Baseplate.py

unknown

Apr 30, 2021

CONTENTS

1 Table of Contents **3**
- 1.1 Tutorial . . . 3
- 1.2 User Guide . . . 13
- 1.3 API Documentation . . . 19
- 1.4 CLI Tools . . . 98
- 1.5 Linters . . . 104

2 Appendix **109**

Python Module Index **111**

Index **113**

It's much easier to manage a bunch of services when they all have the same shape: the way they're developed, the way they interact with the infrastructure they run on, and the way they interact with each other. Baseplate is reddit's specification for the common shape of our services. This library, Baseplate.py, is the Python implementation of that specification.

Baseplate.py integrates with existing application frameworks and provides battle-tested libraries to give you everything you need to build a well-behaving production service without having to reinvent the wheel.

Here's a simple Baseplate.py HTTP service built using the Pyramid web framework:

```
from baseplate import Baseplate
from baseplate.clients.sqlalchemy import SQLAlchemySession
from baseplate.frameworks.pyramid import BaseplateConfigurator
from pyramid.config import Configurator
from pyramid.view import view_config


@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    result = request.db.execute("SELECT date('now');")
    return {"Hello": "World", "Now": result.scalar()}


def make_wsgi_app(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})

    configurator = Configurator(settings=app_config)
    configurator.include(BaseplateConfigurator(baseplate).includeme)
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

Every request to this example service will automatically emit telemetry that allows you to dig into how the service is performing under the hood:

- Timers for how long the whole request took and how long was spent talking to the database.
- Counters for the success/failure of the whole request and each query to the database.
- Distributed tracing spans (including carrying over trace metadata from upstream services and onwards to downstream ones).
- Reporting of stack traces to Sentry on crash.

And you don't have to write any of that.

To get started, *dive into the tutorial*. Or if you need an API reference, look below.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Tutorial

1.1.1 A tiny "Hello, World" service

In this tutorial, we're going to build up a simple service to show off various aspects of Baseplate.py.

Prerequisites

This tutorial expects you to be familiar with Python and the basics of web application development. We will use Python 3.7 and virtual environments. To get set up, see this guide on installing Python.

Make a home for our service

First, let's create a folder and virtual environment to isolate the code and dependencies for this project.

```
$ mkdir tutorial
$ cd tutorial
$ virtualenv --python=python3.7 venv
Running virtualenv with interpreter /usr/bin/python3.7
Using base prefix '/usr'
New python executable in /home/user/tutorial/venv/bin/python3.7
Also creating executable in /home/user/tutorial/venv/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.
$ source venv/bin/activate
```

Build a simple Pyramid service

Pyramid is a mature web framework for Python that we build HTTP services with. We'll start our service out by using it without Baseplate.py at all:

```
$ pip install pyramid
```

Now let's write a tiny Pyramid service, open your editor and put the following in `helloworld.py`:

```
from wsgiref.simple_server import make_server

from pyramid.config import Configurator
from pyramid.view import view_config
```

(continues on next page)

(continued from previous page)

```
@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    return {"Hello": "World"}


def make_wsgi_app():
    configurator = Configurator()
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()


if __name__ == "__main__":
    app = make_wsgi_app()
    server = make_server("127.0.0.1", 9090, app)
    server.serve_forever()
```

Then run it:

```
$ python helloworld.py
```

Now that you have got a server running, let's try talking to it. From another terminal:

```
$ curl localhost:9090
{"Hello": "World"}
```

and the server should have logged about that request:

```
127.0.0.1 - - [06/Aug/2019 23:32:40] "GET / HTTP/1.1" 200 18
```

Great! It does not do much, but we have got a very basic service up and running now.

Breaking it down

See also:

You can get way more detail about what's going on in Pyramid in Pyramid's own tutorial.

There are three things going on in this tiny service. Following how the code actually runs, we start out at the end of the file with the creation of the HTTP server:

```
if __name__ == "__main__":
    app = make_wsgi_app()
    server = make_server("127.0.0.1", 9090, app)
    server.serve_forever()
```

This is using the `wsgiref` module from the Python standard library to run a basic development server. WSGI is the Python standard interface between HTTP servers and applications. Pyramid applications are WSGI applications and can be run on any WSGI server.

Note: This server will do fine for this quick start, but we won't want to stick with it as we scale up as it can't handle multiple requests at the same time.

This server code calls our `make_wsgi_app` function to get the actual application. Let's look at that next:

```
def make_wsgi_app():
    configurator = Configurator()
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

The real workhorse here is the `Configurator` object from Pyramid. This object helps us configure and build an application.

```
    configurator.add_route("hello_world", "/", request_method="GET")
```

First off, we add a route that maps the URL path `/` to the route named `hello_world` when the HTTP verb is `GET`. This means that when a request comes in that matches those criteria, Pyramid will try to find a "view" function that is registered for that route name.

```
    configurator.scan()
```

Then we tell Pyramid to scan the current module for declarative registrations. Because of the `@view_config` decorator, Pyramid will find the `hello_world` function in our service and recognize that we have registered it to handle the `hello_world` route.

```
    return configurator.make_wsgi_app()
```

Finally, we ask the configurator to build a WSGI application based on what we have configured and return that to the server.

At this point, we have done the one-time application startup and handed off our application to the server which is ready to call into it when requests come in. Now it's time to look at the code that actually runs on each request.

```
@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    return {"Hello": "World"}
```

This function gets called each time a matching request comes in. Pyramid will build a `Request` object and pass it into our function as `request`. This contains all the extra information about the request, like form fields and header values. Whatever gets returned from this function will be rendered by the `renderer` we specified in the `@view_config` and then sent to the client.

Summary

We have built a tiny service on Pyramid and understand how the code all fits together. So far, there's been no Baseplate.py at all. Next up, we'll look at what's involved with adding it in.

1.1.2 Servers and Configuration files

In this chapter, we'll lay the foundation for using Baseplate.py in the service.

Install Baseplate.py

First off, let's install Baseplate.py in your virtual environment so we can start using its components.

```
$ pip install baseplate
```

In the previous chapter, we made our service run its own HTTP/WSGI server. Now we're going to use Baseplate.py's server instead which is run with **baseplate-serve**.

```
$ baseplate-serve
usage: baseplate-serve [-h] [--debug] [--reload] [--app-name NAME]
                       [--server-name NAME] [--bind ENDPOINT]
                       config_file
baseplate-serve: error: the following arguments are required: config_file
```

Uh oh! `config_file`!? I guess we have got some more to do first.

A configuration file

Baseplate services rely on configuration to allow them to behave differently in different environments (development, staging, production, etc.). For Baseplate.py, configuration is stored in a file in standard Python INI file format as understood by `configparser`.

Open a new `helloworld.ini` in the tutorial directory and copy this into it:

```
[app:main]
factory = helloworld:make_wsgi_app

[server:main]
factory = baseplate.server.wsgi
```

Breaking it down, there are two sections to this configuration file, `[app:main]` and `[server:main]`.

```
[app:main]
factory = helloworld:make_wsgi_app
```

The first section defines the entrypoint and settings for the application itself. The `factory` is a function that returns an application object. In this case, it lives in the Python module `helloworld` and the function is called `make_wsgi_app`.

```
[server:main]
factory = baseplate.server.wsgi
```

The second section defines what kind of server we'll run and the settings for that server. Since our application is built for HTTP/WSGI, we use the WSGI server in Baseplate.py.

Note: You might notice that both application and server sections have `:main` in their names. By default, Baseplate.py tools like **baseplate-serve** will look for the sections with `main` in them, but you can override this with `--app-name=foo` to look up `[app:foo]` or `--server-name` similarly. This allows you to have multiple applications and servers defined in the same configuration file.

OK! Now let's try **baseplate-serve** with our configuration file.

```
$ baseplate-serve helloworld.ini
Traceback (most recent call last):
  File "/home/user/tutorial/venv/bin/baseplate-serve", line 14, in <module>
    load_app_and_run_server()
  File "/home/user/tutorial/venv/lib/python3.7/site-packages/baseplate/server/__init__
↪.py", line 226, in load_app_and_run_server
    app = make_app(config.app)
  File "/home/user/tutorial/venv/lib/python3.7/site-packages/baseplate/server/__init__
↪.py", line 180, in make_app
    return factory(app_config)
TypeError: make_wsgi_app() takes 0 positional arguments but 1 was given
```

It looks like we'll need a little bit more.

Run the service with `baseplate-serve`

In the previous section, we learned that the `[app:main]` section both tells **`baseplate-serve`** where to find the application *and* holds configuration for that application. The function that we specify in `factory` needs to take a dictionary of the raw configuration values as an argument. Let's add that to our service.

```
from pyramid.config import Configurator
from pyramid.view import view_config


@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    return {"Hello": "World"}


def make_wsgi_app(app_config):
    configurator = Configurator(settings=app_config)
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

All we had to do was add one parameter. We also pass it through to Pyramid's Configurator so any framework-specific settings can be picked up.

Since we're not using the `wsgiref` server anymore, we can drop the whole `if __name__ == "__main__":` section at the end of the file now.

Alright, third time's the charm, right?

```
$ baseplate-serve --debug helloworld.ini
{"message": "No metrics client configured. Server metrics will not be sent.", ...
{"message": "Listening on ('127.0.0.1', 9090), PID:2303772", ...
```

Success! The `--debug` flag will turn on some extra log messages, so we can see a request log when we try hitting the service with **`curl`** again.

```
$ curl localhost:9090
{"Hello": "World"}
```

And something shows up in the server's logs:

```
{"message": "127.0.0.1 - - [2021-03-02 15:05:46] \"GET / HTTP/1.1\" 200 126 0.002916",
↪ ...
```

You'll notice the logs look a bit different from before. In fact, they're quite a bit longer (though we've truncated a lot in the examples above.) **baseplate-serve** adds some extra info to help give context to your log entries and structures them as JSON so the log entries can be parsed and indexed.

Summary

We have now made a configuration file and made it possible to run our service with **baseplate-serve**.

So what did any of this do for us? **baseplate-serve** is how we let production infrastructure run our application and interact with it. It knows how to process multiple requests simultaneously and will handle things like the infrastructure asking it to gracefully shut down.

But the real fun of Baseplate.py comes when we start using its framework integration to get some visibility into the guts of the application. Let's see what that looks like in the next chapter.

1.1.3 Observers — Looking under the hood

In the previous chapter we took our basic Pyramid service and made it compatible with **baseplate-serve**. Now we'll start using Baseplate.py inside the service itself so we can see what's going on while handling requests.

Wire up the Baseplate object

The heart of Baseplate.py's framework is the *Baseplate* object. No matter what kind of service you're writing—Pyramid, Thrift, etc.—this class works the same. The magic happens when we use one of Baseplate.py's framework integrations to connect the two things together. Let's do that in our service now.

```
from baseplate import Baseplate
from baseplate.frameworks.pyramid import BaseplateConfigurator
from pyramid.config import Configurator
from pyramid.view import view_config


@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    return {"Hello": "World"}


def make_wsgi_app(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()

    configurator = Configurator(settings=app_config)
    configurator.include(BaseplateConfigurator(baseplate).includeme)
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

This is all we need to do to get basic observability in our service. Line-by-line:

```
    baseplate = Baseplate(app_config)
```

We create a *Baseplate* object during application startup.

```
    baseplate.configure_observers()
```

Then we call *configure_observers()* and pass in the application configuration. We'll talk more about this in a moment.

```
    configurator.include(BaseplateConfigurator(baseplate).includeme)
```

Finally we connect up with Pyramid's framework to integrate it all together.

We can now run our server again and make some requests to it to see what's different.

```
$ baseplate-serve --debug helloworld.ini
{"message": "The following observers are unconfigured and won't run: metrics, tracing,
↪ sentry", ...
{"message": "No metrics client configured. Server metrics will not be sent.", ...
{"message": "Listening on ('127.0.0.1', 9090), PID:2308014", ...
{"message": "127.0.0.1 - - [2021-03-02 15:08:15] \"GET / HTTP/1.1\" 200 145 0.002052",
↪ ...
```

It still works and things don't look too different. The first thing you'll see is the `observers are unconfigured` line. This is there because we called *configure_observers()*. We did not add anything to our configuration file so of course they're all unconfigured!

There is one other change even with those unconfigured observers. The log line for the test request we sent now has a field called "traceID". That's the Trace ID of the request. You'll see in the next section that when a single request causes multiple log lines, they'll all have the same Trace ID which helps correlate them.

Note: In fact, the Trace ID will be the same across all services involved in handling a single end-user request. We'll talk more about this in a later chapter.

That's sort of useful but we can do better. Next, let's configure another observer to get more visibility.

Configure the metrics observer

After the previous section, our application is now wired up to use Baseplate.py with Pyramid. Now we'll turn on an observer to see it in action.

One of the available observers sends metrics to StatsD. We'll turn that on, but since we don't actually have a StatsD running we'll leave it in debug mode that just prints the metrics it would send out to the logs instead.

```
[app:main]
factory = helloworld:make_wsgi_app

metrics.tagging = true
metrics.log_if_unconfigured = true

[server:main]
factory = baseplate.server.wsgi
```

This tells Baseplate to configure the *tagged metrics observer* and that it should log the metrics it would send had we configured a destination. Once we have done that, we can start the server up again.

```
$ baseplate-serve --debug helloworld.ini
{"message": "The following observers are unconfigured and won't run: tracing, sentry",
↪ ...
{"message": "Listening on ('127.0.0.1', 9090), PID:2310914", ...
{"message": "Would send metric b'baseplate.server.latency,endpoint=hello_world:1.
↪80316|ms'", ...
```

(continues on next page)

(continued from previous page)

```
{"message": "Would send metric b'baseplate.server.rate,endpoint=hello_world,
↪success=True:1|c'", ...
{"message": "127.0.0.1 - - [2021-03-02 15:10:34] \"GET / HTTP/1.1\" 200 145 0.004433",
↪ ...
```

If it worked right, `metrics` won't be listed as an unconfigured observer anymore. Now when you make requests to your service you'll see a few extra log lines that say `Would send metric...`. These are the metrics the observer would be sending if we had a StatsD server set up. Also note that the trace ID is the same on all these log lines.

Since our service is super simple, we only get two metrics on each request. The first is a timer that tracks how long the endpoint took to respond to the request. The second metric increments a success or failure counter every time the endpoint responds or crashes.

If you leave your server running long enough, you'll also see some extra metrics appear periodically:

```
{"message": "Would send metric b'runtime.open_connections,hostname=reddit,
↪PID=2311569:0|g'", ...
{"message": "Would send metric b'runtime.active_requests,hostname=reddit,
↪PID=2311569:0|g'", ...
{"message": "Would send metric b'runtime.gc.collections,hostname=reddit,PID=2311569,
↪generation=0:110|g'" ...
...
```

These metrics come out of the server itself and track information that's not specific to an individual request but rather about the overall health of the service. This includes things like statistics from Python's garbage collector, the state of any connection pools, and how many concurrent requests your application is handling.

Summary

We have integrated Baseplate.py's tools into our service and started seeing some of the benefit of its observers. Our service is pretty simple still though, it's about time it actually talks to something else. In the next chapter, we'll add a database and see what that looks like.

1.1.4 Clients — Talking to the outside world

In the previous chapter, we integrated Baseplate.py into our Pyramid application and got some observers working to see how things were performing. Most applications are not so self-contained though and have to talk to other services to do their job. In this chapter, we'll add a dependency on a database to see what that looks like.

Adding a database

We're going to use a popular Python ORM called SQLAlchemy to talk to our database. Let's install that to get started:

```
$ pip install sqlalchemy
```

Now that's installed, we can use Baseplate.py's helpers to add SQLAlchemy to our service.

```
from baseplate import Baseplate
from baseplate.clients.sqlalchemy import SQLAlchemySession
from baseplate.frameworks.pyramid import BaseplateConfigurator
from pyramid.config import Configurator
from pyramid.view import view_config
```

(continues on next page)

(continued from previous page)

```
@view_config(route_name="hello_world", renderer="json")
def hello_world(request):
    result = request.db.execute("SELECT date('now');")
    return {"Hello": "World", "Now": result.scalar()}


def make_wsgi_app(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})

    configurator = Configurator(settings=app_config)
    configurator.include(BaseplateConfigurator(baseplate).includeme)
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

Pretty simple, but there's something subtle going on here. Let's dig into it.

```
    baseplate.configure_context({"db": SQLAlchemySession()})
```

This call to *configure_context()*, during application startup, tells Baseplate.py that we want to add a SQLAlchemy `Session` to the "context" with the name `db`.

What exactly the "context" is depends on what framework you're using, but for Pyramid applications it's the `request` object that Pyramid gives to every request handler.

Note: Why do we pass in the context configuration as a dictionary? It's possible to set up multiple clients at the same time this way. You can even do more complicated things like nesting dictionaries to organize the stuff you add to the context. See *configure_context()* for more info.

```
    result = request.db.execute("SELECT date('now');")
```

Since we have connected the *Baseplate* object with Pyramid and told it to configure the context like this, we'll now see a `db` attribute on the `request` that has that SQLAlchemy session we wanted.

OK. Now we have got that wired up, let's try running it.

```
$ baseplate-serve --debug helloworld.ini
...
baseplate.lib.config.ConfigurationError: db.url: no value specified
```

Ah! It looks like we have got some configuring to do.

Configure the new client

Telling Baseplate.py that we wanted to add the SQLAlchemy session to our context did not actually give it any hint about how that session should be configured. SQLAlchemy can transparently handle different SQL databases for us and the location at which to find them will be different depending on if we're running in production, staging, or development. So it's time for the configuration file again.

```
[app:main]
factory = helloworld:make_wsgi_app

metrics.tagging = true
metrics.log_if_unconfigured = true

db.url = sqlite:///

[server:main]
factory = baseplate.server.wsgi
```

To wire up the database, all we need is to add a SQLAlchemy `URL` to the configuration file. Because we configured the session to use the name `db` the relevant configuration line is prefixed with that name.

We're just going to use an in-memory SQLite database here because it's built into Python and we don't have to install anything else.

Now when we fire up the service, it launches and returns requests with our new data.

```
$ curl localhost:9090
{"Hello": "World", "Now": "2021-03-02"}
```

Great! If you look at the server logs when you make a request, you'll notice there are new metrics:

```
$ baseplate-serve --debug helloworld.ini
...
{"message": "Would send metric b'baseplate.client.latency,client=db,
↪endpoint=execute:0.287533|ms'", ...
{"message": "Would send metric b'baseplate.client.rate,client=db,endpoint=execute,
↪success=True:1|c'", ...
{"message": "Would send metric b'baseplate.server.latency,endpoint=hello_world:9.
↪87291|ms'", ...
{"message": "Would send metric b'baseplate.server.rate,endpoint=hello_world,
↪success=True:1|c'", ...
...
```

The Baseplate.py SQLAlchemy integration automatically tracked usage of our database and reports timers, counters, and other goodies to our monitoring systems.

Summary

We have now hooked up our service to a simple database and when we run queries Baseplate.py automatically tracks them and emits telemetry.

1.2 User Guide

1.2.1 Frequently Asked Questions

Contents

- *Frequently Asked Questions*
  - *Can I serve multiple protocols (Thrift, HTTP, etc.) in one service?*
  - *What do I do about "Metrics batch of N bytes is too large to send"?*
  - *What do I do about "Context objects cannot be re-used"?*
  - *What do I do about "Cannot make child span of parent that already finished"?*

Can I serve multiple protocols (Thrift, HTTP, etc.) in one service?

Yes! While `baseplate-serve` doesn't support serving multiple protocols from the same *process* it's totally fine to run multiple instances of `baseplate-serve` out of one code base. This allows you to present different interfaces to different clients and scale each interface independently.

For example, our example HTTP service from *the tutorial* has an entrypoint function (`make_wsgi_app()`) that sets up the application:

```
def make_wsgi_app(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})

    configurator = Configurator(settings=app_config)
    configurator.include(BaseplateConfigurator(baseplate).includeme)
    configurator.add_route("hello_world", "/", request_method="GET")
    configurator.scan()
    return configurator.make_wsgi_app()
```

using configuration for that application:

```
[app:main]
factory = helloworld:make_wsgi_app

metrics.tagging = true
metrics.log_if_unconfigured = true

db.url = sqlite:///
```

and configuration that tells `baseplate-serve` how to serve the application:

```
[server:main]
factory = baseplate.server.wsgi
```

Note the `factory` setting in each section refers to a module and name in that module. We can use these to point `baseplate-serve` at different pieces of code.

To serve an additional protocol, we need another entrypoint function that returns a different kind of application. Let's add a basic Thrift service as well:

```
def make_processor(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})

    handler = MyHandler()
    processor = MyService.Processor(handler)
    return baseplateify_processor(processor, logger, baseplate)
```

and add application configuration:

```
[app:thrift]
factory = helloworld:make_processor

metrics.namespace = helloworld

db.url = sqlite:///
```

and finally define server configuration:

```
[server:thrift]
factory = baseplate.server.thrift
```

We could now run our HTTP server with `baseplate-serve myconfig.ini` or our Thrift server with `baseplate-serve --server-name=thrift --app-name=thrift myconfig.ini`. The `--server-name` and `--app-name` parameters tell `baseplate-serve` which sections of the config file to use.

There's something bad going on here though. Both server types are doing the exact same stuff with *`Baseplate`* setup and the configuration for them is duplicated. A common pattern to clean this up is to factor out a `make_baseplate` function and use it in both our entrypoints:

```
def make_baseplate(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})


def make_wsgi_app(app_config):
    baseplate = make_baseplate(app_config)

    ...


def make_processor(app_config):
    baseplate = make_baseplate(app_config)

    ...
```

Similarly, you can factor out common configuration items into a `[DEFAULT]` section in the config file which will be automatically inherited by all other sections:

```
[DEFAULT]
metrics.namespace = helloworld
db.url = sqlite:///

[app:main]
factory = helloworld:make_wsgi_app
```

(continues on next page)

(continued from previous page)

