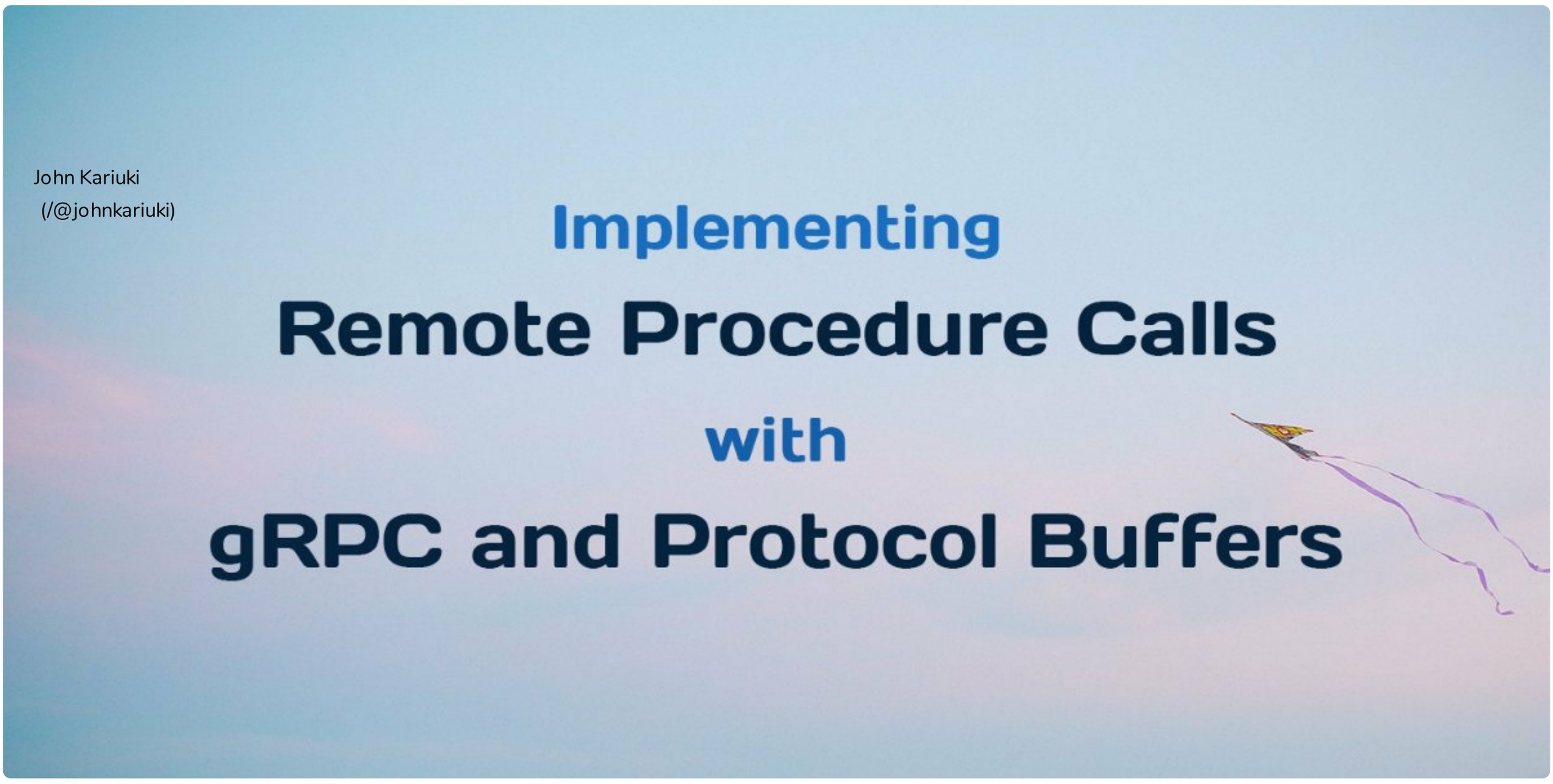
Implementing Remote Procedure Calls With gRPC and Protocol Buffers

John Kariuki (/@johnkariuki) (@_kar_is) (https://twitter.com/_kar_is)

