Spinach Documentation

Release 0.0.11

Nicolas Le Manchet

Sep 21, 2019

Contents

Release v0.0.11. (*Installation*)

Spinach is a Redis task queue for Python 3 heavily inspired by Celery and RQ.

Distinctive features:

- At-least-once or at-most-once delivery per task
- Periodic tasks without an additional process
- Scheduling of tasks in batch
- Embeddable workers for easier testing
- Integrations with *Flask, Django, Logging, Sentry and Datadog*
- Python 3, threaded, explicit... see *design choices* for more details

Installation:

```
pip install spinach
```

Quickstart

```
from spinach import Engine, MemoryBroker

spin = Engine(MemoryBroker())


@spin.task(name='compute')
def compute(a, b):
    print('Computed {} + {} = {}'.format(a, b, a + b))


# Schedule a job to be executed ASAP
spin.schedule(compute, 5, 3)

print('Starting workers, ^C to quit')
spin.start_workers()
```

The `Engine` is the central part of Spinach, it allows to define tasks, schedule jobs to execute in the background and start background workers. *More details*.

The Broker is the backend that background workers use to retrieve jobs to execute. Spinach provides two brokers: MemoryBroker for development and RedisBroker for production.

The `Engine.task()` decorator is used to register tasks. It requires at least a *name* to identify the task, but other options can be given to customize how the task behaves. *More details*.

Background jobs can then be scheduled by using either the task name or the task function:

```
spin.schedule('compute', 5, 3)  # identify a task by its name
spin.schedule(compute, 5, 3)    # identify a task by its function
```

Getting started with spinach:

CHAPTER 1

Installation

1.1 Prerequisites

Spinach is written in Python 3, prior to use it you must make sure you have a Python 3.5+ interpreter on your system.

1.2 Pip

If you are familiar with the Python ecosystem, you won't be surprised that Spinach can be installed with:

```
$ pip install spinach
```

That's it, you can call it a day!

1.3 From Source

Spinach is developed on GitHub, you can find the code at NicolasLM/spinach.

You can clone the public repository:

```
$ git clone https://github.com/NicolasLM/spinach.git
```

Once you have the sources, simply install it with:

```
$ cd spinach
$ pip install -e .
```

CHAPTER 2

Tasks

A tasks is a unit of code, usually a function, to be executed in the background on remote workers.

To define a task:

```
from spinach import Tasks

tasks = Tasks()

@tasks.task(name='add')
def add(a, b):
    print('Computed {} + {} = {}'.format(a, b, a + b))
```

Note: The *args* and *kwargs* of a task must be JSON serializable.

2.1 Retries

Spinach knows two kinds of tasks: the ones that can be retried safely (idempotent tasks) and the ones that cannot be retried safely (non-idempotent tasks). Since Spinach cannot guess if a task code is safe to be retried multiple times, it must be annotated when the task is created.

2.1.1 Non-Retriable Tasks

Spinach assumes that by default tasks are not safe to be retried (tasks are assumed to have side effects).

These tasks are defined with *max_retries=0* (the default):

```
@tasks.task(name='foo')
def foo(a, b):
    pass
```

- use at-most-once delivery, the job may never even start
- jobs are not automatically retried in case of errors

2.1.2 Retriable Tasks

Idempotent tasks can be executed multiple times without changing the result beyond the initial application. It is a nice property to have and most tasks should try to be idempotent to gracefully recover from errors.

Retriable tasks are defined with a positive *max_retries* value:

```
@tasks.task(name='foo', max_retries=10)
def foo(a, b):
    pass
```

- use at-least-once delivery, the job may be executed more than once
- jobs are automatically retried, up to *max_retries* times, in case of errors

2.1.3 Retrying

When a retriable task is being executed it will be retried when it encounters an unexpected exception:

```
@tasks.task(name='foo', max_retries=10)
def foo(a, b):
    l = [0, 1, 2]
    print(l[100])  # Raises IndexError
```

To allow the system to recover gracefully, a default backoff strategy is applied.

spinach.utils.**exponential_backoff**(*attempt: int*, *cap: int = 1200*) → datetime.timedelta

Calculate a delay to retry using an exponential backoff algorithm.

