Protocol Buffers 3

Introduction

“Protocol Buffers” is a combination toolset and format that facilitates the storing and transferring of data in a platform neutral way. It’s similar to formats like JSON/YAML/XML, except that it’s been specifically optimized for transferring data: faster to serialize/deserialize, serialized form takes up less space.

Like other existing formats, protocol buffers…

- are structured hierarchically

Unlike other existing formats, protocol buffers …

- are in binary format.
  (not human readable text)
- require structs to be known at design-time.
  (structs need to be defined beforehand and are strongly typed)
- support versioning of structs.
  (structs can be updated while still being backwards compatible)
- produce tightly coupled implementations.
  (unlike JSON/XML/etc.., there is no generic library that reads/writes generically)
  (custom code is generated to read/write your structs in each supported language)

The toolset provides support for the languages C++, Java, Python, Go, C#, etc..

> **NOTE**: There are 2 publicly available releases: proto2 and proto3. This document focuses on proto3 (latest), which is not compatible with proto2.

Setup

The most generic way to get set up is to download the latest binaries from https://github.com/protocolbuffers/protobuf/releases…

```
$ wget
https://github.com/protocolbuffers/protobuf/releases/download/v3.6.1/protoc
-3.6.1-linux-x86_64.zip
$ unzip protoc-3.6.1-linux-x86_64.zip -d protoc
```

> **NOTE**: The binaries are usually listed at the bottom of the list of released files. Everything up top is source code. For Linux, search specifically for “-linux-x86_64.zip”

Once installed, you'll use the "protoc" binary to compile your structures into source code in whatever languages you're working in…

```
$ cat > tutorial.proto
syntax = "proto3";
package tutorial;
option java_package = "com.example.tutorial";
message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
$ protoc/bin/protoc --java_out=. ./tutorial.proto
$ head ./com/example/tutorial/Tutorial.java
// Generated by the protocol buffer compiler.  DO NOT EDIT!
// source: tutorial.proto

package com.example.tutorial;

public final class Tutorial {
  private Tutorial() {}
  public static void registerAllExtensions(
      com.google.protobuf.ExtensionRegistryLite registry) {
  }
```

In addition, there's also an official Gradle plugin (https://github.com/google/protobuf-gradle-plugin) and a non-official Maven plugin (https://www.xolstice.org/protobuf-maven-plugin).

Definition

Protocol buffers are defined in files with a ".proto" extension and compiled to source code using the "protoc" binary (see Setup section).

An example definition file...

```
syntax = "proto3";

import "otherproject/other.proto";

package tutorial;
option java_package = "com.example.tutorial";
```

```
message User {
   string name = 1;
   uint32 age = 2;
   repeated string emails = 3;
}
```

The first line in the above example sets a syntax to “protobuf3”. We need to do this for all of our definition files. It tells the “protoc” compiler to use the 3rd version of protocol buffers.

> **NOTE**: If left unset, it targets version 2.

> **NOTE**: Language specs for protocol buffers can be found at https://developers.google.com/protocol-buffers/docs/reference/proto3-spec.

Import

A definition file can pull in other definition files using the import keyword. Access will be granted to everything that’s directly defined in the imported file. That means that if the file being imported has imports of its own, those imports won’t be made available to the parent. For example, take the following 3 files…

```
// a.proto
syntax = "proto3";
import "b.proto";
message A {
   B val=1;
   C val2=2;
}

// b.proto
syntax = "proto3";
import "c.proto";
message B {
   C val=1;
}

// c.proto
syntax = "proto3";
message C {
   uint32 val=1;
}
```

A imports B, and B imports C. That means that A has direct access to the items in B, but not to the items in C. If you tried to compile this, you'd get the following error…

```
a.proto:5:5: "C" seems to be defined in "c.proto", which is not imported by
"a.proto".  To use it here, please add the necessary import.
```

To work around this, you can either directly import C into A…

```
// a.proto
syntax = "proto3";
import "b.proto";
import "c.proto";  // notice that A is now directly importing C
message A {
   B val=1;
   C val2=2;
}
```

, or make it so that B includes C's items whenever it's imported (public import)...

```
// b.proto
syntax = "proto3";
import public "c.proto"; // notice the public keyword added here
message B {
   c val=1;
}
```

> **NOTE**: There's also the option to have a "weak" import. Weak imports are imports that are allowed to be missing? Unsure how it all works and they aren't mentioned in the official docs, but I found this on stack overflow: https://stackoverflow.com/q/33933397.
>
> **NOTE**: Import paths are ALWAYS relative. You cannot give an absolute path and you paths cannot start with ./ or ../ .

