salesforce (https://www.salesforce.com/) Dreamforce On-Demand →

(https://www.mulesoft.com/)

Home > Resources > Articles > API U Series >

Understanding the Essentials of gRPC

Understanding the Essentials of gRPC

In the **previous installment (/api-university/what-grpc-api-and-how-does-it-work)** of this series, we described the basics of GRPC in terms of history and overview. We talked about how gRPC emerged from Remote Procedure call technology (RPC) and evolved into a technology that is becoming a fixture on the landscape of distributed computing. Also, we provided a high-level overview of how the technology works.

In this installment, we're going to take a more detailed look at gRPC. We're going to briefly review the basics we discussed prior. We're going to cover the essentials of gRPC architecture in terms of specification and implementation. Then, we're going to take a detailed look at the underlying concepts and practices required to create a gRPC API.

We'll talk about the various approaches for exchanging information under gRPC, synchronous request/response as well as unidirectional and bidirectional streaming. We're going to spend some time examining the Protocol Buffers specification, particularly encoding and decoding. (Protocol Buffers is the binary format by which data is passed between gRPC endpoints.) Finally, we're going to study gRPC messages and procedures as well as the various approaches to creating actual code for a gRPC API.

Let's start with key concepts.

Key Concepts

gRPC is a client-server API technology that allows information to be exchanged in three ways. The first way is in a typical, synchronous request/response interaction. The second way is

IN THIS SERIES

Guide to gRPC; Understanding, Building, and Using gRPC APIs (/api-university/guide-to-grpc-understanding-building-and-using-grpc-apis)

Part 1: What is a gRPC API and How Does it Work? (/api-university/what-grpc-api-and-how-does-it-work)

Part 2: Understanding the Essentials of gRPC (/api-university/understanding-essentials-grpc)

Part 3: How to Auto-Generate gRPC Code Using protoc (/api-university/how-to-auto-generate-grpc-code-using-protoc)

Part 4: How to Make Use of gRPC Libraries and Frameworks (/api-university/how-to-make-use-grpc-libraries-and-frameworks)

is sending a request to the server and having data returned in a stream. This is called unidirectional streaming. The third way is to send data in a stream from the client to the server and have the server return data in a stream back to the client. This is called bidirectional streaming.

HTTP/2 (https://en.wikipedia.org/wiki/HTTP/2) is the specified protocol for data exchange under gRPC. Also, data is exchanged between a client and a server in a binary format known as Protocol Buffers. While the gRPC specification does not forbid using other data formats such as **Thrift (https://en.wikipedia.org/wiki/Apache_Thrift)**, **JSON (https://en.wikipedia.org/wiki/JSON)** or **SOAP (https://en.wikipedia.org/wiki/SOAP)** for encoding data to be shared between client and server, most, if not all, implementations of gRPC use Protocol Buffers. It's become the de-facto standard.

Protocol Buffers is not a self-describing data format. Unlike JSON or XML where one can decipher the field and value in a data structure by just examining data structure itself, Protocol Buffers requires a "reference manual" common to both client and server. This reference manual is the mechanism used by serializers to encode and decode the data that's sent between client and server. This "reference manual" is the `.proto` file. (See Figure 1, below)

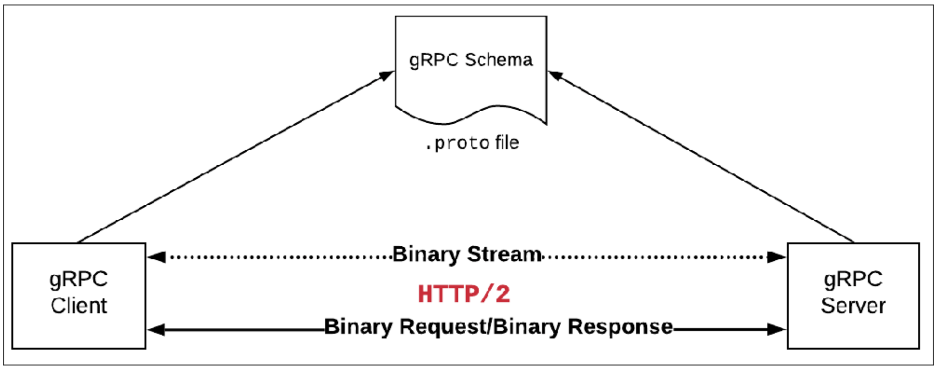


Figure 1: The fundamental structure of a gRPC client-server interaction

The `.proto` file is the authoritative source that describes the procedures that are published by a particular gRPC API. Also, the `.proto` file describes the data structures that a given procedure consumes and emits. The term for a data structure under gRPC is a message. A `.proto` file is a text file, the format of which conforms to the Protocol Buffers language

