Previous topic

Concurrency with Eventlet

Next topic

Testing with Celery

This Page

Show Source

Quick search

Go

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

Signals

Signals allows decoupled applications to receive notifications when certain actions occur elsewhere in the application.

Celery ships with many signals that your application can hook into to augment behavior of certain actions.

Basics

Several kinds of events trigger signals, you can connect to these signals to perform actions as they trigger.

Example connecting to the **`after_task_publish`** signal:

```
from celery.signals import after_task_publish

@after_task_publish.connect
def task_sent_handler(sender=None, headers=None, body=None, **kwargs):
    # information about task are located in headers for task messages
    # using the task protocol version 2.
    info = headers if 'task' in headers else body
    print('after_task_publish for task id {info[id]}'.format(
        info=info,
    ))
```

Some signals also have a sender you can filter by. For example the **`after_task_publish`** signal uses the task name as a sender, so by providing the `sender` argument to **`connect`** you can connect your handler to be called every time a task with name *"proj.tasks.add"* is published:

```
@after_task_publish.connect(sender='proj.tasks.add')
def task_sent_handler(sender=None, headers=None, body=None, **kwargs):
    # information about task are located in headers for task messages
    # using the task protocol version 2.
    info = headers if 'task' in headers else body
    print('after_task_publish for task id {info[id]}'.format(
        info=info,
    ))
```

Signals use the same implementation as **`django.core.dispatch`**. As a result other keyword parameters (e.g., signal) are passed to all signal handlers by default.

The best practice for signal handlers is to accept arbitrary keyword arguments (i.e., `**kwargs`). That way new Celery versions can add additional arguments without breaking user code.

Signals

Task Signals

`before_task_publish`

New in version 3.1.

Dispatched before a task is published. Note that this is executed in the process sending the task.

Sender is the name of the task being sent.

Provides arguments:

- `body`

Task message body.

This is a mapping containing the task message fields, see Version 2 and Version 1 for a reference of possible fields that can be defined.

- `exchange`

  Name of the exchange to send to or a **`Exchange`** object.

- `routing_key`

  Routing key to use when sending the message.

- `headers`

  Application headers mapping (can be modified).

- `properties`

  Message properties (can be modified)

- `declare`

  List of entities (**`Exchange`**, **`Queue`**, or **`binding`** to declare before publishing the message. Can be modified.

- `retry_policy`

  Mapping of retry options. Can be any argument to **`kombu.Connection.ensure()`** and can be modified.

`after_task_publish`

Dispatched when a task has been sent to the broker. Note that this is executed in the process that sent the task.

Sender is the name of the task being sent.

Provides arguments:

- `headers`

  The task message headers, see Version 2 and Version 1 for a reference of possible fields that can be defined.

- `body`

  The task message body, see Version 2 and Version 1 for a reference of possible fields that can be defined.

- `exchange`

  Name of the exchange or **`Exchange`** object used.

- `routing_key`

  Routing key used.

`task_prerun`

Dispatched before a task is executed.

Sender is the task object being executed.

Provides arguments:

- `task_id`

  Id of the task to be executed.

- `task`

  The task being executed.

- `args`

  The tasks positional arguments.

- `kwargs`

  The tasks keyword arguments.

`task_postrun`

Dispatched after a task has been executed.

Sender is the task object executed.

Provides arguments:

- `task_id`

  Id of the task to be executed.

- `task`

  The task being executed.

- `args`

  The tasks positional arguments.

- `kwargs`

  The tasks keyword arguments.

- `retval`

  The return value of the task.

- `state`

  Name of the resulting state.

`task_retry`

Dispatched when a task will be retried.

Sender is the task object.

Provides arguments:

- `request`

  The current task request.

- `reason`

  Reason for retry (usually an exception instance, but can always be coerced to

str).

- `einfo`

  Detailed exception information, including traceback (a **billiard.einfo.ExceptionInfo** object).

task_success

Dispatched when a task succeeds.

