gRPC Fundamentals with Go

gRPC

- gRPC is a remote procedure call (RPC) implementation technology that uses HTTP 2.0 as its underlying transport protocol.

- It’s a modern, high-performance RPC framework that can run in any environment

Advantages of gRPC

- It’s efficient for inter-process communication.
- It’s strongly typed.
- RPC service contracts clearly define the types that you will be using for communication between the applications.

Advantages of gRPC

- This makes distributed application development much more stable, as static typing helps to overcome most of the runtime and interoperability errors.

- It has duplex streaming gRPC has native support for client- or server-side streaming,
  - It is baked into the service definition itself.

Defining a Message Type

```
syntax = "proto3";

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

Defining a Message Type

- The first line of the file specifies that you're using proto3 syntax.

- If you don't do this the protocol buffer compiler will assume you are using proto2.

Field Numbers

- Each field in the message definition has a **unique number**.
- These field numbers are used to identify your fields in the message binary format,
- It should not be changed once your message type is in use.

Field Numbers

- Note that field numbers in the range 1 through 15 take one byte to encode, including the field number and the field's type (you can find out more about this in Protocol Buffer Encoding).
- Field numbers in the range 16 through 2047 take two bytes.
- Should reserve the numbers 1 through 15 for very frequently occurring message elements. Remember to leave some room for frequently occurring elements that might be added in the future.

Field Numbers

- The smallest field number you can specify is 1,
- The largest is 2^{29} - 1, or 536,870,911. You also cannot use the numbers 19000 through 19999

Repeated Fields and Maps

- repeated: this field type can be repeated zero or more times in a well-formed message. The order of the repeated values will be preserved.

- map: this is a paired key/value field type.

Comments

- To add comments to your .proto files, use C/C++-style // and /* ... */ syntax.

```
/* SearchRequest represents a search query, with pagination options to
 * indicate which results to include in the response. */

message SearchRequest {
  string query = 1;
  int32 page_number = 2;  // Which page number do we want?
  int32 result_per_page = 3;  // Number of results to return per page.
}
```

Default Values

- If the encoded message does not contain a particular singular element, the corresponding field in the parsed object is set to the default value for that field. These defaults are type-specific:

- For strings, the default value is the empty string.
- For bytes, the default value is empty bytes.
- For bools, the default value is false.

Default Values

- For numeric types, the default value is zero.
- For enums, the default value is the **first defined enum value**, which must be 0.
- For message fields, the field is not set. Its exact value is language-dependent. See the generated code guide for details.
- The default value for repeated fields is empty (generally an empty list in the appropriate language).

Adding More Message Types

- Multiple message types can be defined in a single .proto file.
- This is useful if you are defining multiple related messages
- For example, if you wanted to define the reply message format that corresponds to your SearchResponse message type, you could add it to the same .proto:

Adding More Message Types

```
message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message SearchResponse {
 ...
}
```

Reserved Fields

- If you update a message type by entirely removing a field, or commenting it out, future users can reuse the field number when making their own updates to the type.

- This can cause severe issues if they later load old versions of the same .proto, including data corruption, privacy bugs, and so on.

Reserved Fields

- One way to make sure this doesn’t happen is to specify that the field numbers of your deleted fields are **reserved**.

- The protocol buffer compiler will complain if any future users try to use these field identifiers.

Reserved Fields

```
message Foo {
  reserved 2, 15, 9 to 11;
  reserved "foo", "bar";
}
```

Enumerations

- When you’re defining a message type, you might want one of its fields to only have one of a pre-defined list of values.

- Enum’s first constant maps to zero:

- Every enum definition **must** contain a constant that maps to zero as its first element

Enumerations

- There must be a zero value, so that we can use 0 as a numeric default value.
- The zero value needs to be the first element, for compatibility with the proto2 semantics where the first enum value is always the default.

Enumerations

```
enum Corpus {
  CORPUS_UNSPECIFIED = 0;
  CORPUS_UNIVERSAL = 1;
  CORPUS_WEB = 2;
  CORPUS_IMAGES = 3;
  CORPUS_LOCAL = 4;
  CORPUS_NEWS = 5;
  CORPUS_PRODUCTS = 6;
  CORPUS_VIDEO = 7;
}

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
  Corpus corpus = 4;
}
```

Nested Types

- You can define and use message types inside other message types,

