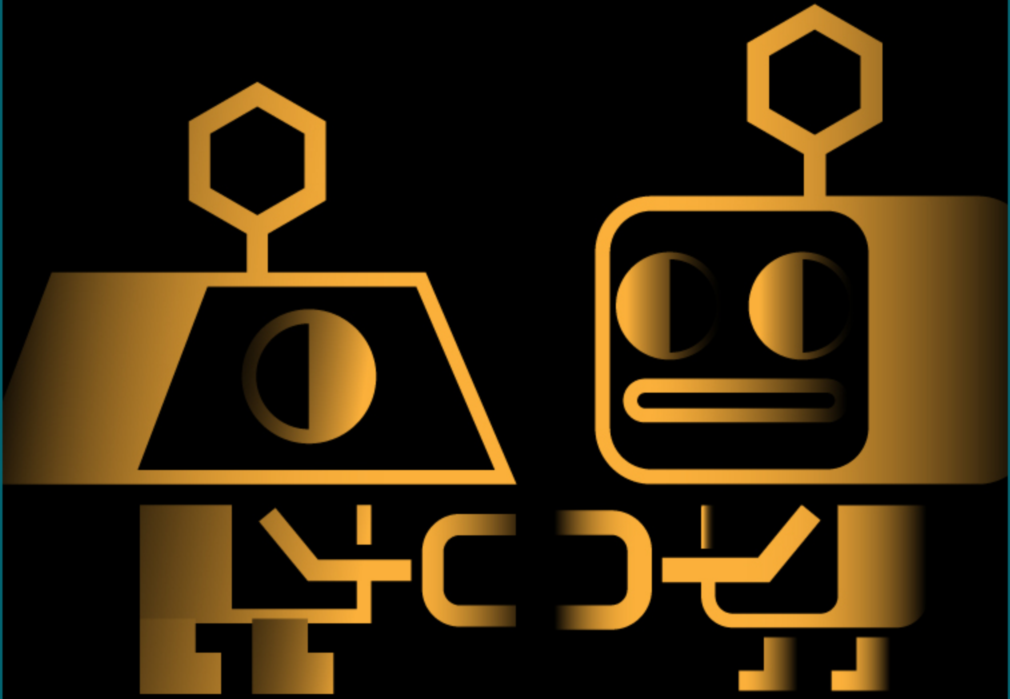
Bleeding Edge Press

PRACTICAL gRPC



Joshua B. Humphries, David Konsumer, David Muto, Robert Ross & Carles Sistare

Practical gRPC

By Joshua Humphries, David Konsumer, David Muto, Robert Ross, and Carles Sistare

Practical gRPC

This book expresses the authors views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Bleeding Edge Press, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Cover: CC0 Creative Commons, Free for commercial use, No attribution required
ISBN: 9781939902580
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Practical gRPC
Authors: Joshua Humphries, David Konsumer, David Muto, Robert Ross, and Carles Sistare
Acquisitions Editor: Christina Rudloff
Editor: Troy Mott
Website: bleedingedgepress.com

Table of Contents

Preface

Who is this book for?

This book is for engineers writing applications with different components that need to talk to each other. If you are a backend developer that wants a solution with better performance than JSON and REST for connecting microservices, this book is for you. If you are looking for strong contracts and streaming when building APIs and services, this book is for you.

This book explains gRPC, at multiple layers and with precise examples, to empower engineers to build these types of systems. It's for engineers that want to know the entire gradient, from the simplest form of issuing an RPC to more advanced functionality and even how the RPC itself is structured on the wire.

What do you need to know prior to reading?

This book is intended for software engineers, so having some level of experience will be very valuable in getting the most from this book.

The book includes code samples in several languages. Most of the examples are simple enough that you don't need to be an expert in any particular language to follow along. In fact, you may not be familiar with the given language at all, but should still be able to follow the logic and intent. Having said that, Go and Ruby developers will likely find themselves most comfortable reading the code because most of the examples are in those two languages.

This book does not attempt to introduce or define things like "service-oriented architectures" and "micro-services," so an intermediate level of experience with these kinds of distributed systems will be valuable. It does provide an overview and brief history of communication protocols, but some familiarity with standards like HTTP/REST and JSON will be really helpful to get the most out of it.

Online examples

All of the code for the example application, as well as many of the code examples throughout the book, can be found in GitHub: **https://github.com/backstopmedia/gRPC-book-example**

What will this book provide?

This book will introduce you to gRPC and explain how it compares and contrasts with similar technologies. It will introduce you to Protocol Buffers,a key technology for using gRPC. It then provides information on how to make use of gRPC, from the most basic of usage, all the way to advanced capabilities. And it also provides lots of information for making the most of gRPC: from best practices for defining and evolving your gRPC APIs to tools in the ecosystem and how they can be leveraged to get the most out of gRPC and even extend it.

By the end of this book, you should have a clear understanding of the following:

- What gRPC is and what its role is in applications.
- How to use Protocol Buffers and gRPC to connect systems.
- The mechanics of gRPC, including the underlying HTTP/2 protocol as well as how data is serialized and de-serialized.
- Tools and practices to aid with successfully implementing gRPC in your applications.

Author Bios

Carles Sistare (**`carlessistare`** on GitHub, **`CarlesSistare`** on Twitter) is a challenge trotter eager of projects who offer the opportunity to learn about the most recent technologies. Keeping a close eye on the open source community, three years ago, he began to be interested in gRPC, and quickly saw the benefits when it comes to microservice communication optimisations. Since then he has been doing several talks in France in order to evangelise gRPC and made **`grpc-promise`**. Carles had the opportunity to co-found **`OGURY`**, ad-tech company based in Paris, where currently works as Head Architect.

David Konsumer (**`konsumer`** on GitHub, **`davidkonsumer`** on Twitter) has been working with open-source for about 20 years, with a strong focus on JavaScript. Before working with Node.js, he used PHP and

Python to accomplish his daily programming tasks. He worked at NodeSource, to make N|Solid, Plex, Janrain, and Intel on the App Developer Program website. He also did lots of agency, freelance, and contract-work, mostly for the web and cloud-based backend work. He taught programming classes at Free Geek, Portland. David is an advocate for accessible programming and the idea that everyone can learn to code, and it should be easy enough to get cool stuff done. He made **`grpcnode`**, **`node-protoc-plugin`**, and **`json2x`**, useful tools for getting things done with gRPC/protobuf in Node.js. David currently works as Lead of Engineering at Gummicube, an App-store Optimzation company based in San Jose, running the development department in Portland, OR.

David Muto (**`pseudomuto`** on GitHub, **`pseudomuto`** on Twitter) has been writing software professionally for over 15 years, and is a strong believer in continuous learning. Over the years he has written production quality code in over a dozen languages, taught application development at **George Brown College** (in Toronto) and worked on everything from web and mobile apps, to low-level systems and RPC services. He's a serial OSS contributor and the author of **protoc-gen-doc**, a tool for generating documentation from **Protocol Buffers**. He also recently published **protokit**, a library for building your own `protoc` plugins in Go. David currently works at **Shopify** leading a team that develops highly available, low-latency, distributed services in the Data Science and Engineering department.

Joshua Humphries (**`jhump`** on GitHub) has been working with Protocol Buffers and building RPC systems and related facilities for over six years. He was first introduced to Protocol Buffers and “Stubby” (gRPC's forebearer) while working at Google. Afterwards, he led a team that worked on protobuf-based RPC at Square, including “smart clients” in Java, Go, and Ruby that handled service discovery, load balancing, automatic retries, and automatic geographic failover. Joshua has been an advocate of gRPC since its initial release. He continues his work with Protocol Buffers and gRPC as part of an infrastructure team at FullStory, a customer experience management platform. He is a contributor to the Go open-source projects for **gRPC** and **Protocol Buffers**, the author of a Go library for Protocol Buffer reflection named **`protoreflect`**, and the author of gRPC-related projects open-sourced by FullStory, including **`grpcurl`**.

Robert Ross, but people call him Bobby Tables (**`bobbytables`** on GitHub, **`bobbytables`** on Twitter) is a full time staff software engineer at **Namely** where he works on the architecture team. He writes with Go and

Ruby primarily, but can swing some JavaScript if needed. He operates **FireHydrant** on the side while listening to the Hamilton soundtrack an unhealthy amount.

Technical Reviewers

Special thanks goes to our technical reviewers who helped make this a better book: Bojan Djurkovic, David Gasquez, and Jeff Willette.

Introduction 1

This book aims to provide a practical approach to learning and using gRPC. It attempts to catalog and teach not only the basics of gRPC, which you could also find in countless blog posts and the gRPC home page itself, but also to show the more interesting (perhaps less well-documented) aspects of gRPC. The book even demonstrates some of the gRPC pitfalls, and what you need to know to overcome them.

The first few chapters in this book introduce gRPC, describing what it is and how it compares and contrasts to other technologies in the same space. We'll also dive into how to actually use and apply it. Next, the book will venture into more advanced techniques. These advanced chapters will arm you with the tools to use gRPC to the fullest, so you can truly harness its features for solving problems and building software systems. There are also more practical concerns covered, like best practices for evolving your RPC interfaces and schemas. The later chapters in the book will detour into related technologies, some that complement gRPC in production environments and some that aid developers in building and testing applications that use gRPC.

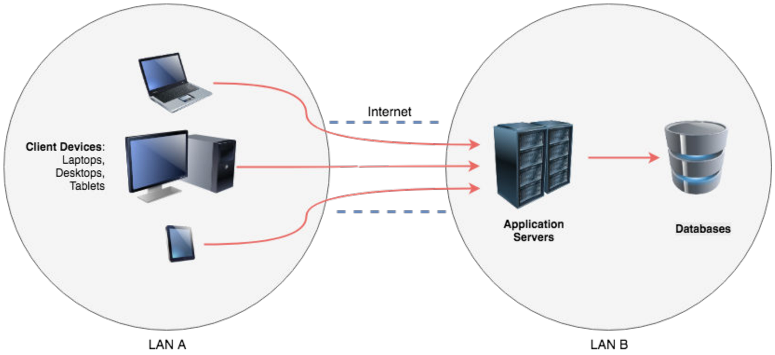
Computer networks and distributed computing

The earliest ideas for connecting multiple computers to form a network came about in the 1960's. Shortly after, the ARPANET (the precursor to the Internet) was created. In the 1970's, email was invented, which became the most widely used distributed application on the ARPANET. In these early days, the power of the network was mostly for sharing information by sending data from one computer to another on the other side of the country. Distributed computing became its own field of computer science in the 1970's. So the study of how multiple computers could be used to solve larger problems than a single computer could solve was of great interest. This fledgling Internet connected the computing resources of numerous universities and government organizations, creating a large pool of compute power.

The ARPANET eventually grew into the Internet we know today, connecting millions of computers all across the globe. It powered the rise of the World Wide Web in the 1990's, and today the Internet is a utility, much like electricity or water. It is available almost everywhere in the USA and also in most of the world. With more recent innovations in mobile computing and embedded systems, the Internet has become an integral component in modern life, not just in business.

Request-response protocols

Numerous networking protocols have been developed for sharing information from one point of the Internet to the other. Almost all of the protocols use the TCP/IP Internet protocol suite. This has enabled the free flow of information across the globe, and it has also had an immeasurable impact on modern business and commerce, which rely greatly on computer networks and distributed computing.



This diagram shows a very simple but typical arrangement. In enterprise settings, the two halves will likely be on the same network, or perhaps a VPN is used to securely connect clients to the servers. For eCommerce, SaaS (Software as a Service) and other Internet offerings, the servers might be hosted in the provider's datacenter or even with a third-party provider like Amazon Elastic Compute Cloud (EC2), Google Cloud Platform, or Microsoft Azure. The main components of interest here are clients, on the left, and servers, on the right.

A **client** is a program that initiates communication-usually by creating a TCP connection. It may be an end-user program, initiating communication to request information or resources that it presents to a user. In web applications, for example, the client is a web browser such as Chrome or Firefox.

A **server** is a program that accepts these client connections and in turn processes their requests. There are many kinds of servers. The diagram shows an application server, so named because it serves data to the client application. (It is also called a web server if it uses HTTP as the communication protocol). As shown in the diagram, the application server may involve other computers in order to serve responses. In that case, the server *also acts as a client*. In the diagram, the application server acts as a database client, requesting information from the database server.

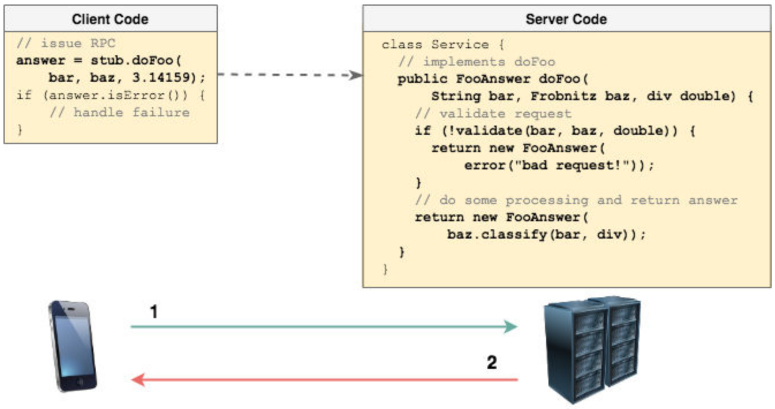
Most networking protocols follow this pattern: a client creates a network connection to the server and sends requests. The server accepts requests, performs some processing, and then sends responses. This request-response flow was conceived in the early days of networking in the 1960's and has been foundational to distributed computing ever since.

Remote Procedure Calls

RPC stands for **R**emote **P**rocedure **C**alls. It is a programming model built on top of request-response network protocols. Issuing an RPC in a client amounts to invoking a procedure from application code. For a server, servicing an RPC amounts to implementing a procedure with a particular signature.

In the client, the objects that expose these procedures are called **stubs**. When application code invokes a procedure on a stub, the stub translates the arguments into bytes and then sends a request to the server, the contents of which are the serialized arguments. When it gets back a response from the server, it translates the bytes into a result value, which is then returned back to the application code that called it.

In the server, the objects that expose these procedures are **service implementations**. The server machinery receives the request, translates the bytes back into procedure arguments, and then invokes a procedure on the service implementation, passing it those arguments. The service implementation performs its business logic and then returns a result, either a value on success or an error code on failure. (In some RPC implementations, servers can return both values and an error code). The server machinery then translates this result into bytes, which then become the response that is sent back to the client.



```
// issue RPC
answer = stub.doFoo(
    bar, baz, 3.14159);
if (answer.isError()) {
    // handle failure
}
```

```
class Service {
  // implements doFoo
  public FooAnswer doFoo(
      String bar, Frobnitz baz, div double) {
    // validate request
    if (!validate(bar, baz, double)) {
      return new FooAnswer(
          error("bad request!"));
    }
    // do some processing and return answer
    return new FooAnswer(
        baz.classify(bar, div));
  }
}
```

1. Client serializes `bar`, `baz`, and `3.14159` to bytes and sends on network connection to server.
2. Server serializes `FooAnswer` result and sends on network connection to client.

RPC is not a new programming idiom: proposals for remote procedure call semantics were written in the 70's, and practical RPC implementations appeared in the 80's, such as the Network File System (NFS).

gRPC is a cross-platform RPC system that supports a wide variety of programming languages. It excels at providing high performance and ease of use, to greatly simplify the construction of all types of distributed systems.

Summary

This chapter introduced the domain in which gRPC is used (distributed computing), and provided a brief overview of how this domain has evolved over time.

In the next chapter, we will go into greater detail as to what exactly gRPC is, and how it compares to similar technologies.

What is gRPC? 2

A logical place to start when answering, “What is gRPC,” is to first talk about what the letters mean. It is an acronym after all. In the introduction, we briefly covered RPC: **R**emote **P**rocedure **C**alls. This is the programming idiom that gRPC presents to application developers.

So what is the “g” in “gRPC?” This technology was created by Google as an open source evolution of their internal RPC technology named Stubby, and they continue to be the stewards of the official open source project. So the “g” is widely thought to stand for “Google.” Google has tried to make it clear, however, that they want a community collaborating with and accepting contributions and input from developers outside of Google. To that end, the “g” has never officially meant “Google.” In fact, a new meaning is assigned to the “g” for each release. In August 2016 during the 1.0 release of gRPC the acronym stood for "**g**RPC **R**emote **P**rocedure **C**alls.” In the numerous minor releases since then, the “g” has gone through equally numerous redefinitions, including good, green, gambit, glamorous, and glorious. This history of what the “g” stands for is documented in their main repo on Github: **https://github.com/grpc/grpc/blob/master/doc/g_stands_for.md**.

The next sections will go into more detail about what gRPC is, starting with some of the principles on which gRPC is built. You’ll also learn what distinguishes gRPC from other RPC systems, and how it compares to other widely-used technologies.

HTTP and REST

HTTP is the protocol that powers the worldwide web. It stands for **H**yper**T**ext **T**ransfer **P**rotocol. It’s a text-based request-response protocol. Because it is text-based, the protocol itself is human-readable (data payloads, which can be binary, may not be). It was originally designed for accessing documents across the Internet, but it has become a ubiquitous protocol-not just for web browsers, but for all kinds of computer-to-computer interactions. Subsequently, HTTP and “web technology” have become the substrate of choice for many kinds of interactions thanks to common-place open source software components and a range of hardware components that make it possible to build and deploy HTTP-based systems with ease and at great scale.

REST, which stands for **RE**presentational **S**tate **T**ransfer, is based on HTTP. It defines constraints and conventions on top of HTTP that are intended to provide global interoperability and potential for scalability. A key architecture constraint in REST is statelessness: application servers do not store client context in between requests. Any client state needed to process a request is included with each request. This enables systems to grow to very large scale: requests can be load balanced across a large pool of servers, and multiple requests from a single client don’t have to be handled by a single server.

The concepts used to define REST APIs focus on managing “documents” or “entities.” REST’s primitives are the various HTTP methods: GET, PUT, DELETE, POST, PATCH, etc. These methods map, more or less, to CRUD operations on these entities. (CRUD stands for **C**reate, **R**ead, **U**pdate, and **D**elete: the set of actions needed to work with data.) A REST API defines naming conventions so that clients can construct URLs to identify particular documents or entities. It also defines semantics for each of the methods when applied to a particular resource or document. Request and response payloads are used to transmit document contents. This is a very straight-forward model for a service that exposes a database-like or filesystem-like resource: you can use HTTP GET requests to list documents or retrieve document contents, and you can use PUT, PATCH, and DELETE requests to create new documents or to modify or delete existing ones. When a service has more complicated and less document-centric operations, mapping them to REST can be a bit more challenging.

There are numerous RPC systems that are built on top of REST. Some leverage the full flexibility of REST, such as Swagger and JAX-RS (the Java APIs for REST-ful web services). Swagger is a cross-platform and language-agnostic technology whose IDL-the Swagger spec-describes rules for how a set of operations is represented in HTTP requests and responses. The spec describes the HTTP method, patterns for URI paths, and even the request and response structures for each operation. Code generation tools exist to create client stubs and server interfaces from a Swagger spec. JAX-RS, on the other hand, is Java-only, and its IDL is Java itself. A Java interface is used to define the operations, with annotations that control how its methods, arguments, and return values are mapped to the HTTP protocol. Servers can provide implementations of the interface. And a client stub is a runtime-generated implementation of that interface, which uses reflection to intercept method calls and transform them into outgoing HTTP requests.

Other RPC systems are based on HTTP. From a certain point of view, they can be seen as narrow subsets of REST: they still adhere to its architectural principles, but put forth significant constraints/conventions on resource naming, methods, and the encoding of documents to simplify the implementation of both clients and servers. Such systems were created during the web's "adolescent era," including XML-RPC and SOAP. As the popularity of XML declined, different approaches to content encoding came into favor, such as JSON-RPC. In fact, gRPC finds itself in this category.

RPC systems

Here we cover the typical traits shared by all RPC systems. There are systems that contain some of these traits, whereas mature systems provide them all:

1. Since it's RPC, the programming model is of course procedure calls: the networking aspect of the technology is abstracted away from application code, making it look almost as if it were a normal in-process function call versus an out-of-process network call.
2. There is a way to define the interfaces as the names and signatures of the procedures that can be invoked along with the data types that are exchanged as arguments and return values. For RPC systems that are language-agnostic (e.g. can be used with multiple programming languages), the interface is typically defined in an **I**nterface **D**efinition **L**anguage, or IDL for short. IDLs can describe the shape of data and interfaces but cannot express business logic.
3. The RPC systems often include code generation tools for transforming the interface descriptions into usable libraries. And they include a runtime library that handles the transport protocol details and provides an ABI for the generated code to hook into that transport implementation. Some systems rely more heavily on runtime reflection and less on code generation. And this can even vary from one programming language to another for the various implementations of a single RPC system.
4. Unlike REST, these systems typically do not expose all of the flexibility of HTTP. Some eschew HTTP completely, opting for a custom

binary TCP protocol. Those that do use HTTP as a transport tend to have rigid conventions for mapping RPCs to HTTP requests, which often cannot be customized. The details of what the HTTP request looks like are meant to be an implementation detail encapsulated in the system's transport implementation.

gRPC

Let's talk about gRPC first. The first thing to note is that the architecture of gRPC is layered:

- The lowest layer is the transport: gRPC uses HTTP/2 as its transport protocol. HTTP/2 provides the same basic semantics as HTTP 1.1 (the version with which nearly all developers are familiar), but aims to be more efficient and more secure. The new features in HTTP/2 that are most obvious at first glance are (1) that it can multiplex many parallel requests over the same network connection and (2) that it allows full-duplex bidirectional communication. We'll learn more about HTTP/2 and the ways it differs from and improves on HTTP 1.1 later in the book.
- The next layer is the channel. This is a thin abstraction over the transport. The channel defines calling conventions and implements the mapping of an RPC onto the underlying transport. At this layer, a gRPC call consists of a client-provided service name and method name, optional request metadata (key-value pairs), and zero or more request messages. A call is completed when the server provides optional response header metadata, zero or more response messages, and response trailer metadata. The trailer metadata indicates the final disposition of the call: whether it was a success or a failure.

  At this layer, there is no knowledge of interface constraints, data types, or message encoding. A message is just a sequence of zero or more bytes. A call may have any number of request and response messages.
- The last layer is the stub. The stub layer is where interface constraints and data types are defined. Does a method accept exactly one request message or a stream of request messages? What kind of data is in each response message and how is it encoded? The answers to these questions are provided by the stub. The stub marries the IDL-defined interfaces to a channel. The stub code is

generated from the IDL. The channel layer provides the ABI that these generated stubs use.

Another key component of gRPC is a technology called Protocol Buffers.Protocol Buffers, or “protobufs” for short, are an IDL for describing services, methods, and messages. A compiler turns IDL into generated code for a wide variety of programming languages, along with runtime libraries for each of those supported programming languages. You’ll learn much more about Protocol Buffers in the next chapter.

It is important to note that Protocol Buffers have a role only in the last layer in the list above: the stub. The lower layers of gRPC, the channel and the transport, are IDL-agnostic. This makes it possible to use any IDL with gRPC (though the core gRPC libraries only provide tools for using protobufs). You can even find unofficial open source implementations of the stub layer using other formats and IDLs, such as `flatbuffers` and `messagepack`.

Other RPC systems

There are too many other RPC systems to realistically review them all, but the table below enumerates the more well-known systems. Some are older technologies that are no longer widely in use (or they are in use primarily in legacy and enterprise systems). Some are newer technologies, like gRPC. They are listed oldest to newest, showing the history and evolution of RPC.

RPC system	Transport	Language support	Notes
Sun/ONC RPC	Custom TCP or UDP	Various (Mainly C)	Early standard for RPC. Used in NFS (**N**etwork **F**ile **S**ystem). Custom “Rpcgen” IDL.
CORBA	Custom TCP	Various	Early standard for RPC and distributed objects. Custom IDL.
DCOM	Custom TCP	Various (Windows Only)	Distributed objects. Proprietary Microsoft serialization. Supports distributed garbage collection. Custom IDL.
Java RMI	Custom TCP	Java	Distributed objects. Uses Java serialization.

RPC system	Transport	Language support	Notes
XML-RPC, SOAP	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses XML for message encoding.
JSON-RPC	HTTP 1.1	Various	Specifies the communication protocol, not the client and server programming models. Uses JSON for message encoding. Technically a subset of REST.
Apache Thrift	Custom TCP	Various	Custom IDL that is similar to protobuf. No streaming.
Apache Avro	Custom TCP *or* HTTP 1.1	Various	Code generation not required. Data includes schema. IDL is formatted as JSON.
Go net/rpc	HTTP 1.1	Go	Uses Go's GOB encoding.
Cap'n Proto	Custom TCP	Various (Mainly C++)	Distributed objects. Promise pipelining. Strives for zero-overhead serialization. Custom IDL.
Twirp	HTTP 1.1	Various	No streaming. Can use JSON as encoding. Uses Protobuf as IDL.

Some notable innovations of RPC systems over the years are the “distributed object” paradigm, Cap’n Proto’s “promise pipelining", and gRPC’s streaming.

To describe a distributed object system, we first need to talk about a key constraint in non-distributed-object RPC systems: the operations that a server exposes and the implementations to which they are bound are fixed. When a server starts up, its service implementations are registered to be exposed by the network server. In an OOP paradigm, this amounts to the service implementations being singletons. To contrast, in a distributed object system, the server can dynamically allocate new objects at runtime whose methods are also exposed via RPC. The existence of these objects can be communicated to clients as RPC return values. A typical RPC result might be a value that is serialized to bytes and transmitted to the client. But with distributed objects, the system might just serialize and transmit “handles,” not the actual values. These handles are then used to create new stubs that are returned to the client application as the RPC result. Calling methods on these stubs in turn issues

RPCs, which result in invoking methods on the underlying instances in the server.

Cap'n Proto's "promise pipelining" allows clients to pipeline many requests without waiting on a server response, even when the arguments to one RPC may depend on the result of an earlier one! It does this by allocating a promise "handle" for each RPC in the client. So, even before the server may have received the first request, the client has a handle to the first response. It can then refer to that promise in a subsequent request, and send that to the server. The server is responsible for resolving these handles: once the result for the first RPC is computed, the server machinery can bind it to the handle in the second RPC and then begin computing a result for the second RPC. This feature is an optimization to decrease the latency for multi-step operations, because it can greatly reduce the amount of time waiting on messages to transit the network.

gRPC's streaming is covered below, but it will also be discussed in more detail later in the book.

Streaming

The closest siblings to gRPC are Thrift, an Apache project, and Twirp, open sourced by Twitch. But neither of these include support for streaming. gRPC, on the other hand, was designed with streaming in mind from the outset.