```
[app:thrift]
factory = baseplate.server.thrift

...
```

For more information on what's going on under the hood here, check out the *Application Startup Sequence*.

What do I do about "Metrics batch of N bytes is too large to send"?

As your application processes a request, it does various actions that get counted and timed. Baseplate.py batches up these metrics and sends them to the metrics aggregator at the end of each request. The metrics are sent as a single UDP datagram that has a finite maximum size (the exact amount depending on the server) that is sufficiently large for normal purposes.

Seeing this error generally means that the application generated a *lot* of metrics during the processing of that request. Since requests are meant to be short lived, this indicates that the application is doing something pathological in that request; a common example is making queries to a database in a loop.

The best course of action is to dig into the application and reduce the amount of work done in a given request by e.g. batching up those queries-in-a-loop into fewer round trips to the database. This has the nice side-effect of speeding up your application too! To get you started, the "batch is too large" error message also contains a list of the top counters in the oversized batch. For example, if you see something like `myservice.clients.foo_service.do_bar=9001` that means you called the `do_bar()` method on `foo_service` over 9,000 times!

Note: For cron jobs or other non-server usages of Baseplate.py, you may need to break up your work into smaller units. For example, if your cron job processes a CSV file of 10,000 records you could create a server span for each record rather than one for the whole job.

Because this does not usually come up outside of legitimate performance issues in the application, there are currently no plans to automatically flush very large batches of metrics (which would silently mask performance issues like this).

What do I do about "Context objects cannot be re-used"?

This means that the application used the same context object, like one created with *`make_context_object()`*, in multiple calls to *`make_server_span()`*. Context objects are tied to a single server span and cannot be re-used between different spans. Make sure to create a new context object for each span. The *`server_context()`* helper can simplify this lifecycle.

What do I do about "Cannot make child span of parent that already finished"?

This means that *`make_child()`* was called on a span that has already finished. This usually happens when an application holds onto a reference to a span and tries to continue doing operations with it after the parent request finished.

This can happen because of the application spawned a number of greenlets to do tasks concurrently but let the root greenlet finish up before the children did. The safest thing to do here is to ensure that the parent properly waits for its children before exiting.

Sometimes this is done deliberately to give a quick response to the client. It's generally best to put background work like this into a separate dedicated work queue as service worker processes are ephemeral and can be killed at any time without warning.

1.2.2 Application Startup Sequence

The most common way to start up a Baseplate.py application is to run one of the the *baseplate-serve* script. This page explains exactly what's going between that command and your application.

Contents

- *Application Startup Sequence*
  - *The Python Interpreter*
  - *Gevent Monkeypatching*
  - *Extending the `PYTHONPATH`*
  - *Listening for signals*
  - *Parsing command line arguments*
  - *Parsing the configuration file*
  - *Configuring Logging*
  - *Loading your code*
  - *Binding listening sockets*
  - *Loading the server*
  - *Handing off*

Note: `baseplate-script` is another way to run code in a Baseplate.py application that is generally useful for ephemeral jobs like periodic crons or ad hoc tasks like migrations. That script follows a much abbreviated form of this sequence.

The Python Interpreter

Because this is a Python application, before any code in Baseplate or your application can run, the Python interpreter itself must set itself up.

There are many many many steps involved in **Python's startup sequence** but for our purposes the most important thing to highlight are a number of environment variables that can configure the interpreter.

Now that the interpeter is up, it runs the actual program we wanted it to (`baseplate-serve`) and the Baseplate.py startup sequence begins.

Gevent Monkeypatching

Before doing anything else, Baseplate.py monkeypatches the standard library to use Gevent, a library that transparently makes Python asynchronous. This allows us to simulate simultaneously processing many requests by interleaving their work and switching the CPU between them as they wait for IO operations like network requests. Monkeypatching replaces most of the APIs in the Python standard library that can block a process with ones provided by Gevent which take advantage of the blocking to swap to other work.

Warning: While Gevent gives us easy concurrency, it does *not* give us parallelism. Python is still only fundamentally processing these requests in one thread, one task at a time. Keep an eye out for code that would not yield to other tasks, like CPU-bound loops, APIs that don't have asynchronous equivalents (like `flock()`), or dropping into gevent-unaware native extensions. See the `blocked_hub` monitor in *Process-level metrics* for a tool that can help debug this class of problem.

Monkeypatching is done as early as possible in the process to ensure that all other parts of the startup sequence use the monkeypatched IO primitives.

For more details on Gevent and how it works, see gevent For the Working Python Developer.

Extending the `PYTHONPATH`

Python uses a list of directories, sourced from the environment variable `PYTHONPATH`, to search for libraries when doing imports. Because it's common to want to run applications from the current directory, Baseplate.py adds the current directory to the front of the path.

Listening for signals

Baseplate.py registers some handlers for `signals` that allow the outside system to interact with it once running. The following signals have handlers defined:

`SIGUSR1` Dump a stack trace to `stdout`. This can be useful for debugging if the process is not responsive.

`SIGTERM` Initiate graceful shutdown. The server will stop accepting new requests and shut down as soon as all currently in-flight requests are processed, or a timeout occurs.

`SIGUSR2` Same as `SIGTERM`. For use with Einhorn.

`SIGINT` Same as `SIGTERM`. For Ctrl-C on the command line.

Parsing command line arguments

Command line arguments are parsed using the Python-standard `argparse` machinery.

`baseplate-serve` only requires one argument: a path to the configuration file for your service. The optional arguments `--app-name` and `--server-name` control which sections of the config file are read. The remaining options control the way the server runs.

Parsing the configuration file

Baseplate.py loads the configuration file from the path given in command line. The raw file on disk is parsed using a `configparser.ConfigParser` with interpolation disabled.

Configuration files are split up into sections that allow for one file to hold configuration for multiple components. There are generally two types of section in the config file: application configuration sections that look like `[app:foo]` and server configuration sections that look like `[server:bar]`. After parsing the configuration file, Baseplate.py uses the section names specified in the `--app-name` and `--server-name` command line arguments to determine which sections to pay attention to. If not specified on the command line, the default section name is `main`. For example, `baseplate-serve --app-name=foo` would load the `[app:foo]` and `[server:main]` sections from the config file.

Note: If you use multiple `app` or `server` blocks you may find yourself with a lot of repetition. You can move duplicated configuration to a meta-section called `[DEFAULT]` and it will automatically be inherited in all other sections in the file (unless overridden locally).

The server configuration section is used to determine which server implementation to use and then the rest of the configuration is passed onto that server for instantiation. See *Loading the server* for more details. The application configuration section determines how to load your application and then the rest of the configuration is passed onto your code, see the *Loading your code* section for more details.

Configuring Logging

Next up, Baseplate.py configures Python's `logging` system. The default configuration is:

- Logs are written to `stdout`.
- The default log level is `INFO` unless the `--debug` command line argument was passed which changes the log level to `DEBUG`.
- A baseline structured logging format is applied to log messages, see *the logging observer's documentation* for details.

This configuration affects all messages emitted through `logging` (but not e.g. `print()` calls).

If a `[loggers]` section is present in your configuration file, `logging` is given a chance to override configuration using the standard logging config file format. This can be useful if you want finer grain control of what messages get filtered out etc.

Loading your code

The next step is to load up your application code.

Baseplate.py looks inside the selected `[app:foo]` section for a setting named `factory`. The value of this setting should be the full name of a callable, like `my.module:my_callable` where the part before the colon is a module to import and the part after is a name within that module. The referenced module is imported with `importlib.import_module()` and then the referenced name is retrieved with `getattr()` on that module object.

Once the callable is loaded, Baseplate.py passes in the parsed settings from the selected `[app:foo]` section and waits for the function to return an application object. This is where your application can do all of its one-time startup logic outside of request processing.

Binding listening sockets

Unless running under Einhorn, Baseplate.py needs to create and bind a socket for the server to listen on. The address bound to is selected by the `--bind` option and defaults to `127.0.0.1:9090`.

Two socket options are applied when binding a socket:

SO_REUSEADDR This allows us to bind the socket even when connections from previous incarnations are still lingering in `TIME_WAIT` state.

SO_REUSEPORT This allows multiple instances of our application to bind to the same socket and the kernel distributes connections to them according to a deterministic algorithm. See this explanation of SO_REUSEPORT for more information. This generally only is useful under Einhorn where multiple processes are run on the same host.

Loading the server

Baseplate.py now loads the actual server code that will run the main application loop from here on out.

This process is very similar to loading your application code. The `factory` setting in the selected `[server:foo]` section of the configuration file is inspected to determine which code to load. This is generally one of the server implementations in Baseplate.py but you can write your own in your application if needed. Once loaded, the rest of the configuration is passed onto the loaded callable.

The new server object has expectations of what kind of application object your application factory returned. For example, an HTTP server expects a **WSGI** callable while the Thrift server expects a `TProcessor` object.

Handing off

Once everything is set up, Baseplate.py writes "Listening on <address>" to the log and hands off control to the server object which is expected to serve forever (unless one of the signals registered above is received) and use your application to handle requests.

1.3 API Documentation

1.3.1 Observability Framework

The observability framework is the part of Baseplate.py that integrates with other Python application frameworks to bring automatic telemetry to your application.

baseplate

The heart of the Baseplate framework is its telemetry system. Here's an incomplete example of an application built with the framework:

```
def do_something(request):
    result = request.db.execute("SELECT date('now');")

def make_app(app_config):
    ... snip ...

    baseplate = Baseplate(app_config)
    baseplate.configure_observers()
    baseplate.configure_context({"db": SQLAlchemySession()})

    ... snip ...
```

When a request is made that routes to the `do_something` handler, a *`ServerSpan`* is automatically created to track the time spent processing the request in our application. If the incoming request has trace headers, the constructed server span will have the same IDs as the upstream service's child span or else it will start a new trace with randomized values.

When we call `request.db.execute(...)` in the handler, Baseplate creates a child *`Span`* object to represent the time taken running the query.

The creation of the server and child spans trigger callbacks on all the *`ServerSpanObserver`* and *`SpanObserver`* objects registered. Because we called *`configure_observers()`* in our setup, this means we have observers that send telemetry about how our service is functioning.

Framework Configuration

At the root of each application is a single instance of *Baseplate*. This object can be integrated with various other frameworks (e.g. Thrift, Pyramid, etc.) using one of *the integrations*.

class baseplate.**Baseplate**(*app_config=None*)

The core of the Baseplate framework.

This class coordinates monitoring and tracing of service calls made to and from this service. See *baseplate.frameworks* for how to integrate it with the application framework you are using.

__init__(*app_config=None*)

Initialize the core observability framework.

Parameters **app_config** (Optional[Dict[str, str]]) – The raw configuration dictionary for your application as supplied by **baseplate-serve** or **baseplate-script**. In addition to allowing framework-level configuration (described next), if this is supplied you do not need to pass the configuration again when calling *configure_observers()* or *configure_context()*.

Baseplate services can identify themselves to downstream services in requests. The name a service identifies as defaults to the Python module the *Baseplate* object is instantiated in. To override the default, make sure you are passing in an *app_config* and configure the name in your INI file:

```
[app:main]
baseplate.service_name = foo_service

...
```

configure_observers()

Configure diagnostics observers based on application configuration.

This installs all the currently supported observers that have settings in the configuration file.

See *baseplate.observers* for the configuration settings available for each observer.

Return type None

configure_context(*context_spec*)

Add a number of objects to each request's context object.

Configure and attach multiple clients to the *RequestContext* in one place. This takes a full configuration spec like *baseplate.lib.config.parse_config()* and will attach the specified structure onto the context object each request.

For example, a configuration like:

```
baseplate = Baseplate(app_config)
baseplate.configure_context({
    "cfg": {
        "doggo_is_good": config.Boolean,
    },
    "cache": MemcachedClient(),
    "cassandra": {
        "foo": CassandraClient("foo_keyspace"),
        "bar": CassandraClient("bar_keyspace"),
    },
})
```

would build a context object that could be used like:

```
assert context.cfg.doggo_is_good == True
context.cache.get("example")
context.cassandra.foo.execute()
```

Parameters

- **app_config** – The raw stringy configuration dictionary.
- **context_spec** (Dict[str, Any]) – A specification of what the configuration should look like.

Return type None

add_to_context(*name*, *attribute_config*)

Add an attribute or a structure of attributes to each request's context object.

The given attribute config object can be one of the following:

- An arbitrary object to be added to the *RequestContext*.
- A factory with a method named make_object_for_context. On each request, the factory will be asked to create an appropriate object to attach to the *RequestContext*.
- A dict containing arbitrary objects, factories, or other dicts. In this case, a nested object will be added to the context. Each item of the dict will be processed using the same rules to become an attribute of the nested object.

Parameters

- **name** (str) – The attribute on the context object to attach the created object to. This may also be used for metric/tracing purposes so it should be descriptive.
- **attribute_config** (Any) – A configuration object.

Return type None

Per-request Context

Each time a new request comes in to be served, the time taken to handle the request is represented by a new *ServerSpan* instance. During the course of handling that request, our application might make calls to remote services or do expensive calculations, the time spent can be represented by child *Span* instances.

Spans have names and IDs and track their parent relationships. When calls are made to remote services, the information that identifies the local child span representing that service call is passed along to the remote service and becomes the server span in the remote service. This allows requests to be traced across the infrastructure.

Small bits of data, called annotations, can be attached to spans as well. This could be the URL fetched, or how many items were sent in a batch, or whatever else might be helpful.

class baseplate.**RequestContext**(*context_config*, *prefix=None*, *span=None*, *wrapped=None*)

The request context object.

The context object is passed into each request handler by the framework you're using. In some cases (e.g. Pyramid) the request object will also inherit from another base class and carry extra framework-specific information.

Clients and configuration added to the context via *configure_context()* or *add_to_context()* will be available as an attribute on this object. To take advantage of Baseplate's automatic monitoring, any interactions with external services should be done through these clients.

Baseplate.**make_context_object**()
Make a context object for the request.

Return type *RequestContext*

Baseplate.**make_server_span**(*context*, *name*, *trace_info=None*)
Return a server span representing the request we are handling.

In a server, a server span represents the time spent on a single incoming request. Any calls made to downstream services will be new child spans of the server span, and the server span will in turn be the child span of whatever upstream request it is part of, if any.

Parameters

- **context** (*RequestContext*) – The *RequestContext* for this request.
- **name** (str) – A name to identify the type of this request, e.g. a route or RPC method name.
- **trace_info** (Optional[*TraceInfo*]) – The trace context of this request as passed in from upstream. If None, a new trace context will be generated.

Return type *ServerSpan*

Baseplate.**server_context**(*name*)
Create a server span and return a context manager for its lifecycle.

This is a convenience wrapper around a common pattern seen outside of servers handling requests. For example, simple cron jobs or one-off scripts might want to create a temporary span and access the context object. Instead of calling *make_context_object()* followed by *make_server_span()* manually, this method bundles it all up for you:

```
with baseplate.server_context("foo") as context:
    context.redis.ping()
```

Note: This should not be used within an existing span context (such as during request processing) as it creates a new span unrelated to any other ones.

Return type Iterator[*RequestContext*]

class baseplate.**ServerSpan**(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*, *name*, *context*, *baseplate=None*)
A server span represents a request this server is handling.

The server span is available on the *RequestContext* during requests as the span attribute.

finish(*exc_info=None*)
Record the end of the span.

Parameters **exc_info** (Optional[Tuple[Optional[Type[BaseException]], Optional[BaseException], Optional[traceback]]]) – If the span ended because of an exception, this is the exception information. The default is None which indicates normal exit.

Return type None

incr_tag(*key*, *delta=1*)
Increment a tag value on the span.

This is useful to count instances of an event in your application. In addition to showing up as a tag on the span, the value may also be aggregated separately as an independent counter.

Parameters

- **key** (str) – The name of the tag.
- **value** – The amount to increment the value. Defaults to 1.

Return type None

log(*name*, *payload=None*)
Add a log entry to the span.

Log entries are timestamped events recording notable moments in the lifetime of a span.

Parameters

- **name** (str) – The name of the log entry. This should be a stable identifier that can apply to multiple span instances.
- **payload** (Optional[Any]) – Optional log entry payload. This can be arbitrary data.

Return type None

make_child(*name*, *local=False*, *component_name=None*)
Return a child Span whose parent is this Span.

The child span can either be a local span representing an in-request operation or a span representing an outbound service call.

In a server, a local span represents the time spent within a local component performing an operation or set of operations. The local component is some grouping of business logic, which is then split up into operations which could each be wrapped in local spans.

Parameters

- **name** (str) – Name to identify the operation this span is recording.
- **local** (bool) – Make this span a LocalSpan if True, otherwise make this span a base Span.
- **component_name** (Optional[str]) – Name to identify local component this span is recording in if it is a local span.

Return type *Span*

register(*observer*)
Register an observer to receive events from this span.

Return type None

set_tag(*key*, *value*)
Set a tag on the span.

Tags are arbitrary key/value pairs that add context and meaning to the span, such as a hostname or query string. Observers may interpret or ignore tags as they desire.

Parameters

- **key** (str) – The name of the tag.
- **value** (Any) – The value of the tag.

Return type None

start()
Record the start of the span.

This notifies any observers that the span has started, which indicates that timers etc. should start ticking.

Spans also support the context manager protocol, for use with Python's `with` statement. When the context is entered, the span calls *`start()`* and when the context is exited it automatically calls *`finish()`*.

Return type None

class baseplate.**Span**(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*, *name*, *context*, *baseplate=None*)
A span represents a single RPC within a system.

register(*observer*)
Register an observer to receive events from this span.

Return type None

start()
Record the start of the span.

This notifies any observers that the span has started, which indicates that timers etc. should start ticking.

Spans also support the context manager protocol, for use with Python's `with` statement. When the context is entered, the span calls *`start()`* and when the context is exited it automatically calls *`finish()`*.

Return type None

set_tag(*key*, *value*)
Set a tag on the span.

Tags are arbitrary key/value pairs that add context and meaning to the span, such as a hostname or query string. Observers may interpret or ignore tags as they desire.

Parameters

- **key** (str) – The name of the tag.
- **value** (Any) – The value of the tag.

Return type None

incr_tag(*key*, *delta=1*)
Increment a tag value on the span.

This is useful to count instances of an event in your application. In addition to showing up as a tag on the span, the value may also be aggregated separately as an independent counter.

Parameters

- **key** (str) – The name of the tag.
- **value** – The amount to increment the value. Defaults to 1.

Return type None

log(*name*, *payload=None*)
Add a log entry to the span.

Log entries are timestamped events recording notable moments in the lifetime of a span.

Parameters

- **name** (str) – The name of the log entry. This should be a stable identifier that can apply to multiple span instances.
- **payload** (Optional[Any]) – Optional log entry payload. This can be arbitrary data.

Return type None

finish(*exc_info=None*)
Record the end of the span.

Parameters **exc_info** (Optional[Tuple[Optional[Type[BaseException]], Optional[BaseException], Optional[traceback]]]) – If the span ended because of an exception, this is the exception information. The default is None which indicates normal exit.

Return type None

make_child(*name*, *local=False*, *component_name=None*)
Return a child Span whose parent is this Span.

Return type *Span*

class baseplate.**TraceInfo**(*trace_id: str*, *parent_id: Optional[str]*, *span_id: str*, *sampled: Optional[bool]*, *flags: Optional[int]*)
Trace context for a span.

If this request was made at the behest of an upstream service, the upstream service should have passed along trace information. This class is used for collecting the trace context and passing it along to the server span.

property trace_id
The ID of the whole trace. This will be the same for all downstream requests.

property parent_id
The ID of the parent span, or None if this is the root span.

property span_id
The ID of the current span. Should be unique within a trace.

property sampled
True if this trace was selected for sampling. Will be propagated to child spans.

property flags
A bit field of extra flags about this trace.

classmethod new()
Generate IDs for a new initial server span.

This span has no parent and has a random ID. It cannot be correlated with any upstream requests.

Return type *TraceInfo*

classmethod from_upstream(*trace_id*, *parent_id*, *span_id*, *sampled*, *flags*)
Build a TraceInfo from individual headers.

Parameters

- **trace_id** (Optional[str]) – The ID of the trace.
- **parent_id** (Optional[str]) – The ID of the parent span.
- **span_id** (Optional[str]) – The ID of this span within the tree.
- **sampled** (Optional[bool]) – Boolean flag to determine request sampling.
- **flags** (Optional[int]) – Bit flags for communicating feature flags downstream

Raises ValueError if any of the values are inappropriate.

Return type *TraceInfo*

Observers

To actually do something with all these spans, Baseplate provides observer interfaces which receive notification of events happening in the application via calls to various methods.

The base type of observer is *BaseplateObserver* which can be registered with the root *Baseplate* instance using the *register()* method. Whenever a new server span is created in your application (i.e. a new request comes in to be served) the observer has its *on_server_span_created()* method called with the relevant details. This method can register *ServerSpanObserver* instances with the new server span to receive events as they happen.

Spans can be notified of five common events:

- *on_start()*, the span started.
- *on_set_tag()*, a tag was set on the span.
- *on_log()*, a log was entered on the span.
- *on_finish()*, the span finished.
- *on_child_span_created()*, a new child span was created.

New child spans are created in the application automatically by various client library wrappers e.g. for a call to a remote service or database, and can also be created explicitly for local actions like expensive computations. The handler can register new *SpanObserver* instances with the new child span to receive events as they happen.

It's up to the observers to attach meaning to these events. For example, the metrics observer would start a timer `on_start()` and record the elapsed time to StatsD `on_finish()`.

`Baseplate.`**`register`**(*observer*)
Register an observer.

Parameters **observer** (*BaseplateObserver*) – An observer.

Return type None

class `baseplate.`**`BaseplateObserver`**
Interface for an observer that watches Baseplate.

on_server_span_created(*context*, *server_span*)
Do something when a server span is created.

Baseplate calls this when a new request begins.

Parameters

- **context** (*RequestContext*) – The *RequestContext* for this request.
- **server_span** (*ServerSpan*) – The span representing this request.

Return type None

class `baseplate.`**`ServerSpanObserver`**
Interface for an observer that watches the server span.

on_child_span_created(*span*)
Do something when a child span is created.

SpanObserver objects call this when a new child span is created.

Parameters **span** (*Span*) – The new child span.

Return type None

on_finish(*exc_info*)
Do something when the observed span is finished.

Parameters **exc_info** (Optional[Tuple[Optional[Type[BaseException]], Optional[BaseException], Optional[traceback]]]) – If the span ended because of an exception, the exception info. Otherwise, None.

Return type None

on_incr_tag(*key*, *delta*)
Do something when a tag value is incremented on the observed span.

Return type None

on_log(*name*, *payload*)
Do something when a log entry is added to the span.

Return type None

on_set_tag(*key*, *value*)
Do something when a tag is set on the observed span.

Return type None

on_start()
Do something when the observed span is started.

Return type None

class baseplate.**SpanObserver**
Interface for an observer that watches a span.

on_start()
Do something when the observed span is started.

Return type None

on_set_tag(*key*, *value*)
Do something when a tag is set on the observed span.

Return type None

on_incr_tag(*key*, *delta*)
Do something when a tag value is incremented on the observed span.

Return type None

on_log(*name*, *payload*)
Do something when a log entry is added to the span.

Return type None

on_finish(*exc_info*)
Do something when the observed span is finished.

Parameters **exc_info** (Optional[Tuple[Optional[Type[BaseException]], Optional[BaseException], Optional[traceback]]]) – If the span ended because of an exception, the exception info. Otherwise, None.

Return type None

on_child_span_created(*span*)
Do something when a child span is created.

SpanObserver objects call this when a new child span is created.

Parameters **span** (*Span*) – The new child span.

Return type None

baseplate.clients

Helpers that integrate common client libraries with Baseplate.py.

This package contains modules which integrate various client libraries with Baseplate.py's instrumentation. When using these client library integrations, trace information is passed on and metrics are collected automatically.

Instrumented Client Libraries

baseplate.clients.cassandra

Cassandra is a database designed for high-availability, high write throughput, and eventual consistency.

Baseplate.py supports both the base Python Cassandra driver and the Cassandra ORM, CQLMapper.

Example

To integrate the Cassandra driver with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
   app_config,
   {
      ...
      "foo": CassandraClient("mykeyspace"),
      ...
   }
)
```

configure it in your application's configuration file:

```
[app:main]

...

# required: a comma-delimited list of hosts to contact to find the ring
foo.contact_points = cassandra-01.local, cassandra-02.local

# optional: the port to connect to on each cassandra server
# (default: 9042)
foo.port = 9999

# optional: the name of a CredentialSecret holding credentials for
# authenticating to cassandra
foo.credential_secret = secret/my_service/cassandra-foo

...
```

and then use the attached `Session`-like object in request:

```
def my_method(request):
    request.foo.execute("SELECT 1;")
```

Configuration

class baseplate.clients.cassandra.**CassandraClient**(*keyspace*, ***kwargs*)
Configure a Cassandra client.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *cluster_from_config()* for available configuration settings.

Parameters **keyspace** (str) – Which keyspace to set as the default for operations.

class baseplate.clients.cassandra.**CQLMapperClient**(*keyspace*, ***kwargs*)
Configure a CQLMapper client.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *cluster_from_config()* for available configuration settings.

Parameters **keyspace** (str) – Which keyspace to set as the default for operations.

baseplate.clients.cassandra.**cluster_from_config**(*app_config*, *secrets=None*, *prefix='cassandra.'*, *execution_profiles=None*, ***kwargs*)
Make a Cluster from a configuration dictionary.

The keys useful to *cluster_from_config()* should be prefixed, e.g. `cassandra.contact_points` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Cluster` constructor. Any keyword arguments given to this function will be passed through to the `Cluster` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `contact_points` (required): comma delimited list of contact points to try connecting for cluster discovery
- `port`: The server-side port to open connections to.
- **`credentials_secret` (optional): the key used to retrieve the database** credentials from `secrets` as a *CredentialSecret*.

Parameters **execution_profiles** (Optional[Dict[str, ExecutionProfile]]) – Configured execution profiles to provide to the rest of the application.

Return type Cluster

Classes

class baseplate.clients.cassandra.**CassandraContextFactory**(*session*)
Cassandra session context factory.

This factory will attach a proxy object which acts like a `cassandra.cluster.Session` to an attribute on the *RequestContext*. The `execute()`, `execute_async()` and `prepare()` methods will automatically record diagnostic information.

Parameters **session** (*cassandra.cluster.Session*) – A configured session object.

class baseplate.clients.cassandra.**CQLMapperContextFactory**(*session*)
CQLMapper ORM connection context factory.

This factory will attach a new CQLMapper `cqlmapper.connection.Connection` to an attribute on the *RequestContext*. This Connection object will use the same proxy object that CassandraContextFactory attaches to a context to run queries so the *execute* command will automatically record diagnostic information.

Parameters **session** (*cassandra.cluster.Session*) – A configured session object.

baseplate.clients.kombu

This integration adds support for sending messages to queue brokers (like RabbitMQ) via Kombu. If you are looking to consume messages, check out the *baseplate.frameworks.queue_consumer* framework integration instead.

Example

To integrate it with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
   app_config,
   {
      ...
      "foo": KombuProducer(),
      ...
   }
)
```

configure it in your application's configuration file:

```
[app:main]

...