Part 5: gRPC in the Real World: The Kubernetes Container Runtime Interface (/api-university/grpc-real-world-kubernetes-container-runtime-interface)

Part 6: How Gitlab puts gRPC in the Real World (/api-university/how-gitlab-puts-grpc-real-world)

Part 7: How Ably.io Uses gRPC APIs to Streamline Its Messaging Service (/api-university/how-ablyio-uses-grpc-apis-to-streamline-its-messaging-service)

Part 8: How to Build a Streaming API Using gRPC (/api-university/how-to-build-streaming-api-using-grpc) specification published by Google. You can read the **specification (https://developers.google.com/protocol-buffers/docs/proto3)** here.

Creating the `.proto` File

The first thing you're going to do when creating a gRPC API is to create the `.proto` file. As mentioned above, the `.proto` file will describe the procedures (a.k.a functions) published by the API as well as the messages the procedures will use.

Once the `.proto` file is created it will be used by your programming language of choice to encode and decode data for transmission between client and server into the Protocol Buffers binary format. (See Figure 2, below)

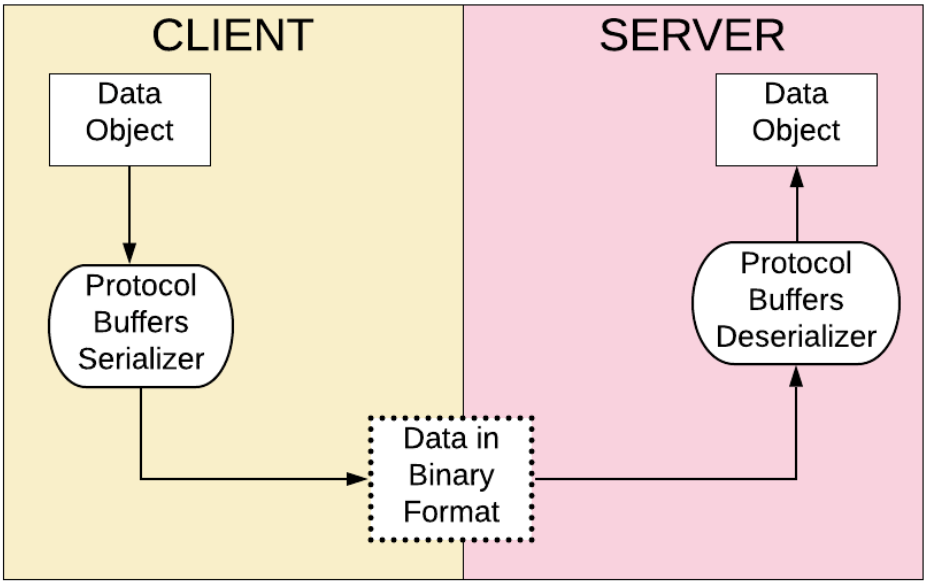
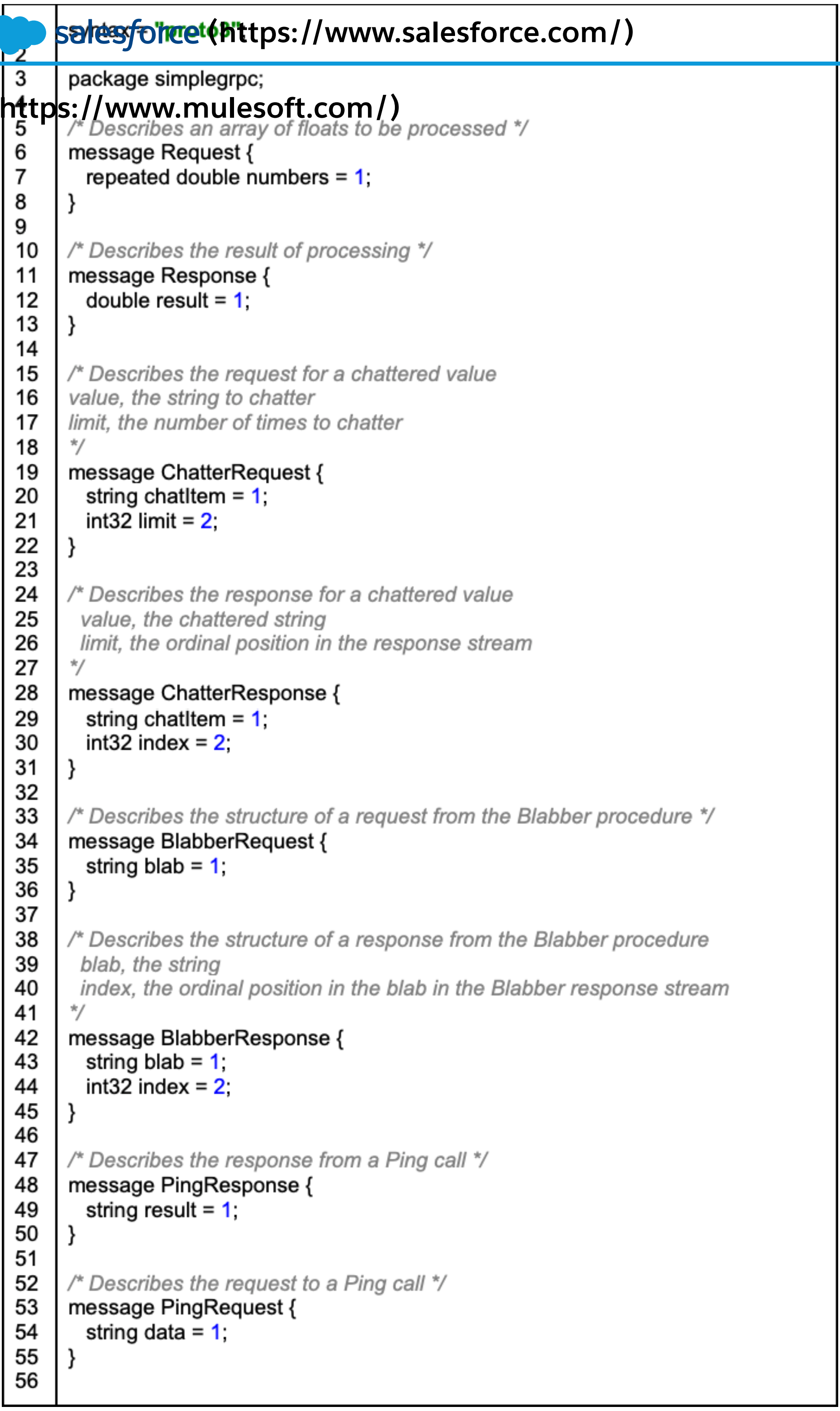