Streaming allows a request or response to have an arbitrarily large size, such as operations that require uploading or downloading a huge amount of information. Most RPC systems require that the arguments of an RPC be represented as data structures in memory that are then serialized to bytes and sent on the network. When a very large amount of data must be exchanged, this can mean significant memory pressure on both the client process and the server process. And it means that operations must typically impose hard limits on the size of request and response messages, to prevent resource exhaustion. Streaming alleviates this by allowing the request or response to be an arbitrarily long sequence of messages. The cumulative total size of a request or response stream may be incredibly large, but clients and servers do not need to store the entire stream in memory. Instead, they can operate on a subset of data, even as little as just one message at a time.

Not only does gRPC support streaming, but it also supports full-duplex bidirectional streams. Bidirectional means that the client can use a stream to upload an arbitrary amount of request data and the server can use a stream to send back an arbitrary amount of response data, all in the same RPC. The novel part is the "full-duplex" part. Most request-response protocols, including HTTP 1.1 are "half-duplex." They support bidirectional communication (HTTP 1.1 even supports bidirectional streaming), but the two directions cannot be used at the same time. A request must first be fully uploaded before the server begins responding; only after the client is done transmitting can the server then reply with its full response. gRPC is built on HTTP/2, which explicitly supports full-duplex streams, which means that the client can upload request data at the same time the server is sending back response data. This is very powerful and eliminates the need for things like web sockets, which is an extension of HTTP 1.1, to allow full-duplex communication over an HTTP 1.1 connection. Thanks to streaming, applications can build very sophisticated conversational protocols on top of gRPC.

Where to use gRPC

So now you know that gRPC is a request-response protocol for streaming RPC that uses Protocol Buffers to define interfaces and messages. But, what about *where* or *why* you would use gRPC, or what programming languages can be used with gRPC.

The "where" is pretty easy: you can leverage gRPC almost anywhere you have two computers communicating over a network:

1. **Microservices**: gRPC shines as a way to connect servers in service-oriented environments. One of the original problems its predecessor, Stubby, aimed to solve was wiring together microservices. It is well-suited for a wide variety of arenas: from medium and large enterprises systems all the way to "web-scale" eCommerce and SaaS offerings.
2. **Client-Server Applications**: gRPC works just as well in client-server applications, where the client application runs on desktop or mobile devices. It uses HTTP/2, which improves on HTTP 1.1 in both latency and network utilization. This means you get improved response times and longer battery life.
3. **Integrations and APIs**: gRPC is also a way to offer APIs over the Internet, for integrating applications with services from third-party providers. As an example, many of Google's Cloud APIs are exposed via gRPC. This is an alternative to REST+JSON, but it does not have to be mutually exclusive. There are tools for easily exposing gRPC services over REST+JSON, such as `grpc-gateway`.

4. **Browser-based Web Applications**: The last big area superficially seems like a poor fit. JavaScript code running in a browser cannot directly utilize gRPC since gRPC has a strict requirement for HTTP/2, but browser XHRs do not. However, as mentioned above, there are tools for exposing your gRPC APIs as REST+JSON, where they can then be easily consumed by browser clients.

For each of the above situations where you might use gRPC, there are alternatives. In fact, REST and JSON are a sort of de-facto standard for all of these situations. So why would you use gRPC instead? There are several dimensions along which gRPC wins out over the others, particularly over REST and JSON:

1. **Performance/Efficiency**: HTTP 1.1 is a verbose protocol and JSON is a very verbose message format. They are great for human-readability, but less so for computer-readability, requiring a good deal of string parsing. HTTP 1.1 also has a severe limitation of how a single connection can be used for multiple requests: all requests must be sent back in the order the corresponding requests were received. So clients that use pipelining will see head-of-line blocking delays, but the later responses may have been computed quickly, and must wait for earlier responses to be computed and transmitted before they can be sent. And the other alternative, using a connection for only one request at a time and then using a pool of connections to issue parallel requests, consumes more resources in both clients and servers as well as potentially in between proxies and load balancers.
   HTTP/2 and Protocol Buffers do not have these problems. HTTP/2 is much less verbose thanks largely to header compression. And it supports multiplexing many requests over a single connection. Protocol Buffers, unlike JSON, were designed to be both compact on the wire and efficient for computers to parse.
   The result is that gRPC can reduce resource usage, resulting in lower response times compared to using REST and JSON. This also means reduced network usage and longer battery life for clients running on mobile devices.
2. **Productive Programming Model**: The programming model with gRPC is simple to understand and leads to developer productivity. Defining interfaces and canonical message formats in an IDL means a lot of boilerplate code is auto-generated.
   Forget the days of manually wiring up server handlers based on URI paths and then manually marshalling paths, query string pa-

rameters, requests, and response bodies. Similarly, forget the days of manually creating HTTP request objects, with all of the same overhead on the server side.

While there are myriad tools and libraries that can alleviate this burden for REST+JSON (including a great number of home-grown, proprietary solutions within organizations and projects), they tend to vary significantly from one programming language to another, possibly even from one project to another. And in some cases they are incomplete: one library, for example, may address one aspect (reduce boilerplate in servers) but fail to address others (common definition of interfaces and message schemas, reduce boilerplate in clients). gRPC is thorough: it addresses all of these concerns. It also does so in a way that is consistent across numerous programming languages. If you write code in multiple languages and/or in a polyglot environment, this is particularly poignant.

3. **Streaming**: One of gRPC's "killer features" is full-duplex bidirectional streaming. While the great majority of RPCs will be simple "unary" operations (single simple request, and single response), there are often cases where something more sophisticated is called for. Whether its affinity (often for performance reasons), server-push facilities for sending notifications, or something more complicated, it can be done using gRPC streams.

   Historically, this required writing a custom TCP-based protocol. This can be a good solution in the datacenter, but is much less realistic when clients and servers are separated by a WAN or the Internet: open source proxy/load balancing software and hardware load balancers aren't as effective when they don't understand the protocol. Even in the datacenter, where you might have a fairly homogeneous set of microservices that can all speak the protocol, developing a custom protocol and client and server libraries is not trivial.

   It is possible to do some of these sophisticated things with HTTP 1.1. The most flexible and widely-supported approach involves using web sockets, which basically let you create a custom TCP-based protocol, tunneling it over HTTP connections. But this still comes with the cost of inventing the protocol and implementing custom clients and servers.

4. **Security**: gRPC was designed with security in mind. It is of course possible to use HTTP/2, and thus gRPC, in an insecure way with plain text connections. But when using TLS (transport-level security, sometimes called SSL), HTTP/2 is more strict than HTTP 1.1. It

allows only TLS 1.2 or higher; numerous cipher suites are blacklisted because they provide inadequate security; and compression and re-negotiation are disabled. This greatly reduces the surface area of TLS vulnerabilities to which HTTP/2 connections are subjected.

Now that we know where gRPC is well-suited and why it is a good fit, the last question is, “What programming languages can you use with gRPC?” The following table shows the languages supported by the core gRPC and Protocol Buffers projects. In addition to these officially supported languages, you can also find unofficial open source gRPC libraries for other languages (such as Rust, Erlang, and TypeScript to name just a few).

Language	**Notes**
C++	Based on the C core implementation. One of the original languages supported by `protoc`.
Java	One of the original languages supported by `protoc`.
Python	One of the original languages supported by `protoc`.
Go	Support is provided via a protoc plugin named `protoc-gen-go`. Reflection support (1) is limited.
Ruby	Based on the C core implementation. Lacks reflection support.
C#	Based on the C core implementation. Lacks reflection support.
JavaScript	Node.js only (not for browsers (2)). Two implementations: one based on the C core, another that is pure JavaScript.
Android Java (aka “javanano”)	A lighter-weight approach to generated Java classes: simpler representation, mutable POJOs (instead of immutable POJOs with corresponding builders), and much fewer generated methods. Lacks reflection support. No support for gRPC servers.
Objective-C	Based on the C core implementation. Lacks reflection support. No support for gRPC servers.
PHP	Based on the C core implementation. Lacks reflection support. No support for gRPC servers.

Language	Notes
Dart	Support is provided via a protoc plugin named `protoc-gen-dart`. Lacks reflection support.

(1) The term "reflection support" refers to being able to use Protocol Buffer descriptors to define and interact with message types at runtime. Without reflection, applications must always use `protoc` to generate code for message types. These limitations are only in the core gRPC and Protocol Buffer projects. For most, there exist third-party libraries to fill the gap.

(2) Browsers must use alternative stubs and a web-to-gRPC proxy such as `grpc-gateway`, which is described later in the book, or `grpc-web` **https://github.com/grpc/grpc-web**.

We see some interesting constraints to the supported languages above. It is obvious why Objective-C and Android Java don't support gRPC servers: they target mobile devices that will always act as clients. For PHP, which is often used for building web servers, and the omission is a technical one: PHP integrates with web servers, as a module or via CGI. These interfaces were not designed with HTTP/2 in mind, so they are not compatible (even if the web server such as Apache, Nginx, or Lighttpd does support HTTP/2).

Summary

In this chapter, we learned the details of what gRPC is and what differentiates it from similar technologies. We also learned what makes gRPC a good fit for numerous applications and what languages can be used with gRPC.

The rest of the book focuses on how to actually use gRPC. As you might expect, we'll start with the basics and then move into several more advanced topics for making the most of gRPC.

But first, we take a detour to talk about a technology that is imperative to know in order to use gRPC: Protocol Buffers.

What are Protocol Buffers? 3

gRPC focuses on optimizing three layers of the OSI model:

- Application (HTTP2)
- Presentation (Protobuf)
- Session (RPC)

We'll explain every layer of optimization throughout this book, but now it is time to take a deep dive into Protobuf or Protocol Buffers. One of the biggest concerns when it comes to REST/JSON micro-service architectures is the "I couldn't care less" attitude about bandwidth. The whole JSON message is sent among clients and servers in a plain text form, and its payload becomes repetitive and sometimes unnecessary.

JSON APIs are widely adopted by many developers because they are easy to read and debug, and JSON is implemented natively in many languages, such as NodeJS. For early projects, which need to go fast and have the Firsts prototypes up and running in a very short time, a JSON API can be recommended. But problems can arise. JSON messages are supposed to evolve, but client and server source code can be duplicated, and any feature connected to the same API will need to hard code the same parsing/interpretation functionality.

gRPC uses Protocol Buffers as a remedy to all of these known issues:

- Formal contracts
- Code generation
- Bandwidth optimization

Protocol Buffers is a standard process for serializing data, with the goal to minimize the data representation of a given structure. Protocol Buffers were created by Google early in 2001, but not publically released until 2008. The method offers a syntax in order to describe plenty of strong typing variables, as well as an IDL (Interface Description Language) in order to autogenerate source code with several programming languages and easily parse a stream of bytes into the given object. A .proto file needs to be created, and in it we'll add all desired messages/objects with their specific fields, as well as the services that use those messages. Once the .proto file is edited, we need to compile it with `protoc` in order to generate the classes for every given message.

Here you can see a typical .proto file example:

```
syntax = "proto2";

message Film {
    string title = 1;
    string director = 2;
    string producer = 3;
    optional string release_date = 4;
    repeated string characters = 5;
}
```

Proto version

The explanations in this book will be based on proto2. For the record, the version is up to proto3, and here are some of the enhancements:

- Since all fields are already optional by default, proto3 got rid of `optional`.
- Grouped fields are no longer supported. As explained later in the book, Nested fields will be good enough.
- ....

Tags

As you can see in the example above all variables are strong typed (string), and they are followed by a sequential numbered tag. That will be the order of the serialization of all the fields for the message `Film`, and it will be mandatory to be able to parse the binary message. This order will of course not have to be modified at any time. Once a `Film` has been serialized, it isn't possible to reconstruct the original structure.

Scalar types

Scalar types, in contrast with compound types, are single value types. Every language implementation has its own equivalent:

Protobuf	C++	Java	Python	Go
double	double	double	float	*float64

Protobuf	C++	Java	Python	Go
int32	int32	int	int	*int32
int64	int64	long	long	*int64
uint32	uint32	int	int	*uint32
uint64	uint64	long	long	*uint64
sint32	int32	int	int	*int32
sint64	int64	long	long	*int64
fixed32	uint32	int	int	*uint32
fixed64	uint64	long	long	*uint64
sfixed32	int32	int	int	*int32
sfixed64	int64	long	long	*int64
bool	bool	boolean	bool	*bool
string	string	String	unicode	*string
bytes	string	ByteString	str	[]byte

All types are encoded with a variable size depending on the magnitude of the value. Fixed types have fixed sizes (4 or 8 bytes).

This table of types is based on the official documentation for Google Protobuf

Extended types information on other languages is supported in proto3

Nested types

Nested types allow you to define messages inside of other messages.

```
syntax = "proto2";

message Film {
    message Character {
        string name = 1;
        int64 birth = 2;
    }
    string title = 1;
    string director = 2;
```

```
        string producer = 3;
        optional string release_date = 4;
        repeated Character characters = 5;
    }
```

In the example above, if Character needs to be used elsewhere it may be imported as follows:

```
    message OtherFilm {
        repeated Film.Character characters = 1;
    }
```

Maps

Associative maps are allowed in protobuf:

```
    syntax = "proto2";

    message Film {
        message Character {
            string name = 1;
            int64 birth = 2;
        }
        map<string, Character> roles =  1;
    }
```

The key part can be any scalar type defined above, except for floats, doubles, and bytes.

Oneof

Oneof becomes very handy in terms of space optimization when you have many optional fields in a message and know that at most only one will be set. The implementation of Oneof shares the same memory space for all the fields, so it doesn't need to reserve space for the other tags having empty fields. Of course with the language implementation, you should have a `WhichOneOf` method in order to determine what was the set value.

```
    syntax = "proto2";

    message Film {
        message Character {
```

```
        oneof has {
            string laser_gun_model = 1;
            int32 laser_sword_power = 2;
        }
    }
}
```

In this example setting `laser_gun_model` or `laser_sword_power` allows you to know which type of weapon the `Character` has. Looking into the value in the example provides you with additional information (gun model as a string, or laser power as an integer).

Enums

Enums help you to spare a lot of bytes in serialized messages, specially when the strings sent through the network are repetitive, such us status codes, error messages, etc. The serialized value will contain only the tag of the field instead of the whole value.

```
syntax = "proto2";

message Film {
    enum ProducerCompanies {
        UNKNOWN = 0;
        WARNER_BROS = 1;
        PARAMOUNT = 2;
        NETFLIX = 3;
    }
    ProducerCompanies Producer = 1 [default = UNKNOWN];
}
```

Notice that the `UNKNOWN` enum type is highly recommended when the `Producer` set up in the code doesn't exist in the list. When it doesn't exist, it is automatically set to 0, to avoid future inconsistencies.

Services

The services implemented in gRPC are defined as RPCs (Remote Procedure Calls) in the `.proto` file.

RPC was designed for developers to be able to execute a procedure in their source code, without caring about its execution. You can focus your efforts on coding the procedure, without worrying about serialization, communication, or security.

A service example can be defined as follows:

```
syntax = "proto2";

service Starwars {

    rpc GetFilm(GetFilmRequest) returns (GetFilmResponse);

}
```

Thanks to this definition, the protocol buffer compiler will generate the service interface code and stubs (client and server side) suiting the chosen language. You can create your own RPC system, or you can use gRPC.

Importing other proto files

As mentioned earlier in the Nested Messages section, you can create messages that may be used inside other messages.

A flatter form of doing the same thing to avoid nesting to the infinity would be creating every single message on the root level. It would look this:

```
syntax = "proto2";

message Character {

    oneof has {
        string laser_gun_model = 1;
        int32 laser_sword_power = 2;
    }

}

message Film {

    repeated Character characters = 1;

}
```

For scaffolding reasons, or just because you want to make the code cleared, you could define `Character` in another file. You can import the `.proto` file as follows:

```
syntax = "proto2";

import "character.proto";

message Film {

    repeated Character characters = 1;

}
```

As you can imagine, you can't use imported `.proto` files that are not imported in your own `.proto` file. Cascade importing does not work, unless you use the `import public` notion:

```
syntax = "proto2";

import public "character.proto";
```

This notion is useful when the time comes to move `.proto` files locations, and you don't want to change every single file in the import file path. This would be the strategy:

- Move the `.proto` file to the new location.
- In the old location, leave an empty `.proto` file.
- The only mandatory line that needs to contain is an `import public` to the new location.

There won't be any regression, and legacy code will still work as always.

Repetitive fields go first

It is widely recommended that you prioritize the association of the first tag positions with the fields that are supposed to be recurrently filled.

As you will see in the Encoding section, the positions from 1 to 15 will only take one byte (instead of two or more) for pointing out the position in the binary object.

In conclusion, if you have a message with more than 16 fields, and some of them are optional, don't forget to place the optional ones at the end of the message.

Polymorphism

The idea of this section is to explain how inheritance may be implemented in order to enable polymorphism among messages. There are two methods, depending on the proto version that we are using:

- Proto 2 uses Extensions
- Proto 3 deprecates Extensions in favor of Any statement

Extensions are a range of tags reserved for future extended messages so you can use them for different purposes:

```
syntax = "proto2";

message Film {

    extensions 50 to 99;

}

message TerrorFilm {

    extend Film {

        required TerrorFilm film = 50;

    }

    bool has_zombies = 1;

}

message ScienceFictionFilm {

    extend Film {

        required ScienceFictionFilm film = 51;

    }

    bool has_jedis = 1;
}
```

In terms of binary serialization the final message will look the same as a non extended message, but in terms of code generated for accessing those fields, it's a different story. Here is an example for setting the extended field `film` in a Java implementation.

```
ScienceFictionFilm scienceFictionFilm =
createMyScienceFictionFilm();
Film film =
  Film.newBuilder()
    .setExtension(ScienceFictionFilm.film,
scienceFictionFilm)
    .build();
```

Any is the new approach for Proto 3. You can use any type without having to define it in the protobuf file.

```
syntax = "proto3";

import "google/protobuf/any.proto";

message Film {

    google.protobuf.Any film_type = 1;

}
```

To achieve this, every language has its own methods to work with. For instance, Java proposes the `pack` and `unpack` methods.

Packages

Packages may be defined in the `.proto` file in order to be imported from other files. Depending on the language, it will have different consequences.

- C++ `.proto` package becomes the namespace
- Java `.proto` package becomes `.java` package
- Python just ignores it, the only thing that counts is the folder structure

Versioning proto files

Any API is supposed to evolve and when a formal contract is defined between a client and a server, such as a `.proto` file intends to do, we need to be very careful when designing the structure of every message. Once a message is serialized, the only way to unmarshall the binary payload is to know the exact strategy used for constructing this bit sequence.

In the next sections we'll explain the approach of protobuf for serializing every type of data so that we'll fully understand the why and the how of the following tips:

- What happens with the deprecated fields? The tip is to create all fields as `optional`. Proto version 3 has adopted the optional statement by default when defining fields. This will avoid future deprecations to raise errors.
- What happens with the new fields? The tip is to add them always at the end. Replacing old tags should be forbidden, so the association of a field with its tag number should always remain the same.

Different approaches for generating proto classes

Let's examine the various languages for generating proto classes.

C++

```
protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/sfapi.proto
```

C

```
protoc -I=$SRC_DIR --csharp_out=$DST_DIR $SRC_DIR/sfapi.proto
```

GO

```
go get golang.org/x/sys/unix
go get -u github.com/golang/protobuf/protoc-gen-go

protoc -I=$SRC_DIR --go_out=$DST_DIR $SRC_DIR/sfapi.proto
```

Java

```
protoc -I=$SRC_DIR --java_out=$DST_DIR $SRC_DIR/sfapi.proto
```

Python

```
protoc -I=$SRC_DIR --python_out=$DST_DIR $SRC_DIR/sfapi.proto
```

NodeJS

On the fly class generation with protobuf.js:

```
const grpc = require('grpc')
const proto = grpc.load('./proto/sfapi.proto').sfapi.v1
```

And here are some advanced functionalities.

Option declarations

Option declarations may be declared to engage different behaviors when each language generates the code responsible for the interpretation of the `.proto` file.

Here you can find between the lines the full list of possible protobuf options. They are defined as optional fields with default values, and interpreted by each compiler.

Three kinds of options may be defined:

File options

1. *Package and namespaces*

```
    // Generated java classes will belong to com.starwars
package
    option java_package = "com.starwars";

    // Almost always necessary and very handy to avoid
conflicts when importing packages
    option go_package = "github.com/backstopmedia/gRPC-book-
example/com/starwars";

    option csharp_namespace = "Com.Starwars";

    option php_namespace = "com.starwars"
```

1. *Output class names*
   Some options are useful for forcing the naming of the classes for the generated code.

   ```
   option java_outer_classname = "StarWars";

   // You should use upper-cased prefixes (3-5 characters)
   as Apple recommends
   option objc_class_prefix = "SWS";

   option php_class_prefix = "SWS";
   ```

1. *Multiple files*
   The Java code generator will create different `.java` files for each root level message.

   ```
   option java_multiple_files = true;
   ```

2. *Compiler options*
   Only for C++ and Java code generators

   ```
   option optimize_for = SPEED;
   ```

   Possible options are:

   - SPEED (default): highly optimized generated code.
   - CODE_SIZE: it optimizes the final size of the generated code, by sharing classes and using reflection. The performances are worse, so the execution is slower.
   - LITE_RUNTIME: the generator uses libprotobuf-lite runtime instead of libprotobuf. The code generated will be much smaller and the execution much faster, but some features will be missing.

Field options

1. *Deprecation*
   The field is marked as deprecated, and every language will do what is most appropriate.

   ```
   optional int32 old_field = 1 [deprecated=true];
   ```

2. *Pack 'repeated' fields*

Packed fields allow you to optimize space once the `repeated` fields are serialized. It avoids sending the tag and the type of every value in the 'Array.'

```
optional repeated int32 field = 1 [packed=true];
```

3. *JavaScript types*
   For the `int64`, `uint64`, `sint64`, `fixed64`, and `sfixed64` scalar types you can tell the JavaScript code generator to use some other specific JavaScript type when decoding the fields. When the values are too high, there can be some precision problems with the JavaScript Number values.
   Possible options are:
   - JS_NORMAL (default): use the default type.
   - JS_STRING: use String type, so no precision is lost.
   - JS_NUMBER: use the Number JavaScript type, which may end up with some precision problem.

```
uint64 id = 1 [jstype = JS_STRING];
```

Message options

1. *Deprecation*
   The message is marked as deprecated, and every language will do the most appropriate thing.

```
optional int32 old_field = 1 [deprecated=true];
```

Custom options

We can define custom options in addition to those that are predefined. Note that custom options are based internally on `extensions` and are deprecated in version `proto3`. In order to still allow the functionality that `extensions` in `proto3` are still functional, only allow Custom options.

Here is an example:

```
import "google/protobuf/descriptor.proto";

extend google.protobuf.MessageOptions {
  // From 50000 to 99999 is reserved for custom options
  optional string max_actors_allowed = 50000;
}

extend google.protobuf.FileOptions {
  optional string my_file_option = 50001;
}

option (my_file_option) = "Custom File Option Example";

message Film {
  option (max_actors_allowed) = "100";
}
```

In order to collect these options, each language will have its own procedures.

```
String value = MyProtoFile.Film.getDescriptor().getOptions()
  .getExtension(MyProtoFile.maxActorsAllowed);
```

Understand encoding

In this section we'll explore the way in which protobuf encodes each type of data. The goal is to understand why protobuf is such a big win in terms of bandwidth optimization.

The first thing to realize is that we'll use different strategies with the data type that we want to encode. Once every field is binary encoded we'll need to concatenate each one of them in the order that we specified in the tags of the `.proto` file. That's how the receiver will be able to decode every message. In addition to the field encoding there are different strategies to delimit where the next field begins. This allows `.proto` files to evolve, adding more fields. If the receiver doesn't know the tag, it can be ignored and the receiver can move on to the next tag.

Here is a list of all the encoding type strategies:

Strategy	Name	Types
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
5	32-bit	fixed32, sfixed32, float

Varint

Varint is a strategy that consists of compressing integers into a smaller space, instead of sending the same fixed amount of bytes without taking into consideration the magnitude of the value.

For instance, let's say that we want to encode the number 227 in the following protobuf message:

```
message Test {
  int32 a = 1;
}

# JSON representation { "a": 227 }
```

The number 227 expressed as a binary:

```
127 => 1110 0011
```

We'll need to split the whole binary number into groups of 7 bits. In this case one byte will be split in two.

```
1110 0011 => 000 0001  110 0011
```

As you can see, there is a bit missing for every byte that we need to send. So we will use this bit to know if it is the end of the message. The last byte of the message will set this bit to 0.

```
000 0001  110 0011 => '0'000 0001  '1'110 0011
```

Finally, we reverse all of he bytes:

```
0000 0001  1110 0011 => 1110 0011  0000 0001
```

As you can see, with only two bytes we can encode the number 227, which in this case is an `int32`, so it is encoded with four bytes. Note that in this case, `int32` or `int64` would be protobuf encoded with the same size.

Once the value is encoded, we need to prepend two vital pieces of information:

- Tag number (1, in this case)
- Encoding strategy (0, in this case)

This is the method to encode the so called **wire type**:

```
# Left shift 3 bits the tag number of the the key:
0000 0001 => 0000 1000

# Use the three right bits for indicating the encoding
strategy code (0, 1, 2, 5)
# In this case 0
0000 1'000' => 0000 1000
```

Finally, what we get is:

```
0000 1000  1110 0011  0000 0001 => 08 E3 01
```

Varint for signed values

If we apply the same strategy for negative numbers (`int32` or `int64`) we'll never spare space because the left bits will be always 1, instead of 0. So for an `int32` we would always use 4 bytes, even for small magnitudes.

This is why it's highly recommended to set types to `sint32` or `sint64` when you know that numbers may be negative most of the time.

The strategy used will then be **ZigZag encoding**:

```
# For sint32:
(n << 1) ^ (n >> 31)

# For sint64:
(n << 1) ^ (n >> 63)
```

Or here is an easier way to understand the same thing:

```
if (n == 0) {
  return 0;
} else if (n > 0) {
  return 2 * n;
} else if (n < 0) {
  return 2 * Math.abs(n) - 1;
}
```

Non-Varint values (type 1 and 5)

The strategy of encoding the wire type with non-Varint values will be the same as before, and the value will be stored simply in little-endian byte order.

Length-delimited (type 2)

The strategy of encoding the wire type will be the same as before, followed by the length of the string, and the value of the string will be ASCII encoded.

If you want to encode 'test,' use the following message:

```
message Test {
  string a = 1;
}
```

You will obtain the following code:

```
# Wire type: 0A
# Length:    04
0A 04 74 65 73 74
```

Embedded messages

Embedded messages will be encoded as a length-delimited type:

```
message Test {
  int32 a = 1;
}

message Test1 {
  Test b = 2;
}

# JSON representation { "b": { "a": 227 } }
```

This should be represented as:

```
# Wire type: 1A
# Length:    03
1A 03 08 E3 01
```

Summary

In conclusion, choosing protobuf messages as the presentation layer makes gRPC much more powerful in every sense, compared to other alternatives (JSON, XML, CSV, ...).

