Adnan Ahmed

I am a Software Engineering Lead working at **tajawal** in Dubai. In my spare time, you can find me getting my hands dirty on **github** and this is the place where I blab about the technical stuff and sometimes about the stuff in general.

November 1, 2019

gRPC with Node.js and TypeScript

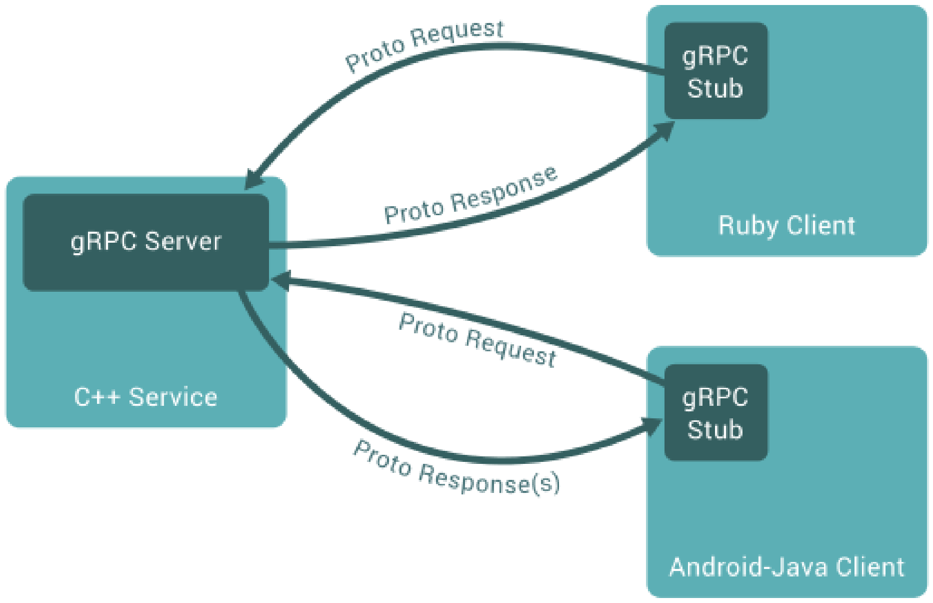
The quest for optimizing the communication over the network has been going on for ages. 1990s brought us TCP/IP protocols for networking and we saw the rise of technologies such as CORBA, DCOM, and Java RMI. With the evolution of web in 2000s, HTTP started to become the defacto for communication and people started to use XML over HTTP for the communication and we saw this combination giving boom to SOAP and WSDL which provided language agnostic communication between the systems. Moving forward, as the web evolved, JavaScript and JSON started to become popular and JSON started to *replace* XML as the preferred wire-transfer format and resulted in an unofficial standard called REST. It did not completely replace SOAP but most of the developer focus shifted towards REST while the enterprise applications and corporates which require strict adherence to standards and schema definitions, stayed with SOAP.

The recent hotness in the industry is gRPC which is a lightweight communication protocol from Google with a support for over a dozen languages. I recently got the chance to work with gRPC in Node.js – this article is a brief introduction to gRPC and how to use it with Node.js and TypeScript.

gRPC and Proto Files

gRPC is basically a high performance RPC framework created by Google. It runs over HTTP2 and it is the default protocol that is used instead of JSON on the network. By default gRPC uses protocol buffers as IDL (Interface Definition Language) to define the structure for the service interface and structure for the payload messages. Using the IDL we can generate type safe DTO's (Data transfer object) and client server implementations in multiple languages (like, Go, PHP, Ruby, Python, Objective-C, Node.js, Java, C, C#, Java). The types are converted to binary format when calling the remote procedures. So a server generated in C++ can communicate transparently with a client written in Java or Ruby.

The image given below gives an conceptual overview.



One of the biggest difference between gRPC and REST is the format of the payload. REST messages typically contain JSON. There's no defined interface for the request and response so it's safe to say that you can send anything in request and response. Where gRPC on the other hand uses defined interfaces for request and response that are defined using protocol buffers. This gives you a huge win over the REST API's and calling the services in gRPC is just like calling a local function. Also in terms of benchmark gRPC is much faster than REST.

