Kadabra Documentation

Release 0.0.1

Alex Landau

December 13, 2016

Contents

You need to know what's going on with your Python application. How many people signed up today? How often does it crash? How long did it take to run your weekly processing jobs? How many orders did you fill yesterday?

You should be able to answer these questions quickly and easily. It shouldn't be a hassle to check the status of your web app, or answer important business questions about your service. And it shouldn't cost a fortune. There are plenty of metrics services out there, but they cost a lot of money and sometimes even require some sort of contractual commitment. And they usually give you more than you really need.

Kadabra provides a simple API to instrument your application code to record metrics and a performant, reliable agent to publish your metrics into a database. It is cost-effective, scales with your application, is fully unit-tested, and best of all, it runs *completely on open-source software.*

If you're willing to put in a bit of work, you can save a lot of money and maintain control of your application infrastructure.

Head on over to the Overview section to get started.

CHAPTER 1

Contents:

1.1 Overview

At a high level Kadabra consists of three components:

- A client for collecting the metrics in your application. You will configure the client API and instrument your application code to record relevant metrics, such as user signups or 500 errors.
- Channels for temporarily queueing the metrics to be published asynchronously. Currently the only supported channel is Redis, so all you'll need to do is make sure you have a local Redis server running side-by-side with your application (this is covered later, don't fret!).
- An agent for dequeueing metrics and publishing them. You will provide configuration and run the agent in a dedicated process, separate from your application.

Metrics are published asynchronously to have minimal impact on your application's performance. The client provides a simple interface for gathering metrics in your application, and the agent takes metrics from the intermediate channel and publishes them.

If you're ready to start using Kadabra, head on over to Installation.

1.2 Installation

Kadabra is a library available through the Python package index (pip). Installation is simple:

```
$ sudo pip install Kadabra
```

I recommend you use virtualenv to isolate your installs to specific development environments and avoid installing libraries system-wide.

Once you've installed Kadabra (ideally in a virtualenv), you're ready to head to the Getting Started section.

1.3 Getting Started

In this section we will demonstrate the basic functionality of Kadabra by publishing some metrics using the *DebugPublisher*, which just outputs the metrics to a logger.

Make sure you have installed Kadabra following the directions in the Installation section.

1.3.1 Install and Run Redis Server

First, we need to have an instance of Redis server running locally. Redis is very easy to setup and run. If you haven't used it before, follow the docs here to get your Redis server up and running. We will assume the Redis server is running locally on port **6379** (the default port).

1.3.2 Run the Agent

Next, we will run the agent, which will watch for metrics and publish them (in this case, to a logger setup to write to stdout). Put the following code into a file called **run_agent.py**:

```
import logging, sys, kadabra

handler = logging.StreamHandler(sys.stdout)

agent_logger = logging.getLogger("kadabra.agent")
agent_logger.setLevel("INFO")
agent_logger.addHandler(handler)

publisher_logger = logging.getLogger("kadabra.publisher")
publisher_logger.setLevel("INFO")
publisher_logger.addHandler(handler)

agent = kadabra.Agent()
agent.start()
```

This code sets up the agent's logger (so we can see what it's doing) and the publisher's logger (so we can see the metrics get printed).

Now we can run the agent:

```
python run_agent.py
```

You should see some output indicating that the agent has started running, meaning it is listening to Redis for metrics. If you see some output about connection errors, ensure that the Redis server is actually running and that you can connect to it.

1.3.3 Publish Some Metrics

Now, in a separate terminal, open up a Python interpreter. We'll record some metrics and publish them. First, let's initialize the Kadabra client:

```
>>> import kadabra
>>> client = kadabra.Kadabra()
```

Now let's get a *MetricsCollector* object from our client. This is the API we use to record metrics in our application. It is thread-safe and can be shared throughout our application:

```
>>> metrics = client.metrics()
```

In a real application, the metrics code would be located in places where we care about recording important information, such as customer sales, user signups, or application failures. But since we are just demonstrating basic functionality, let's record a few counters and a timer:

```
>>> import datetime
>>> metrics.add_count("myCount", 1.0)
>>> metrics.add_count("myOtherCount", 1.0)
```

```
>>> metrics.add_count("myOtherCount", 1.0)
>>> metrics.set_timer("myTimer", datetime.timedelta(seconds=30), kadabra.Units.MILLISECONDS)
```

Note: Timers record a `datetime.timedelta`, but you can report the actual value in any of the units from *`kadabra.Units`*.

Now we're ready to send our metrics for publishing:

```
>>> client.send(metrics.close())
```

Go over to the terminal where your agent is running. You should see the metrics output as serialized JSON. Lastly, hit CTRL+C to gracefully shut down the agent.

1.3.4 Configuration

You can configure the client API and the Agent using a dictionary when you instantiate them to control various aspects of their functionality. For more information see the Configuration section.

1.3.5 Publishing to Storage

The *`DebugPublisher`* just serializes the metrics into JSON and outputs them to a logger. You could pipe this output into another program which writes the metrics into more permanant storage. But it would be best to publish the metrics directly into a database that is designed for metrics. Time-series databases are ideal for storing metrics data.

One such database engine is InfluxDB, which is capable of storing metrics with indexed tags and provides mechanisms for querying those metrics in useful ways. Kadabra ships with an *`InfluxDBPublisher`* that can publish metrics straight to an InfluxDB server - you just provide the host, port, and database name.

