Welcome to #APIFestival

What will be exploring today?

- ➔ A quick intro to gRPC
- ➔ How it works
- ➔ How to define gRPC services(messages & services).
- ➔ What are the pros & cons
- ➔ Dart & Flutter
- ➔ Pros & Cons
- ➔ A simple events service with authentication.
- ➔ Types of services(Unary, Server-Streaming, Client-Streaming, Bi-directional Streaming)
- ➔ How clients can consume these events

What is gRPC?

How many of you are familiar with gRPC?

What is gRPC?

gRPC is a modern open source high performance Remote Procedure Call (RPC) framework that can run in any environment.

- Uses protocol buffers for service definition
- Uses http2
- Language agnostic.
- Streaming capabilities

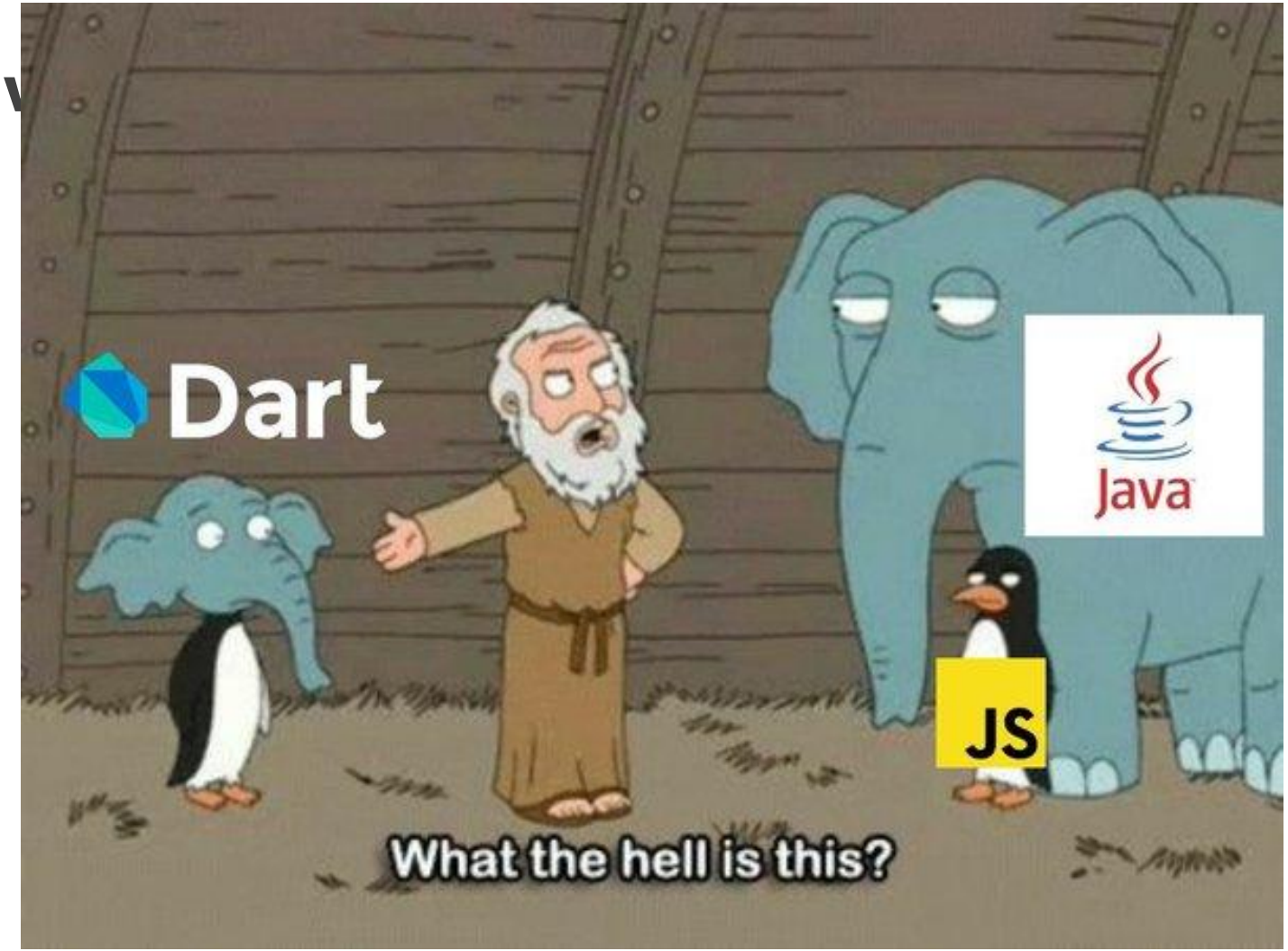
Why use gRPC?