Package

Packages are like namespaces/packages in programming languages -- they're used to prevent name conflicts. You can use the package keyword to specify that your definition file belongs to some package. For example…

```
// a.proto
syntax = "proto3";
package company.serviceA;
import "b.proto";
message Msg{
   serviceB.Msg val=1;
```

```
}

// b.proto
syntax = "proto3";
package company.serviceB;
message Msg {
   uint32 val=1;
}
```

The code generated by the “protoc” compiler will appropriately convert the package structure to the targeted languages. For example, generated C++ code will be wrapped in the matching namespace hierarchy while generated Java code will contain the appropriate package declarations and sit in the appropriate directories.

Certain languages have more complex naming standards for packages. For example, the C++ namespace company.serviceA may be workable but for Java you’ll likely want a package name like com.company.serviceA.networking.protobuf. You can do this using options. For example…

```
// a.proto
syntax = "proto3";
package company.serviceA;
option java_package = "com.company.serviceA.networking.protobuf";
import "b.proto";
message Msg {
   serviceB.Msg val=1;
}
```

NOTE: See https://developers.google.com/protocol-buffers/docs/proto3#packages

Message

Messages are the structures that get serialized/deserialized. For example...

```
message User {
   option (my_option1) = true;
   option (my_option2) = 5;

   string name = 1;
   uint32 age = 2;
   repeated string emails = 3;
}
```

A message contains zero or more fields that can be set/get at runtime. Messages can also have options set on them (accessible programmatically at runtime).

> **NOTE**: Make sure to respect naming conventions when naming your message.

> **NOTE**:According to the docs, message options are an advanced feature. Most users will never need them.

> **NOTE**: Message can nest other messages as well as enumerations.

Field

Message fields are comprised of 4 elements:

- rule (not required)
- type
- name
- tag
- options (not required)

For example…

```
repeated uint64 vals = 1 [packed=true];
```

> **NOTE**: In the old version (v2), you could specify a default value as an option. If no default was provided, it would internally assign the default to the zero value (e.g. 0 for integers/floats, empty string, empty byte array, etc..). In v3, this is no longer a thing. You cannot provide a default and if you leave a field unset it will default to the value zero.

Field Rule

Field rules define how a field is to be treated. If a field is set to “repeated”, the field represents an array of values that share the same type. If not set, it’s a single value of type.

> **NOTE**: Remember that we’re dealing with v3 of protocol buffers. In the old version (v2), the field rules you could specify where optional/required/repeated. Optional/required defined whether a value must be provided for that field. In v3, this is no longer a thing: all fields are optional.

Field Name

Field names should be all lowercase letters where words are separated by underscores.

> **NOTE**: See naming conventions.

Field Tag

A field tags is the unique identifier for a field. While the field name is the human-friendly identifier of a field, the field tag is what's used internally. For example…

```
message 3dPoint {
    double x = 1;
    double y = 2;
    double z = 3;
}

message 2dPoint {
    double x = 1;
    double y = 2;
}
```

This is a required part of the field definition -- it must be explicitly declared. Note that field tags…

- must be an integers in the range [1, 2^{29}-1] but not between [19000, 19999].
- must be unique to the message they're encapsulated in.

These tags are are used to identify fields when serializing/deserializing. Because variable-length encoding is used to send these tags, you should aim to…

- keep tag numbers as small as possible.
- order them such that the fields most likely to be set have the smallest tag numbers (unset fields aren't serialized).

  NOTE: Tags are likely encoded as 32-bits, the first 3 bits of which are likely used as part of the variable length encoding (hence the range [1, 2^{29}-1]) and the range [19000, 19999] used for internal purposes.

Field Type

Supported types include built-in types (e.g. string or int64), enumerations, and messages. For example…
Example of fields in a message…

```
message 3dPoint {
    double x = 1;
    double y = 2;
    double z = 3;
    UvCoordinates uvs = 4; // UvCoordinates is another message
    BufferType e = 5; // BufferType is an enum
}
```

The following built-in types are provided...