For a guide on how to set up Kadabra to publish metrics using InfluxDB, see Using with InfluxDB.

1.3.6 Learning More

You now have everything you need to use Kadabra in your application. You can find out more about Collecting Metrics, Sending Metrics, and Publishing Metrics in the corresponding sections. For a complete look at the API, see API.

1.4 Configuration

Kadabra has a default set of configuration values. You can override the default configuration by specifying a dictionary when you initialize the *`Kadabra`* and *`Agent`* whose keys are the configuration you want to override and whose values are the values you want to use.

For exampe, configuring the client API to use a different Redis port might look like this:

```
my_config = {
    'CLIENT_CHANNEL_ARGS': {
        'port': 6500
}
client = kadabra.Kadabra(configuration=my_config)
```

Note that for configuration values that are dictionaries (like arguments for initializing the channel), you only need to specify the keys/values that you are overriding - the defaults will be used for the other arguments.

Configuring the Agent to write to a local InfluxDB server (using the default arguments) might look like this:

```
my_config = {
    'AGENT_CHANNEL_ARGS': {
        'port': 6500
    },
    'AGENT_PUBLISHER_TYPE': 'influxdb'
}
agent = kadabra.Agent(configuration=my_config)
```

A full list of the recognized configuration keys, descriptions of each, and their default values for the client API and the Agent are provided below.

1.4.1 Client API

CLIENT_DEFAULT_DIMENSIONS	If specified, any collectors instantiated from the client will have these dimensions upon instantiation. Should be dictionary of strings to strings. **Default:** `{}`
CLIENT_TIMESTAMP_FORMAT	The (Python-style) format to use for timestamps when serializing metrics to be sent over the channel. Must match the format of the agent that is publishing those metrics. **Default:** `%Y-%m-%dT%H:%M:%S.%fZ`
CLIENT_CHANNEL_TYPE	The type of the channel to use for transporting metrics. Currently the only accepted value is 'redis'. **Default:** `redis`
CLIENT_CHANNEL_ARGS	Dictionary of overrides for the default channel arguments. Keys should match the argument names for the channel constructor. You can specify any, all, or none of the arguments to override; the defaults will be used for any arguments that are not overridden. **Default:** *None*

1.4.2 Agent

AGENT_LOGGER_NAME	The name of the logger that the agent will use to log messages. **Default:** `kadabra.agent`
AGENT_CHANNEL_TYPE	The type of the channel to use for receiving metrics. Currently the only accepted value is 'redis'. **Default:** `redis`
AGENT_CHANNEL_ARGS	Dictionary of overrides for the default channel arguments. Keys should match the argument names for the channel constructor. You can specify any, all, or none of the arguments to override; the defaults will be used for any arguments that are not overridden. **Default:** *None*
AGENT_PUBLISHER_TYPE	The type of the publisher to use for publishing metrics. The acceptable values are 'debug' and 'influxdb'. **Default:** `debug`
AGENT_PUBLISHER_ARGS	Dictionary of overrides for the default publisher arguments. Keys should match the argument names for the publisher constructor. You can specify any, all, or none of the arguments to override; the defaults will be used for any arguments that are not overridden. **Default:** None
AGENT_RECEIVER_THREADS	The number of threads the agent will use for publishing metrics from the channel. **Default:** *3*
AGENT_NANNY_FREQUENCY_SECONDS	How often the agent will check for metrics that have been in-progress for a long time so that they can be republished. **Default:** *30*
AGENT_NANNY_THRESHOLD_SECONDS	How many seconds metrics must be in-progress before they are considered in-progress for a "long time" (and will be retried by the nanny). **Default:** *60*
AGENT_NANNY_QUERY_LIMIT	The maximum number of in-progress metrics that the nanny will process at once. This is necessary because the in-progress queue is always changing, so the nanny must take a "snapshot" of the currently in-progress metrics. **Default:** *5000*
AGENT_NANNY_THREADS	The number of threads the agent will use for re-publishing metrics that have been in-progress for a long time. **Default:** *3*

1.5 Collecting Metrics

This section describes in detail the kinds of metrics you can gather in your application using a *MetricsCollector* which can be retrieved via the client's *metrics()* method, and the semantics of using the collector.

1.5.1 Dimensions

*Dimension*s are simply key-value pairs that allow you to categorize and label metrics based on shared traits such as "environment" (e.g. *beta*, *prod*) or "jobType" (e.g. *notifyUsers*, *processData*):

```
>>> metrics.set_dimension("name", "value")
```

Dimensions will apply to all counters and timers associated with the collector.

When you use a publisher such as the *InfluxDBPublisher* you will publish metrics into a database. For databases that support it, dimensions will become indexed fields on your data, allowing you to efficiently query and filter on these dimensions.

Note that dimensions should never be tied to an "unbounded" domain of values (like, for example, user ID). You should keep dimensions to a small domain of known values to prevent your indexes from exploding in size.

1.5.2 Counters

*Counter*s track floating point values identified by a key. The collector's *add_count()* method will create a

counter if it doesn't exist or add a value to an existing counter. Once the collector is closed and sent, only the final value will be reported for each counter:

```
>>> metrics.add_count("myCount", 1)
>>> metrics.add_count("myCount", 3)
>>> closed = metrics.close()
>>> len(closed.counters)
1
>>> closed.counters[0].name
'myCount'
>>> closed.counters[0].value
4.0
```

This allows you to aggregate counts locally before publishing them as a single metric.

