# 

[Introduction](#h.msnf4ya4l9aa)

[Hello](#h.af32ede0ntg5)

[Getting ProtoRPC](#h.ygm805s2z7a5)

[Installation](#h.1ve08y588vbu)

[Defining a service](#h.uew8wejt3vp0)

[What a service is made of](#h.dko46n6eg205)

[The postservice](#h.29vd7woeeaef)

[Working with messages](#h.vzfqndlz311a)

[Defining a service](#h.gwq8v5uu353x)

[Registering the service](#h.xqzpscxrpxm3)

[Testing the service from the command line](#h.hr9bwy3hfv6d)

[More fields](#h.29jwf6318m3h)

[Default values](#h.5pgie57m76n6)

[Enum types](#h.2h8290zh4tws)

[More complex messages](#h.rpu9rltso3pk)

[Message fields](#h.5r9fza9bgq4k)

[Repeated fields](#h.2np0euocp7ku)

[Implement get\_notes](#h.9fnlu8rtr7mq)

[The forms interface](#h.mgl4hy13i1e1)

[The RegistryService](#h.u16jyt30boo)

[Viewing a method form](#h.tx6tkpjeyopv)

[Creating a client program](#h.j3s7a0napbai)

[Generate postservice module from the server](#h.ajmd7lky9f03)

[Service stubs](#h.sb6eiu2s1a5j)

[Invoke remote method](#h.jnqtbslzyvjv)

[Sending parameters](#h.2m8b5n36e59x)

# 

# **Introduction**

ProtoRPC is a simple way to create well defined and easy to use web-based Remote Procedure Calls (RPC) services. An RPC service is a collection of message types and remote methods that provide a structured way for software clients to interact with your web applications.

For example, defining RPCs between your applications Javascript and its server can be very repetitive and error prone. On the server side it requires defining a URL for each method, checking the format of each parameter so that they can be converted to the right type and converting them to a usable Python object. On the client side, you have to check user input and transform them in to an HTTP request, then transform the response data back into a Javascript object. To help simplify this kind of translation between different languages, ProtoRPC provides a structured way to define your RPC message and services in only one place as server-side Python objects. Because it is possible to define these messages and services using only the Python programming language, it's easy to get started developing your own services.

## **Hello**

Here is an example of a very simple service definition:

from google.appengine.ext import webapp

from google.appengine.ext.webapp import util

from protorpc import messages

from protorpc import remote

from protorpc.webapp import service\_handlers

class HelloRequest(messages.Message):

my\_name = messages.StringField(1, required=True)

class HelloResponse(messages.Message):

hello = messages.StringField(1, required=True)

class HelloService(remote.Service):

@remote.method(HelloRequest, HelloResponse)

def hello(self, request):

return HelloResponse(hello='Hello there, %s!' %

request.my\_name)

service\_mappings = service\_handlers.service\_mapping(

[('/hello', HelloService),

])

application = webapp.WSGIApplication(service\_mappings)

def main():

util.run\_wsgi\_app(application)

if \_\_name\_\_ == '\_\_main\_\_':

main()

This very simple service receives a message from a remote client that contains someones name (HelloRequest.my\_name) and sends back a greeting for that person (HelloResponse.hello). Notice that there is no need to translate the Python message types into JSON objects, nor is it necessary to handle the underlying HTTP request. This is all done in the background for you.

One way to use this service from Javascript is by using JQuery:

$.ajax({url: ‘/hello.hello’,

type: 'POST',

contentType: 'application/json',

data: ‘{ my\_name: Bob }’,

dataType: 'json',

success: function(response) {

// The response is { hello: “Hello there, Bob!” }

alert(response.hello);

}

});

This will send a JSON request to the server, which will return a JSON response to the browser. Since ProtoRPC messages are specified using a well defined set of data-types, the resulting response maps very cleanly to Javascript objects by jQuery for use by the success handler.

# **Getting ProtoRPC**

The best way to work with ProtoRPC is by accessing the source code repository directly using [Mercurial](http://www.google.com/url?q=http%3A%2F%2Fmercurial.selenic.com%2F&sa=D&sntz=1&usg=AFQjCNHxqy4rE5UfjxR4uy-R-aep38MfIw). Please see the [Source](http://code.google.com/p/google-protorpc/source/checkout) page for more information about cloning the repository to your local machine.