- Speed
- Protocol Buffers(size, de(serialization),language support)
- Broad language support
- Streaming capabilities inbuilt
- Low latency and small
- Easy to use generated classes which offer type-safety on type-safe languages :-)
- Learn a new tool :-)

And why Dart?

- What I’m motivated with is, one stack for all your needs.
- If you can enjoy the fluency of Dart on the front-end, why not also on the backend?

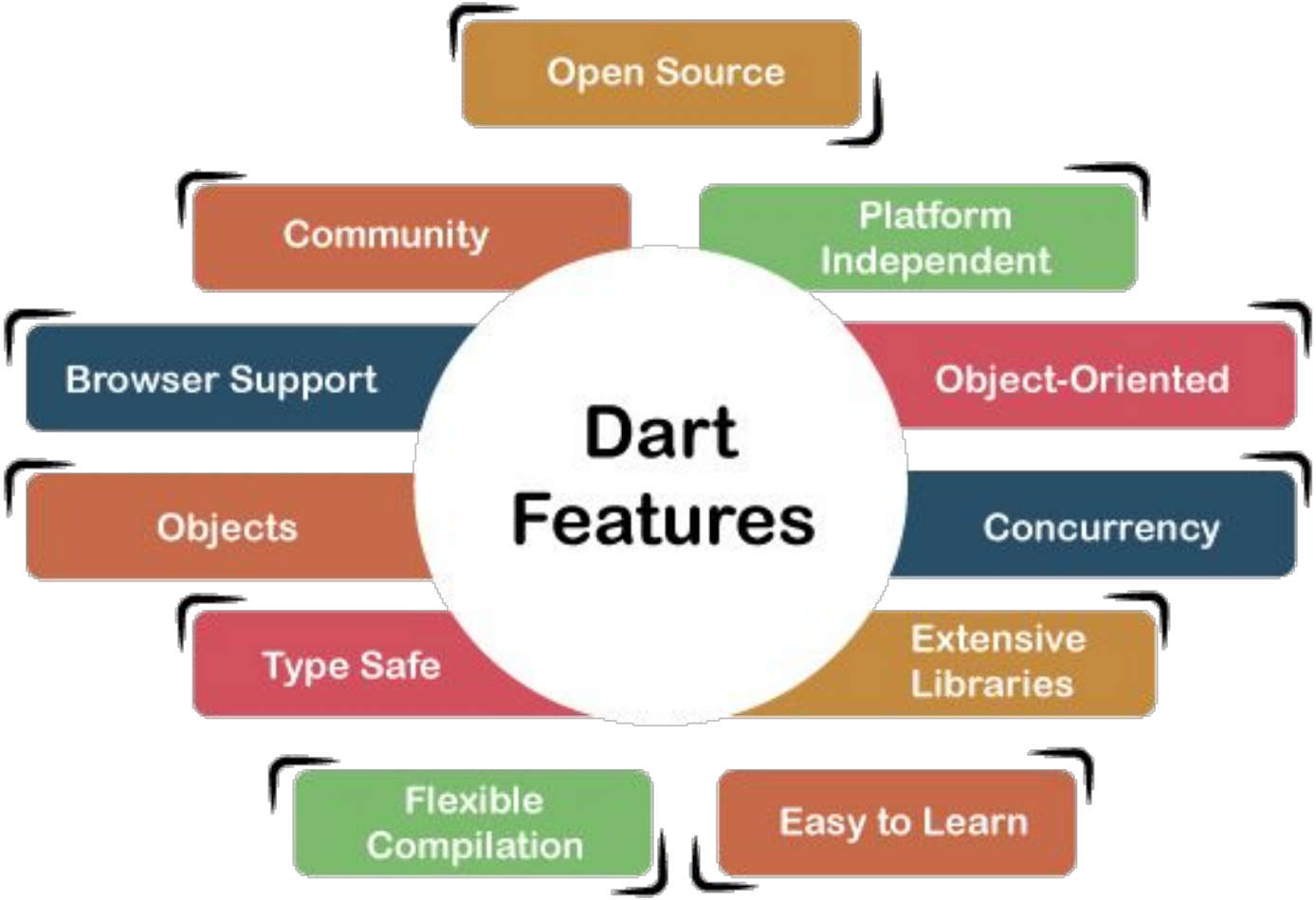
And v



And \

Advantages of Dart

I'm


Open Source
Community
Platform Independent
Browser Support
Object-Oriented
Dart Features
Objects
Concurrency
Type Safe
Extensive Libraries
Flexible Compilation
Easy to Learn

1. Speed

gRPC is roughly 7 times faster than REST when receiving data & roughly 10 times faster than REST when sending data for this specific payload. This is mainly due to the tight packing of the Protocol Buffers which reduces

gRPC uses HTTP/2 which provides better performance and reduced latency

2. Protocol Buffers

Protocol Buffers is a free and open-source cross-platform data format used to serialize structured data. It is an alternative to REST/GraphQL/etc.

