django-grpc-framework

Release 0.2.1

Jul 23, 2021

Contents

Django gRPC framework is a toolkit for building gRPC services with Django. Officially we only support proto3.

CHAPTER 1

User's Guide

This part of the documentation begins with installation, followed by more instructions for building services.

1.1 Installation

1.1.1 Requirements

We requires the following:

- Python (3.6, 3.7, 3.8)
- Django (2.2, 3.0)
- Django REST Framework (3.10.x, 3.11.x)
- gRPC
- gRPC tools
- proto3

1.1.2 virtualenv

Virtualenv might be something you want to use for development! let's create one working environment:

```
$ mkdir myproject
$ cd myproject
$ python3 -m venv env
$ source env/bin/activate
```

It is time to get the django grpc framework:

```
$ pip install djangogrpcframework
$ pip install django
$ pip install djangorestframework
$ pip install grpcio
$ pip install grpcio-tools
```

1.1.3 System Wide

Install it for all users on the system:

```
$ sudo pip install djangogrpcframework
```

1.1.4 Development Version

Try the latest version:

```
$ source env/bin/activate
$ git clone https://github.com/fengsp/django-grpc-framework.git
$ cd django-grpc-framework
$ python setup.py develop
```

1.2 Quickstart

We're going to create a simple service to allow clients to retrieve and edit the users in the system.

1.2.1 Project setup

Create a new Django project named `quickstart`, then start a new app called `account`:

```
# Create a virtual environment
python3 -m venv env
source env/bin/activate
# Install Django and Django gRPC framework
pip install django
pip install djangorestframework
pip install djangogrpcframework
pip install grpcio
pip install grpcio-tools
# Create a new project and a new application
django-admin startproject quickstart
cd quickstart
django-admin startapp account
```

Now sync the database:

```
python manage.py migrate
```

1.2.2 Update settings

Add `django_grpc_framework` to `INSTALLED_APPS`, settings module is in `quickstart/settings.py`:

```
INSTALLED_APPS = [
    ...
    'django_grpc_framework',
]
```

1.2.3 Defining protos

Our first step is to define the gRPC service and messages, create a file `quickstart/account.proto` next to `quickstart/manage.py`:

```
syntax = "proto3";

package account;

import "google/protobuf/empty.proto";

service UserController {
    rpc List(UserListRequest) returns (stream User) {}
    rpc Create(User) returns (User) {}
    rpc Retrieve(UserRetrieveRequest) returns (User) {}
    rpc Update(User) returns (User) {}
    rpc Destroy(User) returns (google.protobuf.Empty) {}
}

message User {
    int32 id = 1;
    string username = 2;
    string email = 3;
    repeated int32 groups = 4;
}

message UserListRequest {
}

message UserRetrieveRequest {
    int32 id = 1;
}
```

Or you can generate it automatically based on `User` model:

```
python manage.py generateproto --model django.contrib.auth.models.User --fields id,
↪username,email,groups --file account.proto
```

Next we need to generate gRPC code, from the `quickstart` directory, run:

```
python -m grpc_tools.protoc --proto_path=./ --python_out=./ --grpc_python_out=./ ./
↪account.proto
```

1.2.4 Writing serializers

Then we're going to define a serializer, let's create a new module named `account/serializers.py`:

```
from django.contrib.auth.models import User
from django_grpc_framework import proto_serializers
import account_pb2


class UserProtoSerializer(proto_serializers.ModelProtoSerializer):
    class Meta:
        model = User
        proto_class = account_pb2.User
        fields = ['id', 'username', 'email', 'groups']
```

1.2.5 Writing services

Now we'd write some a service, create `account/services.py`:

```
from django.contrib.auth.models import User
from django_grpc_framework import generics
from account.serializers import UserProtoSerializer


class UserService(generics.ModelService):
    """
    gRPC service that allows users to be retrieved or updated.
    """
    queryset = User.objects.all().order_by('-date_joined')
    serializer_class = UserProtoSerializer
```