Alternatively, download and unpack the latest [snapshot](http://code.google.com/p/google-protorpc/downloads/detail?name=google-protorpc-snapshot.zip). Once downloaded, unzip the snapshot to a convenient location.

# **Installation**

ProtoRPC is still in the early phase of development. For now, the easiest way to work with ProtoRPC is with [Google App Engine](http://code.google.com/appengine). To use ProtoRPC in to your application, simply copy the <protorpc-source-dir>/python/protorpc in your App Engine application directory. See App Engine documentation for information about setting up and testing App Engine applications.

## **Defining a service**

If you’ve gone through the [App Engine Getting Started](http://code.google.com/appengine/docs/python/gettingstarted/) guide, you ended up building a guestbook application. This guestbook application is also available in the Python SDK as the guestbook demo. The guestbook is a typical web application that allows people to go to a web page and enter a message for other visitors of the guestbook to read. As written, the guestbook application has a strictly web based interface meant for users to interact with directly. However, it does not expose functionality that other web applications could easily use to access that information. For example, someone might want to write a tool that reads the messages posted by uses and makes a time-series graph of posts per day.

## **What a service is made of**

A ProtoRPC is defined a single Python class that contains any number of declared remote methods. Each remote method accepts a specific set of parameters as a request and returns a specific response. These request and response parameters are user defined classes known as messages.

## **The postservice**

The module postservice contains a ProtoRPC service that exposes two remote methods used for accessing data of the guestbook application:

* **post\_note:** Post a note to the App Engine datastore.
* **get\_notes:** Get notes that were previously posted.

Create the postservice module by creating a file called postservice.py in your application directory. All the definition needed for the new service will go in this file.

## **Working with messages**

Messages are the fundamental data-type used in ProtoRPC. They are defined by declaring a class that inherits from the Message base class and defining class attributes that corresponds to each of its fields.

For example, the guestbook service allows users to post a note. Let’s define a message that represents such a note:

from protorpc import messages

class Note(messages.Message):

text = messages.StringField(1, required=True)

when = messages.IntegerField(2)

The note message is defined by two fields, text and when. Each field of a message has a specific type. The text field is a unicode string representing the content of a users post to the guestbook page. The when field is an integer representing when the timestamp of when the note was posted. Each field is also given a unique numeric value that is used by the underlying network protocol to instead of the name for its identification. Values for the fields can be set using the constructor of the Note class:

import time

note\_instance = Note(text=u’Hello guestbook!’,

when=int(time.time())

Values on a message can also be read and set like normal Python attribute values. For example, to change the message:

print note\_instance.text

note\_instance.text = u‘Good-bye guestbook!’

print note\_instance.text

The output of this would be:

Hello guestbook!

Good-bye guestbook!

Single value fields like text and when can also be required or optional. By default fields are optional, but can be marked as required setting required=True. Required fields must be set to a value in order for the message instance to be considered initialized. Only properly initialized messages are accepted by ProtoRPC service methods.

## **Defining a service**

A service is a class definition that inherits from the Service base-class. Remote methods of a service are indicated by using the method decorator. Every method of a service accepts a single message as it’s parameter and returns a single message as its response.

Let’s define the first method of the PostService. If you’ve followed the guestbook tutorial from App Engine, you might recall that guest book greetings are put in the Datastore using the [guestbook.Greeting](http://code.google.com/appengine/docs/python/gettingstarted/usingdatastore.html) class. The PostService will also use the Greeting class to store a post in the datastore.

import datetime

from protorpc import message\_types

from protorpc import remote

import guestbook

class PostService(remote.Service):

@remote.method(Note, message\_types.VoidMessage)

def post\_note(self, request):

if request.when:

when = datetime.datetime.utcfromtimestamp(request.when)

else:

when = datetime.datetime.now()

note = guestbook.Greeting(content=request.text, date=when)

note.put()

return message\_types.VoidMessage()

The method decorator takes two parameters. The first parameter is the expected request type. The second parameter is the expected response type. To post a note, the post\_note method accepts a Note instance as it’s request type. For the response type, ProtoRPC comes with a built-in type called a VoidMessage, which is defined as a message with no fields. This means that the post\_note message does not return anything useful to its caller. If it returns without error the message is considered to have been posted.

