Hurricane

Blueshoe

Oct 05, 2022

CONTENTS:

Hurricane is an initiative to fit Django perfectly with Kubernetes. It is supposed to cover many capabilities in order to run Django in a cloud-native environment, including a Tornado-powered Django application server. It was initially created by Blueshoe GmbH.

Table of Contents

CHAPTER

ONE

INTRODUCTION

Hurricane is an initiative to fit Django perfectly with Kubernetes. It is supposed to cover many capabilities in order to run Django in a cloud-native environment, including a Tornado-powered Django application server. It was initially created by Blueshoe GmbH.

Django was developed with the batteries included-approach and already handles most of the challenges around web development with grace. It was initially developed at a time when web applications got deployed and run on a server (physical or virtual). With its pragmatic design it enabled many developers to keep up with changing requirements, performance and maintenance work. However, service architectures have become quite popular for complex applications in the past few years. They provide a modular style based on the philosophy of dividing overwhelming software projects into smaller and more controllable parts. The advantage of highly specialized applications gained prominence among developers, but introduces new challenges to the IT-operation. However, with the advent of Kubernetes and the cloud-native development philosophy a couple of new possibilities emerged to run those service-based applications even better. Kubernetes is a wonderful answer for just as many IT-operation requirements as Django is for web development. The inherent monolithic design of Django can be tempting to roll out recurring operation patterns with each application. It's not about getting Django to run in a Kubernetes cluster (you may already solved this), it's about integrating Django as tightly as possible with Kubernetes in order to harness the full power of that platform. Creating the most robust, scalable and secure applications with Django by leveraging the existing expertise of our favorite framework is the main goal of this initiative.

Using a Tornado-powered application server gives several advantages compared to the standard Django application server. It is a single-threaded and at the same time a non-blocking server, that includes a builtin IO Loop from asyncio library. Django application server is blocked while waiting for the client. On the other hand a Tornado application server can handle processes asynchronously and thus is not blocked while waiting for the client or the database. This also gives the possibility to run webhooks and other asynchronous tasks directly in the application server, avoiding the usage of external asynchronous task queues such as Celery.

CHAPTER

TWO

APPLICATION SERVER

2.1 Run the application server

In order to start the Django app run the management command *serve*:

```
python manage.py serve
```

This command simply starts a Tornado-based application server ready to serve your Django application. There is no need for any other application server.

Command options for *serve*-command:

Serve Command Option	Description
`--static`	Serve collected static files
`--media`	Serve media files
`--autoreload`	Reload code on change
`--debug`	Set Tornado's Debug flag (don't confuse with Django's DEBUG=True)
`--port`	The port for Tornado to listen on (default is port 8000)
`--interface`	Set a host name for probe server
`--startup-probe`	The exposed path (default is /startup) for probes to check startup
`--readiness-probe`	The exposed path (default is /ready) for probes to check readiness
`--liveness-probe`	The exposed path (default is /alive) for probes to check liveness
`--probe-port`	The port for Tornado probe routes to listen on (default is the next port of –port)
`--req-queue-len`	Threshold of queue length of request, which is considered for readiness probe, default value is 10
`--no-probe`	Disable probe endpoint
`--no-metrics`	Disable metrics collection
`--command`	Repetitive command for adding execution of management commands before serving
`--check-migrations`	Check if all migrations were applied before starting application
`--webhook-url`	If specified, webhooks will be sent to this url
`--pycharm-host`	The host of the pycharm debug server
`--pycharm-port`	The port of the pycharm debug server. This is only used in combination with the '–pycharm-host' option
`--max-lifetime`	If specified, maximum requests after which pod is restarted

Please note: `req-queue-len` parameter is set to a default value of 10. It means, that if the length of the asynchronous tasks' queue will exceed 10, readiness probe will return the status 400 until the length of the queue gets below the `req-queue-len` value. Adjust this parameter if you want the asynchronous task queue to be larger than 10.

2.2 Django System Custom Checks

The liveness-probe endpoint invokes Django system check framework. This endpoint is called in a certain interval by Kubernetes, hence we get regular checks on the application. That's a well-suited approach to integrate custom checks (please check out our guide on how to do that, or refer to the Django documentation) and get health and sanity checks for free.