It is an exponential backoff with random jitter to prevent failures from being retried at the same time. It is a good fit for most applications.

Parameters

- **attempt** – the number of attempts made
- **cap** – maximum delay, defaults to 20 minutes

To be more explicit, a task can also raise a `RetryException` which allows to precisely control when it should be retried:

```
from spinach import RetryException

@tasks.task(name='foo', max_retries=10)
def foo(a, b):
    if status_code == 429:
        raise RetryException(
            'Should retry in 10 minutes',
            at=datetime.now(tz=timezone.utc) + timedelta(minutes=10)
        )
```

class spinach.task.**RetryException**(*message*, *at: Optional[datetime.datetime] = None*)

Exception raised in a task to indicate that the job should be retried.

Even if this exception is raised, the *max_retries* defined in the task still applies.

Parameters **at** – Optional date at which the job should be retried. If it is not given the job will be retried after a randomized exponential backoff. It is advised to pass a timezone aware datetime to lift any ambiguity. However if a timezone naive datetime if given, it will be assumed to contain UTC time.

A task can also raise a `AbortException` for short-circuit behavior:

```
.. autoclass:: spinach.task.AbortException
```

2.2 Periodic tasks

Tasks marked as periodic get automatically scheduled. To run a task every 5 seconds:

```
from datetime import timedelta

from spinach import Engine, MemoryBroker

spin = Engine(MemoryBroker())
every_5_sec = timedelta(seconds=5)


@spin.task(name='make_coffee', periodicity=every_5_sec)
def make_coffee():
    print("Making coffee...")


print('Starting workers, ^C to quit')
spin.start_workers()
```

Periodic tasks get scheduled by the workers themselves, there is no need to run an additional process only for that. Of course having multiple workers on multiple machine is fine and will not result in duplicated tasks.

Periodic tasks run at most every *period*. If the system scheduling periodic tasks gets delayed, nothing compensates for the time lost. This has the added benefit of periodic tasks not being scheduled if all the workers are down for a prolonged amount of time. When they get back online, workers won't have a storm of periodic tasks to execute.

2.3 Tasks Registry

Before being attached to a Spinach `Engine`, tasks are created inside a `Tasks` registry.

Attaching tasks to a `Tasks` registry instead of directly to the `Engine` allows to compose large applications in smaller units independent from each other, the same way a Django project is composed of many small Django apps.

This may seem cumbersome for trivial applications, like the examples in this documentation or some single-module projects, so those can create tasks directly on the `Engine` using:

```
spin = Engine(MemoryBroker())

@spin.task(name='fast')
def fast():
    time.sleep(1)
```

Note: Creating tasks directly in the `Engine` is a bit like creating a Flask app globally instead of using an *app factory*: it works until a change introduces a circular import. Its usage should really be limited to tiny projects.

class spinach.task.**Tasks**(*queue: Optional[str] = None*, *max_retries: Optional[numbers.Number] = None*, *periodicity: Optional[datetime.timedelta] = None*)

Registry for tasks to be used by Spinach.

Parameters

- **queue** – default queue for tasks
- **max_retries** – default retry policy for tasks
- **periodicity** – for periodic tasks, delay between executions as a timedelta

add(*func: Callable*, *name: Optional[str] = None*, *queue: Optional[str] = None*, *max_retries: Optional[numbers.Number] = None*, *periodicity: Optional[datetime.timedelta] = None*)

Register a task function.

Parameters

- **func** – a callable to be executed
- **name** – name of the task, used later to schedule jobs
- **queue** – queue of the task, the default is used if not provided
- **max_retries** – maximum number of retries, the default is used if not provided
- **periodicity** – for periodic tasks, delay between executions as a timedelta

```
>>> tasks = Tasks()
>>> tasks.add(lambda x: x, name='do_nothing')
```

schedule(*task: Union[str, Callable, spinach.task.Task]*, **args*, ***kwargs*)

Schedule a job to be executed as soon as possible.

Parameters

- **task** – the task or its name to execute in the background
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

This method can only be used once tasks have been attached to a Spinach `Engine`.

schedule_at(*task: Union[str, Callable, spinach.task.Task]*, *at: datetime.datetime*, **args*, ***kwargs*)

Schedule a job to be executed in the future.