Since Note.when is an optional field, it may not have been set by the caller. When this happens, the value of when is set to None. The post\_note remote method take the absence of a specific timestamp to mean that the current time should be used.

The response message is simply instantiated by the remote method and used as its return value.

## **Registering the service**

You can publish your new service using the App Engine webapp framework. ProtoRPC has a small library to make it easy to do this. Create a new file called services.py in your application directory. Add the following to create your webapp based service:

from google.appengine.ext import webapp

from google.appengine.ext.webapp import util

from protorpc.webapp import service\_handlers

import postservice

# Register mapping with application.

application = webapp.WSGIApplication(

service\_handlers.service\_mapping(

[('/postservice', postservice.PostService)]),

debug=True)

def main():

util.run\_wsgi\_app(application)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Add the following handlers to your app.yaml file:

- url: /postservice.\*

script: services.py

## **Testing the service from the command line**

It’s now possible to test your service using a curl or a similar command-line tool.

% curl -H \

'content-type: application/json' \

-d {"text": "Hello guestbook!"}'\

http://localhost:8080/postservice.post\_note

If the server returns an empty JSON response it means that the note has been sucessfully posted. You can see the note by going to your guestbook application in your browser: http://localhost:8080/.

## **More fields**

Now that we have the ability to post new messages to the postservice, let’s add a new get\_notes method so that there is a way to get them. First, define a request message in postservice.py above the service class.

class GetNotesRequest(messages.Message):

limit = messages.IntegerField(1, default=10)

on\_or\_before = messages.IntegerField(2)

class Order(messages.Enum):

WHEN = 1

TEXT = 2

order = messages.EnumField(Order, 3, default=Order.WHEN)

### **Default values**

When sent to the post service, this message requests a number of notes on or before a certain date and in a particular order. The limit field indicates the maximum number of notes to fetch. If limit is not explicitly set, limit will be set to 10 as indicated by the default keyword argument.

### **Enum types**

The order field introduces a new enum field type. Use enum when the value of a field is restricted to a limited number of known symbolic values. In this case, an enum is defined to indicate how the server should order the notes in the response. To define the enum values, create a sub-class of the Enum class. Each name must be assigned a unique number for the type. Each number is converted to an instance of the enum type and can be accessed from the class.

print 'Enum value Order.%s has number %d' % (Order.WHEN.name,

Order.WHEN.number)

Each enum value has a special charactaristic that makes them easy to convert to their name or their number. Instead of accessing the name and number attribute, just convert each value to a string or an integer.

print 'Enum value Order.%s has number %d' % (Order.WHEN,

Order.WHEN)

Enum fields are declared similarly to other fields except they must have the enum type as its first parameter before the field number. Enum fields can also have default values.

## **More complex messages**

Now let’s define the get\_notes response message. The response needs to be a collection of Note messages.

class Notes(messages.Message):

notes = messages.MessageField(Note, 1, repeated=True)

### **Message fields**

Messages can contain other messages. In the case of the Notes.notes field defined above, we provide Note as the first parameter before the field number to indicate this.

### **Repeated fields**

The Notes.notes field is also a repeated field as indicated by the repeated keyword argument. Values of repeated fields must be lists of the field type of their declaration. In this case, Notes.notes must a list of Note instances.

For example, here is how to create a Notes object:

response = Notes(notes=[Note(text='This is note 1'),

Note(text='This is note 2')])

print 'First Note is:', response.notes[0].text

print 'Second Note is:', response.notes[1].text

## **Implement get\_notes**

Add the get\_notes method to the PostService class.