In all the subsequent examples, we use an example app `components` with an example model `Component`. Here is an example of a custom check:

```
# src/apps/components/checks.py
import logging

from django.core.checks import Error

from apps.components.models import Component

logger = logging.getLogger("hurricane")


def example_check(app_configs=None, **kwargs):
    """
    Check for existence of the MODEL Component in the database
    """

    # your check logic here
    errors = []
    logger.info("Our check has been called :]")
    if not Component.objects.filter(title="Title").exists():
        errors.append(
            Error(
                "an error",
                hint="There is no main engine in the spacecraft, it need's to exist with␣
↪the name 'Title'. "
                "Please create it in the admin or by installing the fixture.",
                id="components.E001",
            )
        )

    return errors
```

The registration of a check can be done in the configuration file of the corresponding app. For instance:

```
# apps/components/apps.py
from django.apps import AppConfig


class ComponentsConfig(AppConfig):
    default_auto_field = "django.db.models.BigAutoField"
    name = "apps.components"

    def ready(self):
        from django.core.checks import register
```

(continues on next page)

(continued from previous page)

```
    from apps.components.checks import example_check

    register(example_check, "hurricane", deploy=True)
```

In this case, the check is registered upon the readiness of the application. It means, that only after all the services of the app i.e. the database are started, the check is registered and executed. If readiness is not required, check can be registered in the main body of the config class.

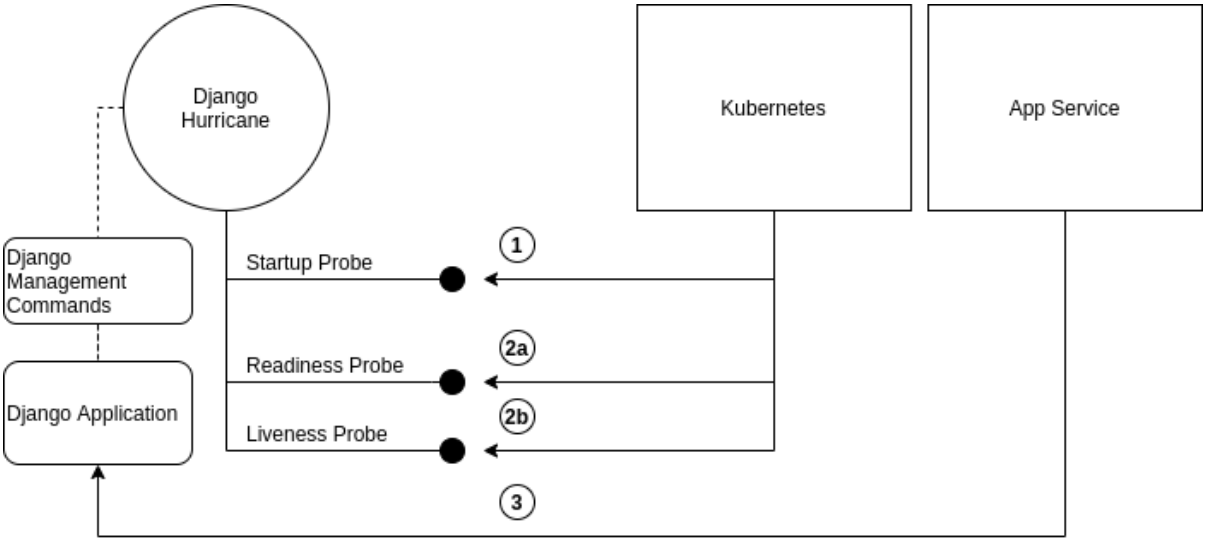
Please note: register function takes as an argument a check function and a "hurricane" tag. It is absolutely essential to register the check with this tag. Additionally *deploy=True* needs to be set.

The register function can be used as a decorator in different ways. For more information, please refer to the Django system check framework.

2.3 Probe endpoints

There are three standard probe endpoints: startup-probe, liveness-probe and readiness-probe. All probe endpoints are called regularly by Kubernetes, it allows to monitor the health and the status of the application. Upon unhealthy declared applications (error-level) Kubernetes will restart the application and remove unhealthy PODs once a new instance is in a healthy state. A port for the probe route is separated from the application's port. If the probe port is not specified, it will be set to the application port plus one e.g. if the application port is 8000, the probe port will be set to 8001. For more information about probes on a Kubernetes side, please refer to Configure Liveness, Readiness and Startup Probes.