December 12, 2016

#node.js (/tutorials?dFR[tags][0]=node-js) #python (/tutorials?dFR[tags][0]=python) #real time (/tutorials?dFR[tags][0]=real-time)

2 Comments



Code (https://github.com/johnkariuki/grpc-protobuf-tutorial)

<OL> OF CONTENTS

1. Introduction To RPC
2. When and Why to use RPC
3. RPC vs REST. Fight!
4. Implementing RPC with gRPC and Protocol Buffers
5. Advantages Of Using The gRPC Framework
6. Defining Our Sample Application
7. Creating Protocol Buffer Message Types And Service
8. Creating And Testing The Server
9. Creating The Clients

10. Creating a Node Client
11. Creating a Python Client
12. Conclusion

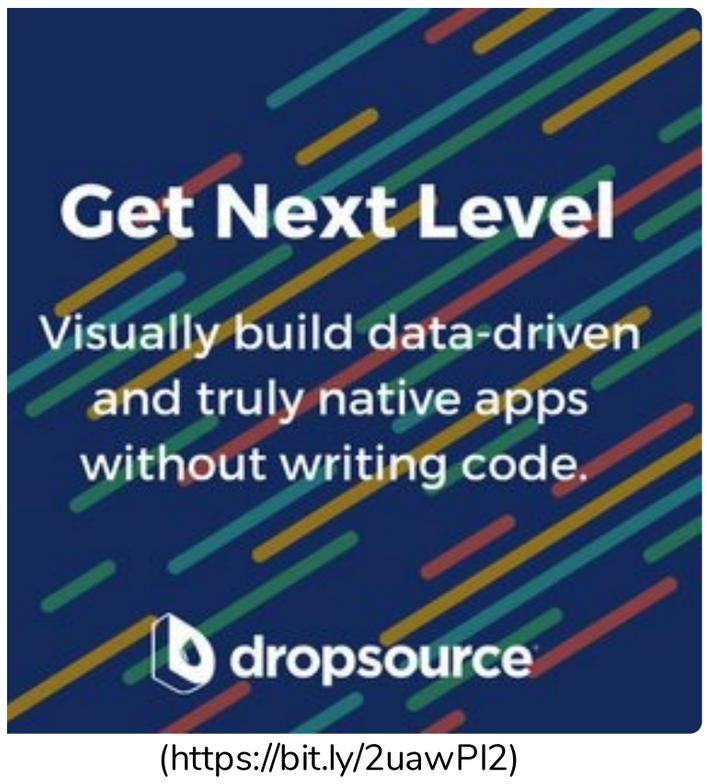
Code (https://github.com/johnkariuki/grpc-protobuf-tutorial)

1

2

(https://synd.co/2tQulT6)

(https://synd.co/2eJbbhF)



(https://bit.ly/2uawPI2)

(https://bit.ly/2uMg0x6)

Introduction To RPC

Remote procedure call (RPC) architecture is popular in building scalable distributed client/server model based applications.

RPC allows a client to a make procedure call (also referred to as subroutine call or function call) to a server on a different address space without understanding the network configuration as if the server was in the same network (making a local procedure call).

In this tutorial, we will implement an RPC client/server model using Google's gRPC and Protocol Buffers.

How RPC Works

Before we dive into the heart of RPC, let's take some time to understand how it works.

1. A client application makes a local procedure call to the client stub (https://en.wikipedia.org/wiki/Stub_(distributed_computing)) containing the parameters to be passed on to the server.
2. The client stub serializes the parameters through a process called marshalling (https://en.wikipedia.org/wiki/Marshalling_(computer_science)) forwards the request to the local client-time library in the local computer which forwards the request to the server stub.
3. The server run-time library receives the request and calls the server stub procedure which unmarshalls (unpacks) the passed parameters and calls the actual procedure.
4. The server stub sends back a response to the client-stub in the same fashion, a point at which the client resumes normal execution.