- double → double-precision float
- float → single-precision float
- int32 → signed 32-bit integer
- int64 → signed 64-bit integer
- uint32 → unsigned 32-bit integer
- uint64 → unsigned 64-bit integer
- sint32 → signed 32-bit integer (encoded differently than int32)
- sint64 → signed 32-bit integer (encoded differently than int64)
- fixed32 → unsigned 32-bit integer (encoded differently than uint32)
- fixed64 → unsigned 64-bit integer (encoded differently than uint64)
- sfixed32 → signed 32-bit integer (encoded differently than int32/sint32)
- sfixed64 → signed 64-bit integer (encoded differently than int64/sint64)
- bool → boolean
- string → UTF-8 or 7-bit ASCII string
- bytes → variable number of bytes

  NOTE: See https://developers.google.com/protocol-buffers/docs/proto3#scalar.

  NOTE: Obviously, double/float are IEEE-754 formats and all signed integers are two's complement.

Notice there are several signed/unsigned integer types (e.g. int32 vs sint32 vs sfixed32). The difference between them is how they're encoded when serialized. Which type you use is dependent on your use case. For example, if you want your field's encoding to...

- always be serialized to 32-bits/64-bits, use sfixed32/sfixed64.
- be optimized for positive numbers, use int32/int64.
- be optimized for negative numbers, use sint32/sint64.

Oneof

A oneof groups multiple message fields in such a way that at most only 1 of the fields can be set. The source code generated for a oneof ensures compliance and serializes in a way that makes it more efficient (only 1 placeholder in memory for all fields). For example…

```
message A {
   uint32 item_a = 1;
   oneof item_b {
       string item_b_1 = 2;
       int64 item_b_2 = 3;
       double item_b_3 = 4;
   }
   uint32 item_c = 5;
}
```

In the above example, only one of the following items can be set: item_b_1, item_b_2, or item_b_3. If you set one, the other 2 will get cleared out. The source code generated by “protoc” for a oneof includes functionality that identifies which field was set. For example, generated Java code for the above example will have a method called getItemBCase() .

> **NOTE**: repeat field rule is not allowed for fields inside a oneof group.

Map

A map is a built-in helper that lets a single field hold on to multiple key-value pairings (similar to a Java map). For example…

```
message A {
   uint32 item_a = 1;
   map<string, AInner> item_b = 2;
}

message AInner {
   uint32 f_1 = 1;
   uint32 f_2 = 2;
}
```

Maps have the following restrictions/gotchas:

- maps are not ordered -- don’t expect any particular order when iterating over a map.
- repeat field rule is not allowed for a map.
- key type for a map must be either a integer, bool, or string.
    - You cannot set enumerations or messages as key types.
    - You cannot set other built-in types (e.g. float, double, bytes) as the key type.
- entries without a value have undefined serialization behaviour (always set a value).

There is no restriction for what a value type can be, but just make sure you always set a value (see last point in list above).

Any

The “Any” type is a special placeholder when the definition of the type isn’t known at design-time. For example…

```
import "google/protobuf/any.proto";  // MUST HAVE THIS IMPORT

message A {
   uint32 a = 1;
   google.protobuf.Any b = 2;  // Any TYPE MUST BE FULLY QUALIFIED
```

```
}
```

> **NOTE**: If you're seeing messages about any.proto not being found, you likely have a bad install. Install protocol buffers using the official binaries and it'll work. Do not use the distribution from apt or yum or whatever.

In the above example, field b is an "Any" type. At runtime, you'll be able to convert field b to/from a real type using the Pack() and Unpack() methods. For example, in Java…

```
MessageB messageB = ...;

// to set
Any any = Any.pack(messageB);
messageA.setB(any);

// to get
messageB = messageA.getB().unpack(MessageB.class);
```

An "Any" message can hold on to a URL that uniquely identifies the type of serialized message.

> **NOTE**: Unsure why a URL is used to identify types? The doc seems unclear. See https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Any.html#getTypeUrl-- and https://developers.google.com/protocol-buffers/docs/reference/java/com/google/protobuf/Any.html#pack-T-java.lang.String-.

