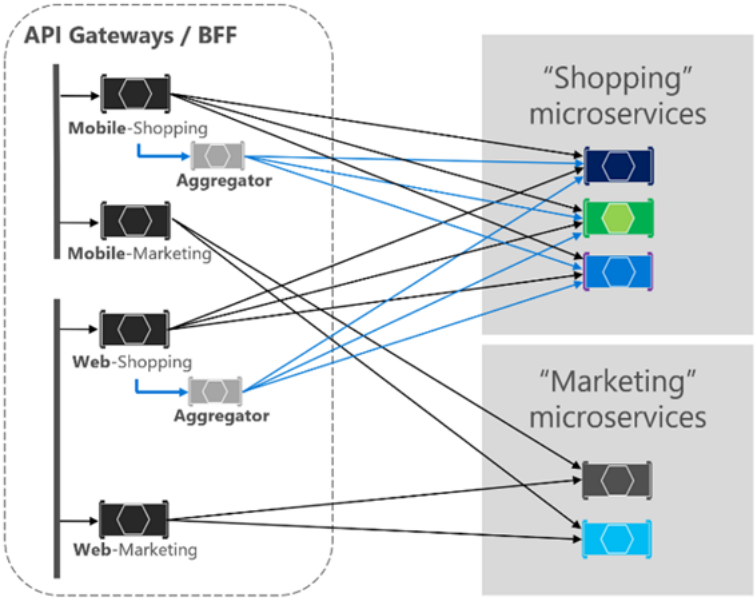
Using gRPC in Microservices for Building a high-performance Interservice Communication with .Net 5

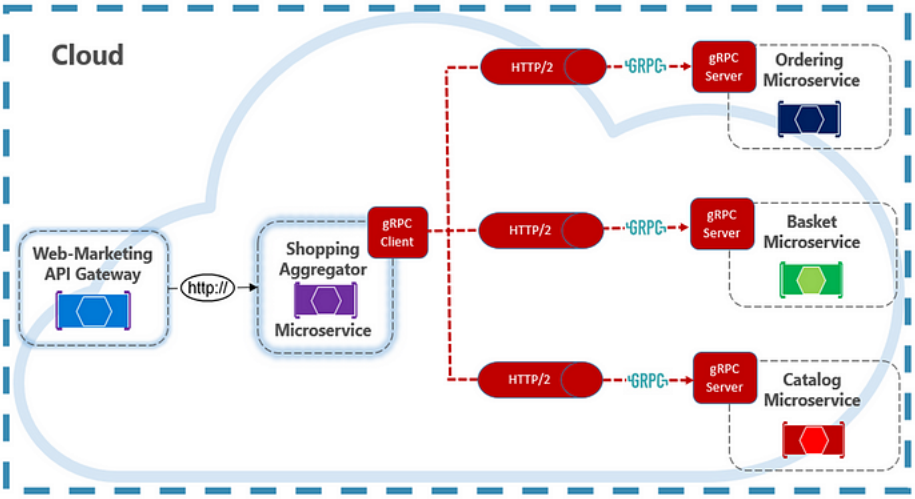


In this article, we're going to learn how to Build a Highly Performant Inter-service Communication with gRPC for ASP NET 5 Microservices.

We will introduce gRPC as a modern high-performance RPC framework for ASP.NET Core and for interservice communication.
gRPC uses HTTP/2 as base transport protocol and ProtoBuf encoding for efficient and fast communication.

gRPC usage of Microservices

Microservices are modern distributed systems so with gRPC in ASP.NET 5, we will develop high-performance, cross-platform applications for building distributed systems and APIs.



It's an ideal choice for communication between backend microservices, internal network applications, or iot devices and services. With the release of ASP.NET 5, Microsoft has added first-class support for creating gRPC services with aspnet5.

This article will led you get started building, developing and managing gRPC servers and clients on distributed microservices architecture.

Step by Step Development w/ Course

I have just published a new course — Using gRPC in Microservices Communication with .Net 5.

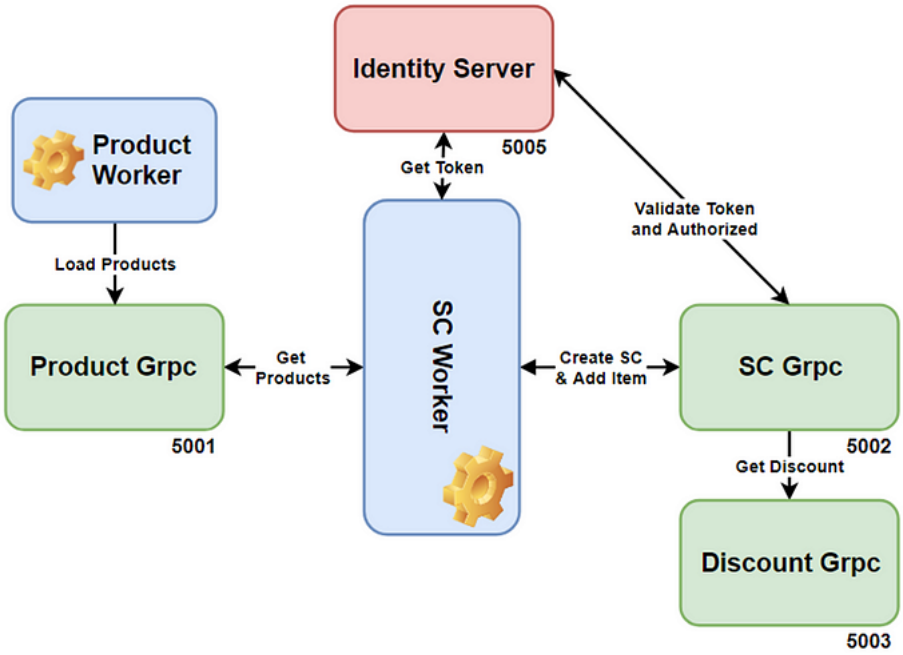
In the course, we are going to build a **high-performance gRPC Inter-Service Communication between backend microservices with .Net 5 and Asp.Net5.**

Source Code

Get the Source Code from AspnetRun Microservices Github — Clone or fork this repository, if you like don't forget the star :) If you find or ask anything you can directly open issue on repository.