1.2.6 Register handlers

Ok, let's wire up the gRPC handlers, edit `quickstart/urls.py`:

```
import account_pb2_grpc
from account.services import UserService


urlpatterns = []


def grpc_handlers(server):
    account_pb2_grpc.add_UserControllerServicer_to_server(UserService.as_servicer(),␣
↪server)
```

We're done, the project layout should look like:

```
.
./quickstart
./quickstart/asgi.py
./quickstart/__init__.py
./quickstart/settings.py
./quickstart/urls.py
./quickstart/wsgi.py
./manage.py
./account
./account/migrations
```

(continues on next page)

(continued from previous page)

```
./account/migrations/__init__.py
./account/services.py
./account/models.py
./account/serializers.py
./account/__init__.py
./account/apps.py
./account/admin.py
./account/tests.py
./account.proto
./account_pb2_grpc.py
./account_pb2.py
```

1.2.7 Calling our service

Fire up the server with development mode:

```
python manage.py grpcrunserver --dev
```

We can now access our service from the gRPC client:

```
import grpc
import account_pb2
import account_pb2_grpc


with grpc.insecure_channel('localhost:50051') as channel:
    stub = account_pb2_grpc.UserControllerStub(channel)
    for user in stub.List(account_pb2.UserListRequest()):
        print(user, end='')
```

1.3 Tutorial

This part provides a basic introduction to work with Django gRPC framework. In this tutorial, we will create a simple blog rpc server. You can get the source code in tutorial example.

1.3.1 Building Services

This tutorial will create a simple blog gRPC Service.

Environment setup

Create a new virtual environment for our project:

```
python3 -m venv env
source env/bin/activate
```

Install our packages:

```
pip install django
pip install djangorestframework   # we need the serialization
pip install djangogrpcframework
pip install grpcio
pip install grpcio-tools
```

Project setup

Let's create a new project to work with:

```
django-admin startproject tutorial
cd tutorial
```

Now we can create an app that we'll use to create a simple gRPC Service:

```
python manage.py startapp blog
```

We'll need to add our new `blog` app and the `django_grpc_framework` app to `INSTALLED_APPS`. Let's edit the `tutorial/settings.py` file:

```
INSTALLED_APPS = [
    ...
    'django_grpc_framework',
    'blog',
]
```

Create a model

Now we're going to create a simple `Post` model that is used to store blog posts. Edit the `blog/models.py` file:

```
from django.db import models


class Post(models.Model):
    title = models.CharField(max_length=100)
    content = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['created']
```

We also need to create a migration for our post model, and sync the database:

```
python manage.py makemigrations blog
python manage.py migrate
```

Defining a service

Our first step is to define the gRPC service and messages, create a directory `tutorial/protos` that sits next to `tutorial/manage.py`, create another directory `protos/blog_proto` and create the `protos/blog_proto/post.proto` file:

```
syntax = "proto3";

package blog_proto;

import "google/protobuf/empty.proto";

service PostController {
    rpc List(PostListRequest) returns (stream Post) {}
    rpc Create(Post) returns (Post) {}
    rpc Retrieve(PostRetrieveRequest) returns (Post) {}
    rpc Update(Post) returns (Post) {}
    rpc Destroy(Post) returns (google.protobuf.Empty) {}
}

message Post {
    int32 id = 1;
    string title = 2;
    string content = 3;
}

message PostListRequest {
}

message PostRetrieveRequest {
    int32 id = 1;
}
```

For a model-backed service, you could also just run the model proto generator:

```
python manage.py generateproto --model blog.models.Post --fields=id,title,content --
↪file protos/blog_proto/post.proto
```

Then edit it as needed, here the package name can't be automatically inferred by the proto generator, change `package post` to `package blog_proto`.