Enumeration

Enumerations are just like normal enumerations in most programming languages (e.g. Java). For example…

```
enum CardType {
    option allow_alias = true;

    SPADE = 0,
    DIAMOND = 1,
    HEART = 2,
    CLUB = 3 [ opt1="value1", opt2="value2"]
}
```

Enumerations can have options on both the enumeration as well as the enumerated values. Just like messages, users should rarely ever need to use them.

NOTE: Make sure to respect naming conventions when naming your enumeration. Your enumerated values should be all uppercase and where words are separated by underscores. The enumeration itself should be camel case.

NOTE: It's recommended that you have one of your enumerated values set to 0. The default value for an enumeration type is 0.

NOTE: Enumerations can be nested inside of messages. See messages.

Service

Service are interfaces that use your messages. They're typically used for generating interfaces for RPC implementations. For example…

```
Service MyService {
    option allow_alias = true;

    rpc Method (RequestMsg) returns (ResponseMsg);
    rpc MethodStreamResp (RequestMsg) returns (stream ResponseMsg);
    rpc MethodStreamReq (stream RequestMsg) returns (ResponseMsg);
    rpc MethodStreamBi (stream RequestMsg) returns (stream ResponseMsg);
    rpc MethodWithOptions (RequestMsg) returns (ResponseMsg) {
        option my_option = "hi";
    }
}
```

Notice that options can be placed on both the service and the individual method signatures on that service. Individual method signatures…

- must accept exactly 1 message type (must be msg, can't be built-in/enum/void/etc...).
- must return exactly 1 message type (must be msg, can't be built-in/enum/void/etc...).
- can be set to accept single requests / deliver single responses.
- can be set to accept a stream of requests / deliver a single response.
- can be set to accept single requests / deliver a stream of responses.
- can be set to accept a stream of requests / deliver a stream of responses.

The interfaces generated for these services can be used standalone, but they were designed to be used by some higher-level layer -- gRPC is the commonly used higher-level layer (brief introduction in the gRPC section).

NOTE: Make sure to respect naming conventions when naming your service. Your service as well as methods should be camelcase.

NOTE: MethodStreamReq in the example above stream multiple requests and generates a single response. gRPC buffers the requests internally until everything arrives before allowing you to compute a response. This isn't the case with MethodStreamBi -- you can start sending responses right away.

NOTE: Recall the "repeated" field rule. It only applies to fields in a message, not to rpc inputs/outputs. However, you can use stream (which is kinda like repeat) OR you can wrap input/output message types in a secondary message that repeats. For example…

```
message MethodResponse {
   uint32 unix_timestamp = 0;
   string name = 1;
   bool whatever = 2;
}
message MethodResponseWrapper {
   repeat MethodResponse resp = 0;
}
```

Versioning

As your product grows and goes through revisions, its common for the structure of your protobuf messages to change. Fields are typically added and sometimes replaced. For example, you may have initially designed your message like this…

```
message User {
   uint32 id = 1;
   string email = 2;
}
```

but due to changes in requirements they now have to be like this...

```
message User {
   uint32 id = 1;
   Email email = 3;
}

message Email {
   string alias = 1;
   string domain = 2;
}
```

There are several things to be aware of when updating your protocol buffers.

NOTE: The subsections below focus on adding/removing fields, but you can also change types on a field (this is an advanced feature that you likely shouldn't ever have to use?).

> See https://developers.google.com/protocol-buffers/docs/proto3#updating for more information.

Add Fields

Adding a field is straightforward: add it into your message just as you would any other field. For example, an existing message...

```
message User {
   uint32 id = 1;
   string email = 2;
}
```

can be revised to add a new field…

```
message User {
   uint32 id = 1;
   string email = 2;
   string address = 3;  // new field
}
```

If you serialize in the old format but deserialize in the revised format, the address field won't have any data associated with it -- it will default to zero value on deserialization (e.g. 0 for integers/floats, empty string, empty byte array, etc..). In the example above, the address field will default to an empty string.

If you serialize in the revised format but deserialize in the old format, the address field will be missing -- it will get treated as an unknown field. Depending on the version of protocol buffers you're using, unknown fields may be retained or silently discarded.