Probe server creates handlers for three endpoints: startup, readiness and liveness.



where **1** is a Kubernetes startup probe, it returns a response with a status 400, if the application has not started yet or/and management commands are not finished yet. After finishing management commands and starting HTTP Server this endpoint will return a response of status 200 and from that point, Kubernetes will know, that the application was started, so readiness and liveness probes can be polled. **2a** and **2b** are readiness and liveness probes respectively. Kubernetes will poll these probes, only after the startup probe returns 200 for the first time. The readiness probe checks the length of the request queue, if it is larger than the threshold, it returns 400, which means, that application is not ready for further requests. The liveness probe uses Django system check framework to identify problems with the Django application. **3** are api requests, sent by the application service, which are then handled in Django application.

2.4 Management commands

Management commands can be added as options for the hurricane serve command. Kubernetes is be able to poll startup probe and if management commands are still running, it knows, that it should not restart the container yet. Management commands can be given as repeating arguments to the serve management command e.g.:

```
python manage.py serve --command makemigrations --command migrate
```

If you want to add some options to the specific management command take both this command and it's options in the quotation marks:

```
python manage.py serve --command "compilemessages --no-color"
```

Please note: management commands should be given in the order, which is required for django application. Each management command is then executed sequentially. Commands, which depend on other commands should be given after the commands they depend on. E.g. management_command_2 is depending on management_command_1, thus the serve command should look like this:

```
python manage.py serve --command management_command_1 --command management_command_2
```

Probe server, which defines handlers for every probe endpoint, runs in the main loop. Execution of management commands does not block the main event loop, as it runs in a separate executor. This way probes can be called by Kubernetes during the execution of the management commands. Upon successful execution of management commands, the HTTP server is started. If command execution was interrupted due to some error, the main loop is stopped and the HTTP server is not going to be started.

2.5 Webhooks

Webhooks can be specified as command options of *serve*-command. Right now, there are available two webhooks: startup- webhook and liveness-webhook. First is an indicator of the status of startup probe. Startup-webhook sends a status, and depending on success or failure of startup process it can send either positive or negative status. Liveness-webhook is triggered, when liveness-webhook url is specified and the liveness-probe is requested and the change of the health state is detected. For instance, if liveness probe is requested, but there was no change of the health variable, no webhook will be sent. Similarly, readiness webhook is sent upon the change of it's state variable. Webhooks run as asynchronous processes and thus do not block the asyncio-loop. If the specified url is wrong or it cannot handle webhook properly, an error or a warning will be logged. Response of the webhook should be 200 to indicate the success of receiving webhook.

Creating new webhook types The new webhook types can be specified in an easy manner in the hurricane/webhooks/webhook_types.py file. They need to specify Webhook class as a parent class. After creating a new webhook class, you can specify a new argument of the management command to parametrize the url, to which webhook will be sent. Then, you can just create an object of webhook and run it at the place in code, where it should be executed. Run method should have several methods i.e. url (to which webhook should be sent) and status (webhook on success or failure).

2.6 Check migrations

When check-migrations option is enabled, hurricane checks if database is available and subsequently checks if there are any unapplied migrations. It is executed in a separate thread, so the main thread with the probe server is not blocked.

2.7 Settings

`HURRICANE_VERSION` - is sent together with webhooks to distinguish between different versions.

2.8 Logging

It should be ensured, that the *hurricane* logger is added to Django logging configuration, otherwise log outputs will not be displayed when application server will be started. Log level can be easily adjusted to own needs.

Example:

```
LOGGING = {
    "version": 1,
    "disable_existing_loggers": True,
    "formatters": {"console":
                       {"format": "%(asctime)s %(levelname)-8s %(name)-12s %(message)s"}
                   },
    "handlers": {
        "console": {
            "class": "logging.StreamHandler",
            "formatter": "console",
            "stream": sys.stdout,
        }
    },
    "root": {"handlers": ["console"], "level": "INFO"},
    "loggers": {
        "hurricane": {
            "handlers": ["console"],
            "level": os.getenv("HURRICANE_LOG_LEVEL", "INFO"),
            "propagate": False,
        },
    },
}
```

CHAPTER

THREE

AMQP WORKER

3.1 Run the AMQP (0-9-1) Consumer

In order to start the Django-powered AMQP consumer following *consume*-command can be used:

```
python manage.py consume HANDLER
```

This command starts a Pika-based amqp consumer which is observed by Kubernetes. The required *Handler* argument is the dotted path to an *_AMQPConsumer* implementation. Please use the *TopicHandler* as base class for your handler implementation as it is the only supported exchange type at the moment. It's primarily required to implement the *on_message(...)* method to handle incoming amqp messages.

