rio Documentation

Release 0.1.0

Ju Lin

May 05, 2016

Contents

Rio is a simple, scalable and reliable distributed event-driven system.

Rio receives actions via RESTful APIs and then triggers a bunch of subscribed webhooks simultaneously and asynchronously.

Rio is still working in progress. Rio is a thin wrapper of Celery and Flask, allowing you to use multiple kinds of broker to run async webhooks, such as RabbitMQ, Redis, ZeroMQ, SQLAlchemy, etc. It is Open Source and licensed under BSD License.

Contents:

CHAPTER 1

Quick Start

Rio is designed to be out-of-the-box, yet powerful to extend. If you never have experience in using Rio before, this tutorial will help you getting started.

Getting started with Rio is a three steps process:

- *Installing the Rio server*
- *Configure an Integration*
- *Configure The DSN*

1.1 Installing the Rio server

For more details about how to install the Rio server, see *Installation*.

Basically, you need a Unix based OS, Python 2.7. You can use a database as broker and backend, or use redis as broker and backend, or mix using RabbitMQ as broker and database as backend. It's all up to you.

1.2 Configure an Integration

To send messages to Rio you will need to use an SDK which support your platform. If you can not find platform listed below, you can simply use the JSON APIs to send messages.

Below is a list of Rio SDKs:

- Python SDK

1.3 Configure The DSN

After you have created a project and a sender in Rio, you will be given a DSN value. This is basically similar to Sentry DSN. It is a standard URL and a configuration parameter for Rio clients. The DSN can be found in Rio by navigation to project settings.

CHAPTER 2

Introduction

In computer programming, event-driven programming is a programming paradigm in which the flow the the program is determined by events. Event-driven programming is widely used in GUI development. Since microservice architecture is thriving these year, a demand to send messages from one system to many other systems is rising. Rio is a system trying to solve this problem.

Rio is standing on giants' shoulders: Flask + Celery. In Rio, there are a job queue playing the role of main loop, and once message sent to job queue, a bunch of HTTP webhooks will be triggered simultaneously. Logging and monitoring are important task as well in Rio. You can easily find out latest bad behaviour webhook calling and retrigger it if possible.

Communication between services is a tough problem for developers. There are two popular paradigm to complete asynchronous lightweight messaging tasks: Choreography and Orchestration. And Rio has a flavour of Choreography. As producer of the message doesn't have to know what other service supposed to do, it just provides an event, to which consumers may respond or no. On the other hand, as consumer of the message doesn't have to keep listening on message queue, it just provides an handler, to which consumer may be called or no. As a result, both two kinds of system need only behave theirselves.

It is recommended to put webhook under firewall protection so that villainous cracker have no opportunity to attack.

2.1 Example Usage

Rio assumes you have a sender service with SDK integrated, and some receiver services which implement HTTP callback tasks.

In Rio, you need to create a project first to receive message and traffic payload. Before sending message, you have to create handlers for an event in project. These operation can be done via CLI tools or Dashboard.

In sender side, you need to send message:

```
from rio_client import Client
client = Client(dsn='http://sender:*********@rio.intra.example.org/project')
client.emit('comment-published', {'ip': 127.0.0.1, 'content': 'I am a spammer'})
```

In receiver side, you need to define a simple webhook. For instance, this is a Flask view function:

```
@app.route('/webhook/comment/antispam', methods=['POST])
def antispam_comment():
    if is_spam_content(request.form['content']):
        ban_ip(request.form['ip])
    return jsonify(status='success', retval=0)
```

Or in Ruby on Rails:

```
def antispam_comment
    ban_ip(params[:ip]) if is_spam_content(params[:content])
    render :json => {:status => 'success', :retval => 0}
```

CHAPTER 3

Installation

This guide will step you through setting up a Python-based virtualenv, installing the required packages, and configuring the basic web service.

3.1 Dependencies

Some basic prerequisites which you'll need in order to run Rio:

- A UNIX-based operating system.
- Python 2.7
- python-setuptools, python-pip, python-dev, libffi-dev, libssl-dev, libyaml-dev
- A broker. It might be one of RabbitMQ(recommend), Redis(recommend), MongoDB, ZeroMQ, CouchDB, SQLAlchemy, Django ORM, Amazon SQS, and more..
- A result store. It might be one of AMQP, Redis, memcached, MongoDB, SQLAlchemy, Django ORM, Cassandra
- Nginx (nginx-full)
- A dedicated domain to host Rio on (i.e. rio.your-corp.com).