Next we need to generate gRPC code, from the `tutorial` directory, run:

```
python -m grpc_tools.protoc --proto_path=./protos --python_out=./ --grpc_python_out=./
↪ ./protos/blog_proto/post.proto
```

Create a Serializer class

Before we implement our gRPC service, we need to provide a way of serializing and deserializing the post instances into protocol buffer messages. We can do this by declaring serializers, create a file in the `blog` directory named `serializers.py` and add the following:

```
from django_grpc_framework import proto_serializerss
from blog.models import Post
from blog_proto import post_pb2


class PostProtoSerializer(proto_serializers.ModelProtoSerializer):
    class Meta:
        model = Post
        proto_class = post_pb2.Post
        fields = ['id', 'title', 'content']
```

Write a service

With our serializer class, we'll write a regular grpc service, create a file in the `blog` directory named `services.py` and add the following:

```
import grpc
from google.protobuf import empty_pb2
from django_grpc_framework.services import Service
from blog.models import Post
from blog.serializers import PostProtoSerializer


class PostService(Service):
    def List(self, request, context):
        posts = Post.objects.all()
        serializer = PostProtoSerializer(posts, many=True)
        for msg in serializer.message:
            yield msg

    def Create(self, request, context):
        serializer = PostProtoSerializer(message=request)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return serializer.message

    def get_object(self, pk):
        try:
            return Post.objects.get(pk=pk)
        except Post.DoesNotExist:
            self.context.abort(grpc.StatusCode.NOT_FOUND, 'Post:%s not found!' % pk)

    def Retrieve(self, request, context):
        post = self.get_object(request.id)
        serializer = PostProtoSerializer(post)
        return serializer.message

    def Update(self, request, context):
        post = self.get_object(request.id)
        serializer = PostProtoSerializer(post, message=request)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return serializer.message

    def Destroy(self, request, context):
        post = self.get_object(request.id)
        post.delete()
        return empty_pb2.Empty()
```

Finally we need to wire there services up, create `blog/handlers.py` file:

```
from blog._services import PostService
from blog_proto import post_pb2_grpc


def grpc_handlers(server):
    post_pb2_grpc.add_PostControllerServicer_to_server(PostService.as_servicer(),␣
↪server)
```

Also we need to wire up the root handlers conf, in `tutorial/urls.py` file, include our blog app's grpc handlers:

```
from blog.handlers import grpc_handlers as blog_grpc_handlers


urlpatterns = []


def grpc_handlers(server):
    blog_grpc_handlers(server)
```

Calling our service

Now we can start up a gRPC server so that clients can actually use our service:

```
python manage.py grpcrunserver --dev
```

In another terminal window, we can test the server:

```
import grpc
from blog_proto import post_pb2, post_pb2_grpc


with grpc.insecure_channel('localhost:50051') as channel:
    stub = post_pb2_grpc.PostControllerStub(channel)
    print('----- Create -----')
    response = stub.Create(post_pb2.Post(title='t1', content='c1'))
    print(response, end='')
    print('----- List -----')
    for post in stub.List(post_pb2.PostListRequest()):
        print(post, end='')
    print('----- Retrieve -----')
    response = stub.Retrieve(post_pb2.PostRetrieveRequest(id=response.id))
    print(response, end='')
    print('----- Update -----')
    response = stub.Update(post_pb2.Post(id=response.id, title='t2', content='c2'))
    print(response, end='')
    print('----- Delete -----')
    stub.Destroy(post_pb2.Post(id=response.id))
```

1.3.2 Using Generic Services

We provide a number of pre-built services as a shortcut for common usage patterns. The generic services allow you to quickly build services that map closely to database models.

Using mixins

The create/list/retrieve/update/destroy operations that we've been using so far are going to be similar for any model-backend services. Those operations are implemented in gRPC framework's mixin classes.

Let's take a look at how we can compose the services by using the mixin classes, here is our `blog/services` file again:

```
from blog.models import Post
from blog.serializers import PostProtoSerializer
from django_grpc_framework import mixins
from django_grpc_framework import generics


class PostService(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  mixins.RetrieveModelMixin,
                  mixins.UpdateModelMixin,
                  mixins.DestroyModelMixin,
                  generics.GenericService):
    queryset = Post.objects.all()
    serializer_class = PostProtoSerializer
```

We are building our service with `GenericService`, and adding in `ListModelMixin`,''CreateModelMixin'', etc. The base class provides the core functionality, and the mixin classes provice the `.List()` and `.Create()` handlers.