Figure 2: The de-facto standard for exchanging data between client and server under gRPC is Protocol Buffers

Also, the `.proto` file can be used as the source for auto-generating boilerplate code in the programming language of choice. (We'll cover auto-generating code later in this article.) Listing 1 below shows the complete `.proto` file for demonstration gRPC API **SimpleService (https://github.com/programmableweb/simple-node-grpc)** that accompanies this article. Notice that the `.proto` file defines a package named `simplegrpc` at Line 3. Messages are defined from Lines 6 to 55, and procedures are defined at Lines 58 to 66.



```
syntax = "proto3";

package simplegrpc;

/* Describes an array of floats to be processed */
message Request {
  repeated double numbers = 1;
}

/* Describes the result of processing */
message Response {
  double result = 1;
}

/* Describes the request for a chattered value
value, the string to chatter
limit, the number of times to chatter
*/
message ChatterRequest {
  string chatItem = 1;
  int32 limit = 2;
}

/* Describes the response for a chattered value
  value, the chattered string
  limit, the ordinal position in the response stream
*/
message ChatterResponse {
  string chatItem = 1;
  int32 index = 2;
}

/* Describes the structure of a request from the Blabber procedure */
message BlabberRequest {
  string blab = 1;
}

/* Describes the structure of a response from the Blabber procedure
  blab, the string
  index, the ordinal position in the blab in the Blabber response stream
*/
message BlabberResponse {
  string blab = 1;
  int32 index = 2;
}

/* Describes the response from a Ping call */
message PingResponse {
  string result = 1;
}

/* Describes the request to a Ping call */
message PingRequest {
  string data = 1;
}
```

Listing 1: The `.proto` file for the SimpleService demonstration API

We're going to cover the messages, procedures, and namespaces in the sections that follow. The particulars of implementing the SimpleService API based on the `.proto` file shown above in Listing 1 are covered in the corresponding articles to this installment that describe creating gRPC APIs using code auto-generation and **programming with libraries and frameworks (/api-university/how-to-make-use-grpc-libraries-and-frameworks)**.

Defining Messages

A message represents a custom gRPC data type described in Protocol Buffers format. A message is composed of fields. A field's type can be one of the Protocol Buffers scalar types. Also, a field's type can be another message. Table 1 below lists the scalar types defined by the Protocol Buffers specification. A description of complex types is provided in the section, Understanding the Protocol Buffers Message Format, that follows.