Parameters

- **task** – the task or its name to execute in the background
- **at** – Date at which the job should start. It is advised to pass a timezone aware datetime to lift any ambiguity. However if a timezone naive datetime if given, it will be assumed to contain UTC time.
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

This method can only be used once tasks have been attached to a Spinach `Engine`.

schedule_batch(*batch: spinach.task.Batch*)
Schedule many jobs at once.

Scheduling jobs in batches allows to enqueue them fast by avoiding round-trips to the broker.

Parameters **batch** – *Batch* instance containing jobs to schedule

task(*func: Optional[Callable] = None*, *name: Optional[str] = None*, *queue: Optional[str] = None*, *max_retries: Optional[numbers.Number] = None*, *periodicity: Optional[datetime.timedelta] = None*)
Decorator to register a task function.

Parameters

- **name** – name of the task, used later to schedule jobs
- **queue** – queue of the task, the default is used if not provided
- **max_retries** – maximum number of retries, the default is used if not provided
- **periodicity** – for periodic tasks, delay between executions as a timedelta

```
>>> tasks = Tasks()
>>> @tasks.task(name='foo')
>>> def foo():
...     pass
```

2.4 Batch

class spinach.task.**Batch**
Container allowing to schedule many jobs at once.

Batching the scheduling of jobs allows to avoid doing many round-trips to the broker, reducing the overhead and the chance of errors associated with doing network calls.

In this example 100 jobs are sent to Redis in one call:

```
>>> batch = Batch()
>>> for i in range(100):
...     batch.schedule('compute', i)
...
>>> spin.schedule_batch(batch)
```

Once the *Batch* is passed to the `Engine` it should be disposed off and not be reused.

schedule(*task: Union[str, Callable, spinach.task.Task], *args, **kwargs*)
Add a job to be executed ASAP to the batch.

Parameters

- **task** – the task or its name to execute in the background
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

schedule_at(*task: Union[str, Callable, spinach.task.Task], at: datetime.datetime, *args, **kwargs*)
Add a job to be executed in the future to the batch.

Parameters

- **task** – the task or its name to execute in the background

- **at** – Date at which the job should start. It is advised to pass a timezone aware datetime to lift any ambiguity. However if a timezone naive datetime if given, it will be assumed to contain UTC time.
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

CHAPTER 3

Jobs

A `Job` represents a specific execution of a task. To make an analogy with Python, a `Task` gets instantiated into many `Job`, like a *class* that gets instantiated into many *objects*.

3.1 Job

class spinach.job.**Job**

Represent the execution of a `Task` by background workers.

The *Job* class should not be instantiated by the user, instead jobs are automatically created when they are scheduled.

Variables

- **id** – UUID of the job
- **status** – *JobStatus*
- **task_name** – string name of the task
- **queue** – string name of the queue
- **at** – timezone aware *datetime* representing the date at which the job should start
- **max_retries** – int representing how many times a failing job should be retried
- **retries** – int representing how many times the job was already executed
- **task_args** – optional tuple containing args passed to the task
- **task_kwargs** – optional dict containing kwargs passed to the task

3.2 Job Status

class spinach.job.**JobStatus**

Possible status of a *Job*.

Life-cycle:

- Newly created jobs first get the status *NOT_SET*
- Future jobs are then set to *WAITING* until they are ready to be *QUEUED*
- Jobs starting immediately get the status *QUEUED* directly when they are received by the broker
- **Jobs are set to *RUNNING* when a worker start their execution**
  - if the job terminates without error it is set to *SUCCEEDED*
  - if the job terminates with an error and can be retried it is set to *WAITING* until it is ready to be queued again
  - if the job terminates with an error and cannot be retried it is set to *FAILED* for ever

See *Signals* to be notified on some of these status transitions.

FAILED = 5
Job failed and will not be retried

NOT_SET = 0
Job created but not scheduled yet

QUEUED = 2
Job is in a queue, ready to be picked by a worker

RUNNING = 3
Job is being executed

SUCCEEDED = 4
Job is finished, execution was successful

WAITING = 1
Job is scheduled to start in the future

CHAPTER 4

Engine

The Spinach `Engine` is what connects tasks, jobs, brokers and workers together.

