archer Documentation

Release 0.1

Wang Haowei

October 29, 2016

Contents

CHAPTER 1

Archer: develop Thrift RPC service the Flask way

Welcome to Archer's documentation. This documentation is divided into different parts. I recommend that you get started with *Installation* and then head over to the *Quickstart*. Besides the quickstart, there is also a more detailed *tutorial* that shows how to create a complete (albeit small) application with Archer. If you'd rather dive into the internals of Archer, check out the *API* documentation.

Archer depends on two external libraries: the Thriftpy interpreter engine and the Click cli parser. These libraries are not documented here. If you want to dive into their documentation, check out the following links:

- Thriftpy Documentation
- Click Documentation

CHAPTER 2

User's Guide

This part of the documentation, which is mostly prose, begins with some background information about Archer, then focuses on step-by-step instructions for web development with Archer.

2.1 foreword

2.1.1 Why Thrift

The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, JavaScript, Node.js, Smalltalk, OCaml and Delphi and other languages.

You just wrie a thrift file:

```
thrift --gen <language> <Thrift filename>
```

after compiling for a given language, the corresponding SDK files are generated.

2.1.2 Why Thriftpy

Thriftpy is a Python implementation of Thrift which generates SDK modules dynamically when some thrift file is loaded, No SDK files any more, making development procedure more fluently.

2.2 Installation

Archer depends on some external libraries, like Thriftpy and Click. Thriftpy is an Thrift interface definition language interpreter written in Python which would load a thrift file and generate SDK module on the fly, saving you the time for compiling the thrift file by hand.

Click is a Python package for creating beautiful command line interfaces in a composable way with as little code as necessary.

So how do you get all that on your computer quickly? There are many ways you could do that, but the most kick-ass method is virtualenv, so let's have a look at that first.

You will need Python 2.6 or newer to get started, so be sure to have an up-to-date Python 2.x installation. Python 3.x would also be OK.

2.2.1 virtualenv

Virtualenv is probably what you want to use during development, and if you have shell access to your production machines, you'll probably want to use it there, too.

What problem does virtualenv solve? If you like Python as much as I do, chances are you want to use it for other projects besides Archer-based RPC applications. But the more projects you have, the more likely it is that you will be working with different versions of Python itself, or at least different versions of Python libraries. Let's face it: quite often libraries break backwards compatibility, and it's unlikely that any serious application will have zero dependencies. So what do you do if two or more of your projects have conflicting dependencies?

Virtualenv to the rescue! Virtualenv enables multiple side-by-side installations of Python, one for each project. It doesn't actually install separate copies of Python, but it does provide a clever way to keep different project environments isolated. Let's see how virtualenv works.

If you are on Mac OS X or Linux, chances are that one of the following two commands will work for you:

```
$ sudo easy_install virtualenv
```

or even better:

```
$ sudo pip install virtualenv
```

One of these will probably install virtualenv on your system. Maybe it's even in your package manager. If you use Ubuntu, try:

```
$ sudo apt-get install python-virtualenv
```

Once you have virtualenv installed, just fire up a shell and create your own environment. I usually create a project folder and a `venv` folder within:

```
$ mkdir myproject
$ cd myproject
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip.............done.
```

Now, whenever you want to work on a project, you only have to activate the corresponding environment. On OS X and Linux, do the following:

```
$ . venv/bin/activate
```

If you are a Windows user, the following command is for you:

```
$ venv\scripts\activate
```

Either way, you should now be using your virtualenv (notice how the prompt of your shell has changed to show the active environment).

And if you want to go back to the real world, use the following command:

```
$ deactivate
```

After doing this, the prompt of your shell should be as familiar as before.

Now, let's move on. Enter the following command to get Archer activated in your virtualenv:

```
$ pip install Archer
```

A few seconds later and you are good to go.

2.2.2 System-Wide Installation

This is possible as well, though I do not recommend it. Just run **pip** with root privileges:

```
$ sudo pip install Archer
```

(On Windows systems, run it in a command-prompt window with administrator privileges, and leave out **sudo**.)

2.2.3 Living on the Edge

If you want to work with the latest version of Archer, there are two ways: you can either let **pip** pull in the development version, or you can tell it to operate on a git checkout. Either way, virtualenv is recommended.

Get the git checkout in a new virtualenv and run in development mode:

```
$ git clone http://github.com/eleme/archer.git
Initialized empty Git repository in ~/dev/archer/.git/
$ cd archer
$ virtualenv venv
New python executable in venv/bin/python
Installing setuptools, pip.............done.
$ . venv/bin/activate
$ python setup.py develop
...
Finished processing dependencies for Archer
```

This will pull in the dependencies and activate the git head as the current version inside the virtualenv. Then all you have to do is run `git pull origin` to update to the latest version.

