**Contents**

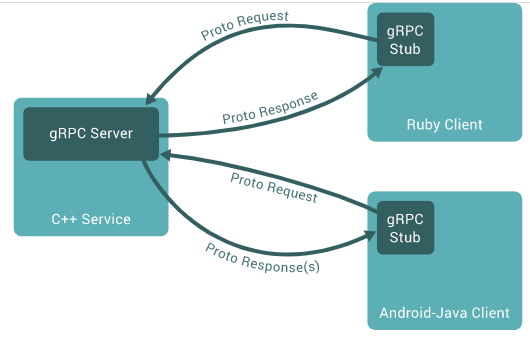
|  |  |  |  |
| --- | --- | --- | --- |
| Point | | Topic | Page |
| 1.0 |  | Introduction to gRPC……………………………………………………………………………. | 2 |
|  | 1.1 | How GRPC Works?................................................................................... | 3 |
| 2.0 |  | RPC Types……………………………………………………………………………………………. | 3 |
|  | 2.1 | Unary………………………………………………………………………………………………….. | 3 |
|  | 2.2 | Streaming……………………………………………………………………………………………. | 3 |
| 3.0 |  | Working with Protocol Buffers…………………………………………………………….. | 3 |
| 4.0 |  | Tooling support for .proto files……………………………………………………………. | 4 |
| 5.0 |  | Benefits of Adopting GRPC HTTP/REST Features………………………………….. | 5 |
| 6.0 |  | Core concepts and architecture…………………………………………………………… | 5 |
| 7.0 |  | Synchronous vs. asynchronous……………………………………………………………. | 6 |
| 8.0 |  | Authentication…………………………………………………………………………………….. | 7 |
| 9.0 |  | Supported auth mechanisms……………………………………………………………….. | 7 |
| 10.0 |  | Channels……………………………………………………………………………………………… | 7 |
| 11.0 |  | REST vs. gRPC………………………………………………………………………………………. | 7 |
| 12.0 |  | Create a simple gRPC application using in c#........................................... | 8 |
|  | 12.1 | Create a gRPC service and client application using with C # language….. | 8 |
|  | 12.2 | Configure gRPC in ASP.NET Core………………………………………………………….. | 11 |
|  | 12.3 | Create a gRPC client in ASP.NET Core…………………………………………………… | 12 |
|  |  |  |  |

**1.0 Introduction to gRPC**

gRPC is a RPC platform developed by Google which was announced and made open source in late Feb 2015. gRPC are a recursive acronym which means, gRPC Remote Procedure Call.

gRPC has two parts, the gRPC protocol, and the data serialization. By default gRPC utilizes Protobuf for serialization, but it is pluggable with any form of serialization .

In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object, making it easier for you to create distributed applications and services. As in many RPC systems, gRPC is based around the idea of defining a service, specifying the 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 (referred to as just a client in some languages) that provides the same methods as the server.



gRPC clients and servers can run and talk to each other in a variety of environments - from servers inside Google to your own desktop - and can be written in any of gRPC’s supported languages. So, for example, we can easily create a gRPC server in Java/C# with clients in Go, Python, or Ruby. In addition, the latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into our applications.

**1.1 How GRPC Works?**

* A client application is able to call methods directly on a server-side application present on other machines.
* Service is defined, methods are specified which can be further remotely called with their parameters and return types.
* On the other hand, the server runs a GRPC server to handle client calls.
* It uses protocol buffers as the Interface Definition Language to enable communication between two different systems used for describing the service interface and the structure of payload messages.
* HTTP/2 – GRPC is basically a protocol built on top of HTTP/2. HTTP/2 is used as a transport.
* Protobuf serialization – Messages that we serialize both for the request and response are encoded with protocol buffers.
* Clients open one long-lived connection to GRPC server.
* A new HTTP/2 stream for each RPC call.
* Allows Client-Side and Server-Side Streaming.
* Bidirectional Streaming.