# required: where to find the queue broker
foo.hostname = rabbit.local

# optional: the rabbitmq virtual host to use
foo.virtual_host = /

# required: which type of exchange to use
foo.exchange_type = topic

# optional: the name of the exchange to use (default is no name)
foo.exchange_name = bar

...
```

and then use the attached `Producer`-like object in request:

```
def my_method(request):
    request.foo.publish("boo!", routing_key="route_me")
```

Serialization

This integration also supports adding custom serializers to Kombu via the *baseplate.clients.kombu.KombuSerializer* interface and the *baseplate.clients.kombu.register_serializer* function. This serializer can be passed to the *baseplate.clients.kombu.KombuProducerContextFactory* for use by the *baseplate.clients.kombu.KombuProducer* to allow for automatic serialization when publishing.

In order to use a custom serializer, you must first register it with Kombu using the provided *baseplate.clients.kombu.register_serializer* function.

In-addition to the base interface, we also provide a serializer for Thrift objects: *baseplate.clients.kombu.KombuThriftSerializer*.

Example

```
serializer = KombuThriftSerializer[ThriftStruct](ThriftStruct)
register_serializer(serializer)
```

Interface

class baseplate.clients.kombu.**KombuSerializer**(**args*, ***kwds*)
Interface for wrapping non-built-in serializers for Kombu.

baseplate.clients.kombu.**register_serializer**(*serializer*)
Register *serializer* with the Kombu serialization registry.

The serializer will be registered using *serializer.name* and will be sent to the message broker with the header "application/x-{serializer.name}". You need to call *register_serializer* before you can use a serializer for automatic serialization when publishing and deserializing when consuming.

Return type None

Serializers

class baseplate.clients.kombu.**KombuThriftSerializer**(*thrift_class*, *protocol_factory=<thrift.protocol.TBinaryProtocol.TBinaryPr object>*)
Thrift object serializer for Kombu.

Configuration

class baseplate.clients.kombu.**KombuProducer**(*max_connections=None*, *serializer=None*, *secrets=None*)
Configure a Kombu producer.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *connection_from_config()* and *exchange_from_config()* for available configuration settings.

Parameters

- **max_connections** (Optional[int]) – The maximum number of connections.

- **serializer** (Optional[*KombuSerializer*]) – A custom message serializer.
- **secrets** (Optional[*SecretsStore*]) – *SecretsStore* for non-default connection credentials.

baseplate.clients.kombu.**connection_from_config**(*app_config*, *prefix*, *secrets=None*, ***kwargs*)

Make a Connection from a configuration dictionary.

The keys useful to *connection_from_config()* should be prefixed, e.g. `amqp.hostname` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Connection` constructor. Any keyword arguments given to this function will be passed through to the `Connection` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `credentials_secret`
- `hostname`
- `virtual_host`

Return type `Connection`

baseplate.clients.kombu.**exchange_from_config**(*app_config*, *prefix*, ***kwargs*)

Make an Exchange from a configuration dictionary.

The keys useful to *exchange_from_config()* should be prefixed, e.g. `amqp.exchange_name` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `Exchange` constructor. Any keyword arguments given to this function will be passed through to the `Exchange` constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- `exchange_name`
- `exchange_type`

Return type `Exchange`

Classes

class baseplate.clients.kombu.**KombuProducerContextFactory**(*connection*, *exchange*, *max_connections=None*, *serializer=None*)

KombuProducer context factory.

This factory will attach a proxy object which acts like a `kombu.Producer` to an attribute on the *RequestContext*. The `publish()` method will automatically record diagnostic information.

Parameters

- **connection** (`Connection`) – A configured connection object.
- **exchange** (`Exchange`) – A configured exchange object
- **max_connections** (Optional[`int`]) – The maximum number of connections.

baseplate.clients.memcache

Memcached is a high-performance in-memory key value store frequently used for caching. Pymemcache is a Python client library for it.

Example

To integrate pymemcache with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
   app_config,
   {
      ...
      "foo": MemcacheClient(),
      ...
   }
)
```

configure it in your application's configuration file:

```
[app:main]

...

# required: the host and port to connect to
foo.endpoint = localhost:11211

# optional: the maximum size of the connection pool (default 2147483648)
foo.max_pool_size = 99

# optional: how long to wait for connections to establish
foo.connect_timeout = .5 seconds

# optional: how long to wait for a memcached command
foo.timeout = 100 milliseconds

...
```

and then use the attached `PooledClient`-like object in request:

```
def my_method(request):
    request.foo.incr("bar")
```

Configuration

class baseplate.clients.memcache.**MemcacheClient**(*serializer=None*, *deserializer=None*)

Configure a Memcached client.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *pool_from_config()* for available configuration settings.

Parameters

- **serializer** (Optional[Callable[[str, Any], Tuple[bytes, int]]]) – function to serialize values to strings suitable for being stored in memcached. An example is *make_dump_and_compress_fn()*.
- **deserializer** (Optional[Callable[[str, bytes, int], Any]]) – function to convert strings returned from memcached to arbitrary objects, must be compatible with `serializer`. An example is *decompress_and_load()*.

baseplate.clients.memcache.**pool_from_config**(*app_config*, *prefix='memcache.'*, *serializer=None*, *deserializer=None*)

Make a PooledClient from a configuration dictionary.

The keys useful to *pool_from_config()* should be prefixed, e.g. `memcache.endpoint`, `memcache.max_pool_size`, etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `PooledClient` constructor.

Supported keys:

- **endpoint (required): a string representing a host and port to connect** to memcached service, e.g. `localhost:11211` or `127.0.0.1:11211`.
- **max_pool_size: an integer for the maximum pool size to use, by default** this is `2147483648`.
- **connect_timeout: how long (as** *Timespan()*) to wait for a connection to memcached server. Defaults to the underlying socket default timeout.
- **timeout: how long (as *Timespan()*) to** wait for calls on the socket connected to memcache. Defaults to the underlying socket default timeout.

Parameters

- **app_config** (Dict[str, str]) – the raw application configuration
- **prefix** (str) – prefix for configuration keys
- **serializer** (Optional[Callable[[str, Any], Tuple[bytes, int]]]) – function to serialize values to strings suitable for being stored in memcached. An example is *make_dump_and_compress_fn()*.
- **deserializer** (Optional[Callable[[str, bytes, int], Any]]) – function to convert strings returned from memcached to arbitrary objects, must be compatible with `serializer`. An example is *decompress_and_load()*.

Return type PooledClient

Returns pymemcache.client.base.PooledClient

Classes

class baseplate.clients.memcache.**MemcacheContextFactory**(*pooled_client*)

Memcache client context factory.

This factory will attach a *MonitoredMemcacheConnection* to an attribute on the *RequestContext*. When memcache commands are executed via this connection object, they will use connections from the provided `PooledClient` and automatically record diagnostic information.

Parameters **pooled_client** (PooledClient) – A pooled client.

make_object_for_context(*name*, *span*)

Return an object that can be added to the context object.

Parameters

- **name** (str) – The name assigned to this object on the context.
- **span** (*Span*) – The current span this object is being made for.

Return type *MonitoredMemcacheConnection*

class baseplate.clients.memcache.**MonitoredMemcacheConnection**(*context_name*, *server_span*, *pooled_client*)

Memcache connection that collects diagnostic information.

This connection acts like a PooledClient except that operations are wrapped with diagnostic collection. Some methods may not yet be wrapped with monitoring. Please request assistance if any needed methods are not being monitored.

Serialization/deserialization helpers

Memcache serialization/deserialization (and compression) helper methods.

Memcached can only store strings, so to store arbitrary objects we need to serialize them to strings and be able to deserialize them back to their original form.

New services should use dump_and_compress() and decompress_and_load().

Services that need to read and write to the same memcache instances as r2 should use pickle_and_compress() and decompress_and_unpickle().

baseplate.clients.memcache.lib.**decompress_and_load**(*key*, *serialized*, *flags*)

Deserialize data.

This should be paired with *make_dump_and_compress_fn()*.

Parameters

- **key** (str) – the memcached key.
- **serialized** (bytes) – the serialized object returned from memcached.
- **flags** (int) – value stored and returned from memcached for the client to use to indicate how the value was serialized.

Return type Any

Returns The deserialized value.

baseplate.clients.memcache.lib.**make_dump_and_compress_fn**(*min_compress_length=0*, *compress_level=1*)

Make a serializer.

This should be paired with *decompress_and_load()*.

The resulting method is a chain of json.loads() and zlib compression. Values that are not JSON serializable will result in a TypeError.

Parameters

- **min_compress_length** (int) – the minimum serialized string length to enable zlib compression. 0 disables compression.
- **compress_level** (int) – zlib compression level. 0 disables compression and 9 is the maximum value.

Return type Callable[[str, Any], Tuple[bytes, int]]

Returns The serializer.

baseplate.clients.memcache.lib.**decompress_and_unpickle**(*key*, *serialized*, *flags*)

Deserialize data stored by pylibmc.

> **Warning:** This should only be used when sharing caches with applications using pylibmc (like r2). New applications should use the safer and future proofed *decompress_and_load()*.

Parameters

- **key** (str) – the memcached key.
- **serialized** (bytes) – the serialized object returned from memcached.
- **flags** (int) – value stored and returned from memcached for the client to use to indicate how the value was serialized.

Return type Any

Returns the deserialized value.

baseplate.clients.memcache.lib.**make_pickle_and_compress_fn**(*min_compress_length=0*, *compress_level=1*)

Make a serializer compatible with pylibmc readers.

The resulting method is a chain of pickle.dumps() and zlib compression. This should be paired with *decompress_and_unpickle()*.

> **Warning:** This should only be used when sharing caches with applications using pylibmc (like r2). New applications should use the safer and future proofed *make_dump_and_compress_fn()*.

Parameters

- **min_compress_length** (int) – the minimum serialized string length to enable zlib compression. 0 disables compression.
- **compress_level** (int) – zlib compression level. 0 disables compression and 9 is the maximum value.

Return type Callable[[str, Any], Tuple[bytes, int]]

Returns the serializer method.

baseplate.clients.redis

Redis is an in-memory data structure store used where speed is necessary but complexity is beyond simple key-value operations. (If you're just doing caching, prefer *memcached*). Redis-py is a Python client library for Redis.

Example

To integrate redis-py with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
   app_config,
   {
      ...
      "foo": RedisClient(),
      ...
   }
)
```

configure it in your application's configuration file:

```
[app:main]

...


# required: what redis instance to connect to
foo.url = redis://localhost:6379/0

# optional: the maximum size of the connection pool
foo.max_connections = 99

# optional: how long to wait for a connection to establish
foo.socket_connect_timeout = 3 seconds

# optional: how long to wait for a command to execute
foo.socket_timeout = 200 milliseconds

...
```

and then use the attached `Redis`-like object in request:

```
def my_method(request):
    request.foo.ping()
```

Configuration

class baseplate.clients.redis.**RedisClient**(***kwargs*)

Configure a Redis client.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *pool_from_config()* for available configuration settings.

baseplate.clients.redis.**pool_from_config**(*app_config*, *prefix='redis.'*, ***kwargs*)

Make a ConnectionPool from a configuration dictionary.

The keys useful to *pool_from_config()* should be prefixed, e.g. `redis.url`, `redis.max_connections`, etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the `redis.ConnectionPool` constructor.

Supported keys:

- `url` (required): a URL like `redis://localhost/0`.

- `max_connections`: an integer maximum number of connections in the pool
- **`socket_connect_timeout`: how long to wait for sockets to connect. e.g.** `200 milliseconds` (*`Timespan()`*)
- **`socket_timeout`: how long to wait for socket operations, e.g.** `200 milliseconds` (*`Timespan()`*)

Return type `ConnectionPool`

Classes

class baseplate.clients.redis.**RedisContextFactory**(*connection_pool*)

Redis client context factory.

This factory will attach a *`MonitoredRedisConnection`* to an attribute on the *`RequestContext`*. When Redis commands are executed via this connection object, they will use connections from the provided `redis.ConnectionPool` and automatically record diagnostic information.

Parameters **`connection_pool`** (`ConnectionPool`) – A connection pool.

`report_runtime_metrics`(*batch*)

Report runtime metrics to the stats system.

Parameters **`batch`** (*`Client`*) – A metrics client to report statistics to.

Return type `None`

`make_object_for_context`(*name*, *span*)

Return an object that can be added to the context object.

Parameters

- **`name`** (`str`) – The name assigned to this object on the context.
- **`span`** (*`Span`*) – The current span this object is being made for.

Return type *`MonitoredRedisConnection`*

class baseplate.clients.redis.**MonitoredRedisConnection**(*context_name*, *server_span*, *connection_pool*)

Redis connection that collects diagnostic information.

This connection acts like `redis.StrictRedis` except that all operations are automatically wrapped with diagnostic collection.

The interface is the same as that class except for the *`pipeline()`* method.

`execute_command`(**args*, ***kwargs*)

Execute a command and return a parsed response

Return type `Any`

`pipeline`(*name*, *transaction=True*, *shard_hint=None*)

Create a pipeline.

This returns an object on which you can call the standard Redis commands. Execution will be deferred until `execute` is called. This is useful for saving round trips.

Parameters

- **`name`** (`str`) – The name to attach to diagnostics for this pipeline.

- **transaction** (bool) – Whether or not the commands in the pipeline are wrapped with a transaction and executed atomically.

Return type MonitoredRedisPipeline

transaction(**args*, ***kwargs*)
Not currently implemented.

Return type Any

class baseplate.clients.redis.**MessageQueue**(*name*, *client*)
A Redis-backed variant of *MessageQueue*.

Parameters

- **name** (str) – can be any string.
- **client** (ConnectionPool) – should be a redis.ConnectionPool or redis.BlockingConnectionPool from which a client connection can be created from (preferably generated from the *pool_from_config()* helper).

get(*timeout=None*)
Read a message from the queue.

Parameters **timeout** (Optional[float]) – If the queue is empty, the call will block up to timeout seconds or forever if None, if a float is given, it will be rounded up to be an integer

Raises *TimedOutError* The queue was empty for the allowed duration of the call.

Return type bytes

put(*message*, *timeout=None*)
Add a message to the queue.

Parameters **message** (bytes) – will be typecast to a string upon storage and will come out of the queue as a string regardless of what type they are when passed into this method.

Return type None

unlink()
Not implemented for Redis variant.

Return type None

close()
Close queue when finished.

Will delete the queue from the Redis server (Note, can still enqueue and dequeue as the actions will recreate the queue)

Return type None

Runtime Metrics

In addition to request-level metrics reported through spans, this wrapper reports connection pool statistics periodically via the *Process-level metrics* system. All metrics are tagged with `client`, the name given to *`configure_context()`* when registering this context factory.

The following metrics are reported:

`runtime.pool.size` The size limit for the connection pool.

`runtime.pool.in_use` How many connections have been established and are currently checked out and being used.

New in version 1.5.

Changed in version 2.0: Runtime metrics were changed to use tags.

`baseplate.clients.requests`

Requests is a library for making HTTP requests. Baseplate provides two wrappers for Requests: the "external" client is suitable for communication with third party, potentially untrusted, services; the "internal" client is suitable for talking to first-party services and automatically includes trace and edge context data in requests. Baseplate uses Advocate to prevent the external client from talking to internal services and vice versa. New in version 1.4.

Example

To integrate `requests` with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
    {
        ...
        # see above for when to use which of these
        "foo": ExternalRequestsClient(),
        "bar": InternalRequestsClient(),
        ...
    }
)
```

configure it in your application's configuration file:

```
[app:main]

...

# optional: the number of connections to cache
foo.pool_connections = 10

# optional: the maximum number of connections to keep in the pool
foo.pool_maxsize = 10

# optional: how many times to retry DNS/connection attempts
# (not data requests)
foo.max_retries = 0

# optional: whether or not to block waiting for connections
# from the pool
```

(continues on next page)

(continued from previous page)

```
foo.pool_block = false

# optional: address filter configuration, see
# http_adapter_from_config for all options
foo.filter.ip_allowlist = 1.2.3.0/24

...
```

and then use the attached `Session`-like object in request:

```
def my_method(request):
    request.foo.get("http://html5zombo.com")
```

Configuration

class baseplate.clients.requests.**ExternalRequestsClient**(***kwargs*)
Configure a Requests client for use with external HTTP services.

Requests made with this client **will not** include trace context and *edge context*. This client is suitable for use with third party or untrusted services.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *http_adapter_from_config()* for available configuration settings.

class baseplate.clients.requests.**InternalRequestsClient**(***kwargs*)
Configure a Requests client for use with internal Baseplate HTTP services.

Requests made with this client **will** include trace context and *edge context*. This client should only be used to speak to trusted internal services. URLs that resolve to public addresses will be rejected. It is not possible to override the Advocate address validator used by this client.

> **Warning:** Requesting user-specified URLs with this client could lead to Server-Side Request Forgery. Ensure that you only request trusted URLs e.g. hard-coded or from a local configuration file.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *http_adapter_from_config()* for available configuration settings.

baseplate.clients.requests.**http_adapter_from_config**(*app_config*, *prefix*, ***kwargs*)
Make an HTTPAdapter from a configuration dictionary.

The keys useful to *http_adapter_from_config()* should be prefixed, e.g. `http.pool_connections`, `http.max_retries`, etc. The `prefix` argument specifies the prefix used. Each key is mapped to a corresponding keyword argument on the `HTTPAdapter` constructor.

Supported keys:

- `pool_connections`: The number of connections to cache (default: 10).
- `pool_maxsize`: The maximum number of connections to keep in the pool (default: 10).
- `max_retries`: How many times to retry DNS lookups or connection attempts, but never sending data (default: 0).
- `pool_block`: Whether the connection pool will block when trying to get a connection (default: false).

Additionally, the rules for Advocate's address filtering can be configured with the `filter` sub-keys:

- **filter.ip_allowlist: A comma-delimited list of IP addresses (1.2.3.4)** or CIDR-notation (1.2.3.0/24) ranges that the client can always connect to (default: anything not on the local network).
- **filter.ip_denylist: A comma-delimited list of IP addresses or** CIDR-notation ranges the client may never connect to (default: the local network).
- **filter.port_allowlist: A comma-delimited list of TCP port numbers** that the client can connect to (default: 80, 8080, 443, 8443, 8000).
- **filter.port_denylist: A comma-delimited list of TCP port numbers that** the client may never connect to (default: none).
- **filter.hostname_denylist: A comma-delimited list of hostnames that** the client may never connect to (default: none).
- **filter.allow_ipv6: Should the client be allowed to connect to IPv6** hosts? (default: false, note: IPv6 is tricky to apply filtering rules comprehensively to).

Return type HTTPAdapter

Classes

class baseplate.clients.requests.**BaseplateSession**(*adapter*, *name*, *span*)

A proxy for requests.Session.

Requests sent with this client will be instrumented automatically.

delete(*url*, ***kwargs*)

Send a DELETE request.

See requests.request() for valid keyword arguments.

Return type Response

get(*url*, ***kwargs*)

Send a GET request.

See requests.request() for valid keyword arguments.

Return type Response

head(*url*, ***kwargs*)

Send a HEAD request.

See requests.request() for valid keyword arguments.

Return type Response

options(*url*, ***kwargs*)

Send an OPTIONS request.

See requests.request() for valid keyword arguments.

Return type Response

patch(*url*, ***kwargs*)

Send a PATCH request.

See requests.request() for valid keyword arguments.

Return type Response

post(*url*, ***kwargs*)
Send a POST request.

See `requests.request()` for valid keyword arguments.

Return type `Response`

put(*url*, ***kwargs*)
Send a PUT request.

See `requests.request()` for valid keyword arguments.

Return type `Response`

prepare_request(*request*)
Construct a `PreparedRequest` for later use.

The prepared request can be stored or manipulated and then used with *`send()`*.

Return type `PreparedRequest`

request(*method*, *url*, ***kwargs*)
Send a request.

Parameters

- **method** (`str`) – The HTTP method of the request, e.g. `GET`, `PUT`, etc.
- **url** (`Union`[`str`, `bytes`]) – The URL to send the request to.

See `requests.request()` for valid keyword arguments.

Return type `Response`

send(*request*, ***kwargs*)
Send a `PreparedRequest`.

Return type `Response`

class baseplate.clients.requests.**RequestsContextFactory**(*adapter*, *session_cls*)
Requests client context factory.

This factory will attach a *`BaseplateSession`* to an attribute on the *`RequestContext`*. When HTTP requests are sent via this session, they will use connections from the provided `HTTPAdapter` connection pools and automatically record diagnostic information.

Note that though the connection pool is shared across calls, a new `Session` is created for each request so that cookies and other state are not accidentally shared between requests. If you do want to persist state, you will need to do it in your application.

Parameters

- **adapter** (`HTTPAdapter`) – A transport adapter for making HTTP requests. See *`http_adapter_from_config()`*.
- **session_cls** (`Type`[*`BaseplateSession`*]) – The type for the actual session object to put on the request context.

make_object_for_context(*name*, *span*)
Return an object that can be added to the context object.

Parameters

- **name** (`str`) – The name assigned to this object on the context.
- **span** (*`Span`*) – The current span this object is being made for.

Return type *BaseplateSession*

baseplate.clients.sqlalchemy

SQLAlchemy is an ORM and general-purpose SQL engine for Python. It can work with many different SQL database backends. Reddit generally uses it to talk to PostgreSQL.

Example

To integrate SQLAlchemy with your application, add the appropriate client declaration to your context configuration:

```
baseplate.configure_context(
   app_config,
   {
      ...
      "foo": SQLAlchemySession(),
      ...
   }
)
```

configure it in your application's configuration file:

```
[app:main]

...

# required: sqlalchemy URL describing a database to connect to
foo.url = postgresql://postgres.local:6543/bar

# optional: the name of a CredentialSecret holding credentials for
# authenticating to the database
foo.credentials_secret = secret/my_service/db-foo

...
```

and then use the attached `Session` object in request:

```
def my_method(request):
    request.foo.query(MyModel).filter_by(...).all()
```

Configuration

class baseplate.clients.sqlalchemy.**SQLAlchemySession**(*secrets=None*, ***kwargs*)

Configure a SQLAlchemy Session.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *engine_from_config()* for available configuration settings.

Parameters **secrets** (Optional[*SecretsStore*]) – Required if configured to use credentials to talk to the database.

baseplate.clients.sqlalchemy.**engine_from_config**(*app_config*, *secrets=None*, *prefix='database.'*, ***kwargs*)

Make an `Engine` from a configuration dictionary.

The keys useful to *engine_from_config()* should be prefixed, e.g. `database.url`, etc. The `prefix` argument specifies the prefix used to filter keys.

Supported keys:

- **`url`: the connection URL to the database, passed to** `make_url()` to create the `URL` used to connect to the database.
- **`credentials_secret` (optional): the key used to retrieve the database** credentials from `secrets` as a *CredentialSecret*. If this is supplied, any credentials given in `url` we be replaced by these.
- **`pool_recycle` (optional): this setting causes the pool to recycle connections after** the given number of seconds has passed. It defaults to -1, or no timeout.
- **`pool_pre_ping` (optional): when set to true, this setting causes** sqlalchemy to perform a liveness-check query each time a connection is checked out of the pool. If the liveness-check fails, the connection is gracefully recycled. This ensures severed connections are handled more gracefully, at the cost of doing a *SELECT 1* at the start of each checkout. When used, this obviates most of the reasons you might use pool_recycle, and as such they shouldn't normally be used simultaneously. Requires SQLAlchemy 1.3.
- `pool_size` (optional) : The number of connections that can be saved in the pool.
- `max_overflow` (optional) : Max connections that can be opened beyond the pool size.

Return type `Engine`

Classes

class baseplate.clients.sqlalchemy.**SQLAlchemyEngineContextFactory**(*engine*)

SQLAlchemy core engine context factory.

This factory will attach a SQLAlchemy `sqlalchemy.engine.Engine` to an attribute on the *RequestContext*. All cursor (query) execution will automatically record diagnostic information.

Additionally, the trace and span ID will be added as a comment to the text of the SQL statement. This is to aid correlation of queries with requests.

See also:

The engine is the low-level SQLAlchemy API. If you want to use the ORM, consider using *SQLAlchemySessionContextFactory* instead.

Parameters **`engine`** (`Engine`) – A configured SQLAlchemy engine.

class baseplate.clients.sqlalchemy.**SQLAlchemySessionContextFactory**(*engine*)

SQLAlchemy ORM session context factory.

This factory will attach a new SQLAlchemy `sqlalchemy.orm.session.Session` to an attribute on the *RequestContext*. All cursor (query) execution will automatically record diagnostic information.

The session will be automatically closed, but not committed or rolled back, at the end of each request.

See also:

The session is part of the high-level SQLAlchemy ORM API. If you want to do raw queries, consider using *SQLAlchemyEngineContextFactory* instead.

Parameters **`engine`** (`Engine`) – A configured SQLAlchemy engine.

Runtime Metrics

In addition to request-level metrics reported through spans, this wrapper reports connection pool statistics periodically via the *Process-level metrics* system. All metrics are tagged with `client`, the name given to *`configure_context()`* when registering this context factory.

The following metrics are reported:

`runtime.pool.size` The size limit for the connection pool.

`runtime.pool.open_and_available` How many connections have been established but are sitting available for use in the connection pool.

`runtime.pool.in_use` How many connections have been established and are currently checked out and being used.