2.3 Quickstart

This page gives a good introduction to Archer. It assumes you already have Archer installed. If you do not, head over to the *Installation* section.

2.3.1 A Minimal Application

A minimal Archer application looks something like this:

```
from archer import Archer
app = Archer('PingPong')

@app.api('ping')
def ping():
    return 'pong'
```

provided a hello.thrift file under your working directory, define a service in the file:

```
service PingPong {
    string ping(),
}
```

Note: `Archer` accepts one parameter, that is the name of the service defined in the thrift file.

So what did that code do?

1. First we imported the `Archer` class. An instance of this class will be our Thrift RPC server_side application.
2. Next we create an instance of this class. The first argument is the name of the application
3. We then use the `api()` decorator to tell Archer what name defined in thrift file should trigger our function.
4. The function returns the message for `ping` RPC call, which is a string `pong` here
5. the service PingPong is defined in the hello.thrift file

Just save the python code as `hello.py` (or something similar) and the service definition in `hello.thrift`, run it with your Python interpreter. Make sure to not call your application `archer.py` because this would conflict with Archer itself.

To run the application you can use the **`archer`** command:

```
$ archer run
* Running on 127.0.0.1:6000 in DEBUG mode
```

This launches a very simple built_in server, which is good enough for testing but probably not what you want to use in production. For deployment options see *deployment*.

Now run call the remote function you can also use the **`archer`** command:

```
$ archer call ping
* pong
```

You should see that the string *pong* is returned

Externally Visible Server

you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
archer run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

2.4 tutorial

2.4.1 tutorial

A minimal Archer application looks something like this:

```
from archer import Archer
app = Archer('PingPong')

@app.api('ping')
def ping():
    return 'pong'
```

provided a hello.thrift file under your working directory, define a service in the file:

```
service PingPong {
    string ping(),
}
```

So what did that code do?

1. First we imported the `Archer` class. An instance of this class will be our Thrift RPC server_side application.

2. Next we create an instance of this class. The first argument is the name of the application
3. We then use the `api()` decorator to tell Archer what name defined in thrift file should trigger our function.
4. The function returns the message for `ping` RPC call, which is a string `pong` here
5. the service PingPong is defined in the hello.thrift file

Just save the python code as `hello.py` (or something similar) and the service definition in `hello.thrift`, run it with your Python interpreter. Make sure to not call your application `archer.py` because this would conflict with Archer itself.

To run the application you can use the **`archer`** command:

```
$ archer run
* Running on 127.0.0.1:6000 in DEBUG mode
```

This launches a very simple built_in server, which is good enough for testing but probably not what you want to use in production. For deployment options see *deployment*.

Now run call the remote function you can also use the **`archer`** command:

```
$ archer call ping
* pong
```

You should see that the string *pong* is returned

You can also run a client shell by:

```
$ archer client
>>> client.ping()
```

Externally Visible Server

you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
archer run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

2.5 testing

2.5.1 test client

A test client could be initialized by calling `test_client()`

2.5.2 fake client

A test client could be initialized by calling `fake_client()`

2.6 errorhandling

2.6.1 register error handler

Exceptions occurs during an API calling would be caught by the error handlers registered. if no one is provided, the default handler would catch it. In Archer,an error handler is registed like:

```
app.register_error_handler(error, handler)
```

In which,e is the Exception,f is the error handler,for example:

```
class BasicException(Exception):
    pass

def BasicErrorHandler(meta, result):
    return 'BasicException'

app.register_error_handler(BasicException, BasicErrorHandler)
```

The two arguments meta and result refers to `ApiMeta` and `ApiResultMeta`.

Whenever a BasicException occurs,Archer will catch it and call BasicErrorHandler to handle it.

2.7 event

2.7.1 event

Event is used to add customized hook functions which would be called before or after An api calling.

Archer provides 3 events `before_api_call`, `after_api_call`, `tear_down_api_call`

for *after_api_call*,it would take one argument, which is an instance of `ApiMeta`

it receives two arguments, first is instance of `ApiMeta` and the second is instance of `ApiResultMeta`.

2.8 Command line tools

It's quite easy to fire up a development server by using the `archer` command line utility with `Archer.run()` method.

2.8.1 Command Line

The **`archer`** command line script (*Command line tools*) is strongly recommended for development because it provides a superior reload experience due to how it loads the application. The basic usage is like this:

```
$ archer --app my_application  run
```

This will enable the reloader and then start the server on *http://localhost:6000/*.

If you put your app instance in a python file or a `__init__.py` file in some directory under the `root_path`, archer will find the app for you automatically. In such cases, just:

```
$ archer run
```

And the server would start the same way. Super easy!

2.8.2 In Code

The alternative way to start the application is through the `Archer.run()` method. This will immediately launch a local server exactly the same way the **`archer`** script does.

