

else

{

return null;

}

}

public override async Task GetAllProducts(GetAllProductsRequest request, IServerStreamWriter<ProductModel> responseStream, ServerCallContext context)

{

var allProducts = ProductData.ProductModels.ToList();

foreach(var product in allProducts)

{

await responseStream.WriteAsync(product);

}

}

}

QUICK TIP - You can type in public override which would then given you a list of override-able methods. Simply navigate to the required method and tap on Tab. It creates the template of the entire function for you. That’s quite a lot of development time saved.

Line 9 to 20 is the method that returns a single product based on the Id that is sent by the client. I guess this is a very straightforward method that used LINQ to fetch the data from the product List.

Remember we used something called a stream in the proto file? Line 21 to 28 is the function that can return a stream of Product data back to the client. Note that we are sending data one by one in a stream to the response. As long as the control remains in the foreach loop, the client continues to listen for the data stream.

Finally,

Step6:-open up the starup.cs and add the mapping to the new gRPC Service under the app.UseEndpoints(…)

endpoints.MapGrpcService<ProductService>();

What remains is the client implementation. But before that, let’s rebuild our Server project.

With gRPC, make sure that you rebuild the entire project every time there is a change made. This is because there is a lot of auto generation of code that happens in the background.

At the client side,

**Step:7-** the first thing to copy over the products.proto file from the Server to the client. Make sure you change the properties of the client proto as follows. Make this a practice as soon as copy over the proto files.

grpc-in-aspnet-core-getting-started

**stiep:8-**open up the Program.cs in our Console application and add in the following.

static async Task Main(string[] args)

{

var data = new GetProductByIdModel { ProductId = 2 };

var grpcChannel = GrpcChannel.ForAddress("https://localhost:5001");

var client = new Product.ProductClient(grpcChannel);

var response = await client.GetProductByIdAsync(data);

Console.WriteLine(response);

Console.ReadLine();

using (var clientData = client.GetAllProducts(new GetAllProductsRequest()))

{

while(await clientData.ResponseStream.MoveNext(new System.Threading.CancellationToken()))

{

var thisProduct = clientData.ResponseStream.Current;

Console.WriteLine(thisProduct);

}

}

Console.ReadLine();

}

For demonstration purposes, we will be using both the methods here in the Main Function.

Line 3 creates a new GetProductByIdModel with a productId of 2. This will be something the client will be sending dynamically.

Line 4 creates a new gRPC channel pointing to the address where the gRPC server runs.

Line 5 creates a new client object using the gRPC channel. Note that we will be using this client object to access both the GetByID and GetAll methods.

Line 6, we pass in the data to the client, which returns back the required response.

Similarly, from Line 9 to 16, we return a stream of product data one line at a time.

Let’s run both the Server and Client Project and check out the responses.

\