This document describes the current stable version of Celery (4.2). For development docs, go here.

Extensions and Bootsteps

Custom Message Consumers

You may want to embed custom Kombu consumers to manually process your messages.

For that purpose a special `ConsumerStep` bootstep class exists, where you only need to define the `get_consumers` method, that must return a list of `kombu.Consumer` objects to start whenever the connection is established:

Previous topic

Testing with Celery

Next topic

Configuration and defaults

This Page

Show Source

Quick search

Go

```python
from celery import Celery
from celery import bootsteps
from kombu import Consumer, Exchange, Queue

my_queue = Queue('custom', Exchange('custom'), 'routing_key')

app = Celery(broker='amqp://')


class MyConsumerStep(bootsteps.ConsumerStep):

    def get_consumers(self, channel):
        return [Consumer(channel,
                         queues=[my_queue],
                         callbacks=[self.handle_message],
                         accept=['json'])]

    def handle_message(self, body, message):
        print('Received message: {0!r}'.format(body))
        message.ack()
app.steps['consumer'].add(MyConsumerStep)

def send_me_a_message(who, producer=None):
    with app.producer_or_acquire(producer) as producer:
        producer.publish(
            {'hello': who},
            serializer='json',
            exchange=my_queue.exchange,
            routing_key='routing_key',
            declare=[my_queue],
            retry=True,
        )

if __name__ == '__main__':
    send_me_a_message('world!')
```

Note:

Kombu Consumers can take use of two different message callback dispatching mechanisms. The first one is the `callbacks` argument that accepts a list of callbacks with a `(body, message)` signature, the second one is the `on_message` argument that takes a single callback with a `(message,)` signature. The latter won't automatically decode and deserialize the payload.

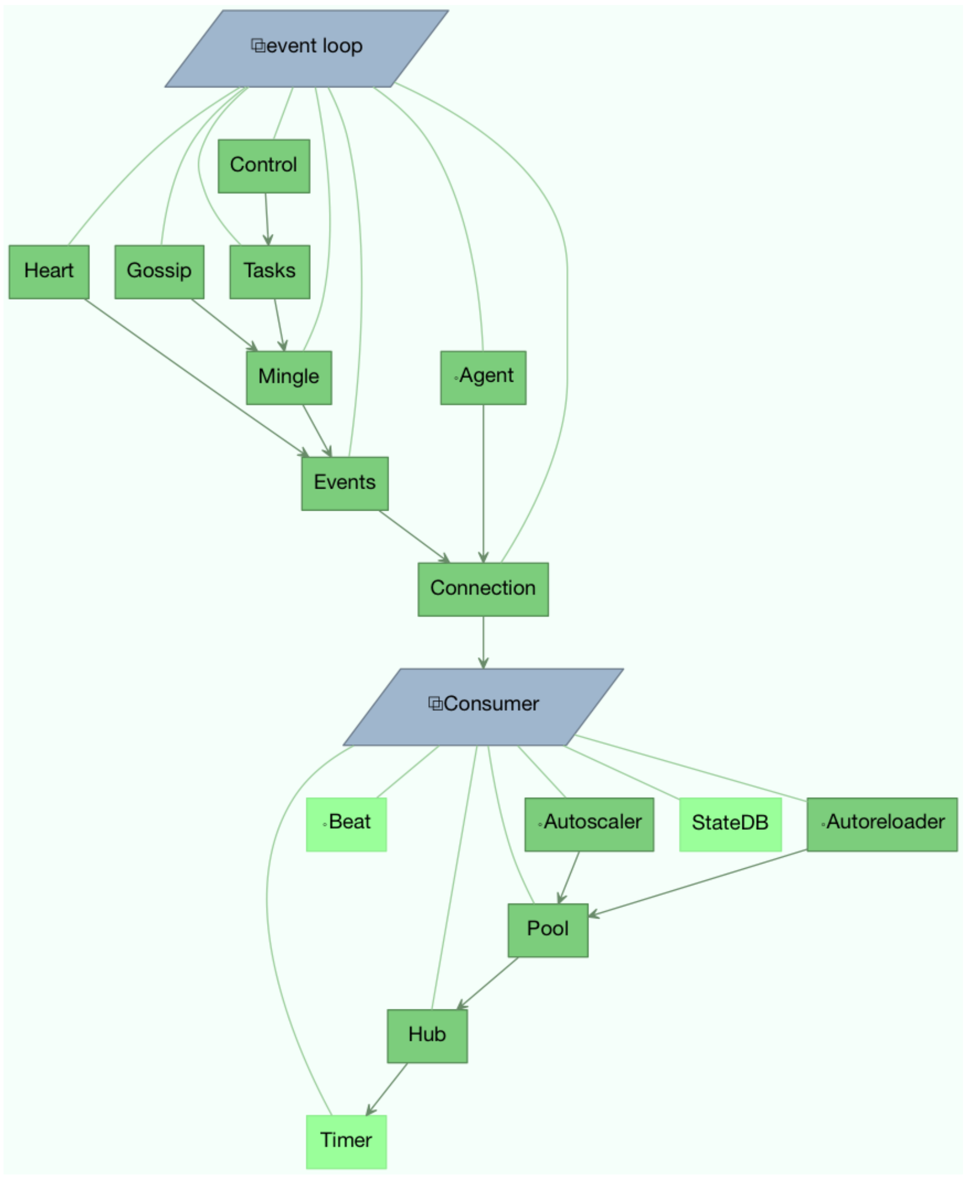
```python
def get_consumers(self, channel):
    return [Consumer(channel, queues=[my_queue],
                     on_message=self.on_message)]


def on_message(self, message):
    payload = message.decode()
    print(
        'Received message: {0!r} {props!r} rawlen={s}'.format(
        payload, props=message.properties, s=len(message.body),
    ))
    message.ack()
```