**2.0 RPC Types**

gRPC offers two essential types for client server communication.

**2.1 Unary**

Essentially these are synchronous requests made to the gRPC server with a single request that blocks until a response is received.

**2.2 Streaming**

Streaming is really powerful and can be accomplished in three different configurations: **client pushing messages to a stream; server pushing messages to a stream; or bidirectional**, where client and server are both sending data in two streams in the same method. In all cases the client initiates the RPC method.

Streams don’t provide any acknowledgement of receipt until the stream completes, which can add complexity when the system needs to cope with node failures or network partitions. This can be mitigated by using a bidirectional stream to return ACKs. If a server is given a chance to kill a connection gracefully a message will be returned indicating the last received message.

**3.0 Working with Protocol Buffers**

By default, gRPC uses Protocol Buffers, Google’s mature open source mechanism for serializing structured data (although it can be used with other data formats such as JSON).

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

|  |
| --- |
| **syntax** = "**proto3**";  **message** **Person** {  **string** name = 1;  **int32** id = 2;  **bool** has\_ponycopter = 3;  } |

* The first line of the file specifies that you're using proto3 syntax: if you don't do this the protocol buffer compiler will assume you are using proto2. This must be the first non-empty, non-comment line of the file.
* The Person message definition specifies three fields (name/value pairs), one for each piece of data that you want to include in this type of message. Each field has a name and a type

By default, server and client assets are generated for each ***\*.proto*** file included in the **<Protobuf>** item group. To ensure only the server assets are generated in a server project, the **GrpcServices** attribute is set to Server.

|  |
| --- |
| <ItemGroup>  <Protobuf Include="Protos\greet.proto" GrpcServices="Server" />  </ItemGroup> |

Similarly, the attribute is set to **Client** in **client projects.**

**4.0 C# Tooling support for .proto files**

The tooling package Grpc.Tools is required to generate the C# assets from **\*.proto** files. The generated assets (files):

This package is required by **both the server and client projects.** The **Grpc.AspNetCore** metapackage includes a reference to Grpc.Tools. Server projects can add **Grpc.AspNetCore** using the Package Manager in Visual Studio or by adding a **<PackageReference>** to the project file:

<PackageReference Include="Grpc.AspNetCore" Version="2.28.0" />

Client projects should directly reference Grpc.Tools alongside the other packages required to use the gRPC client. The tooling package isn't required at runtime, so the dependency is marked with PrivateAssets="All":

|  |
| --- |
| <PackageReference Include="Google.Protobuf" Version="3.11.4" />  <PackageReference Include="Grpc.Net.Client" Version="2.28.0" />  <PackageReference Include="Grpc.Tools" Version="2.28.1"> |

**5.0 Benefits of Adopting GRPC HTTP/REST Features**

* Easy to understand.
* Web infrastructure already built on top of HTTP.
* Great tooling for testing, inspection, and modification.
* Loose coupling between clients/server makes changes easy.
* High-quality HTTP implementations in every language.

**6.0 Core concepts and architecture**

Many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. By default, gRPC uses protocol buffers as the **Interface Definition Language (IDL)** for describing both the service interface and the structure of the payload messages. It is possible to use other alternatives if desired.

|  |
| --- |
| **service** HelloService {  **rpc** SayHello (HelloRequest) **returns** (HelloResponse);  }  **message** **HelloRequest** {  **string** greeting = 1;  }  **message** **HelloResponse** {  **string** reply = 1;  } |

gRPC lets you define four kinds of service method:

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

|  |
| --- |
| **rpc** SayHello(HelloRequest) **returns** (HelloResponse); |

* 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.

|  |
| --- |
| **rpc** LotsOfReplies(HelloRequest) **returns** (stream HelloResponse); |

* 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.

|  |
| --- |
| **rpc** LotsOfGreetings(stream HelloRequest) **returns** (HelloResponse); |

* 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. The order of messages in each stream is preserved.

|  |
| --- |
| **rpc** BidiHello(stream HelloRequest) **returns** (stream HelloResponse); |

**7.0 Synchronous vs. asynchronous**

Synchronous RPC calls that block until a response arrives from the server are the closest approximation to the abstraction of a procedure call that RPC aspires to. On the other hand, networks are inherently asynchronous and in many scenarios it’s useful to be able to start RPCs without blocking the current thread.