Type	Example	Comment
double	2.5	double has 15 decimal digits of precision
float	11.23	double has 7 decimal digits of precision
int32	-4	Range: -2,147,483,648 to 2,147,483,647
int64	4	Range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint32	564	0 to 4,294,967,295; Uses variable-length encoding.
uint64	3	0 to 18,446,744,073,709,551,615; Uses variable-length encoding.
sint32	-99	Signed int value. These more efficiently encode negative numbers in the Protocol Buffer binary format than regular int32

Type	Example	Comment
sint64	99	Signed int value. These more efficiently encode negative numbers in the Protocol Buffer binary format than regular int64
fixed32	45	Always four bytes. More efficient than uint32 if values are often greater than 228
fixed64	45	Always eight bytes. More efficient than uint64 if values are often greater than 256
sfixed32	-76	Always four bytes
sfixed64	76	Always eight bytes
bool	TRUE	True or False, 0 or 1
string	"Hello World"	
bytes	<Buffer 0a 0d 49 20>	

Table 1: The scalar values supported by the Protocol Buffers specifications

Understanding the Protocol Buffers Message Format

A message is constructed according to the structure defined by the Protocol Buffers format. Each message is assigned a unique name. Each field in a message is named too. A field is defined with a data type and an index number. Index numbers are assigned in sequence. Using an index number is critical to the Protocol Buffers message format because when a message is encoded into binary format, fields are identified according to their index numbers. When it comes time to decode the message, deserialization logic uses the index number as the key by which to decipher the data structure into human-readable text. This again is where you see the emphasis on performance in gRPC's architecture. Referencing fields by index vs. fieldnames will always yield a performance benefit. So too will the field ordering that's inherent to such indexing when compared to a format where fields can be in any order (and fields are identified by field name). The designers of gRPC clearly looked for ways to reduce the machine time spent on serialization and deserialization. (We cover field identification later on in this article when we discuss binary encoding according to the Protocol Buffers specification.)

LIsting 2 below shows an example of two Protocol Buffers messages, `Address` and `Person`.

```
 message Address {
   string address = 1
   string city = 2
   string state_province = 3
   string country = 4
   string postal_code = 5
}
message Person {
   string first_name = 1;
   string last_name = 2
   int32 age = 3;
   Address address = 4
}
```

Listing 2: An example of a Protocol Buffers messages

Notice in Listing 2, above, that the message named Address contains five fields, all of which are scalar type string. The message named Person contains four fields, of which three are scalar types (`first_name`, `last_name`, and `age`). The type of the field named address is the message type Address. Creating messages that contain scalar and complex types is quite common. Using a message as the data type promotes code reuse.

The Protocol Buffers specification also supports nested messages. Listing 3 below an example of nesting one message within another.

```
  message Person {
    message Address {
      string address = 1
      string city = 2
      string state_province = 3
      string country = 4
      string postal_code = 5
    }
    string first_name = 1;
    string last_name = 2
    int32 age = 3;
    Repeated Address addresses = 4
  }
```

Listing 3: An example of a nested Protocol Buffers message

Notice that in Listing 3, the `Address` message is nested within the `Person` message. Also, notice that `Address` is the type assigned to the field addresses of the `Person` message. The reserved word `Repeated` is used as part of the declaration for the field `addresses`. `Repeated` indicates that the field is an array of a given type, in this case, `Address`. Fields that are marked `Repeated` return all the values in an array at once, as part of the message. However, if the array was marked with the reserved word `Stream`, the values in the array will be returned back one at a time in a stream. (We'll discuss streams later in this article.)

Understanding Protocol Buffers Serialization

As mentioned above, all data moving between client and server is encoded into and from the Protocol Buffers binary format. (You can read the details of the Protocol Buffers language guide **here (https://developers.google.com/protocol-buffers/docs/proto3)**.) Protocol Buffers encoding and decoding (a.k.a. serialization and deserialization) use the schema defined in a particular `.proto` file to convert a text message into and from binary data.

The critical concept to understand is that Protocol Buffer binary encoding and decoding depends on having access to the schema defined in a the given `.proto` file. As you'll read later in this article, the code that encodes and decodes the binary data exchanged in a gRPC interaction is usually created using a special tool named **protoc (https://grpc.io/docs/protoc-installation/)**. `protoc` can automatically generate client and server code that has the encoding and decoding logic embedded. `protoc` can create code that has a logical model of the gRPC schema embedded.

