**<gRPC > with .Net Core**

**And**

**Its comparative study with other .Net <RPC frameworks>/<REST>**

**Created By:**

Riddhi Roy Choudhury Debasish Das

**Last Updated on:** 12/24/2020

Table of Contents

[**1.** **Introduction To GRPC:** 3](#_Toc59708728)

[**2.** **GRPC Serialization and Protocol Buffer:** 4](#_Toc59708729)

[2.1. What is Protocol buffer 4](#_Toc59708730)

[2.2. GRPC and Protocol Buffer 4](#_Toc59708731)

[**3.** **Comparative Study between RPC And REST:** 6](#_Toc59708732)

[3.1. RPC 6](#_Toc59708733)

[3.2. REST 8](#_Toc59708734)

[**4.** **Microsoft’s implementation of SOA (WCF,WEB API & gRPC):** 10](#_Toc59708735)

[4.1. GRPC As Future of WCF 10](#_Toc59708736)

[4.2. GRPC vs REST : 12](#_Toc59708737)

[4.3. Where to use gRPC? 13](#_Toc59708738)

[**5.** **GRPC Core Concepts & Architecture** 14](#_Toc59708739)

[5.1. Overview: 14](#_Toc59708740)

[5.2. Using the API: 15](#_Toc59708741)

[5.3. Synchronus vs Asynchronus: 15](#_Toc59708742)

[5.4. RPC Life Cycle 16](#_Toc59708743)

[5.5. Deadlines/Timeouts: 17](#_Toc59708744)

[5.6. Deadlines/Timeouts: 17](#_Toc59708745)

[5.7. Cancelling an RPC 17](#_Toc59708746)

[5.8. Channels 17](#_Toc59708747)

[**6.** **Create a simple gRPC application using in c#** 18](#_Toc59708748)

[6.1. Creating a gRPC service : 18](#_Toc59708749)

[6.1.1. Understanding the proto file: 21](#_Toc59708750)

[6.1.2. Code generation using .proto file: 21](#_Toc59708751)

[6.1.3. Generated c# Asset: 23](#_Toc59708752)

[6.1.4. Configure gRPC in ASP.NET Core 24](#_Toc59708753)

[6.2. Creating a gRPC Client: 25](#_Toc59708754)

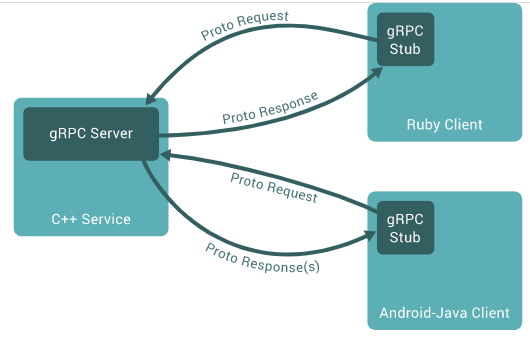
[**7.** **Hosting GRPC :** 29](#_Toc59708755)

[**8.** **Sample Project github refference** 32](#_Toc59708756)

## **Introduction To GRPC:**

GRPC is an open source Remote Procedure Call system Initially developed at google on 2015. GRPC uses protocol buffers as both its Interface Definition Language (IDL) and as its underlying message interchange format.

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.



The beauty with GRPC lies in , it declares the service is language agnostic Interface Definition Language (IDL) which is Protobuff in this case. Then language specific binding and classes (to write or call the service) can be generated from the IDL in server and client side based on the language server or client prefers to use. Hence, 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, you can easily create a gRPC server in Java 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 your applications.

## **GRPC Serialization and Protocol Buffer:**

## What is Protocol buffer

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages.

Protocol buffers currently support generated code in Java, Python, Objective-C, and C++. With our new proto3 language version, you can also work with Dart, Go, Ruby, and C#, with more languages to come.

References for how to use Protocol buffer with c# : <https://developers.google.com/protocol-buffers/docs/csharptutorial>

## GRPC and Protocol Buffer

gRPC’s secret sauce lies in the way the serialization is handled. It is based on [Protocol Buffers](https://developers.google.com/protocol-buffers/docs/overview), an open-source mechanism for serializing structured data, which is language and platform neutral. Similar to XML, Protocol Buffers are verbose and descriptive. But they are smaller, faster, and more efficient than other wire-format protocols. Any custom data type that needs to be serialized will be defined as a Protocol Buffer in gRPC.

By default, gRPC uses [Protocol Buffers](https://developers.google.com/protocol-buffers/docs/overview), Google’s mature open-source mechanism for serializing structured data (although it can be used with other data formats such as JSON). Here is a quick intro to how it works. If you are already familiar with protocol buffers, feel free to skip ahead to the next section.

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 is a simple example:

**message** **Person** {

**string** name = 1;

**int32** id = 2;

**bool** has\_ponycopter = 3;

}

Then, once you’ve specified your data structures, you use the protocol buffer compiler protoc to generate data access classes in your preferred language(s) from your proto definition. These provide simple accessors for each field, like name() and set\_name(), as well as methods to serialize/parse the whole structure to/from raw bytes. So, for instance, if your chosen language is C++, running the compiler on the example above will generate a class called Person. You can then use this class in your application to populate, serialize, and retrieve Person protocol buffer messages.

You define gRPC services in ordinary proto files, with RPC method parameters and return types specified as protocol buffer messages:

*// The greeter 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;

}

gRPC uses protoc (protobuff compiler) with a special gRPC plugin to generate code from your proto file: you get generated gRPC client and server code, as well as the regular protocol buffer code for populating, serializing, and retrieving your message types. You’ll see an example of this below.

## **Comparative Study between RPC And REST:**

All these three rudimentarily refers to the same thing called Web Service. Now in simple terms what is a Web Service? Web Services are the key point of Integration for different applications belonging to different Platforms, Languages, systems. are set of platform They are independently exposed functions which can be used from remote server over the Internet. There are basically two parties involved in this, one which provides a set of exposed functions and the another one ,commonly known as web services consumers, is the party which uses the functionality and services provided by web services providing party.

