API Paradigms

Matthew Rindner

{ REST:API }

GraphQL

gRPC

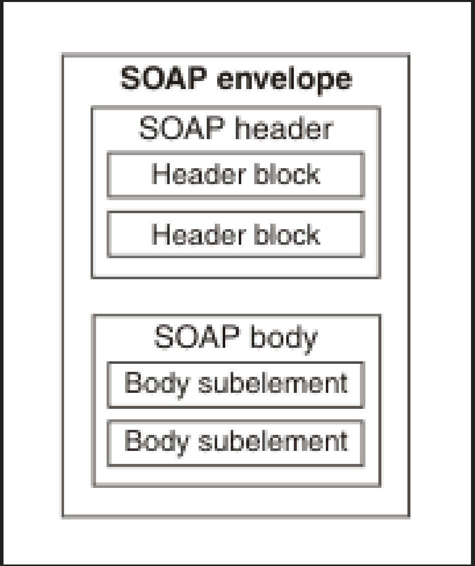
Background

- API stands for Application Programming Interface.
- APIs allow different software applications to communicate with each other.
- There are several types of APIs paradigms, including RESTful APIs, GraphQL, SOAP APIs and RPCs.

SOAP web services

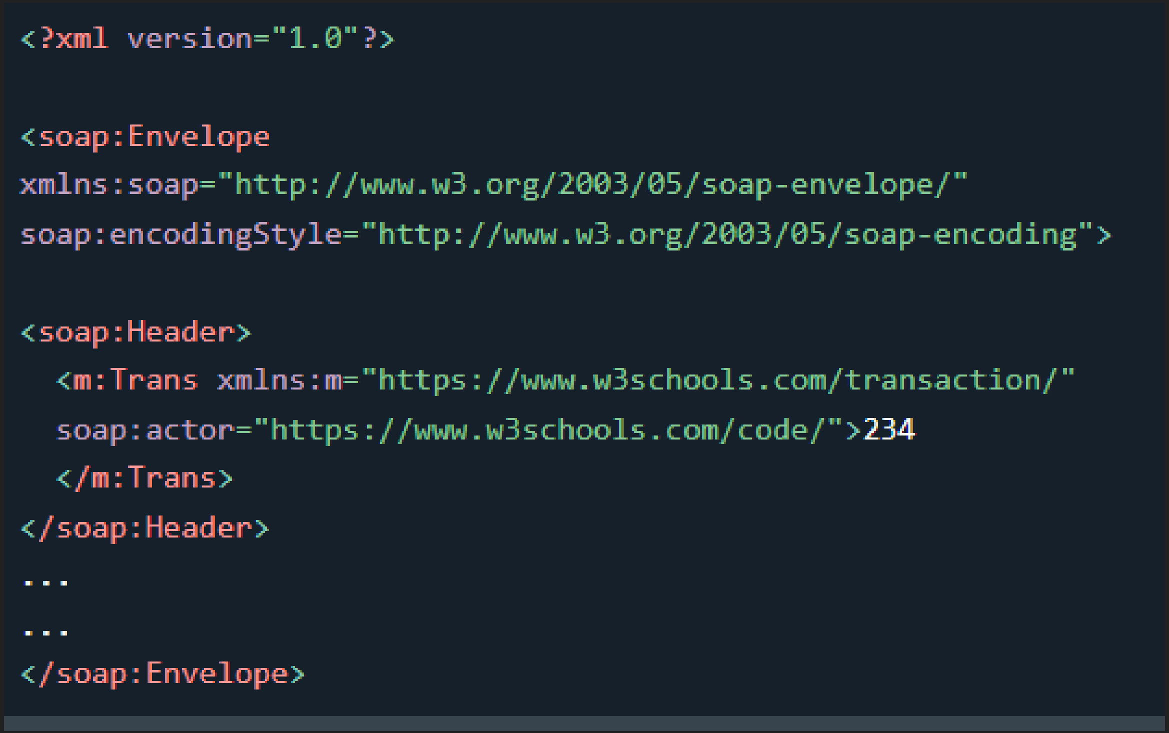
(Simple* Object Access Protocol)

- platform independent
- Built around rpc
- Based on XML - organized by tags in hierarchy tree (DOM)
- Uses CRUD HTTP methods (GET, POST, DELETE)



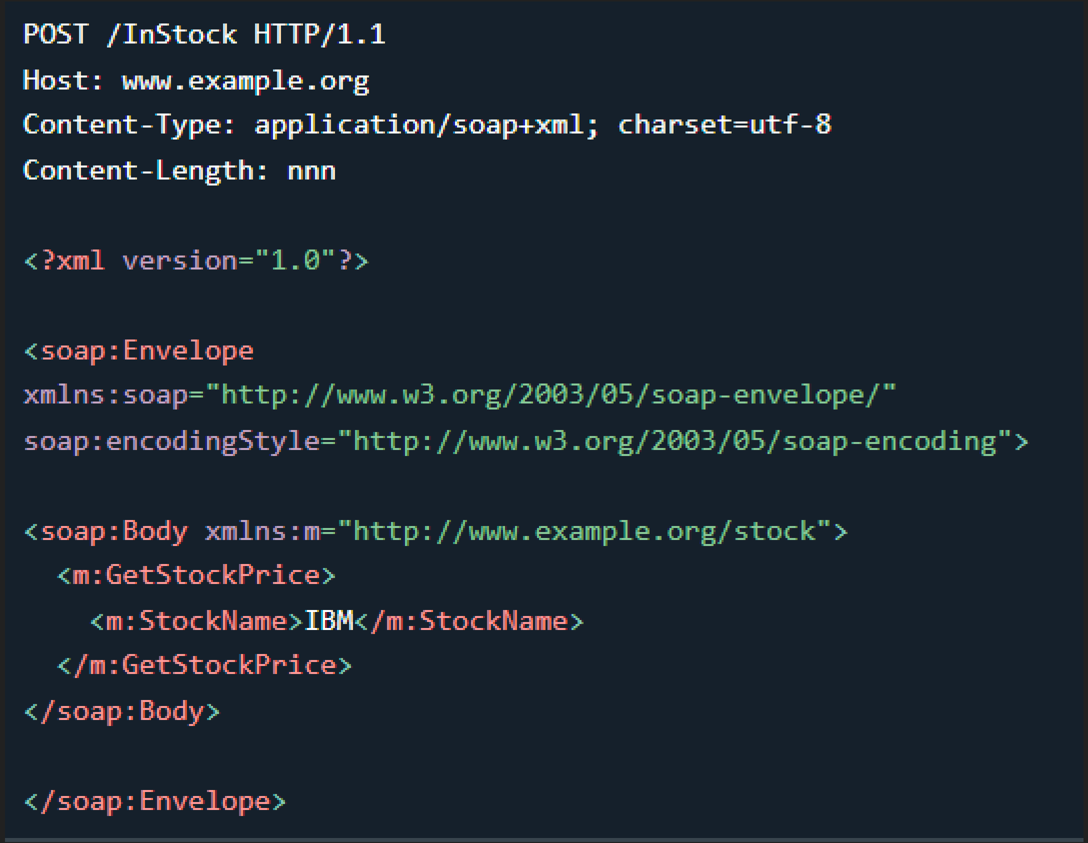
- <Envelope> - root element in DOM
- <header> - contains application-specific information (authentication, payment, etc)
- <Body> - contains the actual SOAP message intended for the endpoint of the message.
    - Contains optional <fault> tags

SOAP header block



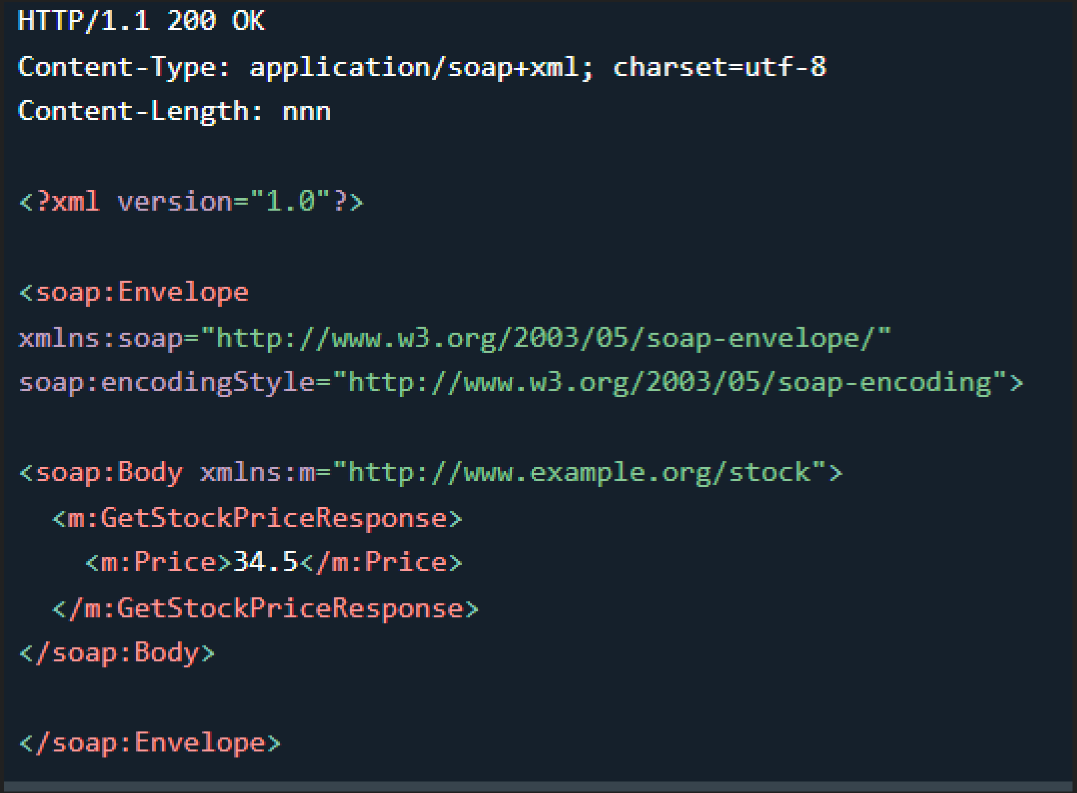
```xml
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Header>
  <m:Trans xmlns:m="https://www.w3schools.com/transaction/"
  soap:actor="https://www.w3schools.com/code/">234
  </m:Trans>
</soap:Header>
...
...
</soap:Envelope>
```

SOAP request



```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

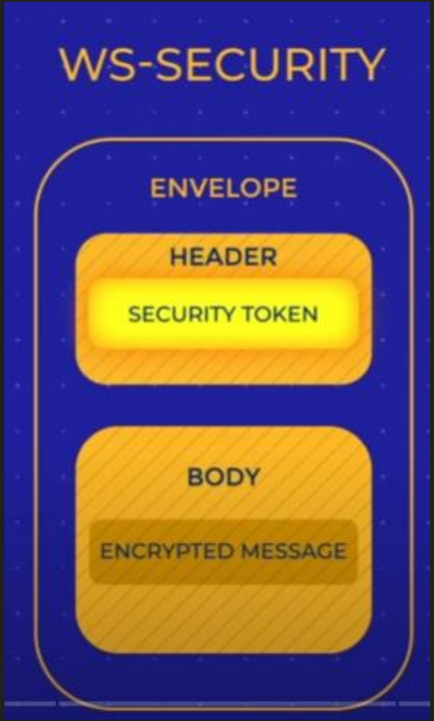
SOAP response



```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

SOAP Security

- SOAP APIs will authenticate and authorize the API calls,
- Web Service-Security is a SOAP extension that provides a number of security features for SOAP APIs.
  - describes how to sign and encrypt SOAP messages
  - Built on XML encryption
- Most importantly, WS-Security enables end-to-end security, authorization of senders
- Operation chaining