Unlike the alternatives, protobuf allows you to be performant in all of the following criterias:

- Parsing efficiency
- Upgrading messages and adding new fields
- Ignoring/Deprecating fields
- Having a minimum payload depending on the data, not the structure
- Applying hierarchical data

gRPC basics 4

To get started with gRPC, you'll need to know how to do a few things:

1. Define an RPC interface: a service in a `.proto` file.
2. Create an implementation of that RPC interface.
3. Expose the service by creating a gRPC server and registering your service implementation with it.
4. Create a gRPC client and connect it to a server.
5. Create stubs that use the gRPC client.
6. Use the stub to send RPC requests to the server.
7. *Profit!*

In this chapter we will guide you through these exact steps. Along the way, we'll cover basic error handling in both the server and the client. And you'll also learn about gRPC metadata and how to use it.

Vocabulary

First thing's first: let's make sure we agree on the definitions for a few key terms that will come up throughout this chapter and even the rest of the book.

Client

When we say client, we are referring to the program that is responsible for establishing connections to a server and sending RPC requests to it. In gRPC, operations are always initiated by the client. Note that a server can also be a client: in microservice architectures, programs are very often both. A server, let's call it "server A," will receive an RPC request. In the course of processing the request, the server may need to send an RPC to another server, which we'll call "server B." In this case, "server A" is also acting as a client. With service-oriented architectures, it is not uncommon for numerous servers to be involved in the processing of a single operation.

The term “gRPC client” refers more specifically to the gRPC client library. This is the component inside the client program that handles establishing connections to servers, sending requests, and translating responses.

Server

When we say server, we are referring to a program that exposes one or more gRPC services via the gRPC protocol. It is the program that handles clients’ requests and replies to them.

The term “gRPC server,” or “server machinery,” refers to the gRPC server library. This is the component in the server that handles accepting socket connections, translating requests, invoking methods on the server’s service implementations, and sending responses.

Connection

The term connection is a virtual pipe that connects the client to the server. Depending on how the gRPC client is configured, it may be one physical socket, or it may be multiple sockets for load balancing across multiple servers.

In gRPC, connections always use the HTTP/2 protocol. The gRPC client is what establishes these connections for sending requests to servers.

Service

A service is an interface through which clients interact with servers. It is the contract, defining the set of operations and the type of data each operation requires.

In gRPC, the service is defined in `*.proto` source files. It is really just a name and a set of named methods.

Method

A method is a single operation that a client can invoke. When a client issues an RPC, they are invoking a method. Each method has a signature that is comprised of the method’s name, the type of data it accepts as input (the request), and the type that it returns (the response).

Stream

Streams will be covered in depth in a later chapter. But know that a stream is a sequence of zero or more messages. In gRPC, either the request or the response (or both) can be a stream. In this chapter, we’ll start with the simple case, where each request and response is a single message.

Stub

The stub is an object used by clients. It exposes a service's interfaces. Invoking a method on the stub hooks into the gRPC client to actually send a request to a server.

Stubs are usually generated by invoking `protoc`, though there may be some languages where the stubs are generated dynamically, at runtime.

In general, stubs are cheap, but connections are not. In a long-running program, connections are usually created at initialization time and then kept for the duration of the program because they are stateful and not as cheap to create. Stubs, on the other hand, are cheap to create. So it is less important to create stubs early and re-use them: it is okay to just create them as needed.

Service implementation

The service implementation contains the business logic that is executed when RPC methods are invoked. This contains all of the handlers that are used by the server to process RPCs. When a client invokes a particular method on its stub, the gRPC server library receives the request and in turn invokes the method of the same name on the service implementation.

Creating and Implementing Services

In the previous chapter, we learned a lot about Protocol Buffers. In particular, we saw what the interface definition language looks like, how to define messages and services, and how to use the `protoc` tool to generate code in our desired target language.

To create a service, we use what we learned in that chapter to write a `.proto` source file that defines the service and the request and response types it needs. Let's start small: here is a service with just two methods for retrieving information about movies from a database stored on the server. We'll create a file named `proto/sfapi.proto` that looks like so:

Protocol Buffer

```
syntax = "proto3";

// The "google/protobuf" import folder provides several
types that
// are included with protoc. This one is a representation of
// timestamps/dates.
import "google/protobuf/timestamp.proto";
```

```
option go_package = "proto";

service Starfriends {
  // Get a single Film by unique ID
  rpc GetFilm(GetFilmRequest) returns (GetFilmResponse);

  // Get a list of all Films
  rpc ListFilms(ListFilmsRequest) returns
(ListFilmsResponse);
}

message GetFilmRequest {
  string id = 1;
}

message GetFilmResponse {
  Film film = 1;
}

message ListFilmsRequest {
  // no arguments for listing films
}

message ListFilmsResponse {
  repeated Film films = 1;
}

message Film {
  string id = 1;
  string title = 2;
  string director = 3;
  string producer = 4;
  google.protobuf.Timestamp release_date = 5;
}
```

As you can see, method definitions in Protocol Buffers only allow a single argument type, which must be a message type. So to pass multiple arguments to a method, you must package them all up in a single request message that has one field for each argument. The same goes for the return type of a method: it must be a message.

What is not shown in the IDL is that methods can actually return a message of the defined type *or* an error code. We'll see what that looks like later in this chapter.

Compiling the Protocol

Now that we have created the service definition, we can compile this to our server language. The `protoc` tool compiles the `*.proto` source files into a target language. But, out of the box, it only performs code generation for message and enum types, not for services.

In order to generate code for services, we must use a `protoc` plugin. The gRPC documentation site has tutorials for each supported language (**https://grpc.io/docs/tutorials/**). By reviewing the tutorials for the language you are using, you will find how to use the gRPC plugin to generate gRPC-specific code for services.

We'll use Go for the server-side examples in this chapter. Unlike some of the other officially supported languages, generating Go code requires a plugin: `protoc-gen-go`. This plugin provides standard output, for messages and enums, and also provides gRPC output if you specify `plugins=grpc` in the `--go_out` argument when invoking `protoc`.

```
# first we install the protoc-gen-go plugin
go get github.com/golang/protobuf/protoc-gen-go
# now we can invoke protoc
protoc --go_out=plugins=grpc:. ./proto/sfapi.proto
```

The above command takes the proto source file and generates a corresponding `service.pb.go`. The `--go_out` parameter tells it where to write the generated code. In the above example, the destination is `.`, so the files are generated relative to the current directory. That means the generated file will actually be in the same directory where the proto source file resides. (The part after the `--go_out=` and before the following `:` will be parsed as plugin arguments by the Go plugin, which is how we activate the gRPC-specific code generation.)

This generated file contains Go type definitions for any messages and enums in the proto source. Each message results in a Go struct (`oneof` fields result in a little bit more Go code being generated). And each enum results in a Go named type whose underlying type is `int32`. Note that the `service.pb.go` file that is created by `protoc` should never be modified by hand -- any manual edits to the file will be undone if and when `protoc` is invoked to re-generate the file.

Because the invocation above specified `plugins=grpc` as a plugin argument, the output also has two interfaces for each service: one used by clients and one used by servers. (Streaming methods will result in a little bit more Go code being generated, too.) The two interfaces are easy to distinguish: their name is the name of the service plus a `Client` or `Server` suffix. In our example, this results in interfaces named `StarfriendsClient` and `StarfriendsServer`.

Creating a Service Implementation

The next step is to write Go code to provide an implementation of our newly-generated server interface. In Go, any type that provides methods with the same signatures as the interface implicitly implements that interface. So we just create a type that provides the necessary methods in a file named `service/impl.go`:

Go

```go
package service

import (
  "time"

  "github.com/golang/protobuf/ptypes"
  "github.com/golang/protobuf/ptypes/timestamp"
  "golang.org/x/net/context"
  "google.golang.org/grpc/codes"
  "google.golang.org/grpc/status"

  "../proto"
)

// To start with, we'll hardcode the database of films.
var films = []*proto.Film{
  &proto.Film{
    Id:          "4",
    Title:       "A New Hope",
    Director:    "George Lucas",
    Producer:    "Gary Kurtz, Rick McCallum",
    ReleaseDate: toProto(1977, 5, 25),
  },
  &proto.Film{
    Id:          "5",
    Title:       "The Empire Strikes Back",
    Director:    "Irvin Kershner",
    Producer:    "Gary Kurtz, Rick McCallum",
    ReleaseDate: toProto(1980, 5, 17),
  },
  &proto.Film{
    Id:          "6",
    Title:       "Return of the Jedi",
    Director:    "Richard Marquand",
    Producer:    "Howard G. Kazanjian, George Lucas, Rick
McCallum",
    ReleaseDate: toProto(1983, 5, 25),
  },
}

func toProto(year, month, day int) *timestamp.Timestamp {
  t := time.Date(year, time.Month(month), day, 0, 0, 0, 0,
time.Local)
  ts, err := ptypes.TimestampProto(t)
  if err != nil {
    panic(err)
  }
  return ts
}

type StarfriendsImpl struct {
}

// GetFilm queries a film by ID or returns an error if not
found.
func (s *StarfriendsImpl) GetFilm(ctx context.Context,
    req *proto.GetFilmRequest) (*proto.GetFilmResponse,
error) {

  var film *proto.Film
  for _, f := range films {
    if f.Id == req.Id {
      film = f
      break
    }
  }
  if film == nil {
    return nil, status.Errorf(codes.NotFound, "no film with
id %q", req.Id)
  }
  return &proto.GetFilmResponse{Film: film}, nil
}

// ListFilms returns a list of all known films.
func (s *StarfriendsImpl) ListFilms(ctx context.Context,
    req *proto.ListFilmsRequest) (*proto.ListFilmsResponse,
error) {

  return &proto.ListFilmsResponse{Films: films}, nil
}

// compile-type check that our new type provides the
// correct server interface
var _ proto.StarfriendsServer = (*StarfriendsImpl)(nil)
```

The above example shows a `*StarfriendsImpl` type that has one exported method for each of the methods we defined in the proto source for the `Starfriends` service.

The implementation is very simple; it uses a hard-coded list of films that serve as the server's database. Of particular interest is the case in `GetFilm` where the client has requested an unrecognized film ID. We use gRPC's `status` package to create the error. This allows us to communicate a gRPC error code to the client. Using other kinds of errors can result in the client seeing simply "Unknown" as the error code.

The last line is idiomatic Go to assert, at compile-time, that a type implements a particular interface.

MORE ABOUT GRPC CODES

In the above service implementation, we used the "Not Found" code. gRPC uses a custom list of 17 error codes. They are in no way similar to HTTP status codes. Their numeric values range from 0 (which means "OK", no error) to 16. The gRPC runtime libraries provide ways of returning error codes to clients, such as the `status` and `codes` packages that were used in the service implementation above.

Below is a list of the error codes.

Code	Numeric Value	Description
OK	0	The zero value is the code used for successful operations. It means that no error occured.
Cancelled	1	This code is an acknowledgement of a client cancellation. If a client chooses to cancel an outstanding RPC, this is the error code that the server should record. When used correctly, a client will never actually see this code: the RPC has been cancelled, so any error code will not be delivered.
Unknown	2	This is the default error code. If the server can provide no information about the nature of the error, then the cause is unknown. Note that "Internal" is often a better choice when an unexpected error occurs.

Code	Numeric Value	Description
Invalid Argument	3	This indicates that the request is malformed. When validating a request, if an argument is invalid and *could never be* valid, then this is the right code. If the argument is invalid due to the current state of the system, then “Failed Precondition” is the better choice.
Deadline Exceeded	4	This code means that the client’s requested deadline has not been met. Like the “Cancelled” code, this error code should not usually be observed by clients: after their deadline has passed, the RPC will be automatically cancelled, and the client will not see any error response. It is instead more useful on the server for classifying failures.
Not Found	5	This indicates that a request field refers to a resource that does not exist or instructs the server to query for information that does not exist.
Already Exists	6	This indicates that a request has requested creation of a resource that already exists. This is a specialization of the “Failed Precondition” code.
Permission Denied	7	This code means that the caller is not allowed to perform the operation. This should be due to authorization controls. If the caller cannot perform the operation due to a quota, use “Resource Exhausted” instead. If the caller is not allowed, but it might be if it were authenticated, use “Unauthenticated” instead.
Resource Exhausted	8	The server has exhausted some resource and cannot complete the operation. This could be due to quota limitations for the caller, but it could also indicate some other resource, like disk space, has been exhausted.
Failed Precondition	9	This code means the operation cannot be completed because some part of the system is not in the correct state. Validation that must query for a resource’s existing state is likely validating a precondition. If the state being checked is an optimistic concurrency checks, as part of a multi-operation sequence, “Aborted” is probably more appropriate. The caller typically must fix the system state before retrying.
Aborted	10	This code means that the operation was aborted, such as due to an optimistic concurrency failure. This often means that retrying a multi-operation sequence

Code	Numeric Value	Description
		may possibly correct the issue, but retrying just the one aborted step will not.
Out of Range	11	This is a specialization of "Failed Precondition". It means the precondition that failed was a bound check. It should be used in iteration operations to indicate that the end of the iteration has been reached.
Unimplemented	12	The request operation is not implemented. This may be used by server's that only expose part of a service, and not its full interface.
Internal	13	This code means an internal error has occurred in the server.
Unavailable	14	This code means that either the server or the requested operation is temporarily unavailable. The client may retry, with backoff, to see if it becomes available.
Data Loss	15	This is a specialization of "Internal" and further conveys that some sort of unrecoverable loss or corruption of data has occurred.
Unauthenticated	16	This code indicates the request operation requires authentication, but the request has either not included authentication credentials or has included invalid credentials.

Exposing services

Now that we have created a Protocol Buffer service, compiled it to Go, and created an implementation of the server interface, we are ready to spin up a server program that clients can use to invoke the logic in our service implementation.

The mechanism for registering service implementations differs slightly for each target language. The gRPC documentation site's tutorials provides examples for each of the officially supported languages.

For Go, in addition to the server interface, `protoc` also generates a registration method that servers use to expose service implementations. Each such method accepts a `*grpc.Server` and a service implementation. In a given server, only a single implementation of a particular service is allowed. (Attempts to register more than one implementation of the same service interface will result in a panic.)

So let's pull together this information by writing a Go program that will expose our service implementation. We'll create a file named `server.go` with the following contents:

Go

```
package main

import (
  "flag"
  "fmt"
  "net"
  "os"
  "os/signal"
  "syscall"

  "google.golang.org/grpc"

  "./proto"
  "./service"
)

func main() {
  port := flag.Int("port", 8080, "The port on which gRPC
server will listen")
  flag.Parse()

  // We're not providing TLS options, so server will use
plaintext.
  lis, err := net.Listen("tcp", fmt.Sprintf(":%d", *port))
  if err != nil {
    fail(err)
  }
  fmt.Printf("Listening on %v\n", lis.Addr())
  svr := grpc.NewServer()

  // register our service implementation
  proto.RegisterStarfriendsServer(svr,
&service.StarfriendsImpl{})

  // trap SIGINT / SIGTERM to exit cleanly
  c := make(chan os.Signal, 1)
  signal.Notify(c, syscall.SIGINT)
  signal.Notify(c, syscall.SIGTERM)
  go func() {
    <-c
    fmt.Println("Shutting down...")
    svr.GracefulStop()
  }()
```

```
    // finally, run the server
    if err := svr.Serve(lis); err != nil {
      fail(err)
    }
  }

  func fail(err error) {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
  }
```

The above code shows how a gRPC server is created in Go. The first thing needed is a listener that accepts client sockets to establish connections. After registering the service implementation, we then call the server's `Serve` method, which starts and runs the gRPC server. This method does not return until the server shuts down, and returns an error if the cause of shutdown was not normal termination.

Normal termination happens when another goroutine calls either the server's `GracefulStop` or `Stop` method. The former gives in-process RPCs a chance to complete, blocking until they finish. The latter cancels any in-process RPCs and stops the server immediately.

Before the call to `svr.Serve` we see a simple way to have the Go server shutdown when it gets a `SIGINT` or `SIGTERM` signal (such as a user pressing `Ctrl+C` to stop the program). This section uses the standard library's `signal` package to configure handlers for these signals. When the program receives one of these signals, a value will be delivered to the configured channel. So when our server sees anything arrive in the channel, it initiates shutdown.

You can now run the server:

```
# build it
go build -o server ./server.go
# and then run it
./server
```

Graceful shutdown and load balancing

If load balancing is used, it may be necessary to first remove a server from the pool of backends before calling `GracefulStop`. For example, many load balancers (or service discovery systems used to provide client-side load balancing) will issue health checks to each server to decide if it is available to accept a load. In that case, the signal handler that shuts down the server should first arrange for health checks to fail, like by setting a flag that the health check logic will examine. It should then wait a fixed amount of time, to make sure that a failed check is observed, before invoking `GracefulStop`. The amount of time to wait depends on the environment, such as how often health checks are issued and how many sequential failed checks are required before a backend is deemed unavailable.

Creating and using stubs

At this point in this chapter, we have a server that exposes a service implementation. Now we focus on the client side. Before a client can send RPCs to the server, it must first establish a connection to the server and then create a stub, whose methods are invoked to send RPCs to the server.

The mechanism for creating a gRPC client, establishing a connection to a server, and instantiating stubs differs from one target language to another. The gRPC documentation site's tutorials provides examples for each of the officially supported languages.

The examples below will show the steps for building a Ruby client. At the end, we'll have a Ruby client that is issuing RPCs to our Go server, demonstrating the interoperability across languages.

Before we start writing Ruby code, we'll first generate the Ruby code from our proto source file. Ruby code generation for messages and enums is built into a `protoc`, so no plugin is needed. We *will* need a plugin, however, to generate the gRPC-specific pieces. All gRPC plugins (other than for Go, Java, and Dart) are in the main gRPC repo, which is also where you will find installation instructions: **https://github.com/grpc/grpc/blob/master/INSTALL.md**.

After installing everything and building the gRPC repo from source, you can then use the gRPC Ruby plugin (aptly named `grpc_ruby_plugin`):

```
protoc --ruby_out=. --grpc_out=. \
    --plugin=protoc-gen-grpc="$(which grpc_ruby_plugin)" \
    ./proto/sfapi.proto
```

The `--ruby_out` parameter triggers the standard Ruby code generation, which is for messages and enums defined in the proto source. The `--grpc_out` parameter then triggers gRPC-specific code generation for

Ruby. (It generates Ruby, not some other language, because of the `--plugin` argument, telling it to invoke `grpc_ruby_plugin`).

Like in the earlier example, when generating Go code, the arguments to these `--*_out` parameters is `.`, which tells `protoc` to generate code relative to the current directory. This will end up generating Ruby source files in the same location as the proto source, in the `./proto` directory.

Dialing the server

Now that we have Ruby code for our gRPC service, the first thing our client code must do is to connect to our running server. Only after a connection is established can we start sending RPCs to the server.

In Ruby, creating the connection and the stub is all done in one step. It is important to note that this means that stubs are *not* cheap to construct in Ruby as they are in other languages such as Go or Java, because they are coupled with a stateful connection to the server.

Below is an example that constructs the stub by dialing the server, which has defaulted to listening on port 8080:

Ruby

```
$LOAD_PATH.unshift File.expand_path("../", __FILE__)
require_relative './proto/sfapi_pb'
require_relative './proto/sfapi_services_pb'

stub = Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)

# Now we can use the stub to make RPCs
# ...
```

Here's an equivalent program, but written in Go, to demonstrate how the connection is first setup, and then stubs are cheap wrappers around the connection:

Go

```
package main

import (
  "google.golang.org/grpc"

  "./proto"
)
```

```
func main() {
  // First we create the connection:
  conn, err := grpc.Dial("localhost:8080",
grpc.WithInsecure())
  if err != nil {
    panic(err)
  }

  // We can now create stubs that wrap conn:
  stub := proto.NewStarfriendsClient(conn)

  // Now we can use the stub to make RPCs
  // ...
}
```

Here you can see the two steps. The `conn` object is a stateful connection to the server, and `stub` is a cheap wrapper that can be used to issue RPCs. The `NewStarfriendsClient` factory method was created by `protoc`. Such a factory is generated for each service in the proto source.

Working with stubs

Now that you have a stub, you can invoke methods that result in sending RPCs to the Go server:

Ruby

```
# First we create the request message
request = GetFilmRequest.new(id: '4')
# Now we can use stub (created in the Ruby example above)
# to send an RPC to the server and get back the response
response = stub.get_film(request)

puts response.inspect
```

The above snippet issues the RPC and prints the response, which has the details for the movie with ID “4”. The output looks like so:

```
<GetFilmResponse: film: <Film: id: "4", title: "A New Hope",
director: "George Lucas",
producer: "Gary Kurtz, Rick McCallum", release_date:
<Google::Protobuf::Timestamp:
seconds: 233380800, nanos: 0>>>
```

One interesting thing to note is that `protoc` converted the name of the method, which was defined as `GetFilm` in the proto source, to idiomatic naming conventions for Ruby methods: `get_film`. It does this for all languages, translating names in the IDL into names that should feel "at home" in the language of the generated code.

The stub has one method for each method defined in the service. The kind of request each method expects is the same as defined in the proto IDL, and the kind of message that is returned also matches the response type defined in the proto IDL.

Errors

The above examples show how to handle a successful RPC when the client successfully receives a response from the server. But what happens if our request is invalid? Or if there is a bug in the server that results in an error?

Earlier in the chapter, when we were creating the Go server, we introduced the gRPC error codes: sixteen error codes that servers can use when categorizing an error. The Go server has a simple example: if the client requests an unknown film ID, it sends back a `Not Found` error code.

In RPC clients, such an error will manifest as an exception:

Ruby

```ruby
# Our server is hard-coded to know about IDs 4-6. It does
# not know about ID 7.
request = GetFilmRequest.new(id: '7')
begin
  response = stub.get_film(request)
  puts response.inspect
rescue GRPC::BadStatus => e
  # RPC exceptions have a numeric 'code' field (the gRPC
error code)
  # and a string 'details' (the message supplied by the
server)
  puts "failed: #{e.details} (code = #{e.code})"
end
```

The `GRPC::BadStatus` is the super-class for RPC errors in Ruby. There is a sub-class for each gRPC error code, such as `GRPC::Cancelled`, `GRPC::NotFound`, and `GRPC::Unavailable`. These sub-classes can be used to handle specific gRPC error codes instead of generically handling any error like in the above example.

Any kind of error that could happen will be translated to an RPC error, not just those where the server has explicitly returned an error code. For example, If there are connection errors, `GRPC::Unavailable` will be raised.

Since RPCs involve network operations, failures are going to happen. So client code should be written to appropriately expect and handle these failures.

Metadata

In addition to exchanging request and response messages between the client and server, gRPC also allows for exchanging metadata. Metadata are key+value pairs, where both the key and value are strings. In fact, a given key may have multiple values. It is directly analogous to request and response headers in HTTP. In fact, gRPC uses headers (and trailers) in the HTTP/2 protocol in order to convey metadata from client to server and back.

Metadata is often used for cross--cutting concerns--cases where you might otherwise have to embed multiple extra request fields *into every single request message*. Examples include authentication credentials, IDs to support distributed tracing, or even client and server properties that could be useful for debugging (such as the language and version they are running).

Since HTTP/2 headers are used to transmit metadata, there are some metadata keys that cannot be used: they are reserved by gRPC for implementing other aspects of the protocol. Like HTTP 1.1, some headers control how clients and servers interpret the request and response data and thus cannot be set by application code. The most obvious example of this is the “Content-Type” header, which gRPC handles since it controls how content is encoded on the wire. The rest of these reserved headers are listed below:

- *HTTP/2 Reserved Headers*: The HTTP/2 protocol reserves any header names that start with a colon “:”. So applications may not use metadata keys that start with a colon.
- **Content-Type**: gRPC sets the content-type for all RPCs to `application/grpc` (with an optional suffix to indicate sub-content-type, such as “+proto” or “+json”). So applications may not use this as the name of their own metadata.

- **TE**: gRPC sets the transfer encodings that it can accept, which always includes “trailers” since that is how trailing metadata and status codes are sent.
- *gRPC Reserved Headers*: gRPC further reserves all headers that start with “grpc-” for internal use. For example, the gRPC status code and error message are conveyed to clients using “grpc-status” and “grpc-message” response trailers. And deadline propagation uses a “grpc-timeout” request header.

Metadata keys that do not match any of the above rules are fair game.

It is worth noting that all metadata keys are converted to lower-case when sent on the network. This is to canonicalize the names to ensure that header compression is effective. This is important to keep in mind since, in gRPC libraries, it may be necessary to query metadata maps using all lower-case keys. (This is required in Go, for example.)

One other important rule to note for naming metadata keys is that any metadata key that ends with “-bin” is assumed to be binary data. On the wire, the value will be base64-encoded (since arbitrary binary data is not allowed in HTTP/2 header values). Most gRPC runtime libraries handle transparently base64-encoding and decoding these binary metadata values, so it should not be a concern for application code.

The examples below show code that can be used inside of an RPC handler, but because metadata is often related to cross-cutting concerns, they are usually handled by interceptors, which run for every RPC request. Interceptors are discussed in detail in the chapter on advanced gRPC. This is usually a better place for inspecting metadata to make policy decisions as well as for adding metadata to every call.

Metadata on the server

In Go, server code can access the request metadata supplied by a client via the `context.Context` argument:

Go

```go
if reqHdrs, ok := metadata.FromIncomingContext(ctx); ok {
  // Must use all-lower-case keys to query metadata
  if who, ok := reqHdrs["who"]; ok {
    // who is a slice of strings; just use the first
    log.Printf("Received request from %s", who[0])
  }
}
```

As seen above, the gRPC `metadata` package provides functions for extracting **incoming** request headers from the context. The package also provides functions for outgoing request headers, but those are for use in clients written in Go.

This same `metadata` package also has factory functions for creating metadata values, either from a sequence of alternating key and value pairs or from a map (string keys and string values). These are used by the server to create response metadata. Also note that response metadata is broken up into two categories: *headers*, which are sent before the response message data, and *trailers*, which are sent at the end (along with the gRPC status code for the operation).

Go

```go
respHdrs := metadata.New(map[string]string{
  "Who":     "starfriends-server",
  "Version": "v1",
})
grpc.SendHeader(ctx, respHdrs)

start := time.Now()
defer func() {
  respTrlrs := metadata.Pairs("duration",
time.Since(start).String())
  grpc.SetTrailer(ctx, respTrlrs)
}()

// ... Do some processing and return response ...
```

In this example, you can see the use of `grpc.SendHeader` and `grpc.SetTrailer` to actually set the response metadata. The context argument must be the same as the one passed to the service implementation (or a child context thereof). The context has a value tucked away that provides access to the underlying transport, which is how the headers and trailers are actually sent to the client. If the wrong context is used, these calls will return an error.