There are different methods available in market for providing web service, but most common are if we start categorizing from high to low level:

* **RPC**
* **REST**
* These two are fundamental ways how two systems can communicate to each other.

## RPC

The RPC can then further be sub-categorized in to the following :

* XML based RPC.
* SOAP based RPC.
* GRPC (A recent one compared to the above two)

**XML-RPC:**

It's a [spec](http://xmlrpc.com/spec.md) and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

It's remote procedure calling using HTTP as the transport and XML as the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

It doesn’t set out to be the solution to every problem. Instead it seeks to be a simple and effective means to request and receive information.

**SOAP Based RPC:**

SOAP (formerly an acronym for Simple Object Access Protocol) is a messaging [protocol](https://en.wikipedia.org/wiki/Protocol_(computing)) specification for exchanging structured information in the implementation of [web services](https://en.wikipedia.org/wiki/Web_service) in [computer networks](https://en.wikipedia.org/wiki/Computer_network). Its purpose is to provide [extensibility](https://en.wikipedia.org/wiki/Extensibility), [neutrality](https://en.wikipedia.org/wiki/Neutrality_(philosophy)), verbosity and independence. It uses [XML Information Set](https://en.wikipedia.org/wiki/XML_Information_Set) for its [message format](https://en.wikipedia.org/wiki/Message_format), and relies on [application layer](https://en.wikipedia.org/wiki/Application_layer) protocols, most often [Hypertext Transfer Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol) (HTTP), although some legacy systems communicate over [Simple Mail Transfer Protocol](https://en.wikipedia.org/wiki/Simple_Mail_Transfer_Protocol) (SMTP), for message negotiation and transmission.

In the initial days of developing web service , it all started with XML-RPC. But XML-RPC was problematic, because ensuring data types of XML payloads is tough. In XML, a lot of things are just strings, so you need to layer meta data on top in order to describe things such as which fields correspond to which data types. This became part of the basis for SOAP (Simple Object Access Protocol). XML-RPC and SOAP, along with custom homegrown solutions, dominated the API landscape for a long time and were all RPC-based HTTP APIs.

**GRPC**

GRPC is newest addition in the row by Google which is the document is all about. It uses Protocol Buffer to encode/Decode messages passed between different systems, and also to define it’s interface .

A modern RPC implementation is [gRPC](https://grpc.io/), which can easily be considered modern (and drastically better) SOAP. It uses a data format called [ProtoBuff](https://developers.google.com/protocol-buffers/), which requires a schema as well as the data instance, much like the WSDL in SOAP.

GRPC focuses on making single interactions as quick as possible, thanks to HTTP/2, and the fact that Protobuff packs down smaller than JSON, but JSON can also be used easily enough.

More to come on this later in this document.

## REST

REST is an architectural \*style\*. RESTful is another term to define REST web services.

Representational State Transfer or REST basically means that each unique URL is a representation of some object. You can get the contents of that object using an HTTP GET, to delete it, you then might use a POST, PUT, or DELETE to modify the object (in practice most of the services use a POST for this).

It is not necessary to use XML as a data interchange format in REST. The format of information(representation) returned can be in XML, JSON, plain text or even in HTML format.

REST architecture is basically focused on two things: Resources and Interface.

Resources are application’s state and functionality which is represented by a unique URL. The resources share a uniform interface to transfer the state between the client and server.

For example the URL, http://example.com/product/11 can be a resource.Suppose, GET method is used to retrieve product detail from that URL, POST method is used to modify the production information and DELETE method can be used to delete the product from the same URL. Here, the HTTP methods works as a interface to access the resources. All resources implement the same uniform interface. The standard methods — in this case, the HTTP verbs — are mapped to resource-specific semantics.

|  |  |
| --- | --- |
| **RPC** | **REST** |
| **Remote Procedure Call** (RPC) is way to describe a mechanism that lets you call a procedure in another process and exchange data by message passing. It typically involves generating some method stubs on the client process that makes making the call appear local, but behind the stub is logic to marshall the request and send it to the server process. The server process then unmarshalls the request and invokes the desired method before repeating the process in reverse to get whatever the method returns back to the client. HTTP is sometimes used as the underlying protocol for message passing, but nothing about RPC is inherently bound to HTTP. Remote Method Invocation (RMI) is closely related to RPC, but it takes remote invocation a step further by making it object oriented and providing the capability to keep references to remote objects and invoke their methods. | **Representational State Transfer** (REST) is an architectural style that imposes a particular set of constraints on an interface to achieve a set of goals. REST enforces a client/server model, where the client is interested in gaining information and acting on a set of resources that are managed by the server. The server tells the client about resources by providing a representation of one or more resources at a time and providing actions that can either get a new representation of resources or manipulate resources. All communication between the client and the server must be stateless and cachable. Implementations of a REST architecture are said to be RESTful. |

|  |  |
| --- | --- |
| SOAP | REST |
| * Lightweight – not a lot of extra xml markup (which a soap envelop has) * Human Readable Results * Easy to build – no toolkits required * API Flexibility and simplicity - REST has significant benefits over SOAP-based services. Any developer can figure out how to create and modify a URI to access different Web resources. SOAP, on the other hand, requires specific knowledge of a new XML specification, and most developers will need a SOAP toolkit to form requests and parse the results. * Another benefit of the RESTful interface is that requests and responses can be short. SOAP requires an XML wrapper around every request and response. While SOAP proponents argue that strong typing is a necessary feature for distributed applications. | * Easy to consume – sometimes * Has defined Standard for exchanging data for ex: type checking, adhering to a contract * SOAP provides relatively stronger typing since it has a fixed set of supported data types. It therefore guarantees that a return value will be available directly in the corresponding native type in a particular platform * SOAP is more secure than REST as it uses WS-Security for transmission along with Secure Socket Layer. |

## **Microsoft’s implementation of SOA (WCF,WEB API & gRPC):**

Windows Communication Foundation(WCF) is a framework built by MIcrosoft in early 2000s for building web services. WCF mainly uses SOAP to enable interoperability with other systems using protocols like HTTP/1.1, Net.TCP, and so on. WCF was designed to support both SOAP and REST based on the type of binding specified. WCF SOAP is potentially a RPC service which encode/decode s data in XML or using DataContarct serializer.

gRPC is a modern framework for building networked services and distributed applications. Imagine the performance of Windows Communication Foundation (WCF) NetTCP bindings, combined with the cross-platform interoperability of SOAP.