1.5.3 Timers

*Timer*s associate a key with a `timedelta`, along with a *Unit* representing the offset from seconds that the time should be reported in:

```
>>> metrics.set_timer("myCount", datetime.timedelta(seconds=30),\
        kadabra.Units.MILLISECONDS)
```

Common units are found in *Units*.

Timers can only be set; if you call *set_timer()* with an existing key, the timer will be overwritten with the new value.

1.5.4 Metadata

Metadata are also simple key-value pairs, but unlike dimensions they are not meant to be indexed in a database. They provide a way to associate additional context with your metrics without having to pay the storage costs associated with indexing dimensions. Metadata is particularly useful for storing keys with highly dimensional values (e.g. the user ID associated with a particular metric), which you wouldn't need to query/filter on later.

Metadata can be associated with individual counters or timers by passing in a dictionary to the "metadata" argument when you record a counter or timer:

```
>>> metrics.add_count("myCount", 1.0, metadata={"name", "value"})
>>> metrics.set_timer("myCount", datetime.timedelta(seconds=30),\
        kadabra.Units.MILLISECONDS, metadata={"name", "value"})
```

If you specify metadata for an existing counter or timer, the previous metadata will be *completely* replaced with the new metadata. If you have specified previous metadata for a timer or counter and don't specify metadata on subsequent calls to *add_count()* or *set_timer()* for the same counter or timer, the previous metadata will remain.

The way metadata is ultimately handled depends on the publisher. For example, the *InfluxDBPublisher* will transform the metadata into fields for each measurement.

Note: Don't use *value* or *unit* for metadata keys; these are reserved and will be overwritten.

1.5.5 Timestamps

Because metric data may be published some time after the metric was originally recorded, you will want to associate the timestamp of the metric with when it was originally created, not when it gets published/writted to a database.

Otherwise your metric data may appear delayed and inaccurate.

By default, timestamps are associated with counters when they are first created, and timers each time they are set. You can override this behavior by passing your own `datetime.datetime` to the `timestamp` argument of *`add_count()`* or *`set_timer()`*, which will associate the metric with that timestamp.

For example, if you wanted to set the timestamp for a metric to 5 minutes ago:

```
>>> metrics.add_count("myCount", 1, timestamp=\
        datetime.datetime.utcnow() - datetime.timedelta(minutes=5))
```

For existing timers, any time you set the timestamp it will replace whatever timestamp already exists. However, if you try to set the timestamp for an existing counter, it will only replace the current timestamp if you pass the `replace_timestamp` parameter with a value of *False*:

```
>>> metrics.add_count("myCount", 1, timestamp=datetime.datetime.utcnow(),\
        replace_timestamp=True)
```

Because the timestamp defaults to "now" (in UTC) if unspecified, this allows you to easily update the timestamp of a counter each time you add to it:

```
>>> metrics.add_count("myCount", 1, replace_timestamp=True)
```

If you don't specify `replace_timestamp` the timestamp will remain at whatever value was set when you first created the counter.

1.6 Sending Metrics

This section describes channels in more detail, including the philosophy behind them and how they work internally. You won't need to use channels directly; rather you will configure one when you initialize your *`Kadabra`* and *`Agent`* (and usually the defaults are fine). You also don't need to understand this section to use Kadabra; it's for those who are curious how the internals work.

1.6.1 Why channels?

You could publish metrics to a database directly from your application. However, there are problems with this approach:

- It adds performance overhead. Usually the database will be on a different server, which means you have to pay the cost for an extra network call directly in your application.
- What happens if publishing fails? Should your application fail? Should it silently ignore the metrics that failed to publish? Should it retry publishing them? How many times should it retry?
- You may have multiple applications running on a single host that publish metrics to the same database but with different cadences. You need to make sure your database can handle the load without slowing down and impacting your applications.

Channels solve these issues by providing temporary intermediate storage that allows metrics to be published asynchronously with robust handling of failures.

To use a channel you need to set up the appropriate storage mechanism (e.g. Redis server) and configure your client API and Agent to use it.

1.6.2 How Channels Work

Channels expose four methods:

- **transport()** pushes metrics into the intermediate storage. It is used by *Kadabra* to send metrics for publishing.
- **receive()** pulls metrics from the intermediate storage. It is used by the *Agent* to fetch metrics for publishing. It also moves the metrics into a special "in-progress" queue, indicating that they are in the process of being published.
- **complete()** marks metrics as successfully published, removing them from intermediate storage. It is called by the Agent once metrics have been successfully published.
- **in_progress()** queries metrics from the in-progress queue. It is used by the Agent's *Nanny* to retry metrics that have been in progress for a long time (e.g. if they failed to publish because the backing store experience an outage).

These mechanisms allow your application to efficiently queue metrics for publishing (the performance of **transport()** is very fast) and enables to agent to publish metrics asynchronously, and re-attempt publishing failures.

1.6.3 RedisChannel

The *RedisChannel* sends your metrics over a Redis server at the host, port, and database that you specify. Redis is extremely simple to set up, and provides great performance. The configuration values are:

- **host**: The host of the Redis server. I recommend just using localhost. (Defaults to *localhost*)
- **port**: The port of the Redis server. (Defaults to *6379*)
- **db**: The database on the Redis server to store the metrics before they are published. I highly recommend using a dedicated database for Kadabra to prevent collision with your application keys (if your application uses Redis).