`runtime.pool.overflow` How many connections beyond the pool size are currently being used. See `sqlalchemy.pool.QueuePool` for more information.

Changed in version 2.0: Runtime metrics were changed to use tags.

`baseplate.clients.thrift`

Thrift is a cross-language framework for cross-service communication. Developers write a language-independent definition of a service's API (the "IDL") and Thrift's code generator makes server and client libraries for that API.

Example

To add a Thrift client to your application, ensure that your service has Baseplate.py's Thrift build step integrated in its root `setup.py`:

```
from baseplate.frameworks.thrift.command import ThriftBuildPyCommand

...

setup(
    ...

    cmdclass={
        "build_py": ThriftBuildPyCommand,
    },
)
```

Then add the downstream service's IDL to your application directory:

```
$ cp ~/src/other/other/other.thrift myservice/
```

Baseplate.py will automatically run the Thrift compiler on this when building your application and put the output into a Python package within your application like `yourservice.other_thrift`. Import the client class and add it to your context object:

```
from .other_thrift import OtherService

...

def make_wsgi_app(app_config):
```

(continues on next page)

(continued from previous page)

```
   ...

   baseplate.configure_context(
      app_config,
      {
         ...
         "foo": ThriftClient(OtherService.Client),
         ...
      }
   )
```

then configure it in your application's configuration file:

```
[app:main]

...

# required: the host:port to find the service at
foo.endpoint = localhost:9999

# optional: the size of the connection pool (default 10)
foo.size = 10

# optional: how long a connection can be alive before we
# recycle it (default 1 minute)
foo.max_age = 1 minute

# optional: how long before we time out when connecting
# or doing an RPC (default 1 second)
foo.timeout = 1 second

# optional: how many times we'll retry connecting (default 3)
foo.max_retries = 3

...
```

and finally use the attached client in request:

```
def my_method(request):
    request.foo.is_healthy(
        request=IsHealthyRequest(probe=IsHealthyProbe.READINESS),
    )
```

Classes

class baseplate.clients.thrift.**ThriftClient**(*client_cls*, ***kwargs*)

Configure a Thrift client.

This is meant to be used with *baseplate.Baseplate.configure_context()*.

See *baseplate.lib.thrift_pool.thrift_pool_from_config()* for available configuration settings.

Parameters **client_cls** (Any) – The class object of a Thrift-generated client class, e.g. `YourService.Client`.

class baseplate.clients.thrift.**ThriftContextFactory**(*pool*, *client_cls*)
Thrift client pool context factory.

This factory will attach a proxy object with the same interface as your thrift client to an attribute on the *RequestContext*. When a thrift method is called on this proxy object, it will check out a connection from the connection pool and execute the RPC, automatically recording diagnostic information.

Parameters

- **pool** (*ThriftConnectionPool*) – The connection pool.
- **client_cls** (Any) – The class object of a Thrift-generated client class, e.g. `YourService.Client`.

The proxy object has a `retrying` method which takes the same parameters as *RetryPolicy.new* and acts as a context manager. The context manager returns another proxy object where Thrift service method calls will be automatically retried with the specified retry policy when transient errors occur:

```
with context.my_service.retrying(attempts=3) as svc:
    svc.some_method()
```

Runtime Metrics

In addition to request-level metrics reported through spans, this wrapper reports connection pool statistics periodically via the *Process-level metrics* system. All metrics are tagged with `client`, the name given to *configure_context()* when registering this context factory.

The following metrics are reported:

runtime.pool.size The size limit for the connection pool.

runtime.pool.in_use How many connections have been established and are currently checked out and being used.

Changed in version 2.0: Runtime metrics were changed to use tags.

DIY: The Factory

If a library you want is not supported here, it can be added to your own application by implementing *ContextFactory*.

class baseplate.clients.**ContextFactory**
An interface for adding stuff to the context object.

Objects implementing this interface can be passed to *add_to_context()*. The return value of *make_object_for_context()* will be added to the *RequestContext* with the name specified in `add_to_context`.

report_runtime_metrics(*batch*)
Report runtime metrics to the stats system.

Parameters **batch** (baseplate.lib.metrics.Client) – A metrics client to report statistics to.

Return type None

make_object_for_context(*name*, *span*)
Return an object that can be added to the context object.

Parameters

- **name** (str) – The name assigned to this object on the context.
- **span** (baseplate.Span) – The current span this object is being made for.

Return type Any

To integrate with *configure_context()* for maximum convenience, make a parser that implements *baseplate.lib.config.Parser* and returns your *ContextFactory*.

```
class MyClient(config.Parser):
   def parse(
       self, key_path: str, raw_config: config.RawConfig
   ) -> "MyContextFactory":
      parser = config.SpecParser(
         {
            "foo": config.Integer(),
            "bar": config.Boolean(),
         }
      )
      result = parser.parse(key_path, raw_config)
      return MyContextFactory(foo=result.foo, bar=result.bar)
```

baseplate.frameworks

Helpers for integration with various application frameworks.

This package contains modules which integrate Baseplate with common application frameworks.

Baseplate.py provides integrations with common Python application frameworks. These integrations automatically manage the *ServerSpan* lifecycle for each unit of work the framework processes (requests or messages).

baseplate.frameworks.thrift

Thrift is a cross-language framework for cross-service communication. Developers write a language-independent definition of a service's API (the "IDL") and Thrift's code generator makes server and client libraries for that API.

This module provides a wrapper for a TProcessor which integrates Baseplate's facilities into the Thrift request lifecycle.

An abbreviated example of it in use:

```
logger = logging.getLogger(__name__)


def make_processor(app_config):
    baseplate = Baseplate(app_config)

    handler = MyHandler()
    processor = my_thrift.MyService.Processor(handler)
    return baseplateify_processor(processor, logger, baseplate)
```

baseplate.frameworks.thrift.**baseplateify_processor**(*processor*, *logger*, *baseplate*, *edge_context_factory=None*)

Wrap a Thrift Processor with Baseplate's span lifecycle.

Parameters

- **processor** (TProcessor) – The service's processor to wrap.

- **logger** (Logger) – The logger to use for error and debug logging.
- **baseplate** (*Baseplate*) – The baseplate instance for your application.
- **edge_context_factory** (Optional[*EdgeContextFactory*]) – A configured factory for handling edge request context.

Return type TProcessor

baseplate.frameworks.pyramid

Pyramid is a mature web framework for Python that we build HTTP services with.

This module provides a configuration extension for Pyramid which integrates Baseplate's facilities into the Pyramid WSGI request lifecycle.

An abbreviated example of it in use:

```
def make_app(app_config):
    configurator = Configurator()

    baseplate = Baseplate(app_config)
    baseplate_config = BaseplateConfigurator(
        baseplate,
        trust_trace_headers=True,
    )
    configurator.include(baseplate_config.includeme)

    return configurator.make_wsgi_app()
```

Warning: Because of how Baseplate instruments Pyramid, you should not make an exception view prevent Baseplate from seeing the unhandled error and reporting it appropriately.

class baseplate.frameworks.pyramid.**BaseplateConfigurator**(*baseplate*, *edge_context_factory=None*, *header_trust_handler=None*)

Configuration extension to integrate Baseplate into Pyramid.

Parameters

- **baseplate** (*Baseplate*) – The Baseplate instance for your application.
- **edge_context_factory** (Optional[*EdgeContextFactory*]) – A configured factory for handling edge request context.
- **header_trust_handler** (Optional[*HeaderTrustHandler*]) – An object which will be used to verify whether baseplate should parse the request context headers, for example trace ids. See StaticTrustHandler for the default implementation.

class baseplate.frameworks.pyramid.**HeaderTrustHandler**

Abstract class used by *BaseplateConfigurator* to validate headers.

See *StaticTrustHandler* for the default implementation.

should_trust_trace_headers(*request*)

Return whether baseplate should parse the trace headers from the inbound request.

Parameters **request** (Request) – The request

Return type bool

Returns Whether baseplate should parse the trace headers from the inbound request.

should_trust_edge_context_payload(*request*)
Return whether baseplate should trust the edge context headers from the inbound request.

Parameters **request** (Request) – The request

Return type bool

Returns Whether baseplate should trust the inbound edge context headers

class baseplate.frameworks.pyramid.**StaticTrustHandler**(*trust_headers=False*)
Default implementation for handling headers.

This class is created automatically by BaseplateConfigurator unless you supply your own HeaderTrustHandler

Parameters **trust_headers** (bool) – Whether or not to trust trace and edge context headers from inbound requests. This value will be returned by should_trust_trace_headers and should_trust_edge_context_payload.

Warning: Do not set `trust_headers` to `True` unless you are sure your application is only accessible by trusted sources (usually backend-only services).

should_trust_trace_headers(*request*)
Return whether baseplate should parse the trace headers from the inbound request.

Parameters **request** (Request) – The request

Return type bool

Returns Whether baseplate should parse the trace headers from the inbound request.

should_trust_edge_context_payload(*request*)
Return whether baseplate should trust the edge context headers from the inbound request.

Parameters **request** (Request) – The request

Return type bool

Returns Whether baseplate should trust the inbound edge context headers

Events

Within its Pyramid integration, Baseplate will emit events at various stages of the request lifecycle that services can hook into.

class baseplate.frameworks.pyramid.**ServerSpanInitialized**(*request*)
Event that Baseplate fires after creating the ServerSpan for a Request.

This event will be emitted before the Request is passed along to it's handler. Baseplate initializes the ServerSpan in response to a pyramid.events.ContextFound event emitted by Pyramid so while we can guarantee what Baseplate has done when this event is emitted, we cannot guarantee that any other subscribers to pyramid.events.ContextFound have been called or not.

Health Checker Helper

This module also provides a helper function to extract the health check probe used by the builtin healthchecker out of the request.

baseplate.frameworks.pyramid.**get_is_healthy_probe**(*request*)

Get the thrift enum value of the probe used in http is_healthy request.

Return type int

baseplate.frameworks.queue_consumer

baseplate.frameworks.queue_consumer.kafka

This module provides a `QueueConsumerFactory` that allows you to run a `QueueConsumerServer` that integrates Baseplate's facilities with Kafka.

An abbreviated example of it in use:

```
import confluent_kafka
from baseplate import RequestContext
from typing import Any
from typing import Dict

def process_links(
    context: RequestContext,
    data: Dict[str, Any],
    message: confluent_kafka.Message,
):
    print(f"processing {data}")

def make_consumer_factory(app_config):
    baseplate = Baseplate(app_config)
    return InOrderConsumerFactory.new(
        name="kafka_consumer.link_consumer_v0",
        baseplate=baseplate,
        bootstrap_servers="127.0.0.1:9092",
        group_id="service.link_consumer",
        topics=["new_links", "edited_links"],
        handler_fn=process_links,
    )
```

This will create a Kafka consumer group named `'service.link_consumer'` that consumes from the topics `'new_links'` and `'edited_links'`. Messages read from those topics will be fed to `process_links`.

Factory

class baseplate.frameworks.queue_consumer.kafka.**InOrderConsumerFactory**(*name*, *baseplate*, *consumer*, *handler_fn*, *kafka_consume_batch_size=*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)

Factory for running a `QueueConsumerServer` using Kafka.

The *InOrderConsumerFactory* attempts to achieve in order, exactly once message processing.

This will run a single *KafkaConsumerWorker* that reads messages from Kafka and puts them into an internal work queue. Then it will run a single *KafkaMessageHandler* that reads messages from the internal work queue, processes them with the *handler_fn*, and then commits each message's offset to Kafka.

This one-at-a-time, in-order processing ensures that when a failure happens during processing we don't commit its offset (or the offset of any later messages) and that when the server restarts it will receive the failed message and attempt to process it again. Additionally, because each message's offset is committed immediately after processing we should never process a message more than once.

For most cases where you just need a basic consumer with sensible defaults you can use *InOrderConsumerFactory.new*.

If you need more control, you can create the `Consumer` yourself and use the constructor directly.

classmethod new(*name*, *baseplate*, *bootstrap_servers*, *group_id*, *topics*, *handler_fn*, *kafka_consume_batch_size=1*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)

Return a new *_BaseKafkaQueueConsumerFactory*.

This method will create the `Consumer` for you and is appropriate to use in most cases where you just need a basic consumer with sensible defaults.

This method will also enforce naming standards for the Kafka consumer group and the baseplate server span.

Parameters

- **name** (`str`) – A name for your consumer process. Must look like "kafka_consumer.{group_name}"
- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **bootstrap_servers** (`str`) – A comma delimited string of kafka brokers.
- **group_id** (`str`) – The kafka consumer group id. Must look like "{service_name}.{group_name}" to help prevent collisions between services.
- **topics** (`Sequence[str]`) – An iterable of kafka topics to consume from.
- **handler_fn** (`Callable[[RequestContext, Any, Message], None]`) – A *baseplate.frameworks.queue_consumer.kafka.Handler* function that will process an individual message.

- **kafka_consume_batch_size** (int) – The number of messages the *KafkaConsumerWorker* reads from Kafka in each batch. Defaults to 1.
- **message_unpack_fn** (Callable[[bytes], Any]) – A function that takes one argument, the *bytes* message body and returns the message in the format the handler expects. Defaults to *json.loads*.
- **health_check_fn** (Optional[Callable[[Dict[str, Any]], bool]]) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.

Return type _BaseKafkaQueueConsumerFactory

__init__(*name*, *baseplate*, *consumer*, *handler_fn*, *kafka_consume_batch_size=1*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)
_BaseKafkaQueueConsumerFactory constructor.

Parameters

- **name** (str) – A name for your consumer process. Must look like "kafka_consumer.{group_name}"
- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **consumer** (Consumer) – An instance of Consumer.
- **handler_fn** (Callable[[*RequestContext*, Any, Message], None]) – A *baseplate.frameworks.queue_consumer.kafka.Handler* function that will process an individual message.
- **kafka_consume_batch_size** (int) – The number of messages the *KafkaConsumerWorker* reads from Kafka in each batch. Defaults to 1.
- **message_unpack_fn** (Callable[[bytes], Any]) – A function that takes one argument, the *bytes* message body and returns the message in the format the handler expects. Defaults to *json.loads*.
- **health_check_fn** (Optional[Callable[[Dict[str, Any]], bool]]) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.

class baseplate.frameworks.queue_consumer.kafka.**FastConsumerFactory**(*name*, *baseplate*, *consumer*, *handler_fn*, *kafka_consume_batch_size=1*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)

Factory for running a QueueConsumerServer using Kafka.

The *FastConsumerFactory* prioritizes high throughput over exactly once message processing.

This will run a single *KafkaConsumerWorker* that reads messages from Kafka and puts them into an internal work queue. Then it will run multiple *KafkaMessageHandler`s that read messages from the internal work queue, processes them with the `handler_fn*. The number of *KafkaMessageHandler* processes is controlled by the *max_concurrency* parameter in the *~baseplate.server.queue_consumer.QueueConsumerServer* configuration. Kafka partition offsets are automatically committed by the *confluent_kafka.Consumer* every 5 seconds,

so any message that has been read by the *KafkaConsumerWorker* could be committed, regardless of whether it has been processed.

This server should be able to achieve very high message processing throughput due to the multiple *KafkaMessageHandler* processes and less frequent, background partition offset commits. This does come at a price though: messages may be processed out of order, not at all, or multiple times. This is appropriate when processing throughput is important and it's acceptable to skip messages or process messages more than once (maybe there is ratelimiting in the handler or somewhere downstream).

Messages processed out of order: Messages are added to the internal work queue in order, but one worker may finish processing a "later" message before another worker finishes processing an "earlier" message.

Messages never processed: If the server crashes it may not have processed some messages that have already had their offsets automatically committed. When the server restarts it won't read those messages.

Messages processed more than once: If the server crashes it may have processed some messages but not yet committed their offsets. When the server restarts it will reprocess those messages.

For most cases where you just need a basic consumer with sensible defaults you can use *FastConsumerFactory.new*.

If you need more control, you can create the `Consumer` yourself and use the constructor directly.

classmethod new(*name*, *baseplate*, *bootstrap_servers*, *group_id*, *topics*, *handler_fn*, *kafka_consume_batch_size=1*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)

Return a new *_BaseKafkaQueueConsumerFactory*.

This method will create the `Consumer` for you and is appropriate to use in most cases where you just need a basic consumer with sensible defaults.

This method will also enforce naming standards for the Kafka consumer group and the baseplate server span.

Parameters

- **name** (`str`) – A name for your consumer process. Must look like "kafka_consumer.{group_name}"
- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **bootstrap_servers** (`str`) – A comma delimited string of kafka brokers.
- **group_id** (`str`) – The kafka consumer group id. Must look like "{service_name}.{group_name}" to help prevent collisions between services.
- **topics** (`Sequence[str]`) – An iterable of kafka topics to consume from.
- **handler_fn** (`Callable[[RequestContext, Any, Message], None]`) – A *baseplate.frameworks.queue_consumer.kafka.Handler* function that will process an individual message.
- **kafka_consume_batch_size** (`int`) – The number of messages the *KafkaConsumerWorker* reads from Kafka in each batch. Defaults to 1.
- **message_unpack_fn** (`Callable[[bytes], Any]`) – A function that takes one argument, the *bytes* message body and returns the message in the format the handler expects. Defaults to *json.loads*.
- **health_check_fn** (`Optional[Callable[[Dict[str, Any]], bool]]`) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.

Return type `_BaseKafkaQueueConsumerFactory`

__init__(*name*, *baseplate*, *consumer*, *handler_fn*, *kafka_consume_batch_size=1*, *message_unpack_fn=<function loads>*, *health_check_fn=None*)
_BaseKafkaQueueConsumerFactory constructor.

Parameters

- **name** (str) – A name for your consumer process. Must look like "kafka_consumer.{group_name}"
- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **consumer** (Consumer) – An instance of Consumer.
- **handler_fn** (Callable[[*RequestContext*, Any, Message], None]) – A *baseplate.frameworks.queue_consumer.kafka.Handler* function that will process an individual message.
- **kafka_consume_batch_size** (int) – The number of messages the *KafkaConsumerWorker* reads from Kafka in each batch. Defaults to 1.
- **message_unpack_fn** (Callable[[bytes], Any]) – A function that takes one argument, the *bytes* message body and returns the message in the format the handler expects. Defaults to *json.loads*.
- **health_check_fn** (Optional[Callable[[Dict[str, Any]], bool]]) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.

baseplate.frameworks.queue_consumer.kombu

Kombu is a library for interacting with queue brokers.

This module provides a QueueConsumerFactory that allows you to run a QueueConsumerServer that integrates Baseplate's facilities with Kombu.

An abbreviated example of it in use:

```
import kombu
from baseplate import RequestContext
from typing import Any

def process_links(
    context: RequestContext,
    body: Any,
    message: kombu.Message,
):
    print(f"processing {body}")

def make_consumer_factory(app_config):
    baseplate = Baseplate(app_config)
    exchange = Exchange("reddit_exchange", "direct")
    connection = Connection(
      hostname="amqp://guest:guest@reddit.local:5672",
      virtual_host="/",
    )
    queue_name = "process_links_q"
    routing_keys = ["link_created"]
    return KombuQueueConsumerFactory.new(
        baseplate=baseplate,
```

(continues on next page)

(continued from previous page)

```
        exchange=exchange,
        connection=connection,
        queue_name=queue_name,
        routing_keys=routing_keys,
        handler_fn=process_links,
    )
```

This will create a queue named `'process_links_q'` and bind the routing key `'link_created'`. It will then register a consumer for `'process_links_q'` to read messages and feed them to `process_links`.

Factory

class baseplate.frameworks.queue_consumer.kombu.**KombuQueueConsumerFactory**(*baseplate*, *name*, *connection*, *queues*, *handler_fn*, *error_handler_fn=None*, *health_check_fn=None*, *serializer=None*, *worker_kwargs=None*)

Factory for running a `QueueConsumerServer` using Kombu.

For simple cases where you just need a basic queue with all the default parameters for your message broker, you can use *KombuQueueConsumerFactory.new*.

If you need more control, you can create the `Queue` s yourself and use the constructor directly.

classmethod new(*baseplate*, *exchange*, *connection*, *queue_name*, *routing_keys*, *handler_fn*, *error_handler_fn=None*, *health_check_fn=None*, *serializer=None*, *worker_kwargs=None*)

Return a new *KombuQueueConsumerFactory*.

This method will create the `Queue` s for you and is appropriate to use in simple cases where you just need a basic queue with all the default parameters for your message broker.

Parameters

- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **exchange** (Exchange) – The *kombu.Exchange* that you will bind your `Queue` s to.
- **exchange** – The *kombu.Connection* to your message broker.
- **queue_name** (str) – Name for your queue.
- **routing_keys** (Sequence[str]) – List of routing keys that you will create `Queue` s to consume from.
- **handler_fn** (Callable[[*RequestContext*, Any, Message], None]) – A function that will process an individual message from a queue.

- **error_handler_fn** (Optional[Callable[[*RequestContext*, Any, Message, Exception], None]]) – A function that will be called when an error is thrown while executing the *handler_fn*. This function will be responsible for calling *message.ack* or *message.requeue* as it will not be automatically called by *KombuMessageHandler*'s *handle* function.
- **health_check_fn** (Optional[Callable[[Dict[str, Any]], bool]]) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.
- **serializer** (Optional[*KombuSerializer*]) – A *baseplate.clients.kombu.KombuSerializer* that should be used to decode the messages you are consuming.
- **worker_kwargs** (Optional[Dict[str, Any]]) – A dictionary of keyword arguments used to configure a queue consumer.

Return type *KombuQueueConsumerFactory*

__init__(*baseplate*, *name*, *connection*, *queues*, *handler_fn*, *error_handler_fn=None*, *health_check_fn=None*, *serializer=None*, *worker_kwargs=None*)
KombuQueueConsumerFactory constructor.

Parameters

- **baseplate** (*Baseplate*) – The Baseplate set up for your consumer.
- **exchange** – The *kombu.Exchange* that you will bind your Queue s to.
- **queues** (Sequence[Queue]) – List of Queue s to consume from.
- **queue_name** – Name for your queue.
- **routing_keys** – List of routing keys that you will create Queue s to consume from.
- **handler_fn** (Callable[[*RequestContext*, Any, Message], None]) – A function that will process an individual message from a queue.
- **error_handler_fn** (Optional[Callable[[*RequestContext*, Any, Message, Exception], None]]) – A function that will be called when an error is thrown while executing the *handler_fn*. This function will be responsible for calling *message.ack* or *message.requeue* as it will not be automatically called by *KombuMessageHandler*'s *handle* function.
- **health_check_fn** (Optional[Callable[[Dict[str, Any]], bool]]) – A *baseplate.server.queue_consumer.HealthcheckCallback* function that can be used to customize your health check.
- **serializer** (Optional[*KombuSerializer*]) – A *baseplate.clients.kombu.KombuSerializer* that should be used to decode the messages you are consuming.
- **worker_kwargs** (Optional[Dict[str, Any]]) – A dictionary of keyword arguments used to create queue consumers.

Errors

class baseplate.frameworks.queue_consumer.kombu.**FatalMessageHandlerError**

An error that signals that the queue process should exit.

Raising an Exception that is a subclass of FatalMessageHandlerError will cause the KombuMessageHandler to re-raise the exception rather than swallowing it which will cause the handler thread/process to stop. This, in turn, will gracefully shut down the QueueConsumerServer currently running.

Exceptions of this nature should be reserved for errors that are due to problems with the environment rather than the message itself. For example, a node that cannot get its AWS credentials.

baseplate.observers

Observers watch Baseplate for events that happen during requests, such as requests starting and ending and service calls being made. Observers can also add attributes to the *RequestContext* for your application to use during the request.

To enable observers, call *configure_observers()* on your *Baseplate* object during application startup and supply the application configuration. See each observer below for what configuration options are available.

```
def make_wsgi_app(app_config):
    baseplate = Baseplate(app_config)
    baseplate.configure_observers()

    ...
```

Logging

The logging observer adds request-specific metadata to log lines coming out of your application.

Changed in version 1.4: Logs are now formatted as JSON objects.

Configuration

No configuration is necessary, this observer is always enabled when you call *configure_observers()*.

If your application is run with **baseplate-serve**, logging can be controlled with Python's standard logging configuration. See *Logging* for more information.

Outputs

The **baseplate-serve** command will automatically set up log formatting.

When directly used from a TTY, a simplified human-readable message format is emitted. Only the level and message are included.

In production usage, log entries are formatted as JSON objects that can be parsed automatically by log analysis systems. Log entry objects contain the following keys:

message The message.

level The name of the log level at which the log entry was generated, e.g. INFO, WARNING, etc.

Along with name, this can be useful for *configuring logging to squelch noisy messages*.

name The name of the `Logger` used.

Along with `level`, this can be useful for *configuring logging to squelch noisy messages*.

traceID The Trace ID of the request within context of which the log entry was generated. This can be used to correlate log entries from the same root request within and across services.