Using model service

If you want all operations of create/list/retrieve/update/destroy, we provide one already mixed-in generic services:

```
class PostService(generics.ModelService):
    queryset = Post.objects.all()
    serializer_class = PostProtoSerializer
```

1.3.3 Writing and running tests

Let's write some tests for our service and run them.

Writing tests

Let's edit the `blog/tests.py` file:

```
import grpc
from django_grpc_framework.test import RPCTestCase
from blog_proto import post_pb2, post_pb2_grpc
from blog.models import Post


class PostServiceTest(RPCTestCase):
    def test_create_post(self):
        stub = post_pb2_grpc.PostControllerStub(self.channel)
        response = stub.Create(post_pb2.Post(title='title', content='content'))
        self.assertEqual(response.title, 'title')
        self.assertEqual(response.content, 'content')
        self.assertEqual(Post.objects.count(), 1)

    def test_list_posts(self):
        Post.objects.create(title='title1', content='content1')
        Post.objects.create(title='title2', content='content2')
        stub = post_pb2_grpc.PostControllerStub(self.channel)
```

(continues on next page)

(continued from previous page)

```
        post_list = list(stub.List(post_pb2.PostListRequest()))
        self.assertEqual(len(post_list), 2)
```

Running tests

Once you've written tests, run them:

```
python manage.py test
```

1.4 Services

Django gRPC framework provides an `Service` class, which is pretty much the same as using a regular gRPC generated servicer interface. For example:

```
import grpc
from django_grpc_framework.services import Service
from blog.models import Post
from blog.serializers import PostProtoSerializer


class PostService(Service):
    def get_object(self, pk):
        try:
            return Post.objects.get(pk=pk)
        except Post.DoesNotExist:
            self.context.abort(grpc.StatusCode.NOT_FOUND, 'Post:%s not found!' % pk)

    def Retrieve(self, request, context):
        post = self.get_object(request.id)
        serializer = PostProtoSerializer(post)
        return serializer.message
```

1.4.1 Service instance attributes

The following attributes are available in a service instance.

- `.request` - the gRPC request object
- `.context` - the `grpc.ServicerContext` object
- `.action` - the name of the current service method

1.4.2 As servicer method

classmethod Service.**as_servicer**(***initkwargs*)

Returns a gRPC servicer instance:

```
servicer = PostService.as_servicer()
add_PostControllerServicer_to_server(servicer, server)
```

1.5 Generic services

The generic services provided by gRPC framework allow you to quickly build gRPC services that map closely to your database models. If the generic services don't suit your needs, use the regular `Service` class, or reuse the mixins and base classes used by the generic services to compose your own set of ressable generic services.

For example:

```
from blog.models import Post
from blog.serializers import PostProtoSerializer
from django_grpc_framework import generics


class PostService(generics.ModelService):
    queryset = Post.objects.all()
    serializer_class = PostProtoSerializer
```

1.5.1 GenericService

This class extends `Service` class, adding commonly required behavior for standard list and detail services. All concrete generic services is built by composing `GenericService`, with one or more mixin classes.

Attributes

Basic settings:

The following attributes control the basic service behavior:

- `queryset` - The queryset that should be used for returning objects from this service. You must set this or override the `get_queryset` method, you should call `get_queryset` instead of accessing this property directly, as `queryset` will get evaluated once, and those results will be cached for all subsequent requests.
- `serializer_class` - The serializer class that should be used for validating and deserializing input, and for serializing output. You must either set this attribute, or override the `get_serializer_class()` method.
- `lookup_field` - The model field that should be used to for performing object lookup of individual model instances. Defaults to primary key field name.
- `lookup_request_field` - The request field that should be used for object lookup. If unset this defaults to using the same value as `lookup_field`.

Methods

class django_grpc_framework.generics.**GenericService**(***kwargs*)

Base class for all other generic services.

filter_queryset(*queryset*)

Given a queryset, filter it, returning a new queryset.

get_object()

Returns an object instance that should be used for detail services. Defaults to using the lookup_field parameter to filter the base queryset.

get_queryset()

Get the list of items for this service. This must be an iterable, and may be a queryset. Defaults to using `self.queryset`.