You can overwrite any or none of these values in the `CLIENT_CHANNEL_ARGS` and `AGENT_CHANNEL_ARGS` configuration keys. For more information on how to configure the client API and Agent see Configuration.

Note: Make sure your agent and client use the same channel type and arguments. Otherwise your metrics will not get published!

Generally I recommend just running the Redis server locally on the host that is running the application(s) from which you want to get metrics. In fact, you should probably run it as part of your deployment stack. For more information see Running in Production.

1.7 Publishing Metrics

Metrics are published by receiving them over the channel and calling the publisher's **publish()** method. This functionality is handled entirely by the *Agent*. All you need to do is configure it and call *start()* from a dedicated process. This section describes the Agent in more detail as well as the different publishers that are available to you.

1.7.1 Agent

The agent is a program that runs in a dedicated process. It reads metrics from the configured channel and publishes them into the configured backing store, completely independent of the application(s) that is/are sending metrics over the channel.

The agent basically manages a *Receiver* and a *Nanny*. The receiver manages a list of *ReceiverThread*s which poll the channel for any metrics that need to be published. The nanny periodically queries the the channel for any metrics that are in the process of being published, and attempts to publish any that have been in that state for a long time (over a certain threshold of seconds) using *NannyThread*s.

This allows the agent to be robust to publishing failures, and be scaled indepedently from your application (by increasing the number of receiver and nanny threads as appropriate, depending on how often your application publishes metrics).

One important implication from this design is that publishers should be idempotent, meaning publishing the exact same metrics multiple times should not impact your underlying database. For example, InfluxDB will not create a new measurement if you publish metrics with the same timestamp, dimensions, names, metadata, and values. This means that you will not end up with duplicated metrics from the agent retrying to publish the same metrics multiple times.

1.7.2 DebugPublisher

The *DebugPublisher* simply takes a logger name, and will log metrics (as serialized JSON) to the logger at the INFO level. This is useful for ensuring that your metrics are being properly calculated and reported by your application during development, before you use a publisher that goes to a database.

1.7.3 InfluxDBPublisher

The *InfluxDBPublisher* sends metrics to InfluxDB, a powerful open-source time series database. Each counter and timer will result in a seperate measurement which has the name of the counter or timer.

For each measurement:

- Dimensions will become tags (which are indexed, making queries for specific dimension values extremely fast)
- Counters will have a single field called *value* with the counter value
- Timers will have two fields: *value* which contains the timer value (as the `timedelta` seconds multiplied by the unit's seconds offset) and *unit* which is the name of the unit.
- Metadata will become additional fields (each key becomes a field's name, and each value becomes the field's value).

The configuration values for this publisher are:

- **host**: The host of the InfluxDB server. (Defaults to *localhost*)
- **port**: The port of the InfluxDB server. (Defaults to *8086*)
- **database**: The name of the InfluxDB database to use. Make sure you create this database on the server before you start sending metrics to it! (Defaults to *kadabra*)
- **timeout**: The timeout in seconds to wait for the InfluxDB server to respond before failing to publish. (Defaults to *5*)

You can overwrite any or none of these values in the `AGENT_PUBLISHER_ARGS` configuration key. For more information on how to configure the Agent see Configuration.

For a guide to use Kadabra with InfluxDB see Using with InfluxDB.

1.8 Using with InfluxDB

InfluxDB is a great open-source time-series database that works very well for storing metrics. It is scalable, simple to use, and has an active developer community. Plus, open-source dashboarding software such as Graphana includes

plugins for InfluxDB, making it very easy to create dashboards from your Kadabra metrics.

This section of the docs will help you set up Kadabra with a locally-running instance of InfluxDB from scratch.

1.8.1 Install Kadabra and Redis

First, make sure you have installed Kadabra and have a locally running Redis server by following the directions from Getting Started. Make sure you can connect to Redis using the CLI:

```
=> redis-cli
127.0.0.1:6379> ping
PONG
```

1.8.2 Install and Run InfluxDB

Now you need to install the InfluxDB server by following the directions here. Make sure the InfluxDB server is running and that you can connect to it using the CLI:

```
  influx
Visit https://enterprise.influxdata.com to register for updates, InfluxDB
server management, and monitoring.
Connected to http://localhost:8086 version 0.9.6.1
InfluxDB shell 0.9.6.1
>
```

Note that your InfluxDB might be a later version.

1.8.3 Create the Metrics Database

You will create the database that Kadabra will use to store metrics. The easiest way to do this is through the command line:

```
> create database kadabra;
>
```

Make sure the database exists by switching to it (we will use this to see the metrics we send shortly):

```
> use kadabra;
Using database kadabra
>
```

1.8.4 Configure and Run the Agent

Save the following into a file called **run_agent.py**:

```
import logging, sys, kadabra

handler = logging.StreamHandler(sys.stdout)

agent_logger = logging.getLogger("kadabra.agent")
agent_logger.setLevel("INFO")
agent_logger.addHandler(handler)

publisher_logger = logging.getLogger("kadabra.publisher")
publisher_logger.setLevel("INFO")
```

```
publisher_logger.addHandler(handler)

agent = kadabra.Agent(configuration={"AGENT_PUBLISHER_TYPE": "influxdb"})
agent.start()
```

The default arguments for the *InfluxDBPublisher* target the InfluxDB server running locally on port 8086, using the *kadabra* database.