@remote.method(GetNotesRequest, Notes)

def get\_notes(self, request):

query = guestbook.Greeting.all().order('-date')

if request.on\_or\_before:

when = datetime.datetime.utcfromtimestamp(

request.on\_or\_before)

query.filter('date <=', when)

notes = []

for note\_model in query.fetch(request.limit):

if note\_model.date:

when = int(time.mktime(note\_model.date.utctimetuple()))

else:

when = None

note = Note(text=note\_model.content, when=when)

notes.append(note)

if request.order == GetNotesRequest.Order.TEXT:

notes.sort(key=lambda note: note.text)

return Notes(notes=notes)

Don’t forget to import the time module at the top of the file

import time

## **The forms interface**

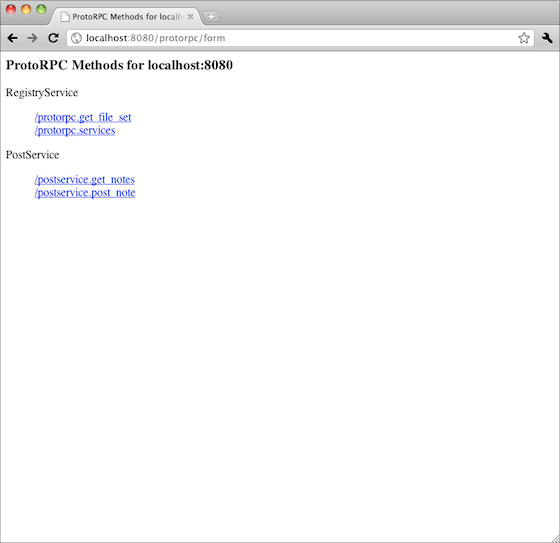
There is another way to test ProtoRPC remote methods using the webapp framework. The service\_mapping function used to register and publish the service to the web actually adds a few other web handlers including a forms interface. The forms interface lets you view all of your registered services and methods as a web page. You can bring up forms that you can fill out and post to directly your service without needing to use a tool like curl. All you need to do is expose the right path in your application by adding the following lines to your app.yaml.

- url: /protorpc.\*

script: services.py

You can view the ProtoRPC forms interface by navigating your browser to http://localhost:8080/protorpc/form.

You will see something like this:

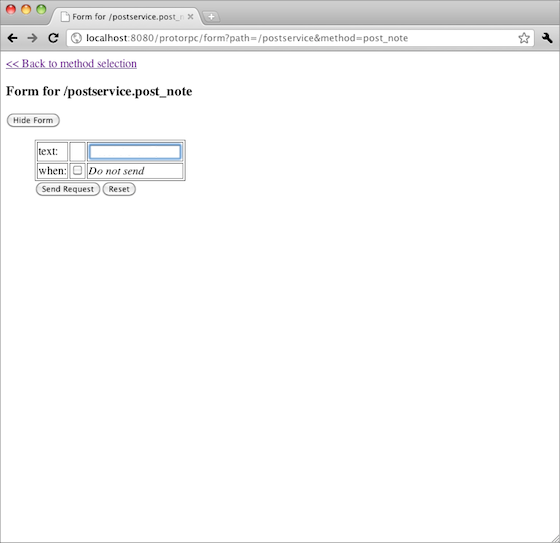


This shows that two registered services: The RegistryService on /protorpc and the PostService on /postservice. But only the PostService was explicitly registered! What gives?

### **The RegistryService**

Unless otherwise specified, the service\_mapping function automatically registers a built in service called RegistryService. The registry services is configured with information about registered services. This is a very useful for service that you can use to discover what kinds of services are available on a web site. It’s also the service that makes the forms interface possible. For now, it’s enough to know that it’s there.

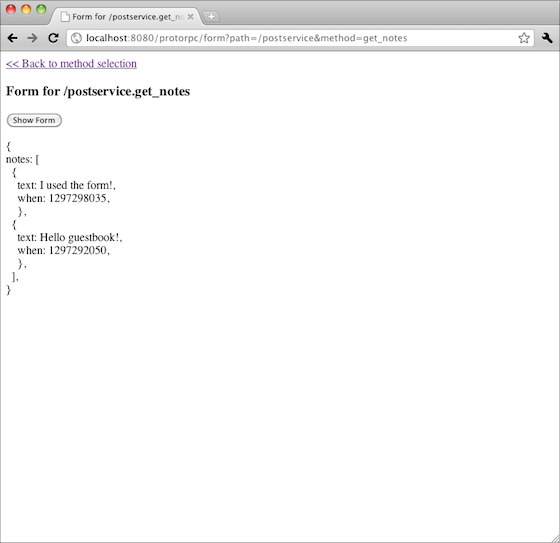
### **Viewing a method form**

Let’s take a look at the form for the PostService.post\_note method. Click on that link to navigate to it. You should see something like this:

The form has a field corresponding to the ones defined on the Note class. Since the when field is optional, it is set to not be sent to the server by default. To edit that field and send its value to the server, you can click on the checkbox to the right of its label, but don’t do that now. For now, enter a message in to the text field, such as “I used the form!” and then click the “Send Request” button below the form. If the request is successful, the form will disappear and the response will appear in its place. Like the response from the curl command, it will just be an empty JSON response.

