gRPCAlchemy Documentation

Release 0.7.3

GuangTian Li

Apr 23, 2021

Contents:

CHAPTER 1

gRPCAlchemy

The Python micro framework for building gPRC application based on official gRPC project.

- Free software: MIT license
- Documentation: https://grpcalchemy.readthedocs.io.

1.1 Installation

```
$ pipenv install grpcalchemy
```

Only **Python 3.6+** is supported.

1.2 Example

1.2.1 Server

```
from grpcalchemy.orm import Message, StringField
from grpcalchemy import Server, Context, grpcmethod

class HelloMessage(Message):
```

(continues on next page)

(continued from previous page)

```
    text: str

class HelloService(Server):
    @grpcmethod
    def Hello(self, request: HelloMessage, context: Context) -> HelloMessage:
        return HelloMessage(text=f'Hello {request.text}')

if __name__ == '__main__':
    HelloService.run()
```

Then Using gRPC channel to connect the server:

```
from grpc import insecure_channel

from protos.helloservice_pb2_grpc import HelloServiceStub
from protos.hellomessage_pb2 import HelloMessage

with insecure_channel("localhost:50051") as channel:
    response = HelloServiceStub(channel).Hello(
        HelloMessage(text="world")
    )
```

1.3 Features

- gPRC Service Support
- **gRPC Message Support**
  - Scalar Value Types
  - Message Types
  - Repeated Field
  - Maps
- Define Message With Type Hint
- Middleware
- App Context Manger
- Error Handler Support
- Streaming Method Support
- gRPC-Health Checking and Reflection Support (Alpha)
- Multiple Processor Support

1.4 TODO

- Test Client Support
- Async Server Support

CHAPTER 2

Installation

2.1 Stable release

To install gRPCAlchemy, run this command in your terminal:

```
$ pipenv install grpcalchemy
```

This is the preferred method to install gRPCAlchemy, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for gRPCAlchemy can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/GuangTianLi/grpcalchemy
```

Or download the tarball:

```
$ curl  -OL https://github.com/GuangTianLi/grpcalchemy/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Or using pipenv install straightly:

```
$ pipenv install -e git+https://github.com/GuangTianLi/grpcalchemy#egg=grpcalchemy
```

CHAPTER 3

Usage

To use gRPCAlchemy in a project:

```
from grpcalchemy.orm import Message
from grpcalchemy import Server, Context, grpcmethod

class HelloMessage(Message):
    __filename__ = 'hello'
    text: str

class HelloService(Server):
    @grpcmethod
    def Hello(self, request: HelloMessage, context: Context) -> HelloMessage:
        return HelloMessage(text=f'Hello {request.text}')

if __name__ == '__main__':
    HelloService.run()
```

3.1 Defining our Message

Any message which is used in RPC method must have a explicit schema. We can use py files generated by proto files by the grpc_tools straightly. However defining the schemas by our ORM can help to iron out bugs involving incorrect types or missing fields, and also allow us to define utility methods on our message in the same way that traditional ORMs do.

In our Tutorial Application we need to send several different types of message. We will need to have a collection of **users**, so that we may link posts to an individual. We also need to send our different types of **posts** (eg: text, image and link) in the RPC method. To aid navigation of our Tutorial Application, posts may have **tags** associated with them, so that the list of posts shown to the user may be limited to posts that have been assigned a specific tag. Finally, it would be nice if **comments** could be added to posts. We'll start with **users**, as the other document models are slightly more involved.

3.1.1 Users

Just as if we were using a RPC Message with an ORM, we need to define which fields a `User` may have, and what types of data they might have:

```
from grpcalchemy.orm import Message
class User(Message):
    email: str
    first_name: str
    last_name: str
```

3.1.2 Posts, Comments and Tags

Now we'll think about how to define the rest of the information. To associate the comments with individual posts, We'd also need a link message to provide the many-to-many relationship between posts and tags.

Posts

We can think of `Post` as a base class, and `TextPost`, `ImagePost` and `LinkPost` as subclasses of `Post`.

```
from grpcalchemy.orm import Message
class Post(Message):
    title: str
    author: str

class TextPost(Post):
    content: str

class ImagePost(Post):
    image_path: str

class LinkPost(Post):
    link_url: str
```

We are storing a reference to the author of the posts using a `ReferenceField` object. These are equal to use other message types in RPC message.

Tags

Now that we have our Post models figured out, how will we attach tags to them? RPC message allows us to define lists of items natively. So, for both efficiency and simplicity's sake, we'll define the tags as strings directly within the post. Let's take a look at the code of our modified `Post` class:

```
from typing import List
from grpcalchemy.orm import Message, Repeated
class Post(Message):
    title: str
    author: User
    tags: Repeated[str]
```

The `ListField` object that is used to define a Post's tags takes a field object as its first argument — this means that you can have lists of any type of field (including lists).

Note: We don't need to modify the specialized post types as they all inherit from `Post`.

Comments

A comment is typically associated with *one* post.utility methods, in exactly the same way we do with regular documents:

```
from grpcalchemy.orm import Message
class Comment(Message):
    content: str
    name: str
```

We can then define a list of comment documents in our post message:

```
from typing import List
from grpcalchemy.orm import Message, Repeated
class Post(Message):
    title: str
    author: User
    tags: Repeated[str]
    comments: Repeated[Comment]
```

3.2 Defining our gRPC Method

`grpcmethod` is a decorator indicating gRPC methods.

The `valid gRPC Method` must be with explicit type hint to define the type of request and return value.