How RPC works

When and Why to use RPC

While in the process of building out a scalable microservice architecture (https://medium.com/technology-learning/building-out-antifragile-microservice-andela-design-consideration-d6e03a185d6a#.az23n38sb) and Andela (https://andela.com/), we are implementing REST to expose API endpoints, through and API gateway, to external applications (mostly web apps in AngularJS and ReactJS). Inter service communication is then handled using RPC.

Remote procedure call's data serialization into binary format for inter-process (server to server) communication makes it ideal for building out scalable microservice architecture by improving performance.

> RPC client and server run-time stubs take care of the network protocol and communication so that you can focus on your application.

RPC vs REST. Fight!

While RPC and REST use the request/response HTTP protocol to send and receive data respectively, REST is the most popular and preferred client-server communication approach.

Before we dive into RPC. Let's make a comparison of the two.

- REST as an architectural style implements HTTP protocol

- REST as an architectural style implements HTTP protocol between a client and a server through a set of constraints, typically a method (**GET**) and a resource or endpoint (**/users/1**).
- RPC implements client and server stubs that essentially make it possible to make calls to procedures over a network address as if the procedure was local. Simply put, RPC exposes methods (say, **getUsers()**) in the server to the client.

The choice of architecture to use entirely remains at the discretion of the development teams and the nature of the system. Some developers have argued against comparing the two, and rightfully so because they can work well together.

Read more about how the two compare on this article (https://www.linkedin.com/pulse/rest-vs-rpc-soa-showdown-joshua-hartman) by Joshua Hartman (https://www.linkedin.com/in/joshuahartman).

Implementing RPC with gRPC and Protocol Buffers

Google managed gRPC (http://www.grpc.io/) is a very popular open source RPC framework with support of languages such as C++, Java, Python, Go, Ruby, Node.js, C#, Objective-C and PHP.

gRPC is a modern open source high performance RPC framework that can run in any environment. It

can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in last mile of distributed computing to connect devices, mobile applications and browsers to backend services.

http://www.grpc.io/about/ (http://www.grpc.io/about/)

As we mentioned, with gRPC, a client application can directly call methods on a server application on a different network as if the method was local. The beauty about RPC is that it is language agnostic. This means you could have a grpc server written in Java handling client calls from node.js, PHP and Go.

how gRPC works

gRPC allows you to define a service which specifies the methods that you can call remotely, the parameters that can be passed to the procedure call and the return responses. The server then implements this definitions and creates a new grpc server to handle the procedure calls from the client.

By default, gRPC implements Protocol Buffers (https://developers.google.com/protocol-buffers/) to serialize structured data as well as define the parameters and return responses for the callable methods.

Advantages Of Using The gRPC Framework

The gRPC Framework

Before we get into your first RPC application with gRPC, it is good to look at why we should go for gRPC as opposed to other RPC frameworks such as Apache's Thrift (https://thrift.apache.org/).

- gRPC is built on HTTP/2 (https://http2.github.io/) under it's BSD license. What this means is that the procedure calls get the goodness of HTTP/2 to build real time applications taking advantages of features such as bidirectional streaming, flow control, header compression and multiplexing requests.
- gRPC's default serialization protocol, Protocol Buffer, also transmits data in binary format which is smaller and faster as compared to good old JSON and XML.
- Protocol buffer's latest version proto3 (https://github.com/google/protobuf/releases) which makes it easy to define sevices and automatically generate client libraries as we will see later.

Defining Our Sample Application

We will be building a simple RPC based app that let's you employees know if they are eligible for work leave or not, if they are, we will proceed to grant them leave days. The rundown of how we will achieve this

1. Define our protocol buffer message types and service using the proto3 language guide (https://developers.google.com/protocol-buffers/docs/proto3).
2. Create a gRPC server (in Node.js) based on the proto file we define and test it out.
3. Create gRPC clients (in Node.js and Python).
4. Make RPC calls from the respective clients to the server.