If you are overriding a handler method, it is important that you call `get_queryset()` instead of accessing the `queryset` attribute as `queryset` will get evaluated only once.

Override this to provide dynamic behavior, for example:

```
def get_queryset(self):
    if self.action == 'ListSpecialUser':
        return SpecialUser.objects.all()
    return super().get_queryset()
```

get_serializer(**args*, ***kwargs*)
Return the serializer instance that should be used for validating and deserializing input, and for serializing output.

get_serializer_class()
Return the class to use for the serializer. Defaults to using *self.serializer_class*.

get_serializer_context()
Extra context provided to the serializer class. Defaults to including `grpc_request`, `grpc_context`, and `service` keys.

1.5.2 Mixins

The mixin classes provide the actions that are used to privide the basic service behavior. The mixin classes can be imported from `django_grpc_framework.mixins`.

class django_grpc_framework.mixins.**ListModelMixin**

List(*request*, *context*)
List a queryset. This sends a sequence of messages of `serializer.Meta.proto_class` to the client.

Note: This is a server streaming RPC.

---

class django_grpc_framework.mixins.**CreateModelMixin**

Create(*request*, *context*)
Create a model instance.

The request should be a proto message of `serializer.Meta.proto_class`. If an object is created this returns a proto message of `serializer.Meta.proto_class`.

perform_create(*serializer*)
Save a new object instance.

class django_grpc_framework.mixins.**RetrieveModelMixin**

Retrieve(*request*, *context*)
Retrieve a model instance.

The request have to include a field corresponding to `lookup_request_field`. If an object can be retrieved this returns a proto message of `serializer.Meta.proto_class`.

class django_grpc_framework.mixins.**UpdateModelMixin**

Update(*request*, *context*)
Update a model instance.

The request should be a proto message of `serializer.Meta.proto_class`. If an object is updated this returns a proto message of `serializer.Meta.proto_class`.

perform_update(*serializer*)
Save an existing object instance.

class django_grpc_framework.mixins.**DestroyModelMixin**

Destroy(*request*, *context*)
Destroy a model instance.

The request have to include a field corresponding to `lookup_request_field`. If an object is deleted this returns a proto message of `google.protobuf.empty_pb2.Empty`.

perform_destroy(*instance*)
Delete an object instance.

1.5.3 Concrete service classes

The following classes are the concrete generic services. They can be imported from `django_grpc_framework.generics`.

class django_grpc_framework.generics.**CreateService**(***kwargs*)
Concrete service for creating a model instance that provides a `Create()` handler.

class django_grpc_framework.generics.**ListService**(***kwargs*)
Concrete service for listing a queryset that provides a `List()` handler.

class django_grpc_framework.generics.**RetrieveService**(***kwargs*)
Concrete service for retrieving a model instance that provides a `Retrieve()` handler.

class django_grpc_framework.generics.**DestroyService**(***kwargs*)
Concrete service for deleting a model instance that provides a `Destroy()` handler.

class django_grpc_framework.generics.**UpdateService**(***kwargs*)
Concrete service for updating a model instance that provides a `Update()` handler.

class django_grpc_framework.generics.**ReadOnlyModelService**(***kwargs*)
Concrete service that provides default `List()` and `Retrieve()` handlers.

class django_grpc_framework.generics.**ModelService**(***kwargs*)
Concrete service that provides default `Create()`, `Retrieve()`, `Update()`, `Destroy()` and `List()` handlers.

You may need to provide custom classes that have certain actions, to create a base class that provides `List()` and `Create()` handlers, inherit from `GenericService` and mixin the required handlers:

```
from django_grpc_framework import mixins
from django_grpc_framework import generics

class ListCreateService(mixins.CreateModelMixin,
                        mixins.ListModelMixin,
                        GenericService):
    """
    Concrete service that provides ``Create()`` and ``List()`` handlers.
    """
    pass
```

1.6 Proto Serializers

The serializers work almost exactly the same with REST framework's `Serializer` class and `ModelSerializer`, but use `message` instead of `data` as input and output.