```
message SearchResponse {
  message Result {
    string url = 1;
    string title = 2;
    repeated string snippets = 3;
  }
  repeated Result results = 1;
}
```

```
message SomeOtherMessage {
  SearchResponse.Result result = 1;
}
```

Updating A Message Type

- If an existing message type no longer meets all your needs –
- for example, you'd like the message format to have an extra field – but you'd still like to use code created with the old format
- It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

Updating A Message Type

- Don't change the field numbers for any existing fields.

- If you add new fields, any messages serialized by code using your "old" message format can still be parsed by your new generated code.

- You should keep in mind the default values for these elements so that new code can properly interact with messages generated by old code.

Updating A Message Type

- Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing
- Fields can be removed, as long as the field number is not used again in your updated message type
- Make the field number reserved, so that future users of your .proto can't accidentally reuse the number

Oneof

- If you have a message with many fields and where at most one field will be set at the same time.
- You can enforce this behavior and save memory by using the oneof feature.
- Oneof fields are like regular fields except all the fields in a oneof share memory, and at most one field can be set at the same time.
- A oneof cannot be repeated.

Oneof

- Setting any member of the oneof automatically clears all the other members.

- You can check which value in a oneof is set (if any) using a special case() or WhichOneof() method, depending on your chosen language.

- Note that if multiple values are set, the last set value as determined by the order in the proto will overwrite all previous ones.

Defining RPC

- If you want to use your message types with an RPC (Remote Procedure Call) system, you can define an RPC service interface in a .proto file

- The protocol buffer compiler will generate service interface code and stubs in your chosen language

```
service SearchService {
  rpc Search(SearchRequest) returns (SearchResponse);
}
```

File location

- Prefer not to put .proto files in the same directory as other language sources.

- Consider creating a sub package proto for .proto files, under the root package for your project.

File Structure

- Files should be named lower_snake_case.proto.
- All files should be ordered in the following manner:
    1. License header (if applicable)
    2. File overview
    3. Syntax
    4. Package
    5. Imports (sorted)
    6. File options
    7. Everything else

Message and Field Names

- Use CamelCase (with an initial capital) for message names – for example, SongServerRequest.
- Use underscore_separated_names for field names for example, song_name.
- message SongServerRequest { string song_name = 1; }

Repeated Fields

- Use pluralized names for repeated fields.

- **repeated string** keys = 1**;**
- **repeated MyMessage** accounts **=** 17**;**

Enums

- Use CamelCase (with an initial capital) for enum type names and CAPITALS_WITH_UNDERSCORES for value names

```
enum FooBar {
  FOO_BAR_UNSPECIFIED = 0;
  FOO_BAR_FIRST_VALUE = 1;
  FOO_BAR_SECOND_VALUE = 2;
}
```

Services

- If your .proto defines an RPC service, you should use CamelCase (with an initial capital) for both the service name and any RPC method names:

