gRPC - GraphQL Server

# Introduction

It exposes a GraphQL query to interact with a gRPC server. This server is beneficial for services, which cannot support RPC may be because of design constraints. The gRPC services are defined using Protocol buffers; In this assignment *book.proto* defines the messages and services for the gRPC calls.

# Architecture Design

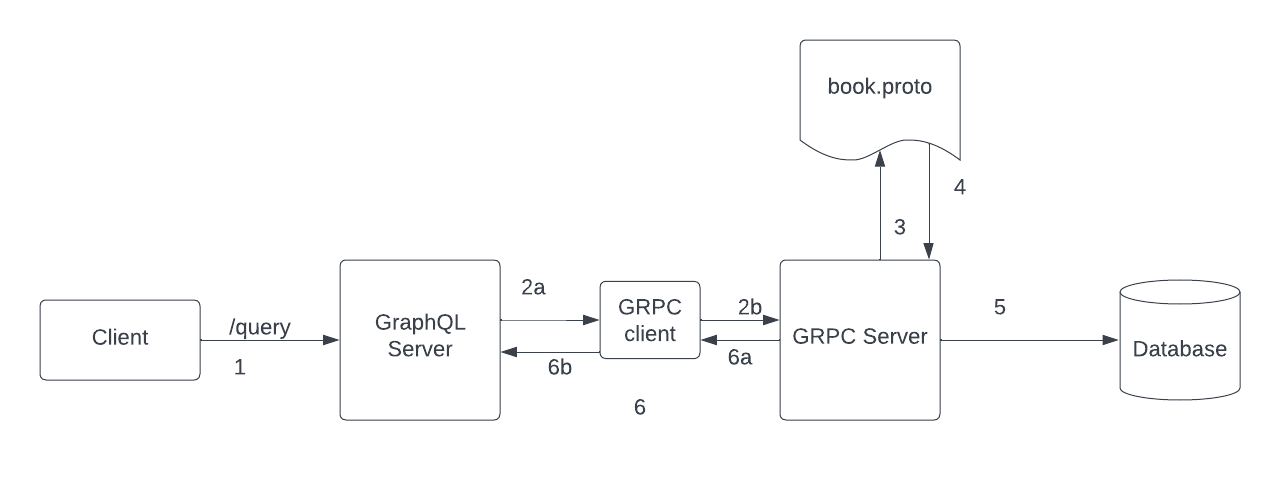


Fig 1.1 - gRPC-GraphQL workflow

1. The Client queries the GraphQL server to perform mutation or query

query {

books(where: null) {

id

author

title

summary

isbn

}

}

1. The GraphQL Server uses the gRPC client to interact with the gRPC server. The GQL servers calls the appropriate client handlers to and client calls the server.
2. Book.proto defines the messages and services required to for the gRPC server
3. The server queries the mongoDB
4. The message response is returned to the gRPC client
5. The GraphQL returns the response to the client (playground, postman or any other graphql clients)

## Schemas

### gRPC Messages

message BookResponse {

string id = 1;

string author = 2;

string isbn = 3;

string title = 4;

string summary = 5;

}

message Books {

repeated BookResponse book = 1;

}

message BookRequest {

string author = 2;

string isbn = 3;

string title = 4;

string summary = 5;

}

message UpdateBookRequest {

string id = 1;

BookRequest bookRequest = 2;

}

service BookHandlers {

rpc CreateBook(BookRequest) returns (BookResponse){};

rpc GetBooks(google.protobuf.StringValue) returns (Books){};

rpc UpdateBook(UpdateBookRequest) returns (BookResponse){};

rpc DeleteBook(google.protobuf.StringValue) returns (google.protobuf.BoolValue){};

}

### 

### 

### 

### 

### GraphQL Schema

type Book {

id: ID!

author: String!

isbn: String!

title: String!

summary: String

}

input BookInput {

author: String!

isbn: String!

title: String!

summary: String

}

input BookFilters {

id: ID

}

type Query {

books(where: BookFilters):[Book]

}

type Mutation {

createBook(input: BookInput!): Book!

updateBook(id: ID!, input: BookInput!): Book!

deleteBook(id: ID!): Boolean!

}

## Challenges

1. The had the idea what is gRPC and It uses in microservices architecture but I haven’t implemented gRPC server and never worked with protocol buffers
2. Another challenge I faced was how to communicate with gRPC server using GraphQL. I have worked with GraphQL server and client in my previous origanization (SmithRX).

## Overcame challenges

1. I divided the projects into three distinct parts: grpc-server, gql-server, and integrating gql-grpc server.
2. Initially, my approach involved a deep dive into the [grpc-go](https://github.com/grpc/grpc-go/tree/master) repository. I diligently studied how to define messages and services within the gRPC architecture, meticulously following the examples section while exploring various usage scenarios. This thorough examination allowed me to successfully create the book.proto file, subsequently generating the essential book.pb.go and book\_grpc.pb.go files. Furthermore, I meticulously implemented the required services in the handlers/book.go file, establishing a seamless connection with MongoDB and executing vital CRUD operations.
3. Then, I generated the necessary schemas, creating the imperative models\_gen.go and interface\_gen.go files using book.graphql and the gqlgen library—a tool we had previously utilized in our organization. Given my lack of experience in building a gql-server from scratch, I referred back to the gqlgen library documentation whenever clarification was needed.
4. The complexity arose when I had to connect both components, with an absence of comprehensive documentation for guidance. In my quest for a solution, I explored the [grpc-gql-gateway](https://github.com/ysugimoto/grpc-graphql-gateway) library, which utilized [grpc-gateway](https://github.com/grpc-ecosystem/grpc-gateway) to craft a reverse proxy linking gRPC with GraphQL. Despite my initial attempts at implementing this solution, I found the approach lacking and unsuitable for our needs.
5. The moment occurred when I observed Postman using the imported book.proto file to facilitate gRPC calls. This observation prompted me to revisit the [grpc-go](https://github.com/grpc/grpc-go/tree/master) examples, where I made a crucial discovery: the gRPC client defined in the book\_grpc.pb.go file could serve as the bridge to communicate with the server. Postman was essentially leveraging the same method.
6. Consequently, in my gql-server resolvers file, I seamlessly integrated the gRPC client, ensuring effective communication with the gRPC server. This strategic decision streamlined the entire process, bringing both parts of the project into harmonious synchronization.