Overall Picture

See the overall picture. You can see that we will have 6 microservices which we are going to develop.
We will use Worker Services and Asp.Net 5 Grpc applications to build client and server gRPC components defining proto service definition contracts.



Basically we will implement **e-commerce** logic with only gRPC communication. We will have 3 gRPC server applications which are **Product** — **ShoppingCart** and **Discount** gRPC services. And we will have 2 worker services which are **Product** and **ShoppingCart Worker Service**. Worker services will be client and perform operations over the gRPC server applications. And we will secure the **gRPC services** with standalone **Identity Server** microservices with **OAuth 2.0** and **JWT** token.

ProductGrpc Server Application

First of all, we are going to develop ProductGrpc project. This will be asp.net gRPC server web application and expose apis for Product Crud operations.

Product Worker Service

After that, we are going to develop Product Worker Service project for consuming ProductGrpc services. This product worker service project will be the client of ProductGrpc application and generate products and insert bulk product records into Product database by using client streaming gRPC proto services of ProductGrpc application. This operation will be in a time interval and looping as a service application.

ShoppingCartGrpc Server Application

After that, we are going to develop ShoppingCartGrpc project. This will be asp.net gRPC server web application and expose apis for SC and SC items operations. The grpc services will be create sc and add or remove item into sc.

ShoppingCart Worker Service

After that, we are going to develop ShoppingCart Worker Service project for consuming ShoppingCartGrpc services. This ShoppingCart worker service project will be the client of both ProductGrpc and ShoppingCartGrpc application. This worker service will read the products from ProductGrpc and create sc and add product items into sc by using gRPC proto services of ProductGrpc and ShoppingCartGrpc application. This operation will be in a time interval and looping as a service application.

DiscountGrpc Server Application

When adding product item into SC, it will retrieve the discount value and calculate the final price of product. This communication also will be gRPC call with SCGrpc and DiscountGrpc application.

Identity Server

Also, we are going to develop centralized standalone Authentication Server with implementing IdentityServer4 package and the name of microservice is Identity Server.
Identity Server4 is an open source framework which implements OpenId Connect and OAuth2 protocols for .Net Core.
With IdentityServer, we can provide protect our SC gRPC services with OAuth 2.0 and JWT tokens. SC Worker will get the token before send request to SC Grpc server application.

> *By the end of this article, you will have a practical understanding of how to use gRPC to implement a fast and distributed microservices systems.*
> *And Also you'll learn how to secure protected grpc services with IdentityServer in a microservices architecture.*

Background

You can follow the previous article which explains overall microservice architecture of this example.

Check for the previous article which explained overall microservice architecture of this repository.

Prerequisites

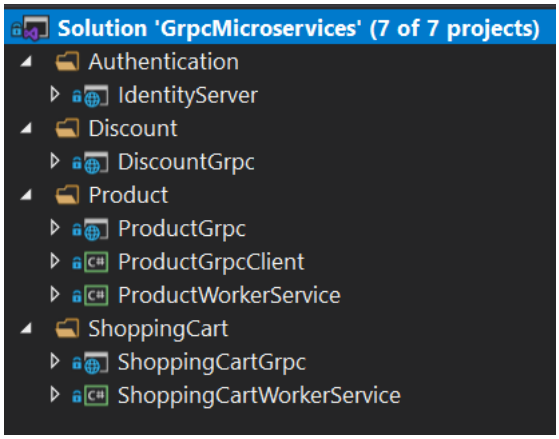
- Install the .NET 5 or above SDK
- Install Visual Studio 2019 v16.x or above

Introduction

We will implement **e-commerce** logic with only gRPC communication. We will have 3 gRPC server applications which are **Product** — **ShoppingCart** and **Discount** gRPC services. And we will have 2 worker services which are **Product** and **ShoppingCart Worker Service**. Worker services will be client and perform operations over the gRPC server applications. And we will secure the **gRPC services** with standalone **Identity Server** microservices with **OAuth 2.0** and **JWT** token.

Code Structure

Let's check our project code structure on the visual studio solution explorer window. You can see the 4 solution folder and inside of that folder you will see Grpc server and client worker projects which we had see on the overall picture.

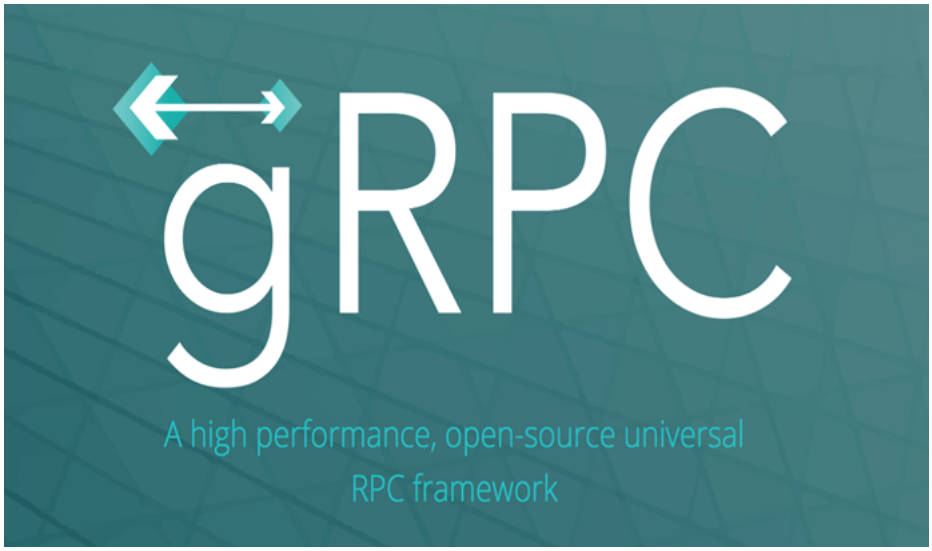


If we expand the projects, you will see that;
Under Product folder; **ProductGrpc** is a gRPC aspnet application which includes crud api operations.
ProductWorkerService is a WorkerService template application which consumes and perform operations over the product grpc server application.
As the same way you can follow the **ShoppingCart** and **Discount** folders.
And also we have **IdentityServer** is a standalone Identity Provider for our architecture.