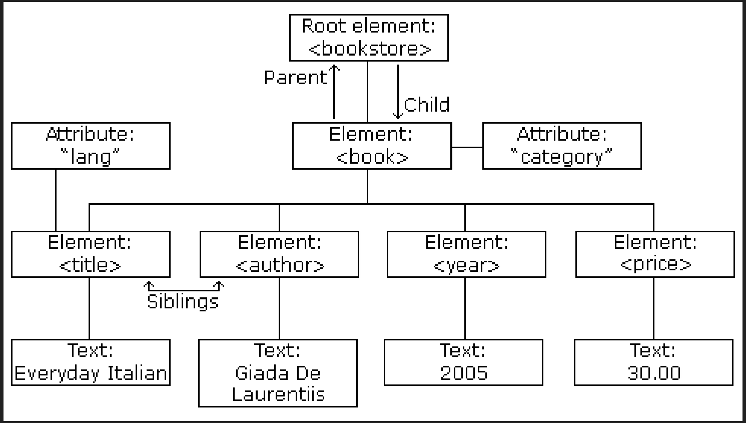


SOAP cons

- SOAP is a standard protocol with strict rules
  - Not flexible
- SOAP APIs forgo performance speed for higher complexity (Security).
  - Operation change is server-side taxing
- Has tight coupling between server and clients
  - To update a message property, must change both client and server
  - Harder to maintain
- XML message structure VERY verbose

XML

- Large XML files can take exponential space
  - DOM + opening/closing tags
- Needs processing power/dedicated tool to parse xml data into required format
- XML’s broad scope
  - Are limited by imagination (what you wean to type)



JSON

- JSON is lightweight and easy to parse
  - Its string and arrays
- Supported by all browsers
- Seamlessly works with AJAX
- Smaller size means faster transmission time/space
- Human readable

REST - Representational State transfer

REST is a specification that dictates how distributed systems communicate with each other

- Resource based
- Language agnostic
- Stateless
- Cacheable

REST suggests boundaries instead of a rigid structure

- Uniform interface
- Client server autonomy
- Layered system architecture
- Stateless interactions
- HTTP caching

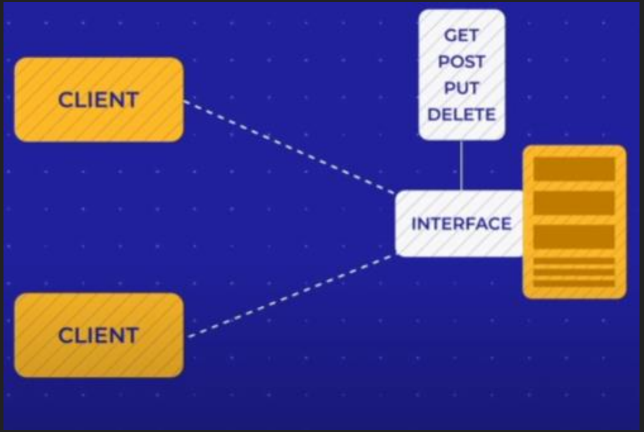
REST constraints

Uniform interface

- One naming convention - nouns
- One endpoint format
  - uri + http method + resource
- One data format - JSON

Client server autonomy

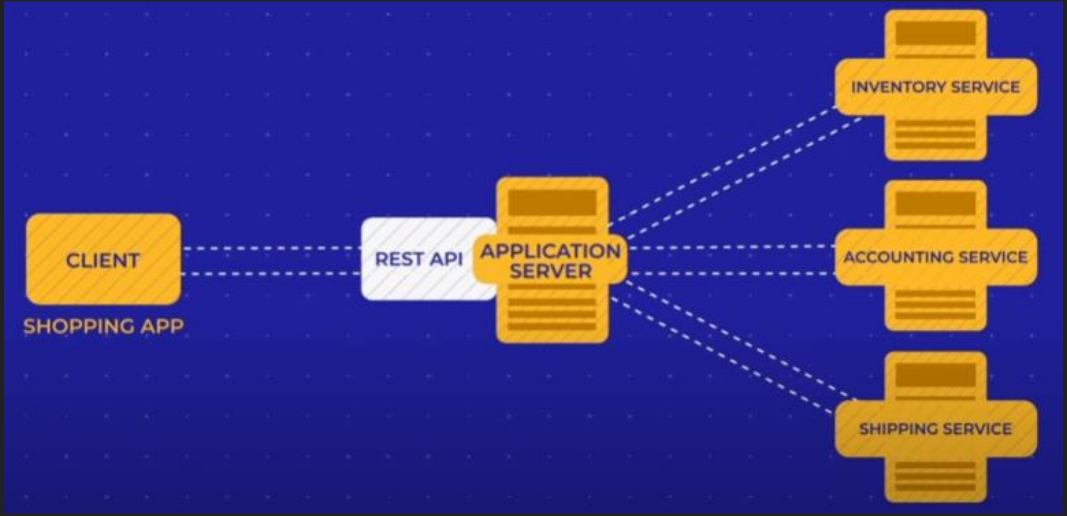
- Client and server implementation is abstracted
- Allows for server and client changes without affecting communication



REST constraints

Layered system Architecture

- Given a system have multiple servers
- Clients will only interact with the API on the app server
- App server will aggregate the reply for the client
- Allows for greater server scalability



REST constraints

Stateless Interactions

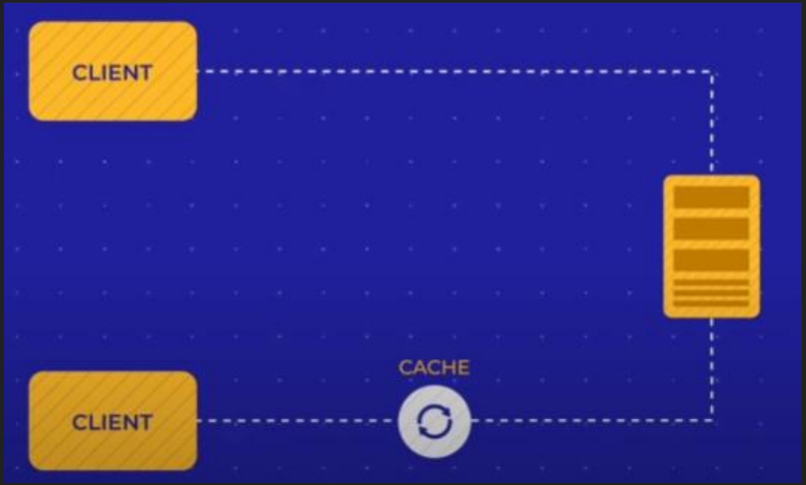
- Treat every request as a new request
- No past session or request data is stored
  - Different from SOAP
  - Increase server side efficiency
- Makes web app easy to scale

Code On Demand

- Not commonly used
- Client asks server for key generating code
- API will fetch code from the server
  - Runs code on client side

Caching

- clients can retain content and reduce load on servers

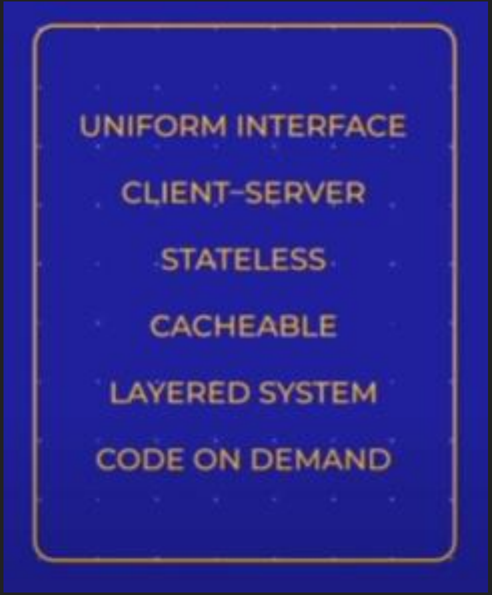


RESTful web service

Constraints enforce simplicity -> easier development

REST is an standard so an api that meets the stand of REST is a RESTful API

Not necessary to enforce every constraint



UNIFORM INTERFACE

CLIENT-SERVER

STATELESS

CACHEABLE

LAYERED SYSTEM

CODE ON DEMAND

Resource allocation

- Each resource is assigned to a specific url/uri
    - Ex. https://example.com/api/v3/products
    - Protocol + host address + path to resource

- Each resource must be a noun
    - /products (Good) /getProducts (Bad)
    - HTTP methods will act as the verb

HTTP methods

CRUD = Create, read, update, delete

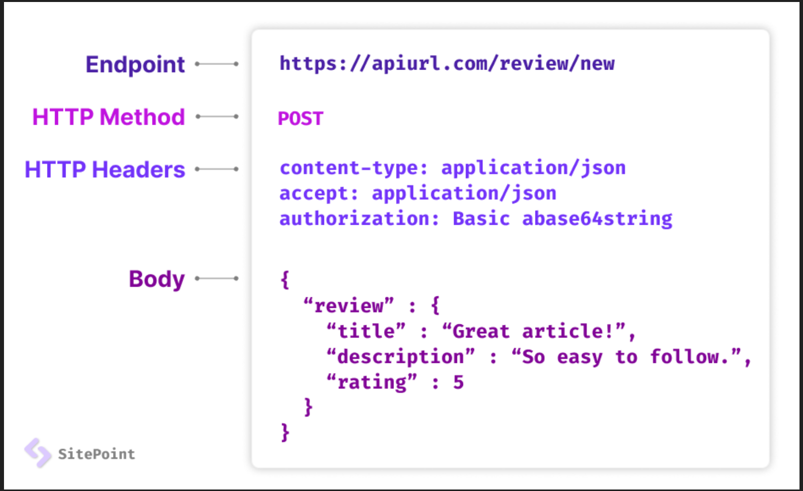
- GET - retrieve data from a specified resource okay next we have a
- POST – submit / create data to be process to a specific resource (forms)
- PUT - update a specified resource (adjax)
- DELETE - delete a specified resource

Others

- HEAD - same as get, but does not returns a body, only header
- OPTIONS - see the supported http methods
- PATCH - update partial resources

REST HTTP requests

The requests must also follow a specific format

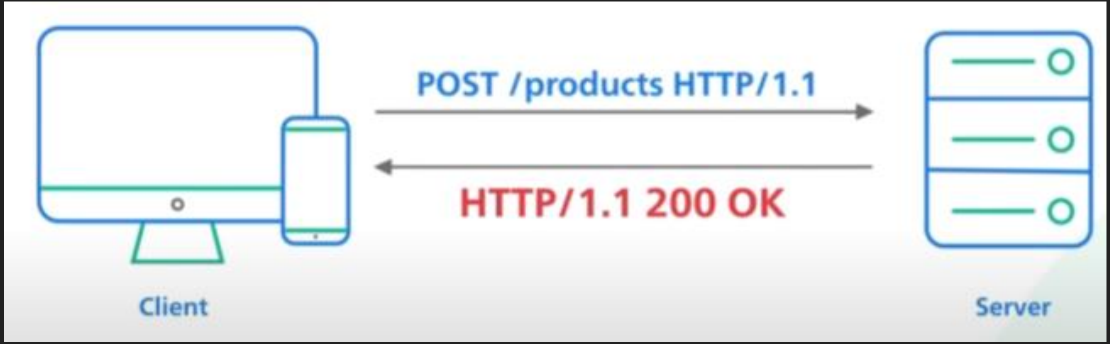


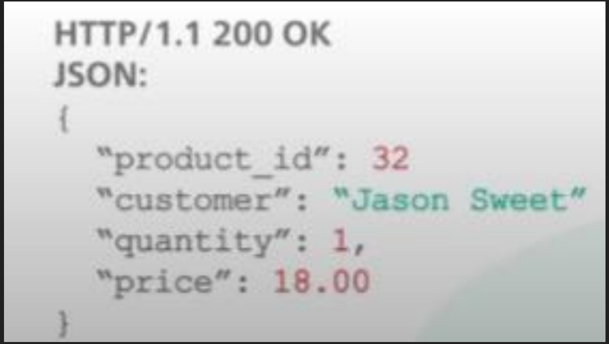
```
https://apiurl.com/review/new

POST

content-type: application/json
accept: application/json
authorization: Basic abase64string

{
  "review" : {
    "title" : "Great article!",
    "description" : "So easy to follow.",
    "rating" : 5
  }
}
```

URI	Used to identify the resource via an endpoint. Ex. URL
Method	HTTP verb (GET, POST,...).
Protocol	Typically HTTP/1.0 or 2.0.
Header	Metadata (message format, cache settings, request authorization, cookies).
Body	Payload being sent as a JSON object (Ex. parameters). (optional)