In addition to these two methods, there is also a `grpc.SetHeader` function, which sets header metadata but does not send it to the client yet. It will be combined with metadata from other calls to `grpc.SetHeader` and with any metadata provided in a call to `grpc.SendHeader`. The call to `SendHeader` must be the last: subsequent calls to set or send headers will return an error since the headers have already been sent.

Similarly, it is okay to call `grpc.SetTrailer` multiple times. All of the metadata provided to multiple calls are combined. The trailers are actually sent when the server handler function returns.

The example above shows both ways of creating response metadata: using `metadata.New` and using `metadata.Pairs`. Both of these mechanisms will properly canonicalize the metadata keys by converting them to lower-case. It is not advised to use map literals or Go's type conversion to cast maps to the `metadata.MD` type since that means the keys may not be properly canonicalized.

Note that with the `metadata.Pairs`, you can specify the same key multiple times to provide multiple values. But that cannot be done with `metadata.New`, which accepts a map that has a single string for each key. Though `New` accepts a `map[string]string`, the actual underlying type of `metadata.MD` is `map[string][]string`.

Metadata on the client

Let's move on to view the other side of the transaction: how to send request metadata from the client and how to receive the server's header and trailer response metadata.

Request metadata is sent by supplying a keyword argument named `metadata` whose value is a hash:

Ruby

```
response = stub.get_film(request,
    metadata: {'who' => 'starfriends-ruby-client', 'version'
=> 'v1'})
```

The example supplies strings for the value of the hash. But it is also allowed to supply an array of strings for keys that have multiple values.

Receiving response metadata is a little trickier: we instruct the stub to return a handle for the RPC, called an `Operation`, instead of returning the actual response message. At this point, no RPC has been made. You must then call the operation's `execute` method to actually initiate the RPC:

Ruby

```
op = stub.get_film(request, return_op: true)
response = op.execute
```

In this example, you first get an operation for the call and then execute it. The `execute` method blocks until the server sends back a response. It returns the response message or raises an error, just as if we had invoked the stub without the `return_op` argument.

It is important to realize that just invoking the stub this way, with the `return_op` argument, *does not send any request to the server*. You must call the operation's `execute` method to actually make the call.

Now that you have an operation, after the call completes, you can query the operation for the server's response metadata:

Ruby

```
puts 'server provided headers: #{op.metadata}'
puts 'server provided trailers: #{op.trailing_metadata}'
```

These two properties, `metadata` and `trailing_metadata`, both return hashes. For each key, if there is exactly one value present, the value in the hash will be a string. But if the key has more than one value, then the value in the hash will be an array.

Summary

In this chapter, you learned the basics for building and using a gRPC service. You created a service interface using Protocol Buffers. You implemented a simple service in Go and wrote a client program in Ruby. You saw how errors can be handled in both the client and the server, and you learned how to send and receive metadata.

Now that you've seen the basics, you'll next see the server code fleshed out in the form of an example application, "Star Friends," that we refer to in other parts of the book. After that, you'll move into more advanced topics, starting with streaming.

Star Friends 5

It's useful to start out with a simple example that includes the basics, so we can build a toolbox of concepts. In this chapter, you will learn how to design your gRPC application, and think in terms of the simplest building-blocks used to construct your API. This will be the basis that other chapters will build on, as you learn gRPC.

The example application is a simple cross-referenced set of models based on the **SWAPI**, with all of the data changed to be about "Star Friends" our fictitious set of movies about space travel and adventure. It includes info about Planets, Starships, Vehicles, People, Films and Species from all of these imaginary movies.

The demo github repo is at **https://github.com/backstopmedia/gRPC-book-example**.

If you want to follow along, clone it like this:

```
git clone https://github.com/backstopmedia/gRPC-book-
example.git
```

Before writing code

Before starting an application, it's a good idea to describe the models and interface you think you might use. Before writing any code, think "What are the nouns and verbs in the story my API tells?"

Nouns

In any story, nouns describe the things that exist and can be acted upon. All of our application's nouns are inter-connected with many-to-many relationships.

```
<table>
  <thead>
    <tr>
      <th>Noun</th>
      <th>Relationships</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>each <code>Film</code></td>
      <td>
        <ul>
          <li>has some characters (<code>Person</code>)</li>
          <li>is set on some planets (<code>Planet</code>)</
li>
          <li>has some starships (<code>Starship</code>)</li>
          <li>has some other vehicles (<code>Vehicle</
code>)</li>
          <li>features different species (<code>Species</
code>)</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td>each <code>Planet</code></td>
      <td>
        <ul>
          <li>has residents (<code>Person</code>)</li>
          <li>is featured in some movies (<code>Film</
code>)</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td>each <code>Starship</code></td>
      <td>
        <ul>
          <li>has pilots (<code>Person</code>)</li>
          <li>is featured in some movies (<code>Film</
code>)</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td>each <code>Person</code></td>
      <td>
        <ul>
          <li>is featured in some movies (<code>Film</
code>)</li>
```

```
          <li>is some species (<code>Species</code>)</li>
          <li>might pilot some vehicles (<code>Vehicle</
code>)</li>
          <li>might pilot some starships (<code>Starship</
code>)</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td>each <code>Species</code></td>
      <td>
        <ul>
          <li>is featured in some movies (<code>Film</
code>)</li>
          <li>has some characters in it (<code>Person</
code>)</li>
        </ul>
      </td>
    </tr>
    <tr>
      <td>each <code>Vehicle</code></td>
      <td>
        <ul>
          <li>has pilots (<code>Person</code>)</li>
          <li>is featured in some movies (<code>Film</
code>)</li>
        </ul>
      </td>
    </tr>
  </tbody>
</table>
```

Verbs

In any story, verbs describe the things that happen. In this application, we only need a `List` and `Get` operation for our nouns, at least for now. There are also two contrived examples for showing off streaming and custom-errors (`ListStarshipActions` and `ValidateSpecies`, respectively).

How the service works

All of our noun-data comes from `data.json`, and is looked up on request.

Getting the data

The **SWAPI** has a lot of great data. We need to collect it, and clean up some data-modeling issues (e.g., numbers described as strings, etc.) and validate that we have it all in the expected format.

tools/get_data.js collects all the data and fixes it to have a neater shape. It's a collector & data-massager, thrown together to ensure everything is in a decent initial format. It was built through iterative trial-and-error, inspecting the generated data until it looked clean and well-organized. Cross-referenced string IDs were chosen instead of original `int` fields, to make them more unique, across all of the types.

tools/gen_proto.js generates an initial proto for the data, so we don't have to do it by hand. After this, the JSON data was validated against it's proto definition in **tools/validate.js**. This script shouldn't be used again, because after this point our team was using hand-tuned data & proto in the rest of the book. This highlights one of the strengths of an organization-wide living spec, in that it can be quickly changed and iterated on, until the team agrees on a spec, and everyone's code should more or less work well with it, without too much of a fuss.

Defining the Service

Let's start with the header.

The proto-definition starts with this:

```
syntax = "proto3";

import "google/api/annotations.proto";

package sfapi.v1;
option go_package = "proto";
```

This clearly marks what version of proto we are using (proto3, so all fields are `optional`) and prepares the loader for HTTP annotations (used by grpc-gateway, among others) and gives our API a namespace `sfapi.v1`.

Nouns

We start with a proto description of or nouns. Here is an example for `Film`:

```
message Film {
  string title = 1;
  int32 episode_id = 2;
  string opening_crawl = 3;
  string director = 4;
  string producer = 5;
  string release_date = 6;
  repeated string character_ids = 7; // Person.id
  repeated string planet_ids = 8;    // Planet.id
  repeated string starship_ids = 9;  // Starship.id
  repeated string vehicle_ids = 10;  // Vehicle.id
  repeated string species_ids = 11;  // Species.id
  string id = 12;
}
```

Verbs

Now, we need our `List` and `Get` for every noun. Endpoint-specific input messages are used for each, so that they can be expanded upon later. Here is an example, again, for `Film`:

```
// Get a single Film
rpc GetFilm(GetFilmRequest) returns (GetFilmResponse) {
  option (google.api.http) = {
    get: "/sfapi/v1/films/{id}"
  };
}

// Get a list of Films
rpc ListFilms(ListFilmsRequest) returns (ListFilmsResponse) {
  option (google.api.http) = {
    get: "/sfapi/v1/films"
  };
}

message GetFilmRequest {
  string id = 1;
}

message GetFilmResponse {
  Film film = 1;
}

message ListFilmsRequest {
}

message ListFilmsResponse {
  repeated Film films = 1;
}
```

The google.api.http annotation comes from google/api/annotations.proto and tells your gateway (if you are using one) how to forward the RPC call.

Server Implmentation

We'll start with JavaScript. Follow along in the example repo here: **https://github.com/backstopmedia/gRPC-book-example/blob/master/examples/nodejs/server.js**. We need some generic List and some Get functions:

JavaScript

```
// generic handler for Get* RPCs
const getHandler = (id, recordName, dataName) => {
  const record = data[dataName].find(r => r.id === id)
  if (record) {
    return {[recordName]: record}
  } else {
    return Promise.reject(new NotFoundError(`That $
{recordName} was not found.`))
  }
}

// generic handler for most List* RPCs
const listHandler = (recordName) => ({[recordName]:
data[recordName]})
```

Now, we'll setup a demo event-emitter for fake Starship events:

```
const starshipTracker = emitonoff()

// fake ship events are happening!
setInterval(() => {
  const action = StarshipActions[ (StarshipActions.length *
Math.random()) | 0 ]
  const starship = data.starships[ (data.starships.length *
Math.random()) | 0 ]
  starshipTracker.emit('StarshipAction', {action, starship})
}, 1000)
```

Every second, this picks a random `StarshipAction action` and `starship`, then emits it on `starshipTracker`, which is an event-pipe that can be subscribed to, later.

Now, we just need to glue all of the RPC functions to their associated generic function, add a `ValidateSpecies` demo function to illustrate error-handling, and add a subscriber called `ListStarshipActions` for streaming-events:

JavaScript

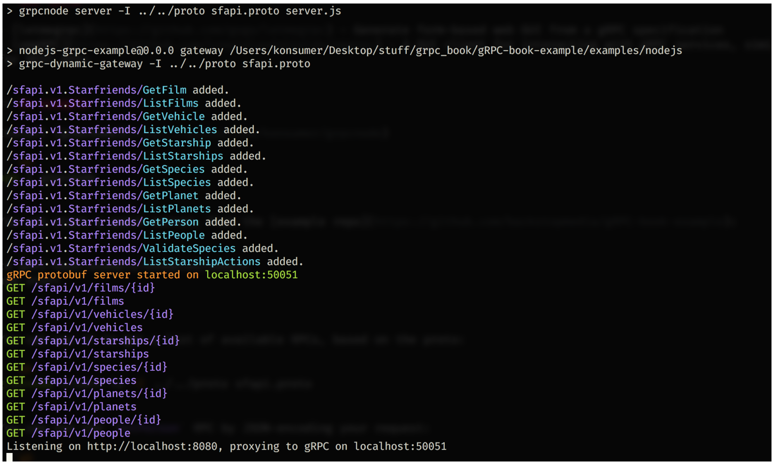
```
module.exports = {
  sfapi: {
    v1: {
      Starfriends: {
        GetFilm: ({ request: { id } }) => getHandler(id,
'film', 'films'),

        ListFilms: () => listHandler('films'),

        GetVehicle: ({ request: { id } }) => getHandler(id,
'vehicle', 'vehicles'),

        ListVehicles: () => listHandler('vehicles'),

        GetStarship: ({ request: { id } }) => getHandler(id,
'starship', 'starships'),

        ListStarships: () => listHandler('starships'),

        GetSpecies: ({ request: { id } }) => getHandler(id,
'species', 'species'),

        ListSpecies: () => listHandler('species'),

        GetPlanet: ({ request: { id } }) => getHandler(id,
'planet', 'planets'),

        ListPlanets: () => listHandler('planets'),

        GetPerson: ({ request: { id } }) => getHandler(id,
'person', 'people'),

        ListPeople: () => listHandler('people'),

        ValidateSpecies: ({ request: { name } }) => {
          if (name.length < 2) {
            return Promise.reject(new
InvalidArgumentError('The name provided is not long enough'))
```

```
            }
            return {}
          },

          ListStarshipActions: (call) => {
            const handler = ev => call.write(ev)
            starshipTracker.on('StarshipAction', handler)
            call.on('cancelled', () => {
              starshipTracker.off('StarshipAction', handler)
              call.end()
            })
          }
        }
      }
    }
  }
```

All of the `Get`/`List` business should be self-explanatory. `ValidateSpecies` is just to illustrate errors, and `ListStarshipActions` is a streaming RPC, which will listen to `StarshipAction` events on `starshipTracker` as long as the user is connected to it (and on disconnect, will uncouple `handler`.)

Starting it up

In the `examples/nodejs/` directory, you will need to ensure that the dependencies are setup with `npm i`, then run `npm start` to start your local server & HTTP1/JSON gateway.



```
> grpcnode server -I ../../proto sfapi.proto server.js

> nodejs-grpc-example@0.0.0 gateway /Users/konsumer/Desktop/stuff/grpc_book/gRPC-book-example/examples/nodejs
> grpc-dynamic-gateway -I ../../proto sfapi.proto

/sfapi.v1.Starfriends/GetFilm added.
/sfapi.v1.Starfriends/ListFilms added.
/sfapi.v1.Starfriends/GetVehicle added.
/sfapi.v1.Starfriends/ListVehicles added.
/sfapi.v1.Starfriends/GetStarship added.
/sfapi.v1.Starfriends/ListStarships added.
/sfapi.v1.Starfriends/GetSpecies added.
/sfapi.v1.Starfriends/ListSpecies added.
/sfapi.v1.Starfriends/GetPlanet added.
/sfapi.v1.Starfriends/ListPlanets added.
/sfapi.v1.Starfriends/GetPerson added.
/sfapi.v1.Starfriends/ListPeople added.
/sfapi.v1.Starfriends/ValidateSpecies added.
/sfapi.v1.Starfriends/ListStarshipActions added.
gRPC protobuf server started on localhost:50051
GET /sfapi/v1/films/{id}
GET /sfapi/v1/films
GET /sfapi/v1/vehicles/{id}
GET /sfapi/v1/vehicles
GET /sfapi/v1/starships/{id}
GET /sfapi/v1/starships
GET /sfapi/v1/species/{id}
GET /sfapi/v1/species
GET /sfapi/v1/planets/{id}
GET /sfapi/v1/planets
GET /sfapi/v1/people/{id}
GET /sfapi/v1/people
Listening on http://localhost:8080, proxying to gRPC on localhost:50051
```

Summary

Now that you know how the example application works, feel free to modify it and use your understanding in the chapters that follow to learn more advanced techniques. Next up is a in-depth look at streaming.

Streaming gRPC calls 6

So far we've been looking at unary RPC calls, which are calls that involve the client sending a single request to the server and the server responding with a single response. There are a lot of scenarios where this is sufficient. When you want to send large amounts of data between applications, however, you might not want to wait for the entire payload to be generated before either sending or receiving.

gRPC supports three methods of streaming that can help out in these cases:

- Server streaming for streaming records from the server to the client
- Client streaming for streaming records from the client to the server
- Bidirectional streaming, which involves the client and the server sending streams to each other.

Let's take a look at each streaming type in a little more detail. For each one there's an example application. If you'd rather jump straight to the code, you can view the source files at **https://github.com/backstop-media/gRPC-book-example/tree/master/chapters/streaming**.

Server streaming

The first method of streaming we'll look at is server streaming. Server streaming is when the client sends a single request to the server, and the server responds with zero or more messages as part of a single RPC response. A typical scenario might be performing a query that can return many results.

To illustrate this, let's look at an example of a service that defines a method for searching a database. Clients will send in a search term and an optional maximum number of results to be returned.

We start by making a new folder in *$GOPATH/src/practical_grpc* called *streaming*. We'll need a few directories to be created first, since `protoc` won't create them for you.

You can run the following to get started:

```
mkdir -p $GOPATH/src/practical_grpc/streaming
cd $GOPATH/src/practical_grpc/streaming
mkdir -p ./{proto,server/proto,database/proto}
```

Creating the protobuf file

Create a new file at *proto/database.proto* with the following content.

Protocol Buffer

```
syntax = "proto3";

package practical.grpc.v1;

service Database {
  rpc Search(SearchRequest) returns (stream SearchResponse);
}

message SearchRequest {
  string term       = 1;
  int64 max_results = 2;
}

message SearchResponse {
  string matched_term = 1;
  string content      = 2;
  int32 rank          = 3;
}
```

This service defines a single RPC that will receive a `SearchRequest` message and stream back a set of `SearchResponse` messages to the client. Each response will contain the term that was matched, the rank for the result, and the content that was matched.

Implementing the server

Before we can implement the server, we'll need to generate some code from the protobuf. To do this, we'll use a Docker container so we don't need to worry about having `protoc` installed with all of the necessary plugins:

```
docker run --rm -v $(pwd):/defs \
  namely/protoc-all:1.9 -d proto -l go -o server/proto
```

This will create a new file at *server/protos/database.pb.go* that contains all of the code we'll need to implement the server.

Let's create our server implementation. Create a new file at *server/database.go* with the following content:

Go

```go
package main

import (
    "log"

    pb "practical_grpc/streaming/server/proto"
)

// DatabaseService is an implementation of the Database
service in database.proto
type DatabaseService struct{}

// Search returns a stream of matching search results
func (db *DatabaseService) Search(r *pb.SearchRequest, s
pb.Database_SearchServer) error {
    responses := []string{
        "Highest ranked content",
        "Some ranked content",
        "Some ranked content",
        "Lowest ranked content",
    }

    for idx, resp := range responses {
        result := &pb.SearchResponse{MatchedTerm: r.Term,
Rank: int32(idx + 1), Content: resp}

        if err := s.Send(result); err != nil {
            log.Printf("Error sending message to the client:
%v", err)
            return err
        }
    }

    return nil
}
```

The implementation of the Search method would normally fetch matches from a database somewhere, but to keep the demo simple we've just used a static set of strings.

Each result is sent back to the client using the provided Database_SearchServer interface. This is automatically generated for you and made available by the runtime. Calling Send will send the response back to the client. You need to be careful to check the err that might be returned. This can happen if the underlying connection is broken or if the client decides to terminate the request due to a missed deadline or some other client specified condition.

Finally, returning nil indicates that the server has sent all of the messages.

Implementing the client

Now that we have the server defined, let's create a Ruby client to test drive it. As we did for the server, we'll need to have protoc generate the necessary client stubs before we get started.

```
docker run --rm -v $(pwd):/defs \
  namely/protoc-all:1.9 -f proto/database.proto -l ruby -o
database
```

This will create two new files in *data/proto*, *database_pb.rb*, and *database_services_pb.rb*. These files define the messages and stubs we'll need to call our service.

Let's create the client code in *database/client.rb*.

Ruby

```ruby
# frozen_string_literal: true

$LOAD_PATH.unshift('.')
require 'proto/database_services_pb'

# Adding some aliases to make the code easier to read
Stub    = Practical::Grpc::V1::Database::Stub
Request = Practical::Grpc::V1::SearchRequest

# Get the search term from CLI args
term = ARGV.first

begin
  stub    = Stub.new('localhost:
```

```
8080', :this_channel_is_insecure)
  request = Request.new(term: term, max_results: 10)

  stub.search(request).each do |response|
    puts "Term: #{response.matched_term}"
    puts "Rank: #{response.rank}"
    puts "Content: #{response.content}"
    puts
  end
rescue StandardError => e
  code = e.respond_to?(:code) ? e.code : 'Unknown'
  puts "Code: #{code}, Type: '#{e.class}', Message:
#{e.message}"
end
```

Notice the `unshift` call to the load path. This ensures that code defined under this path will be loaded before any lib or bundle code. This is necessary due to the way that protoc generates the Ruby stubs. If you look in *database/proto/database_services_pb.rb* you'll see it has `require 'proto/database_pb.rb'` in there.

Running the example

Now that we have the server and clients implemented, we need to add a small executable that will run the server.

Create a new file at *server/main.go* with the following content:

Go

```go
package main

import (
    "google.golang.org/grpc"

    "log"
    "net"

    pb "practical_grpc/streaming/server/proto"
)

func main() {
    s := grpc.NewServer()
    pb.RegisterDatabaseServer(s, new(DatabaseService))

    log.Print("Starting RPC server on port 8080...")
    list, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Fatalf("failed to setup tcp listener: %v", err)
    }

    if err := s.Serve(list); err != nil {
        log.Fatalf("Failed to start server: %v", err)
    }
}
```

This will start the gRPC server on `localhost:8080`.

Now, let's try out our new service. We'll need two terminal windows open. One for running the server and another for running the client.

In the first terminal, let's build and start the server.

```
go build -o server ./server
./server
```

You should see a message like: `2018/04/14 16:27:55 Starting RPC server on port 8080...`. This means the server is running and waiting for connections.

In the second terminal, we'll run the client to generate a request to the service.

```
cd database/
ruby client.rb searchTerm
```

The output should look like this:

```
Term: term
Rank: 1
Content: Highest ranked content

Term: term
Rank: 2
Content: Some ranked content

Term: term
Rank: 3
Content: Some ranked content

Term: term
Rank: 4
Content: Lowest ranked content
```

Client streaming

Client streaming is the opposite of server streaming. In client streaming RPCs, the client sends multiple messages to the server, which in turn sends back a single response. The response is typically returned after receiving all of the client messages, but that isn't a hard requirement.

To get a better understanding of client streaming, let's take a look at another example application. This time we'll create a service that accepts a set of files. Once the client has sent all of its files, the service will return a zip file containing all of the files that were received.

Creating the protobuf file

As we did in the server streaming example, let's start with the protobuf definition in proto/archiver.proto.

Protocol Buffer

```
syntax = "proto3";

package practical.grpc.v1;

service Archiver {
  rpc Zip(stream ZipRequest) returns (ZipResponse);
}

message ZipRequest {
  string file_name = 1;
  bytes contents   = 2;
}

message ZipResponse {
  bytes zipped_contents = 1;
}
```

Notice that the RPC accepts a *stream* of `ZipRequest` objects. Each of these represents a single file that will be included in the final zip archiver returned by the server.

Implementing the server

We'll need to generate some code again in order to implement the server. Run the following to create *server/proto/archiver.pb.go*:

```
docker run --rm -v $(pwd):/defs \
  namely/protoc-all:1.9 -d proto -l go -o server/proto
```

Now we can implement the server in *server/archiver.go*:

Go

```
package main

import (
    "archive/zip"
    "bytes"
    "io"
    "log"

    pb "practical_grpc/streaming/server/proto"
)

// ArchiverService is an implementation of the Archiver
service in archiver.proto
type ArchiverService struct{}

// Zip generates a zip file from the streamed request
messages
func (as *ArchiverService) Zip(stream pb.Archiver_ZipServer)
error {
    buf := new(bytes.Buffer)
    zf := zip.NewWriter(buf)

    for {
        // get or wait for the next request object in the
stream
        req, err := stream.Recv()

        // we're done, send the zip file
        if err == io.EOF {
            if err = zf.Close(); err != nil {
                log.Printf("Error creating the zip file:
%v", err)
                return err
            }

            return stream.SendAndClose(&pb.ZipResponse{
                ZippedContents: buf.Bytes(),
            })
        }

        // an error occured when getting the request object
        if err != nil {
            log.Printf("Error reading the request: %v", err)
            return err
        }

        f, err := zf.Create(req.FileName)
        if err != nil {
            log.Printf("Error creating the zip file entry:
%v", err)
            return err
        }

        if _, err := f.Write(req.Contents); err != nil {
            log.Printf("Error writing zip file: %v", err)
            return err
        }
    }
}
```

The basic idea here is to start a loop that will end when either the client has sent all of its messages, resulting in an `io.EOF` error when you call `Recv`, or if any other error occurs while streaming in the messages.

In this case, create a `ZipWriter` that will be used for the duration of the request (outside of the `for` loop). Each message you receive from the client is added to the writer. Once you have all of the messages from the client, send a response back with the compressed bytes from the writer.

The client can then save the response to a local file on disk.

Implementing the client

With the server implemented, let's move on to the client. This time, you'll use a Python script to send files to the archive service and store the result in *compressed.zip* in the current directory.

You'll need a new directory for the Python code and need to include the `grpcio` package in order to make gRPC calls. To do that, run the following commands:

```
mkdir -p archiver/proto && cd archiver
pip install --user pipenv
pipenv install grpcio==1.11.0
cd ../
```

Next, you'll need to generate the Python client stubs like you did for the database example.

```
docker run --rm -v $(PWD):/defs \
  namely/protoc-all:1.9 -f proto/archiver.proto -l python -o
archiver
```

Now let’s define the client code in *archiver/main.py*:
Python

```python
import sys

import grpc

from proto import archiver_pb2
from proto import archiver_pb2_grpc as rpc


def _read(path):
    with open(path, 'r') as stream:
        return archiver_pb2.ZipRequest(file_name=path,
contents=stream.read())


def archive(file_paths):
    """Send each file to the archiver service to be zipped
up.

    The resulting zip file will be written to compressed.zip
in
    the current working directory.
    """
    channel = grpc.insecure_channel('localhost:8080')
    stub = rpc.ArchiverStub(channel)

    files = iter([_read(path) for path in file_paths])
    future = stub.Zip.future(files)

    # we'll just let errors go to the console
    response = future.result()

    with open('compressed.zip', 'w') as stream:
        stream.write(response.zipped_contents)


if __name__ == '__main__':
    archive(sys.argv[1:])
```

The archive function takes a set of file paths supplied in the command line and creates an iterator of `ZipRequest` messages to be sent to the server. We've used a *future* here to allow us to potentially do some other work while waiting for the requests to be generated and sent. Calling `future.result()` will block until the server has sent its response.

At this point, you can write the response from the server to *compressed.zip*.

Running the example

You'll need to update the server binary to include the new service. To do this, add the following line in the main func in *server/main.go*:

Go

```
// put this just before pb.RegisterDatabaseServer(s,
new(DatabaseService))
pb.RegisterArchiverServer(s, new(ArchiverService))
```

Again, you'll need two terminal windows to run this example. In the first terminal window, build and run the server.

```
go build -o server ./server
./server
```

In the second terminal window run the client.

```
cd archiver
pipenv run python main.py Pipfile Pipfile.lock
```

This will send *Pipfile* and *Pipfile.lock* to the archiver service, which will zip them up and return a zip file that will be written to *compressed.zip*. You can verify this by extracting the zip file (somewhere else) and comparing the contents with the files in the current directory.