Before we start we should learn the basics of terminology.

What is gRPC ?

gRPC (gRPC Remote Procedure Calls) is an open source remote procedure call (RPC) system initially developed at Google.
gRPC is a framework to efficiently connect services and build distributed systems.



It is focused on high performance and uses the HTTP/2 protocol to transport binary messages. It is relies on the Protocol Buffers language to define service contracts. Protocol Buffers, also known as Protobuf, allow you to define the interface to be used in service to service communication regardless of the programming language.

It generates cross-platform client and server bindings for many languages. Most common usage scenarios include connecting services in microservices style architecture and connect mobile devices, browser clients to backend services.

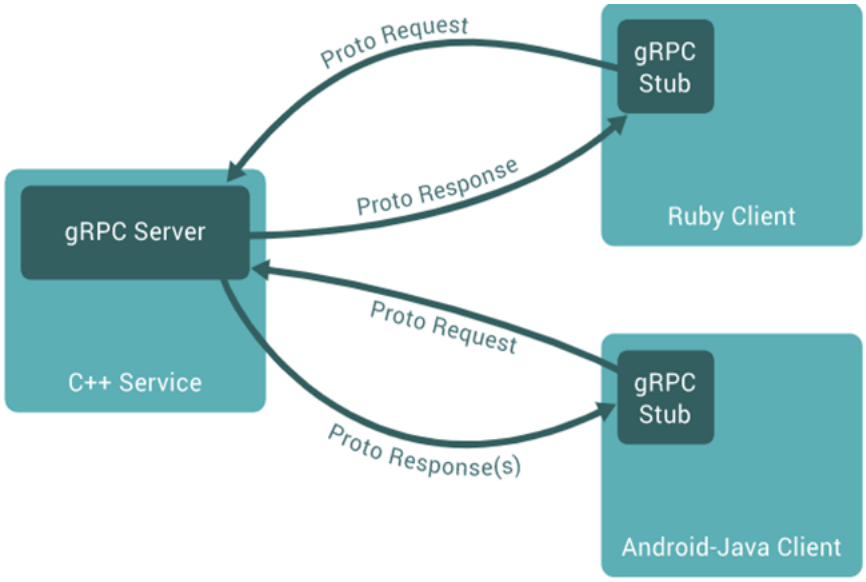
The gRPC framework allows developers to create services that can communicate with each other efficiently and independently from their preferred programming language.

Once you define a contract with Protobuf, this contract can be used by each service to automatically generate the code that sets up the communication infrastructure.

This feature simplifies the creation of service interaction and, together with high performance, makes gRPC the ideal framework for creating microservices.

How gRPC works ?

In GRPC, a client application can directly call a method on a server application on a different machine like it were a local object, making it easy for you to build distributed applications and services.

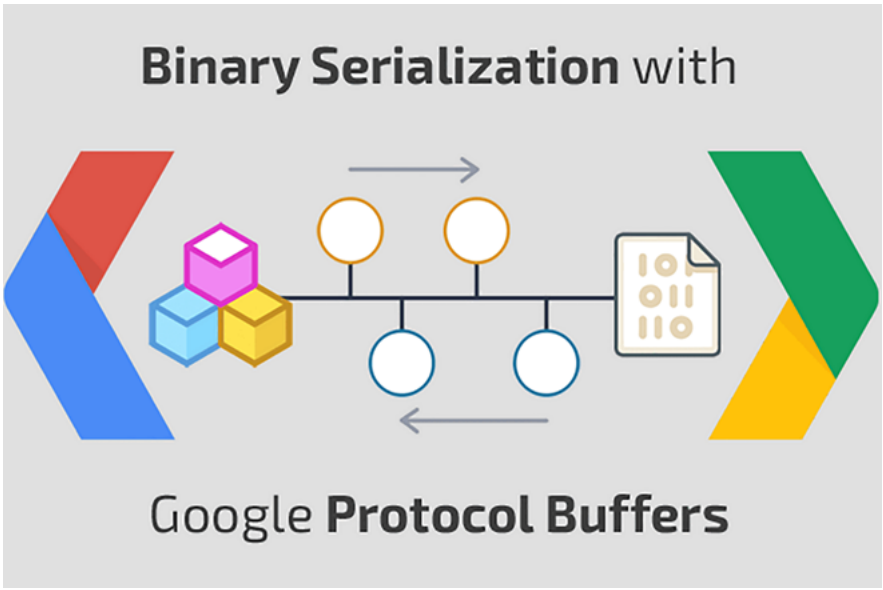


As with many RPC systems, gRPC is based on the idea of defining a service that specifies methods that can be called remotely with their parameters and return types. On the server side, the server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub that provides the same methods as the server.

gRPC clients and servers can work and talk to each other in a different of environments, from servers to your own desktop applications, and that can be written in any language that gRPC supports. For example, you can easily create a gRPC server in Java or c# with clients in Go, Python or Ruby.

Working with Protocol Buffers

gRPC uses Protocol Buffers by Default.
Protocol Buffers are Google's open source mechanism for serializing structured data.