```
service FooService {
  rpc GetSomething(GetSomethingRequest) returns (GetSomethingResponse);
}
```

Large Data Sets

- Protocol Buffers are not designed to handle large messages.
- As a general rule of thumb, if you are dealing in messages larger than a megabyte each, it may be time to consider an alternate strategy.
- That said, Protocol Buffers are great for handling individual messages *within* a large data set.

Large Data Sets

- Usually, large data sets are a collection of small pieces, where each small piece is structured data.
- Even though Protocol Buffers cannot handle the entire set at once, using Protocol Buffers to encode each piece greatly simplifies your problem.
- Now all you need is to handle a set of byte strings rather than a set of structures.

Large Data Sets

- Protocol Buffers do not include any built-in support for large data sets because different situations call for different solutions.

- Sometimes a simple list of records will do while other times you want something more like a database.

- Each solution should be developed as a separate library, so that only those who need it need pay the costs.

gRPC

- gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types

- In gRPC, a client application can directly call a method on a server application on a different machine as if it were a local object

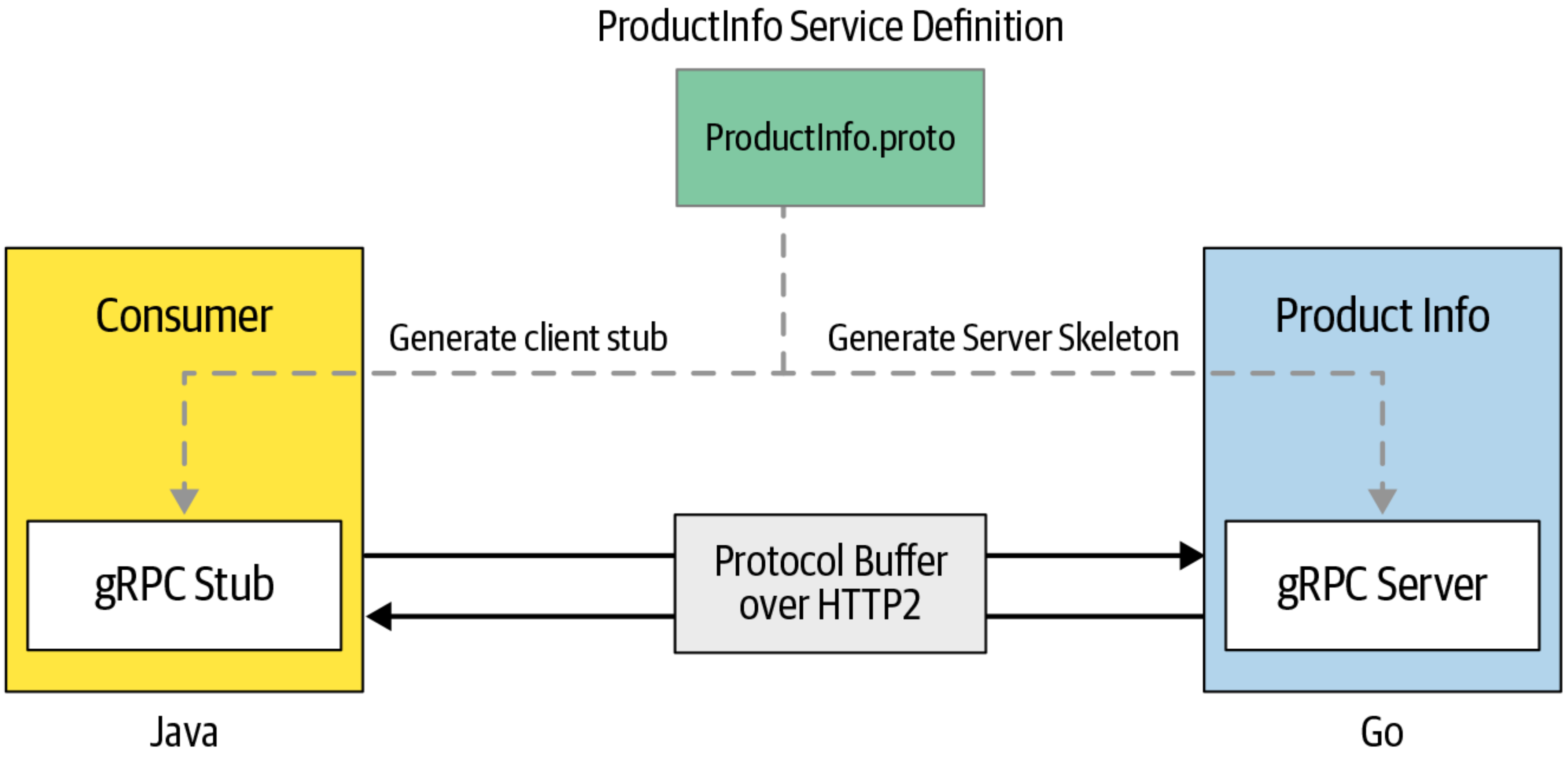
- Making it easier for you to create distributed applications and services.

gRPC

- On the server side, the server implements this interface and runs a gRPC server to handle client calls.

- On the client side, the client has a stub (referred to as just a client in some languages) that provides the same methods as the server.

gRPC Communication



gRPC Communication Patterns

- Simple RPC (Unary RPC)
- Server streaming RPCs
- Client streaming RPCs
- Bidirectional streaming RPCs

Unary RPC