Using Iterator to define Stream gRPC Method:

```
from typing import Iterator

class HelloService(Server):

    @grpcmethod
    def UnaryUnary(self, request: HelloMessage, context: Context) -> HelloMessage:
        return HelloMessage(text=f'Hello {request.text}')

    @grpcmethod
    def UnaryStream(self, request: HelloMessage, context: Context) ->␣
↪Iterator[HelloMessage]:
        yield HelloMessage(text=f'Hello {request.text}')

    @grpcmethod
    def StreamUnary(self, request: Iterator[HelloMessage], context: Context) ->␣
↪HelloMessage:
        for r in request:
            pass
        return HelloMessage(text=f'Hello {r.text}')

    @grpcmethod
    def StreamStream(self, request: Iterator[HelloMessage], context: Context) ->␣
↪Iterator[HelloMessage]:
```

(continues on next page)

(continued from previous page)

```
        for r in request:
            yield HelloMessage(text=f'Hello {r.text}')
```

The above code is equal to an RPC service with a method:

```
syntax = "proto3";

service HelloService {
    rpc StreamStream (stream HelloMessage) returns (stream HelloMessage) {}

    rpc StreamUnary (stream HelloMessage) returns (HelloMessage) {}

    rpc UnaryStream (HelloMessage) returns (stream HelloMessage) {}

    rpc UnaryUnary (HelloMessage) returns (HelloMessage) {}
}
```

3.3 Using Blueprint to Build Your Large Application

gRPCAlchemy uses a concept of blueprints for making gRPC services and supporting common patterns within an application or across applications. `Blueprint` can greatly simplify how large applications work.

```
from typing import List, Type
from grpcalchemy.orm import Message
from grpcalchemy import Server, Context, Blueprint

class MyService(Server):
    @classmethod
    def get_blueprints(cls) -> List[Type[Blueprint]]:
        return [HelloService]

class HelloMessage(Message):
    __filename__ = 'hello'
    text: str

class HelloService(Blueprint):
    @grpcmethod
    def Hello(self, request: HelloMessage, context: Context) -> HelloMessage:
        return HelloMessage(text=f'Hello {request.text}')


if __name__ == '__main__':
    MyService.run()
```

3.4 Configuration

You can define your custom config by inherit from `DefaultConfig` which defined a list of configuration available in gRPCAlchemy and their default values.

Note: DefaultConfig is defined by *configalchemy* - https://configalchemy.readthedocs.io

```
from grpcalchemy import DefaultConfig

from hello import HelloService

class MyConfig(DefaultConfig):
    ...

config = MyConfig()

HelloService.run(config=config)
```

3.5 Middleware

Middleware is a framework of hooks into gRPCAlchemy's request/response processing.

Costume middleware can implement by overriding `Blueprint.before_request`, `Blueprint.after_request`, `Server.process_request` and `Server.process_response`.

CHAPTER 4

grpcalchemy

4.1 grpcalchemy package

4.1.1 grpcalchemy.server module

4.1.2 grpcalchemy.blueprint module

4.1.3 grpcalchemy.config module

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/GuangTianLi/grpcalchemy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

5.1.4 Write Documentation

gRPCAlchemy could always use more documentation, whether as part of the official gRPCAlchemy docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/GuangTianLi/grpcalchemy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *grpcalchemy* for local development.

1. Fork the *grpcalchemy* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/grpcalchemy.git
```

3. Install your local copy into a virtualenv. Assuming you have Pipenv installed, this is how you set up your fork for local development:

```
$ cd grpcalchemy/
$ make init
$ pipenv shell
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass lint and the tests, including testing other Python versions:

```
$ make lint
$ make test
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6+. Check https://travis-ci.org/GuangTianLi/grpcalchemy/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ make test
```

5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 6

Credits

6.1 Development Lead

- GuangTian Li <guangtian_li@qq.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

History

7.1 0.7.*(2021-03-20)

- Improve initialize function of message
- Remove default feature in message
- Refactor composite message Type
- Support gRPC with xDS
- Add *PROTO_AUTO_GENERATED* setting to make runtime proto generation optional

7.2 0.6.*(2020-10-27)

- fix [#36] compatibility in windows
- fix [#34] compatibility in windows
- gRPC-Health Checking and Reflection Support (Alpha)
- Multiple Processor Support

7.3 0.5.0(2020-04-27)

- Support Streaming Method
- Deprecate request parameter in app context and handle exception

7.4 0.4.0(2019-09-24)

- Support related directory path to generate protocol buffer files
- Enable use type hint to define message
- Add error handle to handle Exception
- Add `get_blueprints` to get blueprints need to register

7.5 0.3.0(2019-08-19)

https://github.com/GuangTianLi/grpcalchemy/projects/1

7.6 0.2.7-10(2019-04-16)

- Support SSL
- Improve Implement of Server with grpc.server
- Support YAML file in Config Module
- Improve Config Module
- Add context in current rpc

7.7 0.2.5-6(2019-03-06)

- Implement Rpc Context
- Improve Config Module

7.8 0.2.4(2019-03-01)

- Implement Globals Variable
- Implement APP Context

7.9 0.2.2-3 (2019-02-26)

- Improve Config module
- Improve rpc_call_wrap

7.10 0.2.1 (2019-02-14)

- Implement Own gRPC Server
- Implement gRPC Server Test Client

7.11 0.2.0 (2019-01-30)

- Change gRPCAlchemy Server register to register_blueprint
- Make gRPCAlchemy Server inherit from Blueprint
- Support Json Format
- Support Inheritance Message

7.12 0.1.6 (2019-01-21)

- Various bug-fixes
- Improve tests
- Change Client API
- Add PreProcess And PostProcess
- Import Config Object
- Add Event Listener
- Change Field Object Into Descriptor

7.13 0.1.5 (2018-12-14)

- Various bug-fixes
- Improve tests
- Add client

7.14 0.1.4 (2018-12-11)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search