Only present if the log entry was generated during a request. Otherwise see `thread`.

pathname The path to the Python source for the module that generated the log entry.

module The name of the module in which the log entry was generated.

funcName The name of the function that generated the log entry.

lineno The line number on which the log entry was generated.

process The OS-level Process ID of the process that generated the log entry.

processName The name of the process that generated the log entry (as set on `multiprocessing.current_process().name`).

thread The name of the thread that generated the log entry (as set on `threading.current_thread().name`).

This may be absent if the log entry was generated from within processing of a request, in which case `traceID` will be included instead.

Older logging

Before v1.4, log entries were written in a custom format:

```
17905:7296338476964580186:baseplate.lib.metrics:DEBUG:Blah blah
^     ^                   ^                     ^     ^
|     |                   |                     |     Log message
|     |                   |                     Log level
|     |                   Name of the logger
|     Trace ID of the request
Process ID
```

Direct Use

Any log messages emitted with the Python standard `logging` interfaces will be annotated by this observer.

StatsD Metrics

The metrics observer emits StatsD-compatible time-series metrics about the performance of your application. These metrics are useful to get a cross-sectional view of how your application is performing in a broad sense.

Configuration

Make sure your service calls *configure_observers()* during application startup and then add the following to your configuration file to enable and configure the StatsD metrics observer.

```
[app:main]

...


# required: the prefix added to all metrics emitted.
# if present, the observer is enabled.
metrics.namespace = myservice

# optional: an endpoint to send the metrics datagrams to.
# if not specified, metrics will only be emitted to debug logs.
metrics.endpoint = statsd.local:8125

# optional: the percent of statsd metrics to sample
# if not specified, it will default to 100% (all metrics sent)
# config must be passed to the `Baseplate` constructor to use this option
metrics_observer.sample_rate = 100%

...
```

Outputs

For each span in the application, the metrics observer emits a *Timer* tracking how long the span took and increments a *Counter* for success or failure of the span (failure being an unexpected exception).

For the *ServerSpan* representing the request the server is handling, the timer has a name like `{namespace}.server.{route_or_method_name}` and the counter looks like `{namespace}.server.{route_or_method_name}.{success,failure}`. If the request *timed out* an additional counter will be emitted with path `{namespace}.server.{route_or_method_name}.timed_out`.

For each span representing a call to a remote service or database, the timer has a name like `{namespace}.clients.{context_name}.{method}` and the counter `{namespace}.clients.{context_name}.{method}.{success,failure}` where `context_name` is the name of the client in the context configuration.

Calls to *incr_tag()* will increment a counter like `{namespace}.{tag_name}` by the amount specified.

When using **baseplate-serve**, various process-level runtime metrics will also be emitted. These are not tied to individual requests but instead give insight into how the whole application is functioning. See *Process-level metrics* for more information.

Direct Use

When enabled, the metrics observer also adds a *Client* object as an attribute named `metrics` to the *RequestContext*:

```
def my_handler(request):
    request.metrics.counter("foo").increment()
```

To keep your application more generic, it's better to use local spans for custom local timers and *incr_tag()* for custom counters.

StatsD Tagged Metrics

The tagged metrics observer emits StatsD-compatible time-series metrics about the performance of your application with tags in the InfluxStatsD format. The tags added to the metrics are configurable: any tags that pass through the *set_tag()* function are filtered through a user-supplied allowlist in the configuration file.

Configuration

Make sure your service calls *configure_observers()* during application startup and then add the following to your configuration file to enable and configure the StatsD tagged metrics observer.

```
[app:main]

...

# required to enable observer
metrics.tagging = true

# optional: which span tags should be attached to metrics. see below.
#
# `endpoint` and `client` are always allowed
metrics.allowlist = foo, bar, baz

# optional: the percent of statsd metrics to sample.
#
# if not specified, it will default to 100% (all metrics sent)
metrics_observer.sample_rate = 100%

...
```

Tag Allowlist

Wavefront supports a maximum of 20 tags per cluster and 1000 distinct time series per metric. Baseplate integrations of frameworks come out of the box with some default tags set via *set_tag()*, but to append them to the metrics they must be present in the configuration file via `metrics.allowlist`.

In order to find these tags to put in the allowlist, look through the code base for calls to *set_tag()* or check a zipkin trace in Wavefront to see all the tags on a span.

Outputs

For each span in the application, the metrics observer emits a *Timer* tracking how long the span took and increments a *Counter* for success or failure of the span (failure being an unexpected exception).

A key differentiation from the *untagged StatsD metrics observer* is that the emitted outputs from baseplate no longer contain a namespace prefix. Prepending the namespace must be configured in Telegraf via the `name_prefix` input plugin configuration.

For the *ServerSpan* representing the request the server is handling, the timer has a name like `baseplate.server.latency,endpoint={route_or_method_name}` and the counter looks like `baseplate.server.rate,success={True,False},endpoint={route_or_method_name}`.

For each span representing a call to a remote service or database, the timer has a name like `baseplate.clients.latency,client={name},endpoint={method}` and the counter `baseplate.clients.rate,client={name},endpoint={method},success={True,False}`.

When using **`baseplate-serve`**, various process-level runtime metrics will also be emitted. These are not tied to individual requests but instead give insight into how the whole application is functioning. See *Process-level metrics* for more information.

Direct Use

When enabled, the metrics observer also adds a *`Client`* object as an attribute named `metrics` to the *`RequestContext`* which can take an optional tags parameter in the form of a `dict`:

```
def my_handler(request):
    request.metrics.counter("foo", {"bar": "baz"}).increment()
```

To keep your application more generic, it's better to use local spans for custom local timers and *`incr_tag()`* for custom counters.

Sentry (Crash Reporting)

The Sentry observer integrates sentry-sdk with your application to record tracebacks for crashes to Sentry.

Changed in version 2.0: The underlying library for communicating with sentry was changed from Raven to sentry-sdk.

Configuration

Make sure your service calls *`configure_observers()`* during application startup and then add the following to your configuration file to enable and configure the Sentry observer.

```
[app:main]

...

# required to enable the observer
# the DSN provided by Sentry for your project
sentry.dsn = https://decaf:face@sentry.local/123

# optional: the environment this application is running in
sentry.environment = staging

# optional: percent chance that a given error will be reported
# (defaults to 100%)
sentry.sample_rate = 37%

# optional: comma-delimited list of fully qualified names of exception
# classes to not report.
sentry.ignore_errors = my_service.UninterestingException

...
```

Outputs

Any unexpected exceptions that cause the request to crash (including outside request context) will be reported to Sentry. The Trace ID of the current request will be included in the context reported to Sentry.

Direct Use

When enabled, the error reporting observer also adds a `sentry_sdk.Hub` object as an attribute named `sentry` to the *`RequestContext`*:

```
def my_handler(request):
    try:
        ...
    except Exception:
        request.sentry.capture_exception()
```

Tracing

The tracing observer reports span information to a Zipkin-compatible distributed trace aggregator. This can be used to build up cross-service views of request processing across many services.

See the OpenTracing overview for more info on what tracing is and how it is helpful to you.

Configuration

Make sure your service calls *`configure_observers()`* during application startup and then add the following to your configuration file to enable and configure the tracing observer.

```
[app:main]

...

# required to enable observer
# the name of the service reporting traces
tracing.service_name = my_service

# optional: traces won't be reported if not set
# the name of the POSIX queue the trace publisher sidecar
# is listening on
tracing.queue_name = some-queue

# optional: what percent of requests to report spans for
# (defaults to 10%). note: this only kicks in if sampling
# has not already been determined by an upstream service.
tracing.sample_rate = 10%

...
```

Server Timeouts

The timeout observer ends processing of requests in your service if they take too long. This is particularly important when an upstream service times out on its end and retries requests to your services which will cause a pileup.

This is entirely configured in-service at the moment and no headers from upstream services are yet taken into account.

> **Warning:** The timeout mechanism is entirely cooperative. If request processing is taking a long time because it is doing compute-heavy actions and not yielding to the event loop it might go on longer than the allotted timeout.

New in version 1.2.

Changed in version 1.3.3: The default timeout was changed from 10 seconds to no timeout. Having a default timeout was confusing and broke jobs like crons.

Configuration

Make sure your service calls *configure_observers()* during application startup. By default, requests will not time out unless you configure them. The following configuration settings allow you to customize this.

```
[app:main]

...

# optional: defaults to no timeout if not specified. this timeout
# is used for any endpoint not specified in the by_endpoint
# section below.
# note: leaving this unconfigured is deprecated.
# can be set to 'infinite' to disable the timeout altogether.
server_timeout.default = 200 milliseconds

# optional: defaults to false. if enabled, tracebacks will be
# printed to the logs when timeouts occur.
server_timeout.debug = true

# optional: timeout values for specific endpoints. the name
# used must match the name of the server span generated.
# this overrides the default timeout.
# - thrift services: the name of the thrift RPC method
# - pyramid services: the name of the route (config.add_route)
# can be set to 'infinite' to disable the timeout altogether.
server_timeout.by_endpoint.is_healthy = 300 milliseoncds
server_timeout.by_endpoint.my_method = 12 seconds

...
```

Outputs

When a request times out, Baseplate.py will end the greenlet processing that request and emit some diagnostics:

- A log entry like `Server timed out processing for 'is_healthy' after 0.30 seconds`. If `server_timeout.debug` was configured to `True`, the full stack trace of the place the greenlet timed out will also be included.
- A *counter metric* named `{namespace}.server.{route_or_method_name}.timed_out`.
- A tag on the *server span sent to distributed tracing* indicating `timed_out=True`.

1.3.2 The Library

Baseplate also provides a collection of "extra batteries". These independent modules provide commonly needed functionality to applications. They can be used separately from the rest of Baseplate.

baseplate.lib.config

Configuration parsing and validation.

This module provides `parse_config` which turns a dictionary of stringy keys and values into a structured and typed configuration object.

For example, an INI file like the following:

```
[app:main]
simple = true
cards = clubs, spades, diamonds
nested.once = 1
nested.really.deep = 3 seconds
some_file = /var/lib/whatever.txt
sample_rate = 37.1%
interval = 30 seconds
default_from_env = blah
```

Might be parsed like the following. Note: when running under the baseplate server, The `config_parser.items(...)` step is taken care of for you and `raw_config` is passed as the only argument to your factory function.

```
>>> raw_config = dict(config_parser.items("app:main"))

>>> CARDS = config.OneOf(clubs=1, spades=2, diamonds=3, hearts=4)
>>> cfg = config.parse_config(raw_config, {
...     "simple": config.Boolean,
...     "cards": config.TupleOf(CARDS),
...     "nested": {
...         "once": config.Integer,
...
...         "really": {
...             "deep": config.Timespan,
...         },
...     },
...     "some_file": config.File(mode="r"),
...     "optional": config.Optional(config.Integer, default=9001),
...     "sample_rate": config.Percent,
...     "interval": config.Fallback(config.Timespan, config.Integer),
```

(continues on next page)

(continued from previous page)

```
...     "default_from_env": config.DefaultFromEnv(config.String, env_var_name),
... })

>>> print(cfg.simple)
True

>>> print(cfg.cards)
[1, 2, 3]

>>> print(cfg.nested.really.deep)
0:00:03

>>> cfg.some_file.read()
'cool'

>>> cfg.some_file.close()

>>> cfg.sample_rate
0.371

>>> print(cfg.interval)
0:00:30

>>> print(cfg.default_from_env)
blah
```

Parser

baseplate.lib.config.**parse_config**(*config*, *spec*)

Parse options against a spec and return a structured representation.

Parameters

- **config** (Dict[str, str]) – The raw stringy configuration dictionary.
- **spec** (Dict[str, Union[*Parser*, Dict[str, Any], Callable[[str], ~T]]]) – A specification of what the configuration should look like.

Raises *ConfigurationError* The configuration violated the spec.

Return type ConfigNamespace

Returns A structured configuration object.

Value Types

Each option can have a type specified. Some types compose with other types to make complicated expressions.

baseplate.lib.config.**String**(*text*)

A raw string.

Return type str

baseplate.lib.config.**Float**(*text*)

A floating-point number.

Return type float

baseplate.lib.config.**Integer**(*text=None*, *base=10*)
An integer.

To prevent mistakes, this will raise an error if the user attempts to configure a non-whole number.

Parameters **base** (int) – (Optional) If specified, the base of the integer to parse.

Return type Union[int, Callable[[str], int]]

baseplate.lib.config.**Boolean**(*text*)
True or False, case insensitive.

Return type bool

baseplate.lib.config.**Endpoint**(*text*)
A remote endpoint to connect to.

Returns an *EndpointConfiguration*.

If the endpoint is a hostname:port pair, the `family` will be `socket.AF_INET` and `address` will be a two-tuple of host and port, as expected by `socket`.

If the endpoint contains a slash (`/`), it will be interpreted as a path to a UNIX domain socket. The `family` will be `socket.AF_UNIX` and `address` will be the path as a string.

Return type *EndpointConfiguration*

baseplate.lib.config.**Timespan**(*text*)
A span of time.

This takes a string of the form "1 second" or "3 days" and returns a `datetime.timedelta` representing that span of time.

Units supported are: milliseconds, seconds, minutes, hours, days.

Return type timedelta

baseplate.lib.config.**Base64**(*text*)
A base64 encoded block of data.

This is useful for arbitrary binary blobs.

Return type bytes

baseplate.lib.config.**File**(*mode='r'*)
A path to a file.

This takes a path to a file and returns an open file object, like returned by `open()`.

Parameters **mode** (str) – an optional string that specifies the mode in which the file is opened.

Return type Callable[[str], IO]

baseplate.lib.config.**Percent**(*text*)
A percentage.

This takes a string of the form "37.2%" or "44%" and returns a float in the range [0.0, 1.0].

Return type float

baseplate.lib.config.**UnixUser**(*text*)
A Unix user name or decimal ID.

The parsed value will be the integer user ID.

Return type int

baseplate.lib.config.**UnixGroup**(*text*)
A Unix group name or decimal ID.

The parsed value will be the integer group ID.

Return type int

baseplate.lib.config.**OneOf**(**options*)
One of several choices.

For each `option`, the name is what should be in the configuration file and the value is what it is mapped to.

For example:

```
OneOf(hearts="H", spades="S")
```

would parse:

```
"hearts"
```

into:

```
"H"
```

Return type Callable[[str], ~T]

baseplate.lib.config.**TupleOf**(*item_parser*)
A comma-delimited list of type T.

At least one value must be provided. If you want an empty list to be a valid choice, wrap with *Optional()*.

Return type Callable[[str], Sequence[~T]]

If you need something custom or fancy for your application, just use a callable which takes a string and returns the parsed value or raises ValueError.

Combining Types

These options are used in combination with other types to form more complex configurations.

baseplate.lib.config.**Optional**(*item_parser*, *default=None*)
An option of type T, or `default` if not configured.

Return type Callable[[str], Optional[~T]]

baseplate.lib.config.**Fallback**(*primary_parser*, *fallback_parser*)
An option of type T1, or if that fails to parse, of type T2.

This is useful for backwards-compatible configuration changes.

Return type Callable[[str], ~T]

baseplate.lib.config.**DictOf**(*spec*)
A group of options of a given type.

This is useful for providing data to the application without the application having to know ahead of time all of the possible keys.

```
[app:main]
population.cn = 1383890000
population.in = 1317610000
population.us = 325165000
population.id = 263447000
population.br = 207645000
```

```
>>> cfg = config.parse_config(raw_config, {
...     "population": config.DictOf(config.Integer),
... })

>>> len(cfg.population)
5

>>> cfg.population["br"]
207645000
```

It can also be combined with other configuration specs or parsers to parse more complicated structures:

```
[app:main]
countries.cn.population = 1383890000
countries.cn.capital = Beijing
countries.in.population = 1317610000
countries.in.capital = New Delhi
countries.us.population = 325165000
countries.us.capital = Washington D.C.
countries.id.population = 263447000
countries.id.capital = Jakarta
countries.br.population = 207645000
countries.br.capital = Brasília
```

```
>>> cfg = config.parse_config(raw_config, {
...     "countries": config.DictOf({
...         "population": config.Integer,
...         "capital": config.String,
...     }),
... })

>>> len(cfg.countries)
5

>>> cfg.countries["cn"].capital
'Beijing'

>>> cfg.countries["id"].population
263447000
```

Data Types

class baseplate.lib.config.**EndpointConfiguration**(*family: socket.AddressFamily*, *address: Union[baseplate.lib.config.InternetAddress, str]*)

A description of a remote endpoint.

This is a 2-tuple of (`family` and `address`).

family One of `socket.AF_INET` or `socket.AF_UNIX`.

address An address appropriate for the `family`.

See also:

baseplate.lib.config.Endpoint()

Add a new parser

class baseplate.lib.config.**Parser**(**args*, ***kwds*)

Base class for configuration parsers.

Exceptions

exception baseplate.lib.config.**ConfigurationError**(*key*, *error*)

Raised when the configuration violates the spec.

baseplate.lib.crypto

Utilities for common cryptographic operations.

```
message = "Hello, world!"

secret = secrets.get_versioned("some_signing_key")
signature = make_signature(
    secret, message, max_age=datetime.timedelta(days=1))

try:
    validate_signature(secret, message, signature)
except SignatureError:
    print("Oh no, it was invalid!")
else:
    print("Message was valid!")
```

```
Message was valid!
```

Message Signing

baseplate.lib.crypto.**make_signature**(*secret*, *message*, *max_age*)

Return a signature for the given message.

To ensure that key rotation works automatically, always fetch the secret token from the secret store immediately before use and do not cache / save the token anywhere. The `current` version of the secret will be used to sign the token.

Parameters

- **secret** (*VersionedSecret*) – The secret signing key from the secret store.
- **message** (`str`) – The message to sign.
- **max_age** (`timedelta`) – The amount of time in the future the signature will be valid for.

Return type `bytes`

Returns An encoded signature.

baseplate.lib.crypto.**validate_signature**(*secret*, *message*, *signature*)

Validate and assert a message's signature is correct.

If the signature is valid, the function will return normally with a *SignatureInfo* with some details about the signature. Otherwise, an exception will be raised.

To ensure that key rotation works automatically, always fetch the secret token from the secret store immediately before use and do not cache / save the token anywhere. All active versions of the secret will be checked when validating the signature.

Parameters

- **secret** (*VersionedSecret*) – The secret signing key from the secret store.
- **message** (`str`) – The message payload to validate.
- **signature** (`bytes`) – The signature supplied with the message.

Raises *UnreadableSignatureError* The signature is corrupt.

Raises *IncorrectSignatureError* The digest is incorrect.

Raises *ExpiredSignatureError* The signature expired.

Return type *SignatureInfo*

class baseplate.lib.crypto.**SignatureInfo**(*version: int*, *expiration: int*)

Information about a valid signature.

Variables

- **version** – The version of the packed signature format.
- **expiration** – The time, in seconds since the UNIX epoch, at which the signature will expire.

Exceptions

exception baseplate.lib.crypto.**SignatureError**
Base class for all message signing related errors.

exception baseplate.lib.crypto.**UnreadableSignatureError**
Raised when the signature is corrupt or wrongly formatted.

exception baseplate.lib.crypto.**IncorrectSignatureError**
Raised when the signature is readable but does not match the message.

exception baseplate.lib.crypto.**ExpiredSignatureError**(*expiration*)
Raised when the signature is valid but has expired.

The `expiration` attribute is the time (as seconds since the UNIX epoch) at which the signature expired.

baseplate.lib.datetime

Extensions to the standard library *datetime* module.

baseplate.lib.datetime.**datetime_to_epoch_milliseconds**(*dt*)
Convert datetime object to epoch milliseconds.

Return type int

baseplate.lib.datetime.**datetime_to_epoch_seconds**(*dt*)
Convert datetime object to epoch seconds.

Return type int

baseplate.lib.datetime.**epoch_milliseconds_to_datetime**(*ms*)
Convert epoch milliseconds to UTC datetime.

Return type datetime

baseplate.lib.datetime.**epoch_seconds_to_datetime**(*sec*)
Convert epoch seconds to UTC datetime.

Return type datetime

baseplate.lib.datetime.**get_utc_now**()
Get current offset-aware datetime which has timezone information.

Return type datetime

baseplate.lib.edgecontext

Services deep within the backend often need to know information about the client that originated the request, such as what user is authenticated or what country they're in. Baseplate services can get this information from the edge context which is automatically propagated along with calls between services.

Changed in version 2.0: The implementation built into Baseplate.py was extracted into its own library. See <https://reddit-edgecontext.readthedocs.io/en/latest/> for an example implementation.

class baseplate.lib.edgecontext.**EdgeContextFactory**
Abstract base for a factory that parses edge context data.

abstract from_upstream(*header_value*)
Parse a serialized edge context header from an inbound request.

Parameters **header_value** (Optional[bytes]) – The raw bytes of the header payload.

Return type Any

baseplate.lib.events

Client library for sending events to the data processing system.

This is for use with the event collector system. Events generally track something that happens in production that we want to instrument for planning and analytical purposes.

Events are serialized and put onto a message queue on the same server. These serialized events are then consumed and published to the remote event collector by a separate daemon.

Building Events

Thrift Schema v2 Events

For modern Thrift-based events: import the event schemas into your project, instantiate and fill out an event object, and pass it into the queue:

```
import time
import uuid

from baseplate import Baseplate
from baseplate.lib.events import EventQueue
from baseplate.lib.events import serialize_v2_event

from event_schemas.event.ttypes import Event


def my_handler(request):
    event = Event(
        source="baseplate",
        action="test",
        noun="baseplate",
        client_timestamp=time.time() * 1000,
        uuid=str(uuid.uuid4()),
    )
    request.events_v2.put(ev2)


def make_wsgi_app(app_config):
    ...

    baseplate = Baseplate(app_config)
    baseplate.configure_context(
        {
            ...

            "events_v2": EventQueue("v2", serialize_v2_events),

            ...
        }
    )

    ...
```

Queuing Events

class baseplate.lib.events.**EventQueue**(*name*, *event_serializer*)
A queue to transfer events to the publisher.

Parameters

- **name** (str) – The name of the event queue to send to. This specifies which publisher should send the events which can be useful for routing to different event pipelines (prod/test/v2 etc.).
- **event_serializer** (Callable[[~T], bytes]) – A callable that takes an event object and returns serialized bytes ready to send on the wire. See below for options.

put(*event*)
Add an event to the queue.

The queue is local to the server this code is run on. The event publisher on the server will take these events and send them to the collector.

Parameters **event** (*~T*) – The event to send. The type of event object passed in depends on the selected event_serializer.

Raises *EventTooLargeError* The serialized event is too large.

Raises *EventQueueFullError* The queue is full. Events are not being published fast enough.

Return type None

The EventQueue also implements *ContextFactory* so it can be used with *add_to_context()*:

```
event_queue = EventQueue("production", serialize_v2_event)
baseplate.add_to_context("events_production", event_queue)
```

It can then be used from the *RequestContext* during requests:

```
def some_service_method(self, context):
    event = Event(...)
    context.events_production.put(event)
```

Serializers

The event_serializer parameter to *EventQueue* is a callable which serializes a given event object. Baseplate comes with a serializer for the Thrift schema based V2 event system:

baseplate.lib.events.**serialize_v2_event**(*event*)
Serialize a Thrift struct to bytes for the V2 event protocol.

Parameters **event** (Any) – A Thrift struct from the event schemas.

Return type bytes

Exceptions

exception baseplate.lib.events.**EventError**
Base class for event related exceptions.

exception baseplate.lib.events.**EventTooLargeError**(*size*)
Raised when a serialized event is too large to send.

exception baseplate.lib.events.**EventQueueFullError**
Raised when the queue of events is full.

This usually indicates that the event publisher is having trouble talking to the event collector.

Publishing Events

Events that are put onto an *EventQueue* are consumed by a separate process and published to the remote event collector service. The publisher is in baseplate and can be run as follows:

```
$ python -m baseplate.sidecars.event_publisher --queue-name something config_file.ini
```

The publisher will look at the specified INI file to find its configuration. Given a queue name of `something` (as in the example above), it will expect a section in the INI file called `[event-publisher:something]` with content like below:

```
[event-publisher:something]
collector.hostname = some-domain.example.com

key.name = NameOfASecretKey
key.secret = Base64-encoded-blob-of-randomness

metrics.namespace = a.name.to.put.metrics.under
metrics.endpoint = the-statsd-host:1234
```

baseplate.lib.experiments

Changed in version 2.0: The experiments framework was moved to <https://reddit-experiments.readthedocs.io/en/latest/>.

baseplate.lib.file_watcher

Watch a file and keep a parsed copy in memory that's updated on changes.

The contents of the file are re-loaded and parsed only when necessary.