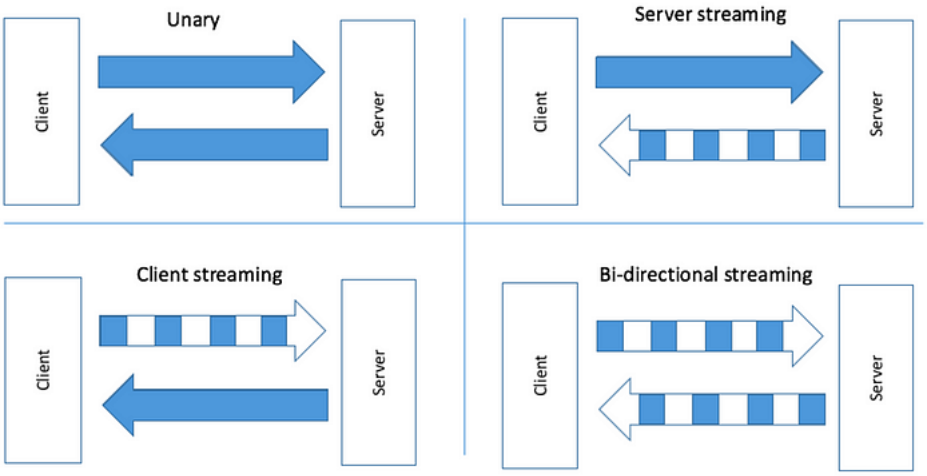
When working with protocol buffers, the first step is to define the structure of the data you want to serialize in a proto file: this is an ordinary text file with the extension .proto.
The protocol buffer data is structured as messages where each message is a small logical information record containing a series of name-value pairs called fields.

Once you've determined your data structures, you use the protocol buffer compiler protocol to create data access classes in the languages you prefer from your protocol definition.

You can find the whole language guide into google's offical documentation of protocol buffer language. Let me add the link as below.
https://developers.google.com/protocol-buffers/docs/overview

gRPC Method Types — RPC life cycles

gRPC lets you define four kinds of service method:



Unary RPCs where the client sends a single request to the server and returns a single response back, just like a normal function call.

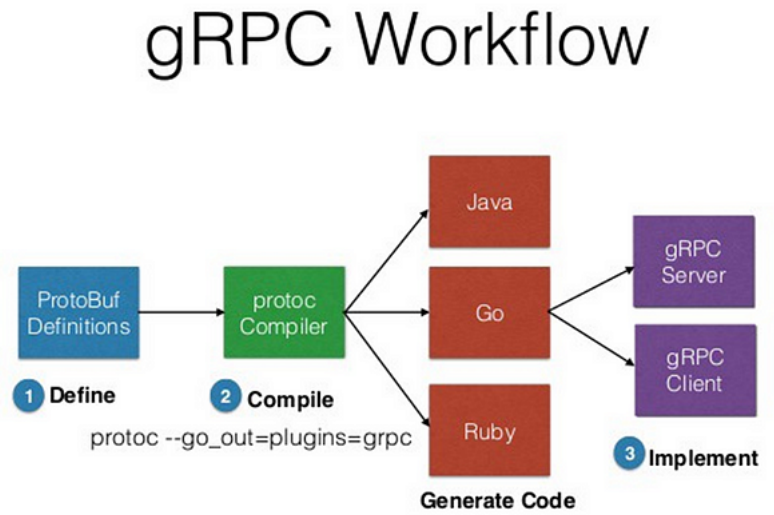
Server streaming RPCs where the client sends a request to the server and gets a stream to read a sequence of messages back. The client reads from the returned stream until there are no more messages. gRPC guarantees message ordering within an individual RPC call.

Client streaming RPCs where the client writes a sequence of messages and sends them to the server, again using a provided stream. Once the client has finished writing the messages, it waits for the server to read them and return its response. Again gRPC guarantees message ordering within an individual RPC call.

Bidirectional streaming RPCs where both sides send a sequence of messages using a read-write stream. The two streams operate independently, so clients and servers can read and write in whatever order they like: for example, the server could wait to receive all the client messages before writing its responses, or it could alternately read a message then write a message, or some other combination of reads and writes.

gRPC Development Workflow

So far we had a good definitions of gRPC and proto buffer files. So now we can summarize the development workflow of gRPC.



gRPC uses a contract-first approach to API development. Services and messages are defined in *.proto files.

gRPC uses Protocol Buffers so we are start to developing protobuf file.
Protocol Buffers is a way to define the structure of data that you want to serialize.

Once we define the structure of data in a file with .proto extension, we use protoc compiler to generate data access classes in your preferred language(s) from your proto definition.
This will generate the data access classes from your application. We choose the C# client during the article.

Advantages of gRPC

General advantages of gRPC:

- Using HTTP / 2

These differences of HTTP / 2 provide 30–40% more performance. In addition, since gRPC uses binary serialization, it needs both more performance and less bandwidth than json serialization.

- Higher performance and less bandwidth usage than json with binary serialization
- Supporting a wide audience with multi-language / platform support
- Open Source and the powerful comminity behind it
- Supports Bi-directional Streaming operations
- Support SSL / TLS usage
- Supports many Authentication methods

gRPC vs REST

gRPC is in an advantadge position against REST-based APIs that have become popular in recent years. Because of the protobuf format, messages take up less space and therefore communication is faster.