If you're building from source you'll also need:

Node.js 4 or newer.

3.2 Setting up an Environment

The first thing you'll need is the Python *virtualenv* package. You probably already have this, but if not, you can install it with:

```
$ pip install -U virtualenv
```

It's also available as python-virtualenv on ubuntu in the package manager.

Once that's done, choose a location for the environment, and create it with the virtualenv command. For our guide, we're going to choose */var/www/rio*:

```
$ mkdir /var/www/rio
$ virtualenv --distribute /var/www/rio
```

Finally, activate your virtualenv:

```
$ source /www/rio/bin/activate
```

3.3 Install Rio

Once you've got the environment setup, you can install Rio and all its dependencies with the same command you used to grab virtualenv:

```
$ pip install -U rio
```

To check installation successfully, run Rio CLI, via *rio*:

```
$ rio --help
```

3.4 Installing from Source

If you are going to install from source, you will need to install *npm*. Once your system is prepared, symlink your source into the virtualenv:

```
$ python setup.py develop
```

3.5 Initializing the Configuration

To create default configuration, you will use the *init* subcommand of *rio*. You can specify an alternative configuration path as the argument to init, otherwise it will use the default of current directory:

```
$ rio init /etc/rio
```

Set *RIO_CONF* as an environment variable so that rio can find this directory later:

```
$ export RIO_CONF=/etc/rio
```

The *init* subcommand create a config.py. Use your flavoured text editor to edit *config.py* file to adjust to your infrastructure.

You need to configure:

- configure_broker
- configure_storage_backend

3.6 Running Migrations

Rio provides an easy way to run migrations on the database on version upgrades. Before running it for the first time you'll need to make sure you've created the database:

```
mysql> CREATE DATABASE rio;
```

Once done, you can create the initial schema using the upgrade command:

```
$ rio upgrade
```

3.7 Starting the Web Service

Rio provides a built-in webserver (powered by Gunicorn) to get you off the ground quickly. You can also setup Rio as WSGI application by specifying wsgi application *rio.app:app*. To start the built-in webserver run *rio start*:

```
$ rio start web
```

You should now be able to test the web service by visiting http://localhost:8009/.

3.8 Starting Background Workers

A large amount of Rio's work is managed via background workers. These need run in addition to the web service workers:

```
$ rio start worker
```

3.9 Process Management

It is recommended to using process management software to keep Rio processes alive. *supervisor* is a fancy tool to archive that. This is an example of supervisor config part:

```
[program:rio-web]
directory=/www/rio/
environment=RIO_CONF="/etc/rio"
command=/www/rio/bin/rio start web
autostart=true
autorestart=true
redirect_stderr=true
stdout_logfile=syslog
stderr_logfile=syslog

[program:rio-worker]
directory=/www/rio/
environment=RIO_CONF="/etc/rio"
command=/www/rio/bin/sentry start worker
autostart=true
autorestart=true
redirect_stderr=true
stdout_logfile=syslog
stderr_logfile=syslog
```

3.10 Setup a Reverse Proxy

You'll use the builtin HttpProxyModule within Nginx to handle proxying:

```
upstream rio_servers {
    server    127.0.0.1:9001;
}

server {
    listen 80;
    server_name rio.intra.yourcorp.com;

    location / {
        client_max_body_size 10M;
        proxy_redirect     off;
        proxy_set_header   Host             $host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;
        proxy_pass         http://rio_servers;
    }
}
```

3.11 Removing Old Data

One of the most important things you're going to need to be aware of is storage costs. The stale data in Backend storage should be automatically removed by a cron job:

```
$ crontab -e
0 0 * * * RIO_CONF=/etc/rio rio cleanup --days=30
```

CHAPTER 4

Upgrading

4.1 Upgrading the Package

$ pip install -U rio

4.2 Running migrations

$ rio upgrade

4.3 Restarting Services

$ supervisorctl restart rio-web $ supervisorctl restart rio-worker

CHAPTER 5

Configurations

5.1 Must set configuration items

5.1.1 SECRET_KEY

the secret key.

DO NOT LEAK IT.