REST HTTP response

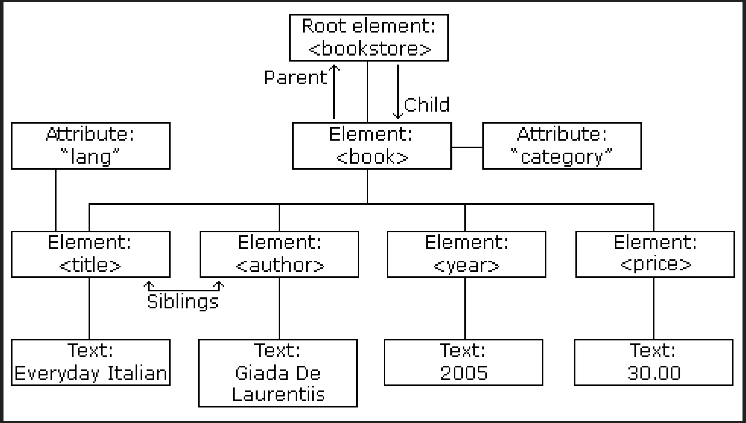
Protocol	Typically HTTP/1.0 or 2.0.
HTTP status code	200, 400, 500
Header	Metadata (content length, content type, date)
Response body	return data in JSON




```
HTTP/1.1 200 OK
JSON:
{
  "product_id": 32
  "customer": "Jason Sweet"
  "quantity": 1,
  "price": 18.00
}
```

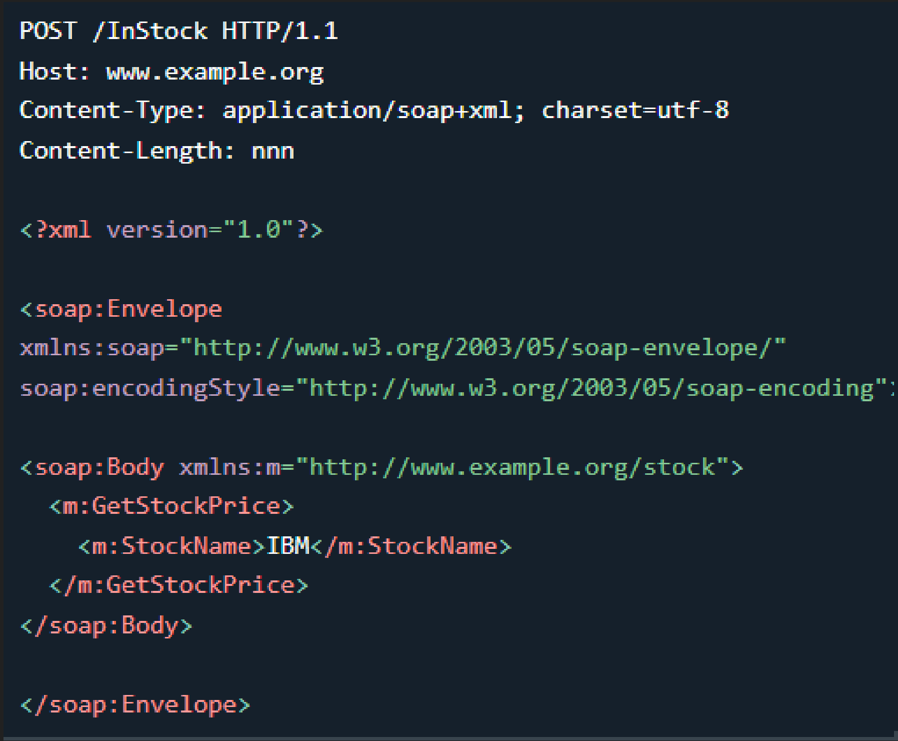
XML vs JSON

- Large XML files can take exponential space
  - DOM + opening/closing tags
- Needs processing power/dedicated tool to parse xml data into required format
- XML’s broad scope
  - Are limited by imagination (what you wean to type)



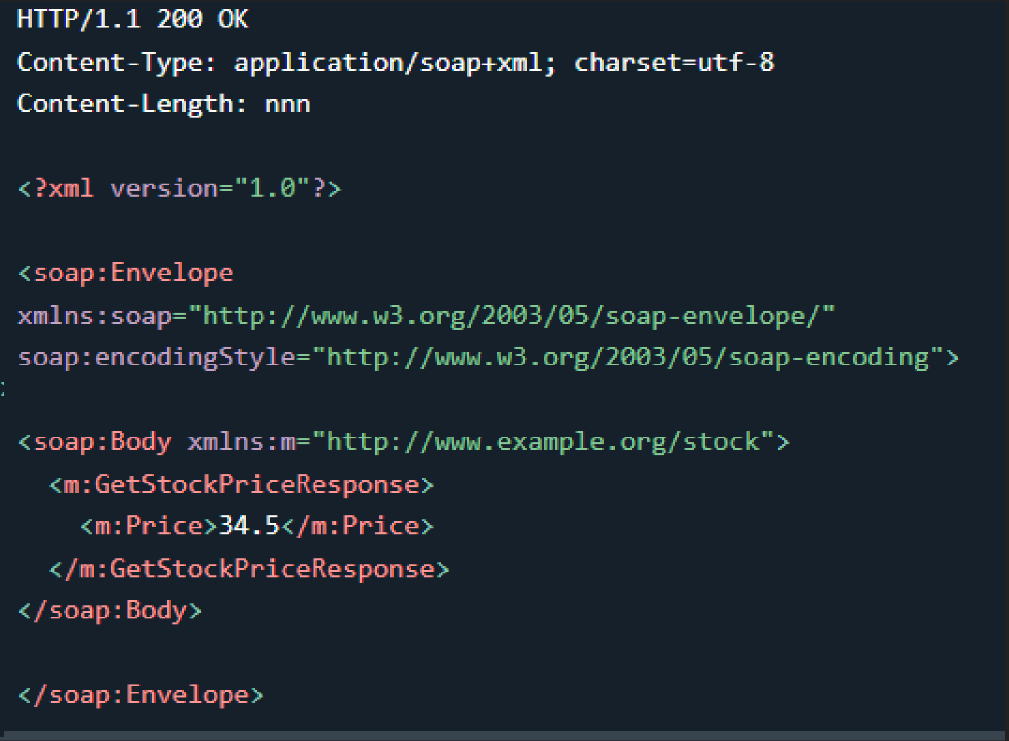
- JSON is lightweight and easy to parse
  - Its string and arrays
- Supported by all browsers
- Seamlessly works with AJAX
- Smaller size means faster transmission time/space
- Human readable

SOAP request



```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding"

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

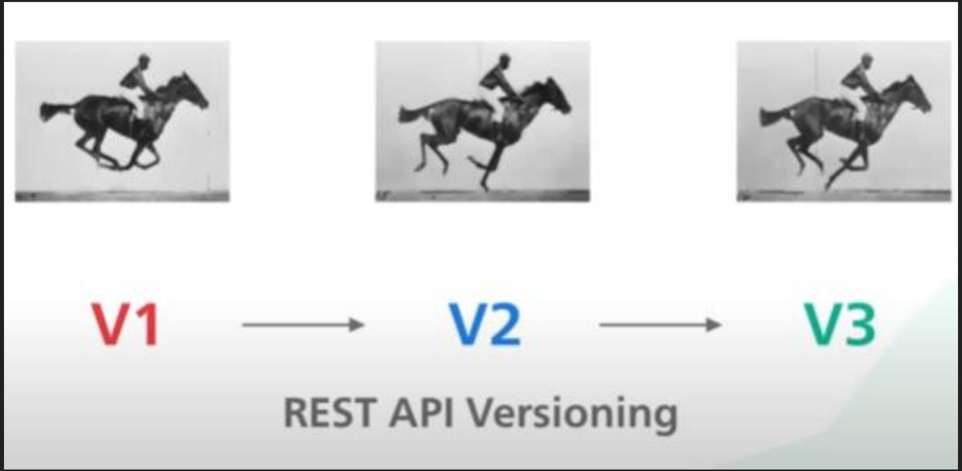
SOAP response



```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Versioning

- Versioning allows an implementation to provide backward compatibility
  - if we introduce breaking changes from one version to another, clients have time to move to new version
- Done by prefixing the uri



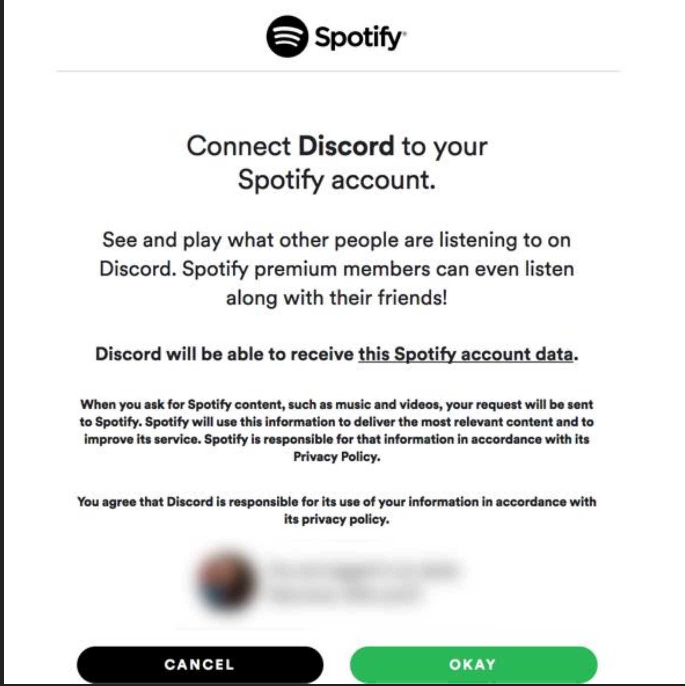
Authentication in REST

Uses OAuth 2.0 framework

- No password sharing
- Ability to revoke access to application individually
- Thus users have limited access to resources

Use of a separate authentication server

- Client makes OAuth request => is granted/denied
- Resource server sends client an access token
  - Token determines what resources client can access
- Protected resource is sent back to client



Connect **Discord** to your Spotify account.

See and play what other people are listening to on Discord. Spotify premium members can even listen along with their friends!

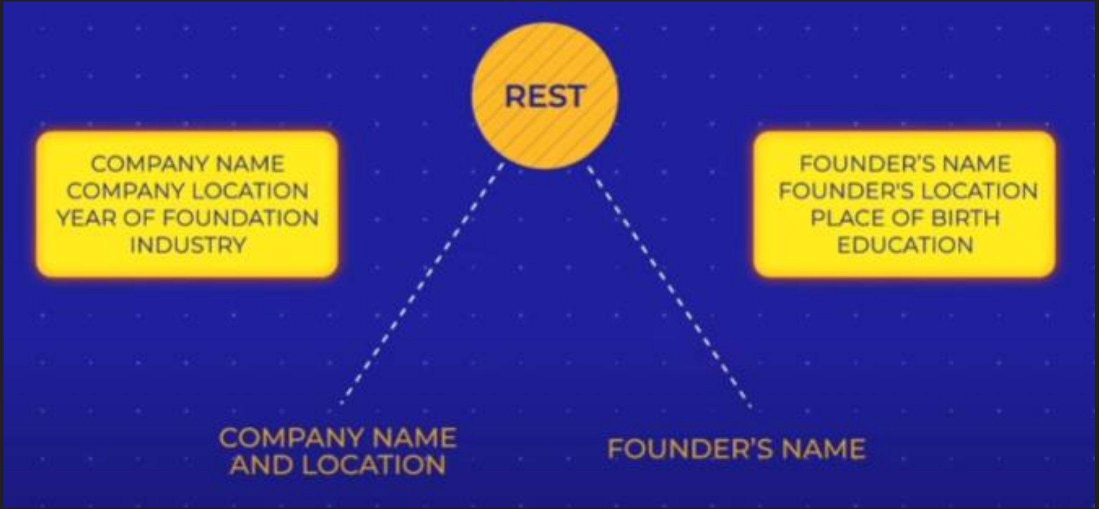
Discord will be able to receive this Spotify account data.

When you ask for Spotify content, such as music and videos, your request will be sent to Spotify. Spotify will use this information to deliver the most relevant content and to improve its service. Spotify is responsible for that information in accordance with its Privacy Policy.

You agree that Discord is responsible for its use of your information in accordance with its privacy policy.