Bidirectional streaming

The final streaming method is bidirectional (bidi) streaming. This involves having the client and the server streaming messages to each other. Messages are guaranteed to be received in the order they were sent (on both ends), but there is no need to wait for the other side to start or finish sending before you start sending messages.

Let's take a look at an example application that uses bidi streaming. Here we'll create a word counting application. The basic idea is to implement something like a distributed map/reduce job, where files are mapped by the tokenizer service, and the driver (client) handles the reduce step (i.e. merging the results).

Clients will stream files to the server, which for each message, will generate a map of word counts in the file. The client will then merge the results as they come in to form a final unified word map (occurrences in all of the files).

Creating the protobuf file

As always, let's start by defining the protobuf in *proto/tokenizer.proto*:

Protocol Buffer

```
syntax = "proto3";

package practical.grpc.v1;

service Tokenizer {
  rpc Tokenize(stream TokenizeRequest) returns (stream
TokenizeResponse);
}

message TokenizeRequest {
  bytes file_contents = 1;
}

message TokenizeResponse {
  map<string, int64> words = 1;
}
```

Notice how the request and response types are both marked with `stream`.

Implementing the server

As usual, we'll need to generate some code with `protoc`. Run the following to create *server/proto/tokenizer.pb.go*:

```
docker run --rm -v $(pwd):/defs \
  namely/protoc-all:1.9 -d proto -l go -o server/proto
```

Now create the service implementation in *server/tokenizer.go*.

Go

```
package main

import (
    "bufio"
    "bytes"
    "io"
    "log"
    "strings"

    pb "practical_grpc/streaming/server/proto"
)

type TokenizerService struct{}

func (ts *TokenizerService) Tokenize(s
pb.Tokenizer_TokenizeServer) error {
    for {
        req, err := s.Recv()
        if err == io.EOF {
            return nil
        }

        // an error occured when getting the request object
        if err != nil {
            log.Printf("Got an error receiving from the
client: %v", err)
            return err
        }

        rdr :=
bufio.NewReader(bytes.NewReader(req.FileContents))
        scanner := bufio.NewScanner(rdr)
        scanner.Split(bufio.ScanWords)

        results := &pb.TokenizeResponse{
            Words: make(map[string]int64),
        }

        for scanner.Scan() {
            word := strings.TrimSpace(scanner.Text())

            if _, ok := results.Words[word]; ok {
                results.Words[word]++
            } else {
                results.Words[word] = 1
            }
        }

        if err = s.Send(results); err != nil {
            log.Printf("Got an error sending to the client:
%v", err)
            return err
        }
    }
}
```

Similar to the client streaming method, start with a for loop that breaks when the client has finished sending messages, or when an error occurs.

Each message that's received the service creates a map of words and their counts. This map is then sent back to the client for aggregation/reduction purposes.

This looks very similar to the client streaming code. The main difference here is the call to `Send` rather than `SendAndClose` when sending messages back to the client.

The method used to count words is just for demonstration purposes. In a real application you should handle punctuation, capitalization, and other edge cases appropriately.

Implementing the client

Next, let's implement the client. This time we'll make a node client that streams files to the service and reduces the results to a single map of word counts.

To start, create a new folder and add a few NPM modules as dependencies.

```
mkdir -p tokenizer/proto && cd tokenizer
npm install async@2.6.0 google-protobuf@3.5.0 grpc@1.10.1
cd ../
```

Next, generate the client stubs as we did for the other examples:

```
docker run --rm -v $(PWD):/defs \
  namely/protoc-all:1.9 -f proto/tokenizer.proto -l node -o
tokenizer
```

This will create *tokenizer/proto/tokenizer_pb.js* and *tokenizer/proto/tokenizer_grpc_pb.js*. We'll use these in our script to manage the communication with the service.

With the generated protobuf code, and the dependencies in place, you can now write the client code.

In *tokenizer/index.js*, add the following code:
JavaScript

```
var fs = require('fs')

var async = require('async')
var grpc = require('grpc')

var messages = require('./proto/tokenizer_pb.js')
var services = require('./proto/tokenizer_grpc_pb.js')
var client = new services.TokenizerClient('localhost:8080',
grpc.credentials.createInsecure())

function tokenizeFiles(files, done) {
  var call = client.tokenize()
  var error = null
  var wordsMap = {}

  call.on('data', function(resp) {
    resp.toObject().wordsMap.forEach(function(obj) {
      var word = obj[0]
      var count = obj[1]

      wordsMap[word] = (wordsMap[word] || 0) + count
    })
  })

  call.on('end', function() { done(error, wordsMap) })

  sendCalls = files.map(function(file) {
    return function(callback) {
      fs.readFile(file, function(err, data) {
        if (err != null) {
          callback(err, null)
          return
        }

        var req = new messages.TokenizeRequest()
        req.setFileContents(data.toString('base64'))

        call.write(req)
        callback(null, null)
      })
    }
  })

  async.parallel(sendCalls, function(err, _) {
    error = err
```

```
      call.end()
    })
  }

  tokenizeFiles(process.argv.slice(2), function(err, results) {
    if (err != null) {
      console.log('An Error occured:', err)
      process.exit(1)
    }

    console.log('All finished')
    console.log(results)
  })
```

We're using the async library to parallelize the process of sending messages to the server. Each file path (from CLI args) is read into a `TokenizeRequest` message and sent to the service.

When all of the request have been sent in, we close the channel by calling `call.end()`.

As the results are returned from the server, build up a map of all of the word counts. When all requests have been processed, the results are printed to the console.

NOTE: This is again, just for demonstration purposes. When dealing with streaming APIs the volume of messages being sent and received may be too large to keep in memory (our map in this example).

Running the example

You need to update the server binary to include the new service. To do this, add the following line in the main func in *server/main.go*:

Go

```
// put this just after pb.RegisterDatabaseServer(s,
new(DatabaseService))
pb.RegisterTokenizerServer(s, new(TokenizerService))
```

Again, you need two terminal windows to run this example. In the first terminal window, build and run the server.

```
go build -o server ./server
./server
```

In the second terminal window you can run the client.

```
cd tokenizer
node index.js package.json index.js
```

This will send *package.json* and *index.js* to the service to be tokenized. The resulting map will be printed to the console.

Summary

In this chapter we covered the three modes of streaming with gRPC. We created example applications to demonstrate streaming records to the client (server streaming), streaming records to the server (client streaming), and finally streaming messages back forth between the client and server (bidi streaming).

At this point, you should have a good grasp of the basics. In the next chapter, we'll take a look at some of the more advanced features of gRPC including authentication, interceptors, and metadata.

Advanced gRPC 7

Now that we've learned some of the higher levels of how gRPC works, we're going to explore some of the more advanced aspects of gRPC:

1. How errors are encoded from servers to clients.
2. gRPC Interceptors for both clients / servers.
3. Simple authentication using JSON web tokens.
4. Timeouts and how gRPC implements them.
5. Cancelling RPCs, including some advanced recipes that make use of cancellation.

Error management

In the basics section of gRPC, we briefly covered errors the different error codes a server can respond with. In gRPC, error codes are helpful for indicating what exactly happened, whether it be success or failure. But there are times where more than just "not found" needs to be reported back to a calling client.

For example, if you're attempting to create a resource inside of a server, and that server decides that the input on the request is invalid (a malformed email for example), it may need to include more information back to the client to make the appropriate decision other than "invalid data."

Using the malformed email as an example case, a client may want to display to the user attempting to create an account that the email is invalid. After all, the user wouldn't like if the page only displayed "invalid data." Instead, a helpful error message would be nice so they can act on the field that was invalid.

gRPC provides this functionality by allowing you to encode in additional information in the status reported back from a server. When a gRPC server reports status it uses the `google.rpc.Status` message. This simple message definition has three fields on it for indicating success/failures in RPCs. The message looks like the following:

```
message Status {
  // The status code, which should be an enum value of
[google.rpc.Code][google.rpc.Code].
  int32 code = 1;

  // A developer-facing error message, which should be in
English. Any
  // user-facing error message should be localized and sent
in the
  // [google.rpc.Status.details][google.rpc.Status.details]
field, or localized by the client.
  string message = 2;

  // A list of messages that carry the error details.  There
is a common set of
  // message types for APIs to use.
  repeated google.protobuf.Any details = 3;
}
```

The first field is the gRPC status code that is one of the codes discussed in the Basics of gRPC section. The second field, message, is usually a generic message commonly used in either reporting services, logs, etc. It's not designed to be used to display human friendly errors. That's where the third field comes in.

The `details` field is a repeated field of the Any message. Part of the "Well Known Types" protobufs package, it is used to encode *anything* into protocol buffers. This means you can encode in other protocol buffers of any type into your RPC status messages to include details such as human-friendly error messages. For example, you can declare a protobuf message called "InvalidKey" that contains a key and message fields.

```
message InvalidKey {
  string key = 1;
  string message = 2;
}
```

This message definition can be used for when a potential user generated error has occurred on a feature, such as creating an account.

```
InvalidKey{Key: "email", Message: "The email provided is
invalid"}
```

Let's take a look at a simple validation endpoint for Star Friends' species in the Go language. In this endpoint we're going to validate the length of a name of a species. If the name is too short, return an `InvalidArgument` gRPC error code, but also include some human friendly details that the client can use to display to the user.

```
func (s *Server) ValidateSpecies(ctx context.Context, r
*pb.ValidateSpeciesRequest) (*pb.ValidateSpeciesResponse,
error) {
    if len(r.GetName()) < 2 {
        status := status.New(codes.InvalidArgument,
"invalid")
        key := &pb.InvalidKey{Key: "name", Message: "The
name provided is not long enough"}

        status, err := status.WithDetails(key)
        if err != nil {
            return nil, err
        }

        return nil, status.Err()
    }

    return &pb.ValidateSpeciesResponse{}, nil
}
```

In this handler we're checking that the name length is at least two characters long. If it isn't, generate a new status message with the `InvalidArgument` code and a message of "invalid." You then initialize the proto message `InvalidKey` with the name and human friendly version of the name length error. You can then append this proto message to a status message using the `WithDetails` method (specific to Go, but other languages support this same functionality as well).

This encodes the InvalidKey message into the status before we ship it back to the client saying "something went wrong."

How gRPC encodes errors

When a gRPC server responds, it includes the status of the response in the headers portion of the HTTP response. If nothing went wrong with the request, the `:status` field is set on the response with the HTTP code of 200. The gRPC status is encoded into the `grpc-status` field with the numeric identifier (3 for `InvalidArgument` in our example). For our example, we respond with additional details, so gRPC encodes the `google.rpc.Status` message as base64 and sets the `grpc-status-details-bin` with the details included. This is what allows clients to receive additional details from our server.

On the client side, clients can read the field `grpc-status-details-bin`, decode it back to `google.rpc.Status`, and then read the additional details field for the Any field.

Here are the headers that will be set on the response for the original caller:

Header	Usage
:status	The HTTP status code
content-type	The HTTP content type
grpc-status	The gRPC status code
grpc-message	A simple message included with the response
grpc-status-details-bin	An base64 encoded `google.rpc.Status` message

Using Ruby as an example, you can retrieve human friendly errors with relative ease using a few of the included gRPC/Google helpers.

```ruby
$LOAD_PATH.unshift File.expand_path("../proto", __FILE__)
require_relative './proto/sfapi_pb'
require_relative './proto/sfapi_services_pb'

# These are required to work with status messages easily
require 'grpc/google_rpc_status_utils'
require 'google/protobuf/well_known_types'

stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)

begin
  request = Sfapi::V1::ValidateSpeciesRequest.new(name: 'a')
  response = stub.validate_species(request)
rescue GRPC::InvalidArgument => e
  status =
GRPC::GoogleRpcStatusUtils.extract_google_rpc_status(e.to_status)

  status.details.each do |detail|
    error = detail.unpack(Sfapi::V1::InvalidKey)
```

```
      puts "#{error.key}: #{error.message}"
    end
  end
```

Here we're calling the `ValidateSpecies` endpoint with an invalid name intentionally to generate an exception of `GRPC::InvalidArgument`. The server responds accordingly with additional details encoded into the error in the `grpc-status-details-bin`. You can then extract the status and iterate over the additional details, unpacking them into the `InvalidKey` message, and finally you can print the results out.

This pattern is incredibly useful for transporting more than just a code and simple string. Often times a calling client will need more information to react appropriately upon failures. The ability to encode additional details into the `google.rpc.Status` message makes this possible.

KEEP IN MIND

These additional features of gRPC are not enabled by default in many language implementations. In our Ruby example you must require two additional files in order to gain this functionality.

In reality, gRPC is doing nothing more than encoding a `Status` message and sending across the wire. Since the `google.rpc.Status` and `google.protobuf.Any` messages are just encoded bytes, helpers are commonly provided to help extract and make these bytes useful in your applications. These "Well Known Types" will have their own implementations of these helpers per language.

Interceptors

There are scenarios where you may want the ability to inject in metadata before sending a request to a gRPC server. Another possible requirement is that you need to reject a request to a server based on some prerequisite such as authentication. In both of these scenarios you might use a pattern called "middleware". A very common feature of application frameworks like Ruby on Rails is the ability to add and modify a middleware stack. This includes checking sessions, CSRF protection, or even just compressing the response body before sending it back.

In the gRPC world this functionality is called "interceptors". An interceptor allows a client or server to wrap RPC requests and responses. This includes streaming and unary RPC calls as well. An interceptor has the ability to manipulate almost everything about a procedure call, but is more commonly used for wrapping around logic for things such as authentication, metrics, distributed tracing, etc.

The way it works is that gRPC will check if an interceptor has been registered for the current call. This can be a unary or streaming client or server call. For example, if a client has a unary interceptor attached to it the procedure goes in this order:

1. Check if there's an interceptor.
2. If there is, pass the request information to it (this includes the method name and request protobuf message).
3. Your interceptor has the opportunity to modify the message or return an error.
4. The client stub takes the result of the interceptor (modified message or error) and continues as normal.

Interceptors have slightly different implementations across the supported languages of gRPC. For example, The Ruby gRPC library supports adding multiple interceptors to a client stub, whereas Go only supports adding one and you're responsible for building your own middleware chain. Packages such as Go gRPC Middleware (**https://github.com/grpc-ecosystem/go-grpc-middleware**) offer this functionality for you. Let's take a look at a few scenarios where interceptors in gRPC make a lot of sense.

Logging unary RPCs

In some scenarios, you may want to log RPC calls made to a gRPC server. This is a great opportunity to use a server interceptor to wrap the logic performed with some simple tagged logs. For example, let's say we want to know how long the request took? Did the server handler error at all? Or we can even add metadata to the response.

Using Go as an example, we can make a simple function that wraps all calls to our gRPC server to log requests that have occurred.

```
func LoggingInterceptor(ctx context.Context, req
interface{}, info *grpc.UnaryServerInfo, handler
grpc.UnaryHandler) (out interface{}, err error) {
    out, err = handler(ctx, req)
    logrus.Info("handled rpc")
    return out, err
}
```

As you can see here, we're creating a function that accepts the RPC context, the request, general info about the request (method name for example), and the *original* handler would have received this request if was not accepted. It is the interceptors responsibility to call the handler should it be necessary. There are cases where an interceptor would *not* call the handler, which we'll cover here shortly.

You can extend this logging functionality by including the method name for the request as well. All interceptors receive information about the call. In Go's case, you have it on the `grpc.UnaryServerInfo` type. Let's add a field to our log line that includes the method name.

```
func LoggingInterceptor(ctx context.Context, req
interface{}, info *grpc.UnaryServerInfo, handler
grpc.UnaryHandler) (out interface{}, err error) {
    out, err = handler(ctx, req)
    logrus.WithField("method",
info.FullMethod).Info("handled rpc")

    return out, err
}
```

There are several possibilities for logging with interceptors. You can potentially include metrics to perform simple timings and send them to statsd, mutate the reply from the handler, or catch errors to report them to an error reporting service.

Capturing and mutating errors

There are times where you may want to intercept an error that comes out of a server handler and instead return a different error to the client. For example, you may have an endpoint that retrieves information from a database and replies with it. However, retrieving that information may sometimes fail because the given parameters to search for it came up with zero results. In common application frameworks like Ruby on Rails, an exception is raised such as `ActiveRecord::RecordNotFound` and the framework handles responding with an HTTP status code of 404. In the world of gRPC, however, we need to get creative to handle this behavior. Interceptors are another great use case for handling errors appropriately based on their type.

In our Star Friends API it's possible to ask for a singular resource that does not exist in our database. Internally within our database provider code, we return a type of `NotFoundErr`, which then gets bubbled up to the server handler:

```
func (s *Server) GetSpecies(ctx context.Context, r
*pb.GetSpeciesRequest) (*pb.GetSpeciesResponse, error) {
    species, err := s.db.GetSpeciesByID(ctx, r.Id)
    if err != nil {
        return nil, err
    }

    return &pb.GetSpeciesResponse{Species: species}, nil
}
```

In our `if` block we check if any error was returned by our database provider. If something did in fact error, we bubble that error up to the original remote caller in our server handler.

Let's see what happens when we request a species that does not exist by using a Ruby client as an example.

```
$LOAD_PATH.unshift File.expand_path("../", __FILE__)
require_relative './proto/sfapi_pb'
require_relative './proto/sfapi_services_pb'

stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)

begin
  request = Sfapi::V1::GetSpeciesRequest.new(id: 'bunk')
  response = stub.get_species(request)
  puts "Film Title: #{response.film.title}"
rescue StandardError => e
  puts "Code: #{e.code}, ErrorClass: '#{e.class}', Message:
#{e.message}"
end
```

In this example we initialize a simple client stub using the Ruby gRPC gem and ask for a species from our server with an ID that does not exist (bunk). When we run this snippet, we'll receive an error:

```
$ ruby client_example.rb
Code: 2, ErrorClass: 'GRPC::Unknown', Message: 2:species
with id 'bunk' not found
```

The error message is helpful, but the error code is not accurate. As we discussed in the basics of grpc section, a server can return status codes to better indicate the response.

Sometimes, however, the error might be benign. There can be times where your database simply does not have the resource that was requested, like in our example. In this case, it makes sense to return to the client an error code of NotFound (the numeric error code of 5). Given this strategy, you can write a simple interceptor to check the error type and return an entirely different error code to better reflect to the client what happened.

Using the same function signature as our previous example, we can write another interceptor to handle this in Go:

```
func ErrorsInterceptor(ctx context.Context, req interface{},
info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (out
interface{}, err error) {
    out, err = handler(ctx, req)

    switch tErr := err.(type) {
    case NotFoundErr:
        return out, grpc.Errorf(codes.NotFound, tErr.Error())
    }

    return out, err
}
```

In this code we're checking the type of the error returned by the handler to see if we need to swap it for a more helpful error like the NotFound gRPC error code. If we do, we return from the function early, otherwise we return the original values the handler gave us.

After this interceptor is added to our gRPC server, our error codes are automatically adjusted whenever a database "not found" error occurs.

```
$ ruby client_example.rb
Code: 5, ErrorClass: 'GRPC::NotFound', Message: 5:species
with id 'bunk' not found
```

You can see this by running the small Ruby snippet again to see that the text printed out now includes Code: 5. Because of our gRPC server interceptor, no API endpoint needs to concern itself about error codes they are returning. Instead, the interceptor itself handles the logic to return the proper error codes to the calling clients.

Adding timing to stubs

Interceptors are not only used for server side applications, they can also be added to gRPC client stubs. When an interceptor is added to a stub, it is enacted on every call to a service endpoint. This introduces a lot of possibilities for client side communication, such as retries, distributed tracing headers, and payload normalization. With client retries, for example, we can add a timing interceptor that times how long each RPC call takes and reports it. This is useful for sending metrics to something like StatsD or Prometheus, as examples. Let's take a look at another Ruby example to see how this might work.

Ruby

```
$LOAD_PATH.unshift File.expand_path("../", __FILE__)
require_relative './proto/sfapi_pb'
require_relative './proto/sfapi_services_pb'

class TimingInterceptor < GRPC::ClientInterceptor
  def request_response(request: nil, call: nil, method: nil,
metadata: nil)
    start = Time.now.to_f
    yield
    duration = Time.now.to_f - start
    puts "Call for #{method} took #{duration * 1000}ms"
  end
end

stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure, interceptors:
[ TimingInterceptor.new ])
request = Sfapi::V1::GetSpeciesRequest.new(id: 'HJFy629NoG')
response = stub.get_species(request)
```

In this example we're making a simple interceptor class that has a `request_response` method on it. The Ruby gRPC gem will run this method before every unary RPC call allowing us wrap the request with timing logic. When we run this code we'll see a nice message telling us how long a single call took:

```
$ ruby client_interceptor_example.rb
Call for /sfapi.v1.Starfriends/GetSpecies took 8.04591ms
```

Great! Our interceptor has wrapped our gRPC call and reported how long it took to complete.

Streaming interceptors

Streaming interceptors work just like unary interceptors. Streaming interceptors also have the ability to act upon messages being received and sent to an open gRPC stream. In Ruby, they're almost exactly the same as unary interceptors with some minor tweaks.

```
class TimingInterceptor < GRPC::ClientInterceptor
  def server_streamer(request: nil, call: nil, method: nil,
metadata: nil)
    start = Time.now.to_f
    result = yield
    duration = Time.now.to_f - start
    puts "Call for #{method} took #{duration * 1000}ms"

    return result
  end
end

stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure, interceptors:
[ TimingInterceptor.new ])
request = Sfapi::V1::ListStarshipActionsRequest.new
response = stub.list_starship_actions(request)

response.each do |action|
  puts "#{action.starship.name} #{action.action}"
end
```

In this example, we change the method name in our timing interceptor to `server_streamer`. This method has the same signature as unary calls, however, it *must* return the values that come from `yield`. So we store the result in a variable and ensure the method returns them.

When we run this example, we can see our streaming endpoint returns starship actions as an enumerable list:

```
$ ruby client_streaming_interceptor.rb
Call for /sfapi.v1.Starfriends/ListStarshipActions took
1.78217ms
Millennium Falcon TOOKOFF
Millennium Falcon HYPERDRIVE
Millennium Falcon LANDED
Millennium Falcon HIDING_IN_A_MOUTH
```

Limitations

Interceptors do have their limitations though. They vary by implementation, but it's important to call out a few that you might attempt to implement, but quickly realize it's not possible.

In some implementations of gRPC, it is not possible to re-run failed RPC calls. For example, in the Ruby implementation, if you write an interceptor that will run `yield` again after an exception is raised, gRPC will throw an exception:

```
grpc/generic/active_call.rb:627:in `block in
raise_error_if_already_executed': attempting to re-run a
call (GRPC::Core::CallError)
```

Another limitation is that you cannot modify results in some implementations of gRPC after they have been returned by the RPC endpoint. For example, you cannot uppercase the value in the species name without receiving a frozen string error in Ruby:

```ruby
def request_response(request: nil, call: nil, method: nil,
metadata: nil)
  start = Time.now.to_f
  result = yield
  duration = Time.now.to_f - start
  puts "Call for #{method} took #{duration * 1000}ms"

  # Attempt to mutate the species name
  result.species.name.upcase!
  result
end
```

Running this example with this modification yields an exception:

```
$ ruby client_interceptor_example.rb
Call for /sfapi.v1.Starfriends/GetSpecies took
2.2199153900146484ms
client_interceptor_example.rb:12:in `upcase!': can't modify
frozen String (RuntimeError)
    from client_interceptor_example.rb:12:in
`request_response'
```

Each implementation of gRPC will have different limitations for interceptors because of the nature of the languages they are implemented in. As you implement your own interceptors make sure you've written plenty of tests around them to guarantee their functionality as they modify **every** RPC.

Closing up

As you can see in the last few practical examples, interceptors are a great feature of gRPC that allow you to hook into the RPC lifecycle and react to certain events. They allow you to create abstractions and reduce duplication in your server and client implementations for your applications.

With great power comes great responsibility, though. While interceptors are a great abstraction to add, they can create bugs that are more difficult to track down. This is because interceptors are commonly configured in a separate code path but can affect all aspects of your application. Make sure to document your interceptors and their behavior well to ensure that your team knows they are there.

Authentication

gRPC provides primitives for setting up TLS/SSL between clients and servers. It also provides the ability to setup mutual TLS between clients and servers! However, gRPC doesn't provide the facility of authenticating users/services easily. It is up to the developer to implement their own authentication to verify services have provided the correct credentials on RPC's. In this section we're going to cover adding authentication to a gRPC service using metadata and JWT (JSON Web Tokens) to provide credentials to servers from clients.

Our clients will be providing a signed JWT using the `metadata` field of all RPCs. The JWT will provide a username so the service can use it for any actions taken against it. Authentication is another great candidate to use gRPC interceptors, so that's what we're going to use to implement this functionality.

Let's quickly build a simple `Authentication` interceptor to our server that pulls the JWT out of the `metadata` of the request and verifies if the token has a valid signature.

```
func AuthenticationInterceptor(ctx context.Context, req
interface{}, info *grpc.UnaryServerInfo, handler
grpc.UnaryHandler) (out interface{}, err error) {
    secret := os.Getenv("JWT_SECRET")
    if secret == "" {
        return handler(ctx, req)
    }

    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        return nil, errGrpcUnauthenticated
    }

    tokenString, ok := md[AuthTokenKey]
    if !ok || len(tokenString) < 1 {
        return nil, errGrpcUnauthenticated
    }

    token, err := jwt.Parse(tokenString[0], func(token
*jwt.Token) (interface{}, error) {
        return []byte(os.Getenv("JWT_SECRET")), nil
    })
    if err != nil {
        return nil, err
    }

    if claims, ok := token.Claims.(jwt.MapClaims); ok &&
token.Valid {
        ctx = context.WithValue(ctx, UserCtxKey{},
claims["un"])
    } else {
        return nil, errGrpcUnauthenticated
    }

    return handler(ctx, req)
}
```

Let's break down this interceptor. First we're checking if we've set a JWT secret for checking token signatures. If we have not set one we'll just call the handler without setting users. This allows us to turn on and off authentication.

Next, we pop the metadata off of the context that was sent with this gRPC request. We inspect it to see if it has any token information on it and then verify the token itself. If the token is valid, we pass the username field on it to the context and call the original handler with our new context that contains the username. If the token is invalid or does not contain any username information on it, we reject the request outright and return a `code:16` or `unauthenticated`.