In order to establish a connection to the broker you case use one of the following options: Load from *Django Settings* or *environment variables*:

Variable	Help
AMQP_HOST	amqp broker host
AMQP_PORT	amqp broker port
AMQP_VHOST	virtual host (defaults to "/")
AMQP_USER	username for broker connection
AMQP_PASSWORD	password for broker connection

The precedence is: 1. command line option (if available), 2. Django settings, 3. environment variable

Command options for *consume*-command:

Please note: `req-queue-len` parameter is set to a default value of 10. It means, that if the length of asynchronous tasks queue will exceed 10, readiness probe will return status 400 until the length of tasks gets below the `req-queue-len` value. Adjust this parameter if you want asynchronous task queue to be larger than 10.

3.2 Example AMQP Consumer

Implementation of a basic AMQP handler with no functionality:

```
# file: myamqp/consumer.py
from hurricane.amqp.basehandler import TopicHandler

class MyTestHandler(TopicHandler):
    def on_message(self, _unused_channel, basic_deliver, properties, body):
```

(continues on next page)

(continued from previous page)

```
            print(body.decode("utf-8"))
            self.acknowledge_message(basic_deliver.delivery_tag)
```

This handler can be started using the following command:

```
python manage.py consume myamqp.consumer.MyTestHandler --queue my.test.topic --exchange␣
↪test --amqp-host 127.0.0.1 --amqp-port 5672
```

CHAPTER

FOUR

TEST HURRICANE

In order to run the entire test suite following commands should be executed:

```
shell
pip install -r requirements.txt
coverage run manage.py test
coverage combine
coverage report
```

Important: the AMQP testcase requires *Docker* to be accessible from the current user as it spins up a container with *RabbitMQ*. The AMQP consumer in a test mode will connect to it and exchange messages using the *TestPublisher* class.

CHAPTER

FIVE

DEBUGGING DJANGO APPLICATIONS

Debugging a python/django or in fact any application running in a kubernetes cluster can be cumbersome. Some of the most common IDEs use different approaches to remote debugging:

1. The Microsoft Debug Adapter Protocol (DAP) is used, among others, by Visual Studio Code and Eclipse. A full list of supporting IDE's can be found here. Here, the application itself must listen on a port and wait for the debug client (in this case: the IDE's debug UI) to connect.
2. Pycharm, which uses the pydevd debugger, sets up a debug server (you will have to configure a host and a port in your IDE debug run config) and waits for the application to connect. Therefore, the application must know where to reach the debug server.

Both approaches would usually require the application to contain code that is specific to the IDE/protocol used by the developer. Django-hurricane supports these two approaches without the need for changes to your django project:

For the Debug Adapter Protocol (Visual Studio Code, Eclipse, ...)

1. Install Django-hurricane with the "debug" option:

   ```
   pip install django-hurricane[debug]
   ```

2. Run it with the "–debugger" flag, e.g.:

   ```
   python manage.py serve --debugger
   ```

3. Optionally, provide a port (default: 5678), e.g.:

   ```
   python manage.py serve --debugger --debugger-port 1234
   ```

4. Now you can connect your IDE's remote debug client (configure the appropriate host and port).

For working with the Pycharm debugger:

1. Install Django-hurricane with the "pycharm" option:

   ```
   pip install django-hurricane[pycharm]
   ```

2. Configure the remote debug server in Pycharm and start it.
3. Run your app with the "–pycharm-host" and "–pycharm-port" flags, e.g.

   ```
   python manage.py serve --pycharm-host 127.0.0.1 --pycharm-port 1234
   ```

4. Now the app should connect to the debug server. Upon connection, the execution will halt. You must resume it from Pycharm's debugger UI.

For both approaches, you may have to configure path mappings in your IDE that map your local source code directories to the corresponding locations inside the running container (e.g. "/home/me/proj/src" -> "/app").