	Goal	REST (HTTP/JSON)	gRPC
COMPATIBILITY	Single source of truth	✗	✔
	Multi-platform + languages built in	✗	✔
	Handle non-breaking changes.	✗	✔
PERFORMANCE	Network: connection handling	Manual, 1 per call ✗	Built-in, Multi per conn ✔
	Speed: Transmission of data	Human-readable Text ✗	Binary ✔
	CPU: Improved resource usage	✗	✔
MAINTENANCE	Tracing	Manual ✗	Easy to plug in ✔
	Logging	Manual ✗	Easy to plug in ✔
	Monitoring	Manual ✗	Easy to plug in ✔

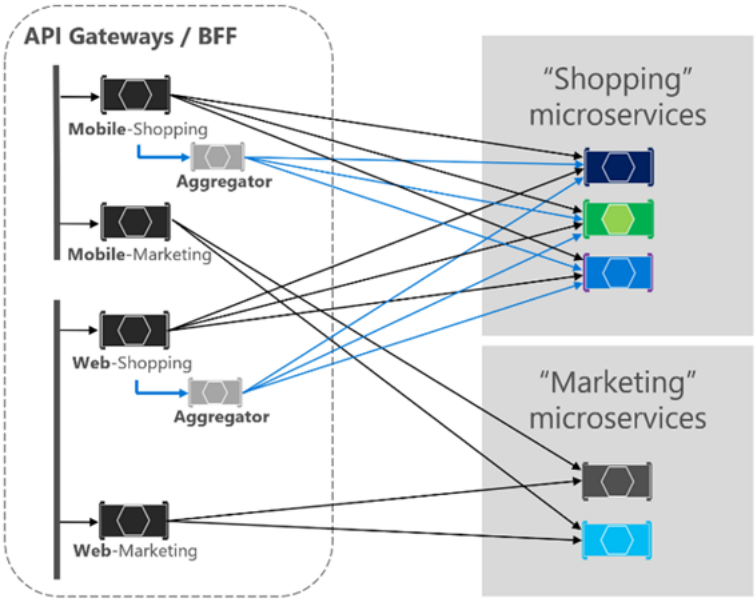
Unlike REST, gRPC works on a contract file basis, similar to SOAP.

Encoding and Decoding part of gRPC requests takes place on the client machine. That’s why the JSON encode / decode you make for REST apis on your machine is not a problem for you here.

You do not need to serialize (serialization / deserialization) for type conversions between different languages because your data type is clear on the contract and the code for your target language is generated from there.

gRPC usage of Microservices Communication

gRPC is primarily used with backend services.

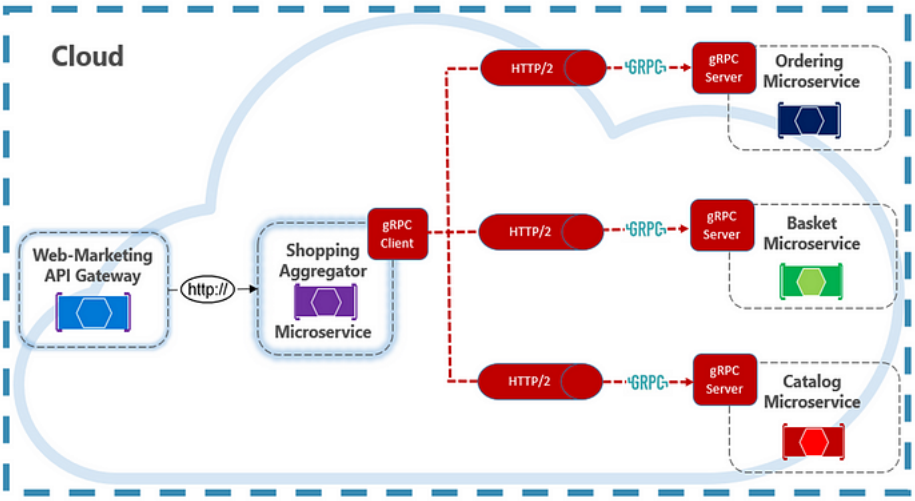


But also gRPC using for the following scenarios:

- Synchronous backend microservice-to-microservice communication where an immediate response is required to continue processing.
- Polyglot environments that need to support mixed programming platforms.
- Low latency and high throughput communication where performance is critical.
- Point-to-point real-time communication — gRPC can push messages in real time without polling and has excellent support for bi-directional streaming.
- Network constrained environments — binary gRPC messages are always smaller than an equivalent text-based JSON message.

Example of gRPC in Microservices Communication

Think about that we have a Web-Marketting API Gateway and this will forward to request to Shopping Aggregator Microservice.



This Shopping Aggregator Microservice receives a single request from a client, dispatches it to various microservices, aggregates the results, and sends them back to the requesting client. Such operations typically require synchronous communication as to produce an immediate response.

In this example, backend calls from the Aggregator are performed using gRPC.
gRPC communication requires both client and server components.

You can see that Shopping Aggregator implements a gRPC client.
The client makes synchronous gRPC calls to backend microservices, this backend microservices are implement a gRPC server.

As you can see that, The gRPC endpoints must be configured for the HTTP/2 protocol that is required for gRPC communication.

In microservices world, most of communication use asynchronous communication patterns but some operations require direct calls. gRPC should be the primary choice for direct synchronous communication between microservices. Its high-performance communication protocol, based on HTTP/2 and protocol buffers, make it a perfect choice.

gRPC with .NET

gRPC support in .NET is one of the best implementations along the other languages.

Last year Microsoft contributed a new implementation of gRPC for .NET to the Cloud Native Computing Foundation — CNCF. Built on top of Kestrel and HttpClient, gRPC for .NET makes gRPC a first-class member of the .NET ecosystem.

gRPC is integrated into .NET Core 3.0 SDK and later.

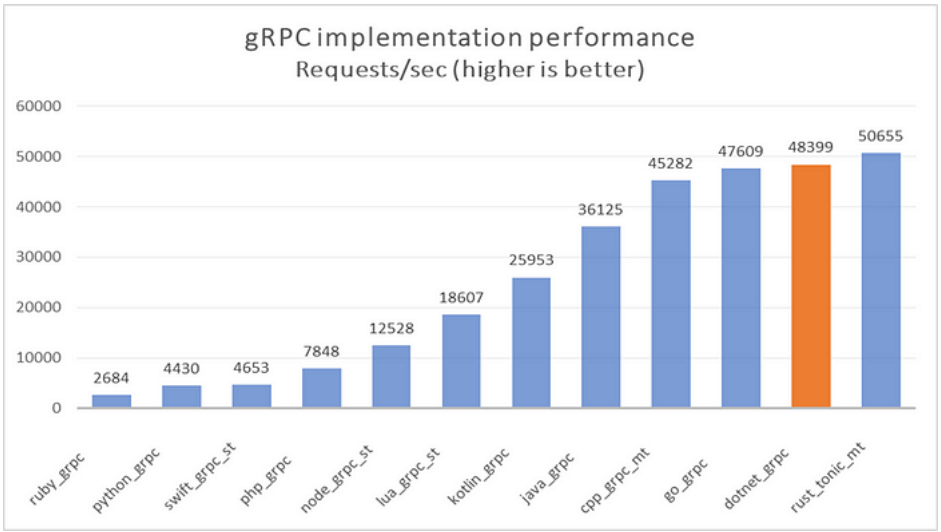
The SDK includes tooling for endpoint routing, built-in IoC, and logging. The open-source Kestrel web server supports HTTP/2 connections.