5.1.2 GRAPH_BACKEND

This item specifies the graph backend. Choices:

- *directory*, default
- *sqlalchemy*

5.1.3 SQLALCHEMY_DATABASE_URI

This item specifies the database. See more at

5.1.4 CELERY_BROKER_URL

This item specifies the broker. See http://celery.readthedocs.org/en/latest/configuration.html#broker-url

5.1.5 CELERY_RESULT_BACKEND

This item specifies the result backend. See http://celery.readthedocs.org/en/latest/configuration.html#database-backend-settings

CHAPTER 6

Webhook

6.1 Setting up a Webhook

6.2 Callback URL

This is the server endpoint that will receive the webhook payload. You can set your webhook callback URL in dashboard.

6.3 Content-Type

Webhooks can be delivered using different content types. Currently, Rio support two basic ways to send data:

- The *application/json* content type will deliver the JSON payload directly as the body of the POST.
- The *application/www-form-urlencoded* content type will send the JSON payload as a form parameter called "payload".

The default content type of *application/www-form-urlencoded*. The content type depends on how you set your webhook *Content-Type* in Webhook headers. Choose the one that best fits your needs.

6.4 Securing your webhooks

Once your server is configured to receive payloads, it will listen for any payload sent to the endpoint you configured. For security reasons, you probably want to limit requests to those coming from Rio. There are a few ways to go about this. For example, you could opt to whitelist requets from Rio's IP address. But a far easier method is to set up a secret token and validate the information.

6.4.1 Setting your secret token

6.4.2 Validating payloads from Rio

CHAPTER 7

Broker

CHAPTER 8

Storage Backend

CHAPTER 9

Worker

CHAPTER 10

Command Line Interface

Rio is cross-platform event driven system built with love.

10.1 Subcommands

- celery
- db
- init
- shell
- start
- runserver

CHAPTER 11

Monitoring

11.1 Health Check

Rio provides several ways to monitor the system status. This may be as simple as "is the backend serving requests" to more in-depth and gauging potential configuration problems. In some cases these checks will be exposed in the UI though generally only to superusers.

The following endpoint is exposed to aid in automated reporting:

```
http://rio.example.com/_health/
```

Generally this is most useful if you're using it as a health check in something like HAProxy.

In HAProxy, you could add this to your config:

```
option httpchk /_health/
```

That said, we also expose additional checks via the same endpoint by passing ?full:

```
$ curl -i http://rio.example.com/_health/?full
HTTP/1.0 500 INTERNAL SERVER ERROR
Content-Type: application/json

{
  "problems":  [
    "Background workers haven't checked in recently. This can mean an issue
     with your configuration or a serious backlog in tasks."
  ]
}
```

11.2 Queue Monitoring

CHAPTER 12

Logging

Sometimes you might want to dive into Rio to find out the data whether it is right or wrong. Python's standard logging module is used to implement informational and debug log output with Rio. You can integrate Rio's logging in a standard way with other libraries and applications.

There are several loggers listed below that can be turned on:

- *rio.tasks* - controls asynchronous tasks running logging. set to *logging.INFO* for requesting webhook, *logging.DEBUG* for requesting webhook and receiving webhook's response, *logging.ERROR* for error response.
- *rio.event* - controls event emitting logging. set to *logging.INFO* for receiving action.

For example, you can writing logging configure codes in config file:

```
import logging

logger = logging.getLogger('rio.tasks')
logger.addHandler(logging.FileHandler('/tmp/rio.log'))
logger.setLevel(logging.DEBUG)

logger = logging.getLogger('rio.event')
logger.addHandler(logging.FileHandler('/tmp/rio.log'))
logger.setLevel(logging.DEBUG)
```

Once an action was emitted, Rio would apply logging into your handlers:

```
$ curl http://example:example@127.0.0.1:5000/event/example/emit/example -X POST
{
  "event": {
    "uuid": "2df0b14b-07b9-42ab-9595-59a58829d505"
  },
  "message": "ok",
  "task": {
    "id": "f1c10766-b428-4ac7-ac0b-6bf2b4420d15"
  }
}

$ cat /tmp/rio.log
EMIT 2df0b14b-07b9-42ab-9595-59a58829d505 "example" "example" {}
REQUEST 2df0b14b-07b9-42ab-9595-59a58829d505 POST http://127.0.0.1:5000 {}
RESPONSE 2df0b14b-07b9-42ab-9595-59a58829d505 POST http://127.0.0.1:5000 {"message": "OK"}
```

CHAPTER 13

API References

CHAPTER 14

Indices and tables

- genindex
- modindex
- search