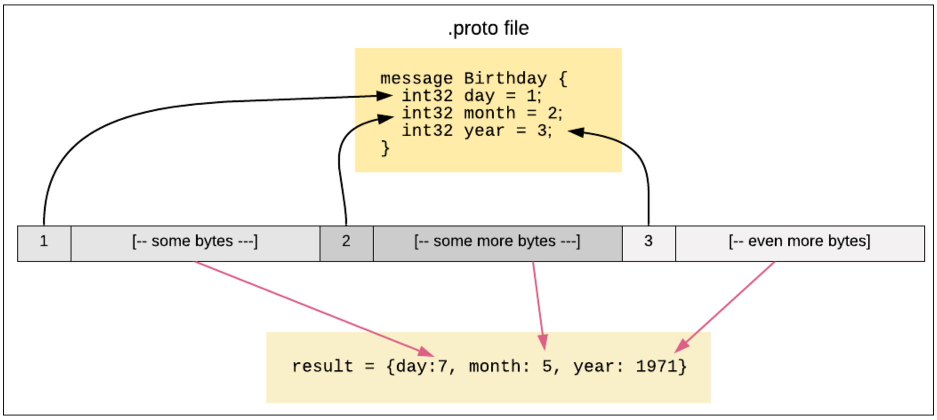
This logical model eliminates the need to have the actual `.proto` file available in order to encode and decode a binary message during runtime. This is called auto generating static code.

Also, `protoc` can auto generate client and server code that loads the `.proto` file into memory at runtime and then uses the in-memory schema to encode and decode a binary message. This approach is gRPC's dynamic code generation approach. Although *ProgrammableWeb* could not find evidence of such a dynamic gRPC implementation in the real world, this approach could be useful in situations where the gRPC API's technical contract is expected to change on a regular basis. It is somewhat akin to the flexibility afforded to REST APIs through hypermedia, though very few REST implementations actually leverage that capability. This dynamism keeps the consuming-client and providing-server in sync with one another when it comes to their mutual understanding of that contract.

However, this dynamic approach also means that one or many .`proto` files with identical content need to be physically accessible to both client and server at runtime which could be a challenge. For example, if the client and server are topologically (in network terms) or geographically separated from one another, it would probably work against gRPC's high performance nature for the client to interrogate a centrally located `.proto` file from across a network at runtime. One fix for this challenge would be to ensure that both the client and server have local access to their own identical copies of the `.proto` file at runtime. But doing so would require a timely and reliable synchronization/replication process that, as best as we can tell, is not provided through gRPC itself. As a result, such a dynamic approach is probably best suited to backend service interactions within the same physical system where a single `.proto` file is local to both the consuming client and providing-server.

Also, keep in mind that dynamic auto generation only creates working code for encoding and decoding binary messages. If the .`proto` file adds a new procedure to the gRPC schema, the business logic for that new procedure still needs to be written by a developer. Again, as mentioned above, having a software development process that is synchronized with changes in the `.proto` file is critical.

Let's take a look at an example of how a .proto file corresponds to binary encoding. Figure 3 below shows a portion of a .proto that defines a Protocol Buffers message Birthday. Birthday has three data fields: `day`, `month` and `year`. Each field is of type `int32`. Notice that the field day is given the index 1, the field month has the index 2 and the field year has the index 3. The encoding process is such that serialization logic will look at an instance of a message and convert each field into a binary chunk. The field's binary chunk will have its index number, its data type and its value laid out as an array of bytes in sequence. What it will NOT have is the name of the field. As mentioned earlier, fields are identified by index number.



```
message Birthday {
  int32 day = 1;
  int32 month = 2;
  int32 year = 3;
}
```

```
result = {day:7, month: 5, year: 1971}
```

Figure 3: Protocol Buffers binary format specification encodes and decodes into binary data according to the message definition in the `.proto` file

When it comes time to deserialize the binary message into a text format, the deserialization logic gets the index number of the binary byte array and looks up the field name according to the index number as defined in the message definition within the `.proto` file. Then the logic extracts the field name from the `.proto` accordingly and associates the field name with the corresponding data as shown in Figure 3, above. All that's left to do is to create a textual representation of the field name and the field value. The text representation can be in any format the serializer/deserializer supports such as XML, YAML or JSON, like so:

```
 {
  day: 7
  month: 5,
  year: 1971
}
```

Be advised that the work of serializing and deserializing data into Protocol Buffers binary involves low level programming with bits and bytes. Such programming requires a certain skills set and a certain way of thinking. And, it's laborious. While being able to do Protocol Buffers encoding might be an intriguing challenge for some, it's work that's more easily done using existing libraries. All of the popular language frameworks for gRPC have serializer/deserializer libraries built in. Thus, there's no need to reinvent the wheel.

However, there are real-world situations in which it's necessary for a programmer to go up against a Protocol Buffer binary directly. Remember, one of the key benefits of gRPC is that it's very fast and efficient when it comes to the use of machine resources. This makes gRPC very attractive to companies that have an extraordinary need for speed, a desire for smaller data center footprints at serious scale (eg: Web scale), or both.