Run the agent in its own terminal window:

```
python run_agent.py
```

1.8.5 Send Some Metrics

Let's write a simple program that calculates the Nth fibonacci number, and records some metrics. We'll call it fib.py:

```
import kadabra, sys, datetime

client = kadabra.Kadabra()
metrics = client.metrics()
metrics.set_dimension("program", "fibonacci")

n = int(sys.argv[1])

start = datetime.datetime.utcnow()
a, b = 0, 1
for i in range(n):
    metrics.add_count("iterations", 1)
    a, b = b, a + b
end = datetime.datetime.utcnow()

metrics.set_timer("runTime", end - start, kadabra.Units.MILLISECONDS)
client.send(metrics.close())

print a
```

This program will take a single command line integer argument, calculate that fibonacci number, and print it. But, it will also time how long this takes and count the number of loop iterations (admittedly a silly metric, since it will always be equal to N), and send these to InfluxDB.

Make sure the agent is running in a seperate terminal, and run the fibonacci program with a reasonable number (like 30):

```
=> python fib.py 30
832040
```

1.8.6 See the Metrics in InfluxDB

Now let's take a look at what ended up in InfluxDB. Using the CLI, let's view what measurements are available in our *kadabra* database:

```
> show measurements;
name: measurements
------------------
name
iterations
runTime
```

There are two measurements available, **iterations** and **runTime** corresponding to the counter and timer we set in our application.

Let's look at **runTime**:

```
> select * from runTime;
name: runTime
----------
time                  program      unit            value
1479333981920492032   fibonacci    milliseconds    0.828
```

time is the Unix epoch timestamp when the metric was created. *program* is the dimension we set. It's actually an indexed tag in InfluxDB, meaning we could efficiently query for all the metrics that share the same *program*. The *unit* tells us how to interpret the *value*: on my machine the fibonacci program calculated the value in 0.828 milliseconds.

Now let's take a look at **iterations**:

```
> select * from iterations;
name: iterations
----------------
time                  program        value
1479333981919803904   fibonacci      30
```

This counter looks mostly the same as our timer, although there is only a *value* field which is, as expected, equal to the value we passed in for N.

1.8.7 Next Steps

You've now used Kadabra to publish metrics into a real database suitable for storing and querying! But the database is running locally, which isn't particularly helpful as your infrastructure starts to grow and incorporate additional hosts. For deployment in a real production environment you'll want to host the InfluxDB server separately from your application hosts. It's easy to use any InfluxDB host with Kadabra; you just need to change the "host" argument to the IP or DNS of the remote InfluxDB host.

1.9 Running in Production

Kadabra's client can be very easily integrated into your application's code. But how do you run the agent alongside your application in a production environment? How do you ensure that when your application shuts down, the agent shuts down gracefully without losing metrics?

You will typically want to run a long-running process like the agent under a process control system such as supervisord. Such a program ensures that the agent is restarted if it is suddenly killed, and will usually be part of your broader application deployment system for managing other processes you might want to run on the same host.

Most of these systems will communicate to the processes under their control the need to shutdown using operating system signals. The process that runs the Kadabra agent should respond to these signals by calling the agent's `stop()` method, which gracefully shuts down the agent and all associated threads, ensuring that none of them are killed in the process of publishing metrics.

For example, you could use this simple program to run your agent:

```
from kadabra import Agent

import logging, sys, os, signal

agent = Agent()
signal.signal(signal.SIGINT, agent.stop)
```

```
signal.signal(signal.SIGTERM, agent.stop)
agent.start()
```

This will ensure that when the process control system sends a `SIGINT` or `SIGTERM` signal to your agent process, it will shut down gracefully.

Note: Although this will prevent the agent from shutting down in the middle of publishing metrics, it does not guarantee that the channel queues will be completely empty. There may still be pending metrics, depending on how often your application publishes metrics and how fast the metrics get published. Thus it's also a good idea to backup your channel periodically so you can restore pending metrics when your application starts up again. For example, if you use the *RedisChannel* you can set up Redis snapshots.

1.10 API

This covers all the documentation for Kadabra's various classes, including the client API, Agent, Metrics, Channels, and Publishers.

1.10.1 Client

class kadabra.**Kadabra**(*configuration=None*)

Main client API for Kadabra. In conjunction with the *MetricsCollector*, allows you to collect metrics from your application and queue them for publishing via a channel.

Typically you will use like so:

```
kadabra = Kadabra()
metrics = kadabra.metrics()
...
metrics.add_count("myCount", 1.0)
...
metrics.set_timer("myTimer", datetime.timedelta(seconds=5))
...
metrics.add_count("myCount", 1.0)
...
kadabra.send(metrics.close())
```

Parameters **configuration** (*dict*) – Dictionary of configuration to use in place of the defaults.

metrics()

Return a *MetricsCollector* initialized with any dimensions as specified by the default dimensions. The collector can be used to gather metrics from your application code.

Return type *MetricsCollector*

Returns A *MetricsCollector* instance.

send(*metrics*)

Send a *Metrics* instance to this client's configured channel so that it can be received and published by the agent. Note that a Metrics instance can be retrieved from a collector by calling its *close()* method.