As .Net framework is gradually getting replaced by .Net core, so any SOAP service which was previously built with WCF can now be migrated in to .Net core gRPC. In other words, Dot Net core GRPC is the new replacement for WCF SOAP services while any RESTful API built with WCF REST can be considered to be migrated to regular ASP.NET Core MVC Web API application.  Also, any RESTful services created with Webapi previously can now be migrated or replaced with .Net Core Web Apis.

But Note: GRPC is not conceptually a replacement for services which were built with REST previously. REST architechture is totally different than GRPC (potentially a RPC service).Instead GRPC is more similar to WCF SOAP services design. Following are the areas where GRPC can be compared to WCF SOAP . We will go step by step here how we implement a service in WCF vs GRPC.

## GRPC As Future of WCF

**Binding:**

In WCF, you define one or several endpoints to expose your services. Each endpoint consists of an address, a binding, and a contract. The address tells where to find the service and the binding defines the transport protocol to use when communicating with it, how to encode the messages, and what security mechanisms to use. You can choose among a number of system-provided bindings or create your own custom ones.

In the .NET implementations of gRPC, the communication is done over HTTP/2. No other transport protocols are supported. On top of the transport layer, there's the concept of a channel. Under the hood, the channel takes care of connecting to the server and handles things such as load balancing and connection pooling. It provides a single virtual connection, which may be backed by several physical connections internally, to a conceptual endpoint. As an application developer, you don't really need to bother with the details of how this is implemented. You write your code against the generated stubs rather than dealing with the channel directly.

**Contracts:**

In GRPC the stubs are generated based on a .proto file, which is nothing but a plain text file with a .proto file extension. It defines the service contract and the structure of the payload messages to be exchanged between the client and the server. Similary in WCF we have something called WSDL, XSD and MEX – that describes the metadata of the WCF SOAP service.

Like the way in WCF we share the WSDL file with client and server to generate the client side and server side stubs, for gRPC we need to share the .proto file to generate the code in languages preferred.

**Code Generation:**

Grpc.Tools is a development dependency that integrates with MSBuild to provide automatic code generation of the .proto file as part of the build process. In order for any code to be generated, you also need to add a <Protobuf> element to your project file where you reference the .proto file.

Grpc.Tools is already packaged with protoc compiler to generate c# code from the .proto file. Now, If you build the project and look in the obj/Debug/netstandard2.x folder (replace Debug with Release or whatever build configuration you may be using) inside the project directory, you should see two generated files named Calculator.cs and CalculatorGrpc.cs if you are generating code for Calculator.proto file. The former contains the CalculatorRequest and CalculatorReply message types and the latter contains the stubs for the gRPC service and client.

Th above is similar to what we do in WCF SOAP . We add service reference using the service endpoint or pointing to the WSDL file to generate the client side code. Or, we also use tools like WCF Red/Blue to generate both Server and client side code.

**Services:**

The next step is to implement the service methods. Create another class library that targets the same version of .NET Standard as the previously created library where the .proto file and the generated code files are located and add a reference to this project. Then add a class that extends the abstract CalculatorService.CalculatorServiceBase class in CalculatorGrpc.cs. Unlike in WCF, there's no service interface to be implemented in gRPC. Instead, code generation using protoc and overriding methods of the generated base class is the standard practice.

**Hosting:**

Once you have implemented the service, you need to host it in a process. Hosting a Grpc.Core service in a managed process is very similar to hosting a WCF service. Instead of creating a ServiceHost to which you add endpoints, you create a Grpc.Core.Server to which you add services and ports.

Though there are lot of similarities the way WCF SOAP and gRPC is implemented and GRPC is future replacement for WCF in .Net Core - there are few basic differences. Here are those:

* GRPC does not use SOAP to mediate between client and service over http. WCF supports SOAP.
* GRPC is only concerned with RPC style communication. WCF supports and promotes REST and POX style services in addition to RPC.
* GRPC provides support for multiple programming languages. WCF supports C# (and the other .net languages).
* GRPC uses protobuf for on-wire serialization, WCF uses either XML/JSON or windows binary.
* GRPC is open source

In short:

GRPC seems a much more focused services framework, it does one job really well and on multiple platforms.

WCF much more general purpose, but limited to .net for the time being (WCF is being ported to .net core but at time of writing only client side functionality is on .net core)

## GRPC vs REST :

Now, let’s do a quick comparison of gRPC and REST to see their differences.