As a result, there are situations in which taking the time to deserialize an entire Protocol Buffers binary into text is unacceptable. Rather than deserialize the entire binary, it's faster just to go into the binary and extract exactly the field of interest according to index number. Again, this is a performance enhancing option that's simply not available in other API architectures like REST.

Accessing the binary with the degree of granularity requires custom code. You need to know the index number of the field you're looking for as well as the data type of that field. And then you need to do parsing at the bit and byte level. It's exacting work. But, given the special nature of the situation writing the custom deserialization can be well worth it. It's a matter of cost versus benefit.

Defining Procedures

Central to gRPC is the idea that clients call procedures (a.k.a functions) directly according to the procedure definition described in the `.proto` file. gRPC specifies that procedures are grouped under an organizational unit called a service. Listing 4 below describes a service named `SimpleService`. Listing 4 is an excerpt from the `.proto` file listed above in Listing 1.

```
service SimpleService {
    rpc Add (Request) returns (Response) { }
    rpc Subtract (Request) returns (Response) { }
    rpc Multiply (Request) returns (Response) {}
    rpc Divide (Request) returns (Response) { }
    rpc Chatter (ChatterRequest) returns (stream Chat
    rpc Blabber (stream BlabberRequest) returns (stre
    rpc Ping (PingRequest) returns (PingResponse) { }
}
```

Listing 4: The methods that make up the service, `SimpleService`

A gRPC service is defined in the `.proto` file using the reserved word `service` followed by the name of the service. Then, the service's methods are enclosed in a pair of curly brackets. Each method in the service is preceded by the reserved word rpc followed by the method name and input message type. The input message type is followed by the reserved word `returns` followed by the message type that is to be returned by the procedure. If the input message type is to be submitted in a stream, then the reserved work `stream` precedes the input message type declaration. The same is true if the method is to return values in a stream.

Let's take a look at a concrete example. The code below shows the declaration for the `SimpleService` method `Blabber` taken from Listing 4 above.

```
  rpc Blabber (stream BlabberRequest) returns (strea
```

where `rpc` is a gRPC a reserved word indicating a gRPC method

`Blabber` is the programmer defined method name

`(stream BlabberRequest)` indicates that instances of a `BlabberRequest` message type will be submitted in a stream to the server. (The `BlabberRequest` message is defined in Listing 1 shown at the beginning of this article.)

`returns` is a gRPC a reserved word indicating that return type is to follow

`(stream BlabberResponse)` indicates that instances of a `BlabberResponse` message type will be returned in a stream from the server. (The `BlabberResponse` message is also defined in Listing 1 shown at the beginning of this article.)

Be advised that the example above illustrates a method that submits data to the server in a stream and then likewise returns data from the server in a stream. However data can be submitted as a simple request with an accompanying response as shown in the following code for the Ping method which is taken from Listing 4 above.

```
  rpc Ping (PingRequest) returns (PingResponse) { }
```

In this case, the `Ping` method sends an instance of the message type `PingRequest` to the gRPC server and the server returns an instance of the message type `PingResponse`.

Avoiding confusion about the custom message names Request and Response

The SimpleService gRPC API shown above in Listing 1 and in Listing 5 below defines messages named `Request` and `Response`. These messages are custom to `SimpleService`. However, even though these message names represent custom messages in `SimpleService`, they might be confused with the standard HTTP communication terms Request and Response.

The `SimpleService` gRPC messages `Request` and `Response` have no intrinsic relation to an HTTP Request or an HTTP Response. That they share the same name is arbitrary.

Another fact to note in terms of service definition in a `.proto` file is that code auto-generation technology as well as language-specific libraries and framework use the procedures defined in the service section of the `.proto` file to create boilerplate procedure code. Using boilerplate code reduces programming labor because all a programmer needs to do is create the logic that is special to the procedure.

For example, in the demonstration gRPC API that accompanies this article and as shown in Listing 4 above, there are `Add()`, `Subtract()`, `Multiply()` and `Divide()` procedures defined in the `.proto` file that do simple math procedures as their names imply. If a code generator or library were not used, the programmer would have to write all the low-level HTTP/2 and routing logic. However, when a code nerator or library is used, all the programmer needs to do is provide the math logic. Thus all a programmer needs to do to implement the Add() procedure, if Node.js is their language of choice, is to use the Node.js library for gRPC and provide code as follows:

```
  function add(call, callback) {
      const input = call.request.numbers;
      const result = input.reduce((a, b) => a + b, 0);
      callback(null, {result});
 }
```

Listing 5

As you can see, while a `.proto` file is an implicit necessity for any gRPC development work, leveraging it to ease programming work provides an added benefit that is significant.