So enough with the talk let's start with the actual work. We will create a greeter gRPC service that will accept your name and in reply will greet you like `Hi, Adnan`

Setting up a Project

Let's begin by creating and empty project

```
mkdir ts-grpc
```

Now go inside the directory and create the directory structure similar to the one given below

```
.
├── dist/                       # Compiled files
├── src/                        # Source files
|   ├── handlers/               # gRPC service handlers
|   |   └── greeter.ts          # Greeter service definitions
|   ├── proto/                  # Proto files
|   |   ├── greeter/            # Greeter gRPC service
|   |   |   └── greeter.proto
|   |   ├── index.ts            # Registers all the proto typescript defi
|   └── server.ts               # Bootstrap server, add middleware (logs,
├── scripts/                    # Generation tools
|   └── protoc.sh               # Script to generate protoc typescript de
└── ...
```

structure.sh hosted with ❤ by **GitHub** **view raw**

Now lets install the dependencies

```
npm init -y
npm install grpc google-protobuf dotenv
npm install typescript @types/node @types/google-protobuf @types/doter
```

Here's the explanation of the packages that we have installed

- grpc to use gRPC with Node.js
- google-protobuf to use Protocol Buffers (.proto) with javascript
- dotenv to load environment variables from .env
- TypeScript and typedefinitions to help in development

Initialize typescript so that later on we can compile the project

```
npx tsc --init
```

After initializing typescript add the below content to your `tsconfig.json` file

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "module": "commonjs",
    "noImplicitAny": true,
    "allowJs": true,
    "esModuleInterop": true,
    "target": "es6",
    "sourceMap": true
  },
  "include": ["./src/**/*"],
  "exclude": ["node_modules"]
}
```

tsconfig.json hosted with ❤ by **GitHub** **view raw**

Now open the `greeter.proto` file and add the below content in it

```
syntax = "proto3";

package greeter;

// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloResponse {
  string message = 1;
}
```

greeter.proto hosted with ❤ by **GitHub** **view raw**

It creates one service `SayHello` that accepts requests of type `HelloRequest` with one field `name` and gives response of type `HelloResponse`. You can read more about the `proto` file syntax here https://developers.google.com/protocol-buffers/docs/proto

Generating TypeScript definitions

Now let's generate the typescript definitions against the gRPC service by using the proto file. To generate the typescript we need some kind of compiler that will translate the `greeter.proto` to typescript definition. Here's the dependencies that we have to install in order to compile the `proto` file

```
npm install grpc-tools grpc_tools_node_protoc_ts --save-dev
```

Here we installed few more dev dependencies

- grpc-tools generate javascript files for the proto files
- grpc_tools_node_protoc_ts generate corresponding typescript d.ts codes according to js codes generated by grpc-tools

After installing the dependencies now we have to write a script that will loop over the all the available `proto` files in the `src/proto/**/*.proto` and compile them. For that open `protoc.sh` and replace with below

```
#!/usr/bin/env bash

BASEDIR=$(dirname "$0")
cd "${BASEDIR}"/../

PROTOC_GEN_TS_PATH="./node_modules/.bin/protoc-gen-ts"
GRPC_TOOLS_NODE_PROTOC_PLUGIN="./node_modules/.bin/grpc_tools_node_protoc_plugin"
GRPC_TOOLS_NODE_PROTOC="./node_modules/.bin/grpc_tools_node_protoc"

for f in ./src/proto/*; do

  # skip the non proto files
  if [ "$(basename "$f")" == "index.ts" ]; then
      continue
  fi

  # loop over all the available proto files and compile them into respective dir
  # JavaScript code generating
  ${GRPC_TOOLS_NODE_PROTOC} \
      --js_out=import_style=commonjs,binary:"${f}" \
      --grpc_out="${f}" \
      --plugin=protoc-gen-grpc="${GRPC_TOOLS_NODE_PROTOC_PLUGIN}" \
      -I "${f}" \
      "${f}"/*.proto