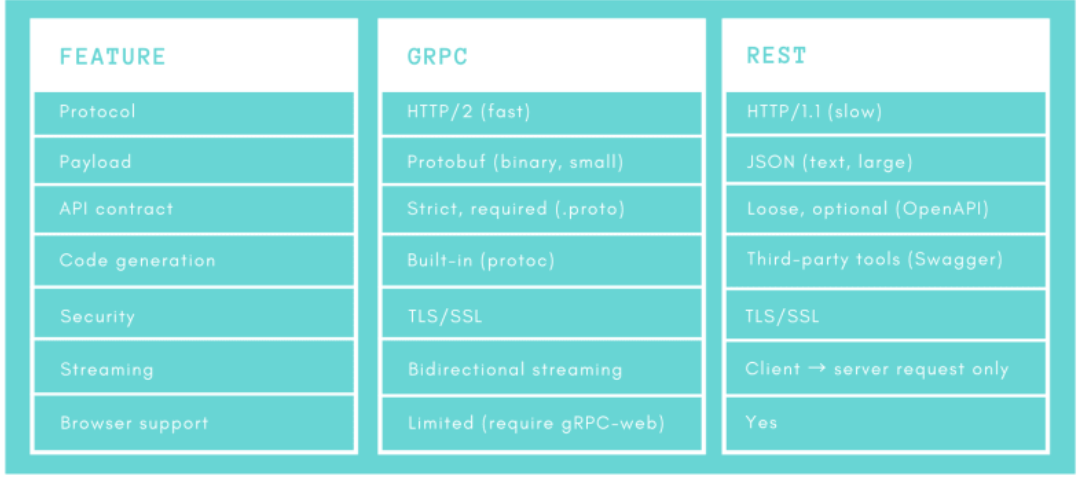
1. First, gRPC uses HTTP/2 which is, as you know, much faster than HTTP/1.1 used in REST by default. Note that today we can enable HTTP/2 in REST as well, but normally it often goes with HTTP/1.1. You can read more about how to enable HTTP/2 for REST in these articles:

* [Migrate your REST API to HTTP/2](https://blog.usejournal.com/migrating-your-rest-apis-to-http-2-why-and-how-8caee7d798fb)
* [HTTP/2 - Benefits for REST API](https://blog.restcase.com/http2-benefits-for-rest-apis/)
* [HTTP/2 - Adventure in Go world](https://posener.github.io/http2/)

1. mSecond, gRPC uses Protocol buffer to serialize payload data, which is binary and smaller, while REST uses JSON, which is text and larger.
2. The API contract in gRPC is strict, and required to be clearly defined in the proto file. 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.

While in REST, it’s often loose and optional. We can define it via OpenAPI if we want, but it’s not mandatory.

1. Code generation is built-in in gRPC with the help of protocol buffer compiler. While in REST, we must use third-party tool like OpenAPI and Swagger.



1. Both gRPC and REST communications are secured with TLS/SSL.
2. Streaming is bidirectional in gRPC, while only 1 way request from client to server in REST.
3. So gRPC is better than REST for most of the things that we’ve mentioned so far. However, there’s one thing that REST is still better: That is browser support. While REST is fully supported by all browsers, the support for gRPC is limited and required [gRPC-web](https://github.com/grpc/grpc-web) with a proxy layer to convert between HTTP/1 and HTTP/2.

## Where to use gRPC?

* So gRPC has a lot of strength, but it also has its own weaknesses. So where and when should we use gRPC in order to take full advantage of it?
* As you might have guessed, microservices is where gRPC really shines, since it enables low-latency and high-throughput communication, as well as strong API contracts.
* gRPC is also suitable for polyglot environments, because it provides code generations out-of-the-box for many programming languages.
* Point-to-point real-time communication is also a good place for gRPC, since it has excellent support for bidirectional streaming.
* And finally, gRPC is a great choice for a network-constrained environment such as mobile application (android/ios), because of its light-weight message format.

## **GRPC Core Concepts & Architecture**

## Overview:

Like 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](https://developers.google.com/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);

## Using the API:

Starting from a service definition in a .proto file, gRPC provides protocol buffer compiler plugins that generate client- and server-side code. gRPC users typically call these APIs on the client side and implement the corresponding API on the server side.

* On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls. The gRPC infrastructure decodes incoming requests, executes service methods, and encodes service responses.
* On the client side, the client has a local object known as stub (for some languages, the preferred term is client) that implements the same methods as the service. The client can then just call those methods on the local object, wrapping the parameters for the call in the appropriate protocol buffer message type - gRPC looks after sending the request(s) to the server and returning the server’s protocol buffer response(s)

## Synchronus vs Asynchronus:

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.

## RPC Life Cycle

In this section, you’ll take a closer look at what happens when a gRPC client calls a gRPC server method. For complete implementation details, see the language-specific pages.

**Unary RPC:**

First consider the simplest type of RPC where the client sends a single request and gets back a single response.

1. Once the client calls a stub method, the server is notified that the RPC has been invoked with the client’s [metadata](https://grpc.io/docs/what-is-grpc/core-concepts/#metadata) for this call, the method name, and the specified [deadline](https://grpc.io/docs/what-is-grpc/core-concepts/#deadlines) if applicable.
2. The server can then either send back its own initial metadata (which must be sent before any response) straight away, or wait for the client’s request message. Which happens first, is application-specific.
3. Once the server has the client’s request message, it does whatever work is necessary to create and populate a response. The response is then returned (if successful) to the client together with status details (status code and optional status message) and optional trailing metadata.
4. If the response status is OK, then the client gets the response, which completes the call on the client side.

**Server streaming RPC**

A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client’s request. After sending all its messages, the server’s status details (status code and optional status message) and optional trailing metadata are sent to the client. This completes processing on the server side. The client completes once it has all the server’s messages.

**Client streaming RPC**

A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message. The server responds with a single message (along with its status details and optional trailing metadata), typically but not necessarily after it has received all the client’s messages.

**Bidirectional streaming RPC**

In a bidirectional streaming RPC, the call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline. The server can choose to send back its initial metadata or wait for the client to start streaming messages.

Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order. For example, a server can wait until it has received all of a client’s messages before writing its messages, or the server and client can play “ping-pong” – the server gets a request, then sends back a response, then the client sends another request based on the response, and so on.

## Deadlines/Timeouts:

gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated with a DEADLINE\_EXCEEDED error. On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.

Specifying a deadline or timeout is language specific: some language APIs work in terms of timeouts (durations of time), and some language APIs work in terms of a deadline (a fixed point in time) and may or maynot have a default deadline.

## Deadlines/Timeouts:

In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match. This means that, for example, you could have an RPC that finishes successfully on the server side (“I have sent all my responses!") but fails on the client side (“The responses arrived after my deadline!"). It’s also possible for a server to decide to complete before a client has sent all its requests.