Also a Visual Studio 2019 template that scaffolds a skeleton project for a gRPC service. Note how .NET Core fully supports Windows, Linux, and macOS.

Both the client and server take advantage of the built-in gRPC generated code from the .NET Core SDK. Client-side stubs provide the plumbing to invoke remote gRPC calls. Server-side components provide gRPC plumbing that custom service classes can inherit and consume.

gRPC performance in .NET 5

gRPC and .NET 5 are really fast.



In a community run benchmark of different gRPC server implementations, .NET gets the highest requests per second after Rust, and is just ahead of C++ and Go.
You can see the image that comparison of performance with other languages.

This result builds on top of the work done in .NET 5. The benchmarks show .NET 5 server performance is 60% faster than .NET Core 3.1. .NET 5 client performance is 230% faster than .NET Core 3.1.

Performance is a very important feature when it comes to communications on scaling cloud applications. So with .Net and gRPC is being very good alternative to implement backend microservices with .net and communicate with gRPC.

HelloWorld gRPC with Asp.Net 5

In this section, we are going to learn how to build gRPC in Asp.Net 5 projects with developing HelloWorld API operations.
We will starting with creating empty web application and develop gRPC implementation with step by step.

We are going to cover the;

- Developing hello.proto Protocol Buffer File (protobuf file) for gRPC Contract-First API Development
- Implementing gRPC Service Class which Inherits from gRPC generated service
- Configure gRPC Service with Registering Asp.Net Dependecy Injection and Configure with Mapping GrpcService in Asp.Net Middleware
- Run the Application as exposing gRPC Services
- Create GrpcHelloWorldClient Client Application for gRPC Server
- Consume Grpc HelloService API From Client Console Application with GrpcChannel
- Scaffolding gRPC Server with gRPC Template of Visual Studio

Create Asp.Net Core Empty Web Project For HelloWorld Grpc

First, we are going to create;

Create Asp.Net Core Web Empty Project :
Right Click Solution — Add new Web Empty — HTTPS — https required for http2 tls protocol
— Solution name : GrpcHelloWorld
— Project name : GrpcHelloWorldServer
After that, the first step we will start with adding Grpc.AspNet Nuget packages in our project.

Add Nuget Package
Grpc.AspNetCore → this includes tools etc packages..

We can continue with the most important part of any gRPC project. Let's Create "Protos" folder and under the Protos folder, add new protobuf file which name is hello.proto :

Create "Protos" folder
Add new protobuf file
hello.proto

gRPC uses a contract-first approach to API development. Services and messages are defined in *.proto files.

Developing hello.proto Protocol Buffer File (protobuf file) for gRPC Contract-First API Development

First, we are going to open hello.proto file. Because, gRPC uses a contract-first approach to API development. Services and messages are defined in *.proto files:

Develop hello.proto file; Let me write first and I will explain details after that.

```
syntax = "proto3";option csharp_namespace = "GrpcHelloWorldServer";package helloworld;service HelloService {
 rpc SayHello (HelloRequest) returns (HelloResponse);
 }message HelloRequest {
 string name = 1;
 }message HelloResponse {
 string message = 1;
 }
```

Ok now, let me explain this proto file;

First, The syntax statement tells the protobuf compiler what syntax we are going to use. it's important to specify which protobuf version we are using.
The second statement is optional and it tells the protobuf compiler to generate C# classes within the specified namespace: GrpcHelloWorldServer.

After that, we have Defined a HelloService. The HelloService service defines a SayHello call. SayHello sends a HelloRequest message and receives a HelloResponse message.

So we are defining the HelloRequest and HelloResponse message types like data transfer objects (DTOs). This messages and services will be generated for accessing from our application.

After that, we should Add this proto file in our project file item groups in order to attach and generate proto file c# codes.
— So Go to **GrpcHelloWorldServer.csproj** project file;

Edit the GrpcHelloWorldServer.csproj project file:
Right-click the project and select Edit Project File.

Add an item group with a <Protobuf> element that refers to the greet.proto file:

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

If not exist, make sure that it should be stored in project file. This will provide to generate C# proto class when build the application.

Now we can build the project. When we build the project, it compiles the proto file and it will generate c# proto class.

Right-click the project and Build the project
it compiles the proto file

Also we can check Properties of Proto file with F4
Build Action = Protobuf compiler
gRPC stub Classes = Server only

As you can see that, we set the “Build Action = Protobuf compiler”, that means when building and compile the project, visual studio also build this hello.proto file with the “Protobuf compiler”.

Also we set the “gRPC stub Classes = Server only”, that means when generate the c# class, it will arrange the codes as a Server of gRPC Api project. So by this way we can implement server logics with generated scaffolding c# proto class.

Check the Generated Class
Click “Show All Files” of Solution Window
Go to -> obj-debug-netcoreapp-protos-> HelloGrpc.cs
See that our class is generated;
public static partial class HelloService

As you can see that, HelloService.cs class generated by visual studio and it will provide to connect gRPC service as a server role. Now we can use this class and implement our server logics with connectting gRPC API.