CHAPTER

SIX

METRICS

Hurricane comes with a small framework to collect and expose metrics about your application.

6.1 Builtin Metrics

Hurricane provides a few builtin metrics:

- request queue length
- overall request count
- average response time
- startup time metric

These metrics are collected from application start until application end. Keep in mind that these metrics do not exactly represent the current state of the application - rather the current state since the start of the process. Startup time metric is used for startup probe. It is set after all management commands were finished and HTTP server was started.

6.2 Custom Metrics

It is possible to define new custom metrics. The new metric class can inherit from StoredMetric class, which defines methods for saving metric value into the registry and for value retrieval from the registry. It should include code variable, which is used as a key for storing and retrieving value from the registry dictionary. Custom metric should be also registered in a metric registry. This can be done by adding the following lines to init file of metrics package:

```
# file: metrics/__init__.py
from hurricane.metrics.requests import <CustomMetricClass>

registry.register(<CustomMetricClass>)
```

6.3 Disable Metrics

If you'd like to disable the metric collection use the *–no-metrics* flag with the serve command:

```
python manage.py serve --no-metrics
```

CHAPTER

SEVEN

TODOS

Application server

- [x] Basic setup, POC, logging
- [x] Different endpoints for all Kubernetes probes
- [x] Extensive documentation
- [x] Django management command execution before serving
- [ ] actual Tornado integration (currently uses the *tornado.wsgi.WSGIContainer*)
- [ ] web sockets with Django 3
- [ ] Testing, testing in production
- [ ] Load-testing, automated performance regression testing
- [ ] Implement the Kubernetes Metrics API
- [x] Implement hooks for calling webservices (e.g. for deployment or health state changes)
- [ ] Add another metrics collector endpoint (e.g Prometheus)

Celery

- [ ] Concept draft
- [ ] Kubernetes health probes for celery workers
- [ ] Kubernetes health probes for celery beat
- [ ] Implement hooks for calling webservices (e.g. for deployment or health state changes)
- [ ] Implement the Kubernetes Metrics API

AMQP

- [x] Concept draft
- [ ] Kubernetes health probes for amqp workers
- [ ] Implement hooks for calling webservices (e.g. for deployment or health state changes)
- [ ] Implement the Kubernetes Metrics API

Guidelines

- [ ] Concept draft
- [ ] Cookiecutter template
- [ ] Container (Docker) best-practices

CHAPTER

EIGHT

API REFERENCE

Here you find detailed descriptions of specific functions and classes.

8.1 hurricane.management

8.1.1 hurricane.management.commands.serve

class hurricane.management.commands.serve.**Command**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Start a Tornado-powered Django web server by using `python manage.py serve <arguments>`.

It can run Django management commands with the `--command` flag, that will be executed asynchronously. The application server will only be started upon successful execution of management commands. During execution of management commands the startup probe responds with a status 400.

Arguments:

- `--static` - serve collected static files
- `--media` - serve media files
- `--autoreload` - reload code on change
- `--debug` - set Tornado's Debug flag
- `--port` - the port for Tornado to listen on
- `--startup-probe` - the exposed path (default is /startup) for probes to check startup
- `--readiness-probe` - the exposed path (default is /ready) for probes to check readiness
- `--liveness-probe` - the exposed path (default is /alive) for probes to check liveness
- `--probe-port` - the port for Tornado probe route to listen on
- `--req-queue-len` - threshold of length of queue of request, which is considered for readiness probe
- `--no-probe` - disable probe endpoint
- `--no-metrics` - disable metrics collection
- `--command` - repetitive command for adding execution of management commands before serving
- `--check-migrations` - check if all migrations were applied before starting application
- `--webhook-url`- If specified, webhooks will be sent to this url
- `--max-lifetime`- If specified, maximum requests after which pod is restarted

- `--static-watch` - If specified, static files will be watched for changes and recollected

add_arguments(*parser*)

Defines arguments, that can be accepted with `serve` command.

handle(**args*, ***options*)

Defines functionalities for different arguments. After all arguments were processed, it starts the async event loop.