- Used to define the format of the API.
- Define messages & RPC calls that will be available to the client & the server. It defines what methods will be visible on the generated client & server stubs.
- Broad language support
- Compact data storage
- Faster parsing(serialization & deserialization) thro generated classes

3. Streaming

gRPC offers streaming capabilities that enable

1. Client-Side Streaming
2. Server-Side Streaming(websockets)
3. Bidirectional Streaming

These streams are super fast when transmitting data since the overhead of creating/closing sockets is removed as compared to how REST APIs need to always open a new connection on each request.

3. Streaming

Once a bidirectional stream has started, streaming messages back and forth is faster than sending messages with multiple unary gRPC calls. Streamed messages are sent as data on an existing HTTP/2 request and eliminates the overhead of creating a new HTTP/2 request for each unary call. 11 Apr 2023

Microsoft Learn
https://learn.microsoft.com › en-us › aspnet › core › perf...

Performance best practices with gRPC - Microsoft Learn

4. Language agnostic

- C/C++
- C#
- Dart
- Go
- Java
- Kotlin
- Node.js
- Objective-C
- PHP
- Python
- Ruby
- Scala
- Rust

Implement your server in any language and let your client can invoke the server in any language.

2.1 How do we get started

1. Define what methods you want the server & client to have.
2. Define the RPC methods & messages you want to pass when sending a Request and what messages you want to return as Responses
3. Define the service name
4. Generate client & server stubs
5. Implement the server business logic
6. Serve the API on a port & host
7. Use the client generated stubs to send requests to the server.

2.2 Quote Service Example

Let’s create a service which will enable us to

1. Get a Quote
2. List N Quotes
3. Stream a new Quote to the client every [x] seconds
4. An open-ended Search for Quotes using Bidirectional Streaming
5. Client streams list of favorite quotes and the server responds once client has finished favoriting.

2.2 Quote Service Example

We start by defining the messages we want the client & server to send to each other as request and responses.

How do we do that? Protocol Buffers

```
service QuoteService {
    rpc GetQuote (GetQuoteRequest) returns (Quote);
    rpc StreamQuotes (StreamQuotesRequest) returns (stream Quote);
    rpc ListQuotes (ListQuotesRequest) returns (ListQuotesResponse);
    rpc FilterQuotes (stream FilterQuotesRequest) returns (stream FilterQuotesResponse);
}
```

```
syntax = "proto3";

package quotes;

service QuoteService {
    rpc GetQuote (GetQuoteRequest) returns (Quote);
    rpc StreamQuotes (StreamQuotesRequest) returns (stream Quote);
    rpc ListQuotes (ListQuotesRequest) returns (ListQuotesResponse);
    rpc FilterQuotes (stream FilterQuotesRequest) returns (stream FilterQuotesResponse);
}

message Quote {
    string id = 1;
    string text = 2;
    string author = 3;
}

message GetQuoteRequest {}

message StreamQuotesRequest {
    int32 streamIntervalInSeconds = 1;
}

message ListQuotesRequest {
    int32 number_of_quotes = 1;
}

message ListQuotesResponse {
    repeated Quote quotes = 1;
}

message FilterQuotesRequest {
    string keyword = 1;
}

message FilterQuotesResponse {
    repeated Quote quotes = 1;
}
```

2.3 Generate Server & Client Stubs

Meet protoc

protoc is a binary executable that allows programmers to auto-generate backing code for an API as defined in the API's . proto file.

