gRPC

09 May 2025 23:38

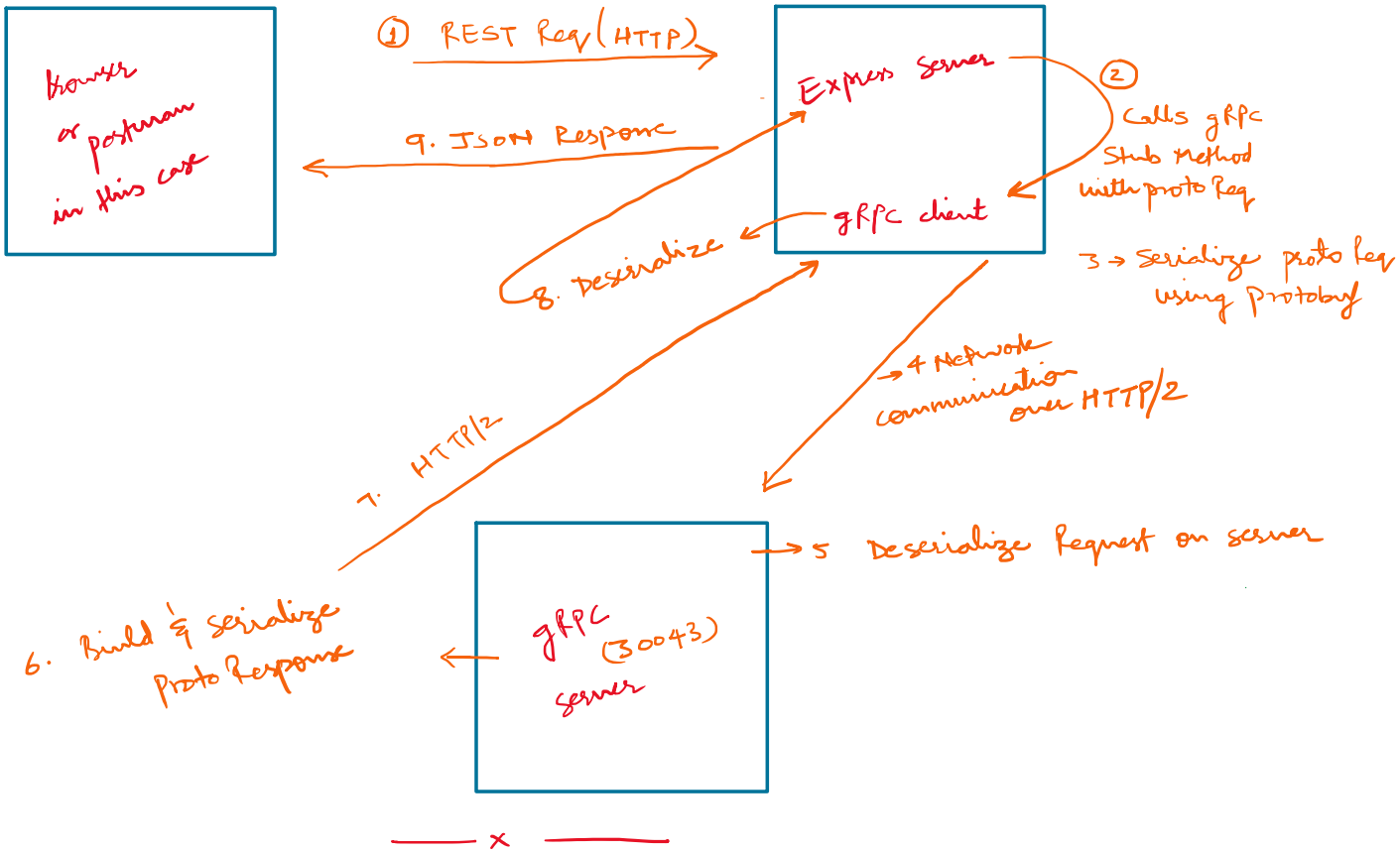
gRPC (Google Remote procedure call), is a modern high performance framework that allows a client to call functions directly on a server located on another machine, just like calling a local function.

RPC → Remote procedure call

It's a way for one computer (or program) to call a function that runs on another computer, just like calling a local function.

— x —

→ Flow of our small gRPC project



— x —

→ .proto file ?

```
syntax = "proto3";
service CustomerService{
    rpc GetAll (Empty) returns (CustomerList){}
    rpc Get(CustomerRequestId) returns (Customer){}
    rpc Insert(Customer) returns (Customer){}
    rpc Update(Customer) returns (Customer){}
    rpc Delete(CustomerRequestId) returns (Empty){}
}
message Empty{}

message CustomerRequestId{
    string id = 1;
}
message Customer{
    string id = 1;
    string name = 2;
    string email = 3;
}
```

customer related methods are declared inside this.