1.6.1 Declaring serializers

Declaring a serializer looks very similar to declaring a rest framework serializer:

```
from rest_framework import serializers
from django_grpc_framework import proto_serializers

class PersonProtoSerializer(proto_serializers.ProtoSerializer):
    name = serializers.CharField(max_length=100)
    email = serializers.EmailField(max_length=100)

    class Meta:
        proto_class = hrm_pb2.Person
```

1.6.2 Overriding serialization and deserialization behavior

A proto serializer is the same as one rest framework serializer, but we are adding the following logic:

- Protobuf message -> Dict of python primitive datatypes.
- Protobuf message <- Dict of python primitive datatypes.

If you need to alter the convert behavior of a serializer class, you can do so by overriding the `.message_to_data()` or `.data_to_message` methods.

Here is the default implementation:

```
from google.protobuf.json_format import MessageToDict, ParseDict

class ProtoSerializer(BaseProtoSerializer, Serializer):
    def message_to_data(self, message):
        """Protobuf message -> Dict of python primitive datatypes.
        """
        return MessageToDict(
            message, including_default_value_fields=True,
            preserving_proto_field_name=True
        )

    def data_to_message(self, data):
        """Protobuf message <- Dict of python primitive datatypes."""
        return ParseDict(
            data, self.Meta.proto_class(),
            ignore_unknown_fields=True
        )
```

The default behavior requires you to provide `ProtoSerializer.Meta.proto_class`, it is the protobuf class that should be used for create output proto message object. You must either set this attribute, or override the `data_to_message()` method.

1.6.3 Serializing objects

We can now use `PersonProtoSerializer` to serialize a person object:

```
>>> serializer = PersonProtoSerializer(person)
>>> serializer.message
name: "amy"
email: "amy@demo.com"
>>> type(serializer.message)
<class 'hrm_pb2.Person'>
```

1.6.4 Deserializing objects

Deserialization is similar:

```
>>> serializer = PersonProtoSerializer(message=message)
>>> serializer.is_valid()
True
>>> serializer.validated_data
OrderedDict([('name', 'amy'), ('email', 'amy@demo.com')])
```

1.6.5 ModelProtoSerializer

This is the same as a rest framework `ModelSerializer`:

```
from django_grpc_framework import proto_serializers
from hrm.models import Person
import hrm_pb2


class PersonProtoSerializer(proto_serializers.ModelProtoSerializer):
    class Meta:
        model = Person
        proto_class = hrm_pb2.Person
        fields = '__all__'
```

1.7 Proto

Django gRPC framework provides support for automatic generation of proto.

1.7.1 Generate proto for model

If you want to automatically generate proto definition based on a model, you can use the `generateproto` management command:

```
python manage.py generateproto --model django.contrib.auth.models.User
```

To specify fields and save it to a file, use:

```
python manage.py generateproto --model django.contrib.auth.models.User --fields id,
↪username,email --file demo.proto
```

Once you've generated a proto file in this way, you can edit it as you wish.

1.8 Server

1.8.1 grpcrunserver

Run a grpc server:

```
$ python manage.py grpcrunserver
```

Run a grpc development server, this tells Django to use the auto-reloader and run checks:

```
$ python manage.py grpcrunserver --dev
```

Run the server with a certain address:

```
$ python manage.py grpcrunserver 127.0.0.1:8000 --max-workers 5
```

1.8.2 Configuration

Root handlers hook

We need a hanlders hook function to add all servicers to the server, for example:

```
def grpc_handlers(server):
    demo_pb2_grpc.add_UserControllerServicer_to_server(UserService.as_servicer(),␣
↪server)
```

You can set the root handlers hook using the `ROOT_HANDLERS_HOOK` setting key, for example set the following in your `settings.py` file:

```
GRPC_FRAMEWORK = {
    ...
    'ROOT_HANDLERS_HOOK': 'path.to.your.curtom_grpc_handlers',
}
```

The default setting is `'{settings.ROOT_URLCONF}.grpc_handlers'`.