It is possible, but unusual, to have multiple Engines running in the same Python interpreter.

class spinach.engine.**Engine**(*broker: spinach.brokers.base.Broker*, *namespace: str = 'spinach'*)

Spinach Engine coordinating a broker with workers.

Parameters

- **broker** – instance of a `Broker`
- **namespace** – name of the namespace used by the Engine. When different Engines use the same Redis server, they must use different namespaces to isolate themselves.

attach_tasks(*tasks: spinach.task.Tasks*)

Attach a set of tasks.

A task cannot be scheduled or executed before it is attached to an Engine.

```
>>> tasks = Tasks()
>>> spin.attach_tasks(tasks)
```

namespace

Namespace the Engine uses.

schedule(*task: Union[str, Callable, spinach.task.Task], *args, **kwargs*)

Schedule a job to be executed as soon as possible.

Parameters

- **task** – the task or its name to execute in the background
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

schedule_at(*task: Union[str, Callable, spinach.task.Task], at: datetime.datetime, *args, **kwargs*)

Schedule a job to be executed in the future.

Parameters

- **task** – the task or its name to execute in the background
- **at** – date at which the job should start. It is advised to pass a timezone aware datetime to lift any ambiguity. However if a timezone naive datetime if given, it will be assumed to contain UTC time.
- **args** – args to be passed to the task function
- **kwargs** – kwargs to be passed to the task function

schedule_batch(*batch: spinach.task.Batch*)
Schedule many jobs at once.

Scheduling jobs in batches allows to enqueue them fast by avoiding round-trips to the broker.

Parameters **batch** – `Batch` instance containing jobs to schedule

start_workers(*number: int = 5*, *queue='spinach'*, *block=True*, *stop_when_queue_empty=False*)
Start the worker threads.

Parameters

- **number** – number of worker threads to launch
- **queue** – name of the queue to consume, see *Queues*
- **block** – whether to block the calling thread until a signal arrives and workers get terminated
- **stop_when_queue_empty** – automatically stop the workers when the queue is empty. Useful mostly for one-off scripts and testing.

stop_workers(*_join_arbiter=True*)
Stop the workers and wait for them to terminate.

4.1 Namespace

Namespaces allow to identify and isolate multiple Spinach engines running on the same Python interpreter and/or sharing the same Redis server.

Having multiple engines on the same interpreter is rare but can happen when using the Flask integration with an app factory. In this case using different namespaces is important to avoid signals sent from one engine to be received by another engine.

When multiple Spinach Engines use the same Redis server, for example when production and staging share the same database, different namespaces must be used to make sure they do not step on each other's feet.

The production application would contain:

```
spin = Engine(RedisBroker(), namespace='prod')
```

While the staging application would contain:

```
spin = Engine(RedisBroker(), namespace='stg')
```

Note: Using different Redis database numbers (0, 1, 2...) for different environments is not enough as Redis pubsubs are shared among databases. Namespaces solve this problem.

CHAPTER 5

Queues

Queues are an optional feature that allows directing a set of tasks to specific workers.

Queues are useful when different tasks have different usage patterns, for instance one task being fast and high priority while another task is slow and low-priority. To prevent the slow task from blocking the execution of the fast one, each task can be attached to its own queue:

```
import time
import logging

from spinach import Engine, MemoryBroker


logging.basicConfig(
    format='%(asctime)s - %(threadName)s %(levelname)s: %(message)s',
    level=logging.DEBUG
)
spin = Engine(MemoryBroker())


@spin.task(name='fast', queue='high-priority')
def fast():
    time.sleep(1)


@spin.task(name='slow', queue='low-priority')
def slow():
    time.sleep(10)


spin.schedule(slow)
spin.schedule(fast)

spin.start_workers(number=1, queue='high-priority', stop_when_queue_empty=True)
```

The task decorator accepts an optional queue name that binds the task to a specific queue. Likewise, passing a queue

name to *start_workers* restricts workers to executing only tasks of this particular queue.

Note: By default all tasks and all workers use the `spinach` queue

Note: Namespaces and queues are different concepts. While queues share the same Spinach `Engine`, namespaces make two Spinach Engines invisible to each other while still using the same broker.

CHAPTER 6

Integrations

Integration with third-party libraries and frameworks.