Packages and Namespaces

The Protocol Buffers specification provides a way to organize services according to a `package`. A `package` is a logical organizational unit that's found in a variety of programming languages and data formats. The actual term for a package in gRPC is `package`. The programming language Java also uses the term `package` as a way to organize code under a common name. Two of the more familiar packages in Java are **java.lang (https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html)** and java.io.

.NET on the other hand organizes code using the term `namespace`. `System` **(https://docs.microsoft.com/en-us/dotnet/api/system?view=netcore-3.1)** is a .NET namespace. The `System` namespace contains frequently used classes such as **Console (https://docs.microsoft.com/en-us/dotnet/api/system.console?view=netcore-3.1)**, **Buffer (https://docs.microsoft.com/en-us/dotnet/api/system.buffer?view=netcore-3.1)** and **Tuple (https://docs.microsoft.com/en-us/dotnet/api/system.tuple?view=netcore-3.1)**. C++ is also a programming language that uses the term `namespace`.

According to the Protocol Buffers **specification (https://developers.google.com/protocol-buffers/docs/proto#packages)**, the way you declare a package name in a **.proto**

(https://developers.google.com/protocol-buffers/docs/proto#packages) file is to use the reserved word package. The .proto file shown in Listing 1 at the beginning of this article declares the package named `simplegrpc` like so:

```
package simplegrpc;
```

While the Protocol Buffers specification allows you to declare a package name in the `.proto` file and thus implicitly associate that package to a service and methods, the way methods are concretely organized under a package name in code is left up to the programming language or framework implementing the particular gRPC API.

> **Q:** Does the programming language really matter?
>
> **A:** Yes. When planning how you're going to implement your gRPC API you need to give careful consideration to the purpose and scope of the API. The details of what the API is intended to do matter.
>
> Any programming language comes with benefits and tradeoffs. For example, Node.JS is based on a single-threaded, event-driven programming model. Thus, it's very good for doing **CRUD (https://en.wikipedia.org/wiki/Create,_read,_update_and_delete)** work with data in databases and in a filesystem. But, when it comes to computation-intensive processing, Node.JS can't take full advantage of multi-core processor computing environments because, with the exception of **worker (https://nodejs.org/api/worker_threads.html)** threads, the default Node.js single thread can only access one core.
>
> If you need to do computationally intensive work, a more efficient multi-threading programming language such as Java, C#, C/C++, Go, Rust, or Python is a better way to go. The trick to creating an effective gRPC API is to make sure the programming language meets the requirements of the API, logically and operationally.

Now that we've covered the essentials of the gRPC and Protocol Buffers, let's dive in and take a look at how to create an gRPC API based on the types and methods defined in a `.proto` file.

Creating Code

There are three ways you can create a gRPC API. One way is to use the **protoc (https://github.com/protocolbuffers/protobuf)** tool provided by the **CNCF (https://www.cncf.io/)** to auto-generate the language-specific objects that correspond to gRPC messages defined in the given `.proto` file. These objects have serialization and deserialization logic built-in. Thus, you can use them in conjunction with an associated gRPC server code running under HTTP/2 to implement the gRPC API. To learn more about using `protoc` to auto-generate gRPC code, read the *ProgrammableWeb* article on the topic **here (/api-university/how-to-auto-generate-grpc-code-using-protoc)**.

Another way is to use a language-specific project that has the libraries that provide an HTTP/2 server and does the routing calls to and from procedures on the gRPC server.*ProgrammableWeb* provides an in-depth article on using frameworks and libraries to implement a gRPC API **here (/api-university/how-to-make-use-grpc-libraries-and-frameworks)**.

The third way is to start from scratch and write the API from the ground up.

Starting from scratch includes writing not only the serializers and deserializers to encode and decode the Protocol Buffers messages to and from binary format, but it also includes writing all the mechanisms required to route a request to the particular gRPC method and then return a response once created. Also, if you're working with streams, your "start-from-scratch" code needs to support unidirectional and bidirectional streaming under HTTP/2. It's a lot of work.

Given the complexity involved when starting from scratch, the approach is not one that's typical in the real world. It requires too much expertise and work just to do simple things. The approaches most companies take is to auto-generate boilerplate code or build custom code that leverages pre-existing libraries to provide gRPC specific functionality.

Putting it All Together

The goal of this article has been to provide a discussion of the basic concepts and practices that are related to working with gRPC. We covered the 3 different types of gRPC interactions: request/response, unidirectional and bidirectional. We described how gRPC uses Protocol Buffers to exchange data between client and server and vice versa. We described the purpose and structure of the `.proto` file, The `.proto` file is the "reference manual" that drives gRPC serialization and deserialization.