Sender is the task object executed.

Provides arguments

- `result`
  Return value of the task.

task_failure

Dispatched when a task fails.

Sender is the task object executed.

Provides arguments:

- `task_id`

  Id of the task.

- `exception`

  Exception instance raised.

- `args`

  Positional arguments the task was called with.

- `kwargs`

  Keyword arguments the task was called with.

- `traceback`

  Stack trace object.

- `einfo`

  The **billiard.einfo.ExceptionInfo** instance.

task_revoked

Dispatched when a task is revoked/terminated by the worker.

Sender is the task object revoked/terminated.

Provides arguments:

- `request`

  This is a **Request** instance, and not `task.request`. When using the prefork pool this signal is dispatched in the parent process, so `task.request` isn't available and shouldn't be used. Use this object instead, as they share many of the same fields.

- `terminated`

  Set to **`True`** if the task was terminated.

- `signum`

  Signal number used to terminate the task. If this is **`None`** and terminated is **`True`** then **`TERM`** should be assumed.

- `expired`

  Set to **`True`** if the task expired.

`task_unknown`

Dispatched when a worker receives a message for a task that's not registered.

Sender is the worker **`Consumer`**.

Provides arguments:

- `name`

  Name of task not found in registry.

- `id`

  The task id found in the message.

- `message`

  Raw message object.

- `exc`

  The error that occurred.

`task_rejected`

Dispatched when a worker receives an unknown type of message to one of its task queues.

Sender is the worker **`Consumer`**.

Provides arguments:

- `message`

  Raw message object.

- `exc`

  The error that occurred (if any).

App Signals

`import_modules`

This signal is sent when a program (worker, beat, shell) etc, asks for modules in the **`include`** and **`imports`** settings to be imported.

Sender is the app instance.

Worker Signals

celeryd_after_setup

This signal is sent after the worker instance is set up, but before it calls run. This means that any queues from the **celery worker -Q** option is enabled, logging has been set up and so on.

It can be used to add custom queues that should always be consumed from, disregarding the **celery worker -Q** option. Here's an example that sets up a direct queue for each worker, these queues can then be used to route a task to any specific worker:

```
from celery.signals import celeryd_after_setup

@celeryd_after_setup.connect
def setup_direct_queue(sender, instance, **kwargs):
    queue_name = '{0}.dq'.format(sender)  # sender is the nodename of th
    instance.app.amqp.queues.select_add(queue_name)
```

Provides arguments:

- `sender`

  Node name of the worker.

- `instance`

  This is the **celery.apps.worker.Worker** instance to be initialized. Note that only the **app** and **hostname** (nodename) attributes have been set so far, and the rest of `__init__` hasn't been executed.

- `conf`

  The configuration of the current app.

celeryd_init

This is the first signal sent when **celery worker** starts up. The `sender` is the host name of the worker, so this signal can be used to setup worker specific configuration:

```
from celery.signals import celeryd_init

@celeryd_init.connect(sender='worker12@example.com')
def configure_worker12(conf=None, **kwargs):
    conf.task_default_rate_limit = '10/m'
```

or to set up configuration for multiple workers you can omit specifying a sender when you connect:

```
from celery.signals import celeryd_init

@celeryd_init.connect
def configure_workers(sender=None, conf=None, **kwargs):
    if sender in ('worker1@example.com', 'worker2@example.com'):
        conf.task_default_rate_limit = '10/m'
    if sender == 'worker3@example.com':
        conf.worker_prefetch_multiplier = 0
```

Provides arguments:

- `sender`

Nodename of the worker.

- `instance`

  This is the **`celery.apps.worker.Worker`** instance to be initialized. Note that only the **`app`** and **`hostname`** (nodename) attributes have been set so far, and the rest of `__init__` hasn't been executed.

- `conf`

  The configuration of the current app.

- `options`

  Options passed to the worker from command-line arguments (including defaults).

worker_init

Dispatched before the worker is started.

worker_ready

Dispatched when the worker is ready to accept work.

heartbeat_sent

Dispatched when Celery sends a worker heartbeat.