8.1.2 hurricane.management.commands.consume

class hurricane.management.commands.consume.**Command**(*stdout=None*, *stderr=None*, *no_color=False*, *force_color=False*)

Starting a Tornado-powered Django AMQP 0-9-1 consumer. Implements consume command as a management command for django application. The new command can be called using `python manage.py consume <arguments>`. Arguments:

- `--queue` - the AMQP 0-9-1 queue to consume from
- `--exchange` - the AMQP 0-9-1 exchange to declare
- `--amqp-port` - the message broker connection port
- `--amqp-host` - the host address of the message broker
- `--amqp-vhost` - the virtual host of the message broker to use with this consumer
- `--handler` - the Hurricane AMQP handler class (dotted path)
- `--startup-probe` - the exposed path (default is /startup) for probes to check startup
- `--readiness-probe` - the exposed path (default is /ready) for probes to check readiness
- `--liveness-probe` - the exposed path (default is /alive) for probes to check liveness
- `--probe-port` - the port for Tornado probe route to listen on
- `--req-queue-len` - threshold of length of queue of request, which is considered for readiness probe
- `--no-probe` - disable probe endpoint
- `--no-metrics` - disable metrics collection
- `--autoreload` - reload code on change
- `--debug` - set Tornado's Debug flag
- `--reconnect` - try to reconnect this client automatically as the broker is available again
- `--max-lifetime`- If specified, maximum requests after which pod is restarted

add_arguments(*parser*)

Defines arguments, that can be accepted with `consume` command.

handle(**args*, ***options*)

Defines functionalities for different arguments. After all arguments were processed, it starts the async event loop.

8.2 hurricane.server

8.2.1 hurricane.server.django

class hurricane.server.django.**DjangoHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

This handler transmits all standard requests to django application. Currently it uses WSGI Container based on tornado WSGI Container.

initialize()

Initialization of Hurricane WSGI Container.

async prepare() → None

Transmitting incoming request to django application via WSGI Container.

class hurricane.server.django.**DjangoLivenessHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

This handler runs with every call to the probe endpoint which is supposed to be used

class hurricane.server.django.**DjangoProbeHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

Parent class for all specific probe handlers.

compute_etag()

Computes the etag header to be used for this request.

By default uses a hash of the content written so far.

May be overridden to provide custom etag implementations, or may return None to disable tornado's default etag support.

async get()

Get method, which runs the check.

async post()

Post method, which runs the check.

set_extra_headers(*path*)

Setting of extra headers for cache-control, namely: no-store, no-cache, must-revalidate and max-age=0. It means that information on requests and responses will not be stored.

class hurricane.server.django.**DjangoReadinessHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

This handler runs with every call to the probe endpoint which is supposed to be used with Kubernetes 'Readiness Probes'. The DjangoCheckHandler calls Django's Check Framework which can be used to determine the application's health state during its operation.

class hurricane.server.django.**DjangoStartupHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

This handler runs with every call to the probe endpoint which is supposed to be used with Kubernetes 'Startup Probes'. It returns 400 response for post and get requests, if StartupTimeMetric is not set, what means that the application is still in the startup phase. As soon as StartupTimeMetric is set, this handler returns 200 response

upon request, which indicates, that startup phase is finished and Kubernetes can now poll liveness/readiness probes.

class hurricane.server.django.**DjangoStaticFilesHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs: Any*)

This handler transmits all static requests to django application. Currently it uses WSGI Container based on tornado WSGI Container.

initialize()

Initialization of Hurricane WSGI Container.

8.2.2 hurricane.server.wsgi

class hurricane.server.wsgi.**HurricaneWSGIContainer**(*handler*, *wsgi_application*)

Wrapper for the tornado WSGI Container, which creates a WSGI-compatible function runnable on Tornado's HTTP server. Additionally to tornado WSGI Container should be initialized with the specific handler.

exception hurricane.server.wsgi.**HurricaneWSGIException**

8.3 hurricane.metrics

8.3.1 hurricane.metrics.base

class hurricane.metrics.base.**AverageMetric**(*code=None*, *initial=None*)

Calculating average of a metric.

classmethod add_value(*value*)

Implements the running (online) average of a metric.

class hurricane.metrics.base.**CounterMetric**(*code=None*, *initial=None*)

Metric, that can be incremented and decremented.

classmethod decrement()

Decrement value from the metric.

classmethod increment()

Increment value to the metric.

class hurricane.metrics.base.**StoredMetric**(*code=None*, *initial=None*)

Base class for storing metrics in registry.

classmethod get()

Getting value of metric from registry.

classmethod get_from_registry()

Getting metric from registry using metric code.

classmethod set(*value*)

Setting new value for metric.