Example:

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

2.8.3 client

Archer also provide the **`archer call`** to easy test a api without any coding:

```
$ archer --app my_application  call api_name param1,param2....
```

if you'd like archer find the app for you, just:

```
$ archer call api_name param1, param2...
```

And if everything is ok, the terminal would echo the return value of this api, or just the string `OK` if nothing is returned.

parameters are just separated by comma or whitespace, so `a b c d` and `a,b,c,d` are both ok.

Archer would handle the parameter type, so *123* would convert to int type. You can specify the type using `:` after a parameter, like `123:string`, so that Archer would known that you want 123 to be a string instead of int.

Non built_in type

Customized types are not supported, as **`call`** command is just for quickly getting some feedback of an api, You need more complicated test cases to ensure your api work correctly, so don't rely heavily on this command.

2.9 Working with the Shell

One of the reasons everybody loves Python is the interactive shell. It basically allows you to execute Python commands in real time and immediately get results back. Archer itself does not come with an interactive shell, because it does not require any specific setup upfront, just import your application and start playing around.

There are however some handy helpers to make playing around in the shell a more pleasant experience. The main issue with interactive console sessions is that you're not really triggering a real rpc call from a client.

This is where some helper functions come in handy. Keep in mind however that these functions are not only there for interactive shell usage, but also for unit testing and other situations that require a faked request context.

2.9.1 Command Line Interface

Thee recommended way to work with the shell is the `archer shell` command which does a lot of this automatically for you. For instance the shell is automatically initialized with a loaded application context. with globals `app`, `fake_client`, `test_client` already set at your hand:

```
>>> thrift_file = app.thrift_file
>>> test_client.ping()
>>> fake_client.ping()
```

You may want to add some other variables to the global scope of the shell by using `shell_context_processor()` method.

For more information see *Command line tools*.

2.10 deployment

2.10.1 deployment

The development Server is not suitable for production use, we did not handle the thrift Protocol and Transport details instead of providing a default one which is suitable for development.

gunicorn_thrift is highly recommended if you'd like to deploy an Archer application, as Archer is designed to work with `gunicorn_thrift`.

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 API

This part of the documentation covers all the interfaces of Archer. For parts where Archer depends on external libraries, we document the most important right here and provide links to the canonical documentation.

3.1.1 Application Object

3.1.2 ApiMeta Object

3.1.3 ApiResultMeta

3.1.4 Test Client

3.1.5 Fake Client

3.1.6 Command Line Interface

archer.cli.**call**

archer.cli.**shell**

archer.cli.**run**

CHAPTER 4

Additional Notes

Design notes, legal information and changelog are here for the interested.

4.1 Design Decisions in Archer

If you are curious why Archer does certain things the way it does and not differently, this section is for you. This should give you an idea about some of the design decisions that may appear arbitrary and surprising at first, especially in direct comparison with other frameworks.

4.1.1 The Explicit Application Object

A thrift application based on gunicorn_thrift has to have one central object that implements the actual application. In Archer this is an instance of the `Archer` class. Each Archer application has to create an instance of this class itself.

That instance is your gunicorn_thrift application, you don't have to remember anything else. If you want to apply a gunicorn_thrift middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object `processor()`).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unittesting and similar things

4.1.2 Thread Locals

Unlike Flask, Archer doesn't use thread local objects, no magic globals like *current_app*, *request* in Flask. We believe that things you can do with thread locals would exist a better way to do without it, and decouple your code with these globals means it would be easier to test and analyse, passing globals around everywhere seems not a good idea. Try to fire up a python shell and type `import this`, one thing you can find is:

```
"Explicit is better than implicit." --zen of Python
```

Say if we have an asynchronous server instead, using thread locals as globals would make no sense and break our application. Any way, no thread locals leaves the door open. We throw the ball to the end user to decide whether thread local would be used in an archer app, for more information , refer to the article GlobalState.

4.1.3 What Archer is, What Archer is Not

Archer will never have a database layer. It will not have a form library or anything else in that direction. Archer itself just bridges to gunicorn_thrift to implement a proper thrift application. It also binds to a few common standard library

packages such as logging. Everything else is up for extensions.

Archer almost does nothing on the client side , as the client language is depend on what you prefer, and how to use the connection is also not predictable. So just implement the client side code the way you like or just whatever to satisfy your need.

Why is this the case? Because people have different preferences and requirements and Archer could not meet those if it would force any of this into the core.

The idea of Archer is to build a good foundation for all thrift applications. Everything else is up to you.

4.2 changes

4.2.1 changes

Archer Changelog

Here you can see the full list of changes between each Archer release.

4.2.2 Version 0.1

First public preview release.

4.2.3 Version 0.2

Add client tools

4.3 licence

4.3.1 licence

This is the MIT license: http://www.opensource.org/licenses/mit-license.php

a

A

C

R

S