Our project structure will be laid out as follows.

```
TEXT
```

```
├── package.json
├── .gitignore
├── .grpclirc
├── server/
    └── index.js
├── client/
    ├── node/
    ├── python/
└── proto/
    └── work_leave.proto
```

Creating Protocol Buffer Message Types And Service

proto/work_leave.proto

```
PROTOBUF
```

```
syntax = "proto3"; //Specify proto3 version.

package work_leave; //Optional: unique package name.
```

```
//Service. define the methods that the grpc server can expose
 to the client.
service EmployeeLeaveDaysService {
  rpc EligibleForLeave (Employee) returns (LeaveEligibility);
  rpc grantLeave (Employee) returns (LeaveFeedback);
}

// Message Type fefinition for an Employee.
message Employee {
  int32 employee_id = 1;
  string name = 2;
  float accrued_leave_days = 3;
  float requested_leave_days = 4;
}

// Message Type definition for LeaveEligibility response.
message LeaveEligibility {
  bool eligible = 1;
}

// Message Type definition for LeaveFeedback response.
message LeaveFeedback  {
  bool granted = 1;
  float accrued_leave_days = 2;
  float granted_leave_days = 3;
}
```

Let's take a logical walk through of the `.proto` file.

- First, we define the protocol buffer language syntax we will be using, in this case, **proto3**
- Next we define a unique package name for our file to prevent name clashes between protocol message types. Note that this is optional.
- Define an RPC service interface that stipulates what parameters each method in the gRPC server takes and returns. In our case, the service name is **EmployeeLeaveDaysService**
- Last but not least, we define a Message Type for each data that we will be transmitting between the client and the server. A Message Type typically consists of a name/value pair where you define a data type, the field name and a unique number tag that identifies the field when serialized into binary format.

Notice how easy it is to interpret the Service method implementations.

- An RPC call to **EligibleForLeave** takes an **Employee** object and

- An RPC call to **EligibleForLeave** takes an **Employee** object and returns a **LeaveEligibility** response.
- An RPC call to **grantLeave** takes an **Employee** object and returns a **LeaveFeedback** response.

Creating And Testing The Server

With our `.proto` file in hand, let's go ahead and create our gRPC server. Since our server is written in Node, go ahead and initalize a new node project and install the grpc (https://www.npmjs.com/package/grpc) package.

BASH

```
npm install --save grpc
```

Next, create a server directory and add the following code to spin up the server.

server/index.js

JAVASCRIPT

```
const grpc = require('grpc');

const proto = grpc.load('proto/work_leave.proto');
const server = new grpc.Server();

//define the callable methods that correspond to the methods
defined in the protofile
server.addProtoService(proto.work_leave.EmployeeLeaveDaysServ

ice.service, {
  /**
  Check if an employee is eligible for leave.
  True If the requested leave days are greater than 0 and wit
hin the number
  of accrued days.
  */
```

```
  eligibleForLeave(call, callback) {
    if (call.request.requested_leave_days > 0) {
      if (call.request.accrued_leave_days > call.request.requ
ested_leave_days) {
        callback(null, { eligible: true });
      } else {
        callback(null, { eligible: false });
      }-1
    } else {
      callback(new Error('Invalid requested days'));
    }
  },

  /**
  Grant an employee leave days
  */
  grantLeave(call, callback) {
    let granted_leave_days = call.request.requested_leave_days
;
    let accrued_leave_days = call.request.accrued_leave_days -
 granted_leave_days;

    callback(null, {
      granted: true,
      granted_leave_days,
      accrued_leave_days
    });
  }
});

//Specify the IP and and port to start the grpc Server, no SS
L in test environment
server.bind('0.0.0.0:50050', grpc.ServerCredentials.createIns
ecure());

//Start the server
server.start();
console.log('grpc server running on port:', '0.0.0.0:50050');
```

First, we load the grpc package and load the proto file using the exposed load method and then create a new instance of ther grpc server.

The server instance gives us access to the following key methods.

- **addProtoService**

Takes two parameters, a link to the service name and a list of methods defined in the respective service. The service name is referenced as

```
<proto>.<package name>.<service name>.service .
```

```
proto + package_name + service_name +service +
```

Each of the defined method takes in a call which contains the payload from the client and a callback function that returns an error and response respectively.

- **bind**

The bind method specifies the IP and port on which to create the server. Since we are on a test environment, we will also create an insecure server(No SSL).

- **start**

The start method simply starts the server.