## Cancelling an RPC

Either the client or the server can cancel an RPC at any time. A cancellation terminates the RPC immediately so that no further work is done.

## Channels

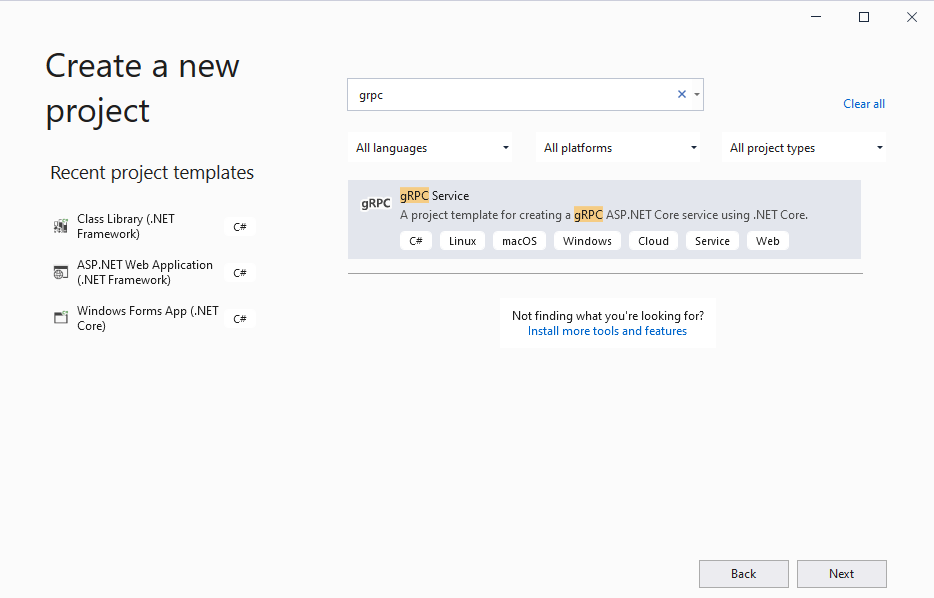
A gRPC channel provides a connection to a gRPC server on a specified host and port. It is used when creating a client stub. Clients can specify channel arguments to modify gRPC’s default behavior, such as switching message compression on or off. A channel has state, including connected and idle.

How gRPC deals with closing a channel is language dependent. Some languages also permit querying channel state.

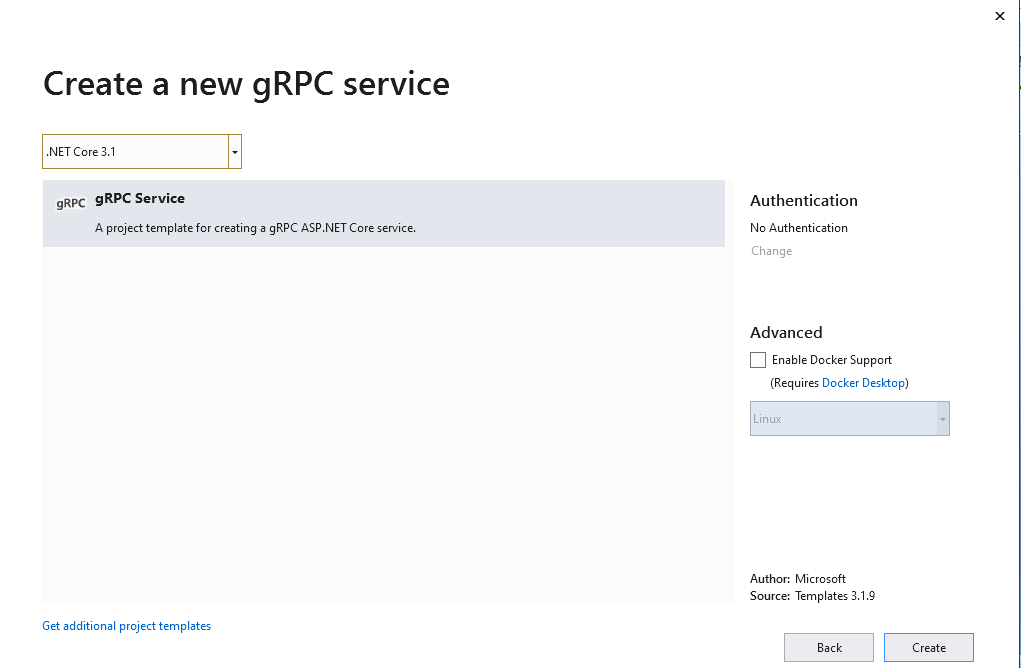
## **Create a simple gRPC application using in c#**

## Creating a gRPC service :

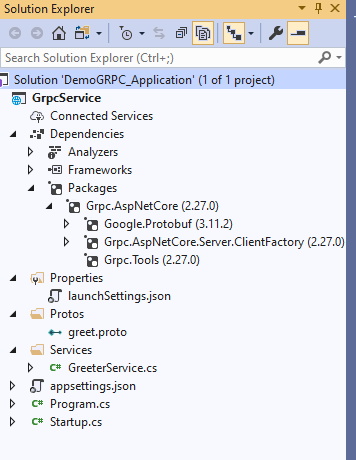
1. First of all we will open the visual studio 2017 or above version. And then create a new project and search the gRPC project template and choose the gRPC project template then click on next button.



1. Then click on create button with default setting.



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



Lets Start the discussion with the Proto File which can be found inside the Protos folder.

gRPC uses a contract-first approach to API development. Protocol buffers (protobuf) are used as the Interface Definition Language (IDL) by default. The \*.proto file contains:

* The definition of the gRPC service.
* The messages sent between clients and servers.