Using our Ruby example from the last section, let's attempt to run it with the new interceptor enabled.

```
$ ruby client_example.rb
Code: 16, ErrorClass: 'GRPC::Unauthenticated', Message:
16:missing authentication token
```

Great! Our gRPC server received a request, called our Authentication interceptor, and rejected the request because it did not contain any authentication information on it. Let's add a fake token to our RPC request metadata and attempt again to see what happens when the token is present but invalid. To do this in Ruby, you can add a `metadata` keyword argument that is a hash. For example:

```
stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)
request = Sfapi::V1::GetSpeciesRequest.new(id: 'bunk')
response = stub.get_species(request, metadata:
{ authentication: "bunk" })
```

When you add the authentication metadata and run the example again, the error changes:

```
ruby client_example.rb
Code: 2, ErrorClass: 'GRPC::Unknown', Message: 2:token
contains an invalid number of segments
```

We've provided an invalid JWT, thus the interceptor rejects the request entirely with an error. Let's provide a *valid* token and attempt to retrieve our species again. In this example, we've set the `JWT_SECRET` environment variable to "bunk." Our updated Ruby code looks like this:

```
stub = Sfapi::V1::Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)
request = Sfapi::V1::GetSpeciesRequest.new(id: 'bunk')
response = stub.get_species(request, metadata:
{ authentication:
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1biI6ImJvYmJ5dGFibGVz
In0.5Vlzb4UdXxsw9zNuIpsesB3tBTx2MFPMaVF_KGH3LqQ" })
```

Our token contains the payload:

```
{
  "un": "bobbytables"
}
```

When we run this example now, we get our expected A Species not found error:

```
$ ruby client_example.rb
Code: 5, ErrorClass: 'GRPC::NotFound', Message: 5:species
with id 'bunk' not found
```

Perfect. Our interceptors are all playing nicely with each other and rejecting unauthenticated requests. This means that all of our RPC endpoints will now be secure without having to add any additional code to them.

RPC Timeouts

In gRPC, clients and servers can specify the amount of time a request may take for both unary and streaming calls. The term used for this functionality is “timeout.” The implementation of the timeout is language specific, but for the most part operates on the same concept.

Timeouts are important to consider if you are working with an application that may take a long time to process a call. If you don’t specify a timeout on a call, it can potentially operate forever (lets say your application accidentally goes into an infinite loop), causing compute resources to eventually block.

To overcome this, you specify a timeout on your RPC calls for how long you are willing to wait for it to complete. If a timeout is not specified on a gRPC request, the call is timeout and is interpreted as infinite.

When a client specifies a timeout, it is added to the request headers at the key `grpc-timeout` with a numeric value and unit. For example to specify a timeout of 1 minute, the header on the request will be `grpc-timeout = 1M`. Available units are:

Unit	Format
Hour	H
Minute	M
Second	S
Millisecond	m
Microsecond	u
Nanosecond	n

Only one unit can be provided while defining a timeout in the raw request payload. The client for your language will handle this for you.

When a request has a designated timeout attached the server should honor the timeout. Client's may also cancel the call if they've determined the timeout has been reached. The Ruby client, for example, will cancel the call client side when a timeout has been specified for a single RPC even if the server is still processing it.

If a client does not provide a timeout for the call that it is making, the server may still provide it's own deadline on the request. Server's can indicate they have stopped processing the request due to a timeout by returning with a gRPC status code of `DeadlineExceeded` (Code 4). However, a server may still have completed the request successfully. From the gRPC documentation on deadline exceeded codes:

"DeadlineExceeded means operation expired before completion. For operations that change the state of the system, this error may be returned even if the operation has completed successfully. For example, a successful response from a server could have been delayed long enough for the deadline to expire."

Servers and Clients are independently responsible for determining if an RPC is successful. If a client has set a timeout on an RPC call, the server may have completed the response and sent it back, but if the client receives the final result *after* the timeout, it will error with `DEADLINE_EXCEEDED`.

For example, a client might set a deadline of three seconds when calling an endpoint named `ChargeCard` in a gRPC server. This endpoint is responsible for charging a customer's credit card for a purchase and relies on a third party to perform the request. The moment the client sends the request is when this three second timer starts. The `ChargeCard` endpoint attempts to charge the card, and the third party responsible for performing the charge takes a little longer than usual, making the endpoint block for 2.5 seconds. The server finishes its response and sends it 2.99 seconds after the client started the call. By the time the client has received the entire response, it's past three seconds, so it will declare that the RPC has exceeded it's deadline and return an error or raise an exception.

So the gRPC server has marked the call as a success because it completed it's job function, however, the client has marked the call as unsuccessful.

In this scenario, it's possible that your client may attempt to *retry* this request. This scenario makes sure you design your APIs to be idempotent so that things like charging a card doesn't happen twice to a poor customer when a deadline exceeded occurs.

Call cancellation

Another cool thing about gRPC requests is that they can be cancelled. This can be useful to reduce wasted resources after an exception--if the application realizes that it no longer needs the response, it can cancel the RPC. The cancellation is propagated to the server, so the server can see that the client has cancelled and stop processing.

For synchronous APIs, like in Ruby, a cancellation can be a little subtle. You might find yourself more likely using deadlines and deadline propagation (described in the previous section) whereby the RPC is automatically cancelled after a configured duration. For unary RPCs in Ruby, another thread is required to cancel an RPC, since the thread that issued it is blocked, waiting on the response.

In Ruby, the steps to make a cancellable RPC are similar to what we learned for examining the server's metadata. Back in the "gRPC Basics" chapter: you get an `Operation` from the stub instead of the actual response:

Ruby

```ruby
op = stub.get_film(request, return_op: true)
```

Keep in mind that *you must invoke the `execute` method* on the operation in order to actually send the RPC. Like you saw in the previous section, the `execute` method will send the RPC, block until the server replies, and then return the response message. It is also important to note that the `execute` method *cannot be invoked more than once*. Other threads that have an `Operation` and want to wait for the RPC to finish must instead use its `wait` method.

Cancelling the RPC requires making the `Operation` accessible to another thread, which can concurrently call the `cancel` method:

Ruby

```ruby
op = stub.get_film(request, return_op: true)
# publish op so another thread can access it
response = op.execute

# in the other thread
op.cancel
```

We saw in the “gRPC Basics” chapter that the `Operation` provides access to the server’s response metadata. In addition to metadata and cancellation facilities, the `Operation` also lets you query if the RPC is done, by invoking the `status` method, which will return `nil` until the RPC finishes. This same `status` method is also how other threads can determine whether the RPC succeeded or not, after it has completed. Once the RPC is complete, the `status` method returns a gRPC status object, whose `code` attribute will be zero on success or one of the gRPC error codes on failure:

Ruby

```
op = stub.get_film(request, return_op: true)
# publish op so another thread can access it
response = op.execute

# in the other thread
if s.nil?
  puts "still waiting"
else if s.code == 0
  puts "completed successfully"
else
  # details has the message sent from the server
  puts "failed: #{s.details} (code = #{s.code})"
end
```

The above example shows another thread printing a snapshot fo the RPC’s result, which may be incomplete if the snapshot is printed for the operation finishes.

Parallelizing steps

An interesting use case for cancellation is when RPCs are issued in parallel, to reduce latency. Issuing them sequentially means that the total latency cannot be less than the sum of round trip times for each RPC. But it is often the case that two or more of the RPCs being issued are independent: RPCs are independent if their request values do not rely on any response values from the other. In this case, both requests can be constructed ahead of time and then two threads can be used to issue the RPCs in parallel.

Cancellation enters the picture when one of the RPCs fails. If all of the operations are required, then we send back an error if and when any of the operations fail. When they are running in parallel, if one operation fails, it is polite to cancel all of the outstanding operations.

Ruby

```
# We're going to run 3 RPCs in parallel
request1 = GetFilmRequest.new(id: '4')
step1 = stub.get_film(request1, return_op: true)
request2 = GetFilmRequest.new(id: '5')
step2 = stub.get_film(request2, return_op: true)
request3 = GetFilmRequest.new(id: '6')
step3 = stub.get_film(request3, return_op: true)

all = [step1, step2, step3]
results = [nil, nil, nil]
error = nil
threads = []

all.each_with_index do |step, i|
  # we execute each step in its own thread
  threads.push(Thread.new {
    begin
      results[i] = step.execute
    rescue => e
      error = e
      # cancel outstanding ops on failure
      all.each do |other|
        other.cancel
      end
    end
  })
end

threads.each do |th|
  th.join
end

if not error.nil? then
  puts "failed: #{error.details} (code = #{error.code})"
else
  results.each do |res|
    puts res.inspect
  end
end
```

This example is using very simple (and inelegant) threading techniques, but it still conveys the point. We execute three RPCs in parallel. If any of them fail, we cancel the others. At the end, we report the failure (if any of them failed) or report all three responses.

Aggressive hedging

Another use case for cancellation is to send the same RPC to two (or more) servers and then keep whichever response comes back first. In this case, any outstanding RPCs can be cancelled when the first response is received.

Ruby

```ruby
# We're going to run 2 RPCs to two different servers
# and choose whichever is fastest
stub1 = Starfriends::Stub.new("localhost:
8080", :this_channel_is_insecure)
stub2 = Starfriends::Stub.new("localhost:
8081", :this_channel_is_insecure)

request = GetFilmRequest.new(id: '4')
step1 = stub1.get_film(request, return_op: true)
step2 = stub2.get_film(request, return_op: true)

all = [step1, step2]
result = nil
error = nil
threads = []

all.each_with_index do |step, i|
  # we execute each step in its own thread
  threads.push(Thread.new {
    begin
      result = step.execute
      # we have a result, so cancel other
      all.each do |other|
        other.cancel
      end
    rescue => e
      error = e if error.nil?
      # don't cancel: the other one might succeed
    end
  })
end

threads.each do |th|
  th.join
end
```

```
if not result.nil? then
    puts result.inspect
else
  puts "failed: #{error.details} (code = #{error.code})"
end
```

This example is very similar to the one above. Here, we are executing two RPCs to different servers. As soon as one finishes successfully, we cancel the other. At the end, we report the response if we received one. If both RPCs failed, we report the error from the one that failed first.

Other languages

Cancellation takes different shapes in other languages, but Ruby is fairly representative. The main alternate approach, used in Go and Java, is to associate operations with a context object, which can be cancelled. In Go, this is particularly simple and leans on the `context.Context` type (now in the standard library as of Go 1.7):

Go

```
ctx, cancel := context.WithCancel(context.Background())

req := &proto.GetFilmRequest{Id: "4"}
resp, err := stub.GetFilm(ctx, req)

// If cancel is invoked from a different goroutine,
// the call will be aborted.
```

As can be seen above, using `context.WithCancel` provides a `cancel` function that can be invoked to cancel the corresponding context. Passing a context in as the first argument to a stub method associates the RPC with that context. Any associated RPCs are cancelled if and when the context is cancelled.

Summary

In this chapter, we learned some more advanced features of gRPC, like how to exchange detailed structured information about errors, how to make use of interceptors to tackle cross-cutting concerns, and how to use timeouts and cancellations. This is a solid foundation for building awesome apps with gRPC!

In the next chapter, we’ll learn more about HTTP/2, the protocol that underlies gRPC and that advances the status quo for HTTP in many ways.

HTTP2 overview 8

gRPC is built on top of the HTTP2 protocol, because the submitted optimizations in relation to HTTP1 perform faster communications between clients and servers.

There are several improvements, such as the cancellation of Header-of-Line Blocking, header compression, the introduction of the binary framing notion, and the support for server push (in order to replace long polling) that make the protocol much more robust and efficient, without modifying the basic application semantics of HTTP.

Early in 2009, two Google software engineers proposed an experimental protocol on top of HTTP, to try to lower load latency on web pages. That protocol was called SPDY. Once the first studies were done, a new branch was created by another group of engineers. HTTP2 was born, taking as a starting point the lessons learned from SPDY.

Both protocols coexisted in parallel for some time. The goal of this partnership was to use SPDY as an experimental branch for testing new propositions, and once validated they were committed in the official HTTP2 branch.

HTTP/1.0 and HTTP/1.1

HTTP (Hypertext Transfer Protocol) is the most expanded way to make two machines communicate. It is a protocol for requesting files over a network, where the payload (both request and response) is transferred as plain text (ASCII).

That's the reason why we can easily use Telnet and type the HTTP commands in order to get the contents.

```
> $ telnet google.com 80
Trying 216.58.201.238...
Connected to google.com.
```

```
Escape character is '^]'.

GET /
# Status
HTTP/1.0 200 OK
# Headers
Expires: -1
Cache-Control: private, max-age=0
..........

# Content
<!doctype html>
..........
</html>
Connection closed by foreign host.
```

Later on, HTTP/1.1 added some performance optimizations:

Keep Alive connections: This prevents TCP connections from being closed every time that a response is sent. In the headers, you can set a timeout to tell how long an idle TCP connection may be kept open, as well as the maximum number of requests that may be sent in one single TCP connection.

```
HTTP/1.1 200 OK
Connection: Keep-Alive
Keep-Alive: timeout=5, max=1000

(body)
```

Chunked encoding transfers: This allows a client or a server to begin to send data by blocks without knowing the total length of the payload.

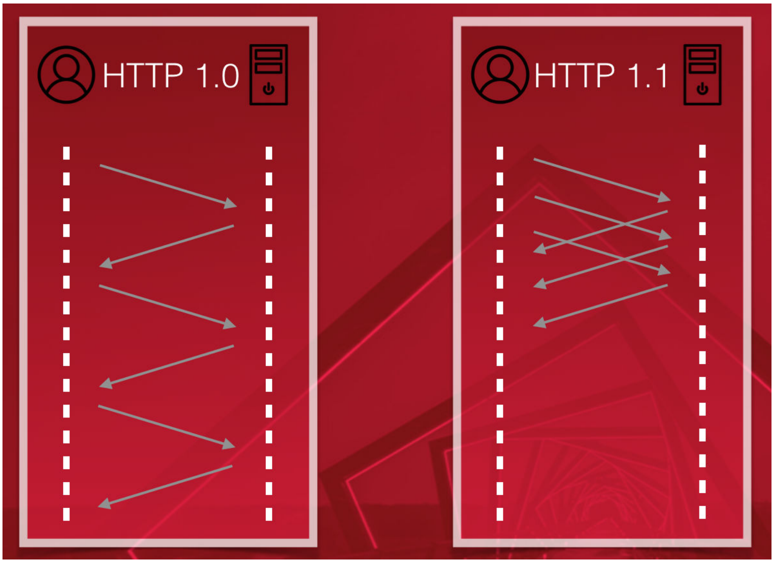
```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

E
Here some data

9
more data

9
some more

0
```

`Transfer-Encoding: chunked` replaces the `Content-Length` header.

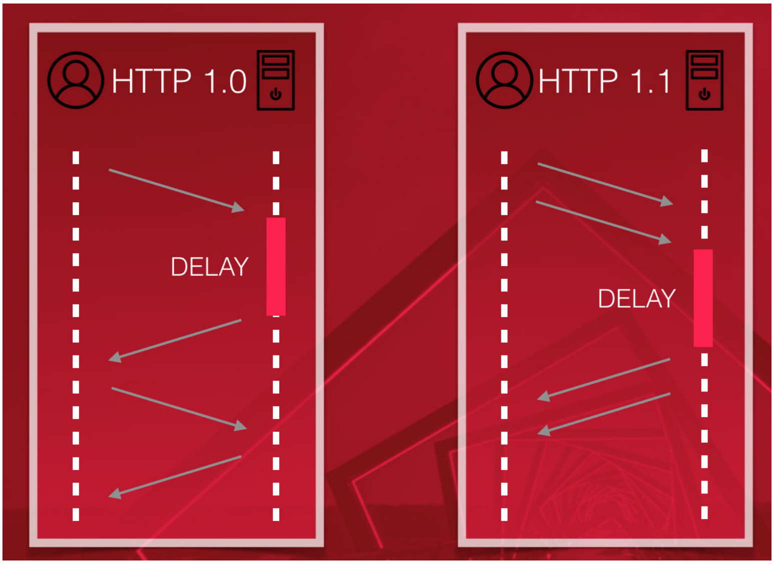
Byte serving of Range requests: This allows you to request for a part of the hosted file that you want to download. For instance, if you need to show a single page from a big `.pdf` file, you can ask only for the needed page.

Request pipelining: This allows you to send multiple HTTP requests within the same TCP connection, without being forced to wait for the precedent responses. That was one of the main handicaps of HTTP/1.0.

Taking advantage of the `Keep-Alive` functionality, HTTP/1.1 makes it possible for the request shooting without having to wait for the response:



You can make good use of the schema above to introduce the notion of Head-of-Line (HOL) blocking, which is a problem that persists with HTTP/1.1. HOL is a degradation of the performance that occurs when a bunch of packets are blocked in the FIFO (First Input First Output) buffer of a switch, because the first packet cannot be forwarded. The reason why a packet cannot be forwarded is that the destination port is busy. If this happens, all of the subsequent requests are blocked, even with the request pipelining introduced by HTTP/1.1. The schema would become something like:



HTTP/2

HTTP/2 was born to accompany the new needs of the Internet's exponential growth. The main goals are optimizing bandwidth, lowering latency, and allowing a higher throughput. The semantics and use cases should not be affected.

The basic improvement of HTTP/2 is the fact that it isn't a plain text protocol anymore, but it is binary. That of course decreases the payload sent all over the network, and furthermore it will allow other optimizations in the protocol.

Verbs and headers stay the same, and the only thing that changes is the way data is encoded or serialized.

HTTP/2 introduces multiple notions and they will be detailed in the following subsections.

Frames

A frame in HTTP/2 is the smallest unit data representation. It contains:

- A frame Header that normally only contains the identifier of the stream that it belongs to.

- The data content, which can have different formats depending on the kind of data that it contains. Most of the frames may contain a **Padding** block at the end of the frame, and it is used to obfuscate the length of the frame for security reasons. Here we have some types of frames:
    - **DATA:** (type=0x0) Contains the payload of the requests or the responses.
    - **HEADERS:** (type=0x1) Equivalent to HTTP/1.x headers, it is used to open a stream and it basically contains a header block fragment, such as:

      ```
      method:  GET
      path:    /index.html
      version: HTTP/2.0
      ```

    - **PRIORITY:** (type=0x02) Specifies the sender advised priority of the stream.
    - **RST_STREAM** (type=0x03) Sent for immediate termination of a stream purposes.
    - **SETTINGS:** (type=0x04) Its purpose is to send the desired configuration of the connection that needs to be established (Enable Push, Max Concurrent Streams, Max Frame Size, ...).
    - **PUSH_PROMISE:** (type=0x5) Notification that a sender wants to begin to send a stream.
    - **PING:** (type=0x06) Intends to check the state of a given connection.
    - **GOAWAY** (type=0x07) Initiate a shutdown connection or signal errors.
    - **WINDOW_UPDATE** (type=0x08) Used to implement flow control, as we'll describe later on.
    - **CONTINUATION** (type=0x09) Used to continue a sequence of header block fragments

Messages

A message is a sequence of frames representing a request or a response.

Streams

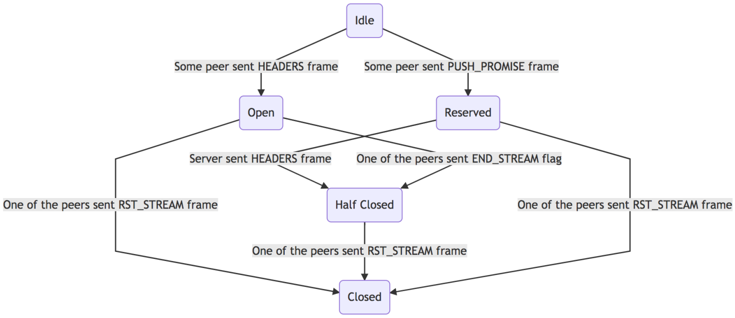
A stream is a bidirectional flow of bytes that in a given connection may carry one or more messages. Each stream is identified by an integer that will be written on the header of any frame.

Note that in a single TCP connection, several streams may be active concurrently at the same time.

Streams have lifecycles and they are represented by transitions among states. These are all the possible states:

- **Idle:** Initial state for any opened stream.
- **Open:** State where both peers may send or receive frames at any moment.
- **Reserved:** One of the peers, having the stream in Idle state, sent a `PUSH_PROMISE` frame in order for the server to begin to push messages to the client.
- **Half Closed:** One of the peers finished to send frames.
- **Closed:** Both peers agreed to terminate the connection.

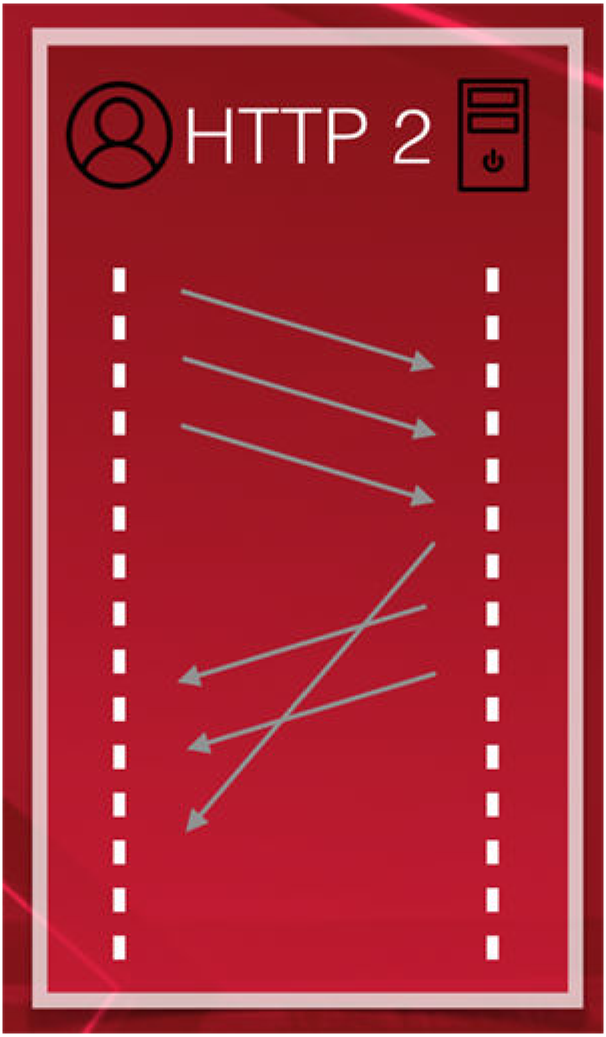
Here you can see the workflow:


Idle
Some peer sent HEADERS frame
Some peer sent PUSH_PROMISE frame
Open
Reserved
Server sent HEADERS frame
One of the peers sent END_STREAM flag
One of the peers sent RST_STREAM frame
Half Closed
One of the peers sent RST_STREAM frame
One of the peers sent RST_STREAM frame
Closed

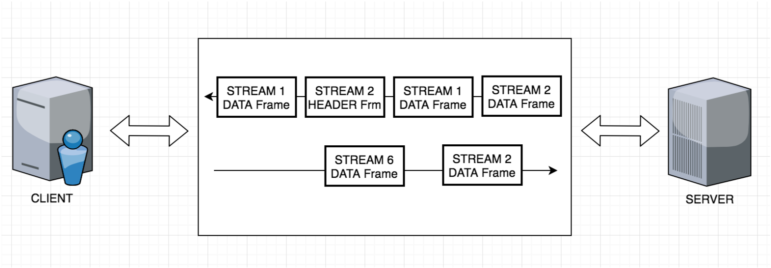
Multiplexing

As mentioned before, HTTP/1.x doesn't allow you to do multiple parallel requests in the same TCP connection. At most, with HTTP/1.1 you'll be able to do multiple requests, but the responses will need to be received in the same order, inducing Header-of-Line blocking.

HTTP/2 and its binary convention allows you to multiplex requests and responses in the same connection so you can take full advantage of the network's resources.



Here's how the frame transfer in a HTTP/2 connection looks like:



Once you give the possibility to HTTP/2 to multiplex all the frames among all the streams, some kind **prioritization** needs to take place, for the communication to carry out. There are two notions for prioritizing resources on the network:

- Dependency among streams: each stream is affected to its parent stream (or none). Of course, the parent will have priority over the son. The son won't have any resource until the parent has been sent.
- Each stream has an associated weight (1 to 256): in the case that two streams have the same parent, we'll prioritize in function of the weight. Two sibling streams with different weights never block one another. The weight number is used to calculate the percentage of the resources that we'll give to each sibling.

Flow control

HTTP/2 multiplexed frames unleash the potential to take full advantage of the network resources, but without a flow control you lose any sense of traffic congestion. Any peer sending data needs to know the ingestion capabilities of the receiver, otherwise the frames can be lost. If the receiver is busy doing other stuff, it needs to be able to communicate to the sender to slow down the cadence.

TCP protocol already considers flow control by communicating the `receive window` of each peer. This is the equivalent buffer size of each end point to hold incoming data. Each TCP ACK signal contains an updated `receive window` in function of the receiver availability.

The problem with TCP flow control, is that it doesn't have enough Application Level granularity. Multiple streams in the same TCP connection prevent the optimizing of the receiving flow for each pair of stream-applications.

So in addition to TCP flow control, HTTP/2 uses the `WINDOW_UPDATE` frame type (type=0x8) to advertise the stream-application `receive window` size.

Server push

Server push is the capability of HTTP/2 to initiate the transfer of data frames from the server side, instead of having to wait for the request of the client. This is obviously a big win, for instance when browsers need to download multiple resources from a webpage, where HTTP/1.x is needed to open one TCP connection for every one of them.

`PUSH_PROMISE` frames are used for this purpose. They are sent by the server to the client, and the client is free to accept or not, and even decide a bunch of settings via the `SETTINGS` frame on how the downloading of the resources will be done. The goal is that the client may have full control of the communication.

Headers

In HTTP/1.x headers are systematically sent in any request or response, in plain text. This is a big waste of bandwidth, that HTTP/2 improves by applying HPACK compression on the `HEADER` frames.

As said before, the purpose of HTTP/2 is to improve performance by reinventing the encoding, but keeping the syntax. This is to preserve the headers as they are, and it's just the way they will be transferred.

HPACK compression uses three methods:

- **Static Dictionary:** **61 commonly used headers** are predefined with default values, so that nothing is defined on the first request, and nothing is sent over the network.
- **Dynamic Dictionary:** if new headers are sent over the transmission, they are added into the dictionary in order to prevent sending them in further messages.
- **Huffman Encoding:** an algorithm for the compression of ASCII characters in the HTTP Headers context. Each character is associated to a bit code. The more frequent a character will be, the shorter its bit code will be. You normally obtain an average of 37% smaller compressed texts.

Load balancing 9

One of the common issues when it comes to scalability is how to balance the load all over the servers intended to ingest the incoming and growing traffic. Plenty of proxies are intended to do so for HTTP/1.x natively, but since HTTP/2 is a binary protocol, instead of plain text, they will need other plugins to achieve it.