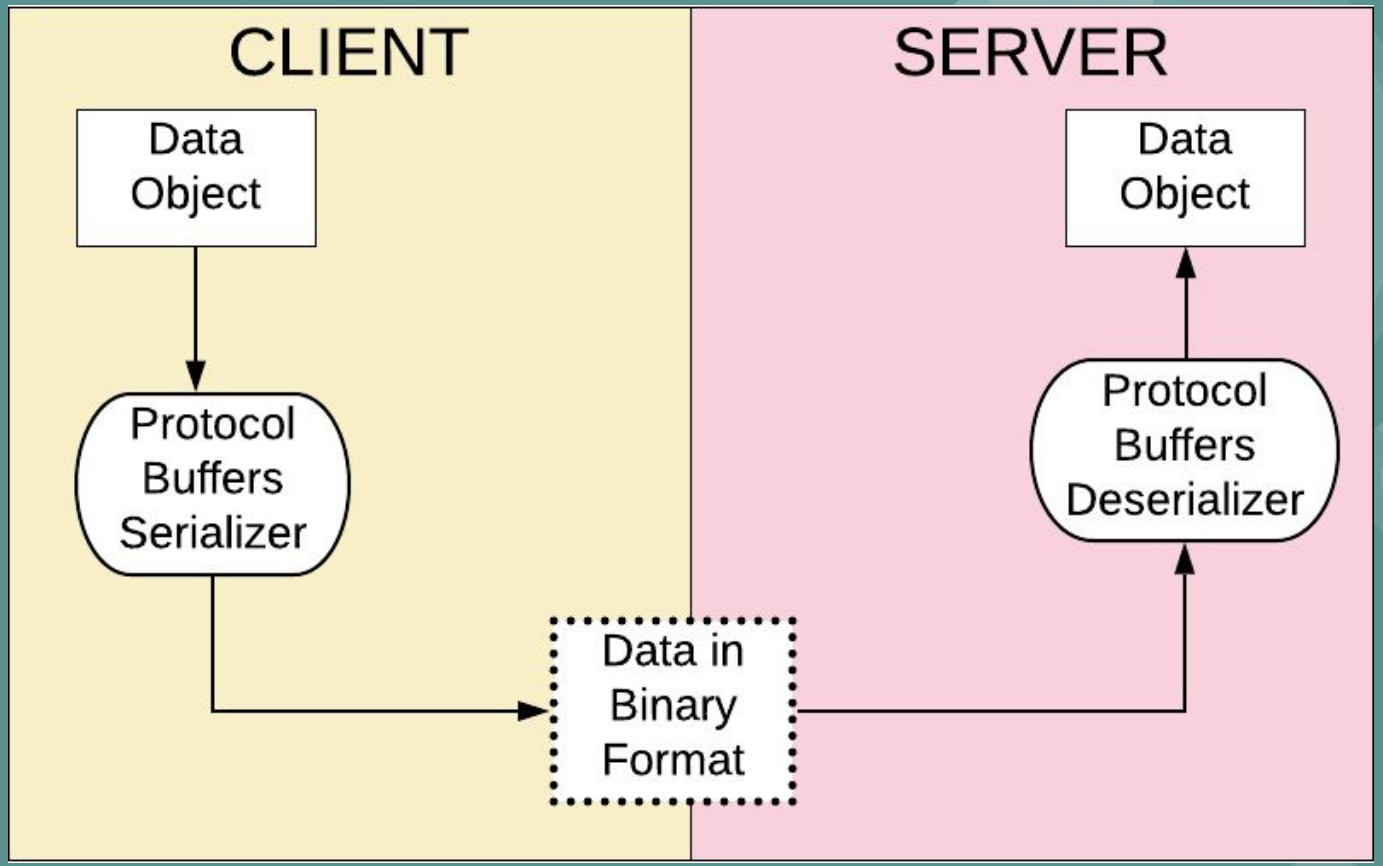
Given a protocol buffer definition in .proto format, we can run protoc to generate Server & Client stubs in any language.

