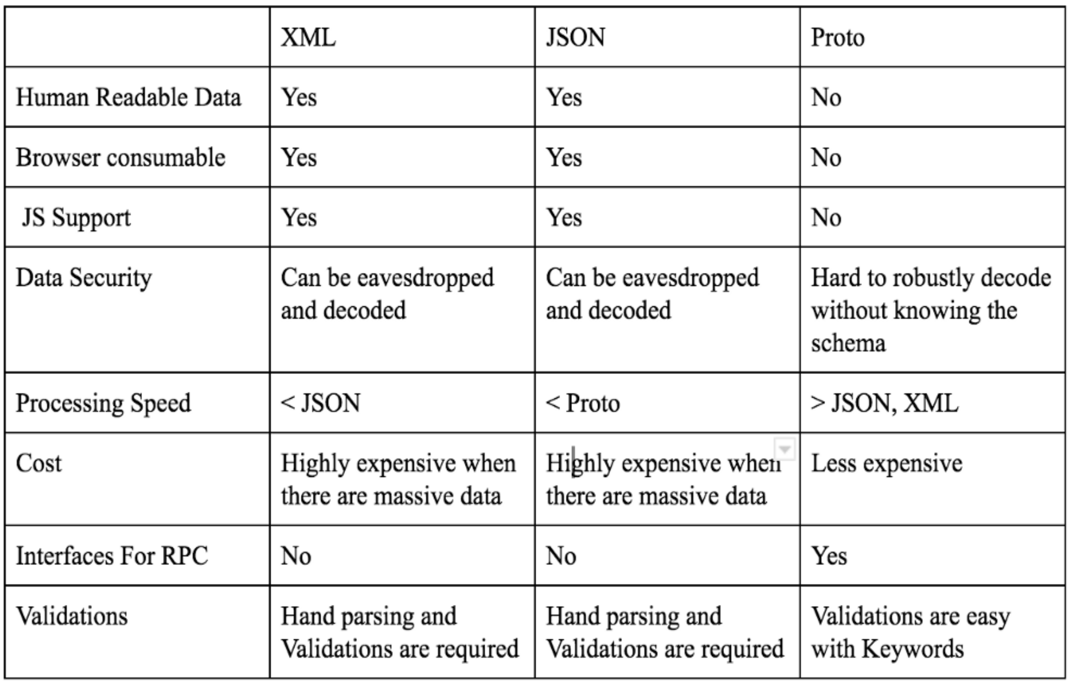
<Note Dalal refers to Dalal Street^{TM} Pragyan Manigma Event>

What RPC?
In distributed computing, a remote procedure call (RPC) is when a computer program causes a procedure (subroutine) to execute in a different address space (commonly on another computer on a shared network), which is coded as if it were a normal (local) procedure call, without the programmer explicitly coding the details for the remote interaction.
GRPC – Google RPC

Why GRPC?

Web infrastructure is already built on top of http
Bi-directional streaming is easy
Supports types and Validations
Faster



	XML	JSON	Proto
Human Readable Data	Yes	Yes	No
Browser consumable	Yes	Yes	No
JS Support	Yes	Yes	No
Data Security	Can be eavesdropped and decoded	Can be eavesdropped and decoded	Hard to robustly decode without knowing the schema
Processing Speed	< JSON	< Proto	> JSON, XML
Cost	Highly expensive when there are massive data	Highly expensive when there are massive data	Less expensive
Interfaces For RPC	No	No	Yes
Validations	Hand parsing and Validations are required	Hand parsing and Validations are required	Validations are easy with Keywords

In RPC there are 4 types of service methods:-

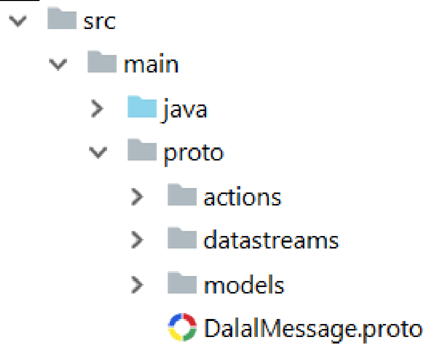
1. Simple RPC (DalalActionService)
2. Client-side streaming (DalalStreamService)
3. Server-side streaming (DalalStreamService)
4. Bidirectional streaming (DalalStreamService)

**In Dalal we'll use simple RPCs and Server-side streaming RPC

With gRPC we can define our service once in a .proto file and implement clients and servers in any of gRPC's supported languages. We also get all the advantages of working with protocol buffers, including efficient serialization, a simple IDL, and easy interface updating.

Proto files define what you'll be passing as Request and what Response you should expect gRPC largely follows HTTP semantics over HTTP/2 but we explicitly allow for full-duplex streaming. It diverges from typical REST conventions as it uses static paths for performance reasons during call dispatch as parsing call parameters from paths, query parameters and payload body adds latency and complexity. It also has formalized a set of errors.

Dir



```
src
  main
    java
    proto
      actions
      datastreams
      models
      DalalMessage.proto
```

DalalMessage.proto contains all the methods that can be called and what all is sent as request and the return value, for example

```
service DalalActionService {

        // Stock trading related functions
  rpc BuyStocksFromExchange(actions.BuyStocksFromExchangeRequest) returns (actions.BuyStocksFromExchangeResponse);
        .
        .
        .
}
```

/actions proto files contain Request and Response message for simple RPC
/datastreams for streaming RPC
/models contain definition of objects, example

```
syntax = "proto3";

package dalalstreet.api.models;

message Stock {
  uint32 id = 1;
  string short_name = 2;
  string full_name = 3;
  string description = 4;
  uint32 current_price = 5;
  uint32 day_high = 6;
  uint32 day_low = 7;
  uint32 all_time_high = 8;
  uint32 all_time_low = 9;
  uint32 stocks_in_exchange = 10;
  uint32 stocks_in_market = 11;
  bool up_or_down = 12;
  uint32 previous_day_close = 13;
  uint32 avg_last_price = 14;
  string created_at = 15;
  string updated_at = 16;
}
```

Learn Protocol Buffers In-Depth:-
https://developers.google.com/protocol-buffers/docs/overview
https://developers.google.com/protocol-buffers/docs/proto3

Once you've defined your messages, you run the protocol buffer compiler for Go on your .proto file to generate data access classes. These provide simple accessors for each field (like name() and set_name()) as well as methods to serialize/parse the whole structure to/from raw bytes.

So, the grpc **stub classes** are generated when you run the protoc compiler and it finds a service declaration in DalalMessage.proto file. The stub classes are the API your client uses to make rpc calls on the service endpoint.

These stubs come in two flavours: blocking and normal (async)

Blocking stubs are synchronous (block the currently running thread) and ensure that the rpc call invoked on it doesn't return until it returns a response or raises an exception. Care should be taken not to invoke an rpc on a blocking stub from the UI thread as this will result in an unresponsive/janky UI.

Asynchronous stubs make non-blocking rpc calls where the response is returned asynchronously via a StreamObserver callback object.

We first need to create a gRPC *channel*. While building channel we pass server.crt and .key files. Once the gRPC *channel* is setup, we **finally** get a client *stub* to perform RPCs.

Go Quick reference:-

https://grpc.io/docs/quickstart/go.html

Reference for Android: Contains .proto files, generation of stubs via protoc, creation of stubs, channel and finally calling methods

https://github.com/Harsh2098/Android-Multiplayer-Tutorial