For example, a JSON file like the following:

```
{
  "one": 1,
  "two": 2
}
```

might be watched and parsed like this:

```
>>> watcher = FileWatcher(path, parser=json.load)
>>> watcher.get_data() == {u"one": 1, u"two": 2}
True
```

The return value of *get_data()* would change whenever the underlying file changes.

class baseplate.lib.file_watcher.**FileWatcher**(*path*, *parser*, *timeout=None*, *binary=False*, *encoding=None*, *newline=None*, *backoff=None*)

Watch a file and load its data when it changes.

Parameters

- **path** (str) – Full path to a file to watch.
- **parser** (Callable[[IO], ~T]) – A callable that takes an open file object, parses or otherwise interprets the file, and returns whatever data is meaningful.
- **timeout** (Optional[float]) – How long, in seconds, to block instantiation waiting for the watched file to become available (defaults to not blocking).
- **binary** (bool) – Should the file be opened in binary mode. If *True* the file will be opened with the mode *"rb"*, otherwise it will be opened with the mode *"r"*. (defaults to *"r"*)
- **encoding** (Optional[str]) – The name of the encoding used to decode the file. The default encoding is platform dependent (whatever locale.getpreferredencoding() returns), but any text encoding supported by Python can be used. This is not supported if *binary* is set to *True*.
- **newline** (Optional[str]) – Controls how universal newlines mode works (it only applies to text mode). It can be *None*, *""*, *"\n"*, *"\r"*, and *"\r\n"*. This is not supported if *binary* is set to *True*.
- **backoff** (Optional[float]) – retry backoff time for the file watcher. Defaults to None, which is mapped to DEFAULT_FILEWATCHER_BACKOFF.

get_data()

Return the current contents of the file, parsed.

The watcher ensures that the file is re-loaded and parsed whenever its contents change. Parsing only occurs when necessary, not on each call to this method. This method returns whatever the most recent call to the parser returned.

Make sure to call this method each time you need data from the file rather than saving its results elsewhere. This ensures you always have the freshest data.

Return type ~T

get_data_and_mtime()

Return tuple of the current contents of the file and file mtime.

The watcher ensures that the file is re-loaded and parsed whenever its contents change. Parsing only occurs when necessary, not on each call to this method. This method returns whatever the most recent call to the parser returned.

When file content was changed, it returns the recent mtime, notify the caller the content is different from previous cached.

Make sure to call this method each time you need data from the file rather than saving its results elsewhere. This ensures you always have the freshest data.

Return type Tuple[~T, float]

Exceptions

exception baseplate.lib.file_watcher.**WatchedFileNotAvailableError**(*path*, *inner*)

Raised when the watched file could not be loaded.

Testing

class baseplate.testing.lib.file_watcher.**FakeFileWatcher**(*data=<class 'baseplate.lib.file_watcher._NOT_LOADED'>*, *mtime=1234*)

Fake file watcher for testing purposes.

Use this in place of a *FileWatcher* in tests to avoid having to load an actual file:

```
>>> file_watcher = FakeFileWatcher()
>>> file_watcher.get_data()
Traceback (most recent call last):
baseplate.lib.file_watcher.WatchedFileNotAvailableError: /fake-file-watcher: no␣
↪value set
>>> file_watcher.data = "test"
>>> file_watcher.get_data()
'test'
```

New in version 1.5.

baseplate.lib.live_data

This component of Baseplate provides real-time synchronization of data across a cluster of servers. It is intended for situations where data is read frequently, does not change super often, and when it does change needs to change everywhere at once. In most cases, this will be an underlying feature of some other system (e.g. an experiments framework.)

There are four main components of the live data system:

- ZooKeeper, a highly available data store that can push change notifications.
- The watcher, a sidecar daemon that watches nodes in ZooKeeper and syncs their contents to disk.
- *FileWatcher* instances in your application that load the synchronized data into memory.
- Something that writes to ZooKeeper (potentially the writer tool).

The watcher daemon and tools for writing data to ZooKeeper are covered on this page.

Watcher Daemon

The watcher daemon is a sidecar that watches nodes in ZooKeeper and syncs their contents to local files on change. It is entirely configured via INI file and is run like so:

```
$ python -m baseplate.sidecars.live_data_watcher some_config.ini
```

Where `some_config.ini` might look like:

```
[live-data]
zookeeper.hosts = zk01:2181,zk02:2181
zookeeper.credentials = secret/myservice/zookeeper_credentials

nodes.a.source = /a/node/in/zookeeper
nodes.a.dest = /var/local/file-on-disk

nodes.b.source = /another/node/in/zookeeper
nodes.b.dest = /var/local/another-file
nodes.b.owner = www-data
nodes.b.group = www-data
nodes.b.mode = 0400
```

Each of the defined `nodes` will be watched by the daemon.

The watcher daemon will touch the `mtime` of the local files periodically to indicative liveliness to monitoring tools.

The Writer Tool

For simple cases where you just want to put the contents of a file into ZooKeeper (perhaps in a CI task) you can use the live data writer. It expects a configuration file with ZooKeeper connection information, like the watcher, and takes some additional parameters on the command line.

```
$ python -m baseplate.lib.live_data.writer some_config.ini \
   input.json /some/node/in/zookeeper
Writing input.json to ZooKeeper /some/node/in/zookeeper...
---

+++

@@ -1,4 +1,4 @@

{
-    "key": "one"
+    "key": "two"
}
Wrote data to Zookeeper.
```

The ZooKeeper node must be created before this tool can be used so that appropriate ACLs can be configured.

Direct access to ZooKeeper

If you're doing something more complicated with your data that the above tools don't cover, you'll want to connect directly to ZooKeeper.

`baseplate.lib.live_data.`**`zookeeper_client_from_config`**(*secrets*, *app_config*, *read_only=None*)

Configure and return a ZooKeeper client.

There are several configuration options:

`zookeeper.hosts` A comma-delimited list of hosts with optional `chroot` at the end. For example `zk01:2181,zk02:2181` or `zk01:2181,zk02:2181/some/root`.

`zookeeper.credentials` (Optional) A comma-delimited list of paths to secrets in the secrets store that contain ZooKeeper authentication credentials. Secrets should be of the "simple" type and contain `username:password`.

zookeeper.timeout (Optional) A time span of how long to wait for each connection attempt.

The client will attempt forever to reconnect on connection loss.

Parameters

- **secrets** (*SecretsStore*) – A secrets store object
- **raw_config** – The application configuration which should have settings for the ZooKeeper client.
- **read_only** (Optional[bool]) – Whether or not to allow connections to read-only ZooKeeper servers.

Return type KazooClient

baseplate.lib.message_queue

This module provides a thin wrapper around POSIX Message queues.

Note: This implementation uses POSIX Message queues and is not portable to all operating systems.

There are also various limits on the sizes of queues:

- The `msgqueue rlimit` limits the amount of space the user can use on message queues.
- The `fs.mqueue.msg_max` and `fs.mqueue.msgsize_max` sysctls limit the maximum number of messages and the maximum size of each message which a queue can be configured to have.

Minimal Example

Here's a minimal, artificial example of a separate producer and consumer process pair (run the producer then the consumer):

```
# producer.py
from baseplate.lib.message_queue import MessageQueue

# If the queue doesn't already exist, we'll create it.
mq = MessageQueue(
    "/baseplate-testing", max_messages=1, max_message_size=1)
message = "1"
mq.put(message)
print("Put Message: %s" % message)
```

You should see:

```
Put Message: 1
```

After running the producer once, we have a single message pushed on to our POSIX message queue. Next up, run the consumer:

```
# consumer.py
from baseplate.lib.message_queue import MessageQueue

mq = MessageQueue(
    "/baseplate-testing", max_messages=1, max_message_size=1)
```

(continues on next page)

(continued from previous page)

```
# Unless a `timeout` kwarg is passed, this will block until
# we can pop a message from the queue.
message = mq.get()
print("Get Message: %s" % message.decode())
```

You'll end up seeing:

```
Get Message: 1
```

The `/baseplate-testing` value is the name of the queue. Queues names should start with a forward slash, followed by one or more characters (but no additional slashes).

Multiple processes can bind to the same queue by specifying the same queue name.

Message Queue Default Limits

Most operating systems with POSIX queues include very low defaults for the maximum message size and maximum queue depths. On Linux 2.6+, you can list and check the values for these by running:

```
$ ls /proc/sys/fs/mqueue/
msg_default  msg_max  msgsize_default  msgsize_max  queues_max
$ cat /proc/sys/fs/mqueue/msgsize_max
8192
```

Explaining these in detail is outside the scope of this document, so we'll refer you to POSIX Message queues (or `man 7 mq_overview`) for detailed instructions on what these mean.

Gotchas

If you attempt to create a POSIX Queue where one of your provided values is over the limits defined under `/proc/sys/fs/mqueue/`, you'll probably end up seeing a vague `ValueError` exception. Here's an example:

```
>>> from baseplate.lib.message_queue import MessageQueue
>>> mq = MessageQueue(
        "/over-the-limit", max_messages=11, max_message_size=8096)
Traceback (most recent call last):
  File "<input>", line 2, in <module>
  File "/home/myuser/baseplate/baseplate/lib/message_queue.py", line 83, in __init__
    max_message_size=max_message_size,
ValueError: Invalid parameter(s)
```

Since the default value for `/proc/sys/fs/mqueue/msg_max` on Linux is 10, our `max_messages=11` is invalid. You can raise these limits by doing something like this as a privileged user:

```
$ echo "50" > /proc/sys/fs/mqueue/msg_max
```

CLI Usage

The *message_queue* module can also be run as a command-line tool to consume, log, and discard messages from a given queue:

```
$ python -m baseplate.lib.message_queue --read /queue
```

or to write arbitrary messages to the queue:

```
$ echo hello! | python -m baseplate.lib.message_queue --write /queue
```

See `--help` for more info.

baseplate.lib.message_queue

A Gevent-friendly POSIX message queue.

class baseplate.lib.message_queue.**MessageQueue**(*name*, *max_messages*, *max_message_size*)

A Gevent-friendly (but not required) inter process message queue.

`name` should be a string of up to 255 characters consisting of an initial slash, followed by one or more characters, none of which are slashes.

Note: This relies on POSIX message queues being available and select(2)-able like other file descriptors. Not all operating systems support this.

get(*timeout=None*)

Read a message from the queue.

Parameters **timeout** (Optional[float]) – If the queue is empty, the call will block up to `timeout` seconds or forever if `None`.

Raises *TimedOutError* The queue was empty for the allowed duration of the call.

Return type bytes

put(*message*, *timeout=None*)

Add a message to the queue.

Parameters **timeout** (Optional[float]) – If the queue is full, the call will block up to `timeout` seconds or forever if `None`.

Raises *TimedOutError* The queue was full for the allowed duration of the call.

Return type None

unlink()

Remove the queue from the system.

The queue will not leave until the last active user closes it.

Return type None

close()

Close the queue, freeing related resources.

This must be called explicitly if queues are created/destroyed on the fly. It is not automatically called when the object is reclaimed by Python.

Return type None

Exceptions

exception baseplate.lib.message_queue.**MessageQueueError**
Base exception for message queue related errors.

exception baseplate.lib.message_queue.**TimedOutError**
Raised when a message queue operation times out.

baseplate.lib.metrics

Application metrics via StatsD.

A client for the application metrics aggregator StatsD. Metrics sent to StatsD are aggregated and written to graphite. StatsD is generally used for whole-system health monitoring and insight into usage patterns.

Basic example usage:

```
from baseplate.lib.metrics import metrics_client_from_config

client = metrics_client_from_config(app_config)
client.counter("events.connect").increment()
client.gauge("workers").replace(4)

with client.timer("something.todo"):
    do_something()
    do_something_else()
```

If you have multiple metrics to send, you can batch them up for efficiency:

```
with client.batch() as batch:
    batch.counter("froozles").increment()
    batch.counter("blargs").decrement(delta=3)

    with batch.timer("something"):
        do_another_thing()
```

and the batch will be sent in as few packets as possible when the *with* block ends.

Clients

baseplate.lib.metrics.**metrics_client_from_config**(*raw_config*)
Configure and return a metrics client.

This expects two configuration options:

metrics.namespace The root key to prefix all metrics in this application with.

metrics.endpoint A `host:port` pair, e.g. `localhost:2014`. If an empty string, a client that discards all metrics will be returned.

metrics.log_if_unconfigured` Whether to log metrics when there is no unconfigured endpoint. Defaults to false.

metrics.swallow_network_errors` When false, network errors during sending to metrics collector will cause an exception to be thrown. When true, those exceptions are logged and swallowed instead. Defaults to false.

Parameters **raw_config** (Dict[str, str]) – The application configuration which should have settings for the metrics client.

Return type *Client*

Returns A configured client.

class baseplate.lib.metrics.**Client**(*transport*, *namespace*)
A client for StatsD.

batch()
Return a client-like object which batches up metrics.

Batching metrics can reduce the number of packets that are sent to the stats aggregator.

Return type *Batch*

counter(*name*, *tags=None*)
Return a Counter with the given name.

The sample rate is currently up to your application to enforce.

Parameters **name** (str) – The name the counter should have.

Return type *Counter*

gauge(*name*, *tags=None*)
Return a Gauge with the given name.

Parameters **name** (str) – The name the gauge should have.

Return type *Gauge*

histogram(*name*, *tags=None*)
Return a Histogram with the given name.

Parameters **name** (str) – The name the histogram should have.

Return type *Histogram*

timer(*name*, *tags=None*)
Return a Timer with the given name.

Parameters **name** (str) – The name the timer should have.

Return type *Timer*

class baseplate.lib.metrics.**Batch**(*transport*, *namespace*)
A batch of metrics to send to StatsD.

The batch also supports the context manager protocol, for use with Python's `with` statement. When the context is exited, the batch will automatically *flush()*.

flush()
Immediately send the batched metrics.

Return type None

counter(*name*, *tags=None*)
Return a BatchCounter with the given name.

The sample rate is currently up to your application to enforce. :type name: str :param name: The name the counter should have.

Return type *Counter*

gauge(*name*, *tags=None*)
Return a Gauge with the given name.

Parameters **name** (str) – The name the gauge should have.

Return type *Gauge*

histogram(*name*, *tags=None*)
Return a Histogram with the given name.

Parameters **name** (str) – The name the histogram should have.

Return type *Histogram*

timer(*name*, *tags=None*)
Return a Timer with the given name.

Parameters **name** (str) – The name the timer should have.

Return type *Timer*

Metrics

class baseplate.lib.metrics.**Counter**(*transport*, *name*, *tags=None*)
A counter for counting events over time.

increment(*delta=1.0*, *sample_rate=1.0*)
Increment the counter.

Parameters

- **delta** (float) – The amount to increase the counter by.
- **sample_rate** (float) – What rate this counter is sampled at. [0-1].

Return type None

decrement(*delta=1.0*, *sample_rate=1.0*)
Decrement the counter.

This is equivalent to *increment()* with delta negated.

Return type None

send(*delta*, *sample_rate*)
Send the counter to the backend.

Parameters

- **delta** (float) – The amount to increase the counter by.
- **sample_rate** (float) – What rate this counter is sampled at. [0-1].

Return type None

class baseplate.lib.metrics.**Timer**(*transport*, *name*, *tags=None*)
A timer for recording elapsed times.

The timer also supports the context manager protocol, for use with Python's with statement. When the context is entered the timer will *start()* and when exited, the timer will automatically *stop()*.

start(*sample_rate=1.0*)
Record the current time as the start of the timer.

Return type None

stop()
Stop the timer and record the total elapsed time.

Return type None

send(*elapsed*, *sample_rate=1.0*)
Directly send a timer value without having to stop/start.

This can be useful when the timing was managed elsewhere and we just want to report the result. :type elapsed: `float` :param elapsed: The elapsed time in seconds to report.

Return type `None`

class `baseplate.lib.metrics.`**Gauge**(*transport*, *name*, *tags=None*)
A gauge representing an arbitrary value.

Note: The StatsD protocol supports incrementing/decrementing gauges from their current value. We do not support that here because this feature is unpredictable in face of the StatsD server restarting and the "current value" being lost.

---

replace(*new_value*)
Replace the value held by the gauge.

This will replace the value held by the gauge with no concern for its previous value.

Note: Due to the way the protocol works, it is not possible to replace gauge values with negative numbers.

Parameters **new_value** (`float`) – The new value to store in the gauge.

Return type `None`

class `baseplate.lib.metrics.`**Histogram**(*transport*, *name*, *tags=None*)
A bucketed distribution of integer values across a specific range.

Records data value counts across a configurable integer value range with configurable buckets of value precision within that range.

Configuration of each histogram is managed by the backend service, not by this interface. This implementation also depends on histograms being supported by the StatsD backend. Specifically, the StatsD backend must support the `h` key, e.g. `metric_name:320|h`.

add_sample(*value*)
Add a new value to the histogram.

This records a new value to the histogram; the bucket it goes in is determined by the backend service configurations.

Return type `None`

baseplate.lib.random

Extensions to the standard library *random* module.

class `baseplate.lib.random.`**WeightedLottery**(*items*, *weight_key*)
A lottery where items can have different chances of selection.

Items will be picked with chance proportional to their weight relative to the sum of all weights, so the higher the weight, the higher the chance of being picked.

Parameters

- **items** (`Iterable`[~T]) – Items to choose from.

- **weight_key** (Callable[[~T], int]) – A function that takes an item in items and returns a non-negative integer weight for that item.

Raises ValueError if any weights are negative or there are no items.

An example of usage:

```
>>> words = ["apple", "banana", "cantelope"]
>>> lottery = WeightedLottery(words, weight_key=len)
>>> lottery.pick()
'banana'
>>> lottery.sample(2)
['apple', 'cantelope']
```

pick()
Pick a random element from the lottery.

Return type ~T

sample(*sample_size*)
Sample elements from the lottery without replacement.

Parameters **sample_size** (int) – The number of items to sample from the lottery.

Return type Iterable[~T]

baseplate.lib.ratelimit

Configuring a rate limiter for your request context requires a context factory for the backend and a factory for the rate limiter itself:

```
redis_pool = pool_from_config(app_config)
backend_factory = RedisRateLimitBackendContextFactory(redis_pool)
ratelimiter_factory = RateLimiterContextFactory(backend_factory, allowance, interval)
baseplate.add_to_context('ratelimiter', ratelimiter_factory)
```

The rate limiter can then be used during a request with:

```
try:
    context.ratelimiter.consume(context.request_context.user.id)
    print('Ratelimit passed')
except RateLimitExceededException:
    print('Too many requests')
```

Classes

class baseplate.lib.ratelimit.**RateLimiter**(*backend*, *allowance*, *interval*)
A class for rate limiting actions.

Parameters

- **backend** (*RateLimitBackend*) – The backend to use for storing rate limit counters.
- **allowance** (int) – The maximum allowance allowed per key.
- **interval** (int) – The interval (in seconds) to reset allowances.

consume (*key*, *amount=1*)
Consume the given *amount* from the allowance for the given *key*.

This will raise *baseplate.lib.ratelimit.RateLimitExceededException* if the allowance for *key* is exhausted.

Parameters

- **key** (str) – The name of the rate limit bucket to consume from.
- **amount** (int) – The amount to consume from the rate limit bucket.

Return type None

class baseplate.lib.ratelimit.**RateLimiterContextFactory** (*backend_factory*, *allowance*, *interval*)
RateLimiter context factory.

Parameters

- **backend_factory** (*ContextFactory*) – An instance of *baseplate.clients.ContextFactory*. The context factory must return an instance of *baseplate.lib.ratelimit.backends.RateLimitBackend*
- **allowance** (int) – The maximum allowance allowed per key.
- **interval** (int) – The interval (in seconds) to reset allowances.

make_object_for_context (*name*, *span*)
Return an object that can be added to the context object.

Parameters

- **name** (str) – The name assigned to this object on the context.
- **span** (*Span*) – The current span this object is being made for.

Return type RateLimiter

class baseplate.lib.ratelimit.**RateLimitExceededException**
This exception gets raised whenever a rate limit is exceeded.

Backends

class baseplate.lib.ratelimit.backends.**RateLimitBackend**
An interface for rate limit backends to implement.

consume (*key*, *amount*, *allowance*, *interval*)
Consume the given *amount* from the allowance for the given *key*.

This will return true if the *key* remains below the *allowance* after consuming the given *amount*.

Parameters

- **key** (str) – The name of the rate limit bucket to consume from.
- **amount** (int) – The amount to consume from the rate limit bucket.
- **allowance** (int) – The maximum allowance for the rate limit bucket.
- **interval** (int) – The interval to reset the allowance.

Return type bool

Memcache

class baseplate.lib.ratelimit.backends.memcache.**MemcacheRateLimitBackendContextFactory**(*memc*
pre-
fix='r

MemcacheRateLimitBackend context factory.

Parameters

- **memcache_pool** (PooledClient) – The memcache pool to back this ratelimit.
- **prefix** (str) – A prefix to add to keys during rate limiting. This is useful if you will have two different rate limiters that will receive the same keys.

make_object_for_context(*name*, *span*)

Return an object that can be added to the context object.

Parameters

- **name** (str) – The name assigned to this object on the context.
- **span** (*Span*) – The current span this object is being made for.

Return type *MemcacheRateLimitBackend*

class baseplate.lib.ratelimit.backends.memcache.**MemcacheRateLimitBackend**(*memcache*,
pre-
fix='rl:')

A Memcache backend for rate limiting.

Parameters

- **memcache** (*MonitoredMemcacheConnection*) – A memcached connection.
- **prefix** (str) – A prefix to add to keys during rate limiting. This is useful if you will have two different rate limiters that will receive the same keys.

consume(*key*, *amount*, *allowance*, *interval*)

Consume the given *amount* from the allowance for the given *key*.

This will return true if the *key* remains below the *allowance* after consuming the given *amount*.

Parameters

- **key** (str) – The name of the rate limit bucket to consume from.
- **amount** (int) – The amount to consume from the rate limit bucket.
- **allowance** (int) – The maximum allowance for the rate limit bucket.
- **interval** (int) – The interval to reset the allowance.

Return type bool

Redis

class baseplate.lib.ratelimit.backends.redis.**RedisRateLimitBackendContextFactory**(*redis_pool*, *prefix='rl:'*)

RedisRateLimitBackend context factory.

Parameters

- **redis_pool** (ConnectionPool) – The redis pool to back this ratelimit.
- **prefix** (str) – A prefix to add to keys during rate limiting. This is useful if you will have two different rate limiters that will receive the same keys.

make_object_for_context(*name*, *span*)

Return an object that can be added to the context object.

Parameters

- **name** (str) – The name assigned to this object on the context.
- **span** (*Span*) – The current span this object is being made for.

Return type *RedisRateLimitBackend*

class baseplate.lib.ratelimit.backends.redis.**RedisRateLimitBackend**(*redis*, *prefix='rl:'*)

A Redis backend for rate limiting.

Parameters

- **redis** (*MonitoredRedisConnection*) – An instance of *baseplate.clients.redis.MonitoredRedisConnection*.
- **prefix** (str) – A prefix to add to keys during rate limiting. This is useful if you will have two different rate limiters that will receive the same keys.

consume(*key*, *amount*, *allowance*, *interval*)

Consume the given *amount* from the allowance for the given *key*.

This will return true if the *key* remains below the *allowance* after consuming the given *amount*.

Parameters

- **key** (str) – The name of the rate limit bucket to consume from.
- **amount** (int) – The amount to consume from the rate limit bucket.
- **allowance** (int) – The maximum allowance for the rate limit bucket.
- **interval** (int) – The interval to reset the allowance.

Return type bool

baseplate.lib.retry

Note: This module is a low-level helper, many client libraries have protocol-aware retry logic built in. Check your library before using this.

Policies for retrying an operation safely.

class baseplate.lib.retry.**RetryPolicy**

A policy for retrying operations.

Policies are meant to be used as an iterable:

```
for time_remaining in RetryPolicy.new(attempts=3):
    try:
        some_operation.do(timeout=time_remaining)
        break
    except SomeError:
        pass
else:
    raise MaxRetriesError
```

yield_attempts()

Return an iterator which controls attempts.

On each iteration, the iterator will yield the number of seconds left to retry, this should be used to set the timeout on the operation being carried out. If there is no maximum time remaining, `None` is yielded instead.

The iterable will raise `StopIteration` once the operation should not be retried any further.

Return type `Iterator`[`Optional`[`float`]]

__iter__()

Return the result of *yield_attempts()*.

This allows policies to be directly iterated over.

Return type `Iterator`[`Optional`[`float`]]

static new(*attempts=None*, *budget=None*, *backoff=None*)

Create a new retry policy with the given constraints.

Parameters

- **attempts** (`Optional`[`int`]) – The maximum number of times the operation can be attempted.
- **budget** (`Optional`[`float`]) – The maximum amount of time, in seconds, that the local service will wait for the operation to succeed.
- **backoff** (`Optional`[`float`]) – The base amount of time, in seconds, for exponential back-off between attempts. `N` in (`N * 2**attempts`).