Parameters **metrics** (Metrics) – The *Metrics* instance to be published.

class kadabra.client.**MetricsCollector**(*timestamp_format*, ***dimensions*)

A class for collecting metrics. Once initialized, instances of this class collect metrics by aggregating counts and keeping track of dimensions and timers. Typically you won't instantiate this class directly, but rather retrieve an instance from the client's *metrics()* method.

*Counter*s are floating point values aggregated over the lifetime of this object, and published as a single value (per counter name).

*Timer*s will be a floating point value along with a unit.

A collector instance can be used to collect metrics until it is closed by calling its *close()* method. After close() has been called, this object can be safely published without the possibility of "losing" additional metrics between the time it is closed and the time it is published.

Although collector objects are thread-safe (meaning the same object can be used by multiple threads), note that any threads that attempt to use a collector instance after it has been closed will throw an exception.

Parameters

- **timestamp_format** (*string*) – The format for timestamps when serializing into a *Metrics* instance.
- **dimensions** (*dict*) – Any dimensions that this object should be initialized with.

add_count(*name*, *value*, *timestamp=None*, *metadata=None*, *replace_timestamp=False*)

Add a new counter to this collector object, or add the value to an existing counter if it already exists.

Parameters

- **name** (*string*) – The name of the counter.
- **value** (*float*) – The floating point value to either initialize a new counter with, or add to an existing one.
- **timestamp** (*datetime*) – The timestamp to use for when this count was recorded. If unspecified, defaults to now (in UTC).
- **metadata** (*dict*) – Any metadata to include with this counter as a dictionary of strings to strings. These will be included as unindexed fields for this counter in certain metrics databases. Note that if you specify this for an existing counter, it will completely overwrite the existing metadata. However if you do not specify it, the previous metadata for the counter will remain unchanged.
- **replace_timestamp** (*bool*) – Whether to replace the exisiting timestamp for a counter if it already exists. This can be set to True if you want to update the timestamp when you add to an existing counter.

Raises **CollectorClosedError** – If this collector object has already been closed.

close()

Close this collector object and return an equivalent *Metrics* object. After this method is called, you can no longer set dimensions, set timers, or add counts to this object.

Return type *Metrics*

Returns A *Metrics* instance from the collector's dimensions, counters, and timers.

Raises **CollectorClosedError** – Raised if this collector object has already been closed.

set_dimension(*name*, *value*)

Set a dimension for this collector object. If it already exists, it will be overwritten with the new value.

Parameters

- **name** (*string*) – The name of the dimension to set.

- **value** (*string*) – The value of the dimension to be set.

Raises **CollectorClosedError** – If this collector object has already been closed.

set_timer(*name*, *value*, *unit*, *timestamp=None*, *metadata=None*)
Set a timer value for this collector object using a timedelta. If it already exists, it will be overwritten with the new value.

Parameters

- **name** (*string*) – The name of the timer to set.
- **value** (*timedelta*) – The value to use for the timer.
- **unit** (Unit) – The unit to use for this timer. Common units are specified in *kadabra.Units*.
- **timestamp** (*datetime*) – The timestamp to use for when this timer was recorded. If unspecified, defaults to now (in UTC).
- **metadata** (*dict*) – Any metadata to include with this timer as a dictionary of strings to strings. These will be included as unindexed fields for this timer in certain metrics databases. Note that if you specify this for an existing timer, it will completely overwrite the existing metadata. However if you do not specify it, the previous metadata for the timer will remain unchanged.

Raises **CollectorClosedError** – If this collector object has already been closed.

class kadabra.client.**CollectorClosedError**
Raised if you try to add metrics to or close a *MetricsCollector* object that has already been closed.

1.10.2 Agent

class kadabra.**Agent**(*configuration=None*)
Reads metrics from a channel and publishes them (see *Publishers*). The agent will spin up threads which listen to the configured channel and attempt to publish the metrics using the configured publisher. The agent will also periodically monitor the metrics that have been in progress for a while and attempt to republish them. Because the agent is meant to run indefinitely, side by side with your application, it should be configured and started in a separate, dedicated process.

Internally this object just manages a *Receiver* and *Nanny*.

Parameters **configuration** (*dict*) – Dictionary of configuration to use in place of the defaults.

start()
Start the agent. It will receive metrics from the channel, publish them, and attempt to republish metrics that have been pending for a long time (in the case of publishing failures). The agent runs until stopped; thus, you should call this method from a dedicated Python process, as it will block until the process is killed, a keyboard interrupt is detected, or the *stop()* method is called.

stop(**args*, ***kwargs*)
Stop the Agent gracefully, ensuring that any pending publish attempts are finished. This method accepts arbitrary arguments so that it can be called from any context (such as a signal handler).

class kadabra.agent.**Receiver**(*channel*, *publisher*, *logger*, *num_threads*)
Manages *ReceiverThread*s which receive metrics from the channel, move them from the queue to in-progress, and attempt to publish them. Publishing failures will result in the metrics remaining in-progress and getting picked up by the *Nanny* which will attempt to republish them.

Parameters

- **channel** (*Channels*) – The channel to read metrics from. See *Channels*.

- **publisher** (*Publishers*) – The publisher to use for publishing metrics. See *Publishers*.
- **logger** (*Logger*) – The logger to use.
- **num_threads** (*integer*) – The number of threads to use for publishing metrics.

start()
Start the receiver by starting up each *ReceiverThread*.

stop()
Stop the receiver by stopping each *ReceiverThread*.

class kadabra.agent.**ReceiverThread**(*channel*, *publisher*, *logger*)
Listens to a channel for metrics and publishes them.