Methods

Output

Input

Data str. that will be sent or received

→ .proto file (proto Buffer)

Significance ? (This tells gRPC)

→ what method exists.

→ what req and response messages types each method uses.

→ How the server & client should communicate

— x —

```
syntax = "proto3";
service CustomerService{
    rpc GetAll (Empty) returns (CustomerList){}
    rpc Get(CustomerRequestId) returns (Customer){}
    rpc Insert(Customer) returns (Customer){}
    rpc Update(Customer) returns (Customer){}
    rpc Delete(CustomerRequestId) returns (Empty){}
}
message Empty{}

message CustomerRequestId{
    string id = 1;
}

message Customer{
    string id = 1;
    string name = 2;
    string email = 3;
    int32 age = 4;
}

message CustomerList{
    repeated Customer customers = 1;
}
```

→ .proto file (proto Buffer)

Significance ? (This tells gRPC)

→ what method exists.

→ what req and response messages types each method uses.

→ How the server & client should communicate

—x—

Now see how our client looks like (Express server in this case).

```
const PROTO_PATH = "./customers.proto";

import grpc from "@grpc/grpc-js";
import protoLoader from "@grpc/proto-loader";

const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
  keepCase: true,
  longs: String,
  enums: String,
  arrays: true,
});      You, yesterday • add gRPC server

const CustomerService =
  grpc.loadPackageDefinition(packageDefinition).CustomerService;

export const client = new CustomerService(
  "127.0.0.1:30043",
  grpc.credentials.createInsecure()
);
```

→ This is client.js inside client folder

→ It creates & exports a gRPC clients instance that knows how to communicate with gRPC server defined in our customerss.proto

—x—

```
import { client } from "./client.js";
import express from "express";
import bodyParser from "body-parser";

const app = express();
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

app.get("/", (req, res) => {···
});

app.post("/create", (req, res) => {
  let newCustomer = {
    id: req.body.id,
    name: req.body.name,
    email: req.body.email,
    phone: req.body.phone,
  };
  client.insert({ newCustomer }, (error, data) => {
    if (error) throw error;
    console.log("Customer created successfully", data);
    res.send({ message: "Customer created successfully" });
  });
});

app.post("/update", (req, res) => {···
});

app.post("/delete", (req, res) => {      You, yesterday • add gRPC server ···
});
app.get("/:id", (req, res) => {···
});
const PORT = 3000;

app.listen(PORT, () => {
  console.log("Server running at port %d", PORT);
});
```

→ This is our Index.js for client

→ Here every express Route is calling Mtds on the gRPC client → which internally calls our gRPC servers.

gRPC server

```
const PROTO_PATH = "./customers.proto";

import grpc from "@grpc/grpc-js";
import protoLoader from "@grpc/proto-loader";

const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
  keepCase: true,
  longs: String,
  enums: String,
  arrays: true,
});
const CustomerProto = grpc.loadPackageDefinition(packageDefinition);
const server = new grpc.Server();
const customers = [
  {
    id: "sdfshdfsd",
    name: "Abhilash Bijalwan",
    age: 26,
    email: "abhilashbijalwan999@gmail.com",
  },
  {
    id: "cvvbcbewb",
    name: "Akshay Saini",
    age: 22,
    address: "aksaini@gmial.com",
  },
];
server.addService(CustomerProto.CustomerService.service, {
  Qodo Gen: Options | Test this method
  getAll: (call, callback) => {
    callback(null, { customers });
  },
  Qodo Gen: Options | Test this method
  get: (call, callback) => {
    let customer = customers.find((n) => n.id == call.request.id);

    if (customer) {
      callback(null, customer);
    } else {
      callback({
        code: grpc.status.NOT_FOUND,
        details: "Not found",
      });
    }
  },
  Qodo Gen: Options | Test this method
  insert: (call, callback) => {···
  },
  Qodo Gen: Options | Test this method
  update: (call, callback) => {···
  },
  Qodo Gen: Options | Test this method
  remove: (call, callback) => {···
  },
});

server.bindAsync(
  "127.0.0.1:30043",
  grpc.ServerCredentials.createInsecure(),
  (err, port) => {
    if (err) {
      console.error(`Error starting gRPC server: ${err}`);
    } else {
      server.start();
      console.log(`gRPC server is listening on ${port}`);
    }
  }
);
```

Static in this case but, In real Inside here dB calls were happening

Requesting

Response

⇒ logic what gRPC server should do when any of the listed methods were called.

Advantages

→ High performance & efficient
  HTTP/2 + protobuf → fast binary serialization
  → smaller payloads than JSON/REST

→ strongly typed contracts

Disadvantage

→ Not Human Readable.

→ Learning Curve.

→ Limited browser support

→ Less flexible than REST

→ strongly typed contracts

→ Supports multiple languages

→ Built in code generation

→ Bidirectional Streaming

→ Better error handling

→ Interpolability

→ Limited browser support

→ Less flexible than REST