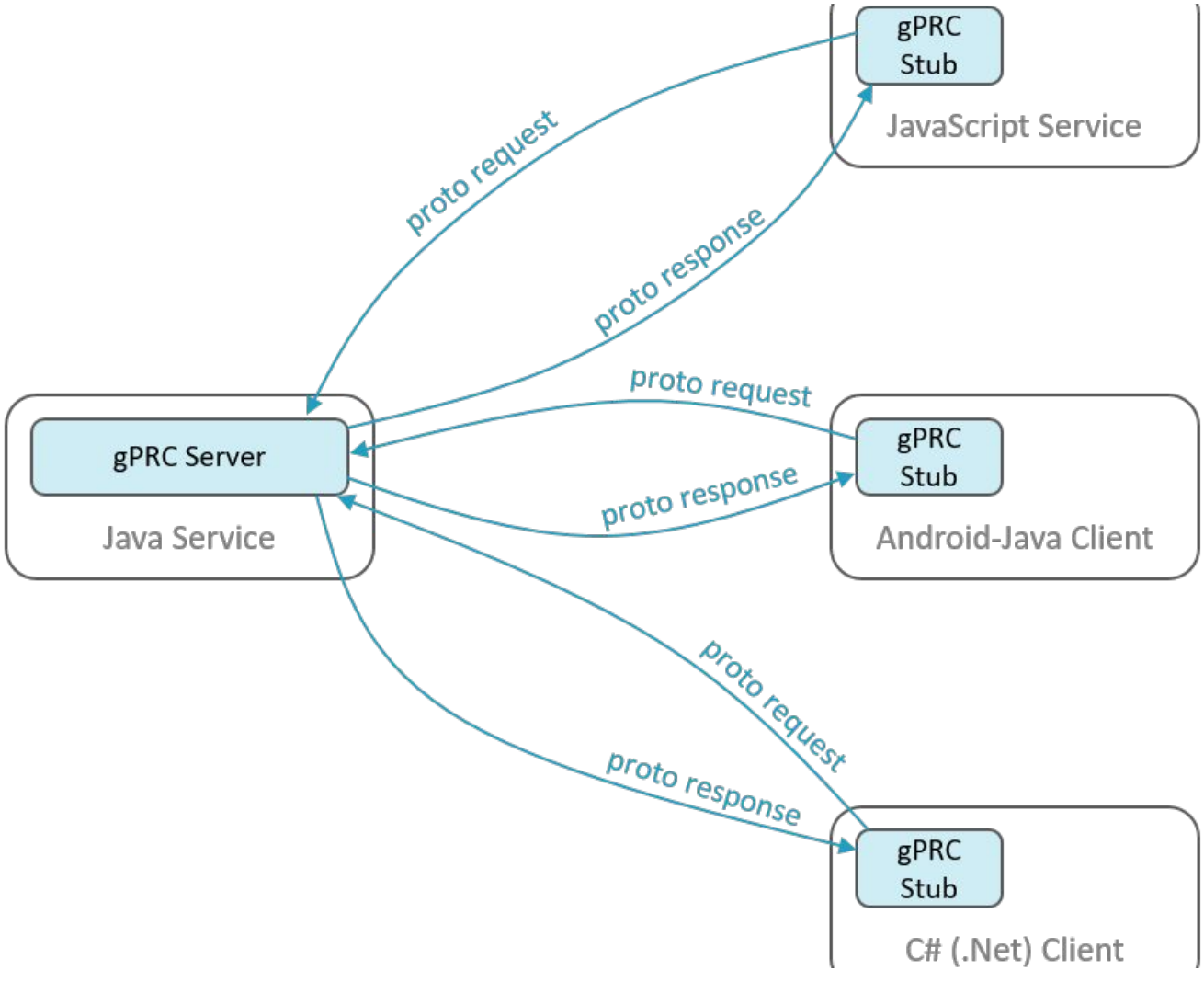
The ServerSide stubs are abstract classes which you'll extend/subclass in order to implement the business logic.