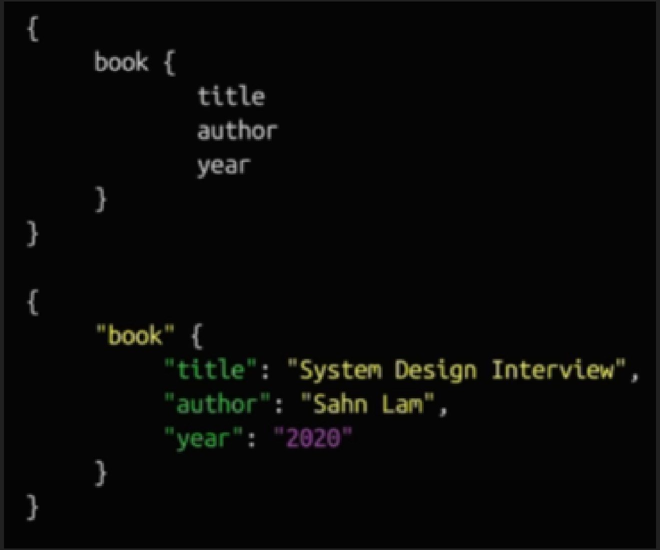
cons

- JSON properties are not strongly typed
  - Further logic to parse and convert properties into data types (uint32, chars, float,..)
- Overfetching - need to make multiple API requests to collect required data
  - This will increase payload size



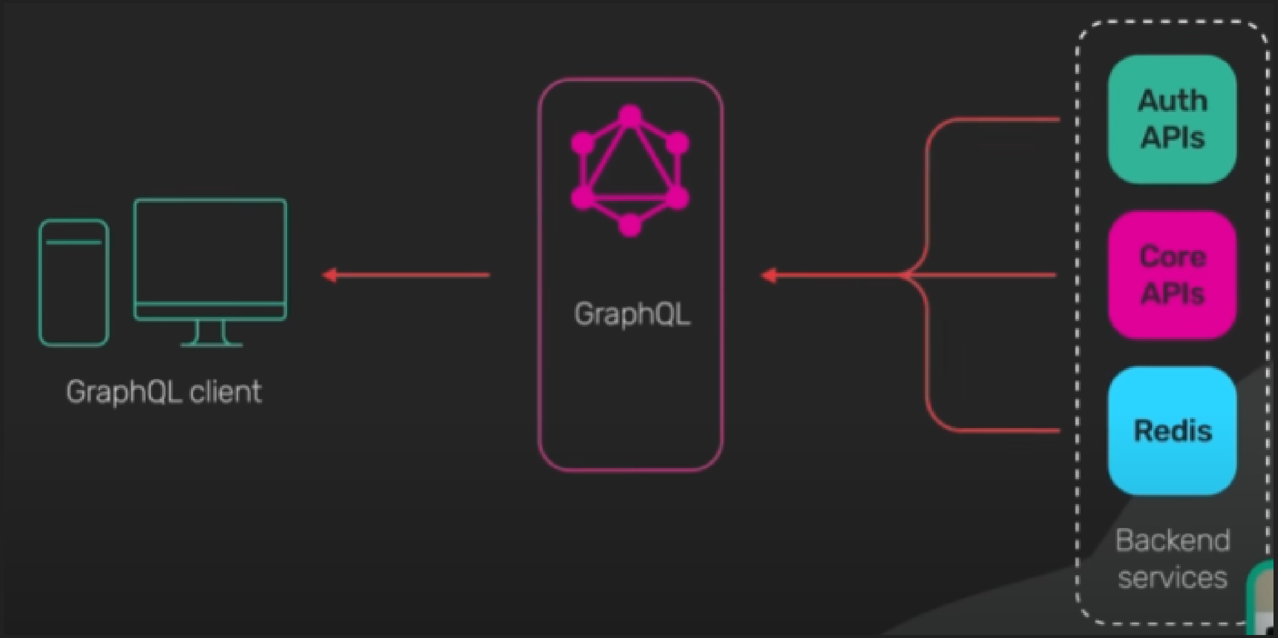
GraphQL

- API framework developed by Meta/Facebook in 2015
- A query language for APIs
- No under/over fetching of data
- Data in request and responses are strongly typed
- It is language and HTTP agnostic.
- Very flexible



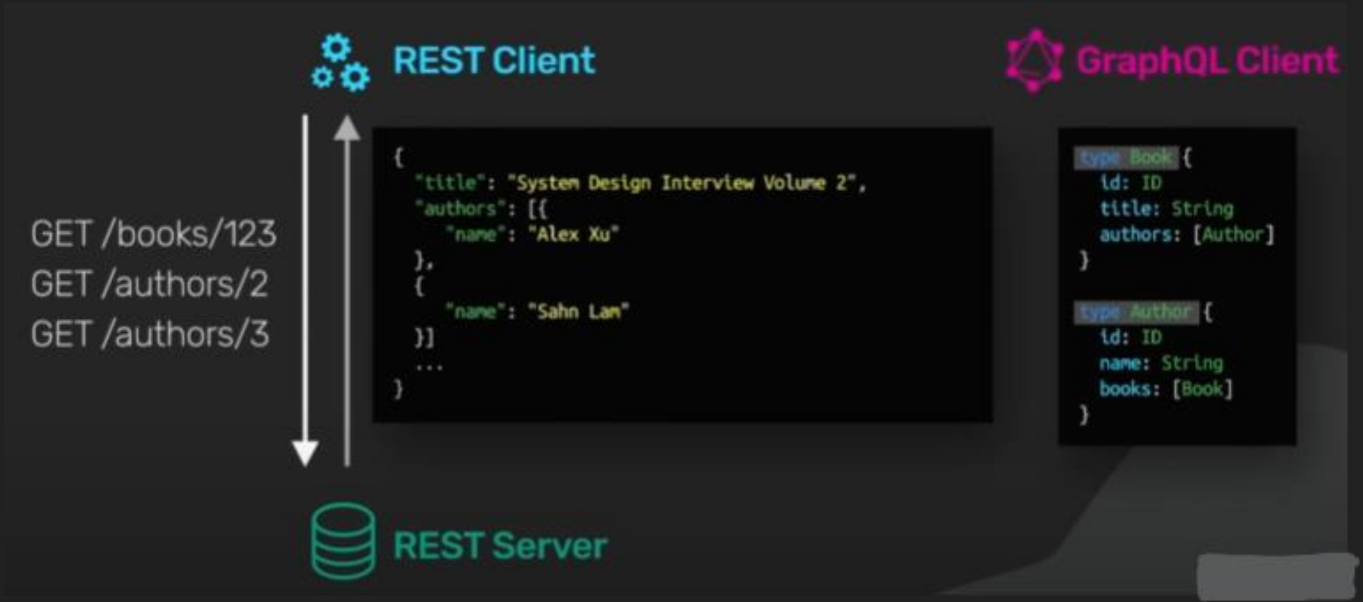
```
{
    book {
        title
        author
        year
    }
}

{
    "book" {
        "title": "System Design Interview",
        "author": "Sahn Lam",
        "year": "2020"
    }
}
```

- GraphQL can decouple frontend from backend.
- Is an application layer server-side technology



The BIG difference from REST

- Access any resource through one endpoint, the /graphQL.
- The client will define the structure, schema, of required data and the server will return the exact JSON data from the schema

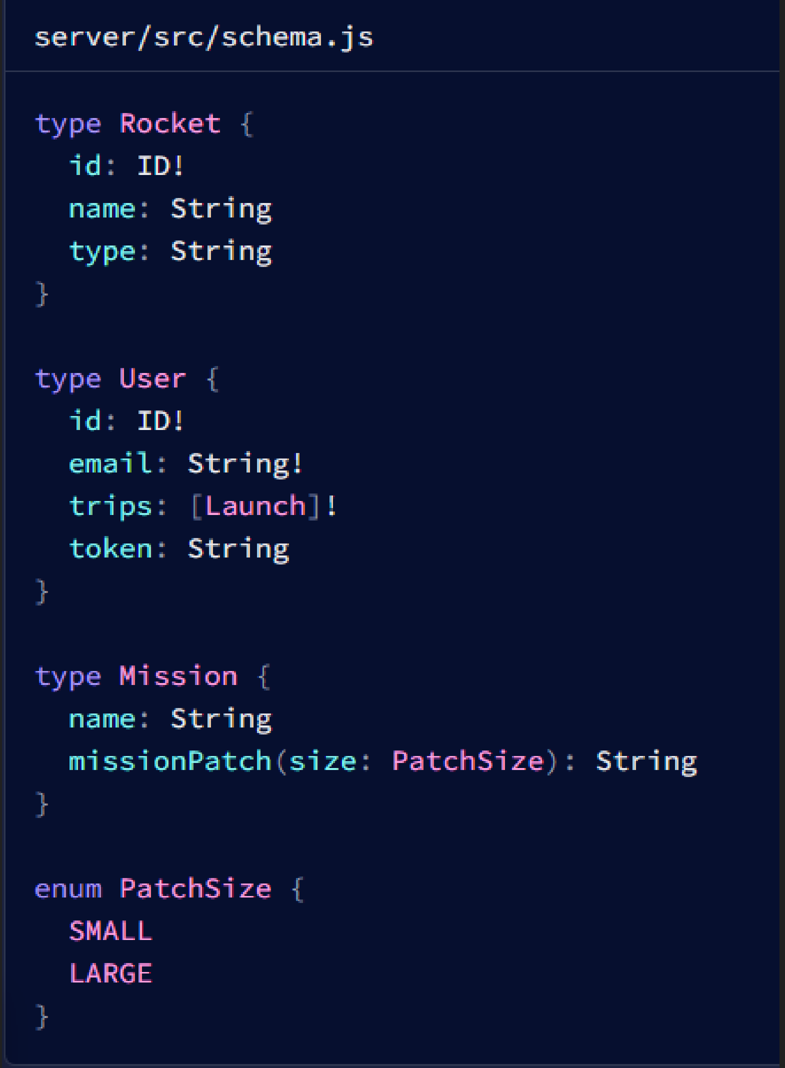


```
{
  "title": "System Design Interview Volume 2",
  "authors": [{
     "name": "Alex Xu"
  },
  {
     "name": "Sahn Lam"
  }]
  ...
}
```

```
type Book {
  id: ID
  title: String
  authors: [Author]
}

type Author {
  id: ID
  name: String
  books: [Book]
}
```

Defining Schemas

- Every graph uses a schema to define the types of data it includes
- Schemas are strongly typed
  - Primitive types
  - Objects
  - Enums
  - Interfaces
  - Unions
  - Input objects
  - ID
- This code must be on both client and server
  - Normally labeled schema.xx

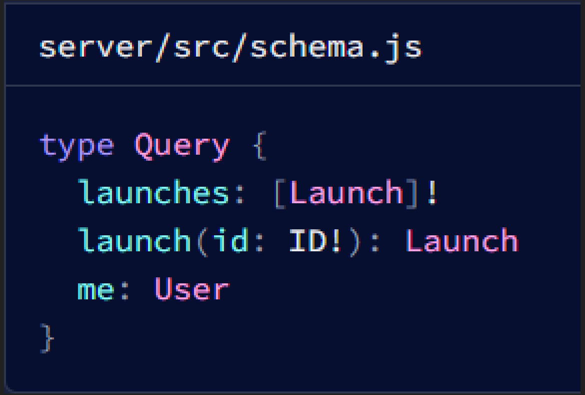


server/src/schema.js

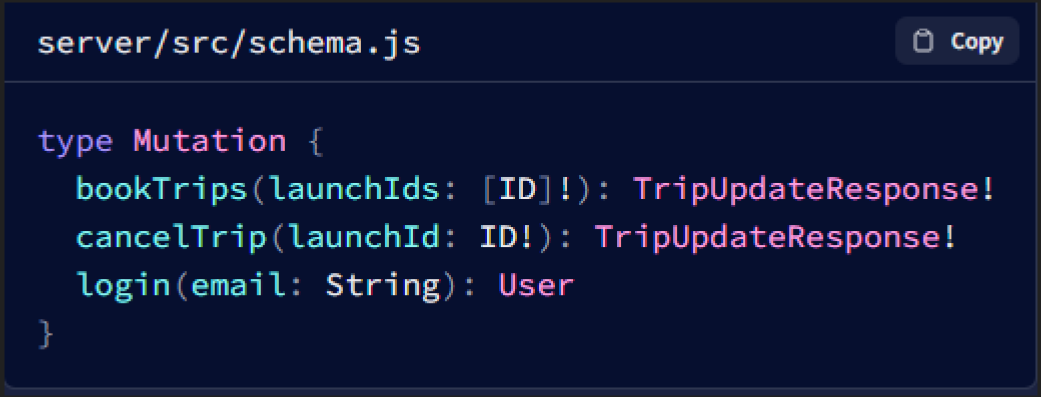
```
type Rocket {
  id: ID!
  name: String
  type: String
}

type User {
  id: ID!
  email: String!
  trips: [Launch]!
  token: String
}

type Mission {
  name: String
  missionPatch(size: PatchSize): String
}

enum PatchSize {
  SMALL
  LARGE
}
```

