Flask-Kadabra Documentation

Release 0.1.0

Alex Landau

December 13, 2016

Contents

How quickly can you figure out how many server errors your Flask app is throwing? Can you determine which of your routes have the highest error rates? Do you know how long, on average, your SQLAlchemy writes take?

Flask-Kadabra extends the capabilities of the Kadabra metrics library to Flask:

- Enable metrics for your routes with a simple decorator.
- Record metrics from anywhere in your application code, organized by your routes.
- Automatically track basic metrics per route such as timing and errors.

Installation is simple:

```
$ pip install Flask-Kadabra
```

Setup is easy:

```
from flask import Flask
from flask_kadabra import Kadabra, record_metrics

app = Flask()
kadabra = Kadabra(app)

@app.route('/')
@record_metrics
def index():
    return "Hello, world!"
```

You can record whatever metrics you want from anywhere in your application code. They will all be grouped under the route you decorated, and recorded at the end of your request, with no performance impact:

```
from flask import g

g.metrics.add_count("userSignup", 1.0)
```

All you have to do is run a local Redis server and run the Kadabra agent side-by-side with your Flask app, and you have metrics!

If you're ready to start, head over to Installation, then Usage.

If you want to jump right into the advanced stuff, you might want to check out Deploying Your Stack.

CHAPTER 1

Contents:

1.1 Installation

You can install Flask-Kadabra through the Python package index (pip). Installation is simple:

```
$ sudo pip install Flask-Kadabra
```

After installing, you should check out Usage.

1.2 Usage

Using Flask-Kadabra is quite simple. After all, the goal is to enable you to record metrics from your Flask application with minimal additional code!

- Initialize the *Kadabra* object with your Flask application object.
- Decorate any of your routes for which you want to record metrics with *record_metrics*.
- Optionally, instrument your application code with any additional metrics you want to record with the `MetricsCollector` object, available as the `metrics` attribute on the `g` object. This is shared across your request, is available anywhere within the Flask application context, and is totally threadsafe. Note that you don't need to record request timing nor 400/500 errors; these will be automatically included for each request.

When you run your Flask app, you'll run a local Redis server along with the Kadabra agent side-by-side with your app, which will enable you to publish your metrics with no impact to your application's performance, whatever its scale. For more information on deployment, check out Deploying Your Stack.

1.2.1 Initialization

You can initialize the *Kadabra* object directly:

```
from flask import Flask
from flask_kadabra import Kadabra

app = Flask()
kadabra = Kadabra(app)
```

Or, you can defer the installation if you are utilizing the `flask:patterns/appfactories` pattern, using the *init_app()* method:

```
from flask import Flask
from flask_kadabra import Kadabra

kadabra = Kadabra()

def create_app():
    app = Flask()
    kadabra.init_app(app)
```

You can configure the underlying Kadabra client API by passing a dictionary as the second argument to the *Kadabra* constructor:

```
config = {
    'CLIENT_CHANNEL_ARGS': {
        'port': 6500
    }
}

app = Flask()
kadabra(app, config)
```

For more information about how to configure the client API, see Kadabra's Configuration documentation.

There are also some configuration values you can specify for your Flask application object to change the behavior of the *Kadabra* object. For more info, check out Configuration.

1.2.2 Enabling Metrics for Your Routes

To record metrics for API requests to one of your routes, simply use the *record_metrics* decorator:

```
@api.route('/')
@record_metrics
def index():
    return "Hello, World!"
```

This will record the request time in milliseconds (as a timer called "RequestTime"), whether the HTTP status code of the response was a server error (as a counter called "Failure" with a value of 0 or 1), and whether the HTTP status code of the response was a client error (as a counter called "ClientError" with a value of 0 or 1). For more information on counters and timers, see Collecting Metrics.

These metrics will be grouped under a "Method" dimension whose value is the name of your view function, as well as any additional dimensions you've specified for the `CLIENT_DEFAULT_DIMENSIONS` key in Kadabra's configuration (see Configuration).

Additionally, a `metrics` attribute will be added to Flask's `g` object. This exposes the underlying `MetricsCollector` API which allows you to add counters and timers in your application code. They will be grouped under the same dimensions as the request time, failure, and client error metrics.

1.2.3 Instrument Your Code with Additional Metrics

Using `g.metrics` you can record additional metrics from your application code. For example, if one of your APIs calls an external third-party service you may want to time the call:

```
start = datetime.datetime.utcnow()
response = requests.get(...) # External call
end = datetime.datetime.utcnow()
g.metrics.set_timer("ExternalCallTime", end - start)
```

Any metrics you record in the context of a request being executed will be grouped together under the same dimensions, meaning the same "Method" and any other dimensions you set via the `CLIENT_DEFAULT_DIMENSIONS` configuration key or elsewhere in your application code.

You can control aspects of how your Flask app uses Kadabra via Configuration.