- In simple RPC, when a client invokes a remote function of a server,
  - the client sends a single request to the server
  - and gets a single response that is sent along with status details and trailing metadata

Server-Streaming RPC

- In server-side streaming RPC, the server sends back a sequence of responses after getting the client’s request message.
- This sequence of multiple responses is known as a “stream.”
- After sending all the server responses, the server marks the end of the stream by sending the server’s status details as trailing metadata to the client.

Client-Streaming RPC

- In client-streaming RPC, the client sends multiple messages to the server instead of a single request.
- The server sends back a single response to the client.
- Once the client has finished writing the messages, it waits for the server to read them and return its response.

Bidirectional-Streaming RPC

- In bidirectional-streaming RPC, the client is sending a request to the server as a stream of messages.

- The server also responds with a stream of messages.

- The call has to be initiated from the client side,

- But after that, the communication is completely based on the application logic of the gRPC client and the server.

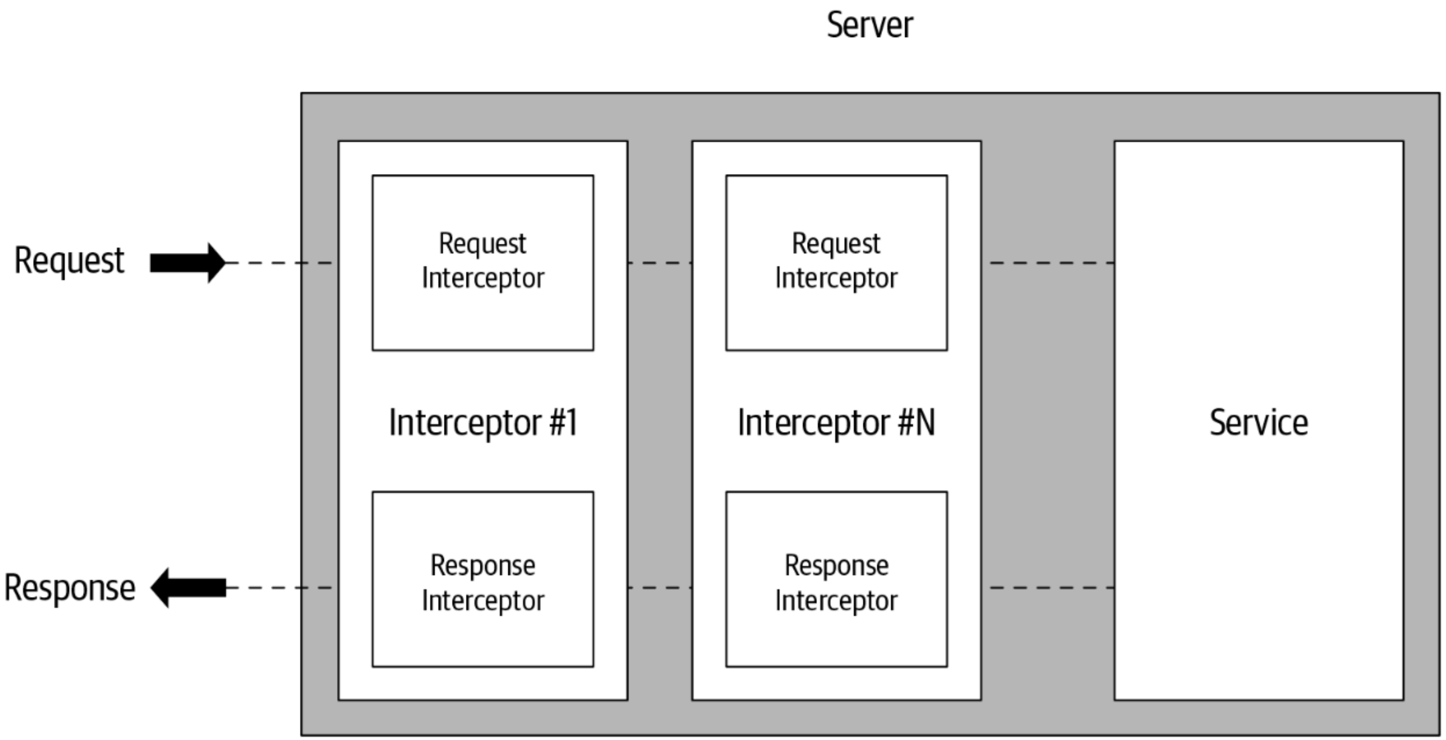
Interceptors

- As you build gRPC applications, you may want to execute some common logic before or after the execution of the remote function, for either client or server applications.

- In gRPC you can intercept that RPC's execution to meet certain requirements such as logging, authentication, metrics, etc

- For unary RPC you can use unary interceptors, while for streaming RPC you can use streaming interceptors.

Interceptors



.proto file and well know type

- https://github.com/protocolbuffers/protobuf/blob/main/src/google/protobuf/descriptor.proto

- https://protobuf.dev/reference/protobuf/google.protobuf/