Operations: Queries and Mutation

- Mutation - GraphQL way of applying modification of resources
  - Equivalent of a POST request
- Queries- GraphQL way for client to receive notifications on data modifications
  - Equivalent of a GET request

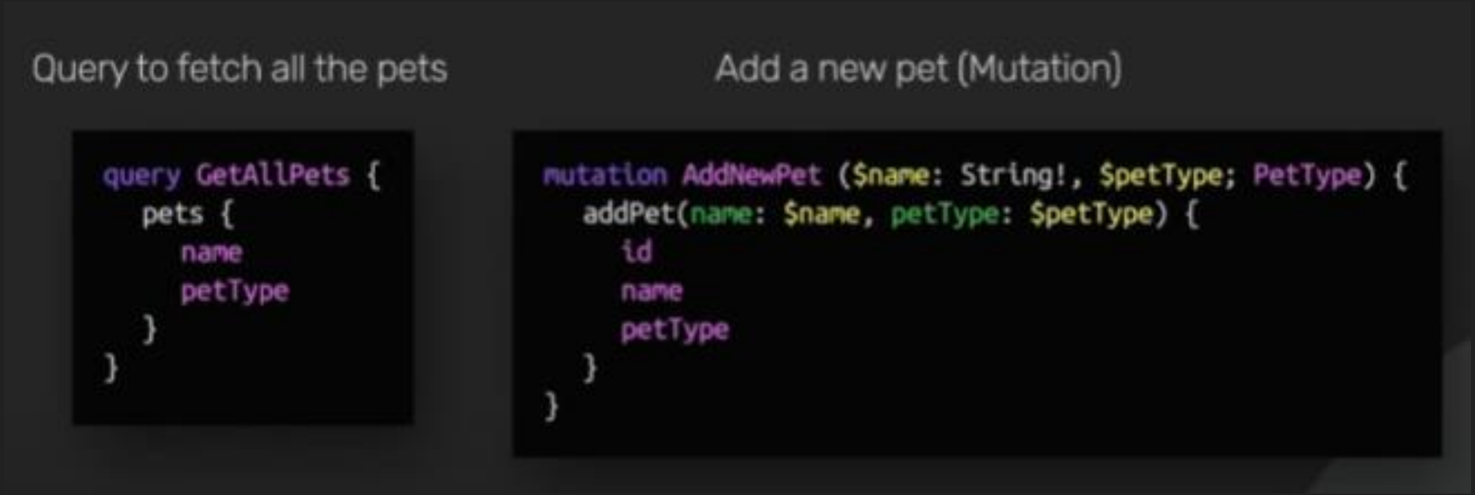


```
server/src/schema.js

type Query {
  launches: [Launch]!
  launch(id: ID!): Launch
  me: User
}
```



```
server/src/schema.js                                              Copy

type Mutation {
  bookTrips(launchIds: [ID]!): TripUpdateResponse!
  cancelTrip(launchId: ID!): TripUpdateResponse!
  login(email: String): User
}
```

Queries and Mutation


Query to fetch all the pets

```
query GetAllPets {
  pets {
    name
    petType
  }
}
```

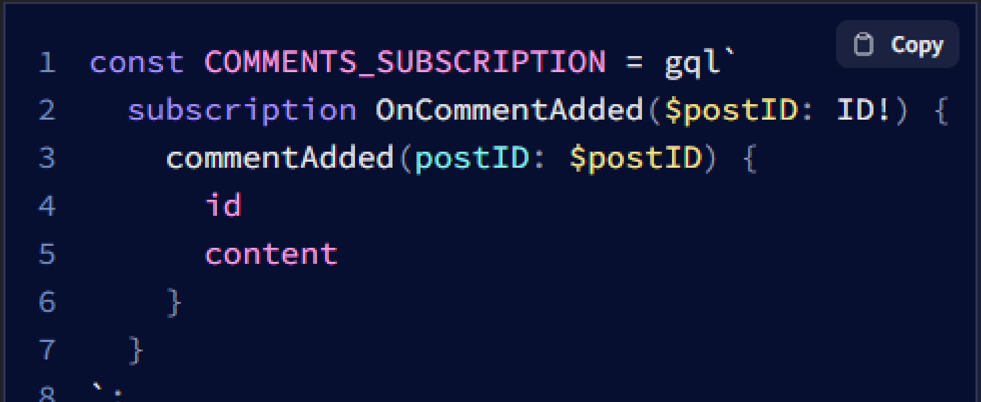
Add a new pet (Mutation)

```
mutation AddNewPet ($name: String!, $petType; PetType) {
  addPet(name: $name, petType: $petType) {
    id
    name
    petType
  }
}
```

Subscriptions

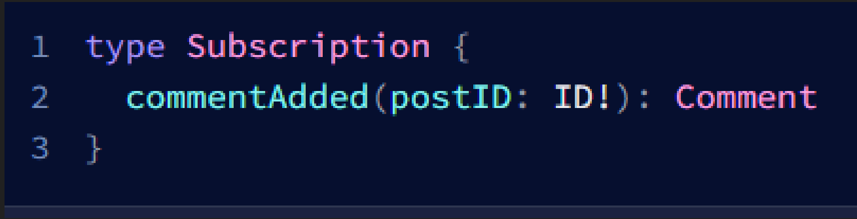
- Subscriptions are long-lasting operations that can change their result over time.
- Will maintain an active connection to GraphQL server allowing server to push updates
  - Through WebSockets protocol in the graphql-ws library.
- Used for Small, incremental changes to large objects
- Low-latency, real-time updates. Ex. chat app receiving new messages

Client side



```
const COMMENTS_SUBSCRIPTION = gql`
  subscription OnCommentAdded($postID: ID!) {
    commentAdded(postID: $postID) {
      id
      content
    }
  }
`;
```

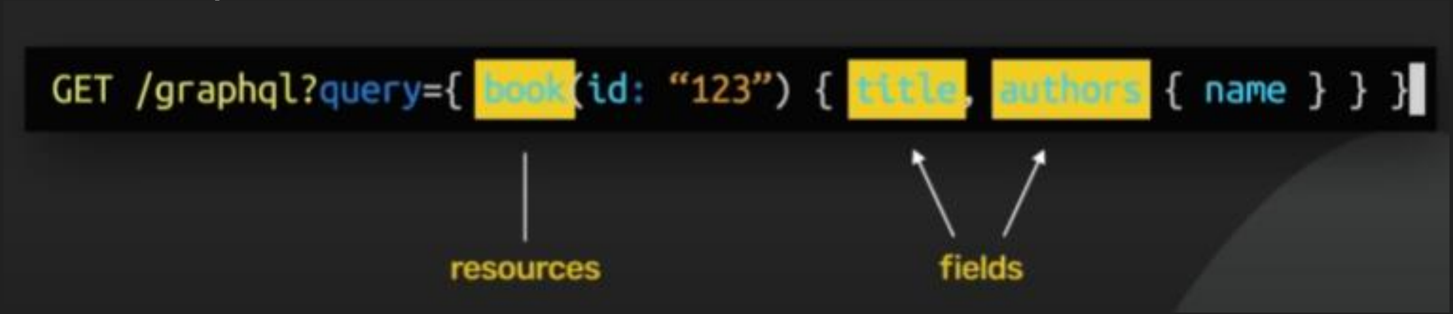
Server Ssde



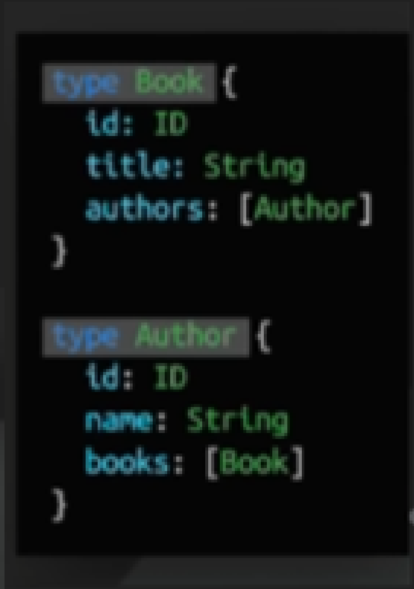
```
type Subscription {
  commentAdded(postID: ID!): Comment
}
```

GraphQL requests and response

GET request format



```
GET /graphql?query={ book(id: “123”) { title, authors { name } } }
```



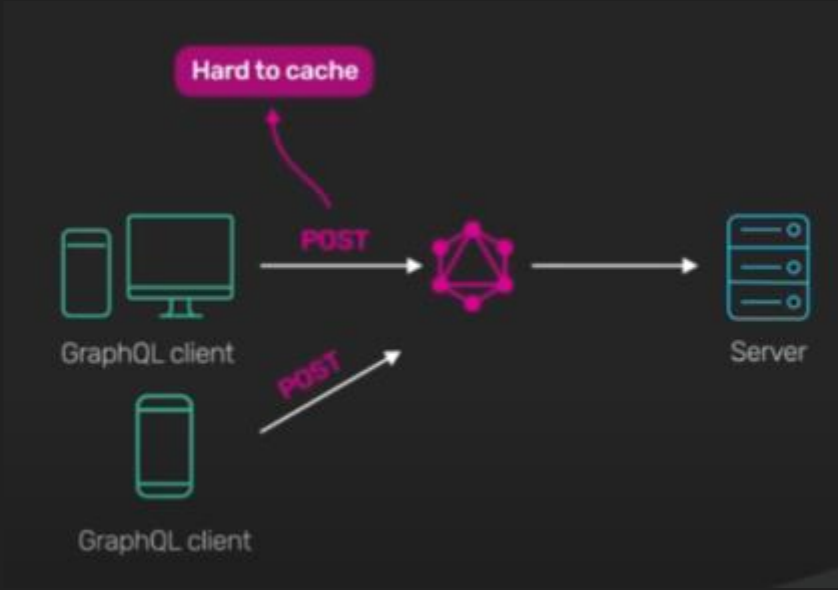
```
type Book {
  id: ID
  title: String
  authors: [Author]
}

type Author {
  id: ID
  name: String
  books: [Book]
}
```

The response is a nested JSON object just like rest

GraphQL drawbacks

- GraphQL requires heavier tooling support, both on the client and server sides.
    - No special libraries need for REST APIs
- Required files for all parties
    - Schema.graphql
    - Codegen.yaml
    - operations.graphql
- Need other tools Like Apollo to implement the specification
    - Apollo framework and tools allows us to build, validate, observe the graph
- Few companies have public graphQL services
    - Ex. Yelp, Github, Spacex
    - All have REST APIs as well

GraphQL drawbacks



- This requires a sizable upfront investment.
  - In development time and resources
  - Adds complexity for CRUD operations
- Caching becomes more difficult
  - REST leverages browser, CDNs, proxies, and web services
  - Caching becomes highly nuanced and not trivial

GraphQL drawbacks

- Its has a sharp learning curve due to niche operations and schema language
  - Schema definition language (SDL)
  - Does Not follow KISS

So REST vs GraphQL like everything SWE → TRADEOFFS 😃

Is there a better way?

maybe?