> **NOTE**: See https://developers.google.com/protocol-buffers/docs/proto3#unknowns.

Remove Fields

Removing a field is straight forward: delete it from your field and reserve the field tag and name so they can't be reused in future revisions of the message. For example…

```
message User {
   uint32 id = 1;
   string email = 2;
   string address = 3;
}
```

can be revised to remove a field…

```
message User {
```

```
   uint32 id = 1;
   string email = 2;
   reserved 3;           // reserve removed field tag 3
   reserved "address";   // reserve removed field name address
}
```

> **NOTE**: If reserving multiple field names/tags, you can use a comma delimited list of items. If reserving a range of field tags, you can use the “to” keyword. For example: reserved 1, 3, 10, 100 to 150, 160;

The behaviour of removed fields is similar to added fields.

If you serialize in the old format but deserialize in the revised format, the address field will be missing -- it will get treated as an unknown field. Depending on the version of protocol buffers you’re using, unknown fields may be retained or silently discarded.

If you serialize in the revised format but deserialize in the old format, the address field won’t have any data associated with it -- it will default to zero value on deserialization (e.g. 0 for integers/floats, empty string, empty byte array, etc..). In the example above, the address field will default to an empty string.

> **NOTE**: See https://developers.google.com/protocol-buffers/docs/proto3#unknowns.

Compiling

The “protoc” compiler allows you to generate code for different languages. Each language has a unique set of options.

> **NOTE**: Officially supported languages are C++, Java, Python, Go, and C#. It also has alpha support for Ruby, C#, ObjC, Javascript, and PHP. The subsections below only cover the first 3 main languages: C++, Java, and Python. For the others, see https://developers.google.com/protocol-buffers/docs/reference/overview.

C++

To generate C++ code, use the cpp_out argument…

```
$ /protoc/bin/protoc --cpp_out=./dst_cpp a.proto
```

Generated C++ code can be…

- optimized for speed, code size, or be linked against a smaller protobuf library (less functionality) via the optimize_for option.

- made to make more efficient use of memory by memory pooling via the cc_enable_arenas option.

```
syntax = "proto3";

option optimize_for = SPEED; // or CODE_SIZE or LITE_RUNTIME
option cc_enable_arenas = true;

message User {
   string name = 1;
   uint32 age = 2;
   string email = 3;
}
```

Java

To generate Java code, use the java_out argument…

```
$ /protoc/bin/protoc --java_out=./dst_java a.proto
```

The generated code has a dependency on the protobuf lib…

```
<dependency>
  <groupId>com.google.protobuf</groupId>
  <artifactId>protobuf-java</artifactId>
  <version>3.6.1</version>
</dependency>
```

Generated Java code can be…

- optimized for speed, code size, or inherit from a reduced protobuf library (less functionality) via the optimize_for option.
- generated with a package hierarchy more appropriate to Java via the java_package option.
- generated with a custom outer classname via java_outer_classname option -- by default all classes in a proto file are bundled inside of a single Java class called ProtoFilename.java (where ProtoFilename is the filename of your .proto).

```
syntax = "proto3";
package company.serviceA;

option optimize_for=SPEED; // or CODE_SIZE or LITE_RUNTIME
option java_package="com.company.serviceA";
```

```
option java_outer_classname = "UserDomain";

message User {
   string name = 1;
   uint32 age = 2;
   Email email = 3;
}

message Email {
   string alias = 1;
   string domain = 2;
}
```

Maven Plugin

Instead of using protoc directly, it's possible to generate code through a maven plugin…

```
<build>
  <extensions>
    <extension>
      <groupId>kr.motd.maven</groupId>
      <artifactId>os-maven-plugin</artifactId>
      <version>1.6.0</version>
    </extension>
  </extensions>
  <plugins>
    <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.6.1</version>
      <executions>
        <execution>
          <goals>
            <goal>compile</goal>
            <goal>test-compile</goal>
          </goals>
        </execution>
      </executions>
      <configuration>

<protocArtifact>com.google.protobuf:protoc:3.4.0:exe:${os.detected.classifi
er}</protocArtifact>
      </configuration>
```

```
    </plugin>
  </plugins>
</build>

<dependencies>
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.6.1</version>
  </dependency>
</dependencies>
```

NOTE: Proto files go under src/main/proto.