The gRPC programming API in most languages comes in both synchronous and asynchronous flavors. You can find out more in each language’s tutorial and reference documentation (complete reference docs are coming soon).

**8.0 Authentication**

gRPC is designed to work with a variety of authentication mechanisms, making it easy to safely use gRPC to talk to other systems. You can use our supported mechanisms - SSL/TLS with or without Google token-based authentication - or you can plug in your own authentication system by extending our provided code.

gRPC also provides a simple authentication API that lets you provide all the necessary authentication information as Credentials when creating a channel or making a call.

**9.0 Supported auth mechanisms**

The following authentication mechanisms are built-in to gRPC:

* **SSL/TLS:**  gRPC has SSL/TLS integration and promotes the use of SSL/TLS to authenticate the server, and to encrypt all the data exchanged between the client and the server. Optional mechanisms are available for clients to provide certificates for mutual authentication.
* **ALTS**: gRPC supports ALTS as a transport security mechanism, if the application is running on Google Cloud Platform (GCP). For details, see one of the following language-specific pages: ALTS in C++, ALTS in Go, ALTS in Java, ALTS in Python.
* **Token-based authentication with Google**: gRPC provides a generic mechanism (described below) to attach metadata based credentials to requests and responses. Additional support for acquiring access tokens (typically OAuth2 tokens) while accessing Google APIs through gRPC is provided for certain auth flows.

**10.0 Channels**

A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub. Clients can specify channel arguments to modify gRPC’s default behavior, such as switching message compression on or off. A channel has state, including connected and idle.

Creating a channel is an expensive operation compared to invoking a remote call so in general we should reuse a single channel for as many calls as possible.

**11.0 REST vs. gRPC**

Biggest differences between REST and gRPC are the format of the payload. REST messages typically contain JSON. This is not a strict requirement, and in theory we can send anything as a response.

gRPC, on the other hand, accepts and returns Protobuf messages. I will discuss the strong typing later, but just from a performance point of view, Protobuf is a very efficient and packed format. JSON, on the other hand, is a textual format.

REST is depends heavily on HTTP (usually HTTP 1.1) and the request-response model. On the other hand, gRPC uses the newer HTTP/2 protocol.

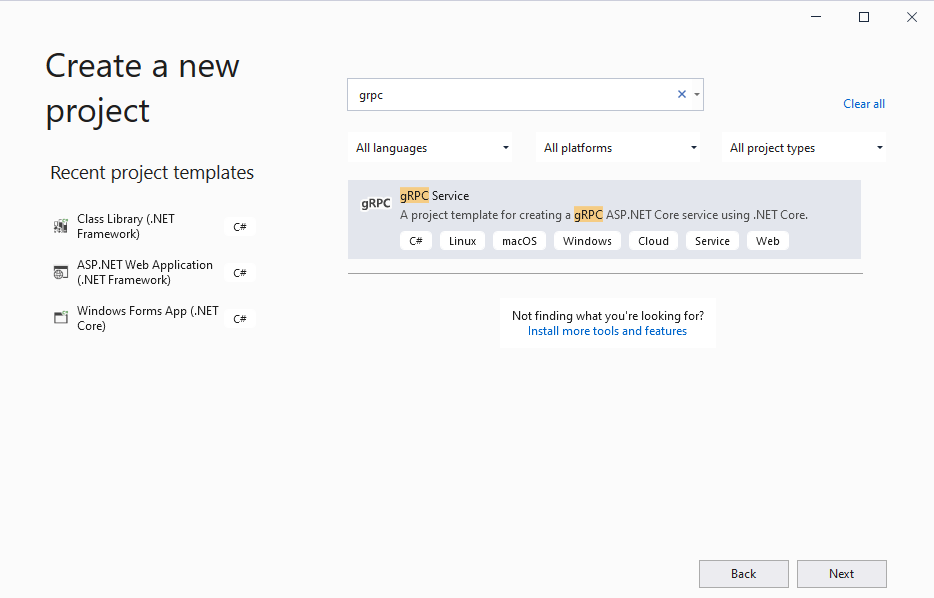
REST is an interesting API. It is built very tightly on top of HTTP. It doesn't just use HTTP as a transport, but embraces all its features and builds a consistent conceptual framework on top of it.