Return type *RetryPolicy*

baseplate.lib.secrets

Application integration with the secret fetcher daemon.

Fetcher Daemon

The secret fetcher is a sidecar that is run as a single daemon on each server. It can authenticate to Vault either as the server itself (through an AWS-signed instance identity document) or through a mounted JWT when running within a Kubernetes pod. It then gets access to secrets based upon the policies mapped to the role it authenticated as. Once authenticated, it fetches a given list of secrets from Vault and stores all of the data in a local file. It will automatically re-fetch secrets as their leases expire, ensuring that key rotation happens on schedule.

Because this is a sidecar, individual application processes don't need to talk directly to Vault for simple secret tokens (but can do so if needed for more complex operations like using the Transit backend). This reduces the load on Vault and adds a safety net if Vault becomes unavailable.

Secret Store

The secret store is the in-application integration with the file output of the fetcher daemon.

baseplate.lib.secrets.**secrets_store_from_config**(*app_config*, *timeout=None*, *prefix='secrets.'*)

Configure and return a secrets store.

The keys useful to *secrets_store_from_config()* should be prefixed, e.g. `secrets.url`, etc.

Supported keys:

`path`: the path to the secrets file generated by the secrets fetcher daemon.

Parameters

- **app_config** (`Dict[str, str]`) – The application configuration which should have settings for the secrets store.
- **timeout** (`Optional[int]`) – How long, in seconds, to block instantiation waiting for the secrets data to become available (defaults to not blocking).
- **prefix** (`str`) – Specifies the prefix used to filter keys. Defaults to "secrets."
- **backoff** – retry backoff time for secrets file watcher. Defaults to None, which is mapped to DEFAULT_FILEWATCHER_BACKOFF.

Return type *SecretsStore*

class baseplate.lib.secrets.**SecretsStore**(*path*, *timeout=None*, *backoff=None*)

Access to secret tokens with automatic refresh when changed.

This local vault allows access to the secrets cached on disk by the fetcher daemon. It will automatically reload the cache when it is changed. Do not cache or store the values returned by this class's methods but rather get them from this class each time you need them. The secrets are served from memory so there's little performance impact to doing so and you will be sure to always have the current version in the face of key rotation etc.

get_raw(*path*)

Return a dictionary of key/value pairs for the given secret path.

This is the raw representation of the secret in the underlying store.

Return type `Dict[str, str]`

get_credentials(*path*)
Decode and return a credential secret.

Credential secrets are a convention of username/password pairs stored as separate values in the raw secret payload.

The following keys are significant:

type This must always be `credential` for this method.

encoding This must be unset or set to `identity`.

username This contains the raw username.

password This contains the raw password.

Return type *CredentialSecret*

get_simple(*path*)
Decode and return a simple secret.

Simple secrets are a convention of key/value pairs in the raw secret payload. The following keys are significant:

type This must always be `simple` for this method.

value This contains the raw value of the secret token.

encoding (Optional) If present, how to decode the value from how it's encoded at rest (only `base64` currently supported).

Return type `bytes`

get_versioned(*path*)
Decode and return a versioned secret.

Versioned secrets are a convention of key/value pairs in the raw secret payload. The following keys are significant:

type This must always be `versioned` for this method.

current, next, and previous The raw secret value's versions. `current` is the "active" version, which is used for new creation/signing operations. `previous` and `next` are only used for validation (e.g. checking signatures) to ensure continuity when keys rotate. Both `previous` and `next` are optional.

encoding (Optional) If present, how to decode the values from how they are encoded at rest (only `base64` currently supported).

Return type *VersionedSecret*

get_vault_url()
Return the URL for accessing Vault directly.

Return type `str`

get_vault_token()
Return a Vault authentication token.

The token will have policies attached based on the current EC2 server's Vault role. This is only necessary if talking directly to Vault.

Return type `str`

get_raw_and_mtime(*path*)
Return raw secret and modification time.

This returns the same data as *get_raw()* as well as a UNIX epoch timestamp indicating the last time the secrets data was updated. This modification time can be used to know when to invalidate downstream caching.

New in version 1.5.

Return type Tuple[Dict[str, str], float]

get_credentials_and_mtime(*path*)
Return credentials secret and modification time.

This returns the same data as *get_credentials()* as well as a UNIX epoch timestamp indicating the last time the secrets data was updated. This modification time can be used to know when to invalidate downstream caching.

New in version 1.5.

Return type Tuple[*CredentialSecret*, float]

get_simple_and_mtime(*path*)
Return simple secret and modification time.

This returns the same data as *get_simple()* as well as a UNIX epoch timestamp indicating the last time the secrets data was updated. This modification time can be used to know when to invalidate downstream caching.

New in version 1.5.

Return type Tuple[bytes, float]

get_versioned_and_mtime(*path*)
Return versioned secret and modification time.

This returns the same data as *get_versioned()* as well as a UNIX epoch timestamp indicating the last time the secrets data was updated. This modification time can be used to know when to invalidate downstream caching.

New in version 1.5.

Return type Tuple[*VersionedSecret*, float]

make_object_for_context(*name*, *span*)
Return an object that can be added to the context object.

This allows the secret store to be used with *add_to_context()*:

```
secrets = SecretsStore("/var/local/secrets.json")
baseplate.add_to_context("secrets", secrets)
```

Return type *SecretsStore*

class baseplate.lib.secrets.**VersionedSecret**(*previous: Optional[bytes]*, *current: bytes*, *next: Optional[bytes]*)
A versioned secret.

Versioned secrets allow for seamless rotation of keys. When using the secret to generate tokens (e.g. signing a message) always use the `current` value. When validating tokens, check against all the versions in `all_versions`. This will allow keys to rotate smoothly even if not done instantly across all users of the secret.

property previous
Alias for field number 0

property current
Alias for field number 1

property next
Alias for field number 2

property all_versions
Return an iterator over the available versions of this secret.

Return type Iterator[bytes]

classmethod from_simple_secret(*value*)
Make a fake versioned secret from a single value.

This is a backwards compatibility shim for use with APIs that take versioned secrets. Try to use proper versioned secrets fetched from the secrets store instead.

Return type *VersionedSecret*

class baseplate.lib.secrets.**CredentialSecret**(*username: str*, *password: str*)
A secret for storing username/password pairs.

Credential secrets allow us to store usernames and passwords together in a single secret. Note that they are not versioned since the general pattern for rotating credentials like this would be to generate a new username/password pair. This object has two properties:

property username
Alias for field number 0

property password
Alias for field number 1

Exceptions

exception baseplate.lib.secrets.**CorruptSecretError**(*path*, *message*)
Raised when the requested secret does not match the expected format.

exception baseplate.lib.secrets.**SecretNotFoundError**(*name*)
Raised when the requested secret is not in the local vault.

exception baseplate.lib.secrets.**SecretsNotAvailableError**(*inner*)
Raised when the secrets store was not accessible.

Testing

class baseplate.testing.lib.secrets.**FakeSecretsStore**(*fake_secrets*)
Fake secrets store for testing purposes.

Use this in place of a *SecretsStore* in tests to avoid having to load an actual file:

```
>>> secrets = FakeSecretsStore({
...     "secrets": {
...         "secret/foo/bar": {
...             "type": "versioned",
...             "current": "hunter2",
...         },
```

(continues on next page)

(continued from previous page)

```
...     },
... })
>>> secrets.get_versioned("secret/foo/bar")
VersionedSecret(previous=None, current=b'hunter2', next=None)
```

New in version 1.5.

baseplate.lib.thrift_pool

A Thrift client connection pool.

Note: See *baseplate.clients.thrift.ThriftContextFactory* for a convenient way to integrate the pool with your application.

The pool lazily creates connections and maintains them in a pool. Individual connections have a maximum lifetime, after which they will be recycled.

A basic example of usage:

```
pool = thrift_pool_from_config(app_config, "example_service.")
with pool.connection() as protocol:
    client = ExampleService.Client(protocol)
    client.do_example_thing()
```

Configuration Parsing

baseplate.lib.thrift_pool.**thrift_pool_from_config**(*app_config*, *prefix*, ***kwargs*)

Make a ThriftConnectionPool from a configuration dictionary.

The keys useful to *thrift_pool_from_config()* should be prefixed, e.g. `example_service.endpoint` etc. The `prefix` argument specifies the prefix used to filter keys. Each key is mapped to a corresponding keyword argument on the *ThriftConnectionPool* constructor. Any keyword arguments given to this function will be also be passed through to the constructor. Keyword arguments take precedence over the configuration file.

Supported keys:

- **`endpoint` (required): A `host:port` pair, e.g. `localhost:2014`,** where the Thrift server can be found.
- `size`: The size of the connection pool.
- **`max_age`: The oldest a connection can be before it's recycled and** replaced with a new one. Written as a *Timespan()* e.g. `1 minute`.
- **`timeout`: The maximum amount of time a connection attempt or RPC call** can take before a TimeoutError is raised. (*Timespan()*)
- **`max_connection_attempts`: The maximum number of times the pool will attempt to** open a connection.

Changed in version 1.2: `max_retries` was renamed `max_connection_attempts`.

Return type *ThriftConnectionPool*

Classes

class baseplate.lib.thrift_pool.**ThriftConnectionPool**(*endpoint*, *size=10*, *max_age=120*, *timeout=1*, *max_connection_attempts=3*, *protocol_factory=<thrift.protocol.THeaderProtocol.THeader object>*)

A pool that maintains a queue of open Thrift connections.

Parameters

- **endpoint** (*EndpointConfiguration*) – The remote address of the Thrift service.
- **size** (int) – The maximum number of connections that can be open before new attempts to open block.
- **max_age** (int) – The maximum number of seconds a connection should be kept alive. Connections older than this will be reaped.
- **timeout** (int) – The maximum number of seconds a connection attempt or RPC call can take before a TimeoutError is raised.
- **max_connection_attempts** (int) – The maximum number of times the pool will attempt to open a connection.
- **protocol_factory** (TProtocolFactory) – The factory to use for creating protocols from transports. This is useful for talking to services that don't support THeaderProtocol.

All exceptions raised by this class derive from TTransportException.

Changed in version 1.2: max_retries was renamed max_connection_attempts.

connection()

Acquire a connection from the pool.

This method is to be used with a context manager. It returns a connection from the pool, or blocks up to timeout seconds waiting for one if the pool is full and all connections are in use.

When the context is exited, the connection is returned to the pool. However, if it was exited via an unexpected Thrift exception, the connection is closed instead because the state of the connection is unknown.

Return type Generator[TProtocolBase, None, None]

baseplate.lib.service_discovery

Integration with Synapse's file_output service discovery method.

Note: Production Baseplate services have Synapse hooked up to a local HAProxy instance which will automatically route connections to services for you if you connect to the correct address/port on localhost. That is the preferred method of connecting to services.

The contents of this module are useful for inspecting the service inventory directly for cases where a blind TCP connection is insufficient (e.g. to give service addresses to a client, or for topology-aware clients like Cassandra).

A basic example of usage:

```
inventory = ServiceInventory("/var/lib/synapse/example.json")
backend = inventory.get_backend()
print(backend.endpoint.address)
```

class baseplate.lib.service_discovery.**ServiceInventory**(*filename*)

The inventory enumerates available backends for a single service.

Parameters **filename** (str) – The absolute path to the Synapse-generated inventory file in JSON format.

get_backends()

Return a list of all available backends in the inventory.

If the inventory file becomes unavailable, the previously seen inventory is returned.

Return type Sequence[*Backend*]

get_backend()

Return a randomly chosen backend from the available backends.

If weights are specified in the inventory, they will be respected when making the random selection.

Raises *NoBackendsAvailableError* if the inventory has no available endpoints.

Return type *Backend*

class baseplate.lib.service_discovery.**Backend**(*id: int*, *name: str*, *endpoint:* baseplate.lib.config.EndpointConfiguration, *weight: int*)

A description of a service backend.

This is a tuple of several values:

id A unique integer ID identifying the backend.

name The name of the backend.

endpoint An *EndpointConfiguration* object describing the network address of the backend.

weight An integer weight indicating how much to prefer this backend when choosing whom to connect to.

Exceptions

exception baseplate.lib.service_discovery.**NoBackendsAvailableError**

Raised when no backends are available for this service.

1.4 CLI Tools

1.4.1 baseplate-healthcheck

Baseplate services have well-defined health-check endpoints. The `baseplate-healthcheck` tool connects to a given service and checks these endpoints to see if they're alive.

Command Line

There are two required arguments on the command line: the protocol of the service to check (`thrift` or `wsgi`) and the endpoint to connect to.

There's also an optional argument on the command line: the probe to check. By default the probe to check is `readiness`, but you can choose one from: `readiness`, `liveness`, and `startup`.

For example, to check a Thrift-based service listening on port 9090 for liveness:

```
$ baseplate-healthcheck thrift 127.0.0.1:9090 --probe liveness
```

or a WSGI (HTTP) service listening on a UNIX domain socket, with default probe

```
$ baseplate-healthcheck wsgi /run/myservice.sock
```

Results

If the service is healthy, the tool will exit with a status code indicating success (0) and print "OK!". If the service is unhealthy, the tool will exit with a status code indicating failure (1) and print an error message explaining what went wrong.

Usage

This script can be used as part of a process to validate a server after creation, or to check service liveliness for a service discovery system.

1.4.2 `baseplate-serve`

Baseplate comes with a simple Gevent-based server for both Thrift and WSGI applications called `baseplate-serve`.

Configuration

There is one required parameter on the command line, the path to an INI-format configuration file. There should be two sections in the file: the `server` section and the `app` section. The section headers look like `server:main` or `app:main` where the part before the `:` is the type of section and the part after is the "name". Baseplate looks for sections named `main` by default but can be overridden with the `--server-name` and `--app-name` options.

The Server

Here's an example of a `server` section:

```
[server:main]
factory = baseplate.server.thrift
stop_timeout = 30
```

The `factory` tells baseplate what code to use to run the server. Baseplate comes with two servers built in:

`baseplate.server.thrift` A Gevent Thrift server.

`baseplate.server.wsgi` A Gevent WSGI server.

Both take two configuration values as well:

max_concurrency The maximum number of simultaneous clients the server will handle. Note that this is how many connections will be accepted, but some of those connections may be idle at any given time.

stop_timeout (Optional) How long, in seconds, to wait for active connections to finish up gracefully when shutting down. By default, the server will shut down immediately.

The WSGI server takes an additional optional parameter:

handler A full name of a class which subclasses `gevent.pywsgi.WSGIHandler` for extra functionality.

There are some additional configuration settings in this section that start with a `monitoring` prefix. For more information on those, see *Process-level metrics*.

The Application

And now the real bread and butter, your `app` section:

```
[app:main]
factory = my_app.processor:make_processor
foo = 3
bar = 22
noodles.blah = one, two, three
```

The `app` section also takes a `factory`. This should be the name of a callable in your code which builds and returns your application. The part before the `:` is a Python module. The part after the `:` is the name of a callable object within that module.

The rest of the options in the `app` section of the configuration file get passed as a dictionary to your application callable. You can parse these options with *`baseplate.lib.config`*.

The application factory should return an appropriate object for your server:

Thrift A `TProcessor`.

WSGI A WSGI callable.

Logging

The baseplate server provides a default configuration for the Python standard `logging` system. The root logger will print to `stdout` with a format that includes trace information. The default log level is `INFO` or `DEBUG` if the `--debug` flag is passed to `baseplate-serve`.

If more complex logging configuration is necessary, the configuration file will override the default setup. The Configuration file format is documented in the standard library.

Automatic reload on source changes

In development, it's useful for the server to restart itself when you change code. You can do this by passing the `--reload` flag to `baseplate-serve`.

This should not be used in production environments.

Einhorn

`baseplate-serve` can run as a worker in Stripe's Einhorn socket manager. This allows Einhorn to handle binding the socket, worker management, rolling restarts, and worker health checks.

Baseplate supports Einhorn's "manual ACK" protocol. Once the application is loaded and ready to serve, Baseplate notifies the Einhorn master process via its command socket.

An example command line:

```
einhorn -m manual -n 4 --bind localhost:9190 \
   baseplate-serve myapp.ini
```

Debug Signal

Applications running under `baseplate-serve` will respond to `SIGUSR1` by printing a stack trace to the logger. This can be useful for debugging deadlocks and other issues.

Note that Einhorn will exit if you send it a `SIGUSR1`. You can instead open up `einhornsh` and instruct the master to send the signal to all workers:

```
$ einhornsh
> signal SIGUSR1
Successfully sent USR1s to 4 processes: [...]
```

Process-level metrics

If your application has registered a metrics client with *`configure_observers()`*, `baseplate-serve` will automatically send process-level metrics every 10 seconds. Which metrics are sent depends on your server configuration, for example:

```
[server:main]
factory = baseplate.server.thrift

monitoring.blocked_hub = 100 milliseconds
monitoring.concurrency = true
```

will enable the `blocked_hub` reporter (configuring it to trigger at a 100ms threshold) and the `concurrency` reporter (which has no special configuration).

The following reporters are available:

`monitoring.blocked_hub` Enabled if a valid *`Timespan()`* is set, defaults to disabled.

> This will turn on Gevent's monitoring thread and report events indicating that Gevent detects the main event loop was blocked by a greenlet for longer than the given time span. This can indicate excessive CPU usage causing event loop starvation or the use of non-patched blocking IO calls. More detailed information, including stack traces, is also printed to the logging system.
>
> Each instance of the hub being blocked will be reported as a *`Timer`* measuring the duration of the blockage.
>
> Note: the performance impact of this reporter is not currently understood. Watch your metrics closely if you turn this on.

`monitoring.concurrency` Enabled if `true`, disabled if `false`. Defaults to enabled.

> This will track the number of in-flight requests being processed concurrently by this server process.

At each report interval, this will update two *Gauge* metrics with the current number of open connections (`open_connections`) and current number of in-flight requests being processed concurrently (`active_requests`).

`monitoring.connection_pool` Enabled if `true`, disabled if `false`. Defaults to disabled.

This will track the usage of connection pools for various clients in the application. The metrics generated will depend on which clients are used.

`monitoring.gc.stats` Enabled if `true`, disabled if `false`. Defaults to enabled.

This will report the Python garbage collector's statistics to the metrics system.

At each report interval, this will update gauges with the current values returned by `gc.get_stats()`.

`monitoring.gc.timing` Enabled if `true`, disabled if `false`. Defaults to disabled.

This will track the duration of time taken by Python's garbage collector doing a collection sweep.

The duration of each pass of the garbage collector will be reported as a timer.

Note: the performance impact of this reporter is not currently understood. Watch your metrics closely if you turn this on.

`monitoring.gc.refcycle` Enabled if a path to a writable directory is set, defaults to disabled.

This should only be used in debugging, it will certainly have a negative performance impact.

This will turn off automatic garbage collection and instead run a sweep every reporting interval. Any objects found by the collector will be graphed using objgraph to help find reference cycles. The resulting graphs images will be written to the directory specified.

The `objgraph` library and `graphviz` package must be installed for this to work properly.

All metrics generated by `baseplate-serve` are prefixed with `runtime` and are tagged with `hostname` and `PID`.

Changed in version 2.0: The `hostname` and `PID` fields of the metric name were moved to tags.

1.4.3 `baseplate-script`

This command allows you to run a piece of Python code with the application configuration loaded similarly to baseplate-serve. The command is `baseplate-script`.

Command Line

There are two required arguments on the command line: the path to an INI-format configuration file, and the fully qualified name of a Python function to run.

The function should be specified as a module path, a colon, and a function name. For example, `my_service.models:create_schema`. The function should take a single argument which will be the application's configuration as a dictionary. This is the same as the application factory used by the server.

Just like with `baseplate-serve`, the `app:main` section will be loaded by default. This can be overridden with the `--app-name` option.

Example

Given a configuration file, `printer.ini`:

```
[app:main]
message = Hello!

[app:bizarro]
message = !olleH
```

and a small script, `printer.py`:

```
def run(app_config, args):
    parser = argparse.ArgumentParser()
    parser.add_argument("name")
    args = parser.parse_args(args)

    print(f"{app_config['message']} {args.name}")
```

You can run the script with various configurations:

```
$ baseplate-script printer.ini printer:run Goodbye.
Hello! Goodbye.

$ baseplate-script printer.ini --app-name=bizarro printer:run Goodbye.
!olleH Goodbye.
```

1.4.4 baseplate-shell

This command allows you to run an interactive Python shell for Baseplate.py services with the application configuration and context loaded. The command is `baseplate-shell`.

This shell can be used for any kind of Baseplate.py service: Thrift, HTTP, etc.

Command Line

This command requires the path to an INI-format configuration file to run.

Just like with `baseplate-serve`, the `app:main` section will be loaded by default. This can be overridden with the `--app-name` option.

By default, the shell will have variables containing the application and the context exposed. Additional variables can be exposed by providing a `setup` function in the `shell` (or `tshell` for backwards compatibility) section of the configuration file.

Example

Given a configuration file, `example.ini`:

```
[app:main]
factory = baseplate.server.thrift

[shell]
setup = my_service:shell_setup
```

and a small setup function, `my_service.py`:

```
def shell_setup(env, env_banner):
    from my_service import models
    env['models'] = models
    env_banner['models'] = 'Models module'
```

You can begin a shell with the models module exposed:

```
$ baseplate-shell example.ini
Baseplate Interactive Shell
Python 2.7.6 (default, Nov 23 2017, 15:49:48)
[GCC 4.8.4]

Available Objects:

  app          This project's app instance
  context      The context for this shell instance's span
  models       Models module
>>>
```

1.5 Linters

Incorporating linters into your service will enforce a coding standard and prevent errors from getting merged into your codebase. The `baseplate.lint` module consists of custom Pylint checkers which add more lint to Pylint. These lints are based on bugs found at Reddit.

1.5.1 Configuration

Getting Started

Install Pylint and ensure you have it and its dependencies added to your requirements-dev.txt file.

Follow the Pylint user guide for instructions to generate a default pylintrc configuration file and run Pylint.

Adding Custom Checkers

In your `pylintrc` file, add `baseplate.lint` to the `[MASTER]` load-plugins configuration.

```
# List of plugins (as comma separated values of python modules names) to load,
# usually to register additional checkers.
load-plugins=baseplate.lint
```

This will allow you to use all the custom checkers in the baseplate.lint module when you run Pylint.

Custom Checkers List

- W9000: no-database-query-string-format

Creating Custom Checkers

If there is something you want to lint and a checker does not already exist, you can add a new one to `baseplate.lint`.