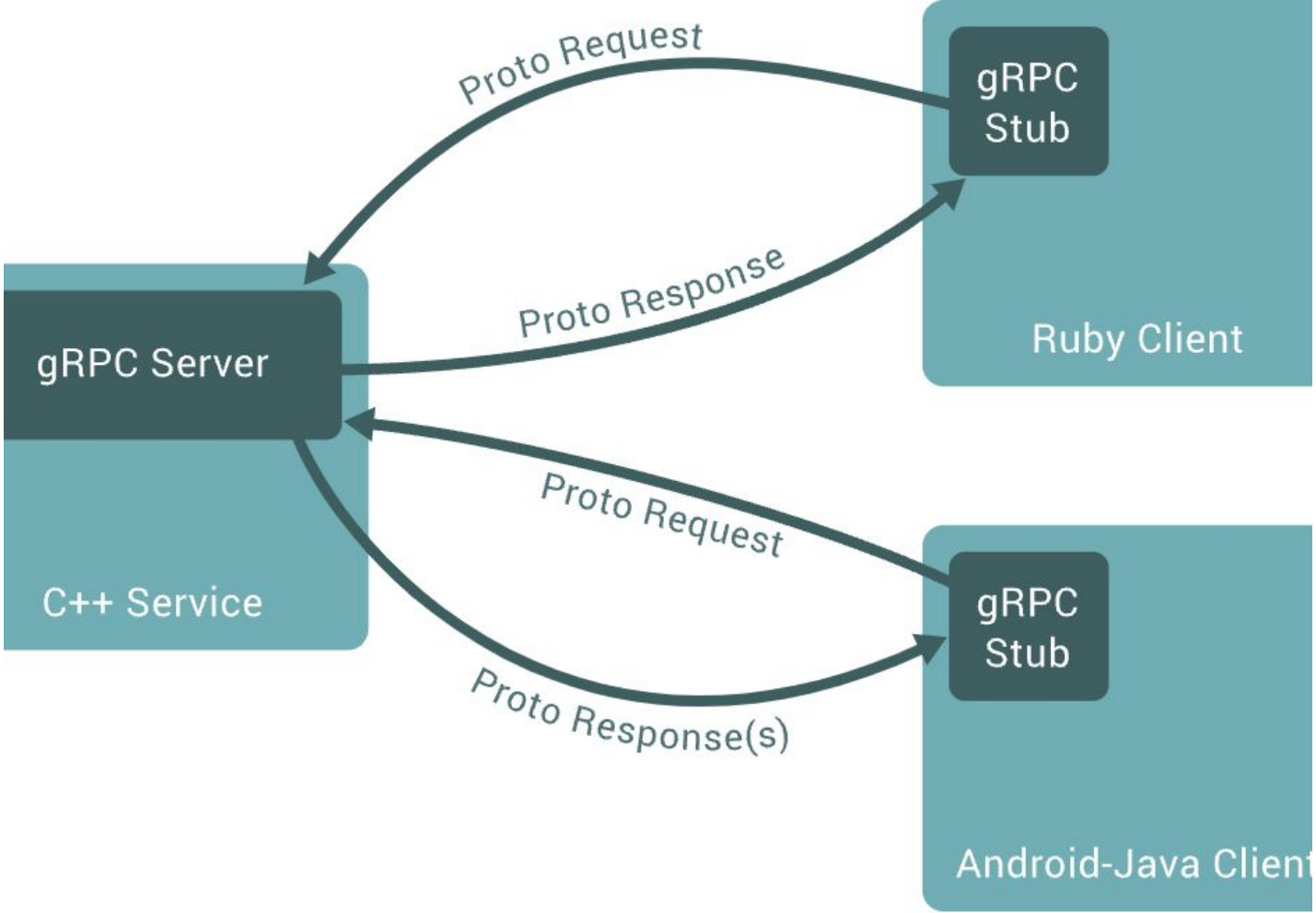
The Client-Side stubs are what you'll use to communicate with the server. The generated code are class instances which have getters, setters and the attributes we defined in the message.

The stubs are responsible for handling serialization & deserialization of data to binary format