Go ahead and add a few more notes to your guestbook. When finished, click on the “Back to method selection” link at the top of the page. From there select the PostService.get\_notes method. You will then see something like this:



Press the “Send Request” button and you will get a more interesting response than when you sent a request to PostService.post\_note. You might get something like:

Click the “Show Form” button to bring up the form again. Try enabling a field, such as order, change its value and send the request again to see if you get a different result.

As you can see, the forms interface is very useful for testing your service.

## **Creating a client program**

Time to try building another webapp that connects to the postservice and does something “useful” with the guestbook data. Let’s call this application “gueststats”. It collects and publishes information about recent posts and displays them to the user.

Create a new App Engine application in a new directory. Copy the protorpc library in to it just like you did the server.

## **Generate postservice module from the server**

The server side version of the postservice module contains the server logic, which might be necessary to keep secret and may have dependencies to libraries that are not necessary to use on the client side. Since the server publishes the RegistryService, it’s possible to automatically generate all the necessary modules needed to write a client. To do that, ProtoRPC provides a command line tool called gen\_protorpc.py. Change your shell to the client application directory and run the command against the server to generate the module source code. This example is written using a unix-like shell on MacOS.

gen\_protorpc.py registry localhost:8080 /postservice

If you take a peek in the application directory, you will see a new file called postservice.py. If you take a look at that file it will look very similar to the postservice.py you wrote for the server except the methods of PostService doesn’t actually have any implemented methods.

### **Service stubs**

Communication with the server happens through the client side stub class. Each server has a Stub sub-class that has all the service methods on it.

from protorpc import transport

import postservice

service = postservice.PostService.Stub(

transport.HttpTransport(

'http://localhost:8080/postservice'))

### **Invoke remote method**

Write a webapp handler that uses the stub to invoke the get\_notes method from the server.

from google.appengine.ext import webapp

from google.appengine.ext.webapp import util

class MainHandler(webapp.RequestHandler):

def get(self):

notes = service.get\_notes().notes

if notes is None:

notes = []

note\_count = len(notes)

note\_lengths = [len(note.text) for note in notes]

total\_characters = sum(note\_lengths)

if note\_count == 0:

average\_characters = 0

else:

average\_characters = total\_characters / note\_count

self.response.out.write('<br>Notes retrieved: %d\n' %

note\_count)

self.response.out.write('<br>Total characters: %d\n' %

total\_characters)

self.response.out.write('<br>Average characters: %d\n' %

average\_characters)

def main():

application = webapp.WSGIApplication([('/', MainHandler)],

debug=True)

util.run\_wsgi\_app(application)

if \_\_name\_\_ == '\_\_main\_\_':

main()

Run another instance of the dev\_appserver that runs on a different port from the guestbook service. When it has started, navigate to that server in your browser to see some states about your notes.

### **Sending parameters**

So far, the client application invoked get\_notes without any parameters. However, we know that server requires a GetNotesRequest object as its request message. It would be nice to take advantage of that request object to customize the response. For example, let’s add a max\_notes query parameter so that we can change the number of notes to fetch from the server.

What’s happening is the stub method secretly creates a GetNotesRequest instance behind the scenes and sends it to the server. You can set fields on that request object by passing in keyword arguments to the get\_notes method. To do this, change the first line of the get() method to:

max\_notes = int(self.request.params.get(

'max\_notes', '10'))

notes = service.get\_notes(limit=max\_notes).notes

Alternatively, you can pass get\_notes a whole instance of GetNotesRequest.

request = postservice.GetNotesRequest(limit=max\_notes)

notes = service.get\_notes(request).notes