1.3 Configuration

Most of the configuration you'll do is for the `Kadabra` client API itself. You'll pass a dictionary containing all the configuration keys and values for any defaults you want to override when you initialize the Flask extension:

```
from flask import Flask
from flask_kadabra import Kadabra

app = Flask()

config = {
    "CLIENT_DEFAULT_DIMENSIONS": {
        "environment": "development"
    }
}

kadabra = Kadabra()
kadabra.init_app(app, config)
```

Or using the constructor directly:

```
kadabra = Kadabra(app, config)
```

Configuration keys, values, and defaults are explained in the Kadabra documentation under Configuration.

However, the Flask extension does support one configuration value itself, which can be stored in the Flask application's `Config`.

DISABLE_KADABRA	If present in the config and set to `True`, metrics will not actually be sent to the channel. This is useful if you are just developing your service and don't need to actually see metrics flowing yet.

1.4 Deploying Your Stack

Deploying your Flask application involves more than just your application code itself. Even the simplest production deployment requires a real webserver to route HTTP requests to your application (see Deployment Options). I refer to the application code plus everything needed for it to smoothly run as a fully-functioning web service as your **stack**. Your stack is the unit that is deployed along with your application code to your local host, a remote server, a virtual machine, or whatever environment from which you're running everything (more on that in a moment).

Kadabra requires two additions to your application's stack:

1. A channel to send metrics asynchronously from your application. Channels are described more in Sending Metrics, but since Redis is currently the only supported channel, this means you will need to be running a Redis server to which your Flask application can connect. Typically this just means running a local Redis instance alongside your application.
2. A properly configured `Agent` running in a dedicated process to publish your application metrics. For more information see Publishing Metrics.

You'll eventually want a backend database where you can publish metrics. The only backend database currently supported is InfluxDB. You can setup an InfluxDB server on your hosting provider or use InfluxDB's hosted options. Until you have your server setup, you can just use the `DebugPublisher` to send metrics to a logger.

Note: Having the metrics in a dedicated database like InfluxDB will be extremely helpful for production environments and allows you to easily set up dashboards and alarming around your service. You should definitely spend some time setting it up when you're ready to deploy into production and start acquiring users.

1.4.1 Infrastructure Management

In the modern era, you will want to make it as easy to deploy your application as possible. You don't want to be manually installing your dependencies, starting the Redis server, kicking off your application, and so on for every deployment. In addition to saving developer pain, this also helps prevent bugs from being introduced during deployment time, and makes it simple to deploy to different environments and cloud providers. Furthermore, you don't want to have to manually restart processes when they inevitably fail; your infrastructure management system should take care of process management for you.

Options include configuration management and orchestration tools such as Puppet, Chef, and Ansible. Personally I like to deploy my applications as containers using Docker. Whatever tool you use, you'll want to be sure that metrics flow from your application to the publisher destination of your choice.

1.4.2 The Full Stack

I think of my basic Flask application (say, one that talks to a SQL database) in five components:

1. The Flask application code itself
2. The HTTP webserver to serve the Flask application (typically gunicorn running locally)
3. The reverse proxy that will listen to external requests and proxy them to gunicorn (typically nginx)
4. A local Redis server for queueing metrics to be published and as a cache if needed
5. The Kadabra `Agent` for publishing metrics

Additionally I usually run an InfluxDB server that can only be accessed by my webserver hosts, which will publish metrics to that server.

Because I use Docker, I typically author a simple compose file with a service for each of the components above. A thorough treatment of "dockerizing" the entire stack is beyond the scope of this section, but the agent is worth talking about.

Running the agent basically consists of configuring it and calling the `start()` method. You can use the code from Getting Started and just run it in a dedicated Python process, with the possible addition of configuration e.g. if your Redis server is refered to as something other than `localhost`.

You will want to run this process under some sort of a process management system, at a minimum something like supervisord but ideally a more robust system like Docker. In my compose file I use the `command` configuration key with something like `python run.py`, where `run.py` contains the agent code. The agent is designed to respond gracefully to shutdown signals like `SIGINT` and `SIGTERM`, and will try to make sure there are no metrics that haven't yet been published before shutting down.

1.5 API

class `flask_kadabra.`**`Kadabra`**(*app=None*, *config=None*)

This object holds ties the Flask application object to the Kadabra library. Each app object gets its own `Kadabra` instance, which it uses to generate a `MetricsCollector` for each request.

Parameters

- **app** (*Flask*) – The Flask application object to initialize.
- **config** (*dict*) – Dictionary of configuration to use for the `Kadabra` client API. For information on the acceptable values see Configuration.

init_app(*app*, *config=None*)

Configure the application to use Kadabra. Requests will have access to a `MetricsCollector` via the `metrics` attribute of Flask's `g` object. You can record metrics anywhere in the context of a request like so:

```
...
g.metrics.add_count("userSignup", 1)
...
```

The metrics object will be closed and sent at the end of the request if any view that handles the request has been annotated with *record_metrics*.

`flask_kadabra.`**`record_metrics`**(*func*)

Views that are annotated with this decorator will cause any request they handle to send all metrics collected via the Kadabra client API. For example:

```
@api.route('/')
@record_metrics
def index():
    return 'Hello, world!'
```

Parameters **func** (*function*) – The view function to decorate.

f

F

I

K

R