6.1 Logging

Spinach uses the standard Python logging package. Its logger prefix is `spinach`. Spinach does nothing else besides creating its loggers and emitting log records. The user is responsible for configuring logging before starting workers.

For simple applications it is enough to use:

```
import logging

logging.basicConfig(
    format='%(asctime)s - %(threadName)s %(levelname)s: %(message)s',
    level=logging.DEBUG
)
```

More complex applications will probably use dictConfig.

6.2 Flask

The Flask integration follows the spirit of Flask very closely, it provides two ways of getting started: a single module approach for minial applications and an application factory approach for more scalable code.

The Spinach extension for Flask pushes an application context for the duration of the tasks, which means that it plays well with other extensions like Flask-SQLAlchemy and doesn't require extra precautions.

6.2.1 Single Module

```
from flask import Flask
from spinach.contrib.flask_spinach import Spinach

app = Flask(__name__)
spinach = Spinach(app)


@spinach.task(name='say_hello')
def say_hello():
    print('Hello from a task')


@app.route('/')
def home():
    spinach.schedule('say_hello')
    return 'Hello from HTTP'
```

6.2.2 Application Factory

This more complex layout includes an Application Factory `create_app` and an imaginary `auth` Blueprint containing routes and tasks.

`app.py`:

```
from flask import Flask
from spinach import RedisBroker
from spinach.contrib.flask_spinach import Spinach

spinach = Spinach()


def create_app():
    app = Flask(__name__)
    app.config['SPINACH_BROKER'] = RedisBroker()
    spinach.init_app(app)

    from . import auth
    app.register_blueprint(auth.blueprint)
    spinach.register_tasks(app, auth.tasks)

    return app
```

`auth.py`:

```
from flask import Blueprint, jsonify
from spinach import Tasks

from .app import spinach


blueprint = Blueprint('auth', __name__)
tasks = Tasks()
```

(continues on next page)

(continued from previous page)

```
@blueprint.route('/')
def create_user():
    spinach.schedule('send_welcome_email')
    return jsonify({'user_id': 42})


@tasks.task(name='send_welcome_email')
def send_welcome_email():
    print('Sending email...')
```

6.2.3 Running workers

Workers can be launched from the Flask CLI:

```
$ FLASK_APP=examples.flaskapp flask spinach
```

The working queue and the number of threads can be changed with:

```
$ FLASK_APP=examples.flaskapp flask spinach --queue high-priority --threads 20
```

Note: When in development mode, Flask uses its reloader to automatically restart the process when the code changes. When having periodic tasks defined, using the MemoryBroker and Flask reloader users may see their periodic tasks scheduled each time the code changes. If this is a problem, users are encouraged to switch to the RedisBroker for development.

6.2.4 Configuration

- `SPINACH_BROKER`, default `spinach.RedisBroker()`
- `SPINACH_NAMESPACE`, defaults to the Flask app name

6.3 Django

A Django application is available for integrating Spinach into Django projects.

To get started, add the application `spinach.contrib.spinachd` to `settings.py`:

```
INSTALLED_APPS = (
    ...
    'spinach.contrib.spinachd',
)
```

On startup, Spinach will look for a `tasks.py` module in all installed applications. For instance `polls/tasks.py`:

```
from spinach import Tasks

from .models import Question

tasks = Tasks()
```

(continues on next page)

(continued from previous page)

```
@tasks.task(name='polls:close_poll')
def close_poll(question_id: int):
    Question.objects.get(pk=question_id).delete()
```

Tasks can be easily scheduled from views:

```
from .models import Question
from .tasks import tasks

def close_poll_view(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    tasks.schedule('polls:close_poll', question.id)
```

Users of the Django Datadog app get their jobs reported to Datadog APM automatically in task workers.

6.3.1 Running workers

Workers can be launched from `manage.py`:

```
$ python manage.py spinach
```

The working queue and the number of threads can be changed with:

```
$ python manage.py spinach --queue high-priority --threads 20
```

6.3.2 Sending emails in the background

The Spinach app provides an `EMAIL_BACKEND` allowing to send emails as background tasks. To use it simply add it to `settings.py`:

```
EMAIL_BACKEND = 'spinach.contrib.spinachd.mail.BackgroundEmailBackend'
SPINACH_ACTUAL_EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
```

Emails can then be sent using regular Django functions:

```
from django.core.mail import send_mail

send_mail('Subject', 'Content', 'sender@example.com', ['receiver@example.com'])
```

6.3.3 Periodically clearing expired sessions

Projects using `django.contrib.sessions` must remove expired session from the database from time to time. Django comes with a management command to do that manually, but this can be automated.