  ${GRPC_TOOLS_NODE_PROTOC} \
      --plugin=protoc-gen-ts="${PROTOC_GEN_TS_PATH}" \
      --ts_out="${f}" \
      -I "${f}" \
      "${f}"/*.proto

done
```

protoc.sh hosted with ❤ by **GitHub** **view raw**

Now we need to make this script executable so that we can actually use it. Run the below command to make it executable

```
sudo chmod +x ./scripts/protoc.sh
```

Now run the script and it will generate our typescript definition files and there respective javascript files in `src/proto/greeter`. Each time you will update your proto file you have run this script to generate new typescript definitions.

```
./scripts/protoc.sh
```

After the typescript definitions have been generated, we need to tell our typescript compiler to include these files during the compilation phase. In order to do that, open `src/proto/index.ts` and replace with below

```
import './greeter/greeter_pb';
import './greeter/greeter_grpc_pb';

export const protoIndex:any = ():void => {
};
```

index.ts hosted with ❤ by **GitHub** **view raw**

You need to generate the typescript definitions and update this file whenever you create new proto files.

Creating handlers

Let's write our greeter handler to define our `SayHello` service so open `src/handlers/greeter.ts` and replace it with below. It creates and handler that will handle all the requests against the greeter service. So later on if you will add new rpc services in your proto file `greeter.proto`, you have to define there respective definition here in this handler.

```
import * as grpc from 'grpc';

import { HelloRequest, HelloResponse } from './proto/greeter/greeter_pb';
import { GreeterService, IGreeterServer } from './proto/greeter/greeter_grpc_pb';

class GreeterHandler implements IGreeterServer {
    /**
     * Greet the user nicely
     * @param call
     * @param callback
     */
    sayHello = (call: grpc.ServerUnaryCall<HelloRequest>, callback: grpc.sendUnary
        const reply: HelloResponse = new HelloResponse();

        reply.setMessage(`Hello, ${ call.request.getName() }`);

        callback(null, reply);
    };
}

export default {
    service: GreeterService,                // Service interface
    handler: new GreeterHandler(),          // Service interface definitions
};
```

greeter.ts hosted with ❤ by **GitHub** **view raw**

See `sayHello` it has the implementation of our `SayHello` rpc service. Which is getting name from the request `HelloRequest` and provides response of type `HelloResponse`

Writing the Server

Now let's write a gRPC server. Open `src/server.ts` and replace with below

```
import 'dotenv/config';
import * as grpc from 'grpc';

import { protoIndex } from './proto';
import greeterHandler from './handlers/greeter';

protoIndex();

const port: string | number = process.env.PORT || 50051;

type StartServerType = () => void;
export const startServer: StartServerType = (): void => {
    // create a new gRPC server
    const server: grpc.Server = new grpc.Server();

    // register all the handler here...
    server.addService(greeterHandler.service, greeterHandler.handler);

    // define the host/port for server
    server.bindAsync(
        `0.0.0.0:${ port }`,
        grpc.ServerCredentials.createInsecure(),
        (err: Error, port: number) => {
            if (err != null) {
                return console.error(err);
            }
            console.log(`gRPC listening on ${ port }`);
        },
    );

    // start the gRPC server
    server.start();
};

startServer();
```

server.ts hosted with ❤ by **GitHub** **view raw**

As you can already guess from the code, we are just creating an instance of gRPC server, registering our greeter service handler and then just starting the server.

Testing our Implementation

Now let's test it. So to test it we have to update our `package.json scripts` section. So add following.

```
...
"scripts": {
    "build": "npx tsc --skipLibCheck",
    "start": "npx tsc --skipLibCheck && node ./dist/server.js"
}
...
```

Now open terminal and run the build command. This will create a `dist` folder and will compile typescript to generate javascript files.