gRPC is to have services with clear interfaces and structured messages for requests and responses. This model translates directly from programming language concepts like interfaces, functions, methods, and data structures. It also allows gRPC to automatically generate client libraries classes.

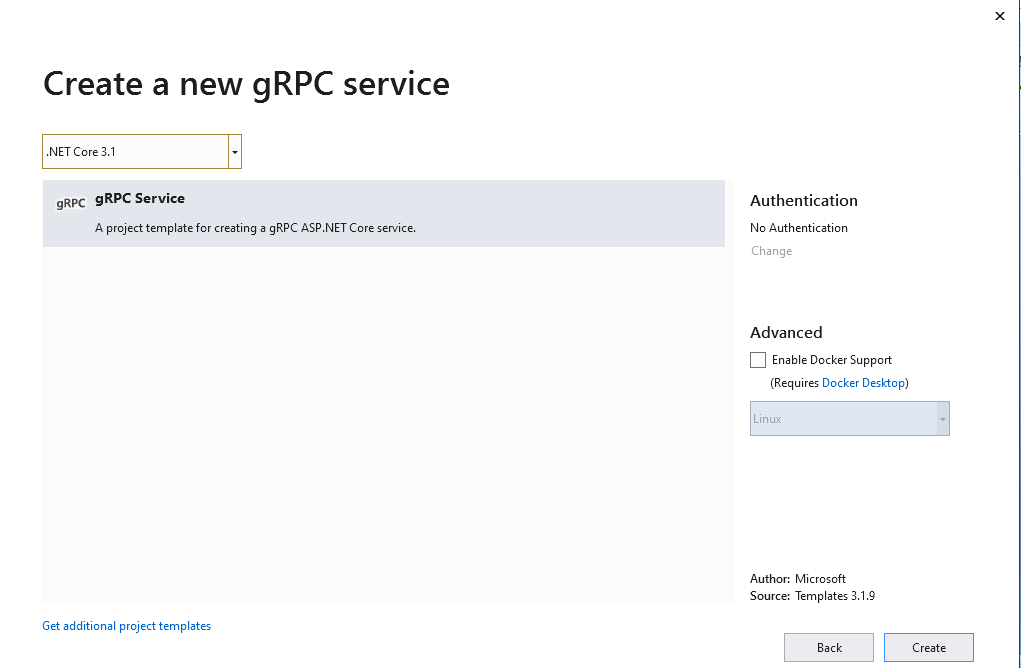
**12.0 Create a simple gRPC application using in c#**

**12.1 Create a gRPC service and client application using with C # language.**

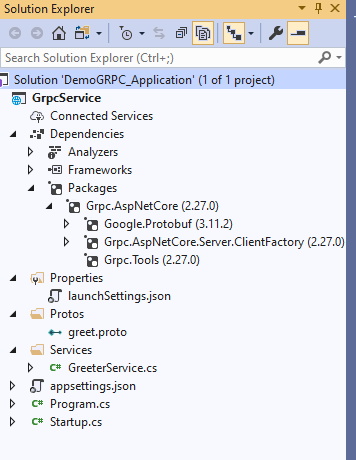
1. First of all we will open the visual studio 2017 or above version. And then create a new project and search the gRPC project template and choose the gRPC project template then click on next button.