The following is an example checker you can reference to create your own.

```
# Pylint documentation for writing a checker: http://pylint.pycqa.org/en/latest/how_
↪tos/custom_checkers.html
# This is an example of a Pylint AST checker and should not be registered to use
# In an AST (abstract syntax tree) checker, the code will be represented as nodes of␣
↪a tree
# We will use the astroid library: https://astroid.readthedocs.io/en/latest/api/
↪general.html to visit and leave nodes
# Libraries needed for an AST checker
from astroid import nodes
from pylint.checkers import BaseChecker
from pylint.interfaces import IAstroidChecker
from pylint.lint import PyLinter


# Basic example of a Pylint AST (astract syntax tree) checker
# Checks for variables that have been reassigned in a function. If it finds a␣
↪reassigned variable, it will throw an error
class NoReassignmentChecker(BaseChecker):
    __implements__ = IAstroidChecker

    # Checker name
    name = "no-reassigned-variable"
    # Set priority to -1
    priority = -1
    # Message dictionary
    msgs = {
        # message-id, consists of a letter and numbers
        # Letter will be one of following letters (C=Convention, W=Warning, E=Error,␣
↪F=Fatal, R=Refactoring)
        # Numbers need to be unique and in-between 9000-9999
        # Check https://baseplate.readthedocs.io/en/stable/linters/index.html#custom-
↪checkers-list
        # for numbers that are already in use
        "W9001": (
            # displayed-message shown to user
```

(continues on next page)

(continued from previous page)

```
            "Reassigned variable found.",
            # message-symbol used as alias for message-id
            "reassigned-variable",
            # message-help shown to user when calling pylint --help-msg
            "Ensure variables are not reassigned.",
        )
    }

    def __init__(self, linter: PyLinter = None):
        super().__init__(linter)
        self.variables: set = set()

    # The following two methods are called for us by pylint/astroid
    # The linter walks through the tree, visiting and leaving desired nodes
    # Methods should start with visit_ or leave_ followed by lowercase class name of␣
↪nodes
    # List of available nodes: https://astroid.readthedocs.io/en/latest/api/astroid.
↪nodes.html

    # Visit the Assign node: https://astroid.readthedocs.io/en/latest/api/astroid.
↪nodes.html#astroid.nodes.Assign
    def visit_assign(self, node: nodes) -> None:
        for variable in node.targets:
            if variable.name not in self.variables:
                self.variables.add(variable.name)
            else:
                self.add_message("non-unique-variable", node=node)

    # Leave the FunctionDef node: https://astroid.readthedocs.io/en/latest/api/
↪astroid.nodes.html#astroid.nodes.FunctionDef
    def leave_functiondef(self, node: nodes) -> nodes:
        self.variables = set()
        return node
```

Add a test to the baseplate test suite following this example checker test.

```
# Libraries needed for tests
import astroid
import pylint.testutils

from baseplate.lint import example_plugin


# CheckerTestCase creates a linter that will traverse the AST tree
class TestNoReassignmentChecker(pylint.testutils.CheckerTestCase):
    CHECKER_CLASS = example_plugin.NoReassignmentChecker

    # Use astroid.extract_node() to create a test case
    # Where you put #@ is where the variable gets assigned
    # example, assign_node_a = test = 1, assign_node_b = test = 2
    def test_finds_reassigned_variable(self):
        assign_node_a, assign_node_b = astroid.extract_node(
            """
        test = 1 #@
        test = 2 #@
            """
        )
```

(continues on next page)

(continued from previous page)

```
        self.checker.visit_assign(assign_node_a)
        self.checker.visit_assign(assign_node_b)
        self.assertAddsMessages(
            pylint.testutils.Message(msg_id="reassigned-variable", node=assign_node_a)
        )

    def test_ignores_no_reassigned_variable(self):
        assign_node_a, assign_node_b = astroid.extract_node(
            """
        test1 = 1 #@
        test2 = 2 #@
            """
        )

        with self.assertNoMessages():
            self.checker.visit_assign(assign_node_a)
            self.checker.visit_assign(assign_node_b)

    def test_ignores_variable_outside_function(self):
        func_node, assign_node_a, assign_node_b = astroid.extract_node(
            """
        def test1(): #@
            test = 1 #@

        def test2():
            test = 2 #@
            """
        )

        with self.assertNoMessages():
            self.checker.visit_assign(assign_node_a)
            self.checker.leave_functiondef(func_node)
            self.checker.visit_assign(assign_node_b)
```

Register your checker by adding it to the register() function:

```
from pylint.lint import PyLinter

from baseplate.lint.db_query_string_format_plugin import NoDbQueryStringFormatChecker


def register(linter: PyLinter) -> None:
    checker = NoDbQueryStringFormatChecker(linter)
    linter.register_checker(checker)
```

Lastly, add your checker message-id and name to *Custom Checkers List*.

CHAPTER

TWO

APPENDIX

- genindex
- modindex

PYTHON MODULE INDEX

b

baseplate, 19
baseplate.clients, 28
baseplate.clients.cassandra, 28
baseplate.clients.kombu, 30
baseplate.clients.memcache, 33
baseplate.clients.memcache.lib, 35
baseplate.clients.redis, 36
baseplate.clients.requests, 40
baseplate.clients.sqlalchemy, 44
baseplate.clients.thrift, 46
baseplate.frameworks, 49
baseplate.frameworks.pyramid, 50
baseplate.frameworks.queue_consumer, 52
baseplate.frameworks.queue_consumer.kafka, 52
baseplate.frameworks.queue_consumer.kombu, 57
baseplate.frameworks.thrift, 49
baseplate.lib.config, 66
baseplate.lib.crypto, 71
baseplate.lib.datetime, 73
baseplate.lib.edgecontext, 73
baseplate.lib.events, 74
baseplate.lib.file_watcher, 76
baseplate.lib.message_queue, 82
baseplate.lib.metrics, 83
baseplate.lib.random, 86
baseplate.lib.ratelimit, 87
baseplate.lib.ratelimit.backends.memcache, 89
baseplate.lib.ratelimit.backends.redis, 90
baseplate.lib.retry, 91
baseplate.lib.secrets, 92
baseplate.lib.service_discovery, 97
baseplate.lib.thrift_pool, 96
baseplate.observers, 59

INDEX

Symbols

__init__() (*baseplate.Baseplate method*), 20
__init__() (*baseplate.frameworks.queue_consumer.kafka.FastConsumerFactory method*), 55
__init__() (*baseplate.frameworks.queue_consumer.kafka.InOrderConsumerFactory method*), 54
__init__() (*baseplate.frameworks.queue_consumer.kombu.KombuQueueConsumerFactory method*), 58
__iter__() (*baseplate.lib.retry.RetryPolicy method*), 91

A

add_sample() (*baseplate.lib.metrics.Histogram method*), 86
add_to_context() (*baseplate.Baseplate method*), 21
all_versions() (*baseplate.lib.secrets.VersionedSecret property*), 95

B

Backend (*class in baseplate.lib.service_discovery*), 98
Base64() (*in module baseplate.lib.config*), 68
baseplate
    module, 19
Baseplate (*class in baseplate*), 20
baseplate.clients
    module, 28
baseplate.clients.cassandra
    module, 28
baseplate.clients.kombu
    module, 30
baseplate.clients.memcache
    module, 33
baseplate.clients.memcache.lib
    module, 35
baseplate.clients.redis
    module, 36
baseplate.clients.requests
    module, 40
baseplate.clients.sqlalchemy
    module, 44
baseplate.clients.thrift
    module, 46
baseplate.frameworks
    module, 49
baseplate.frameworks.pyramid
    module, 50
baseplate.frameworks.queue_consumer
    module, 52
baseplate.frameworks.queue_consumer.kafka
    module, 52
baseplate.frameworks.queue_consumer.kombu
    module, 57
baseplate.frameworks.thrift
    module, 49
baseplate.lib.config
    module, 66
baseplate.lib.crypto
    module, 71
baseplate.lib.datetime
    module, 73
baseplate.lib.edgecontext
    module, 73
baseplate.lib.events
    module, 74
baseplate.lib.file_watcher
    module, 76
baseplate.lib.message_queue
    module, 82
baseplate.lib.metrics
    module, 83
baseplate.lib.random
    module, 86
baseplate.lib.ratelimit
    module, 87
baseplate.lib.ratelimit.backends.memcache
    module, 89
baseplate.lib.ratelimit.backends.redis
    module, 90
baseplate.lib.retry
    module, 91
baseplate.lib.secrets
    module, 92

baseplate.lib.service_discovery
    module, 97
baseplate.lib.thrift_pool
    module, 96
baseplate.observers
    module, 59
BaseplateConfigurator (class in baseplate.frameworks.pyramid), 50
baseplateify_processor() (in module baseplate.frameworks.thrift), 49
BaseplateObserver (class in baseplate), 26
BaseplateSession (class in baseplate.clients.requests), 42
Batch (class in baseplate.lib.metrics), 84
batch() (baseplate.lib.metrics.Client method), 84
Boolean() (in module baseplate.lib.config), 68

C

CassandraClient (class in baseplate.clients.cassandra), 29
CassandraContextFactory (class in baseplate.clients.cassandra), 29
Client (class in baseplate.lib.metrics), 84
close() (baseplate.clients.redis.MessageQueue method), 39
close() (baseplate.lib.message_queue.MessageQueue method), 82
cluster_from_config() (in module baseplate.clients.cassandra), 29
ConfigurationError, 71
configure_context() (baseplate.Baseplate method), 20
configure_observers() (baseplate.Baseplate method), 20
connection() (baseplate.lib.thrift_pool.ThriftConnectionPool method), 97
connection_from_config() (in module baseplate.clients.kombu), 32
consume() (baseplate.lib.ratelimit.backends.memcache.MemcacheRateLimitBackend method), 89
consume() (baseplate.lib.ratelimit.backends.RateLimitBackend method), 88
consume() (baseplate.lib.ratelimit.backends.redis.RedisRateLimitBackend method), 90
consume() (baseplate.lib.ratelimit.RateLimiter method), 87
ContextFactory (class in baseplate.clients), 48
CorruptSecretError, 95
Counter (class in baseplate.lib.metrics), 85
counter() (baseplate.lib.metrics.Batch method), 84
counter() (baseplate.lib.metrics.Client method), 84
CQLMapperClient (class in baseplate.clients.cassandra), 29
CQLMapperContextFactory (class in baseplate.clients.cassandra), 29
CredentialSecret (class in baseplate.lib.secrets), 95
current() (baseplate.lib.secrets.VersionedSecret property), 95

D

datetime_to_epoch_milliseconds() (in module baseplate.lib.datetime), 73
datetime_to_epoch_seconds() (in module baseplate.lib.datetime), 73
decompress_and_load() (in module baseplate.clients.memcache.lib), 35
decompress_and_unpickle() (in module baseplate.clients.memcache.lib), 35
decrement() (baseplate.lib.metrics.Counter method), 85
delete() (baseplate.clients.requests.BaseplateSession method), 42
DictOf() (in module baseplate.lib.config), 69

E

EdgeContextFactory (class in baseplate.lib.edgecontext), 73
Endpoint() (in module baseplate.lib.config), 68
EndpointConfiguration (class in baseplate.lib.config), 71
engine_from_config() (in module baseplate.clients.sqlalchemy), 44
epoch_milliseconds_to_datetime() (in module baseplate.lib.datetime), 73
epoch_seconds_to_datetime() (in module baseplate.lib.datetime), 73
EventError, 76
EventQueue (class in baseplate.lib.events), 75
EventQueueFullError, 76
EventTooLargeError, 76
exchange_from_config() (in module baseplate.clients.kombu), 32
execute_command() (baseplate.clients.redis.MonitoredRedisConnection method), 38
ExpiredSignatureError, 73
ExternalRequestsClient (class in baseplate.clients.requests), 41

F

FakeFileWatcher (class in baseplate.testing.lib.file_watcher), 78
FakeSecretsStore (class in baseplate.testing.lib.secrets), 95
Fallback() (in module baseplate.lib.config), 69

FastConsumerFactory (*class in baseplate.frameworks.queue_consumer.kafka*), 54
FatalMessageHandlerError (*class in baseplate.frameworks.queue_consumer.kombu*), 59
File() (*in module baseplate.lib.config*), 68
FileWatcher (*class in baseplate.lib.file_watcher*), 77
finish() (*baseplate.ServerSpan method*), 22
finish() (*baseplate.Span method*), 24
flags() (*baseplate.TraceInfo property*), 25
Float() (*in module baseplate.lib.config*), 67
flush() (*baseplate.lib.metrics.Batch method*), 84
from_simple_secret() (*baseplate.lib.secrets.VersionedSecret class method*), 95
from_upstream() (*baseplate.lib.edgecontext.EdgeContextFactory method*), 73
from_upstream() (*baseplate.TraceInfo class method*), 25

G

Gauge (*class in baseplate.lib.metrics*), 86
gauge() (*baseplate.lib.metrics.Batch method*), 84
gauge() (*baseplate.lib.metrics.Client method*), 84
get() (*baseplate.clients.redis.MessageQueue method*), 39
get() (*baseplate.clients.requests.BaseplateSession method*), 42
get() (*baseplate.lib.message_queue.MessageQueue method*), 82
get_backend() (*baseplate.lib.service_discovery.ServiceInventory method*), 98
get_backends() (*baseplate.lib.service_discovery.ServiceInventory method*), 98
get_credentials() (*baseplate.lib.secrets.SecretsStore method*), 92
get_credentials_and_mtime() (*baseplate.lib.secrets.SecretsStore method*), 94
get_data() (*baseplate.lib.file_watcher.FileWatcher method*), 77
get_data_and_mtime() (*baseplate.lib.file_watcher.FileWatcher method*), 77
get_is_healthy_probe() (*in module baseplate.frameworks.pyramid*), 52
get_raw() (*baseplate.lib.secrets.SecretsStore method*), 92
get_raw_and_mtime() (*baseplate.lib.secrets.SecretsStore method*), 93
get_simple() (*baseplate.lib.secrets.SecretsStore method*), 93
get_simple_and_mtime() (*baseplate.lib.secrets.SecretsStore method*), 94
get_utc_now() (*in module baseplate.lib.datetime*), 73
get_vault_token() (*baseplate.lib.secrets.SecretsStore method*), 93
get_vault_url() (*baseplate.lib.secrets.SecretsStore method*), 93
get_versioned() (*baseplate.lib.secrets.SecretsStore method*), 93
get_versioned_and_mtime() (*baseplate.lib.secrets.SecretsStore method*), 94

H

head() (*baseplate.clients.requests.BaseplateSession method*), 42
HeaderTrustHandler (*class in baseplate.frameworks.pyramid*), 50
Histogram (*class in baseplate.lib.metrics*), 86
histogram() (*baseplate.lib.metrics.Batch method*), 85
histogram() (*baseplate.lib.metrics.Client method*), 84
http_adapter_from_config() (*in module baseplate.clients.requests*), 41

I

IncorrectSignatureError, 73
incr_tag() (*baseplate.ServerSpan method*), 22
incr_tag() (*baseplate.Span method*), 24
increment() (*baseplate.lib.metrics.Counter method*), 85
InOrderConsumerFactory (*class in baseplate.frameworks.queue_consumer.kafka*), 53
Integer() (*in module baseplate.lib.config*), 67
InternalRequestsClient (*class in baseplate.clients.requests*), 41

K

KombuProducer (*class in baseplate.clients.kombu*), 31
KombuProducerContextFactory (*class in baseplate.clients.kombu*), 32
KombuQueueConsumerFactory (*class in baseplate.frameworks.queue_consumer.kombu*), 57
KombuSerializer (*class in baseplate.clients.kombu*), 31
KombuThriftSerializer (*class in baseplate.clients.kombu*), 31

L

log() (*baseplate.ServerSpan method*), 23

log() (*baseplate.Span method*), 24

M

make_child() (*baseplate.ServerSpan method*), 23
make_child() (*baseplate.Span method*), 25
make_context_object() (*baseplate.Baseplate method*), 21
make_dump_and_compress_fn() (*in module baseplate.clients.memcache.lib*), 35
make_object_for_context() (*baseplate.clients.ContextFactory method*), 48
make_object_for_context() (*baseplate.clients.memcache.MemcacheContextFactory method*), 34
make_object_for_context() (*baseplate.clients.redis.RedisContextFactory method*), 38
make_object_for_context() (*baseplate.clients.requests.RequestsContextFactory method*), 43
make_object_for_context() (*baseplate.lib.ratelimit.backends.memcache.MemcacheRateLimitBackendContextFactory method*), 89
make_object_for_context() (*baseplate.lib.ratelimit.backends.redis.RedisRateLimitBackendContextFactory method*), 90
make_object_for_context() (*baseplate.lib.ratelimit.RateLimiterContextFactory method*), 88
make_object_for_context() (*baseplate.lib.secrets.SecretsStore method*), 94
make_pickle_and_compress_fn() (*in module baseplate.clients.memcache.lib*), 36
make_server_span() (*baseplate.Baseplate method*), 22
make_signature() (*in module baseplate.lib.crypto*), 72
MemcacheClient (*class in baseplate.clients.memcache*), 33
MemcacheContextFactory (*class in baseplate.clients.memcache*), 34
MemcacheRateLimitBackend (*class in baseplate.lib.ratelimit.backends.memcache*), 89
MemcacheRateLimitBackendContextFactory (*class in baseplate.lib.ratelimit.backends.memcache*), 89
MessageQueue (*class in baseplate.clients.redis*), 39
MessageQueue (*class in baseplate.lib.message_queue*), 82
MessageQueueError, 83
metrics_client_from_config() (*in module baseplate.lib.metrics*), 83
module
    baseplate, 19
    baseplate.clients, 28
    baseplate.clients.cassandra, 28
    baseplate.clients.kombu, 30
    baseplate.clients.memcache, 33
    baseplate.clients.memcache.lib, 35
    baseplate.clients.redis, 36
    baseplate.clients.requests, 40
    baseplate.clients.sqlalchemy, 44
    baseplate.clients.thrift, 46
    baseplate.frameworks, 49
    baseplate.frameworks.pyramid, 50
    baseplate.frameworks.queue_consumer, 52
    baseplate.frameworks.queue_consumer.kafka, 52
    baseplate.frameworks.queue_consumer.kombu, 57
    baseplate.frameworks.thrift, 49
    baseplate.lib.config, 66
    baseplate.lib.crypto, 71
    baseplate.lib.datetime, 73
    baseplate.lib.edgecontext, 73
    baseplate.lib.events, 74
    baseplate.lib.file_watcher, 76
    baseplate.lib.message_queue, 82
    baseplate.lib.metrics, 83
    baseplate.lib.random, 86
    baseplate.lib.ratelimit, 87
    baseplate.lib.ratelimit.backends.memcache, 89
    baseplate.lib.ratelimit.backends.redis, 90
    baseplate.lib.retry, 91
    baseplate.lib.secrets, 92
    baseplate.lib.service_discovery, 97
    baseplate.lib.thrift_pool, 96
    baseplate.observers, 59
MonitoredMemcacheConnection (*class in baseplate.clients.memcache*), 35
MonitoredRedisConnection (*class in baseplate.clients.redis*), 38

N

new() (*baseplate.frameworks.queue_consumer.kafka.FastConsumerFactor class method*), 55
new() (*baseplate.frameworks.queue_consumer.kafka.InOrderConsumerFa class method*), 53
new() (*baseplate.frameworks.queue_consumer.kombu.KombuQueueConsu class method*), 57
new() (*baseplate.lib.retry.RetryPolicy static method*), 91
new() (*baseplate.TraceInfo class method*), 25
next() (*baseplate.lib.secrets.VersionedSecret property*), 95
NoBackendsAvailableError, 98

O

on_child_span_created() (*baseplate.ServerSpanObserver method*), 26
on_child_span_created() (*baseplate.SpanObserver method*), 27
on_finish() (*baseplate.ServerSpanObserver method*), 26
on_finish() (*baseplate.SpanObserver method*), 27
on_incr_tag() (*baseplate.ServerSpanObserver method*), 27
on_incr_tag() (*baseplate.SpanObserver method*), 27
on_log() (*baseplate.ServerSpanObserver method*), 27
on_log() (*baseplate.SpanObserver method*), 27
on_server_span_created() (*baseplate.BaseplateObserver method*), 26
on_set_tag() (*baseplate.ServerSpanObserver method*), 27
on_set_tag() (*baseplate.SpanObserver method*), 27
on_start() (*baseplate.ServerSpanObserver method*), 27
on_start() (*baseplate.SpanObserver method*), 27
OneOf() (*in module baseplate.lib.config*), 69
Optional() (*in module baseplate.lib.config*), 69
options() (*baseplate.clients.requests.BaseplateSession method*), 42

P

parent_id() (*baseplate.TraceInfo property*), 25
parse_config() (*in module baseplate.lib.config*), 67
Parser (*class in baseplate.lib.config*), 71
password() (*baseplate.lib.secrets.CredentialSecret property*), 95
patch() (*baseplate.clients.requests.BaseplateSession method*), 42
Percent() (*in module baseplate.lib.config*), 68
pick() (*baseplate.lib.random.WeightedLottery method*), 87
pipeline() (*baseplate.clients.redis.MonitoredRedisConnection method*), 38
pool_from_config() (*in module baseplate.clients.memcache*), 34
pool_from_config() (*in module baseplate.clients.redis*), 37
post() (*baseplate.clients.requests.BaseplateSession method*), 42
prepare_request() (*baseplate.clients.requests.BaseplateSession method*), 43
previous() (*baseplate.lib.secrets.VersionedSecret property*), 94
put() (*baseplate.clients.redis.MessageQueue method*), 39
put() (*baseplate.clients.requests.BaseplateSession method*), 43
put() (*baseplate.lib.events.EventQueue method*), 75
put() (*baseplate.lib.message_queue.MessageQueue method*), 82
Python Enhancement Proposals
    PEP 3333, 19
    PEP 432, 16

R

RateLimitBackend (*class in baseplate.lib.ratelimit.backends*), 88
RateLimiter (*class in baseplate.lib.ratelimit*), 87
RateLimiterContextFactory (*class in baseplate.lib.ratelimit*), 88
RateLimitExceededException (*class in baseplate.lib.ratelimit*), 88
RedisClient (*class in baseplate.clients.redis*), 37
RedisContextFactory (*class in baseplate.clients.redis*), 38
RedisRateLimitBackend (*class in baseplate.lib.ratelimit.backends.redis*), 90
RedisRateLimitBackendContextFactory (*class in baseplate.lib.ratelimit.backends.redis*), 90
register() (*baseplate.Baseplate method*), 26
register() (*baseplate.ServerSpan method*), 23
register() (*baseplate.Span method*), 24
register_serializer() (*in module baseplate.clients.kombu*), 31
replace() (*baseplate.lib.metrics.Gauge method*), 86
report_runtime_metrics() (*baseplate.clients.ContextFactory method*), 48
report_runtime_metrics() (*baseplate.clients.redis.RedisContextFactory method*), 38
request() (*baseplate.clients.requests.BaseplateSession method*), 43
RequestContext (*class in baseplate*), 21
RequestsContextFactory (*class in baseplate.clients.requests*), 43
RetryPolicy (*class in baseplate.lib.retry*), 91

S

sample() (*baseplate.lib.random.WeightedLottery method*), 87
sampled() (*baseplate.TraceInfo property*), 25
SecretNotFoundError, 95
secrets_store_from_config() (*in module baseplate.lib.secrets*), 92
SecretsNotAvailableError, 95
SecretsStore (*class in baseplate.lib.secrets*), 92
send() (*baseplate.clients.requests.BaseplateSession method*), 43

send() (*baseplate.lib.metrics.Counter method*), 85
send() (*baseplate.lib.metrics.Timer method*), 85
serialize_v2_event() (*in module baseplate.lib.events*), 75
server_context() (*baseplate.Baseplate method*), 22
ServerSpan (*class in baseplate*), 22
ServerSpanInitialized (*class in baseplate.frameworks.pyramid*), 51
ServerSpanObserver (*class in baseplate*), 26
ServiceInventory (*class in baseplate.lib.service_discovery*), 98
set_tag() (*baseplate.ServerSpan method*), 23
set_tag() (*baseplate.Span method*), 24
should_trust_edge_context_payload() (*baseplate.frameworks.pyramid.HeaderTrustHandler method*), 51
should_trust_edge_context_payload() (*baseplate.frameworks.pyramid.StaticTrustHandler method*), 51
should_trust_trace_headers() (*baseplate.frameworks.pyramid.HeaderTrustHandler method*), 50
should_trust_trace_headers() (*baseplate.frameworks.pyramid.StaticTrustHandler method*), 51
SignatureError, 73
SignatureInfo (*class in baseplate.lib.crypto*), 72
Span (*class in baseplate*), 24
span_id() (*baseplate.TraceInfo property*), 25
SpanObserver (*class in baseplate*), 27
SQLAlchemyEngineContextFactory (*class in baseplate.clients.sqlalchemy*), 45
SQLAlchemySession (*class in baseplate.clients.sqlalchemy*), 44
SQLAlchemySessionContextFactory (*class in baseplate.clients.sqlalchemy*), 45
start() (*baseplate.lib.metrics.Timer method*), 85
start() (*baseplate.ServerSpan method*), 23
start() (*baseplate.Span method*), 24
StaticTrustHandler (*class in baseplate.frameworks.pyramid*), 51
stop() (*baseplate.lib.metrics.Timer method*), 85
String() (*in module baseplate.lib.config*), 67

T

thrift_pool_from_config() (*in module baseplate.lib.thrift_pool*), 96
ThriftClient (*class in baseplate.clients.thrift*), 47
ThriftConnectionPool (*class in baseplate.lib.thrift_pool*), 97
ThriftContextFactory (*class in baseplate.clients.thrift*), 47
TimedOutError, 83
Timer (*class in baseplate.lib.metrics*), 85
timer() (*baseplate.lib.metrics.Batch method*), 85
timer() (*baseplate.lib.metrics.Client method*), 84
Timespan() (*in module baseplate.lib.config*), 68
trace_id() (*baseplate.TraceInfo property*), 25
TraceInfo (*class in baseplate*), 25
transaction() (*baseplate.clients.redis.MonitoredRedisConnection method*), 39
TupleOf() (*in module baseplate.lib.config*), 69

U

UnixGroup() (*in module baseplate.lib.config*), 68
UnixUser() (*in module baseplate.lib.config*), 68
unlink() (*baseplate.clients.redis.MessageQueue method*), 39
unlink() (*baseplate.lib.message_queue.MessageQueue method*), 82
UnreadableSignatureError, 73
username() (*baseplate.lib.secrets.CredentialSecret property*), 95

V

validate_signature() (*in module baseplate.lib.crypto*), 72
VersionedSecret (*class in baseplate.lib.secrets*), 94

W

WatchedFileNotAvailableError, 78
WeightedLottery (*class in baseplate.lib.random*), 86

Y

yield_attempts() (*baseplate.lib.retry.RetryPolicy method*), 91

Z

zookeeper_client_from_config() (*in module baseplate.lib.live_data*), 79