In addition to proxy solutions, gRPC proposes client load balancing, where the clients somehow know about the backend configuration and they dispatch the messages directly to the servers in a proper and intelligent way.

Which solution should you adopt between client and proxy load balancing? Well, it depends on the context of your application. We'll try to figure out which are the advantages and disadvantages of each option and let you decide what is your optimal configuration.

Client load balancing

The main advantage of client load balancing is obviously its performance. No middle agents between client and servers makes it optimal for low latency constraints.

As you can imagine, however, client load balancing adds more complexity to the architecture. The client is much more sophisticated, since it needs to apply strategies to equilibrate the traffic (Round Robin or others).

The client will need to keep track on the health of each backend, in order to redirect traffic elsewhere if a backend malfunction is detected. This would be the **thick client** approach.

Another alternative would be to use the **Lookaside Load Balancer**, which is kind of a Zookeeper/Consul/Eureka who communicates the client that is the best backend server to communicate with. A good example on using Zookeeper as a Lookaside LB, using a Round Robin strategy in Java, would be on **https://github.com/makdharma/grpc-zookeeper-lb**.

Proxy load balancing

The main advantage of proxy load balancing is the simplicity of the client. It will only need a single endpoint to create a connection with. All of the workload problems, the security issues and the awareness of the health of every backend server, will be completely transparent to the client.

Another advantage is that the TLS certificates must be managed directly by the proxy. Backend servers don’t need to have secured connections, which makes the architecture much less complex.

In return, the latency will be increased and the throughput may be limited by the capabilities of the proxy.

There are two kind of proxies in function of the OSI level that they work on:

- **Transport Level:** this option is dumber, but easier to implement. This is done on the TCP level, where the proxy just checks that the socket is open, in order to know if the backend is up and running. This kind of proxy is very performant since there is no payload treatment. Client data is just copied to the backend connection.
- **Application Level:** when the proxy needs to be a little bit smarter in terms of workload decisioning, this is the best option, in detriment of the added latency. On the application level, the HTTP/2 protocol is parsed in order to inspect each request and make decisions on the fly.

NGINX example

NGINX needs to be compiled with http_ssl and http_v2 modules:

```
$ auto/configure --with-http_ssl_module --with-http_v2_module
```

Configuration should be as following:

```
http {
    server {
        listen 80 http2;

        access_log logs/access.log main;

        location / {
            grpc_pass grpc://localhost:50051;
        }
    }
}
```

Summary

You should now have a thorough understanding of the HTTP2 protocol and the important role it plays in gRPC.

The next chapter covers the techniques to implement so you can safely evolve your gRPC services without introducing breaking changes for your existing consumers.

Service evolution with gRPC 10

Using an IDL like protobuf greatly simplifies a lot of things when it comes to defining your RPCs. The amount of boilerplate code required, especially when working with multiple languages, is greatly reduced. New fields can be safely added without worrying about breaking existing functionality or client behavior. This is only to name a few.

There are a number of cases, however, that require special attention when evolving an existing service. Any time you're updating an existing implementation there are a set of constraints you need to work within. These include, but aren't necessary limited to, maintaining binary and source compatibility, encoding/wire compatibility, and of course behavior compatibility.

Let's take a look at each of these constraints and some of the details required to maintain a functioning service over time while making changes. If you'd like to jump straight to the code, the source code for this chapter is available at **https://github.com/backstopmedia/gRPC-book-example/tree/master/chapters/evolution**

At certain points, we'll be talking about protobuf wire formats. If you're interested in byte-level details about protobuf encoding for specific types you can check out the docs on encoding here: **https://developers.google.com/protocol-buffers/docs/encoding**.

Binary and source compatibility

When using the codegen tools for gRPC (`protoc`, `protoc-gen-grpc`, and related plugins) there are two interfaces you need to be concerned with: the ABI and the API.

- The application binary interface (ABI), as it relates to gRPC, refers to the binary interface of the tool that's being used to generate the protobuf code as well as any code that the generated code will rely on.

- The application programming interface (API) refers to the signatures of the methods we, as the consumer, can call the objects that are available for us, etc.

For the most part, you shouldn't have to worry too much about the ABI unless you're changing the major version of `protoc` being used to generate the code. If this is the case, you'll likely want to bump the version without any code changes first, just to make sure the ABI is compatible.

The API on the other hand, is something you do need to be concerned about every time you make changes to your protobuf files. Let's take a look at the following protobuf as an example.

Protocol Buffer

```
syntax = "proto3";

package practical_grpc.v1;

service MyService {
  rpc MyMethod(MyMethodRequest) returns (MyMethodResponse);
  rpc MyMethod2(MyMethodRequest2) returns
(MyMethodResponse2);
}

message MyMethodRequest {
}

message MyMethodResponse {
  string content = 1;
  int32 count    = 2;
}

...
```

Let's assume that this service is live and that you have a number of existing downstream consumers using it. Once your service is in production, you have to be careful about making changes that could affect existing clients.

Since there are clients out there making requests to the `MyMethod` RPC, it shouldn't come as a surprise that renaming or removing the method would break the API contract.

Changing a method's request or response message types will also change the method signatures, which again, results in breaking the API.

At this point you may be thinking that some languages are dynamic and can potentially handle this kind of change. While that isn't necessarily wrong, generally speaking, you should try to ensure that your services are able to be used by any of the supported gRPC languages. This is especially true when you're not supplying the client library.

Imagine you've taken the time to write a client library for an existing service and then it gets changed and your implementation is broken. If the maintainer says, oh, you should just use *[insert language here]*, you'd probably be a little upset.

Changing primitive types is also something that should be avoided. The temptation is to assume that upgrading types is safe. For example, a field defined as an `int32` should be fine to be turned into an `int64`. This can work out fine in some cases/languages, but not so well for others. In general, this should be avoided. Especially if you intend to support all gRPC languages.

To avoid the need for breaking changes in the future, there are a number of things you can do right at the beginning when you define the service. Let's take a look at a few of those techniques.

Versioning a Service

Adding a version number to either your package or your service name will allow you to simply define a new version when needed. When you make a new version of the service, there will be no effect on the client using the initial version. In the example above you can see that `.v1` was appended to the package name. This version will also be included in the generated code, which makes it possible for you to have more than one version of a service running at the same time.

Forgetting to do this initially isn't such a big deal, however, since you can always add a version identifier to the next version of the service when the time comes.

Define custom request and response objects

In the example above, you may have noticed that `MyMethodRequest` is an empty message object and wondered, why not just use `google.protobuf.Empty`?

The reason for this is to avoid having to write a new version of the method if you decided to, say, take an optional parameter in the future. If you used `google.protobuf.Empty`, you'd need to define a new RPC to avoid breaking the API. By using a custom request message you can just add a new field to `MyMethodRequest`, which allows new clients to use the field and old clients to ignore it.

The same applies to the response object. You can send back a naked primitive or message type, but by wrapping it in a custom response message you maintain the ability to extend it in the future without any breaking changes.

When making changes to protobuf definitions, keeping ABI and API compatibility is critical in maintaining functional downstream consumers.

Knowing that you can't rename or remove methods or messages, change method signatures, nor change primitive types without breaking the API, you're left with the ability to add services, methods, messages, and fields.

Maintaining wire compatibility

Under the hood, all client-server communication is issued across a multiplexed HTTP/2 connection (a channel in gRPC parlance). Messages are encoded using the protobuf wire format, and the particular service and method are extracted from headers sent along with the request.

In gRPC, calls are only distinguishable by their service and method names. For example, given the following protobuf, the target for a call would be *practical_grpc.v1.SomeService/SomeMethod*.

Protocol Buffer

```
syntax = "proto3";

package practical_grpc.v1;

service SomeService {
  rpc SomeMethod(SomeMethodRequest) returns (stream
SomeMethodResponse);
}

...
```

Notice how the method's request and response types are not included here. These, along with the cardinality of the method signature (unary vs streaming) are implicit in the call.

This means, at least in theory, that you *could* change a method from unary to streaming (by prepending a message with *stream*), or change the name of a message type, assuming it had the same structure. However, as discussed in the previous section on ABI/API compatibility, these changes would likely introduce breaking changes for your existing clients, and should be avoided.

Implicit versus explicit values on the wire

Message names in gRPC calls are implicit. That is to say, on the wire you won't find the message name as part of the RPC call, except in some generic cases like Any for example.

Field names are also implicit, except when they're referenced directly by the JSON name or a FieldMask. This also breaks down when they're used or referenced in metadata for example.

Field tags on the other hand are explicit. These are used by the underlying system to construct a message that will be sent to or from the server. Tags cannot be changed without breaking existing clients, and their reuse should be avoided.

Finally, some types are equivalent on the wire. For example, `int32` and `int64` are the same on the wire. This doesn't mean they're treated equally by the client libraries in all languages. You should still be very cautious about upcasts in case a particular client can't handle the change correctly.

As you can see, almost all of the implicit values come with a set of conditions that define cases where they might not be implicit. This can make it difficult to reason about what can safely change between your releases. You should avoid the temptation to exploit these implicit values to avoid potentially breaking API changes for our clients.

While a lot of the parts of a call are (sometimes) implicit, including message names and field names, and some types are equivalent on the wire, like `int32` and `int64`, you still can't safely and reliably change things without breaking API changes for your existing clients.

This means that you're still limited to adding services, methods, messages, and fields.

Behavior compatibility

If you follow the practice of only adding things, maintaining ABI/API compatibility and wire compatibility is fairly straightforward. However, this alone isn't sufficient to ensure your existing client's expected behavior is maintained.

There are a number of things you need to be aware of to ensure that the behavior of your service doesn't change when you add new fields for example. Let's take a look at a few common scenarios.

Default values for new fields

Any new fields added to protobuf messages will be given a *default* value that's specific to the type. For example, a new `int32` field will have a value of `0`, a string will have `""`, and a message field will be `nil` (or the language equivalent).

For a full list of the default values for each field type, see the protobuf docs at **https://developers.google.com/protocol-buffers/docs/proto3#default**.

This is the standard behavior for proto3. Working against the protocol can lead to it's own set of challenges and is generally not recommended. There are times, however, when you might need to check for presence and not just the default value.

If it's necessary, you can box primitive types so that you can check for presence. This is accomplished by using *wrappers.proto* from the Well-Known Types. There's a wrapper for each of the defined primitive values.

For example:

Protocol Buffer

```
syntax = "proto3";
import "google/protobuf/wrapper.proto";

package practical_grpc.v1;

message SomeMessage {
  // allows us to check for `nil` instead of just the
default value
  google.protobuf.StringValue value = 1;
}
```

This relies on the fact that the default value for messages is `nil`. This is equivalent to boxing in other programming languages you may have seen, where primitive types are wrapped by an object so you can do `nil` checks.

Field clearing

Since new fields will receive a default value, you need to be conscious of that in the service implementation. Otherwise you could accidentally override *good* values (i.e. ones that have been explicitly set), with *default* values depending on which code the calling client is using. This process is referred to as field clearing.

Let's look at an example to see how this could happen. Let's assume you have the following protobuf.

Protocol Buffer

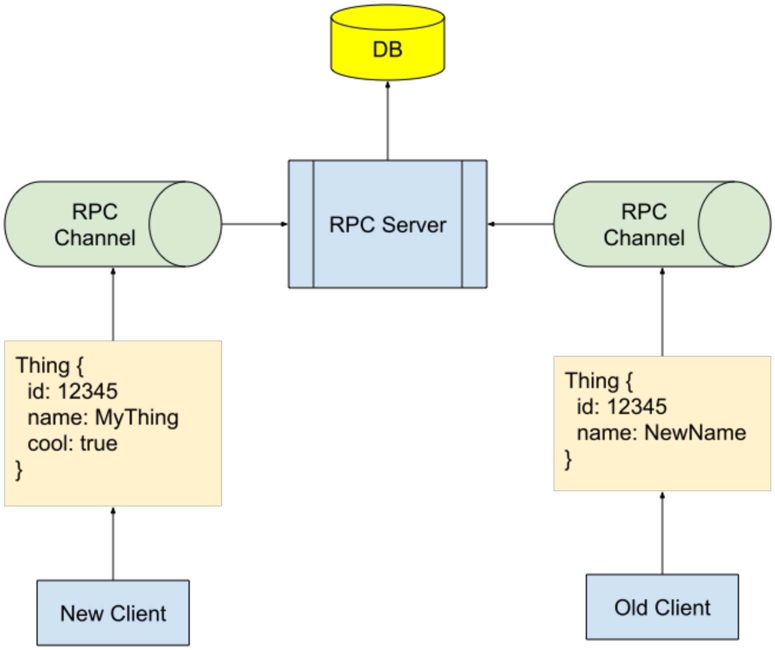
```
syntax = "proto3";

package practical_grpc.v1;

service MyService {
  rpc UpdateThing(UpdateThingRequest) returns
(UpdateThingResponse);
}

message Thing {
  int64 id     = 1;
  string name = 2;
}

message UpdateThingRequest {
  Thing thing = 1;
}

message UpdateThingResponse {
  Thing thing = 1;
}
```

You find out that you need to track whether a thing is cool (by some definition of cool). So you add a boolean field to the `Thing` message like this:

```
...
message Thing {
  int64 id     = 1;
  string name = 2;
  bool cool    = 3;
}
...
```

You then deploy the new version of the service that sets the value appropriately based on the request object.

Now, assume a new client updates a thing to be *cool* by issuing an RPC to the `UpdateThing` method. An existing client that does not know about that new field later updates the same `Thing` object.



What is the value of the `cool` field after the second call? You guessed it, `false`. This is clearly not the intended behavior. Let's take a look at a few different options for handling this scenario.

FIELD MASKS

Field masks define a set of fields to be included in the request. They allow the caller to specify which fields should be included in the call. To use them, you'll need to add a mask attribute to the appropriate message. To get a better idea of how this works, add a field mask to your `UpdateThingRequest` message.

Protocol Buffer

```
syntax = "proto3";
import "google/protobuf/field_mask.proto";
...
...

message UpdateThingRequest {
  Thing thing = 1;

  // allow callers to specify which fields should be updated
  google.protobuf.FieldMask mask = 2;
}
```

With the mask in place, you can now specify which fields should be included when performing an update.

Ruby

```
#!/usr/bin/env ruby
# frozen_string_literal: true

$LOAD_PATH.unshift('.')
require 'proto/thing_services_pb'

Stub    = Practical::Grpc::V1::MyService::Stub
Request = Practical::Grpc::V1::UpdateThingRequest
Thing   = Practical::Grpc::V1::Thing

begin
  stub    = Stub.new('localhost:
8080', :this_channel_is_insecure)
  thing   = Thing.new(id: 8_929_212, name: 'New Name')

  request = Request.new(
    thing: thing,
    mask: Google::Protobuf::FieldMask.new(paths: %w[thing.id
thing.name])
  )

  puts stub.update_thing(request)
rescue StandardError => e
  code = e.respond_to?(:code) ? e.code : 'Unknown'
  puts "Code: #{code}, Type: '#{e.class}', Message:
#{e.message}"
end
```

With the caller specifying a field mask, you'll need to make use of that on the server side. The implementation should look something like this:

Go

```
func (s *MyService) UpdateThing(ctx context.Context, r
*pb.UpdateThingRequest) (*pb.UpdateThingResponse, error) {
    if r.Mask != nil {
        // Update only fields in r.Mask.Paths
        log.Print(r.Mask.Paths)
    }

    return &pb.UpdateThingResponse{Thing: r.Thing}, nil
}
```

While this approach does solve the problem with field clearing, it requires you to use field masks everywhere upfront. Once you implement an RPC method, you would need to include a field mask for each request object to protect against future changes.

BOXING

Another option is to box the new value. This allows you to do a *presence* check and only update the value when it's supplied.

Protocol Buffer

```
syntax = "proto3";
import "google/protobuf/wrappers.proto";

...
...
message Thing {
  Int64 id    = 1;
  String name = 2;

  // wrap for presence checking
  google.protobuf.BoolValue cool = 3;
}
```

This is a little easier, but also requires all new fields to be wrapped. The fields themselves also need wrapping/unwrapping on the client side, which could prove to be unnecessarily verbose.

LEVERAGING UNKNOWN FIELD SUPPORT

Finally, Protobuf 3.5 re-added support for unknown fields. Messages will hold on to fields that are returned from services in an internal map and ensure they are sent back to the server in a normal get, modify, update scenario. This is a feature that was initially removed from Proto3, but added back specifically to handle this type of scenario.

When relying on the unknown field support, keep in mind that this only works for cases when the client fetches an object, modifies the same object, and sends the result back to the server. When creating new objects in the client, unknown fields are simply unknown unknowns.

Dealing with errors

Knowing that a service will evolve, it can be useful to return descriptive error messages so clients can decide how to handle different error cases.

Typically when an error occurs in a service you'll have a status code and a message to let the caller know what happened. On the server side, that might look something like this:

Go

```go
func (svc *Service) DoSomething(ctx context.Context, r
*pb.Message) (*pb.Message, error) {
    res, err := foo.Bar(r.Baz)
    if err != nil {
        return nil, grpc.Errorf(codes.NotFound, err.Error())
    }
    ...
}
```

If there are other error details, they are typically put in the response's metadata. This works as expected, but places the burden of determining which metadata fields are associated with errors on the caller.

To avoid this you can make use of the *google.rpc.Status* message. It looks like this:

Protocol Buffer

```protobuf
message Status {
  int32 code     = 1;
  string message = 2;

  repeated google.protobuf.Any details = 3;
}
```

Notice how the last field is a repeated *Any* field. This allows you to pass any messages you need to send to the caller in a single response without using metadata, at least directly. Clients can initially be written to handle status errors by checking the code and/or the message, as well as enumerating the details.

Let's take a look at an example of this. To start, let's add the following message to *thing.proto*.

Protocol Buffer

```
...
message ThingNotFoundError {
  int64 requested_id = 1;
  string reason       = 2;
}
```

Our service can now return a `Status` error message when a thing is not found and include our custom message in the details. Let's update our service to return a status so we can test it out. To do that, use the following for the implementation of `UpdateThing`:

Go

```
func (s *MyService) UpdateThing(ctx context.Context, r
*pb.UpdateThingRequest) (*pb.UpdateThingResponse, error) {
    // thing, err := findThingByID(r.Thing.Id)
    // we're assuming this error was returned
    err := errors.New("Thing not found")
    if err != nil {
        statusErr := status.New(codes.NotFound, err.Error())
        statusErr, _ =
statusErr.WithDetails(&pb.ThingNotFoundError{
            RequestedId: r.Thing.Id,
            Reason:      "Thing not found, or you do not
have access to it.",
        })

        return nil, statusErr.Err()
    }

    return &pb.UpdateThingResponse{Thing: r.Thing}, nil
}
```

Here you've returned the error from the `Status` object. This includes a code, message, and a custom message in the `details` field. Each message in `details` will be included in the trailing metadata (as `grpc-status-details-bin`) for the response. You'll see how to use that when you implement the client.

Next, let's update the client handle the case when a `Thing` isn't found by informing the user of the reason for the error.

Ruby

```
#!/usr/bin/env ruby
# frozen_string_literal: true

$LOAD_PATH.unshift('.')
require 'grpc/google_rpc_status_utils'
require 'proto/thing_services_pb'

Stub          = Practical::Grpc::V1::MyService::Stub
Request       = Practical::Grpc::V1::UpdateThingRequest
Thing         = Practical::Grpc::V1::Thing
NotFoundError = Practical::Grpc::V1::ThingNotFoundError

begin
  stub  = Stub.new('localhost:
8080', :this_channel_is_insecure)
  thing = Thing.new(id: 8_929_212, name: 'New Name')

  request = Request.new(
    thing: thing,
    mask: Google::Protobuf::FieldMask.new(paths: %w[thing.id
thing.name])
  )

  puts stub.update_thing(request)
rescue StandardError => e
  code = e.respond_to?(:code) ? e.code : 'Unknown'

  if code == GRPC::Core::StatusCodes::NOT_FOUND
    status =
GRPC::GoogleRpcStatusUtils.extract_google_rpc_status(e.to_status)
    puts
NotFoundError.decode(status.details.first.value).reason
    exit 1
  end

  puts "Code: #{code}, Type: '#{e.class}', Message:
#{e.message}"
end
```

This code checks the status code returned by the RPC call. If it’s a NOT_FOUND result, convert the error to a Status protobuf and decode the details into a custom error message (NotFoundError in this case).

The extract_google_rpc_status helper method works by reading the grpc-status-details-bin field from the trailing metadata for the error’s status. This value is included automatically by gRPC when the error returned is a Status error.

In the example above you decode the the error's status, pull the first object from details, and decode that into our custom message type so you can show the reason to the user. This allows for slightly nicer error handling.

While this example is fairly simple, you can imagine being able to add different handling based on the reason in the response, for example. If you later add a new type of error message, existing clients are still able to fall back to the default error handling.

Summary

In this chapter we took a look at some of the common edge cases that come up when evolving gRPC services over time. We dug into ABI/API compatibility concerns, maintaining wire format equivalence, and ensuring that we don't unintentionally change the behavior of our API. We also took a look at common gotchas like field clearing and proper error handling.

Using the techniques discussed here you can safely evolve your services without introducing breaking changes for existing consumers.

In the next chapter, we'll take a look at how to extend your gRPC services with things like custom options and `protoc` plugins.

Extending gRPC services 11

In this chapter we're going to discuss how gRPC can be extended through additional tooling. For example, you may want to expose your gRPC server as a JSON API as well. Perhaps you want to write your own generator for `.proto` files. All of these are possible with the extensibility built into gRPC and protobufs.

In this chapter we're going to cover:

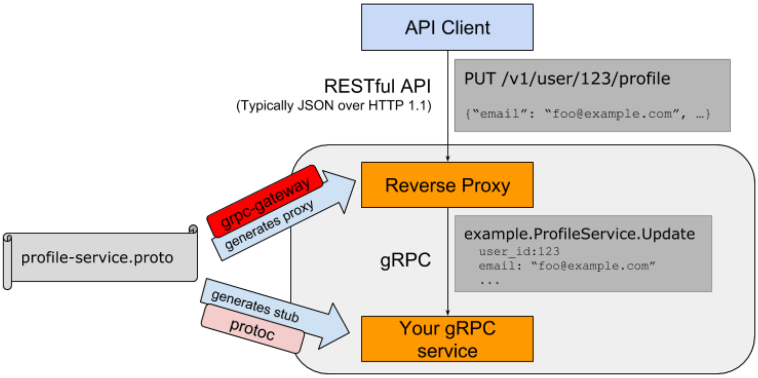
- JSON gateways to gRPC servers using the grpc-gateway tool
- Building your own protoc generator
- Adding custom options to messages, fields, services, and RPCs

These tools enable engineers to quickly add on business critical functionality that gRPC itself does not offer. We'll go over some examples of how they can be used and some drawbacks.

gRPC Gateway

The web over the last few years has been consistently moving to a Frontend > Backend model where the backend is nothing more than a JSON API that JavaScript in the browser uses to render the page. Since gRPC relies on HTTP/2 as the transport, this means browsers can't hit a gRPC server directly.

Since gRPC uses well defined messages and services with the protobuf definition language, you can easily generate code from these definitions. Using a tool called "grpc-gateway" you can actually make a fully functional JSON server that forwards requests to a running gRPC server. It handles the transcoding of the provided JSON to gRPC and vise versa.



From: `https://github.com/grpc-ecosystem/grpc-gateway`

Starting on the left, we define our protobufs and then generate a reverse proxy from them using the grpc-gateway tool. The tool will generate all of the endpoints in the protobuf's "Service" definitions. From here, API clients (such as AJAX/fetch frontends) can make HTTP requests to the proxy.

The proxy deserializes the JSON and then immediately serializes it into it's protobuf counterpart. From there, it uses a connection to the gRPC server to make the equivalent request. Once the server responds, the flow continues but in reverse. The flow deserializes the protobuf response, serializes to JSON, and responds to the original client.

This workflow is incredibly powerful when you want to use gRPC for internal calls to services, but expose certain components to your outside users.

To get started with this tool, your protobuf definition needs some simple modifications to allow the grpc-gateway tool to know how to expose service endpoints. This process involves using the `google.protobuf.Message` annotation to signal this configuration.

Let's create a simple endpoint that declares an HTTP route as well using an annotation message.

```
syntax = "proto3";

import "google/api/annotations.proto";

package sfapi.v1;
option go_package = "proto";

service Starfriends {
  // Get a list of Films
  rpc ListFilms(ListFilmsRequest) returns
(ListFilmsResponse) {
    option (google.api.http) = {
      get: "/sfapi/v1/films"
    };
  }
}
```

In this example we're using the "option" keyword of the definition to indicate an annotation value for this service endpoint. We're then able to set fields on the `HttpRule` message definition for the HTTP method this endpoint maps to.

Once we've added this annotation and installed the `grpc-gateway` tool, we can instruct the `protoc` compiler to use it and generate our JSON gateway code like so:

```
$ protoc -I /usr/local/include -I . \
  -I $GOPATH/src \
  -I $GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/
third_party/googleapis \
  --grpc-gateway_out=logtostderr=true:. \
  ./sfapi.proto
```

Let's break down what this command is doing. On the first line, we're invoking the `protoc` command and including a few paths for it to load additional proto definitions. We're then instructing it to use the `grpc-gateway` tool to generate our Go source for a JSON gateway.

The `--grpc-gateway_out=logtostderr=true:.` flag is including additional flags to the underlying generator as well, where the `:.` is indicating to place the generated files in the current directory. To see a list of all of the flags available by the generator, you can run `protoc-gen-grpc-gateway --help`.