1. Then click on create button with default setting.



1. Default empty project is created. It look like below screenshot:



* When we expand Grpc.AspNetCore, we should see the following three packages:

1. Google.Protobuf
2. Grpc.AspNetCore.Server.ClientFactory
3. Grpc.Tools

* And also we see the solution will be created two folders by default –

1. one named Protos, which will contain the .proto files,
2. And the other named Services, which will contain of strongly-typed service classes.

* Solution by default gRPC service created in project is **GreeterService**, and it is stored inside the **Services folder**. The default **.proto** file created for you with the gRPC project looks like this:

**Example of greet.proto:** We have already discussed about the mentioned below code (please find the above point number **3.0 and 6.0**).

|  |
| --- |
| syntax = "proto3";  option csharp\_namespace = "GrpcService";  package greet;  // The greeting service definition.  service Greeter {  // Sends a greeting  rpc SayHello (HelloRequest) returns (HelloReply);  }  // The request message containing the user's name.  message HelloRequest {  string name = 1;  }  // The response message containing the greetings.  message HelloReply {  string message = 1;  } |

The default Greeter service generated by Visual Studio looks like this

|  |
| --- |
| public class GreeterService : Greeter.GreeterBase  {  private readonly ILogger<GreeterService> \_logger;  public GreeterService(ILogger<GreeterService> logger)  {  \_logger = logger;  }  public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext context)  {  return Task.FromResult(new HelloReply  {  Message = "Hello " + request.Name  });  }  } |

**12.2 Configure gRPC in ASP.NET Core**

By default enable the gRPC support in our ASP.NET Core web application by calling the **AddGrpc()** method in the **ConfigureServices** method as shown below.

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

To add a gRPC service to the routing pipeline, we should call the **MapGrpcService()** method as shown below.

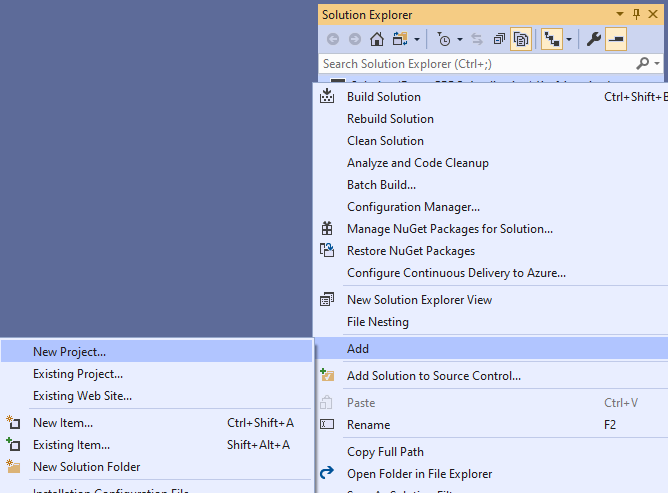
|  |
| --- |
| public void Configure(**IApplicationBuilder** app, **IWebHostEnvironment** env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  app.**UseRouting**();             app.**UseEndpoints**(endpoints =>             {                 endpoints.**MapGrpcService**<**GreeterService**>();                  endpoints.**MapGet**("/", async context =>                 {                     await context.**Response**.**WriteAsync**("Hello World!");                 });             });  } |

**Note**: above the codes are mentioned in **Startup.cs** file.

**12.3 Create a gRPC client in ASP.NET Core**

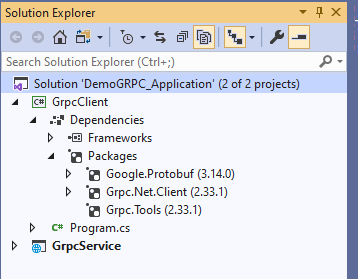
Now we will create the gRPC client application in same solution project folder.