Setting the server interceptors

If you need to add server interceptors, you can do so by setting the

`SERVER_INTERCEPTORS` setting. For example, have something like this in your `settings.py` file:

```
GRPC_FRAMEWORK = {
    ...
    'SERVER_INTERCEPTORS': [
        'path.to.DoSomethingInterceptor',
        'path.to.DoAnotherThingInterceptor',
    ]
}
```

1.9 Testing

Django gRPC framework includes a few helper classes that come in handy when writing tests for services.

1.9.1 The test channel

The test channel is a Python class that acts as a dummy gRPC channel, allowing you to test you services. You can simulate gRPC requests on a service method and get the response. Here is a quick example, let's open Django shell `python manage.py shell`:

```
>>> from django_grpc_framework.test import Channel
>>> channel = Channel()
>>> stub = post_pb2_grpc.PostControllerStub(channel)
>>> response = stub.Retrieve(post_pb2.PostRetrieveRequest(id=post_id))
>>> response.title
'This is a title'
```

1.9.2 RPC test cases

Django gRPC framework includes the following test case classes, that mirror the existing Django test case classes, but provide a test `Channel` instead of `Client`.

- `RPCSimpleTestCase`
- `RPCTransactionTestCase`
- `RPCTestCase`

You can use these test case classes as you would for the regular Django test case classes, the `self.channel` attribute will be an `Channel` instance:

```
from django_grpc_framework.test import RPCTestCase
from django.contrib.auth.models import User
import account_pb2
import account_pb2_grpc


class UserServiceTest(RPCTestCase):
    def test_create_user(self):
        stub = account_pb2_grpc.UserControllerStub(self.channel)
        response = stub.Create(account_pb2.User(username='tom', email='tom@account.com
↪'))
        self.assertEqual(response.username, 'tom')
        self.assertEqual(response.email, 'tom@account.com')
        self.assertEqual(User.objects.count(), 1)
```

1.10 Settings

Configuration for gRPC framework is all namespaced inside a single Django setting, named `GRPC_FRAMEWORK`, for example your project's `settings.py` file might look like this:

```
GRPC_FRAMEWORK = {
    'ROOT_HANDLERS_HOOK': 'project.urls.grpc_handlers',
}
```

1.10.1 Accessing settings

If you need to access the values of gRPC framework's settings in your project, you should use the `grpc_settings` object. For example:

```
from django_grpc_framework.settings import grpc_settings
print(grpc_settings.ROOT_HANDLERS_HOOK)
```

The `grpc_settings` object will check for any user-defined settings, and otherwise fall back to the default values. Any setting that uses string import paths to refer to a class will automatically import and return the referenced class, instead of the string literal.

1.10.2 Configuration values

ROOT_HANDLERS_HOOK

A hook function that takes gRPC server object as a single parameter and add all servicers to the server.

Default: `'{settings.ROOT_URLCONF}.grpc_handlers'`

One example for the hook function:

```
def grpc_handlers(server):
    demo_pb2_grpc.add_UserControllerServicer_to_server(UserService.as_servicer(),␣
↪server)
```

SERVER_INTERCEPTORS

An optional list of ServerInterceptor objects that observe and optionally manipulate the incoming RPCs before handing them over to handlers.

Default: `None`

1.11 Patterns for gRPC

This part contains some snippets and patterns for Django gRPC framework.

1.11.1 Handling Partial Update

In proto3:

1. All fields are optional
2. Singular primitive fields, repeated fields, and map fields are initialized with default values (0, empty list, etc). There's no way of telling whether a field was explicitly set to the default value (for example whether a boolean was set to false) or just not set at all.

If we want to do a partial update on resources, we need to know whether a field was set or not set at all. There are different strategies that can be used to represent `unset`, we'll use a pattern called `"Has Pattern"` here.

Singular field absence

In proto3, for singular field types, you can use the parent message's `HasField()` method to check if a message type field value has been set, but you can't do it with non-message singular types.