Mapping URL params to request fields

You can also define how the URL path of an endpoint maps to the requests message. For example, our definition for retrieving a single film looks like the following:

```
message GetFilmRequest {
  string id = 1;
}
```

We can annotate an endpoint to use the path portion of an HTTP request to fill in the `id` field of this message:

```
rpc GetFilm(GetFilmRequest) returns (GetFilmResponse) {
  option (google.api.http) = {
    get: "/sfapi/v1/films/{id}"
  };
}
```

This allows HTTP clients to retrieve a film by requesting a URL similar to GET `/sfapi/v1/films/S1vZy63c4oz`. Using pseudocode, the grpc-gateway will perform the following logic:

```
film_id = request.path.params["id"]
request = GetFilmRequest{id: film_id}
grpc_upstream.get_film(request)
```

The grpc-gateway project will also handle common failure scenarios as well. For example if we attempt to hit our example server with a film ID that does not exist, we get the proper http response code of “404 Not Found”.

```
$ curl -i  http://localhost:8081/sfapi/v1/films/bunk
HTTP/1.1 404 Not Found
Content-Type: application/json
Trailer: Grpc-Trailer-Content-Type
Date: Sat, 28 Apr 2018 19:50:32 GMT
Transfer-Encoding: chunked

{"error":"film with id 'bunk' not found","code":5}
```

The gRPC gateway will map all gRPC codes to relevant HTTP status codes. Here is the mapping for this conversion:

gRPC	HTTP
OK	OK
Canceled	RequestTimeout
Unknown	InternalServerError
InvalidArgument	BadRequest
DeadlineExceeded	GatewayTimeout
NotFound	NotFound

gRPC	HTTP
AlreadyExists	Conflict
PermissionDenied	Forbidden
Unauthenticated	Unauthorized
ResourceExhausted	TooManyRequests
FailedPrecondition	BadRequest
Aborted	Conflict
OutOfRange	BadRequest
Unimplemented	NotImplemented
Internal	InternalServerError
Unavailable	ServiceUnavailable
DataLoss	InternalServerError

The gRPC Gateway project gives you the ability to use the body portion of a request to map into a protobuf request message. This means if you have endpoints where you want to accept a “POST” request with a body of JSON to say, create a resource, then you can create an annotation to support this as well on your service definition.

```
syntax = "proto3";
package example;

import "google/api/annotations.proto";
import "google/rpc/status.proto";

message CreateAccountRequest {
  string email = 1;
  string password = 2;
  string name = 3;
}

service Accounts {
  rpc CreateAccount(CreateAccountRequest) returns
(google.rpc.Status) {
    option (google.api.http) = {
      post: "/api/v1/accounts"
      body: "*"
    };
    }
  }
```

In this example we're defining a message CreateAccountRequest that contains the fields necessary to register a new account. We then create an HTTP rule for our CreateAccount endpoint that accepts a body that gets mapped to the request message definition. In order to create an account against this endpoint now, we can send a request that looks like:

```
POST /api/v1/locations HTTP/1.1
Host: localhost:8081/api/v1/accounts
Content-Type: application/json

{"email":"bobross@happytrees.com", "password": "mistakes" }
```

In this scenario, our grpc-gateway will deserialize the JSON in our HTTP request body, then map the values into our CreateAccountRequest message and continue forwarding that to our upstream gRPC server.

ADDITIONAL BINDINGS

There are scenarios where you may want additional HTTP routes to map to the same gRPC endpoint. In this scenario, you may define additional routes on the annotation using the additional_bindings field. For example, if we want our ListFilms endpoint to also include a route for /sfapi/v1/all_films, we can specify it by doing:

```
service Starfriends {
  // Get a list of Films
  rpc ListFilms(ListFilmsRequest) returns
(ListFilmsResponse) {
    option (google.api.http) = {
      get: "/sfapi/v1/films"
      additional_bindings {
        get: "/sfapi/v1/all_films"
      }
    };
  }
}
```

Our service now responds to a GET request for the path /sfapi/v1/films and /sfapi/v1/all_films and routes them to the ListFilms

RPC endpoint in our gRPC server. To add more than one additional route, you may specify the `additional_bindings` stanza multiple times as it's a repeated field for the `HttpRule` message.

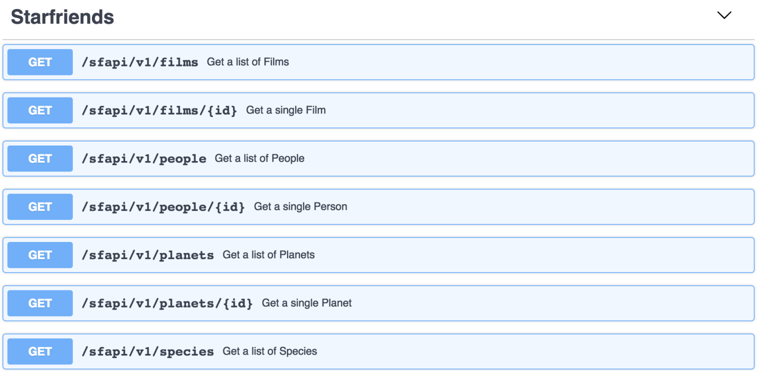
Swagger output

The GRPC Gateway project also ships with a separate `protoc` generator for Swagger output as well. This is very useful in the cases where you'd like to generate JSON clients from a Swagger file, expose your API endpoints for client discovery, or automatically create documentation for your public API. To get started with this optional tool, you can use a similar command to when we generated the JSON Gateway code:

```
$ protoc -I/usr/local/include -I. \
  -I$GOPATH/src \
  -I$GOPATH/src/github.com/grpc-ecosystem/grpc-gateway/
third_party/googleapis \
  --swagger_out=logtostderr=true:. \
  ./sfapi.proto
```

Just like our last example, we're including a few paths for `.proto` files to be loaded and then including a `--swagger_out` option that will generate a Swagger JSON file in the current directory we're in.

You can then upload this generated file to **https://editor.swagger.io** to get a nice view of all of your endpoints and model definitions. For example, here is a nicely generated page of our endpoints.



Starfriends

GET /sfapi/v1/films Get a list of Films

GET /sfapi/v1/films/{id} Get a single Film

GET /sfapi/v1/people Get a list of People

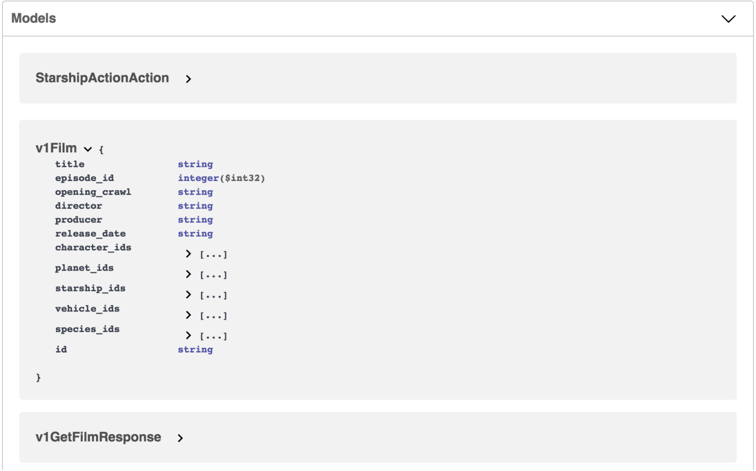
GET /sfapi/v1/people/{id} Get a single Person

GET /sfapi/v1/planets Get a list of Planets

GET /sfapi/v1/planets/{id} Get a single Planet

GET /sfapi/v1/species Get a list of Species

The Swagger file also includes your message definitions, which also translates well:



Models

StarshipActionAction >

```
v1Film {
    title              string
    episode_id         integer($int32)
    opening_crawl      string
    director           string
    producer           string
    release_date       string
    character_ids        > [...]
    planet_ids           > [...]
    starship_ids         > [...]
    vehicle_ids          > [...]
    species_ids          > [...]
    id                 string
}
```

v1GetFilmResponse >

The Swagger model also allows you to easily generate HTTP clients for your gateway in several languages. To get started, checkout **https://swagger.io/swagger-codegen/**.

Custom options

In protobuf, you can customize the IDL, with the IDL itself, by adding your own options. These can be really useful for things that aren't directly related to the model, but would be helpful to know about, like settings for your parser, authentication, or anything else. You can read more about extensions **here**. You can have a look at **protobuf/descriptor.proto** to get a feel for all the things you can override, and there are quite a few.

Here is an example that adds some authentication-options to a field:

```
syntax = "proto2";

import "google/protobuf/descriptor.proto";

package myapp;

message AuthFieldOptions {
  enum RoleRequerd {
    AUTHENTICATED = 0;
    PAID = 1;
    EDITOR = 2;
    ADMIN = 3;
  }
  optional RoleRequerd requiredRole = 1;
}

extend google.protobuf.FieldOptions {
  optional AuthFieldOptions options = 20001;
}
```

This adds a field option `myapp.requiredRole`, based on the package and option-name. `proto2` is used for maximum-compatibility. All extension-numbers should be unique for a field-type. We chose `20001`, because you need to choose an extension-number that is in the range defined in **protobuf/descriptor.proto**, which is `1000 to max`, so we picked a high one, that you can use as a kind of numeric namespace (all `myapp` could be after `20000`.)

Here is how you would use that in another proto definition:

```
  syntax = "proto3";
  package myapp;

  message CoolThing {
    string secrets = 1 [ requiredRole = ADMIN ];
  }
```

In your backend code, you can check the option value before you give up the data in `secrets`. If you need to specify the package, you can use parentheses:

```
  message CoolThing {
    string secrets = 1 [ (coolcompany.myapp.v1).requiredRole =
  ADMIN ];
  }
```

In general it's a good idea to namespace your options with `package` and pick a high `extend` number to avoid clashes with Google's built-in stuff, or any other libraries you might be using. It can also help you find where things are defined.

Protoc plugins

protoc is a general swiss-army-knife for manipulating proto files, and encoding/decoding binary messages. It can be used with your proto to generate docs, any kind of code, linting, and anything else you can think of. Protoc looks in your path for an executable file called `protoc-gen-NAME` and reads options for command-line: `--NAME_out` and `--NAME_opt`. Some of them use a `OUT_PATH:OPTIONS` format for `--NAME_out`.

There are some really great extended plugins for generating things not included in the original protoc. Have a look at **protoc-gen-doc**, which is great for generating markdown, HTML, docbook, and JSON from your proto files. **protoc-gen-lint** is perfect for alerting you to any problems with your proto files, and making your files conform to **Google's Style Guide**.

Install these like this:

```
go get -u github.com/pseudomuto/protoc-gen-doc/cmd/protoc-
gen-doc
go get github.com/ckaznocha/protoc-gen-lint
```

Now, you can use them together if you like:

```
protoc -I ../../../proto/ ../../../proto/sfapi.proto --
doc_out=. --doc_opt=markdown,api.md --lint_out=.
```

Here is a mega-example that will lint, generate docs, and generate pre-computed JavaScript, C sharp, Ruby, and Python interfaces:

```
protoc -I ../../../proto/ ../../../proto/sfapi.proto --
doc_out=. --doc_opt=markdown,api.md --lint_out=. --
js_out=generated --csharp_out=generated --ruby_out=generated
--python_out=generated
```

MAKING YOUR OWN

Protoc plugins have a very simple interface. They should be an executable file, and take protobuf binary messages in the format of `CodeGeneratorRequest` on `stdin`, and return `CodeGeneratorResponse` messages, on `stdout`. If you are familiar with Node.js, an easy way to get started making your own is to use **node-protoc-plugin**. There are som examples in that repo, including a postman workspace generator, if you're using grpcgateway, and a few simple examples that output an express server and log JSON. There is an example in **gRPC-book-example** that uses handlebars to create custom markdown documentation. It's easiest to put the plugin in your path, with the correct name for protoc to find it, but you can use the `--plugin=` flag to use plugins not in your path, and `--plugin=NAME=PATH` to map to a plugin that has a different name than expected.

Here is a quick-start from a Node.js perspective:

```
mkdir myproject
cd myproject
npm init -y
npm i protoc-plugin
touch src/protoc-gen-mydocs
chmod +x src/protoc-gen-mydocs
```

Now, let’s make a plugin that uses a template to make some markdown in `src/protoc-gen-mydocs`:

JavaScript

```
#!/usr/bin/env node
const protocPlugin = require('protoc-plugin')
const Handlebars = require('handlebars')
const { basename } = require('path')
const { readFileSync } = require('fs')

Handlebars.registerHelper({
  findCommentByPath: protocPlugin.findCommentByPath
})

const tpl = readFileSync(`${__dirname}/mydocs.handlebars`)

protocPlugin(protos => protos.map(proto => ({
  name: basename(protos[0].name, '.proto') + '.md',
  content: tpl(proto)
}))
)
```

This will apply the template `src/mydocs.handlebars` to every proto, and output markdown files with the same name in your `--mydocs_out=` location. `findCommentByPath` is exposed as a helper, so it can be used in the template to get comments for things. Handlebars is used because mustache-templates are pretty universal, and they lend themselves nicely to any sort of string-output, but you can use any template-language that will output what you want (for example React works really well for HTML.)

Make sure that anything you want to output to console goes to `stderr` (use `console.error()`) because protoc is expecting only binary protobuf on `stdout`.

This is a simple version. In **gRPC-book-example** there is a more complete documentation example that has helpers to make tables for messages and services, and understands gRPC gateway annotations. In your own system, you can customize it as much as needed.

Anything in the options will be put in the `options` field for the `message`/`service`/`field`/etc., including custom options (like `myapp.requiredRole` defined above.) This illustrates how useful it can be to make your own custom docs plugin that understands all of your custom options and can help users work with the model. You can explain to the user how your auth works, or describe how to get to the gRPC gateway described in your annotations, or any other custom info you want to provide. You can use the same technique to generate things other than documentation, like server or client code in your preferred system.

Summary

With this chapter complete you should now be on your way to extending gRPC through additional tooling in all sorts of useful ways.

In the next and final chapter you will learn how to debug your gRPC server.

Debugging gRPC 12

In this chapter, you'll learn how to interact with and debug a gRPC server.

An issue with gRPC and protobuf, if you are used to REST-like with JSON, is that it can be a bit more complicated to debug and troubleshoot, and the tools are generally more specialized. This section will serve to introduce you to the most useful tools for quickly working with this ecosystem.

CLI tools

Here are some useful tools that you can use to interact with a running gRPC server on your local machine:

- **grpcnode** - CLI tool for quickly making servers and client, dynamically, in JavaScript
- **grpcc** - REPL gRPC command-line client
- **grpc_cli** - gRPC CLI tool
- **Evans** - Expressive universal gRPC (CLI) client
- **grpcurl** - Like cURL, but for gRPC: Command-line tool for interacting with gRPC servers
- **danby** - A grpc proxy for the browser
- **docker-protoc** - Dockerized protoc, grpc-gateway, and grpc_cli commands bundled with Google API libraries.
- **prototool** Useful "Swiss Army Knife" for processing proto files

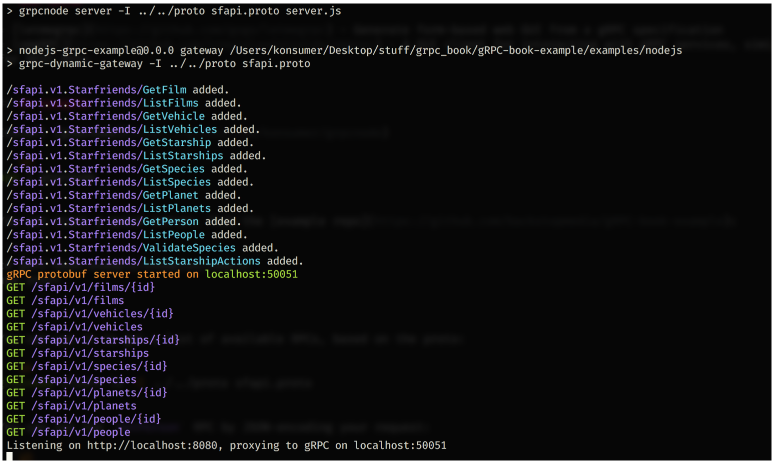
An example

Install **grpcnode**:

```
npm i -g grpcnode
```

You can start a quick “Star Friends” server, from the **example repo**:

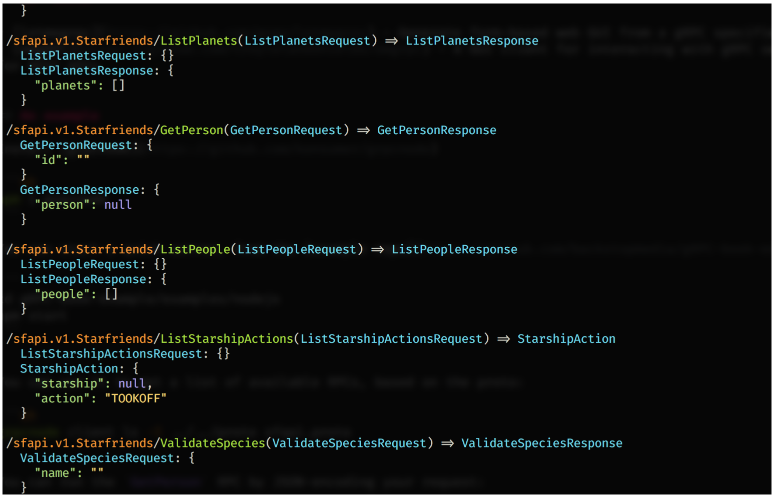
```
git clone https://github.com/backstopmedia/gRPC-book-example.git
cd gRPC-book-example/examples/nodejs
npm i
npm start
```



```
> grpcnode server -I ../../proto sfapi.proto server.js

> nodejs-grpc-example@0.0.0 gateway /Users/konsumer/Desktop/stuff/grpc_book/gRPC-book-example/examples/nodejs
> grpc-dynamic-gateway -I ../../proto sfapi.proto

/sfapi.v1.Starfriends/GetFilm added.
/sfapi.v1.Starfriends/ListFilms added.
/sfapi.v1.Starfriends/GetVehicle added.
/sfapi.v1.Starfriends/ListVehicles added.
/sfapi.v1.Starfriends/GetStarship added.
/sfapi.v1.Starfriends/ListStarships added.
/sfapi.v1.Starfriends/GetSpecies added.
/sfapi.v1.Starfriends/ListSpecies added.
/sfapi.v1.Starfriends/GetPlanet added.
/sfapi.v1.Starfriends/ListPlanets added.
/sfapi.v1.Starfriends/GetPerson added.
/sfapi.v1.Starfriends/ListPeople added.
/sfapi.v1.Starfriends/ValidateSpecies added.
/sfapi.v1.Starfriends/ListStarshipActions added.
gRPC protobuf server started on localhost:50051
GET /sfapi/v1/films/{id}
GET /sfapi/v1/films
GET /sfapi/v1/vehicles/{id}
GET /sfapi/v1/vehicles
GET /sfapi/v1/starships/{id}
GET /sfapi/v1/starships
GET /sfapi/v1/species/{id}
GET /sfapi/v1/species
GET /sfapi/v1/planets/{id}
GET /sfapi/v1/planets
GET /sfapi/v1/people/{id}
GET /sfapi/v1/people
Listening on http://localhost:8080, proxying to gRPC on localhost:50051
```

You can use it to get a list of available RPCs, based on the proto:

```
grpcnode client ls -I ../../proto sfapi.proto
```

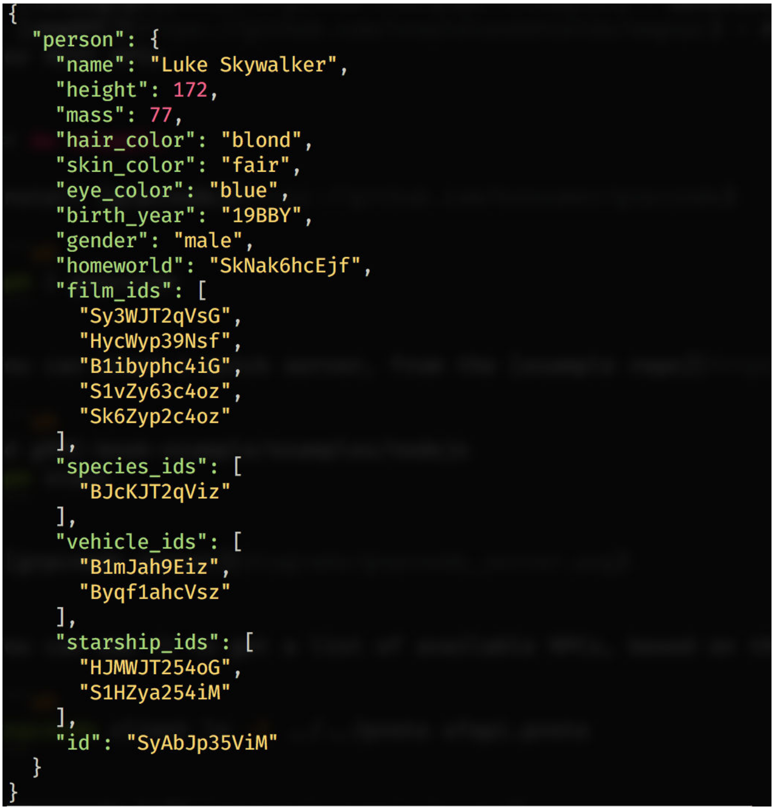


```
  }

/sfapi.v1.Starfriends/ListPlanets(ListPlanetsRequest) ⇒ ListPlanetsResponse
  ListPlanetsRequest: {}
  ListPlanetsResponse: {
    "planets": []
  }

/sfapi.v1.Starfriends/GetPerson(GetPersonRequest) ⇒ GetPersonResponse
  GetPersonRequest: {
    "id": ""
  }
  GetPersonResponse: {
    "person": null
  }

/sfapi.v1.Starfriends/ListPeople(ListPeopleRequest) ⇒ ListPeopleResponse
  ListPeopleRequest: {}
  ListPeopleResponse: {
    "people": []
  }

/sfapi.v1.Starfriends/ListStarshipActions(ListStarshipActionsRequest) ⇒ StarshipAction
  ListStarshipActionsRequest: {}
  StarshipAction: {
    "starship": null,
    "action": "TOOKOFF"
  }

/sfapi.v1.Starfriends/ValidateSpecies(ValidateSpeciesRequest) ⇒ ValidateSpeciesResponse
  ValidateSpeciesRequest: {
    "name": ""
  }
```

You can run the `GetPerson` RPC by JSON-encoding your request:

```
grpcnode client run -I ../../proto sfapi.proto -c '/
sfapi.v1.Starfriends/GetPerson({"id":"SyAbJp35ViM"})'
```

There is also a HTTP/JSON gateway running **https://github.com/konsumer/grpc-dynamic-gateway**, so you can use your favorite way of messing with JSON-servers (curl, postman, etc.).



```
{
  "person": {
    "name": "Luke Skywalker",
    "height": 172,
    "mass": 77,
    "hair_color": "blond",
    "skin_color": "fair",
    "eye_color": "blue",
    "birth_year": "19BBY",
    "gender": "male",
    "homeworld": "SkNak6hcEjf",
    "film_ids": [
      "Sy3WJT2qVsG",
      "HycWyp39Nsf",
      "B1ibyphc4iG",
      "S1vZy63c4oz",
      "Sk6Zyp2c4oz"
    ],
    "species_ids": [
      "BJcKJT2qViz"
    ],
    "vehicle_ids": [
      "B1mJah9Eiz",
      "Byqf1ahcVsz"
    ],
    "starship_ids": [
      "HJMWJT254oG",
      "S1HZya254iM"
    ],
    "id": "SyAbJp35ViM"
  }
}
```

Environment variables

You can troubleshoot and change how C gRPC works by setting a few environment variables. There is a more complete reference of these here **https://github.com/grpc/grpc/blob/master/doc/environment_variables.md** but the most important are GRPC_TRACE and GRPC_VERBOSITY.

GRPC_TRACE

This is a comma-separated list of tracers that provide additional insight into how gRPC C core is processing requests, via debug-logs.

all can additionally be used to turn all traces on. Individual traces can be disabled by prefixing them with -.

refcount will turn on all of the tracers for refcount debugging.

If list_tracers is present, then all of the available tracers will be printed when the program starts up.

Example:

```
export GRPC_TRACE=all,-pending_tags
```

This means "put a trace on all available operations except pending_tags." There are quite a few traces you can turn on and off to debug what part of the request is failing, or just see more detail about what is happening. These will work for any library that uses the C core (for example Ruby, Node, Python.)

Available tracers include:

- api - traces api calls to the C core
- bdp_estimator - traces behavior of bdp estimation logic
- call_combiner - traces call combiner state
- call_error - traces the possible errors contributing to final call status
- channel - traces operations on the C core channel stack
- client_channel - traces client channel activity, including resolve and load balancing policy interaction
- combiner - traces combiner lock state
- compression - traces compression operations
- connectivity_state - traces connectivity state changes to channels
- channel_stack_builder - traces information about channel stacks being built
- executor - traces grpc's internal thread pool ('the executor')
- glb - traces the grpclb load balancer
- http - traces state in the http2 transport engine
- http2_stream_state - traces all http2 stream state mutations
- http1 - traces HTTP/1.x operations performed by gRPC
- inproc - traces the in-process transport
- flowctl - traces http2 flow control

- `op_failure` - traces error information when failure is pushed onto a completion queue
- `pick_first` - traces the pick first load balancing policy
- `plugin_credentials` - traces plugin credentials
- `pollable_refcount` - traces reference counting of 'pollable' objects (only in DEBUG)
- `resource_quota` - traces resource quota objects internals
- `round_robin` - traces the round_robin load balancing policy
- `queue_pluck`
- `queue_timeout`
- `server_channel` - lightweight trace of significant server channel events
- `secure_endpoint` - traces bytes flowing through encrypted channels
- `timer` - timers (alarms) in the gRPC internals
- `timer_check` - more detailed trace of timer logic in gRPC internals
- `transport_security` - traces metadata about secure channel establishment
- `tcp` - traces bytes in and out of a channel
- `tsi` - traces tsi transport security

The following tracers will only run in binaries built in DEBUG mode. This is accomplished by invoking `CONFIG=dbg make <target>`:

- `alarm_refcount` - refcounting traces for grpc_alarm structure
- `metadata` - tracks creation and mutation of metadata
- `closure` - tracks closure creation, scheduling, and completion
- `pending_tags` - traces still-in-progress tags on completion queues
- `polling` - traces the selected polling engine
- `polling_api` - traces the API calls to the polling engine
- `queue_refcount`
- `error_refcount`
- `stream_refcount`
- `workqueue_refcount`
- `fd_refcount`
- `cq_refcount`

- `auth_context_refcount`
- `security_connector_refcount`
- `resolver_refcount`
- `lb_policy_refcount`
- `chttp2_refcount`

GRPC_VERBOSITY

Default gRPC logging verbosity - one of:

- `DEBUG` - log all gRPC messages
- `INFO` - log INFO and ERROR messages
- `ERROR` - log only errors

Decompiling protobuf messages

You have a few options if you don't have the original proto IDL, but want to build it or just guess it from a binary protobuf message.

If you can at all help it (get your hands on the proto IDL for the data) it's recommended that you don't parse binary messages without the proto. It's unreliable, as many fields can share the same types (e.g., it's impossible to tell the difference between number-types and enums.)

- If you just need to see the format, you can use `protoc --decode_raw` to guess at the types.
- You can use **https://github.com/konsumer/rawproto** to extract the same sort of data as JSON or proto, which is a useful start, if you tune the output a bit.
- You can parse it by hand. This is pretty tricky, and not at all recommended because it's the most error-prone, and one of the other options should get you where you need to be, but there is documentation about how the messages are encoded **https://developers.google.com/protocol-buffers/docs/encoding**.

Summary

Now, you should be able to quickly run CLI commands to interact with your running gRPC server at a high-level. You should be able to troubleshoot your server, and see what the problem is at a very low-level.