Parameters

- **channel** (*Channels*) – The channel to read metrics from. See *Channels*.
- **publisher** (*Publishers*) – The publisher to use for publishing metrics. See *Publishers*.
- **logger** (*Logger*) – The logger to use.

run()
Run this thread until stopped.

stop()
Stops this this thread, ensuring that the current run will be the last one.

class kadabra.agent.**Nanny**(*channel*, *publisher*, *logger*, *frequency_seconds*, *threshold_seconds*, *query_limit*, *num_threads*)
Monitors metrics that have been in-progress for a long time and attemps to republish them. This object will periodically query objects in the in-progress queue, and try to republish them if the time between now and when they were serialized is greater than a threshold (indicating the first attempt to publish the metrics failed). It will grab the first X elements from the in-progress queue (where X is a configured value) and add them to a queue, which *NannyThread*s will read from and attempt to republish. If metrics are successfully published they will be marked as complete.

Parameters

- **channel** (*Channels*) – The channel to monitor.
- **publisher** (*Publishers*) – The publisher to use for republishing metrics.
- **logger** (*Logger*) – The logger to use.
- **frequency_seconds** (*integer*) – How often the Nanny should query the in_progress queue.
- **threshold_seconds** (*integer*) – The threshold seconds to determine if metrics should be attempted to be republished.
- **query_limit** (*integer*) – The maximum number of elements to query from the in-progress queue for any given Nanny run. This is necessary because the in-progress queue will constantly be changing. Thus nanny needs to take a "snapshot" rather than iterate through the queue.
- **num_threads** (*integer*) – The number of *NannyThread*s to use for republishing.

start()
Start the nanny by starting up each *NannyThread*.

stop()
Stop the nanny by stopping the Nanny from listening to the channela nd by stopping each *NannyThread*.

class kadabra.agent.**NannyThread**(*channel*, *publisher*, *queue*, *logger*)
Listens to a queue for metrics that have been in progress for a long time and attempts to republish them. If the publishing is successful, marks the metrics as complete.

Parameters

- **channel** (*Channels*) – The channel to mark the metrics as complete upon successful publishing.
- **publisher** (*Publishers*) – The publisher to be used to publish the metrics object.
- **queue** (*Queue*) – The queue to monitor for metrics to republish.
- **logger** (*Logger*) – The Logger to log messages to.

run()
Run this thread until stopped.

stop()
Stops this this thread, ensuring that the current run will be the last one.

1.10.3 Metrics

class kadabra.**Dimension**(*name*, *value*)
Dimensions are used for grouping sets of metrics by shared components. They are key-value string pairs which are meant to be indexed in the metrics storage for ease of querying metrics.

Parameters

- **name** (*string*) – The name of the dimension.
- **value** (*string*) – The value of the dimension.

static deserialize(*value*)
Deserializes a dictionary into a *Dimension* instance.

Parameters **value** (*dict*) – The dictionary to deserialize into a *Dimension* instance.

Return type *Dimension*

Returns A dimension that the dictionary represents.

serialize()
Serializes this dimension to a dictionary.

Return type dict

Returns The dimension as a dictionary.

class kadabra.**Counter**(*name*, *timestamp*, *metadata*, *value*)
A counter metric, which consists of a name and a floating-point value.

Parameters

- **name** (*string*) – The name of the metric.
- **timestamp** (*datetime*) – The timestamp of the metric.
- **metadata** (*dict*) – Metadata associated with this metric, in the form of string-string key-value pairs. This metadata is meant to be stored as non-indexed fields in the metrics storage.
- **value** (*float*) – The floating-point value of this counter.

static deserialize(*value*, *timestamp_format*)
Deserializes a dictionary into a *Counter* instance.

Parameters **value** (*dict*) – The dictionary to deserialize into a *Counter* instance.

Return type *Counter*

Returns A counter that the dictionary represents.

serialize(*timestamp_format*)
Serializes this counter to a dictionary.

Parameters **timestamp_format** (*string*) – The format string for this counter's timestamp.

Return type dict

Returns The counter as a dictionary.

class kadabra.**Timer**(*name*, *timestamp*, *metadata*, *value*, *unit*)
A timer metric representing an elapsed period of time, identified by a datetime.timedelta and a *Unit*.

Parameters

- **name** (*string*) – The name of the timer.
- **timestamp** (*datetime*) – The timestamp of the timer.
- **metadata** (*dict*) – The metadata associated with the timer.
- **value** (*timedelta*) – The value of the timer.
- **unit** (kadabra.Unit) – The unit of the timer value.

static deserialize(*value*, *timestamp_format*)
Deserializes a dictionary into a *Timer* instance.

Parameters **value** (*dict*) – The dictionary to deserialize into a *Timer* instance.

Return type *Timer*

Returns A timer that the dictionary represents.

serialize(*timestamp_format*)
Serializes this timer to a dictionary.

Parameters **timestamp_format** (*string*) – The format string for this timer's timestamp.

Return type dict

Returns The timer as a dictionary.

class kadabra.**Unit**(*name*, *seconds_offset*)
A unit, representing an offset from seconds. This is used by by *kadabra.Timer*s for unambiguous reporting of the timer's value.