We looked at the structure of gRPC messages and procedures as defined in a `.proto` file. Also, we discussed different approaches to creating the code that implements a gRPC API as defined in a `.proto` file. We touched upon auto-generating code or programming with libraries and frameworks.

At this point, you should have the vocabulary and background understanding of RPC necessary to understand the details of the next installment in this series. In the next installment, we're going to take a look at how gRPC is used in the real world. We'll look at how various companies and technologies take advantage of gRPC to move data at the speed required to meet mission-critical demands.

Epilog: Understanding the Demonstration API

The demonstration gRPC API that accompanies this article was created using the **gRPC Node.js libraries (https://www.npmjs.com/package/grpc)** created by Google. The gRPC API demonstration project is named SimpleService. The purpose of SimpleService is to provide examples of the basic building blocks generally found in a gRPC API. The API is composed of the following procedures:

- Add
- Subtract
- Multiply
- Divide
- Chatter
- Blabber

The first four procedures, `Add`, `Subtract`, `Multiple`, and `Divide` behave as their names indicate. They do math operations. These methods facilitate communication between client and server using a synchronous request/response interaction. The procedure `Chatter` is an example of unidirectional streaming between server and client. The procedure, `Blabber`, is an example of bidirectional streaming between client and server. (We'll cover the details of unidirectional and bidirectional streaming later in this series.) The method Ping is a convenience procedure that returns the string that was passed to it from an initiating request. `Ping` supports synchronous request/response communication.

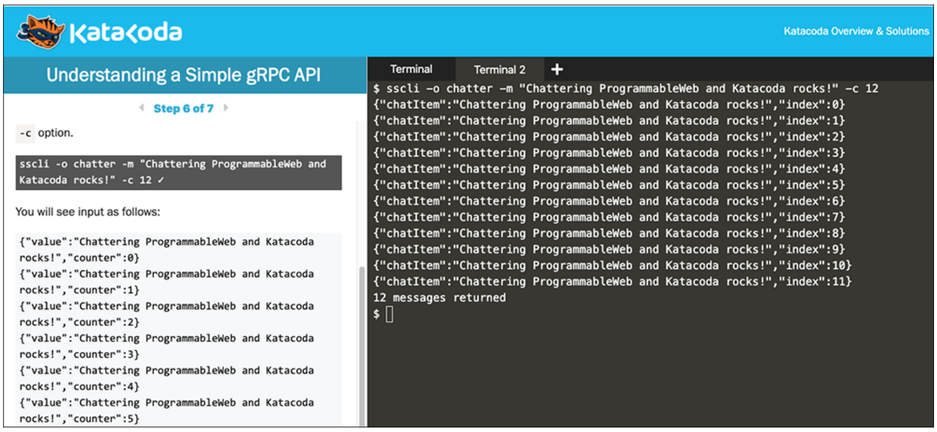
Also, the demonstration project ships with a CLI tool that you can use to work directly with the procedures in the API.

Getting the Source Code

The source code for both the API and CLI tool is in the project, simple-node-grpc version 1.0.1, found in the *ProgrammableWeb* GitHub repository, **here (https://github.com/programmableweb/simple-node-grpc/tree/master/1.0.1)**.

Getting Hands-On Experience

You can get direct hands-on experience using the demonstration API and the accompanying CLI client tool by taking the accompanying interactive scenario on Katacoda. This scenario shows you how to install the SimpleService gRPC API on a virtual machine running under Katacoda. Also, you can use the dedicated CLI tool within the Katacoda virtual machine to exercise the various procedures published by the API.


Katacoda
Katacoda Overview & Solutions
Understanding a Simple gRPC API
Step 6 of 7
Terminal
Terminal 2
+

`-c` option.

```
sscli -o chatter -m "Chattering ProgrammableWeb and Katacoda rocks!" -c 12 ✓
```

You will see input as follows:

```
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":0}
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":1}
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":2}
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":3}
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":4}
{"value":"Chattering ProgrammableWeb and Katacoda rocks!","counter":5}
```

```
$ sscli -o chatter -m "Chattering ProgrammableWeb and Katacoda rocks!" -c 12
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":0}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":1}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":2}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":3}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":4}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":5}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":6}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":7}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":8}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":9}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":10}
{"chatItem":"Chattering ProgrammableWeb and Katacoda rocks!","index":11}
12 messages returned
$
```

(https://www.mulesoft.com/)

ALSO OF INTEREST

Contact Sales (https://www.mulesoft.com/lp/contact)

What is REST API design? (https://www.mulesoft.com/resources/api/what-is-rest-api-design)

What are integration design patterns? (https://www.mulesoft.com/resources/api/what-are-integration-design-patterns)

(https://www.mulesoft.com)

(http[illegible]ity)