1. Right click on solution project and then choose the “Create new project” option.
2. In the “Create new project” window, I have selected the “Console App (.NET Core)” from the list of templates displayed.
3. Click Next



1. Now we need to install the following package from NuGet packages manager
2. **Grpc.Tools** – contains the necessary types that can be used to provide tooling support for protobuf files
3. **Grpc.Net.Client** – contains the .NET Core client that can be used to establish a connection channel and send messages to a particular endpoint
4. **Google.Protobuf** – contains the protobuf APIs that can be leveraged for writing protobuf files

|  |
| --- |
| **Install**-**Package** **Grpc**.**Net**.**Client** **Install**-**Package** **Google**.**Protobuf** **Install**-**Package** **Grpc**.**Tools** |



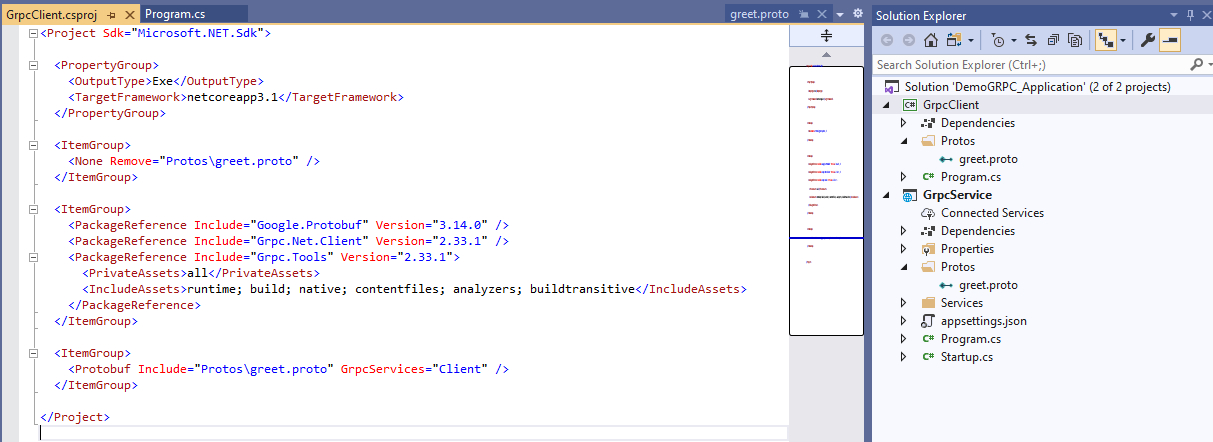
All require package are installed in our gRPC client application.

1. Now add the “**Protos**” folder under the **GrpcClient** project.
2. Then copy the “**greet.proto**” file from GrpcService project under the **Protos** folder and thepaste to the “**Protos**” folder under the **GrpcClient** project.

(Basically proto files are defined for contract between server and client. This file should be same for both server side and client side; otherwise we'll receive the exception)

1. Open the .proj file of the client application on edit mode and insert the following line inside the <**ItemGroup**> tag.

**<Protobuf** Include="Protos\greet.proto" GrpcServices="Client" **/>**



1. Now we call the gRPC service method in the gRPC client application.
   1. We create a custom async method and into the main method. Sample code is mention in below:

|  |
| --- |
| class Program  {  static async Task Main(string[] args)  {  await PrintName();  Console.ReadKey();  }  public static async Task PrintName()  {  string name;  using var grpcChannel = GrpcChannel.ForAddress("https://localhost:5001");  var grpcClient = new Greeter.GreeterClient(grpcChannel);  Console.Write("Enter your name:");  name = Console.ReadLine();  var reply = await grpcClient.SayHelloAsync(new HelloRequest { Name = name });  Console.WriteLine(reply.Message);  }  } |

* OutPut result:

