

# How to build a Go gRPC API for beginners

## 1. Abstract

This use case description paper explains how to write a gRPC API in Go. The paper begins by providing an introduction to Go and gRPC, including their benefits and differences from traditional REST APIs. The paper then explains why Go is a suitable language for building gRPC APIs.

The main focus of the paper is a step-by-step tutorial on building a simple gRPC server in Go, including the use of proto files to define the data structures and operations for the API. The tutorial covers the entire process, from installing the required tools to writing the server-side and client-side implementation in Go.

Finally, the paper then provides a summary of the benefits of using a gRPC API and the advantages of using gRPC over REST APIs, but also contains a warning to think about whether a gRPC API is a suitable option. Then the UCDP finishes with a conclusion that this type of API is an accessible solution and alternative, even with no prior experience.

## 2. Introduction

Let's start with a short explanation about the topics at hand, the Go programming language also known as Golang and the gRPC model for APIs.

Go is a programming language designed with simplicity and efficiency in mind. Despite its simplicity Golang has a lot of potential and is a reliable and competent programming language.

gRPC is a high-performance, open-source RPC (Remote Procedure Call) framework. It is a powerful alternative to the well-known REST API. While it is a more complicated option, the gRPC framework has some clear upsides compared to other API models, like streaming or the use of protocol buffers.

Despite the possibilities, it can be quite daunting and confusing to start developing a gRPC API. This UCDP is made to that developing a gRPC API in Golang is possible for everyone, even those that have no experience with gRPC or Go.

## 3. The Go Language

The Go programming language is the base of the project, so let's begin with the analysis of Golang.

As a beginner going into writing Go blindly may give some trouble regarding go modules and package dependencies. However, Go has a set of commands to initialize a Go module and resolve package downloads and dependencies that are really easy to use. This makes it so that resolving this trouble is only a few google searches away and only makes it a minor issue.

Programming in Go is surprisingly intuitive in almost every way. An important part of Golang is handling pointers, but this is not a problem. Pointers are decently hard to master, but it is really easy to learn it to a usable and functional level in Go. Golang makes the learning process really easy, even with no experience in handling pointers in any programming language. This was my experience for most of the syntax in Go. Take the unique use of "!=" in Go for example. It allows for a smooth creation and of a variable and assigning a value to it with just 2 characters. Another great benefit of Golang is that functions can return multiple values, which can simplify development a lot. In case there is no use for a returned value it can just be discarded it by assigning it to "\_". For loops also make use of this technique. The basic syntax for a for loop is :

```
for index, element := range someSlice {}
```

If there is a use for an index you can just assign it, if not just assign it as "\_". Any for loop will not stray far from the basic format, because it is so easily to adapt, which makes it very clear and understandable. All of these examples are a testament to how simple Golang is.

There are some downsides to simplicity too, however. Take a while loop for example. There is no while keyword in Golang, a while loop in Go is for with a condition. This obviously makes sense, but the words that are used can make it confusing. Another case for this is deleting an item from an array. There is no built in method for this. The easiest way to remove an item at a specific index is to make a new array by putting a sub array from every element before the index and a sub array with every element after the index. This is a workaround in lack for a built in method that can still be done with 1 line of code. This proves that simplicity does not make every situation easier, although that is the case most of the time.

#### 4. The gRPC API framework

A gRPC API is an API that makes use of a proto file so it is necessary to learn how to use it, but what is a proto file?

A proto file is a file that uses the Protocol Buffer language to define data structures and the operations that can be performed on them. Proto files are used in the gRPC framework to define the structure of the data that is exchanged between client and server and the operations that can be performed on that data. A proto file typically consists of a series of message definitions, which define the data structures that will be used in the API, and a series of service definitions, which define the methods that can be called on the server.

To use this proto file, you would first need to compile it into code for your desired language using the Protocol Buffer compiler (protoc). This would generate the necessary code for serializing and deserializing the data structures defined in the proto file and for implementing the service defined in the file. However, it does more than just generate the defined structures and methods, it also generates types and methods necessary for defining client and server variables, which are vital to make a working gRPC API, or use one.

Then you can start working on the actual gRPC API. Setting this up is really easy since with the methods the proto file provides and the gRPC packages that Golang can provide. Just make sure to write methods in the API for each of the methods defined in the proto file and the server can be started.

#### 5. The benefits of gRPC

So what is the reason for learning gRPC, what is the use case of it? There are several reasons gRPC may be beneficial to use.