```
npm run build
```

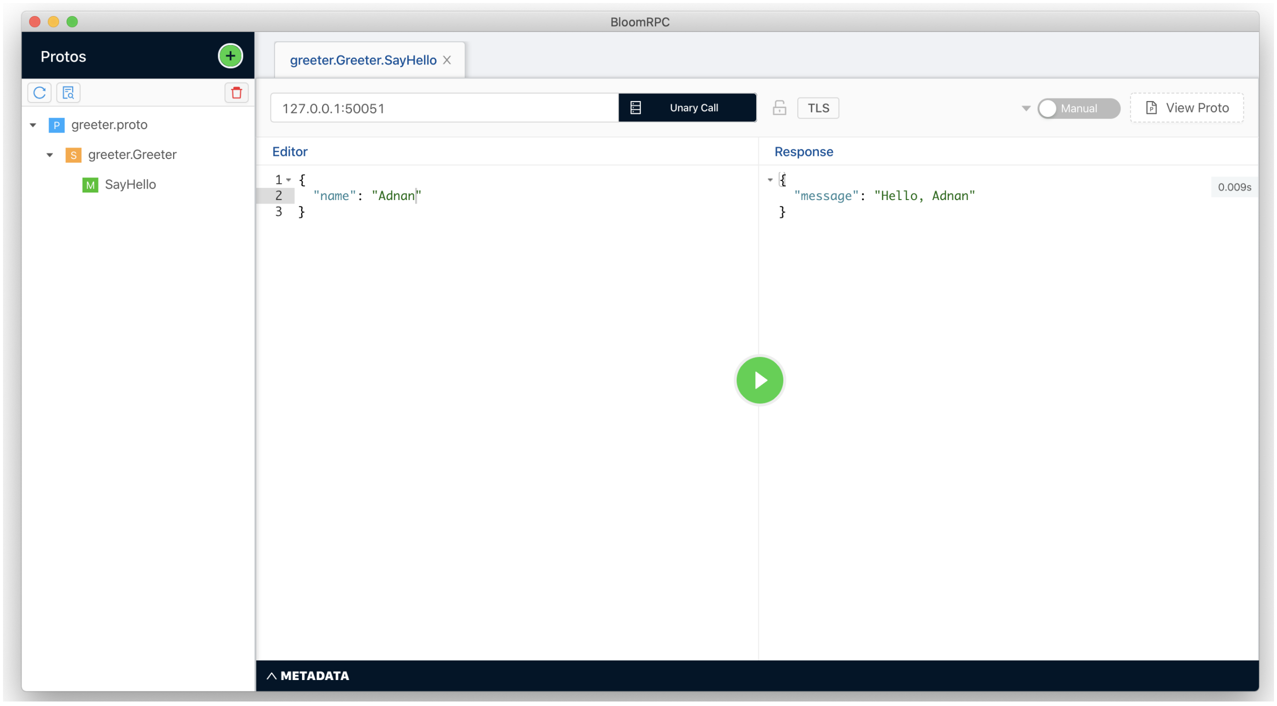
Once the build is finished, let's test our implementation by running the server. Run the command below to start the server

```
npm run start
```

If everything goes well, you should see the message `gRPC listening on 50051`

In order to test our server, I am going to use BloomRPC which is a GUI client to test RPC services.

Follow the installation guide, import the greeter.proto file, update the URL to be `127.0.0.1:50051` and click the PLAY icon and you will see the output similar to the one given below



127.0.0.1:50051

```
{
  "name": "Adnan"
}
```

```
{
  "message": "Hello, Adnan"
}
```

And that wraps it up. You can find the source code from the article here.

Feel free to leave your feedback or questions in the comments section below.

Liked this article? Follow me on twitter @idnan_se and tweet about it

What do you think?

7 Responses

Upvote Funny Love Surprised Angry

Sad

1 Comment **adnanahmed.info** 1 **Login**

Recommend Tweet Share **Sort by Best**

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Muhammad Hasham • 2 months ago

This is really great. A thorough introduction which includes implementation as well

Contact me via mail, linkedIn, twitter or github