Implementing gRPC Service Class which Inherits from gRPC generated service

In this part, we are going to implement gRPC Service Class which Inherits from gRPC generated service class.
Let’s take an action.

First, we are going to create Service folder under our application.
After that we can;

Add Service Class into Service Folder

HelloWorldService

Inherit from gRPC generated service class
public class HelloWorldService : HelloService.HelloServiceBase

As you can see that we should inherit our Service class from generated gRPC server class.
And we can use all features of asp.net in here like logging, configurations and other dependency injections, so its so powerfull to use gRPC with .net. We can use all features of aspnet in this class.
For example let’s add logger object.

```
public class HelloWorldService : HelloService.HelloServiceBase
 {
 private readonly ILogger<HelloWorldService> _logger;public HelloWorldService(ILogger<HelloWorldService> logger)
 {
 _logger = logger;
 }
```

Let me develop our gRPC service method which is **SayHello**. You can implement the method with using “**override**” keyword.

```
public override Task<HelloResponse> SayHello(HelloRequest request, ServerCallContext context)
 {
 string resultMessage = $”Hello {request.Name}”;var response = new HelloResponse
 {
 Message = resultMessage
 };return Task.FromResult(response);
 }
```

Once we generated proto file as a server gRPC stub Classes, we can override actual operation inside the our service class with overriding the actual service method from proto file.
As you can see that, in this method we get the HelloRequest message as a parameter and return to “HelloResponse” message with adding “Hello” keyword.
So that means we have implemented gRPC service method in our application.

Configure gRPC Service with Registering Asp.Net Dependecy Injection and Configure with Mapping GrpcService in Asp.Net Middleware

We are going to configure gRPC Service in our aspnet project. So we will apply 2 steps;
1- Configure gRPC Service with Registering Asp.Net Dependecy Injection — this is in Startup.ConfigureServices method
2- Configure with Mapping GrpcService in Asp.Net Middleware — this is in Startup.Configure method.

First, open Startup.cs and locate the ConfigureServices method;

```
 public void ConfigureServices(IServiceCollection services)
 {
 services.AddGrpc();
 }
```

We added AddGrpc extention method, by this way, it will inject grpc related classes for our application.
After that we should open gRPC api protocol for our application. In order to do that we should Map gRPC service into our endpoints.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
 app.UseEndpoints(endpoints =>
 {
 endpoints.MapGrpcService<HelloWorldService>();
```

As you can see that, we have configured our application as a grpc server. So now we are ready to run application.

Run the application
Change Run Profile to “GRPCHelloWorld” and Run the application
See 5001 https port worked — https://localhost:5001/

See the Logs

```
info: Microsoft.Hosting.Lifetime[0]
Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
Content root path: C:\Users\ezozkme\source\repos\grpc-examples\section1\GrpcHelloWorld\GrpcHelloWorldServer
```

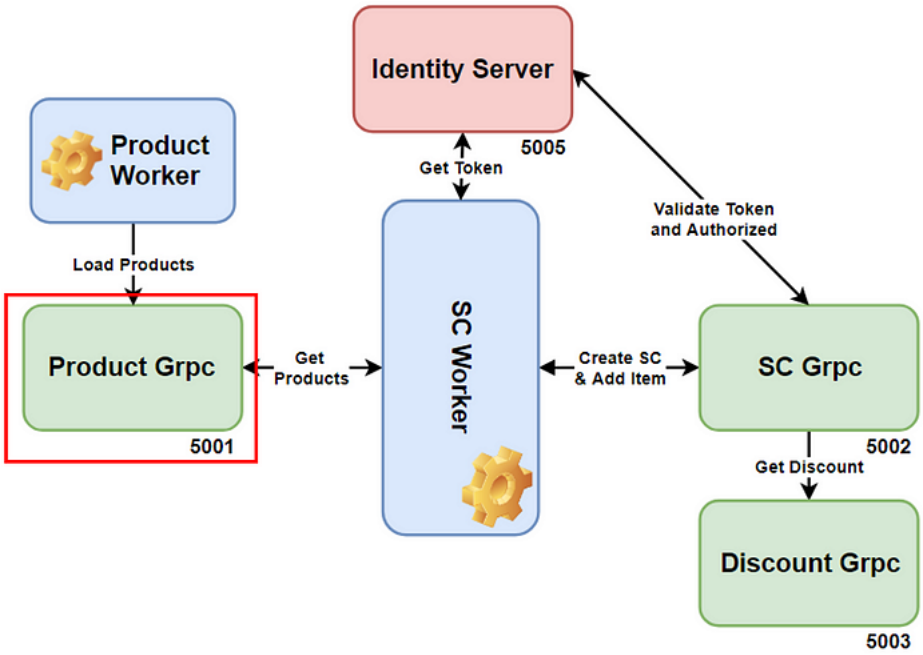
As you can see that port 5001 listening for gRPC calls. Now we can continue to develop client application.

Building Product Grpc Microservices for Exposing Product CRUD APIs

In this section, we are going to perform CRUD operations on Product Grpc microservices. We will exposing apis over the http2 grpc protocol for Product microservices. After that we will consume this grpc service methods from client application so we will develop to client application. This ProductGrpc microservices will be the first part of our big picture, so we will extend microservices with gRPC protocol in order to provide inter-service communication between microservices.

Big Picture

Let’s check out our big picture of the architecture of what we are going to build one by one.



In this section, as you can see the selected box, we are going to build Product Grpc Microservices for Exposing Product CRUD APIs with gRPC.

Let me give some brief information, We are going to;

- Create Product Grpc Microservices Project in Grpc Microservices Solution
  Set Product Grpc Microservices Database with Entity Framework In-Memory Database in Code-First Approach
- Seeding In-Memory Database with Entity Framework Core for ProductGrpc Microservices
- Developing product.proto ProtoBuf file for Exposing Crud Services in Product Grpc Microservices
- Generate Proto Service Class from Product proto File in Product Grpc Microservices
- Developing ProductService class to Implement Grpc Proto Service Methods in Product Grpc Microservices
- Create Client Console Application for Consuming Product Grpc Microservices
- Consume GetProductAsync Product Grpc Server Method from Client Console Application

Let’s take an action.

Create Product Grpc Microservices Project in Grpc Microservices Solution

We are going to Create Product Grpc Microservices Project in Grpc Microservices Visual Studio Solution.
First of all, we are going to create a new solution and project. So this is the first step to start development of our big picture;

Create Asp.Net Grpc Template Project : — HTTPS — https required for http2 tls protocol
— Solution name : GrpcMicroservices
— Project name : ProductGrpc

Lets Quick check the files again;
This project generated by default template of gRPC project;

We can Check the files;

- Nuget Packages
- Protos Folder
- Services Folder
- Startup.cs
- appsettings.json
- launchSettings

And you can see the Run profile of project, they created custom project run profile on https 5001 port.

So, as you can see that, grpc template handled all the things for us that we can proceed with database development of **ProductGrpc** microservices.

Developing product.proto ProtoBuf file for Exposing Crud Services in Product Grpc Microservices

We are going to develop “product.proto” ProtoBuf file for Exposing Crud Services in Product Grpc Microservices.
Let’s take an action.

First of all, we are going to start with deleting Greet proto file and GreetService. Of course we should delete from Startup endpoint middleware.
Delete greet.proto
Delete GreeterService
Comment on Startup
endpoints.MapGrpcService<ProductService>();

Now we can create “product.proto” file under the Protos folder.

Develop proto file
product.proto file;