The proto file here contains the below code:

syntax = "proto3";

option csharp\_namespace = "GrpcTestService";

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;

}

### Understanding the proto file:

* 1. The syntax tells which protobuff version we are using. Note here, only protobuff version>=3 supports code generation in C# , thus allowing gRPC to work using .Net core.
  2. In C#, your generated classes will be placed in a namespace matching the package name if csharp\_namespace is not specified. In our example, the csharp\_namespace option has been specified to override the default, so the generated code uses a namespace of GrpcTestService instead of greet.
  3. service Greeter defines a Greeter service which contains method SayHello.
  4. HelloRequest and HelloReply represents the structure of request and reply for SayHello method.
  5. In Protobuff, if you have already noticed , HelloRequest and HelloReply are of message types.

A message is just an aggregate containing a set of typed fields. Many standard simple data types are available as field types, including bool, int32, float, double, and string. You can also add further structure to your messages by using other message types as field types.

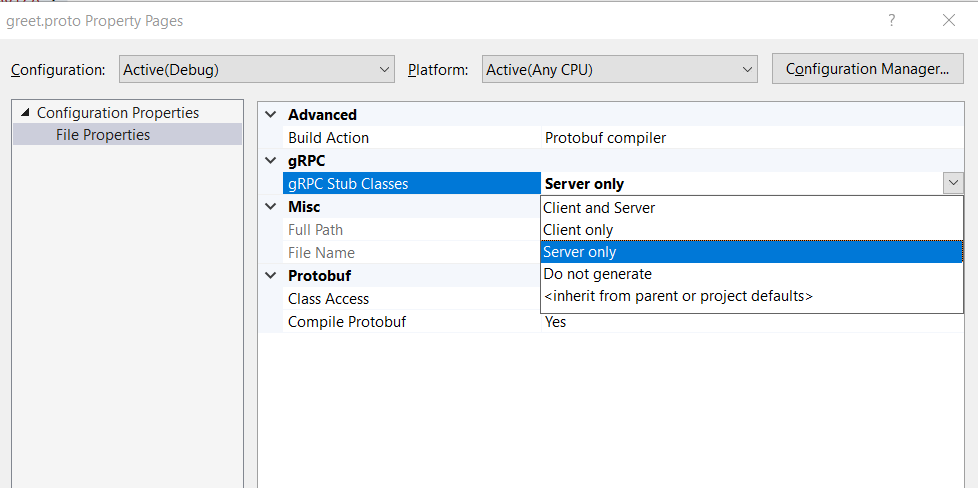
* 1. The " = 1", " = 2" markers on each element identify the unique "tag" that field uses in the binary encoding. Tag numbers 1-15 require one less byte to encode than higher numbers, so as an optimization you can decide to use those tags for the commonly used or repeated elements, leaving tags 16 and higher for less-commonly used optional elements. Each element in a repeated field requires re-encoding the tag number, so repeated fields are particularly good candidates for this optimization.

WCF SOAP: This proto file can be compared with the WSDL file in WCF SOAP from which you can generate client and server proxy codes using Add Service Reference or some ECF tools like WCF Red/Blue.

### Code generation using .proto file:

If you right click and go to the properties of the proto file, you will these four different options to generate Stub classes code.

To generate Service side code Server only needs to be selected , similarly for generating client side code Client Only needs to be selected.

****

Once saved after selecting the desired option, it gets saved the following way in the csProj file,

<ItemGroup>

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

</ItemGroup>

If you notice at the dependencies, you will see there is something called Grpc.Tools . This package contains the protoc compiler along with msbuild. Hence when we compile the project , Grpc.Tools generates the necessary C# asset based on the selection we made in the property.

The generated assets (files):

* Are generated on an as-needed basis each time the project is built.
* Aren't added to the project or checked into source control.
* Are a build artifact contained in the obj directory.

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">

### Generated c# Asset:

The tooling package generates the C# types representing the messages defined in the included \*.proto files.

For server-side assets, an abstract service base type is generated. The base type contains the definitions of all the gRPC calls contained in the .proto file. Create a concrete service implementation that derives from this base type and implements the logic for the gRPC calls. For the greet.proto, the example described previously, an abstract GreeterBase type that contains a virtual SayHello method is generated. A concrete implementation GreeterService overrides the method and implements the logic handling the gRPC call.

Here the concrete service implementation is present in GreeterService.cs file inside Services folder.

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

});

}

}

For client-side assets, a concrete client type is generated. The gRPC calls in the .proto file are translated into methods on the concrete type, which can be called. For the greet.proto, the example described previously, a concrete GreeterClient type is generated. Call GreeterClient.SayHelloAsync to initiate a gRPC call to the server.

static async Task Main(string[] args)

{

// The port number(5001) must match the port of the gRPC server.

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

var client = new Greeter.GreeterClient(channel);

var reply = await client.SayHelloAsync(

new HelloRequest { Name = "GreeterClient" });

Console.WriteLine("Greeting: " + reply.Message);

Console.WriteLine("Press any key to exit...");

Console.ReadKey();

}

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.

### 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.

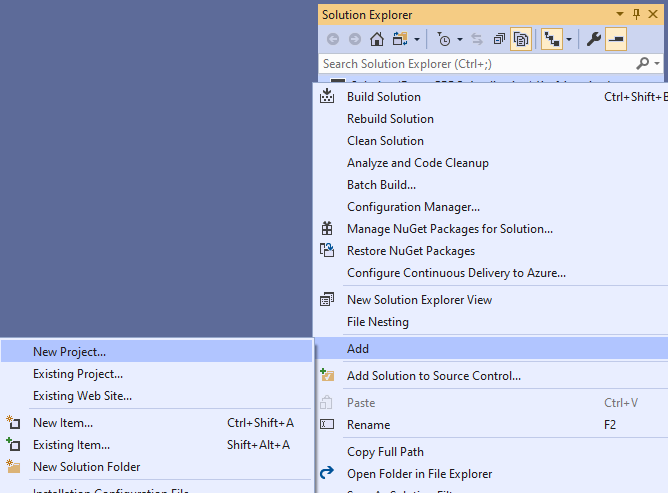
|  |
| --- |
| public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  {  if (env.IsDevelopment())  {  app.UseDeveloperExceptionPage();  }  app.UseRouting();             app.UseEndpoints(endpoints =>             {                 endpoints.MapGrpcService<GreeterService>();          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

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

## Creating a gRPC Client:

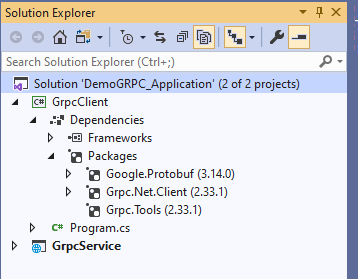
Now we will create the gRPC client application in same solution project folder.