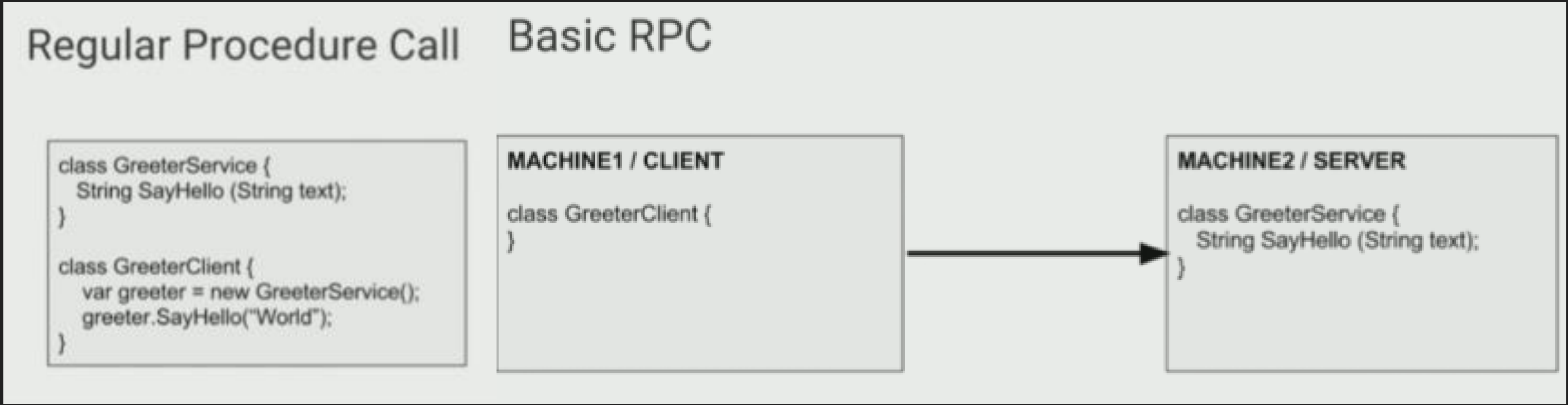
Why I chose this topic?

RPC (Remote Procedure Call)

- RPCs enables one machine to perform code one another AS IF its a local call
- Great for connecting multiple backend services together

Same machine

Multiple machines



```
class GreeterService {
  String SayHello (String text);
}

class GreeterClient {
   var greeter = new GreeterService();
   greeter.SayHello("World");
}
```

```
class GreeterClient {
}
```

```
class GreeterService {
  String SayHello (String text);
}
```

Why not REST

- For simple servies, HTTP REST is enough
  - HTTP verbs are rich
  - Details are well understood

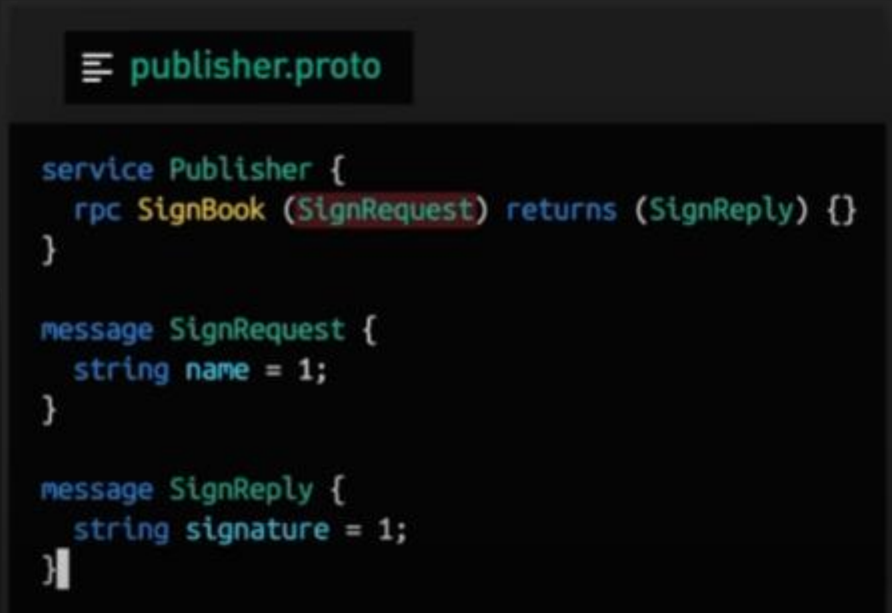
- For more complex services, RPCs provides more flexibility
  - Domain specific: bank transfers
  - More strongly typed experience

gRPC

- Open source RPC framework made by Google in 2016
- Language agnostic
- Easy to use
- Really fast performance with ProtoBufs over HTTP/2
- Enables developers to build microservices based apps
- Can use SSL/TLS tokens for Authentication

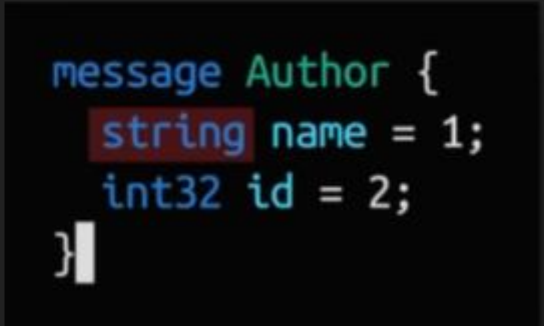
Protocol Buffers

- Language and platform agnostic mechanism for encoding structured data over the wire.
- support strongly-typed schema definitions.
    - Defined in a .proto file
- Services defined in a .proto file by specifying RPC method parameters and return types.

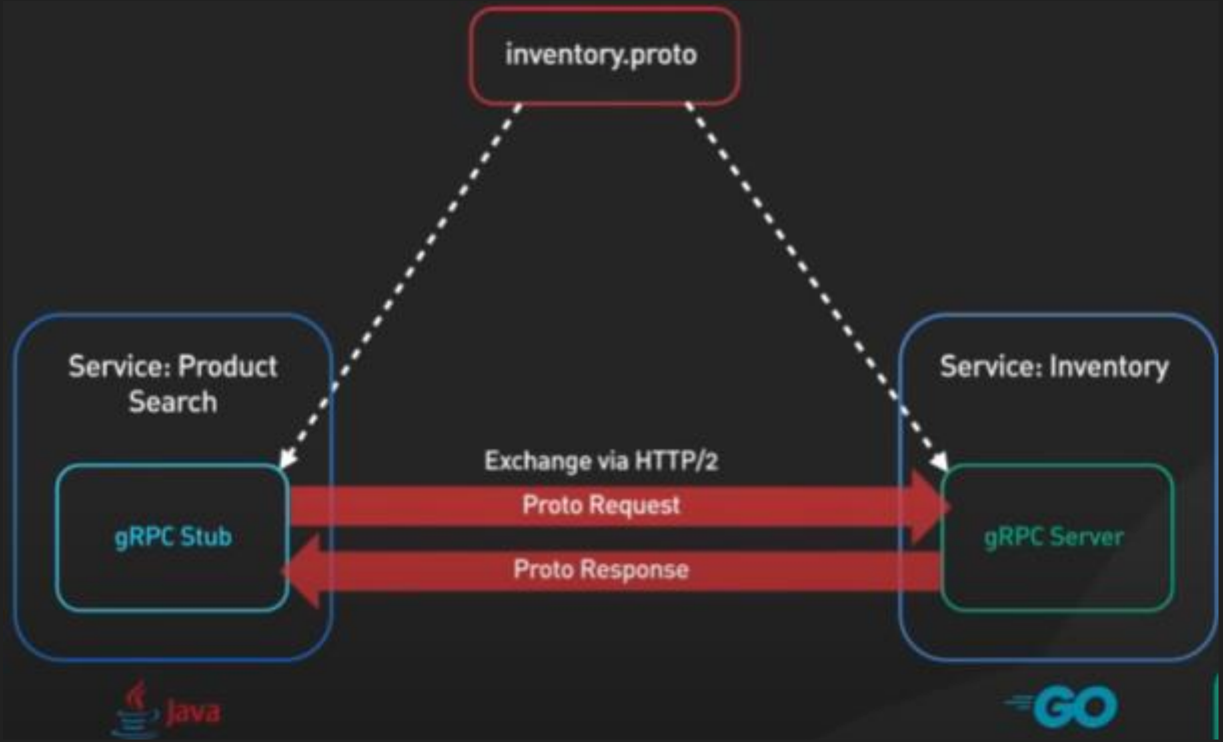


publisher.proto

```
service Publisher {
  rpc SignBook (SignRequest) returns (SignReply) {}
}

message SignRequest {
  string name = 1;
}

message SignReply {
  string signature = 1;
}
```

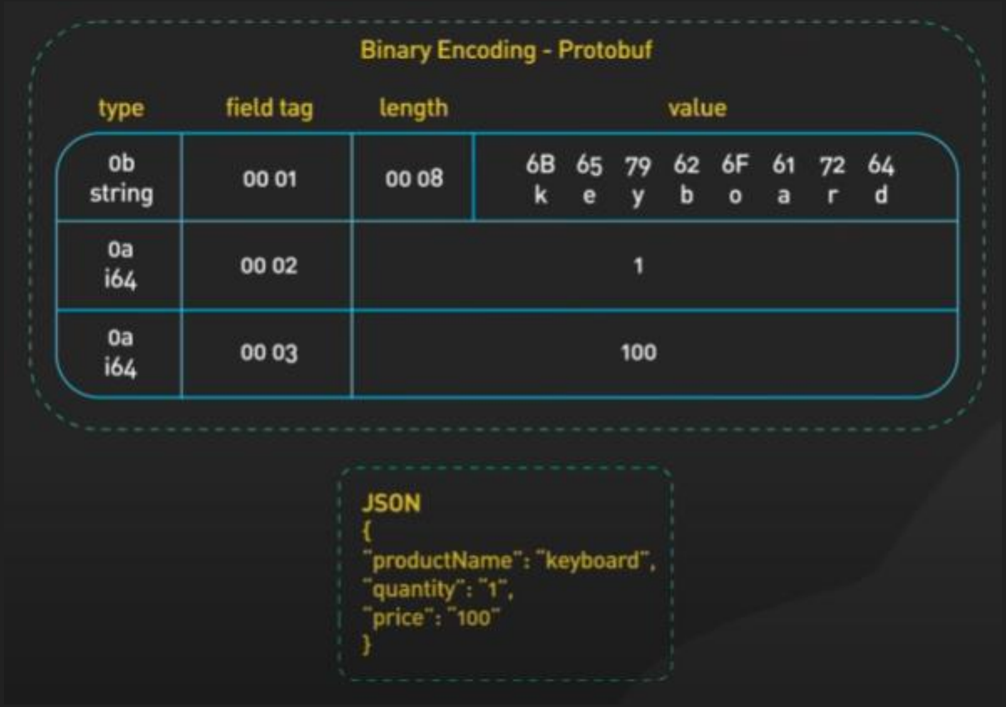


```
message Author {
  string name = 1;
  int32 id = 2;
}
```


inventory.proto
Service: Product Search
gRPC Stub
Exchange via HTTP/2
Proto Request
Proto Response
Service: Inventory
gRPC Server
Java
GO

Performance

- Protocol Buffers is a very efficient binary encoding format.
    - More fast JSON
- gRPC is built on top of HTTP/2
    - Multiplexing
    - Steam prioritization
    - Binary protocols ..
- Allows multiple RPC calls over a TCP connection



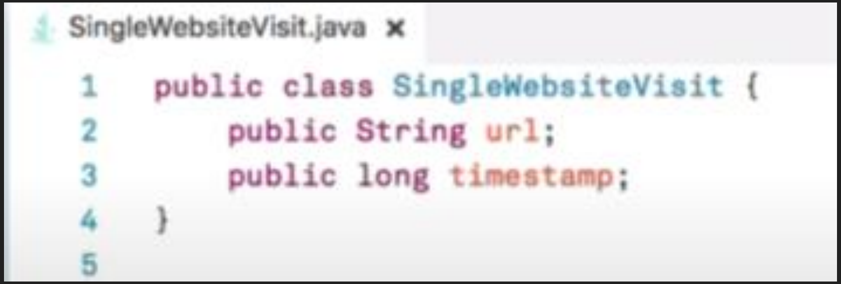
Binary Encoding - Protobuf

type	field tag	length	value
0b string	00 01	00 08	6B 65 79 62 6F 61 72 64 k e y b o a r d
0a i64	00 02	1	
0a i64	00 03	100	

```
JSON
{
"productName": "keyboard",
"quantity": "1",
"price": "100"
}
```