Parameters

- **name** (*string*) – The name of the unit.
- **seconds_offset** (*integer*) – The offset of the unit relative to seconds.

static deserialize(*value*)
Deserializes a dictionary into a *Unit* instance.

Parameters **value** (*dict*) – The dictionary to deserialize into a *Unit* instance.

Return type *Unit*

Returns A unit that the dictionary represents.

serialize()
Serializes this unit to a dictionary.

Return type dict

Returns The unit as a dictionary.

class kadabra.**Units**
Container for commonly used units.

MILLISECONDS = <kadabra.metrics.Unit object>
Unit representing milliseconds.

SECONDS = <kadabra.metrics.Unit object>
Unit representing seconds.

class kadabra.**Metrics**(*dimensions*, *counters*, *timers*, *timestamp_format='%Y-%m-%dT%H:%M:%S.%fZ'*, *serialized_at=None*)
This class encapsulates metrics which can be transported over a channel, and received by the agent. It should only ever be initialized (e.g. instances are meant to be immutable). This guarantees correct behavior with respect to the client (which transports the metrics) and the agent (which receives and publishes the metrics).

Parameters

- **dimensions** (*list*) – *Dimension*s for this set of metrics.
- **counters** (*list*) – *Counter*s for this set of metrics.
- **timers** (*list*) – *Timer*s for this set of metrics.
- **timestamp_format** (*string*) – The format string for timestamps.
- **serialized_at** (*string*) – The timestamp string for when the metrics were serialized, if they were previously serialized.

static deserialize(*value*)
Deserializes a dictionary into a *Metrics* instance.

Parameters **value** (*dict*) – The dictionary to deserialize into a *Metrics* instance.

Return type *Metrics*

Returns A metrics that the dictionary represents.

serialize()
Serializes this set of metrics into a dictionary.

Return type dict

Returns The metrics as a dictionary.

1.10.4 Channels

class kadabra.channels.**RedisChannel**(*host*, *port*, *db*, *logger*, *queue_key*, *inprogress_key*)
A channel for transporting metrics using Redis.

Parameters

- **host** (*string*) – The host of the Redis server.
- **port** (*int*) – The port of the Redis server.

- **db** (*int*) – The database to use on the Redis server. This should be used exclusively for Kadabra to prevent collisions with keys that might be used by your application.
- **logger** (*string*) – The name of the logger to use.

DEFAULT_ARGS = {'db': 0, 'host': 'localhost', 'inprogress_key': 'kadabra_inprogress', 'logger': 'kadabra.channel', 'que
Default arguments for the Redis channel. These will be used by the client and agent to initialize this channel if custom configuration values are not provided.

complete(*metrics*)
Mark metrics as completed by removing them from the in-progress queue.

Parameters **metrics** (Metrics) – The metris to mark as complete.

in_progress(*query_limit*)
Return a list of the metrics that are in_progress.

Parameters **query_limit** (*int*) – The maximum number of items to get from the in progress queue.

Return type list

Returns A list of Metrics that are in progress.

receive()
Receive metrics from the queue so they can be published. Once received, the metrics will be moved into a temporary "in progress" queue until they have been acknowledged as published (by calling *complete()*). This method will block until there are metrics available on the queue or after 10 seconds.

Return type *Metrics*

Returns The metrics to be published, or None if there were no metrics received after the timeout.

send(*metrics*)
Send metrics to a Redis list, which will act as queue for pending metrics to be received and published.

Parameters **metrics** (Metrics) – The metrics to be sent.

1.10.5 Publishers

class kadabra.publishers.**DebugPublisher**(*logger_name*)
Publish metrics to a logger using the given logger name. Useful for debugging.

Parameters **logger_name** (*string*) – The name of the logger to use.

DEFAULT_ARGS = {'logger_name': 'kadabra.publisher'}
Default arguments for this publisher. These will be used by the agent to initialize this publisher if custom configuration values are not provided.

publish(*metrics*)
Publish the metrics by logging them (in serialized JSON format) to the publisher's logger at the INFO level.

Parameters **metrics** (Metrics) – The metrics to publish.

class kadabra.publishers.**InfluxDBPublisher**(*host*, *port*, *database*, *timeout*)
Publish metrics by persisting them into an InfluxDB database. Series will be created for each metric. Each metric name becomes a measurement and dimensions become the tag set. A single field will be created called 'value' which contains the value of the counter or timer. Timers will have an additional field called 'unit' which contains the name of the unit. Any metadata will become additional fields, although note that 'value' is a

reserved name that will be overwritten for both metric types, and 'unit' will be overwritten for timers. For more information about InfluxDB see the *docs <https://docs.influxdata.com/influxdb>*.

Parameters

- **host** (*string*) – The hostname of the InfluxDB database.
- **port** (*int*) – The port of the InfluxDB database.
- **database** (*string*) – The name of the database to use for publishing metrics with this publisher. Note that this database must exist prior to publishing metrics with this publisher - make sure you set it up beforehand!
- **timeout** (*int*) – The timeout to wait for when calling the InfluxDB database before failing.

DEFAULT_ARGS = {'host': 'localhost', 'port': 8086, 'timeout': 5, 'database': 'kadabra'}
Default arguments for this publisher. These will be used by the agent to initialize this publisher if custom configuration values are not provided.

publish(*metrics*)
Publish the metrics by writing them to InfluxDB.

Parameters **metrics** (Metrics) – The metrics to publish.

k

U