First of all is the performance of the model. gRPC is designed to be fast and efficient, with a smaller footprint and lower overhead compared to REST. It uses HTTP/2 as the underlying transport protocol, which allows it to take advantage of features such as multiplexing and streaming.

The second benefit is the strong typing. It means gRPC uses Protocol Buffers for serialization, which provides strong typing and backwards compatibility. This can make it easier to evolve and maintain a gRPC API over time.

Next is bi-directional streaming. gRPC supports bi-directional streaming, which allows both the client and server to send a stream of messages to each other. This can be useful for scenarios where a large amount of data needs to be transferred or where the communication is interactive.

Last is the efficient resource utilization that gRPC has. It has support for streaming and connection reuse can lead to more efficient resource utilization on both the client and server side.

Note that every type of API has their own set of benefits. Such as REST being widely adopted and familiar to many developers, having a large number of available tools and libraries, and being well-suited for certain types of APIs. It is important to evaluate the specific needs and requirements of your API and choose the architecture that is the best fit for your use case. gRPC is a very powerful type of API but it certainly does not fit in every kind of environment. It is best used in big project that need a quick and high performance API, that can handle a lot of data, and can make use of the asynchronous streaming capabilities of gRPC.

## 6. The compatibility between Go and gRPC

Go is an excellent language to write a gRPC API with for several reasons.

Let's begin with the most important reason, Go has excellent support for gRPC. Golang has native support for gRPC and provides a number of libraries and tools to make it easy to build gRPC APIs. This is the main reason on why Go is a good choice, but it is not the only one.

Go's built-in concurrency features, such as goroutines and channels, make it easy to write concurrent code. This can be particularly useful when building a gRPC API and can multiply the functionality and ease the use of the streaming capabilities of gRPC.

Another reason why Go is a good choice is because it's comprehensive standard library provides a wide range of functionality out of the box, including support for networking, serialization, and cryptography. This can make it easier to build a gRPC API without needing to rely on external libraries.

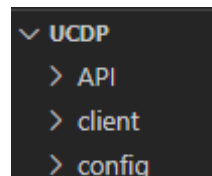
Golang compiles quickly, which can be beneficial when working on a large project with many dependencies, which is often the case since gRPC is made to be an efficient API for larger projects.

Go also has a large and active community of users, with many resources and libraries available online. This can make it easier to get help and support when working with Go and gRPC.

## 7. Building the API

### I. Setup

For the sake of accessibility and clearness we will set up a simple structure. Make an empty directory for the gRPC API, this will be referred to as the project map. Proceed to make 3 new directories in the project map, one for the API, one for configuration files and one for a client to test the API.



Figuur 1 General Structure

There are of course several things to install beforehand.

Since we will be using Golang as our programming language, make sure it is installed and usable. This UCDP was made with Go version *go1.19.3 windows/amd64* and visual studio code.

Additionally Protocol Buffer Compiler (protoc) is required to generate necessary configuration files. The version used is *libprotoc 3.20.2* and all versions can be found at <https://github.com/protocolbuffers/protobuf/releases>.

Go projects need to be initialized. To do so, use the following command in a terminal opened at the project map.

```
go mod init ucdp.com/grpc
```

Go files often have dependencies and make use of external libraries. To resolve and download these dependencies all you have to do is add them in the import section of files and then use the following command on the project map.

```
go mod tidy
```

### II. Proto

Create a new proto file in the configuration directory by creating a new file with a .proto extension, such as *myapi.proto*.

Define the data structures that will be used in the API using message definitions in the proto file.

```
syntax = "proto3"; option go_package = "ucdp.com/grpc/config"; message Person { string name = 1; int32 age = 2; }
```

Define the operations (or methods) that can be performed on the data using service definitions in the proto file.

```
service GreetingService { rpc SayHello (Person) returns (Person) {} }
```

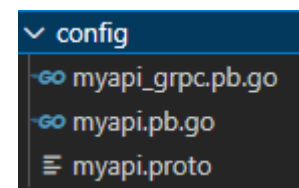
```
config > ≡ myapi.proto > ...
1  syntax = "proto3";
2
3  option go_package = "ucdp.com/grpc/config";
4
5  message Person { string name = 1; int32 age = 2; }
6  service GreetingService { rpc SayHello (Person) returns (Person) {} }
7
```

Figuur 2 Proto File

Compile the proto file using the protoc compiler by using the following command in the configuration map.

```
protoc --go_out=. --go_opt=paths=source_relative --go-grpc_out=. --go-grpc_opt=paths=source_relative
myapi.proto
```

This will result in several files being created based on what was defined in the proto file. These files contain all basic data structures and methods fully worked out. Furthermore, these created files have several data structures and methods that are necessary to create the server and client.



Figuur 3 Proto Directory

### III. API

Write the server-side implementation of the gRPC API in Go. This will include defining the service handler functions that will be called when the service methods are invoked.

```
package main
import ("context" "log" "net" "google.golang.org/grpc" pb "ucdp.com/grpc/config")
type greetingServiceServer struct {pb.UnimplementedGreetingServiceServer}
func (s *greetingServiceServer) SayHello(ctx context.Context, in *pb.Person) (*pb.Person, error) {
    log.Printf(in.Name)
    return &pb.Person{Name: "Hello, " + in.Name}, nil
}
func main() {
    lis, err := net.Listen("tcp", ":50051")
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterGreetingServiceServer(s, &greetingServiceServer{})
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

```
API > ∞ server.go > ...
1  package main
2
3  import (
4      "context"
5      "log"
6      "net"
7
8      "google.golang.org/grpc"
9      pb "ucdp.com/grpc/config"
10 )
11
12 type greetingServiceServer struct {
13     pb.UnimplementedGreetingServiceServer
14 }
15
16 func (s *greetingServiceServer) SayHello(ctx context.Context, in *pb.Person) (*pb.Person, error) {
17     log.Printf(in.Name)
18     return &pb.Person{Name: "Hello, " + in.Name}, nil
19 }
20
21 func main() {
22     lis, err := net.Listen("tcp", ":50051")
23     if err != nil {
24         log.Fatalf("failed to listen: %v", err)
25     }
26     s := grpc.NewServer()
27     pb.RegisterGreetingServiceServer(s, &greetingServiceServer{})
28     if err := s.Serve(lis); err != nil {
29         log.Fatalf("failed to serve: %v", err)
30     }
31 }
32
```

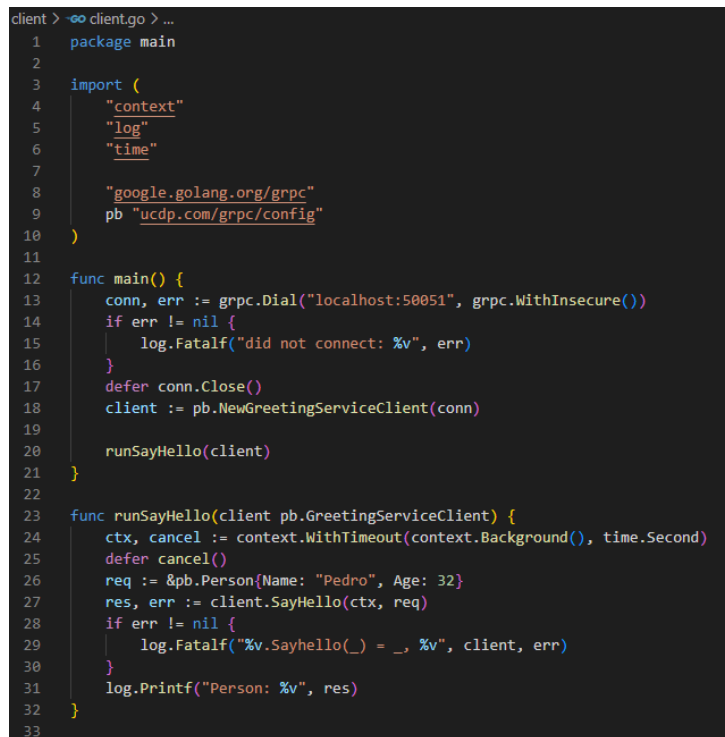
Figuur 4 API File

In this case the SayHello function will just print the name of the person datatype it got as input and send it back.

## IV. Client

Write the client-side implementation of the gRPC API in Go. This will include connecting to the server and invoking the service methods.