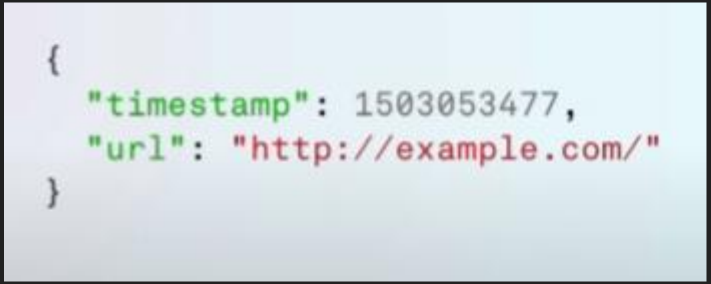
JSON issues

- we can easily parse the JSON to an internal data structure using the built-in JSON library



SingleWebsiteVisit.java

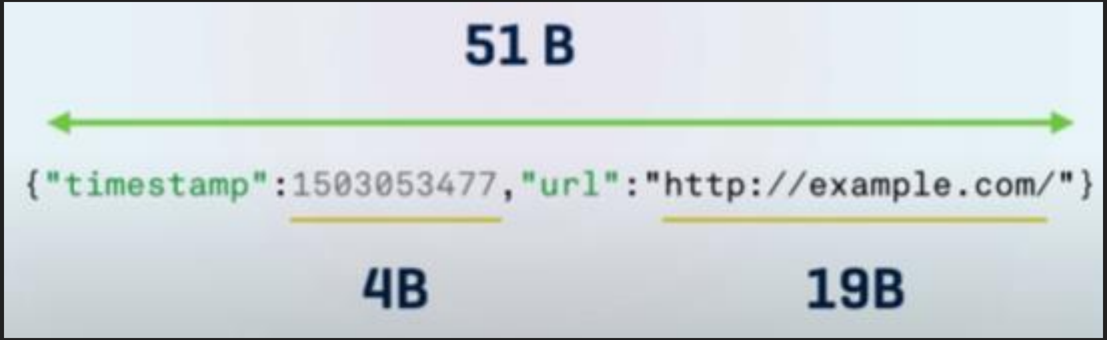
```java
public class SingleWebsiteVisit {
    public String url;
    public long timestamp;
}
```



```json
{
  "timestamp": 1503053477,
  "url": "http://example.com/"
}
```

- Self-contained and human readable
- BUT cost of serializing/deserializing is expensive
- Has unclear types

JSON is massive



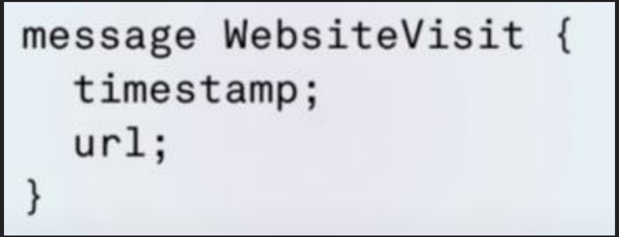
```
{"timestamp":1503053477,"url":"http://example.com/"}
```

- URL is 19 bytes (19 characters)
- Timestamp as a 32bit is 4 bytes
- Total = 23 bytes
- Nearly half is overhead!!

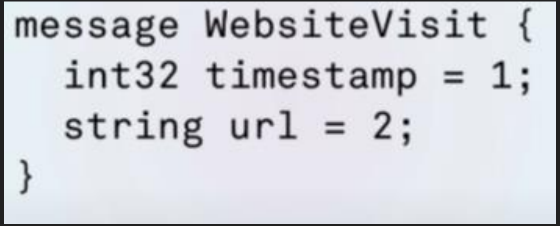
Protobuf request

- Lets save space and power by declaring this schema beforehand.
  - Defined in our .proto file
- Define codes for each field
  - to mark where one field ends and another one starts.

- use this schema to store the types of the data we store.

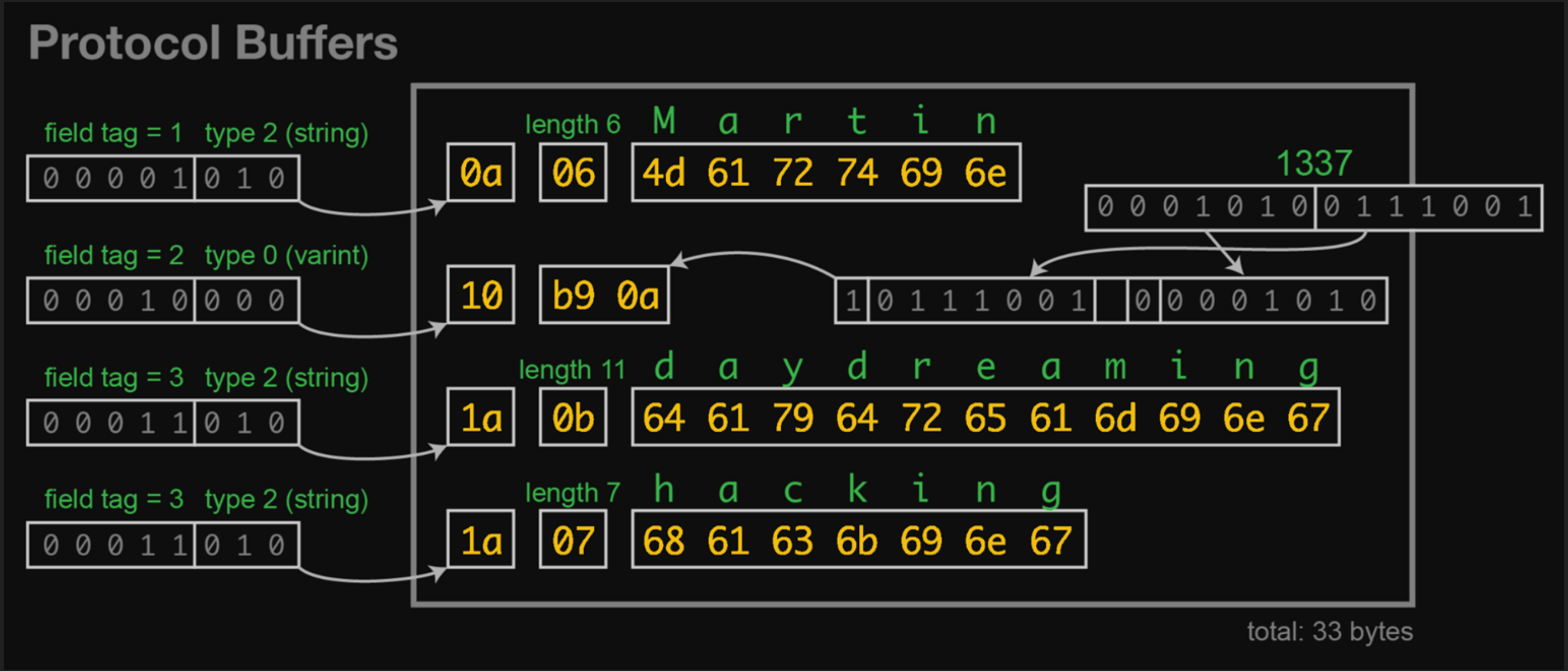


```
message WebsiteVisit {
  timestamp;
  url;
}
```



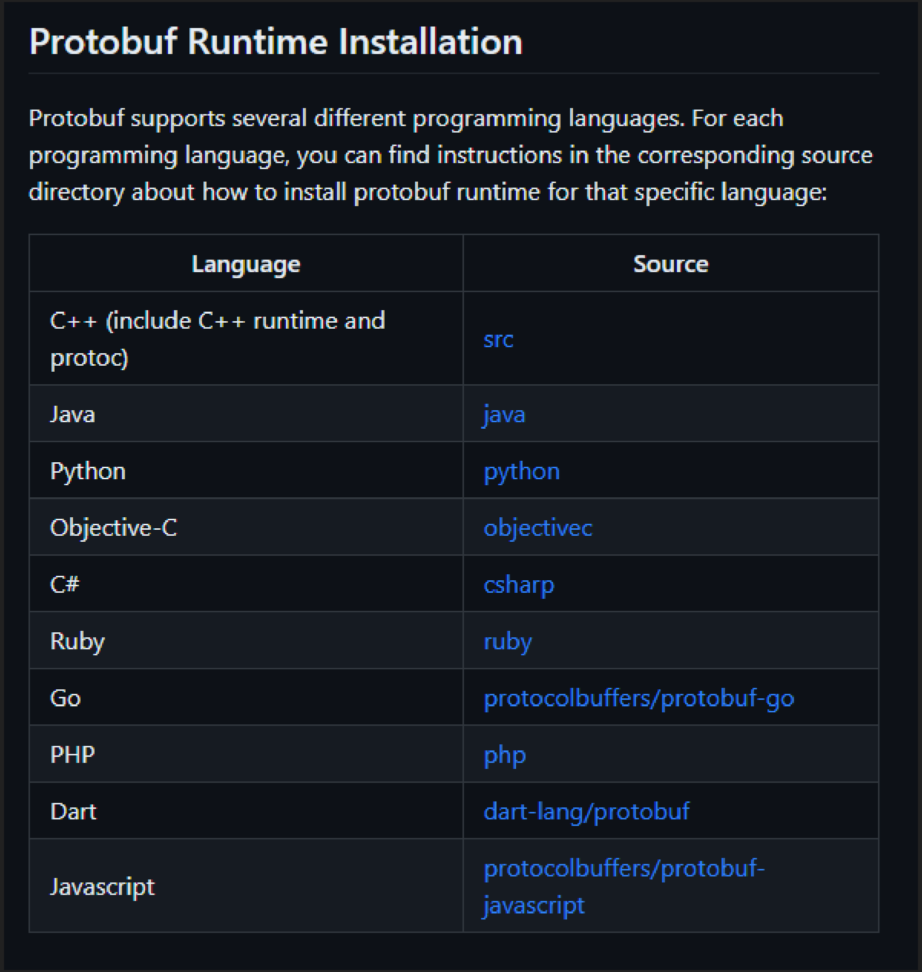
```
message WebsiteVisit {
  int32 timestamp = 1;
  string url = 2;
}
```

Sample Request



Serialization

- But JSON is so simple to read and parse -> its a string
- Binary is not portable on its own
- Google made open-source generators

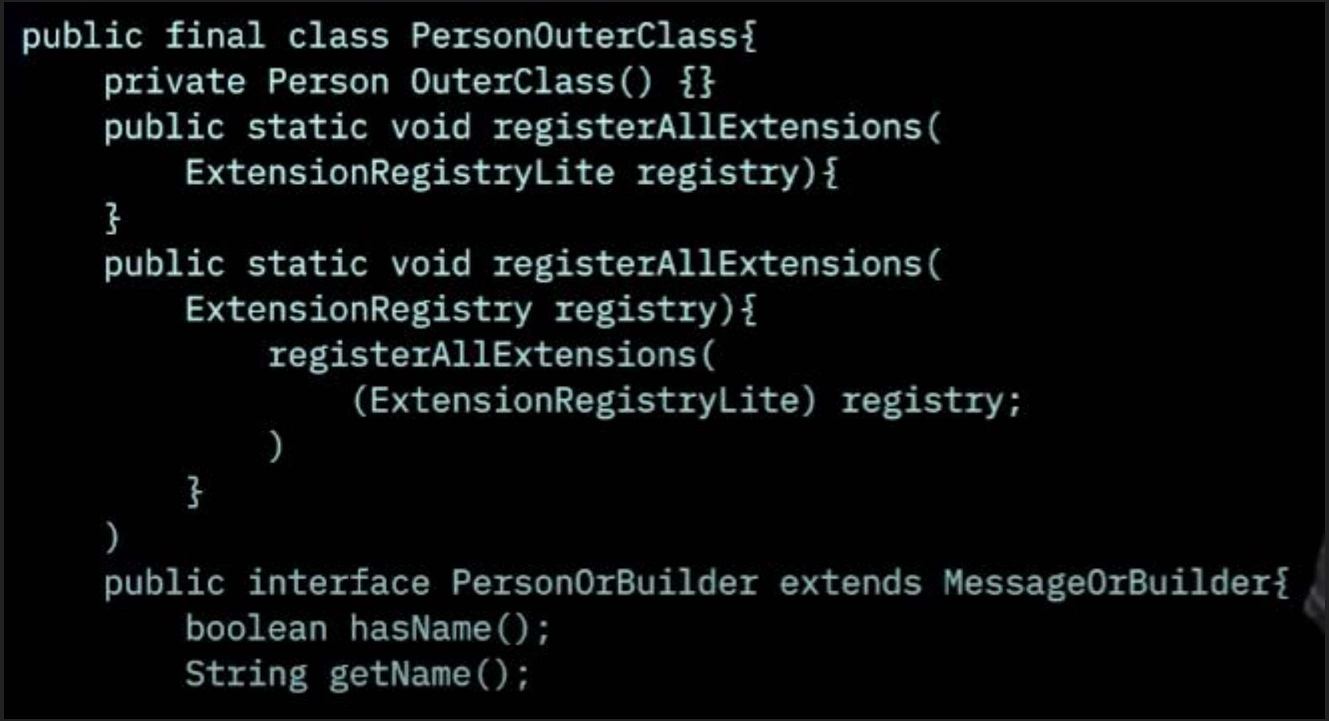


Protobuf Runtime Installation

Protobuf supports several different programming languages. For each programming language, you can find instructions in the corresponding source directory about how to install protobuf runtime for that specific language:

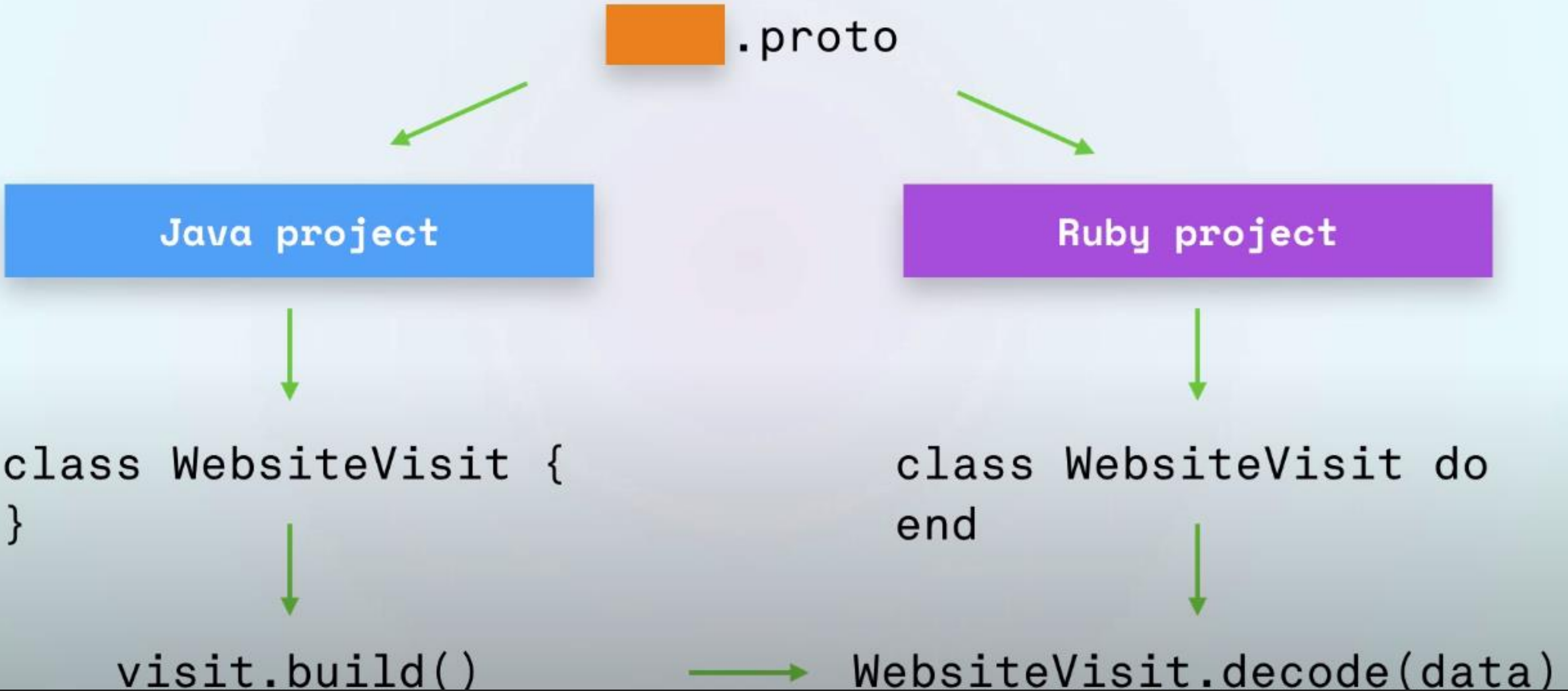
Language	Source
C++ (include C++ runtime and protoc)	src
Java	java
Python	python
Objective-C	objectivec
C#	csharp
Ruby	ruby
Go	protocolbuffers/protobuf-go
PHP	php
Dart	dart-lang/protobuf
Javascript	protocolbuffers/protobuf-javascript

Response



```
public final class PersonOuterClass{
    private Person OuterClass() {}
    public static void registerAllExtensions(
        ExtensionRegistryLite registry){
    }
    public static void registerAllExtensions(
        ExtensionRegistry registry){
            registerAllExtensions(
                (ExtensionRegistryLite) registry;
            )
        }
    )
    public interface PersonOrBuilder extends MessageOrBuilder{
        boolean hasName();
        String getName();
```

Protobuf



```
class WebsiteVisit {
}
```

```
class WebsiteVisit do
end
```

```
visit.build()
```

```
WebsiteVisit.decode(data)
```

JSON

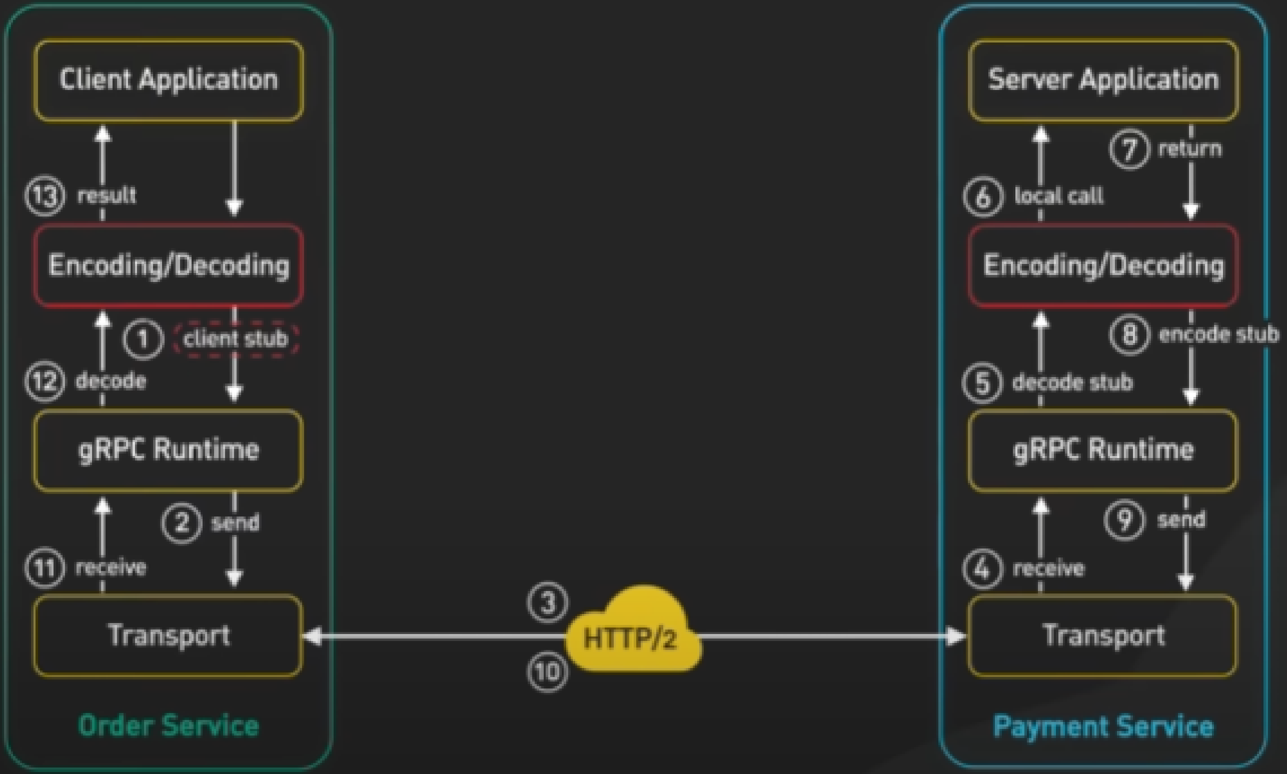
Good for :

- Small volume
- Indeed to inspect
- Messages are varied
- Sending to browsers

Protobuf

Good for:

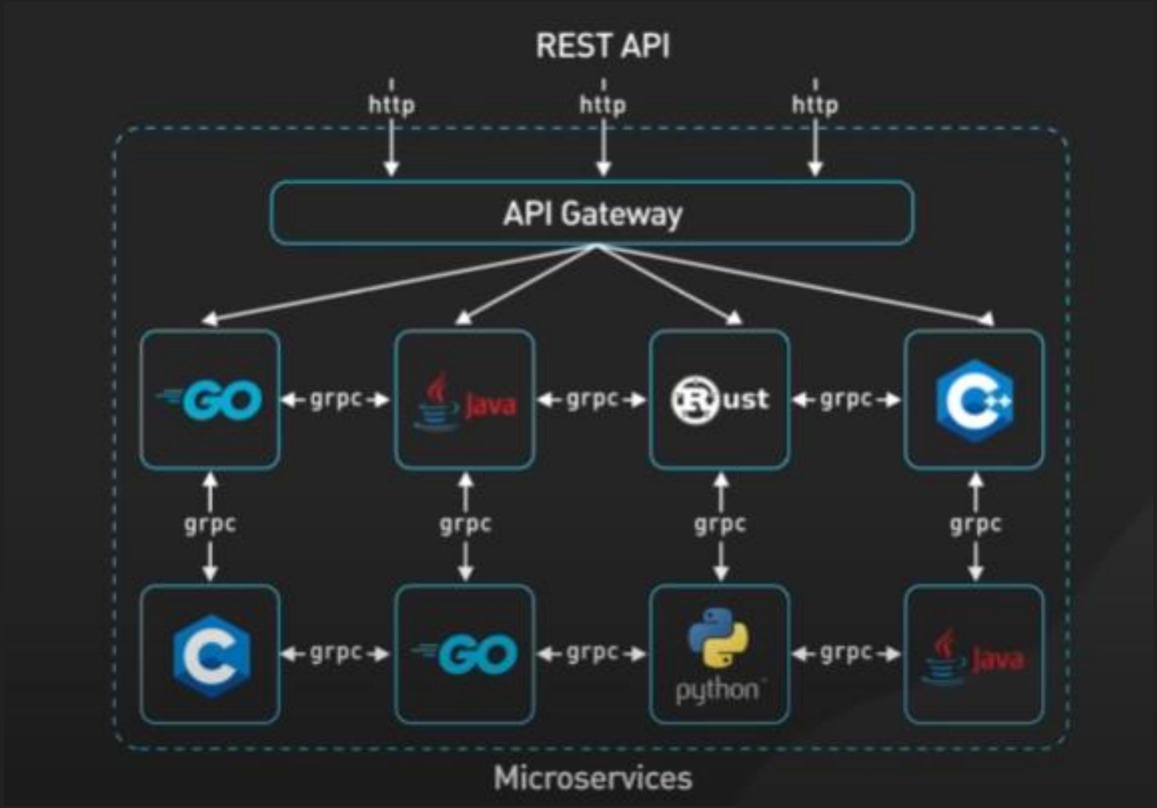
- Higher volumes
- Similar messages
- Performance matters
- Sending to services


Client Application
13 result
Encoding/Decoding
1 client stub
12 decode
gRPC Runtime
2 send
11 receive
Transport
Order Service
3
HTTP/2
10
Server Application
7 return
6 local call
Encoding/Decoding
8 encode stub
5 decode stub
gRPC Runtime
9 send
4 receive
Transport
Payment Service

Web clients??

- gRPC relies on lower-level access to HTTP/2 primitives.
- No browsers currently provide that level of control
- Solution: Use a proxy → gRPC-Web
    - Not fully compatible with gRPC
    - Usage is low

Best for Microservices



Which one??

- Each have pros/cons
- Depends on business logic or goals