Testing out the gRPC server

To take our new server for a test drive, let's globally install a CLI grpc client (https://www.npmjs.com/package/grpcli) before we proceed to create our own clients.

BASH

```
npm install -g grpcli
```

Next, let's start our server.

BASH

```
node server
```

To run the client, navigate to the route of your grpc application and run the grpcli command with the path to the protofile, the IP address and port to connect to and a command to start the client as insecured. You can also specify the configurations in a `.grpclirc` file

BASH

```
grpcli -f proto/work_leave.proto --ip=127.0.0.1 --port=50050
-i
```

Once a connection is established, you will be able to list all callable methods with `rpc list` and call methods with `rpc call` .

Test grpc server with grpcli

Sweet! Now with that working, let's go ahead and create the grpc clients.

Creating The Clients

Like we had mentioned earlier, we will be creating gRPC clients in Node.js and Python just because we can. This will help us appreciate the power of RPC.

For each of the clients we will basically confirm if an employee is eligible for leave and if they are, we will proceed to grant them the requested leave days.

Creating a Node Client

Let's go ahead and create an index.js file in client/node and proceed to create our first client.

JAVASCRIPT

```
const grpc = require('grpc');

const protoPath = require('path').join(__dirname, '../..', 'p
roto');
const proto = grpc.load({root: protoPath, file: 'work_leave.p
roto' });

//Create a new client instance that binds to the IP and port
of the grpc server.
const client = new proto.work_leave.EmployeeLeaveDaysService(
'localhost:50050', grpc.credentials.createInsecure());

  const employees = {
    valid: {
      employee_id: 1,
      name: 'John Kariuki',
      accrued_leave_days: 10,
      requested_leave_days: 4
    },
    ineligible: {
      employee_id: 1,
      name: 'John Kariuki',
      accrued_leave_days: 10,
      requested_leave_days: 20
    },
    invalid: {
      employee_id: 1,
      name: 'John Kariuki',
      accrued_leave_days: 10,
      requested_leave_days: -1
    },
    illegal: {
      foo: 'bar'
    }
  }

  client.eligibleForLeave(employees.valid, (error, response) =>
  {
    if (!error) {
      if (response.eligible) {
        client.grantLeave(employees.valid, (error, response) =>

  {
          console.log(response);
        })
      } else {
        console.log("You are currently ineligible for leave day
  s");
      }
    } else {
      console.log("Error:", error.message);
    }
  }
```

```
});
```

Just like we did for the server, we need to load the proto file and the grpc package. We then create a new client instance that binds to our server's address and port.

At this point, we can now directly make calls to the methods defined in our server. As we had mentioned, each method take in a payload parameter and a callback function that returns an error and response respectively.

Let's take a look at how this looks like on our terminal.

simply call

BASH

```
node client/node
```

node client reponse

Simple, but powerful.

The protocol buffer limits us to how the employee object looks like so an error would be raised if you tried passing the `employees.illegal` object. Give it a spin for the different employees provided.

Creating a Python Client

To build a Python client, here are a few prerequisites.

- Install Python 2.6 or higher
- Install pip version 8 or higher, You can upgrade pip with the following command

BASH

```
python -m pip install --upgrade pip
```

- Install virtualenvwrapper (https://virtualenvwrapper.readthedocs.io/en/latest/) to setup a virtual environment workspace for your project. Create a new virtualenv and proceed set it as the current environment

GRPC

```
mkvirtualenv grpc_tutorial && workon grpc_tutorial
```

- Last but not least create client/python/client.py where our Python client code will reside.

Installing gRPC

In our virtual environment, install the grpc and grpc tools packages

BASH

```
pip install grpcio grpcio-tools
```

Compiling the proto file to Python

The `grpcio-tools` module ships with **protoc**, the protocol buffer compiler as well as plugins for generating server and client code from the

service definitions in the proto file.

Create a generate_pb.py file in client/python and add the following code to compile the protocol buffer file to python.

PYTHON

```
from grpc.tools import protoc

protoc.main(
    (
        '',
        '--proto_path=../../proto/',
        '--python_out=.',
        '--grpc_python_out=.',
        '../../proto/work_leave.proto'
    )
)
```

With that, from the python client directory, simply run:

BASH

```
python generate_pb.py
```

This will generate a `client/python/work_leave_pb2.py` which we will use to create our python client.