Gradle Plugin

Instead of using protoc directly, it's possible to generate code through a maven plugin…

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath 'com.google.protobuf:protobuf-gradle-plugin:0.8.6'
  }
}

apply plugin: 'java'
apply plugin: 'com.google.protobuf'

plugins {
  id "com.google.protobuf" version "0.8.6"
  id "java"
}

dependencies {
    compile 'com.google.protobuf:protobuf-java:3.6.1'
}
```

NOTE: Proto files go under src/main/proto.

NOTE: See https://github.com/google/protobuf-gradle-plugin for more info.

Python

To generate Python code, use the python_out argument…

```
$ /protoc/bin/protoc --python_out=./dst_py a.proto
```

> **NOTE**: There aren't any Python options worth mentioning.

Gotchas and Best Practices

Best Practices

- Always use proper style / naming conventions.
  → protoc will convert to the appropriate naming convention of the destination language.
- Never change field tags once deployed.
  → other systems may break because they'll be expecting the old field tags.
- Reserve the tags and names of removed fields.
  → reserving names prevents you from reusing the name in future revisions, meaning you won't break your source code.
  → reserving tags prevents you from reusing the tag in future revisions, meaning that you won't break existing systems that may still be using them.

Naming Conventions

- Messages, enumerations, and services/methods should be named in camelcase.
- Message fields should be all lowercase where words are separated by underscores.
- Enumerated values should be all uppercase where words are separated by underscores.

So long as this naming convention is respected, the protoc compiler will convert the name to the correct format for the language being output. So for example, a field named world_coordinates would get compiled to Java as 2 methods: getWorldCoordinates() and setWorldCoordinates().

> **NOTE**: See https://developers.google.com/protocol-buffers/docs/style.

Nesting for Organization

You can nest enumerations and messages inside of other messages. For example…

```
message a {
   uint32 item_a = 1;
```

```
    message a_inner {
        uint32 f_1 = 1;
        uint32 f_2 = 2;
    }

    enum my_type {
        A = 0;
        B = 1;
        C = 2;
    }
}
```

This is useful for organization. For example, if you have an enum that's only used by a single message, it may make sense to tie them together.

Speed and Space Consumption

It isn't always the case that protocol buffers are the fastest and most space efficient solution. For example, serialization of protocol buffers is slower in Javascript vs JSON serialization. The serialized data is more space efficient with protocol buffers, but it's still slower to generate.

The idea is that you have an interoperable format that SHOULD be faster.

Languages Types vs Protobuf Types

Not all protocol buffer types are supported by all languages. For example, protocol buffers offer the type uint64 Neither Java nor Javascript support an unsigned 64-bit integer type.

The code generated by protocol buffers tries to work around this. For example, in…

- Java, uint64 is put into a long, which is signed but can still hold all 64-bits.
- Javascript, all 64-bit number types are put into a string.

  NOTE: Remember that Java's only numeric type is "number", which translates to IEEE-754 double format.

gRPC Basics

gRPC (short for Google RPC) is the standard way of implementing services. It essentially acts as the transport layer for your protocol, leaving you to just implement the interface methods in your services and spin up the server/clients.

Much like protocol buffers, gRPC code is generated using the “protoc” tool and isn’t bound to a single language. gRPC can be used with multiple different programming languages, with each language implementation being spec’d out for the norms of that language (e.g. uses common libraries, coding patterns and idioms, etc..).

> **NOTE**: The following subsections are specific to Java. If you’re dealing with another language, it’s best to go over the guides for that language on the grpc website.

Setup

To have “protoc” generate gRPC code for Java, you add the grpc_out argument as well as specify a special grpc plugin. The grpc_out argument points to the destination directory where your generated gRPC-related Java files should go, while the plugin is some executable that “protoc” uses internally for something (unsure what). For example…

```
$ /protoc/bin/protoc
 --grpc_out==./dst_java
 --java_out=./dst_java a.proto
 --plugin=protoc-gen-grpc=...\protoc-gen-grpc-java-?.?.?-windows-x86_64.exe
```

> **NOTE**: You might have to get the protoc-gen-grpc plugin separately from the main protoc package. You can find it on Maven under group:io.grpc artifact:protoc-gen-grpc-java.

The Maven and Gradle plugins should both support gRPC. But you might have provide some extra configuration options. For example, the plugin needs some configuration changes…