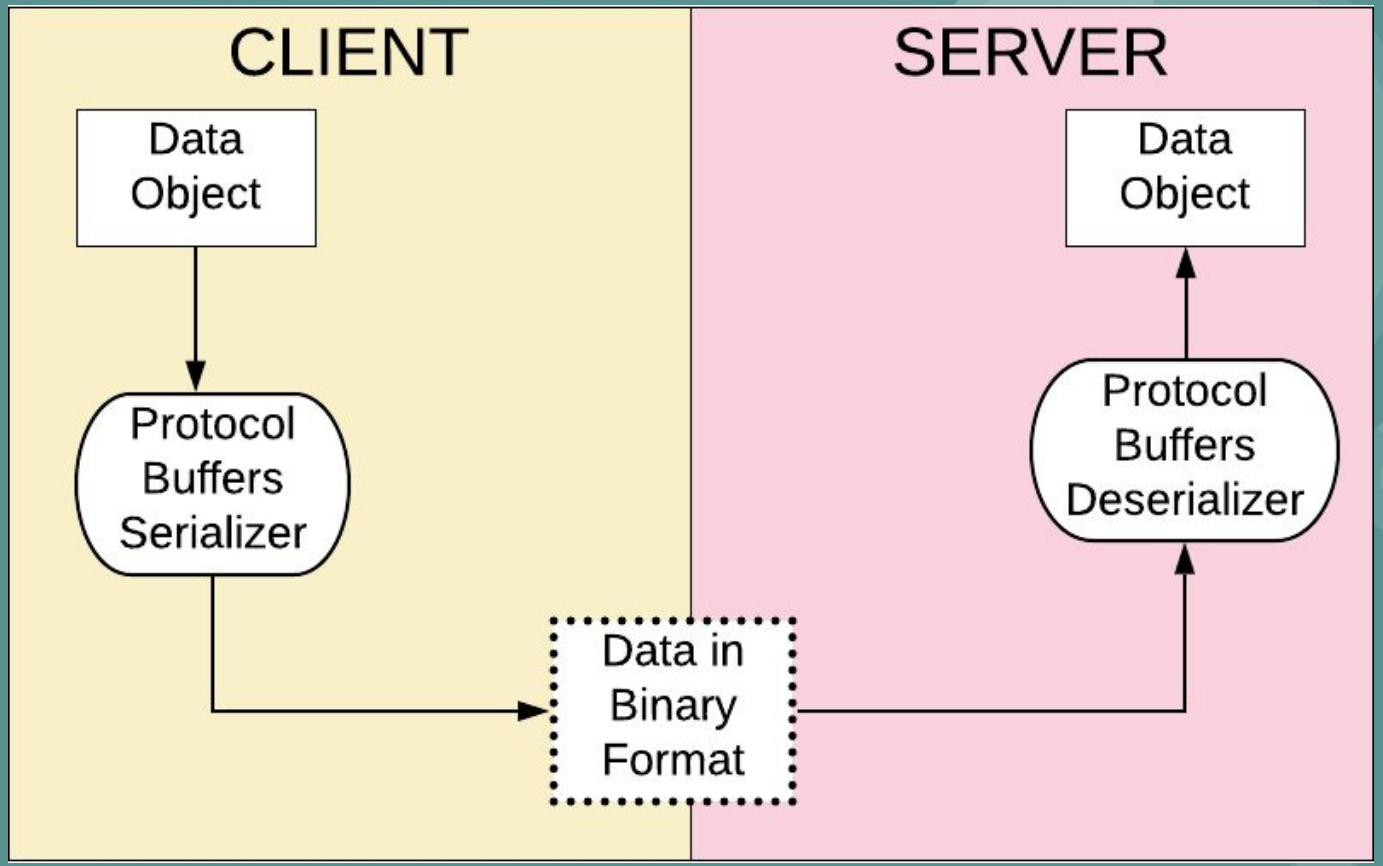
2.4. Diagrams explain it better




gPRC
Stub
JavaScript Service
proto request
proto response
proto request
gPRC Server
Java Service
proto response
gPRC
Stub
Android-Java Client
proto request
proto response
gPRC
Stub
C# (.Net) Client


Proto Request
gRPC Stub
Proto Response
Ruby Client
gRPC Server
Proto Request
gRPC Stub
C++ Service
Proto Response(s)
Android-Java Client

2.4. Diagrams explain it better



2.4. Generating Dart Code example

```
protoc -I protobufs/ --dart_out=grpc:server/lib/src/generated/ protobufs/quote_service.proto
```

2.4. Let's implement the API Service

```dart
import 'package:grpc/grpc.dart';
import 'package:grpg_streaming/src/generated/quote_service.pbgrpc.dart';

class QuoteService extends QuoteServiceBase {
  @override
  Stream<FilterQuotesResponse> filterQuotes(ServiceCall call, Stream<FilterQuotesRequest> request) {
    // TODO: implement filterQuotes
    throw UnimplementedError();
  }

  @override
  Future<Quote> getQuote(ServiceCall call, GetQuoteRequest request) {
    // TODO: implement getQuote
    throw UnimplementedError();
  }

  @override
  Future<ListQuotesResponse> listQuotes(ServiceCall call, ListQuotesRequest request) {
    // TODO: implement listQuotes
    throw UnimplementedError();
  }

  @override
  Stream<Quote> streamQuotes(ServiceCall call, StreamQuotesRequest request) {
    // TODO: implement streamQuotes
    throw UnimplementedError();
  }
}
```

2.4. GetQuote RPC

```
@override
Future<Quote> getQuote(
  ServiceCall call,
  GetQuoteRequest request,
) async {
  Quote quote = await quotesDb.getQuote();
  return quote;
}
```

2.4. ListQuotes RPC

```dart
@override
Future<ListQuotesResponse> listQuotes(
  ServiceCall call,
  ListQuotesRequest request,
) async {
  final numberOfQuotes =
      request.numberOfQuotes == 0 ? 10 : request.numberOfQuotes;
  final randomNQuotes = await quotesDb.getNQuotes(numberOfQuotes);
  return ListQuotesResponse(quotes: randomNQuotes);
}
```

2.4. StreamQuotes RPC

```
@override
Stream<Quote> streamQuotes(
  ServiceCall call,
  StreamQuotesRequest request,
) {
  int durationInSeconds = request.streamIntervalInSeconds;
  Stream<int> periodicStream = Stream.periodic(
    Duration(seconds: durationInSeconds),
    (count) => count,
  );
  periodicStream.doOnCancel(() => print('Client cancelled stream'));

  return periodicStream.asyncMap((event) async => await quotesDb.getQuote());
}
```