Creating the client

in client/python/client.py , let's go ahead and load the grpc module and the python protobuf. We then create an insecure channel to bind to the server's address and port and create a stub from the `EmployeeLeaveDaysService` .

PYTHON

```
import grpc
import work_leave_pb2 as pb

def main():
```

```
def main():
    """Python Client for Employee leave days"""

    # Create channel and stub to server's address and port.
    channel = grpc.insecure_channel('localhost:50050')
    stub = pb.EmployeeLeaveDaysServiceStub(channel)

    # Exception handling.
    try:
        # Check if the Employee is eligible or not.
        response = stub.EligibleForLeave(pb.Employee(employee
_id=1,
                                                     name='Pe
ter Pan',
                                                     accrued_
leave_days=10,
                                                     requeste
d_leave_days=5))
        print(response)

        # If the Employee is eligible, grant them leave days.
        if response.eligible:
            leaveRequest = stub.grantLeave(pb.Employee(employ
ee_id=1,
                                                       name='
Peter Pan',
                                                       accrue
d_leave_days=10,
                                                       reques
ted_leave_days=5))
            print(leaveRequest)
    # Catch any raised errors by grpc.
    except grpc.RpcError as e:
        print("Error raised: " + e.details())

if __name__ == '__main__':
    main()
```

Start the node server and run the python client on a different terminal.

Python client demo

#onclusion

For the PHP developers, gRPC can only support PHP clients currently. You can therefore not create a gRPC server with PHP.

In this tutorial, we have managed to create a gRPC server in Node.js and made RPC calls from a Node.js and Python client based on a protocol buffer definition.

For those of you that dare, you may make pull requests to this tutorial's repository (https://github.com/johnkariuki/grpc-protobuf-tutorial) for both gRPC clients and servers in all the other supported languages. Make sure to abide to the directory structure.

(/@johnkariuki)

John Kariuki (/@johnkariuki)

Software developer at Andela (http://andela.com).

Proficient in PHP with Laravel and Codeigniter.

Conversant with MEAN(MongoDB, Express.js, AngularJS, Node.js) and currently learning Python and Go.

Avid blog reader and fascinated by drones.

I play basketball, swim and jog in my free time.

(/@johnkariuki) (https://twitter.com/_kar_is) (https://facebook.com/kariuki.j)

(https://github.com/andela-jkariuki) (https://instagram.com/_kar_is)

Popular In the Community

BUILD A CRYPTOCURRENCY

GoldBug
2h

kkkkkkk

BUILD A MINI NETFLIX WITH REACT IN 10

BlueApple
21 Jun

I don't think the demo works?

BUILD AN IONIC APP WITH USER

YotaMan
15 Jul

I got this error when I start ionic serve

BUILD A PROGRESSIVE WEB APP: OFFLINE GIT

BlueSword
4d

How are you serving these files while

FORM ROUTING

OliveBeer
10 Jul

Very good tutorials.Thank you so

UNDERSTANDING AND USING LARAVEL

Connor Leech
8 Jul

When you define the macro in the boot

AN INTRODUCTION TO APACHE KAFKA

RedCocktail
5 Jul

This is a very good article about kafka!

WHY JWTS SUCK AS SESSION TOKENS

Julius Koronci
13 Jul

You article has one key error..and that is

DELETING EVENTS

OliveFlask
5 Jul

Really smooth tutorial to create a basic CRUD

NODE.JS TESTS: MOCKING HTTP

EvanBoissonnot
1d

Hi John Thank you for your post ! Really

THE ULTIMATE GUIDE TO PROGRESSIVE WEB

abidikoussai
16 Jun

Well explained guide, very useful for me.

CREATE & DEPLOY A MICROSERVICE IN LESS

GreenBus
2d

Cool Thanx!

Hey guest, welcome to **Scotch!** Sign up and become a member.