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 |



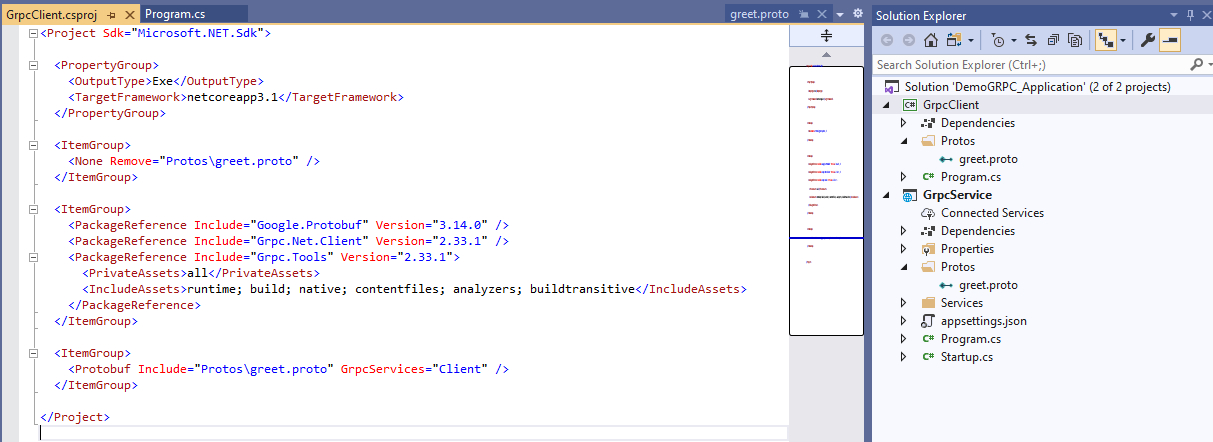
All require package are installed in our gRPC client application.

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 the paste 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. Or you can right click on the proto file , go to properties and select client only to generate client side stub.

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



1. Now we will call the gRPC service method in the gRPC client application. Following are areas worth noticing in the code:
   1. **Configure gRPC Client:** gRPC clients are concrete client types that are [generated from *\*.proto* files](https://docs.microsoft.com/en-us/aspnet/core/grpc/basics?view=aspnetcore-3.1#generated-c-assets). The concrete gRPC client has methods that translate to the gRPC service in the *\*.proto* file. For example, a service called Greeter generates a GreeterClient type with methods to call the service.

A gRPC client is created from a channel. Start by using GrpcChannel.ForAddress to create a channel, and then use the channel to create a gRPC client:

A channel represents a long-lived connection to a gRPC service. When a channel is created, it's configured with options related to calling a service. For example, the HttpClient used to make calls, the maximum send and receive message size, and logging can be specified on GrpcChannelOptions and used with GrpcChannel.ForAddress.

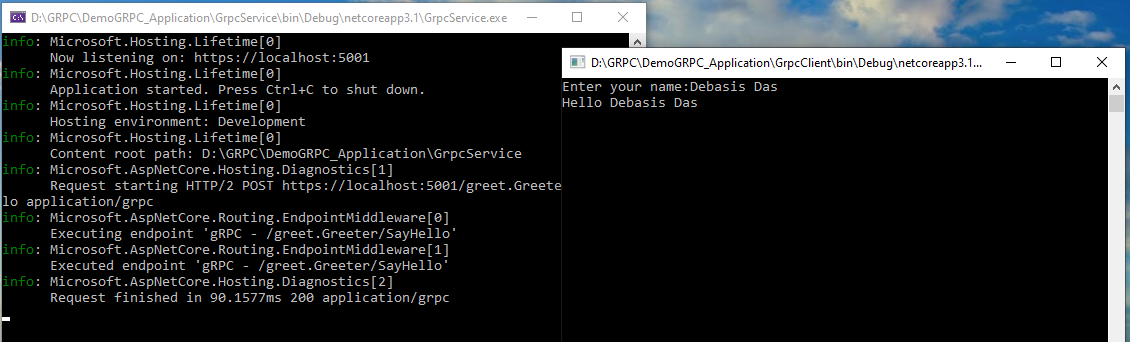
* 1. **Configure TLS :** A gRPC client must use the same connection-level security as the called service. gRPC client Transport Layer Security (TLS) is configured when the gRPC channel is created. A gRPC client throws an error when it calls a service and the connection-level security of the channel and service don't match.

To configure a gRPC channel to use TLS, ensure the server address starts with https. For example, GrpcChannel.ForAddress("https://localhost:5001") uses HTTPS protocol. The gRPC channel automatically negotiates a connection secured by TLS and uses a secure connection to make gRPC calls.