```
syntax = “proto3”;option csharp_namespace = “ProductGrpc.Protos”;import “google/protobuf/timestamp.proto”;
 import “google/protobuf/empty.proto”;service ProductProtoService {
 rpc GetProduct (GetProductRequest) returns (ProductModel);
 rpc GetAllProducts (GetAllProductsRequest) returns (stream ProductModel);rpc AddProduct (AddProductRequest) returns (ProductModel);
 rpc UpdateProduct (UpdateProductRequest) returns (ProductModel);
 rpc DeleteProduct (DeleteProductRequest) returns (DeleteProductResponse);rpc InsertBulkProduct (stream ProductModel) returns (InsertBulkProductRespor
 }message GetProductRequest {
 int32 productId = 1;
 }message GetAllProductsRequest{
 }message AddProductRequest {
 ProductModel product = 1;
 }message UpdateProductRequest {
 ProductModel product = 1;
 }message DeleteProductRequest {
 int32 productId = 1;
 }message DeleteProductResponse {
 bool success = 1;
 }message InsertBulkProductResponse {
 bool success = 1;
 int32 insertCount = 2;
 }message ProductModel{
 int32 productId = 1;
 string name = 2;
 string description = 3;
 float price = 4;
 ProductStatus status = 5;
 google.protobuf.Timestamp createdTime = 6;
 }enum ProductStatus {
 INSTOCK = 0;
 LOW = 1;
 NONE = 2;
 }
```

Ok now, let me explain this proto file;

First, The syntax statement tells the protobuf compiler what syntax we are going to use. it’s important to specify which protobuf version we are using.
The second statement is optional and it tells the protobuf compiler to generate C# classes within the specified namespace: ProductGrpc.Protos.

After that, we have Defined a ProductProtoService.
The ProductProtoService has crud methods which will be the gRPC sevices. And along with this, it has Message classes.

We have defined generic model with ProductModel message type and use this type as a response objects.
For example for rpc GetProduct method we used GetProductRequest message as a request, and use ProductModel as a response.
rpc GetProduct (GetProductRequest) returns (ProductModel);

As the same way we have implemented Add/Update/Delete methods.

We have 1 server stream and 1 client stream methods which are GetAllProducts and InsertBulkProduct.

- rpc GetAllProducts (GetAllProductsRequest) returns (stream ProductModel);
- rpc InsertBulkProduct (stream ProductModel) returns (InsertBulkProductResponse);

As you can see that we put stream keyword according to server or client part of the message. We will see the implementation of this methods.
We also have enum type for ProductStatus, this will also support in proto files and also generates in consume classes.

So we are defining the ProductProtoService and This messages and services will be generated for accessing from our application. As you can see that, we have developed our contract based product.proto protobuf file. Now we can generate server codes from this file.

Developing ProductService class to Implement Grpc Proto Service Methods in Product Grpc Microservices

We are going to develop ProductService class to Implement Grpc Proto Service Methods in Product Grpc Microservices.
Let's take an action.

First of all, we should create a new class :
— Class Name : ProductService.cs

After that we should inherit from the ProtoService class which is generated from Visual Studio.
```
public class ProductService : ProductProtoService.ProductProtoServiceBase
```

Ok, now we can start to implement our main methods of product.proto contract file in the ProductGrpc microservices.

— Let me develop;
— GetProduct method;
```
public override async Task<ProductModel> GetProduct(GetProductRequest request,
 ServerCallContext context)
 {
 var product = await _productDbContext.Product.FindAsync(request.ProductId);
 if (product == null)
 {
 throw new RpcException(new Status(StatusCode.NotFound, $"Product with ID={request.ProductId} is not found."));
 }
 var productModel = _mapper.Map<ProductModel>(product);
 return productModel;
 }
```

In this method, we used **_productDbContext** in order to get product data from database. And return productModel object which is our proto message type class.

Also you can see that we should convert to Timestamp for datetime values. Because Timestamp is google welknow types for the datetime objects, we have to cast our datetime object to Timestamp by using FromDateTime method.
```
CreatedTime = Timestamp.FromDateTime(product.CreateTime)
```

Finally, we should not forget to mapping our ProductService class as a gRPC service in Startup class.
We are going to Register ProductService into aspnet pipeline to new grpc service.

Go to Startup.cs — Configure
```
app.UseEndpoints(endpoints =>
 {
 endpoints.MapGrpcService<ProductService>();
```