Sender is the **`celery.worker.heartbeat.Heart`** instance.

worker_shutting_down

Dispatched when the worker begins the shutdown process.

Provides arguments:

- `sig`

  The POSIX signal that was received.

- `how`

  The shutdown method, warm or cold.

- `exitcode`

  The exitcode that will be used when the main process exits.

worker_process_init

Dispatched in all pool child processes when they start.

Note that handlers attached to this signal mustn't be blocking for more than 4 seconds, or the process will be killed assuming it failed to start.

worker_process_shutdown

Dispatched in all pool child processes just before they exit.

Note: There's no guarantee that this signal will be dispatched, similarly to **`finally`** blocks it's impossible to guarantee that handlers will be called at shutdown, and if called

it may be interrupted during.

Provides arguments:

- `pid`

  The pid of the child process that's about to shutdown.

- `exitcode`

  The exitcode that'll be used when the child process exits.

`worker_shutdown`

Dispatched when the worker is about to shut down.

Beat Signals

`beat_init`

Dispatched when **celery beat** starts (either standalone or embedded).

Sender is the `celery.beat.Service` instance.

`beat_embedded_init`

Dispatched in addition to the `beat_init` signal when **celery beat** is started as an embedded process.

Sender is the `celery.beat.Service` instance.

Eventlet Signals

`eventlet_pool_started`

Sent when the eventlet pool has been started.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

`eventlet_pool_preshutdown`

Sent when the worker shutdown, just before the eventlet pool is requested to wait for remaining workers.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

`eventlet_pool_postshutdown`

Sent when the pool has been joined and the worker is ready to shutdown.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

`eventlet_pool_apply`

Sent whenever a task is applied to the pool.

Sender is the `celery.concurrency.eventlet.TaskPool` instance.

Provides arguments:

- `target`

  The target function.

- `args`

  Positional arguments.

- `kwargs`

  Keyword arguments.

Logging Signals

`setup_logging`

Celery won't configure the loggers if this signal is connected, so you can use this to completely override the logging configuration with your own.

If you'd like to augment the logging configuration setup by Celery then you can use the **`after_setup_logger`** and **`after_setup_task_logger`** signals.

Provides arguments:

- `loglevel`

  The level of the logging object.

- `logfile`

  The name of the logfile.

- `format`

  The log format string.

- `colorize`

  Specify if log messages are colored or not.

`after_setup_logger`

Sent after the setup of every global logger (not task loggers). Used to augment logging configuration.

Provides arguments:

- `logger`

  The logger object.

- `loglevel`

  The level of the logging object.

- `logfile`

  The name of the logfile.

- `format`

  The log format string.

- `colorize`

    Specify if log messages are colored or not.

`after_setup_task_logger`

Sent after the setup of every single task logger. Used to augment logging configuration.

Provides arguments:

- `logger`

    The logger object.

- `loglevel`

    The level of the logging object.

- `logfile`

    The name of the logfile.

- `format`

    The log format string.

- `colorize`

    Specify if log messages are colored or not.

Command signals

`user_preload_options`

This signal is sent after any of the Celery command line programs are finished parsing the user preload options.

It can be used to add additional command-line arguments to the **celery** umbrella command:

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

app = Celery()
app.user_options['preload'].add(Option(
    '--monitoring', action='store_true',
    help='Enable our external monitoring utility, blahblah',
))

@signals.user_preload_options.connect
def handle_preload_options(options, **kwargs):
    if options['monitoring']:
        enable_monitoring()
```

Sender is the **`Command`** instance, and the value depends on the program that was called (e.g., for the umbrella command it'll be a **`CeleryCommand`**) object).

Provides arguments:

- `app`

    The app instance.

- `options`

  Mapping of the parsed user preload options (with default values).

Deprecated Signals

`task_sent`

This signal is deprecated, please use **`after_task_publish`** instead.