Facebook **Google** **Twitter** **Email**

Popular In the Community

BUILD A CRYPTOCURRENCY

GoldBug
2h

kkkkkkk

BUILD A MINI NETFLIX WITH REACT IN 10

BlueApple
21 Jun

I don't think the demo works?

BUILD AN IONIC APP WITH USER

YotaMan
15 Jul

I got this error when I start ionic serve

BUILD A PROGRESSIVE WEB APP: OFFLINE GIT

BlueSword
4d

How are you serving these files while

AN INTRODUCTION TO APACHE KAFKA

RedCocktail
5 Jul

This is a very good article about kafka!

WHY JWTS SUCK AS SESSION TOKENS

Julius Koronci
13 Jul

You article has one key error..and that is

DELETING EVENTS

OliveFlask
5 Jul

Really smooth tutorial to create a basic CRUD

NODE.JS TESTS: MOCKING HTTP

EvanBoissonnot
1d

Hi John Thank you for your post ! Really

Conversation (2)

Sort by **Best**

Have a Disqus Account? **Log In**

Add a comment...

Olive Bullhorn

6 Jun

There is a small typo on the 2nd point of 'How RPC Works' title, it goes '...local client-time...' instead of 'local client run-time...'. Thanks for the article though :)

Reply · Share ·

Vishwasrao Salunkhe

29 Apr

Hi, Just few questions aboit grpc server and client.
Let say if i want to use this thing in my microservices architecture. It has 5 different microservices(lets take all in nodejs) if i want to use grpc wihin them how will be my architecture in terms of grpc server and client? Only one grpc server and 5 client or how it will be?

Reply · Share ·

Terms · Privacy

Add Spot.IM to your site ·

See More Courses (/courses)

(/courses/getting-started-with-angular-2) Free

2.1 hours

Getting Started with Angular v4 (/courses/getting-started-with-angular-2)

(/courses/getting-started-with-react)

1.4 hours

Getting Started with React (/courses/getting-started-with-react)

(/courses/getting-started-with-javascript) Free

 4.7 hours

Getting Started with JavaScript for Web Development (/courses/getting-started-with-javascript)

(/courses/get-to-know-git)

 0.9 hours

Get to Know Git (/courses/get-to-know-git)

Join Scotch

NO GLYPH

High Quality Content

The best tutorials and content that you'll find for web development. Guides, courses, tutorials, and more great content to learn with.

NO GLYPH

Build Real Apps

We won't just go over concepts and "Hello Worlds"; we'll **build real apps together** that you can use at your job or for your portfolio.

NO GLYPH

Not Just How, But Why

There are many different ways to code the same project. We'll show **best practices** and why certain choices are better than others.

Scotch Free

Like your favorite posts

Bookmark content for reference

Post in the forums

Free (/registering?type=free)

Scotch Premium

All of the free features

Access to **all premium content**

Downloadable videos

Access to **live chat**

No ads across all of Scotch

Track **completed** content

$20 (/registering?type=monthly)

(https://bit.ly/2s4Z7Z6)

Scotch.io Stickers!

(https://bit.ly/2tDecCW)

Join Us On Slack

(https://bit.ly/2tDcLEK)

Easiest Local Dev Environment

(https://bit.ly/2eF8uO2)

Practical Angular v4+ Book

scotch

Top shelf learning. Informative tutorials explaining the code **and the choices behind it all.**

(https://twitter.com/scotchio) (https://facebook.com/scotchio) (https://github.com/scotch-io) (https://scotch.io/feed) (https://scotch.io/development)

Scotch (https://scotch.io)

About (/about)

Contact (/contact)

Advertising (/advertise)

Community

Forums (https://scotch.io/lounge)

Slack (http://slack.scotch.io)

Extras

Become a Writer (/write-for-us)

Shop (https://shop.scotch.io)

Scotch Box (https://box.scotch.io)

BROUGHT TO YOU BY... (/ABOUT)

(https://scotch.io/@chris)

Chris Sevilleja
(https://scotch.io/@nick)

Nick Cerminara

2017 © Scotch.io, LLC. All Rights Super Duper Reserved.

Proudly hosted by Digital Ocean (https://m.do.co/c/7a59e9361ab7)