```
<plugin>
  <groupId>org.xolstice.maven.plugins</groupId>
  <artifactId>protobuf-maven-plugin</artifactId>
  <version>0.6.1</version>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <!-- NEXT GOAL IS REQUIRED TO GENERATE GRPC -->
        <goal>compile-custom</goal>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
```

```
<protocArtifact>com.google.protobuf:protoc:3.4.0:exe:${os.detected.classifier}</protocArtifact>
    <!-- PLUGIN IS REQUIRED FOR GENERATING GRPC -->
    <pluginId>grpc-java</pluginId>

<pluginArtifact>io.grpc:protoc-gen-grpc-java:1.15.1:exe:${os.detected.classifier}</pluginArtifact>
  </configuration>
</plugin>
```

and you need to pull in the following dependencies to use gRPC in Java...

```
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-netty-shaded</artifactId>
  <version>1.16.1</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-protobuf</artifactId>
  <version>1.16.1</version>
</dependency>
<dependency>
  <groupId>io.grpc</groupId>
  <artifactId>grpc-stub</artifactId>
  <version>1.16.1</version>
</dependency>
<!-- Javadocs generated for GRPC/Protobufs require the
javax.annotation.Generated, which is not provided in Java9 onward. -->
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
</dependency>
```

> **NOTE**: Remember that if you're using either the Gradle or Maven plugins, your proto files go under src/main/proto.

> **NOTE**: For Gradle, there's a difference for dependencies for Android vs non-Android builds. See https://github.com/grpc/grpc-java for more info.

Service Implementation

gRPC generate stubs for your services. Imagine the following proto file…

```
syntax = "proto3";

package tutorial;
option java_package = "com.offbynull.grpctest.grpc";
option java_outer_classname = "GrpcTest";

message RequestMsg {
    string input = 1;
}

message ResponseMsg {
    string output = 1;
}

service CustomService {
    rpc Method (RequestMsg) returns (ResponseMsg);
    rpc MethodStreamResp (RequestMsg) returns (stream ResponseMsg);
    rpc MethodStreamReq (stream RequestMsg) returns (ResponseMsg);
    rpc MethodStreamBi (stream RequestMsg) returns (stream ResponseMsg);
    rpc MethodWithOptions (RequestMsg) returns (ResponseMsg);
}
```

The “protoc” compiler will generate stubs for the services that you can implement functionality for in your code…

```
package com.offbynull.grpctest;

import com.offbynull.grpctest.grpc.CustomServiceGrpc;
import com.offbynull.grpctest.grpc.GrpcTest;
import io.grpc.stub.StreamObserver;

public class GrpcService extends CustomServiceGrpc.CustomServiceImplBase {
    @Override
    public void method(
            GrpcTest.RequestMsg request,
            StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
        // do something here
    }
```

```
    @Override
    public void methodStreamResp(
            GrpcTest.RequestMsg request,
            StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
        // do something here
    }

    @Override
    public StreamObserver<GrpcTest.RequestMsg> methodStreamReq(
            StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
        // do something here
    }

    @Override
    public StreamObserver<GrpcTest.RequestMsg> methodStreamBi(
            StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
        // do something here
    }

    @Override
    public void methodWithOptions(
            GrpcTest.RequestMsg request,
            StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
        // do something here
    }
}
```

Notice how these methods…

- respond through StreamObserver, regardless of if it was set to send a single response or a stream of responses.
- take in an actual object for single requests, but return a StreamObserver for stream of requests.

The idea with this is that your implementation can be async.

If a single request / single response is expected, you can spin the work off to another thread and return immediately while that thread works. Once that thread is finished, it should push out the response to the response StreamObserver….

```
@Override
public void method(
```

```
        GrpcTest.RequestMsg request,
        StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
    Runnable r = () -> {
        GrpcTest.ResponseMsg resp =GrpcTest.ResponseMsg.newBuilder()
                .setOutput("hi!")
                .build();
        responseObserver.onNext(resp);
        responseObserver.onCompleted();
    }
    new Thread(r).start();
}
```

If a stream of responses is expected, the idea is the same as a single response but you can invoke the response StreamObserver multiple times...