Blueprints

Bootsteps is a technique to add functionality to the workers. A bootstep is a custom class that defines hooks to do custom actions at different stages in the worker. Every bootstep belongs to a blueprint, and the worker currently defines two blueprints: **Worker**, and **Consumer**

Figure A: Bootsteps in the Worker and Consumer blueprints. Starting from the bottom up the first step in the worker blueprint is the Timer, and the last step is to start the Consumer blueprint, that then establishes the broker connection and starts consuming messages.



Worker

The Worker is the first blueprint to start, and with it starts major components like the event loop, processing pool, and the timer used for ETA tasks and other timed events.

When the worker is fully started it continues with the Consumer blueprint, that sets up how tasks are executed, connects to the broker and starts the message consumers.

The `WorkController` is the core worker implementation, and contains several methods and attributes that you can use in your bootstep.

Attributes

`app`

The current app instance.

`hostname`

The workers node name (e.g., *worker1@example.com*)

blueprint

This is the worker **Blueprint**.

hub

Event loop object (**Hub**). You can use this to register callbacks in the event loop.

This is only supported by async I/O enabled transports (amqp, redis), in which case the *worker.use_eventloop* attribute should be set.

Your worker bootstep must require the Hub bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Hub'}
```

pool

The current process/eventlet/gevent/thread pool. See **celery.concurrency.base.BasePool**.

Your worker bootstep must require the Pool bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Pool'}
```

timer

Timer used to schedule functions.

Your worker bootstep must require the Timer bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Timer'}
```

statedb

Database <celery.worker.state.Persistent>` to persist state between worker restarts.

This is only defined if the `statedb` argument is enabled.

Your worker bootstep must require the `Statedb` bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Statedb'}
```

autoscaler

Autoscaler used to automatically grow and shrink the number of processes in the pool.

This is only defined if the `autoscale` argument is enabled.

Your worker bootstep must require the *Autoscaler* bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = ('celery.worker.autoscaler:Autoscaler',)
```

autoreloader

Autoreloader used to automatically reload use code when the file-system changes.

This is only defined if the `autoreload` argument is enabled. Your worker bootstep must require the *Autoreloader* bootstep to use this;

```
class WorkerStep(bootsteps.StartStopStep):
    requires = ('celery.worker.autoreloader:Autoreloader',)
```

Example worker bootstep

An example Worker bootstep could be:

```python
from celery import bootsteps

class ExampleWorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Pool'}

    def __init__(self, worker, **kwargs):
        print('Called when the WorkController instance is constructed')
        print('Arguments to WorkController: {0!r}'.format(kwargs))

    def create(self, worker):
        # this method can be used to delegate the action methods
        # to another object that implements ``start`` and ``stop``.
        return self

    def start(self, worker):
        print('Called when the worker is started.')