```
package main import ("context""log""time""google.golang.org/grpc"pb "ucdp.com/grpc/config")func main()
{conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure())if err != nil {log.Fatalf("did not connect: %v",
err)}defer conn.Close()client := pb.NewGreetingServiceClient(conn)runSayHello(client)}func runSayHello(client
pb.GreetingServiceClient) {ctx, cancel := context.WithTimeout(context.Background(), time.Second)defer
cancel()req := &pb.Person{Name: "Pedro", Age: 32}res, err := client.SayHello(ctx, req)if err != nil
{log.Fatalf("%v.Sayhello(_) = _, %v", client, err)}log.Printf("Person: %v", res)}
```



```
client > client.go > ...
1 package main
2
3 import (
4     "context"
5     "log"
6     "time"
7
8     "google.golang.org/grpc"
9     pb "ucdp.com/grpc/config"
10 )
11
12 func main() {
13     conn, err := grpc.Dial("localhost:50051", grpc.WithInsecure())
14     if err != nil {
15         log.Fatalf("did not connect: %v", err)
16     }
17     defer conn.Close()
18     client := pb.NewGreetingServiceClient(conn)
19
20     runSayHello(client)
21 }
22
23 func runSayHello(client pb.GreetingServiceClient) {
24     ctx, cancel := context.WithTimeout(context.Background(), time.Second)
25     defer cancel()
26     req := &pb.Person{Name: "Pedro", Age: 32}
27     res, err := client.SayHello(ctx, req)
28     if err != nil {
29         log.Fatalf("%v.Sayhello(_) = _, %v", client, err)
30     }
31     log.Printf("Person: %v", res)
32 }
33
```

Figuur 5 Client File

As shown in figure 5, it only takes a few lines of code to connect with the API. When running the client the “main” function will be executed, thus that is where we call another function, runSayHello, that calls the SayHello function of the API.

## 8. Summary


Both Golang and the gRPC model have a lot of potential and can go into extreme detail which makes them very hard subjects to master. There is a difference in the learning difficulty between the two subject. Golang is very simplistic and easy to understand whereas gRPC has a steeper learning curve. gRPC is by no means rocket science, but it is a more complicated model for APIs especially when compared to REST. The difference in difficulty is not exclusive to building the API but also using it. Where a REST API is very easy to use in any circumstance, there should be some thoughts about whether the gRPC capabilities are necessary and how much effort it takes for the application/frontend to use the gRPC API.

## 9. Conclusion

It is absolutely possible to develop a good gRPC API in Go despite having no experience with gRPC or Go. There will always be room for improvement, especially if you begin from nothing. This does not take away that anyone can make a gRPC API in Go if there is demand for one, if some effort is put into learning how to make one.

## References

CodeOpinion. (2022, 23 juni). Where should you use gRPC? And where NOT to use it! YouTube. <https://www.youtube.com/watch?v=4SuFtQV8RCk>

School, T. (2021, 26 juni). Is gRPC better than REST? Where to use it? DEV Community . <https://dev.to/techschoolguru/is-grpc-better-than-rest-where-to-use-it-3blg>

Laith Academy. (2022, 24 februari). Build a Rest API with GoLang. YouTube. [https://www.youtube.com/watch?v=d\\_L64KT3SFM](https://www.youtube.com/watch?v=d_L64KT3SFM)

Forbes, E. (2020, 28 april). Go gRPC Beginners Tutorial. TutorialEdge. <https://tutorialedge.net/golang/go-grpc-beginners-tutorial/>

One Minute Notes. (2022, 23 april). #10 Go Tutorial | Build A CRUD API with gRPC. YouTube. <https://www.youtube.com/watch?v=4gsDeKVfUUU>

TutorialEdge. (2020, 2 mei). Beginners Guide to gRPC in Go! YouTube. [https://www.youtube.com/watch?v=BdzYdN\\_Zd9Q](https://www.youtube.com/watch?v=BdzYdN_Zd9Q)

Mulders, M. (2020, 30 november). Creating a web server with Golang. LogRocket Blog. <https://blog.logrocket.com/creating-a-web-server-with-golang/>

Hitesh Choudhary. (2021, 3 oktober). Creating server for golang frontend. YouTube. <https://www.youtube.com/watch?v=xh79JXJy0yY>

grpc package - google.golang.org/grpc - Go Packages. (z.d.). <https://pkg.go.dev/google.golang.org/grpc>

The Go Programming Language. (z.d.). <https://go.dev>

Quick start. (2022, 7 april). gRPC. <https://grpc.io/docs/languages/go/quickstart/>

Overview | Protocol Buffers |. (z.d.). Google Developers. <https://developers.google.com/protocol-buffers/docs/overview>

Bamimore, O. (2021, 2 december). Concurrency patterns in Golang: WaitGroups and Goroutines. LogRocket Blog. <https://blog.logrocket.com/concurrency-patterns-golang-waitgroups-goroutines/>

Gillis, A. S. (2020, 11 mei). Go (programming language). IT Operations. <https://www.techtarget.com/searchitoperations/definition/Go-programming-language>