```
@Override
public void methodStreamResp(
        GrpcTest.RequestMsg request,
        StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
    Runnable r = () -> {
        GrpcTest.ResponseMsg resp =GrpcTest.ResponseMsg.newBuilder()
                .setOutput("hi!")
                .build();
        responseObserver.onNext(resp);
        responseObserver.onNext(resp);
        responseObserver.onNext(resp);
        responseObserver.onCompleted();
    }
    new Thread(r).start();
}
```

If a stream of requests is expected, you return a StreamObserver that gets invoked for each request and you invoke the response StreamObserver once that work is complete…

```
@Override
public StreamObserver<GrpcTest.RequestMsg> methodStreamReq(
        StreamObserver<GrpcTest.ResponseMsg> responseObserver) {
    return new StreamObserver<GrpcTest.RequestMsg>() {
        @Override
        public void onNext(GrpcTest.RequestMsg value) {
            GrpcTest.ResponseMsg resp =GrpcTest.ResponseMsg.newBuilder()
                    .setOutput("echo:" + value.getInput())
                    .build();
```

```
            responseObserver.onNext(resp);
        }

        @Override
        public void onError(Throwable t) {
            // do something here to report and stop
        }

        @Override
        public void onCompleted() {
            responseObserver.onCompleted(); // call this when done
        }
    };
}
```

Server Launch

Starting and stopping a server is straight forward…

```
package com.offbynull.grpctest;

import io.grpc.Server;
import io.grpc.ServerBuilder;

public class App
{
    public static void main( String[] args ) throws Exception
    {
        // start
        Server server = ServerBuilder
                .forPort(12345)
                // .useTransportSecurity(certPemFile, keyPemFile)
                .addService(new GrpcService()) // add implemented services
                .build()
                .start();

        // stop
        server.shutdown(); // or shutdownNow() to stop immediately
        server.awaitTermination();
    }
}
```

Client Launch

gRPC generates client stubs for your services. It's your choice if you want to use the synchronous or asynchronous variant, but methods that stream requests won't be available in the blocking stub…

```
package com.offbynull.grpctest;

import com.offbynull.grpctest.grpc.CustomServiceGrpc;
import com.offbynull.grpctest.grpc.GrpcTest;
import io.grpc.ManagedChannel;
import io.grpc.netty.shaded.io.grpc.netty.NettyChannelBuilder;
import io.grpc.stub.StreamObserver;

import java.util.concurrent.TimeUnit;

public class App
{
    public static void main( String[] args ) throws Exception
    {
        ManagedChannel channel = NettyChannelBuilder
                .forAddress("localhost", 12345)
                // .sslContext(
                //         GrpcSslContexts
                //                 .forClient()
                //                 .trustManager(certPemFile)
                //                 .build()
                // )
                .build();


        // client that blocks until invokations are finished
            // will not contain methods that stream responses
        CustomServiceGrpc.CustomServiceBlockingStub blockingClient =
                CustomServiceGrpc.newBlockingStub(channel);
        blockingClient.method(...);


        // client that is async
        CustomServiceGrpc.CustomServiceStub nonBlockingClient =
                CustomServiceGrpc.newStub(channel);
        nonBlockingClient.method(req, new
```

```
StreamObserver<GrpcTest.ResponseMsg>() {
            @Override
            public void onNext(GrpcTest.ResponseMsg value) {
                System.out.println(value);
            }

            @Override
            public void onError(Throwable t) {
                System.out.println(t);
            }

            @Override
            public void onCompleted() {
                System.out.println("done!");
            }
        });


        // shutdown
        channel.shutdown(); // shutdownNow to stop without clearing bufs
        channel.awaitTermination(Long.MAX_VALUE, TimeUnit.DAYS);

        // FYI: it's best to invoke shutdown after you've actually
        // send and recv'd everything, so as to shutdown the channel
        // prior to completing your communication.
    }
}
```

For the async variant, the StreamObservers are used similar to how they're implemented for service implementations. That is, if you're streaming

- requests, you'll get a StreamObserver to invoke for each request.
- responses, you'll supply a StreamObserver that will get invoked for each response.