    def stop(self, worker):
        print('Called when the worker shuts down.')

    def terminate(self, worker):
        print('Called when the worker terminates')
```

Every method is passed the current `WorkController` instance as the first argument.

Another example could use the timer to wake up at regular intervals:

```python
from celery import bootsteps


class DeadlockDetection(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Timer'}

    def __init__(self, worker, deadlock_timeout=3600):
        self.timeout = deadlock_timeout
        self.requests = []
        self.tref = None

    def start(self, worker):
        # run every 30 seconds.
        self.tref = worker.timer.call_repeatedly(
            30.0, self.detect, (worker,), priority=10,
        )

    def stop(self, worker):
        if self.tref:
            self.tref.cancel()
            self.tref = None

    def detect(self, worker):
        # update active requests
        for req in worker.active_requests:
            if req.time_start and time() - req.time_start > self.timeout
                raise SystemExit()
```

Consumer

The Consumer blueprint establishes a connection to the broker, and is restarted every time this connection is lost. Consumer bootsteps include the worker heartbeat, the remote control command consumer, and importantly, the task consumer.

When you create consumer bootsteps you must take into account that it must be possible to restart your blueprint. An additional 'shutdown' method is defined for consumer bootsteps, this method is called when the worker is shutdown.

Attributes

`app`

The current app instance.

`controller`

The parent **`WorkController`** object that created this consumer.

`hostname`

The workers node name (e.g., *worker1@example.com*)

`blueprint`

This is the worker **`Blueprint`**.

`hub`

Event loop object (**`Hub`**). You can use this to register callbacks in the event loop.

This is only supported by async I/O enabled transports (amqp, redis), in which case the *worker.use_eventloop* attribute should be set.

Your worker bootstep must require the Hub bootstep to use this:

```
class WorkerStep(bootsteps.StartStopStep):
    requires = {'celery.worker.components:Hub'}
```

`connection`

The current broker connection (**`kombu.Connection`**).

A consumer bootstep must require the 'Connection' bootstep to use this:

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.connection:Connection'}
```

`event_dispatcher`

A **`app.events.Dispatcher`** object that can be used to send events.

A consumer bootstep must require the *Events* bootstep to use this.

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.events:Events'}
```

`gossip`

Worker to worker broadcast communication (**`Gossip`**).

A consumer bootstep must require the *Gossip* bootstep to use this.

```
class RatelimitStep(bootsteps.StartStopStep):
    """Rate limit tasks based on the number of workers in the
    cluster."""
    requires = {'celery.worker.consumer.gossip:Gossip'}

    def start(self, c):
        self.c = c
        self.c.gossip.on.node_join.add(self.on_cluster_size_change)
        self.c.gossip.on.node_leave.add(self.on_cluster_size_change)
        self.c.gossip.on.node_lost.add(self.on_node_lost)
        self.tasks = [
            self.app.tasks['proj.tasks.add']
            self.app.tasks['proj.tasks.mul']
        ]
        self.last_size = None

    def on_cluster_size_change(self, worker):
        cluster_size = len(list(self.c.gossip.state.alive_workers()))
        if cluster_size != self.last_size:
            for task in self.tasks:
                task.rate_limit = 1.0 / cluster_size
            self.c.reset_rate_limits()
            self.last_size = cluster_size

    def on_node_lost(self, worker):
        # may have processed heartbeat too late, so wake up soon
        # in order to see if the worker recovered.
        self.c.timer.call_after(10.0, self.on_cluster_size_change)
```

Callbacks

- `<set> gossip.on.node_join`

  Called whenever a new node joins the cluster, providing a `Worker` instance.

- `<set> gossip.on.node_leave`

  Called whenever a new node leaves the cluster (shuts down), providing a `Worker` instance.

- `<set> gossip.on.node_lost`

  Called whenever heartbeat was missed for a worker instance in the cluster (heartbeat not received or processed in time), providing a `Worker` instance.

  This doesn't necessarily mean the worker is actually offline, so use a time out mechanism if the default heartbeat timeout isn't sufficient.

`pool`

The current process/eventlet/gevent/thread pool. See `celery.concurrency.base.BasePool`.

`timer`

`Timer <celery.utils.timer2.Schedule` used to schedule functions.

`heart`

Responsible for sending worker event heartbeats (`Heart`).

Your consumer bootstep must require the *Heart* bootstep to use this:

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.heart:Heart'}
```

`task_consumer`

The `kombu.Consumer` object used to consume task messages.