8.3.2 hurricane.metrics.registry

class hurricane.metrics.registry.**MetricsRegistry**

Registering metrics and storing them in a metrics dictionary.

8.3.3 hurricane.metrics.requests

class hurricane.metrics.requests.**HealthMetric**(*code=None*, *initial=None*)

class hurricane.metrics.requests.**ReadinessMetric**(*code=None*, *initial=None*)

class hurricane.metrics.requests.**RequestCounterMetric**(*code=None*, *initial=None*)

Defines request counter metric with corresponding metric code.

class hurricane.metrics.requests.**RequestQueueLengthMetric**(*code=None*)

Defines request queue length metric with corresponding metric code.

get_value()

Getting length of the asyncio queue of all tasks.

class hurricane.metrics.requests.**ResponseTimeAverageMetric**(*code=None*, *initial=None*)

Defines response time average metric with corresponding metric code.

class hurricane.metrics.requests.**StartupTimeMetric**(*code=None*, *initial=None*)

8.3.4 hurricane.metrics.exceptions

exception hurricane.metrics.exceptions.**MetricIdAlreadyRegistered**

Exception class for the case, that metric was already registered and should not be registered twice.

8.4 hurricane.amqp

8.4.1 hurricane.amqp.basehandler

class hurricane.amqp.basehandler.**TopicHandler**(*queue_name: str*, *exchange_name: str*, *host: str*, *port: int*, *vhost: Optional[str] = None*, *username: Optional[str] = None*, *password: Optional[str] = None*)

This handler implements Hurricane's base AMQP consumer that handles unexpected interactions with the message broker such as channel and connection closures. The `EXCHANGE_TYPE` is *topic*.

8.4.2 hurricane.amqp.worker

class hurricane.amqp.worker.**AMQPClient**(*consumer_klass: Type[hurricane.amqp.basehandler._AMQPConsumer]*, *queue_name: str*, *exchange_name: str*, *amqp_host: str*, *amqp_port: int*, *amqp_vhost: str*)

This is the AMQP Client that will reconnect, if the nested handler instance indicates that a reconnect is necessary.

run(*reconnect: bool = False*) → None

If reconnect is True, AMQP consumer is running in auto-connect mode. In this case consumer will be executed. If any exception occurs, consumer will be disconnected and after some delay will be reconnected. Then consumer will be restarted. KeyboardInterrupt exception is handled differently and stops consumer. In this case IOLoop will be terminated.

If reconnect is false, consumer will be started, but no exceptions and interruptions will be tolerated.

8.5 hurricane.webhooks

8.5.1 hurricane.webhooks.base

class hurricane.webhooks.base.**Webhook**(*code=None*)

Base class for webhooks in the registry. Run function initiates sending of webhook to the specified url.

classmethod get_from_registry()

Getting webhook from registry using the code.

run(*url: str*, *status:* hurricane.webhooks.base.WebhookStatus, *error_trace: Optional[str] = None*, *close_loop: bool = False*, *loop=None*)

Initiates the sending of webhook in an asynchronous manner. Also specifies the callback of the async process, which handles the feedback and either logs success or failure of a webhook sending process.

url : Url, which webhook should be sent to status : can be either WebhookStatus.FAILED or WebhookStatus.SUCCEEDED depending on the success or failure of the process, which should be indicated by the webhook error_trace : specifies the error trace of the preceding failure close_loop : specifies, whether the main loop should be closed or be left running

class hurricane.webhooks.base.**WebhookStatus**(*value*)

An enumeration.

8.5.2 hurricane.webhooks.registry

class hurricane.webhooks.registry.**WebhookRegistry**

Registering webhooks and storing them in a webhooks dictionary.

8.5.3 hurricane.webhooks.webhook_types

class hurricane.webhooks.webhook_types.**LivenessWebhook**

class hurricane.webhooks.webhook_types.**ReadinessWebhook**

class hurricane.webhooks.webhook_types.**StartupWebhook**

8.5.4 hurricane.webhooks.exceptions

exception hurricane.webhooks.exceptions.**WebhookCodeAlreadyRegistered**

Exception class for the case, that metric was already registered and should not be registered twice.

8.6 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

I

L

M

P

R

S

T

W