For primitive types if you need `HasField` to you could use `"google/protobuf/wrappers.proto"`. Wrappers are useful for places where you need to distinguish between the absence of a primitive typed field and its default value:

```
import "google/protobuf/wrappers.proto";

service PersonController {
    rpc PartialUpdate(PersonPartialUpdateRequest) returns (Person) {}
}

message Person {
    int32 id = 1;
    string name = 2;
    string email = 3;
}

message PersonPartialUpdateRequest {
    int32 id = 1;
    google.protobuf.StringValue name = 2;
    google.protobuf.StringValue email = 3;
}
```

Here is the client usage:

```
from google.protobuf.wrappers_pb2 import StringValue


with grpc.insecure_channel('localhost:50051') as channel:
    stub = hrm_pb2_grpc.PersonControllerStub(channel)
    request = hrm_pb2.PersonPartialUpdateRequest(id=1, name=StringValue(value="amy"))
    response = stub.PartialUpdate(request)
    print(response, end='')
```

The service implementation:

```
class PersonService(generics.GenericService):
    queryset = Person.objects.all()
    serializer_class = PersonProtoSerializer

    def PartialUpdate(self, request, context):
        instance = self.get_object()
        serializer = self.get_serializer(instance, message=request, partial=True)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return serializer.message
```

Or you can just use `PartialUpdateModelMixin` to get the same behavior:

```
class PersonService(mixins.PartialUpdateModelMixin,
                    generics.GenericService):
    queryset = Person.objects.all()
    serializer_class = PersonProtoSerializer
```

Repeated and map field absence

If you need to check whether repeated fields and map fields are set or not, you need to do it manually:

```
message PersonPartialUpdateRequest {
    int32 id = 1;
    google.protobuf.StringValue name = 2;
    google.protobuf.StringValue email = 3;
    repeated int32 groups = 4;
    bool is_groups_set = 5;
}
```

1.11.2 Null Support

In proto3, all fields are never null. However, we can use `Oneof` to define a nullable type, for example:

```
syntax = "proto3";

package snippets;

import "google/protobuf/struct.proto";

service SnippetController {
    rpc Update(Snippet) returns (Snippet) {}
}

message NullableString {
    oneof kind {
        string value = 1;
        google.protobuf.NullValue null = 2;
    }
}

message Snippet {
    int32 id = 1;
    string title = 2;
    NullableString language = 3;
}
```

The client example:

```
import grpc
import snippets_pb2
import snippets_pb2_grpc
from google.protobuf.struct_pb2 import NullValue


with grpc.insecure_channel('localhost:50051') as channel:
    stub = snippets_pb2_grpc.SnippetControllerStub(channel)
    request = snippets_pb2.Snippet(id=1, title='snippet title')
    # send non-null value
    # request.language.value = "python"
    # send null value
    request.language.null = NullValue.NULL_VALUE
    response = stub.Update(request)
    print(response, end='')
```

The service implementation:

```
from django_grpc_framework import generics, mixins
from django_grpc_framework import proto_serializers
from snippets.models import Snippet
import snippets_pb2
from google.protobuf.struct_pb2 import NullValue


class SnippetProtoSerializer(proto_serializers.ModelProtoSerializer):
    class Meta:
        model = Snippet
        fields = '__all__'

    def message_to_data(self, message):
        data = {
            'title': message.title,
        }
        if message.language.HasField('value'):
            data['language'] = message.language.value
        elif message.language.HasField('null'):
            data['language'] = None
        return data

    def data_to_message(self, data):
        message = snippets_pb2.Snippet(
            id=data['id'],
            title=data['title'],
        )
        if data['language'] is None:
            message.language.null = NullValue.NULL_VALUE
        else:
            message.language.value = data['language']
        return message


class SnippetService(mixins.UpdateModelMixin,
                     generics.GenericService):
    queryset = Snippet.objects.all()
    serializer_class = SnippetProtoSerializer
```

CHAPTER 2

Additional Stuff

Changelog and license here if you are interested.

2.1 Changelog

2.1.1 Version 0.2.1

- Fixed close_old_connections in test channel

2.1.2 Version 0.2

- Added test module
- Added proto serializers
- Added proto generators

2.1.3 Version 0.1

First public release.

2.2 License

This library is licensed under Apache License.

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute

patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

   You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for

loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

> To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright [yyyy] [name of copyright owner]

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

> http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Index