Following shows a unary gRPC call from client

|  |
| --- |
| 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:

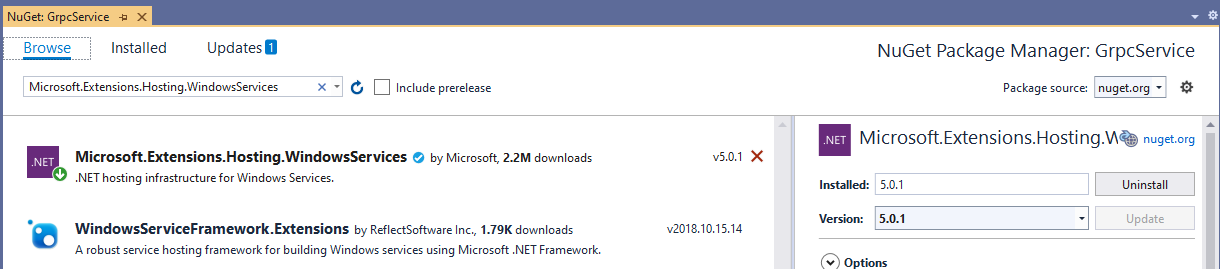


## **Hosting GRPC :**

Although ASP.NET Core 3.0 applications can be hosted in IIS on Windows Server, currently it isn't possible to host a gRPC application in IIS because some of the HTTP/2 functionality isn't supported. This functionality is a goal for a future update to Windows Server.

For now if you are developing a gRPC solution, You can run your application as a Windows service. Or you can run it as a Linux service controlled by [systemd](https://en.wikipedia.org/wiki/Systemd), because of new features in the .NET Core 3.0 hosting extensions.

First of all, to configure ASP.Net core GRPC to run as Windows Service, we need to add the “Microsoft.Extensions.Hosting.WindowsServices” package from NuGet package manager.

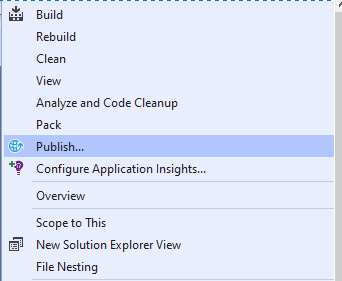


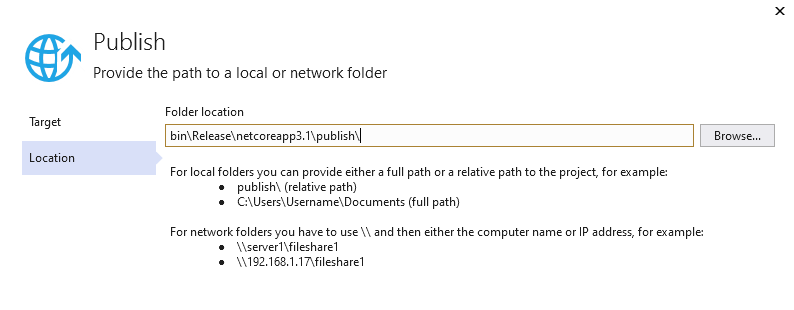
Then we need to add a call to UseWindowsService to the CreateHostBuilder method in Program.cs.

|  |
| --- |
| public static IHostBuilder CreateHostBuilder(string[] args) =>  Host.CreateDefaultBuilder(args)  .UseWindowsService() // Enable running as a Windows service  .ConfigureWebHostDefaults(webBuilder =>  {  webBuilder.UseStartup<Startup>();  }); |

Now publish the application by using one of these methods:

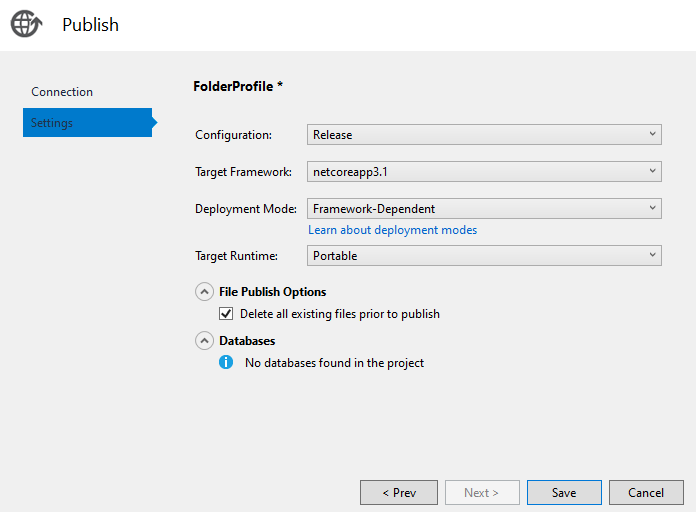
* From Visual Studio by right-clicking the project and selecting Publish on the shortcut menu.
* From the .NET Core CLI. (using dotnet publish command)





The Publish can be done in two modes:

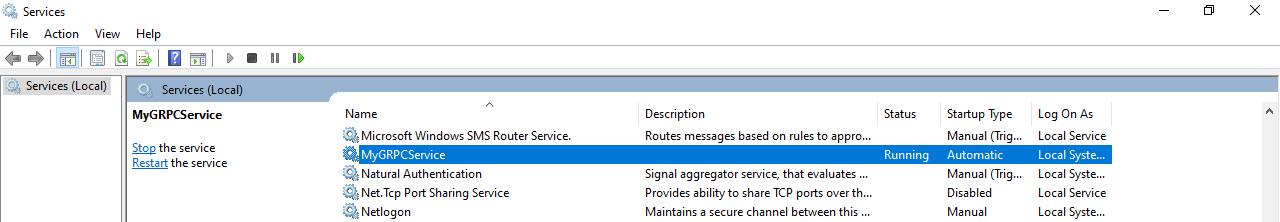
* Framework-Dependent deployment : Framework-dependent deployments require the .NET Core Shared Runtime to be installed on the host where they are run.
* Self-Contained deployment: Self-contained deployments are published with a complete copy of the .NET Core runtime and framework and can be run on any host



When publish is completed then we will need to host the generated exe file to window service from command Windows prompt and run the mention command in command prompt.

|  |
| --- |
| sc.exe create WindowsServiceName binpath= D:\DemoGrpcService\GrpcService\bin\Release\netcoreapp3.1\publish\GrpcService.exe start= auto |





## **Sample Project github refference**

* gRPC service project git url <https://github.com/ajaydas1184/GRPCSampleProject/tree/master/DemoGrpcService>
* gRPC Client project git url <https://github.com/ajaydas1184/GRPCSampleProject/tree/master/GrpcClientWebApplication>