2.4. FilterQuotes RPC

```dart
@override
Stream<FilterQuotesResponse> filterQuotes(
  ServiceCall call,
  Stream<FilterQuotesRequest> request,
) async* {
  request.doOnCancel(() => print('Client cancelled stream'));
  await for (FilterQuotesRequest filterRequest in request) {
    final filter = filterRequest.keyword;
    List<Quote> filteredQuotes = await quotesDb.filterQuotes(filter);
    /// yield is used to emit a value as a response to the client
    yield FilterQuotesResponse(quotes: filteredQuotes);
  }
}
```

3.1 Let’s serve the API

```dart
import 'dart:developer';
import 'dart:io';

import 'package:grpc/grpc.dart';
import 'package:grpg_streaming/src/services/quote_service.dart';

void main(List<String> args) async {
  // Use any available host or container IP (usually `0.0.0.0`).
  final ip = InternetAddress.anyIPv4;
  final port = int.parse(Platform.environment['PORT'] ?? '8080');

  final server = Server.create(
    services: [
      // EventService(database: database),
      // UserService(database: database),
      QuoteService(),
    ],
  );

  // For running in containers, we respect the PORT environment variable.
  await server.serve(address: ip, port: port);
  print('Server listening on port ${server.port}');
}
```

3.2 Client-Side

Like any other APIs running on some HOST & PORT, we need to specify this information when specifying the channel we'll use.
The channel is like a HTTP client which can send requests to a specific server in REST Framework

```
ClientChannel channel = buildGrpcChannel(
  host: host,
  port: port,
  secure: false,
);
final client = QuoteServiceClient(channel);

Quote quote = await client.getQuote(GetQuoteRequest());
```

3.2 Client-Side [GetQuote]

Like any other APIs running on some HOST & PORT, we need to specify this information when specifying the channel we'll use.
The channel is like a HTTP client which can send requests to a specific server in REST Framework

```
ClientChannel channel = buildGrpcChannel(
  host: host,
  port: port,
  secure: false,
);
final client = QuoteServiceClient(channel);

Quote quote = await client.getQuote(GetQuoteRequest());
```

3.2 Client-Side [ListQuotes]

```
ClientChannel channel = buildGrpcChannel(
  host: host,
  port: port,
  secure: false,
);
final client = QuoteServiceClient(channel);
ListQuotesResponse response = await client.listQuotes(
ListQuotesRequest());
List<Quote> quotes = response.quotes;
```

3.2 Client-Side [StreamQuotes]

Here, we send a one-time request to the server requesting a stream of ever changing quotes. We specify a duration in milliseconds which specifies when's the next time we want the server to send a new quote.

```
ClientChannel channel = buildGrpcChannel(
  host: host,
  port: port,
  secure: false,
);
QuoteServiceClient client = QuoteServiceClient(channel);
Stream<Quote> stream = client.streamQuotes(
  StreamQuotesRequest(streamIntervalInSeconds: 4),
);
await for (final quote in stream) {
  print(quote);
}
```

3.2 Client-Side [FilterQuotes]

The aim is to have a low-latency autocomplete. Streams in gRPC are superfast which suit places which need auto-complete.

In our case, the client will pass a Stream of Events[**FilterQuotesRequest**] to the server and the server will return a Stream of Events[**FilterQuotesResponse**]. Once this stream is opened, the client & server will have a long-lived communication channel where they can send and receive messages back and forth using streams.

We'll use a StreamController[**BehaviourSubject in Java**] which will expose a Stream to us. When we give this stream to the server, it will wait for events added to this stream and respond back with the proper response.

In our case, we'll be passing keywords and the server will return a List of Quotes which will be wrapped in a [**FilterQuotesResponse**] as we defined in the protocol buffer definition.

3.2 Client-Side [FilterQuotes] example

```
class FilterQuotesController {
  FilterQuotesController({
    required this.quoteServiceClient,
  }) {
    _streamController = StreamController<String>();
    final filterStream = _streamController.stream.map(
      (event) => FilterQuotesRequest(keyword: event),
    );
    _quotesStream = quoteServiceClient.filterQuotes(filterStream);
  }

  final QuoteServiceClient quoteServiceClient;
  late StreamController _streamController;
  late Stream<FilterQuotesResponse> _quotesStream;

  Stream<FilterQuotesResponse> get quotesStream => _quotesStream;

  void filter(String keyword) async {
    _streamController.add(keyword);
  }

  void dispose() {
    _streamController.close();
  }
}
```

3.2 DEMO TIME

whoami

- Douglas Bett
- Currently enjoying Dart & Flutter.
- I got a passion for location based services & geographic information systems.

socials

@bettdouglas

@bettdoug

https://www.linkedin.com/in/douglas-bett/