Your consumer bootstep must require the *Tasks* bootstep to use this:

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.tasks:Tasks'}
```

`strategies`

Every registered task type has an entry in this mapping, where the value is used to execute an incoming message of this task type (the task execution strategy). This mapping is generated by the Tasks bootstep when the consumer starts:

```
for name, task in app.tasks.items():
    strategies[name] = task.start_strategy(app, consumer)
    task.__trace__ = celery.app.trace.build_tracer(
        name, task, loader, hostname
    )
```

Your consumer bootstep must require the *Tasks* bootstep to use this:

```
class Step(bootsteps.StartStopStep):
    requires = {'celery.worker.consumer.tasks:Tasks'}
```

`task_buckets`

A **`defaultdict`** used to look-up the rate limit for a task by type. Entries in this dict may be None (for no limit) or a **`TokenBucket`** instance implementing `consume(tokens)` and `expected_time(tokens)`.

TokenBucket implements the token bucket algorithm, but any algorithm may be used as long as it conforms to the same interface and defines the two methods above.

`qos`

The **`QoS`** object can be used to change the task channels current prefetch_count value:

```
# increment at next cycle
consumer.qos.increment_eventually(1)
# decrement at next cycle
consumer.qos.decrement_eventually(1)
consumer.qos.set(10)
```

Methods

`consumer.`**`reset_rate_limits`**()

Updates the `task_buckets` mapping for all registered task types.

`consumer.`**`bucket_for_task`**(*type*, *Bucket=TokenBucket*)

Creates rate limit bucket for a task using its `task.rate_limit` attribute.

`consumer.add_task_queue(name, exchange=None, exchange_type=None, routing_key=None, **options):`

Adds new queue to consume from. This will persist on connection restart.

`consumer.`**`cancel_task_queue`**(*name*)

Stop consuming from queue by name. This will persist on connection restart.

`apply_eta_task`(*request*)

Schedule ETA task to execute based on the `request.eta` attribute. (**`Request`**)

Installing Bootsteps

`app.steps['worker']` and `app.steps['consumer']` can be modified to add new bootsteps:

```
>>> app = Celery()
>>> app.steps['worker'].add(MyWorkerStep)  # < add class, don't instanti
>>> app.steps['consumer'].add(MyConsumerStep)

>>> app.steps['consumer'].update([StepA, StepB])

>>> app.steps['consumer']
{step:proj.StepB{()}, step:proj.MyConsumerStep{()}, step:proj.StepA{()}
```

The order of steps isn't important here as the order is decided by the resulting dependency graph (`Step.requires`).

To illustrate how you can install bootsteps and how they work, this is an example step that prints some useless debugging information. It can be added both as a worker and consumer bootstep:

```python
from celery import Celery
from celery import bootsteps

class InfoStep(bootsteps.Step):

    def __init__(self, parent, **kwargs):
        # here we can prepare the Worker/Consumer object
        # in any way we want, set attribute defaults, and so on.
        print('{0!r} is in init'.format(parent))

    def start(self, parent):
        # our step is started together with all other Worker/Consumer
        # bootsteps.
        print('{0!r} is starting'.format(parent))

    def stop(self, parent):
        # the Consumer calls stop every time the consumer is
        # restarted (i.e., connection is lost) and also at shutdown.
        # The Worker will call stop at shutdown only.
        print('{0!r} is stopping'.format(parent))

    def shutdown(self, parent):
        # shutdown is called by the Consumer at shutdown, it's not
        # called by Worker.
        print('{0!r} is shutting down'.format(parent))

    app = Celery(broker='amqp://')
    app.steps['worker'].add(InfoStep)
    app.steps['consumer'].add(InfoStep)
```

Starting the worker with this step installed will give us the following logs:

```
<Worker: w@example.com (initializing)> is in init
<Consumer: w@example.com (initializing)> is in init
[2013-05-29 16:18:20,544: WARNING/MainProcess]
    <Worker: w@example.com (running)> is starting
[2013-05-29 16:18:21,577: WARNING/MainProcess]
    <Consumer: w@example.com (running)> is starting
<Consumer: w@example.com (closing)> is stopping
<Worker: w@example.com (closing)> is stopping
<Consumer: w@example.com (terminating)> is shutting down
```

The `print` statements will be redirected to the logging subsystem after the worker has been initialized, so the "is starting" lines are time-stamped. You may notice that this does no longer happen at shutdown, this is because the `stop` and `shutdown` methods are called inside a *signal handler*, and it's not safe to use logging inside such a handler. Logging with the Python logging module isn't reentrant: meaning you cannot interrupt the function then call it again later. It's important that the `stop` and `shutdown` methods you write is also reentrant.

Starting the worker with `--loglevel=debug` will show us more information about the boot process:

```
[2013-05-29 16:18:20,509: DEBUG/MainProcess] | Worker: Preparing bootste
[2013-05-29 16:18:20,511: DEBUG/MainProcess] | Worker: Building graph...
<celery.apps.worker.Worker object at 0x101ad8410> is in init
[2013-05-29 16:18:20,511: DEBUG/MainProcess] | Worker: New boot order:
    {Hub, Pool, Timer, StateDB, Autoscaler, InfoStep, Beat, Consumer}
[2013-05-29 16:18:20,514: DEBUG/MainProcess] | Consumer: Preparing boots
[2013-05-29 16:18:20,514: DEBUG/MainProcess] | Consumer: Building graph.
<celery.worker.consumer.Consumer object at 0x101c2d8d0> is in init
[2013-05-29 16:18:20,515: DEBUG/MainProcess] | Consumer: New boot order:
    {Connection, Mingle, Events, Gossip, InfoStep, Agent,
     Heart, Control, Tasks, event loop}
[2013-05-29 16:18:20,522: DEBUG/MainProcess] | Worker: Starting Hub
[2013-05-29 16:18:20,522: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,522: DEBUG/MainProcess] | Worker: Starting Pool
[2013-05-29 16:18:20,542: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,543: DEBUG/MainProcess] | Worker: Starting InfoStep
[2013-05-29 16:18:20,544: WARNING/MainProcess]
    <celery.apps.worker.Worker object at 0x101ad8410> is starting
[2013-05-29 16:18:20,544: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,544: DEBUG/MainProcess] | Worker: Starting Consumer
[2013-05-29 16:18:20,544: DEBUG/MainProcess] | Consumer: Starting Connec
[2013-05-29 16:18:20,559: INFO/MainProcess] Connected to amqp://guest@12
[2013-05-29 16:18:20,560: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:20,560: DEBUG/MainProcess] | Consumer: Starting Mingle
[2013-05-29 16:18:20,560: INFO/MainProcess] mingle: searching for neighb
[2013-05-29 16:18:21,570: INFO/MainProcess] mingle: no one here
[2013-05-29 16:18:21,570: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,571: DEBUG/MainProcess] | Consumer: Starting Events
[2013-05-29 16:18:21,572: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,572: DEBUG/MainProcess] | Consumer: Starting Gossip
[2013-05-29 16:18:21,577: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,577: DEBUG/MainProcess] | Consumer: Starting InfoSt
[2013-05-29 16:18:21,577: WARNING/MainProcess]
    <celery.worker.consumer.Consumer object at 0x101c2d8d0> is starting
[2013-05-29 16:18:21,578: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,578: DEBUG/MainProcess] | Consumer: Starting Heart
[2013-05-29 16:18:21,579: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,579: DEBUG/MainProcess] | Consumer: Starting Contro
[2013-05-29 16:18:21,583: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,583: DEBUG/MainProcess] | Consumer: Starting Tasks
[2013-05-29 16:18:21,606: DEBUG/MainProcess] basic.qos: prefetch_count->
[2013-05-29 16:18:21,606: DEBUG/MainProcess] ^-- substep ok
[2013-05-29 16:18:21,606: DEBUG/MainProcess] | Consumer: Starting event
[2013-05-29 16:18:21,608: WARNING/MainProcess] celery@example.com ready.
```

Command-line programs

Adding new command-line options

Command-specific options

You can add additional command-line options to the `worker`, `beat`, and `events` commands by modifying the `user_options` attribute of the application instance.

Celery commands uses the `argparse` module to parse command-line arguments, and so to add custom arguments you need to specify a callback that takes a `argparse.ArgumentParser` instance - and adds arguments. Please see the `argparse` documentation to read about the fields supported.

Example adding a custom option to the **celery worker** command:

```
from celery import Celery

app = Celery(broker='amqp://')

def add_worker_arguments(parser):
    parser.add_argument(
        '--enable-my-option', action='store_true', default=False,
        help='Enable custom option.',
    ),
app.user_options['worker'].add(add_worker_arguments)
```

All bootsteps will now receive this argument as a keyword argument to `Bootstep.__init__`:

```
from celery import bootsteps

class MyBootstep(bootsteps.Step):

    def __init__(self, worker, enable_my_option=False, **options):
        if enable_my_option:
            party()

app.steps['worker'].add(MyBootstep)
```

Preload options

The **celery** umbrella command supports the concept of 'preload options'. These are special options passed to all sub-commands and parsed outside of the main parsing step.

The list of default preload options can be found in the API reference: **`celery.bin.base`**.

You can add new preload options too, for example to specify a configuration template:

```
from celery import Celery
from celery import signals
from celery.bin import Option

app = Celery()

def add_preload_options(parser):
    parser.add_argument(
        '-Z', '--template', default='default',
        help='Configuration template to use.',
    )
app.user_options['preload'].add(add_preload_options)

@signals.user_preload_options.connect
def on_preload_parsed(options, **kwargs):
    use_template(options['template'])
```

Adding new **celery** sub-commands

New commands can be added to the **celery** umbrella command by using setuptools entry-points.

Entry-points is special meta-data that can be added to your packages `setup.py` program, and then after installation, read from the system using the **`pkg_resources`** module.

Celery recognizes `celery.commands` entry-points to install additional sub-commands, where the value of the entry-point must point to a valid subclass of **`celery.bin.base.Command`**. There's limited documentation, unfortunately, but you can find inspiration from the various commands in the **`celery.bin`** package.

This is how the Flower monitoring extension adds the **celery flower** command, by adding an entry-point in `setup.py`:

```
setup(
    name='flower',
    entry_points={
        'celery.commands': [
           'flower = flower.command:FlowerCommand',
        ],
    }
)
```

The command definition is in two parts separated by the equal sign, where the first part is the name of the sub-command (flower), then the second part is the fully qualified symbol path to the class that implements the command:

```
flower.command:FlowerCommand
```

The module path and the name of the attribute should be separated by colon as above.

In the module `flower/command.py`, the command class is defined something like this:

```
from celery.bin.base import Command


class FlowerCommand(Command):

    def add_arguments(self, parser):
        parser.add_argument(
            '--port', default=8888, type='int',
            help='Webserver port',
        ),
        parser.add_argument(
            '--debug', action='store_true',
        )

    def run(self, port=None, debug=False, **kwargs):
        print('Running our command')
```

Worker API

`Hub` - The workers async event loop

supported transports:
amqp, redis

New in version 3.0.

The worker uses asynchronous I/O when the amqp or redis broker transports are used. The eventual goal is for all transports to use the event-loop, but that will take some time so other transports still use a threading-based solution.

`hub.`**`add`**(*fd, callback, flags*)

`hub.`**`add_reader`**(*fd, callback, *args*)

Add callback to be called when `fd` is readable.

The callback will stay registered until explicitly removed using **`hub.remove(fd)`**, or the file descriptor is automatically discarded because it's no longer valid.

Note that only one callback can be registered for any given file descriptor at a time, so calling `add` a second time will remove any callback that was previously registered for that file descriptor.

A file descriptor is any file-like object that supports the `fileno` method, or it can be the file descriptor number (int).

`hub.`**`add_writer`**(*fd*, *callback*, **args*)

Add callback to be called when `fd` is writable. See also notes for **`hub.add_reader()`** above.

`hub.`**`remove`**(*fd*)

Remove all callbacks for file descriptor `fd` from the loop.

Timer - Scheduling events

`timer.call_after(secs, callback, args=(), kwargs=(), priority=0)`

`timer.call_repeatedly(secs, callback, args=(), kwargs=(), priority=0)`

`timer.call_at(eta, callback, args=(), kwargs=(), priority=0)`