Spinach provides a periodic task, disabled by default, to do that. To enable it give it a periodicity in `settings.py`. For instance to clear sessions once per week:

```
from datetime import timedelta

SPINACH_CLEAR_SESSIONS_PERIODICITY = timedelta(weeks=1)
```

6.3.4 Configuration

- `SPINACH_BROKER`, default `spinach.RedisBroker()`
- `SPINACH_NAMESPACE`, default `spinach`
- `SPINACH_ACTUAL_EMAIL_BACKEND`, default `django.core.mail.backends.smtp.EmailBackend`
- `SPINACH_CLEAR_SESSIONS_PERIODICITY`, default `None` (disabled)

6.4 Sentry

With the Sentry integration, failing jobs can be automatically reported to Sentry with full traceback, log breadcrumbs and job information.

The Sentry integration requires Sentry SDK:

```
pip install sentry_sdk
```

It then just needs to be registered before starting workers:

```
import sentry_sdk

from spinach.contrib.sentry_sdk_spinach import SpinachIntegration

sentry_sdk.init(
    dsn="https://sentry_dsn/42",
    integrations=[SpinachIntegration()]
)
```

Note: Users of the deprecated Raven client for Sentry can use the old Spinach integration below.

The old integration requires Raven:

```
pip install raven
```

It then just needs to be registered before starting workers:

```
from raven import Client
from spinach.contrib.sentry import register_sentry

raven_client = Client('https://sentry_dsn/42')
register_sentry(raven_client)

spin = Engine(MemoryBroker())
spin.start_workers()
```

6.5 Datadog

With the Datadog integration, all jobs are automatically reported to Datadog APM.

The integration requires ddtrace, the Datadog APM client for Python:

```
pip install ddtrace
```

The integration just needs to be registered before starting workers:

```
from spinach.contrib.datadog import register_datadog

register_datadog()

spin = Engine(MemoryBroker())
spin.start_workers()
```

This only installs the integration with Spinach, other libraries still need to be patched by ddtrace. It is recommended to run your application patched as explained in the ddtrace documentation.

spinach.contrib.datadog.**register_datadog**(*tracer=None*, *namespace: Optional[str] = None*, *service: str = 'spinach'*)

Register the Datadog integration.

Exceptions making jobs fail are sent to Sentry.

Parameters

- **tracer** – optionally use a custom ddtrace Tracer instead of the global one.
- **namespace** – optionally only register the Datadog integration for a particular Spinach `Engine`
- **service** – Datadog service associated with the trace, defaults to *spinach*

CHAPTER 7

Signals

Signals are events broadcasted when something happens in Spinach, like a job starting or a worker shutting down.

Subscribing to signals allows your code to react to internal events in a composable and reusable way.

7.1 Subscribing to signals

Subscribing to a signal is done via its `connect` decorator:

```
from spinach import signals

@signals.job_started.connect
def job_started(namespace, job, **kwargs):
    print('Job {} started'.format(job))
```

The first argument given to your function is always the namespace of your Spinach `Engine`, the following arguments depend on the signal itself.

7.1.1 Subscribing to signals of a specific Spinach Engine

As your application gets bigger you may end up running multiple Engines in the same interpreter. The `connect_via` decorator allows to subscribe to the signals sent by a specific Spinach `Engine`:

```
from spinach import Engine, MemoryBroker, signals

foo_spin = Engine(MemoryBroker(), namespace='foo')
bar_spin = Engine(MemoryBroker(), namespace='bar')

@signals.job_started.connect_via(foo_spin.namespace)
def job_started(namespace, job, **kwargs):
    print('Job {} started on Foo'.format(job))
```

In this example only signals sent by the *foo* `Engine` will be received.

7.2 Available signals

spinach.signals.**job_started = SafeNamedSignal "job_started"**
Sent by a worker when a job starts being executed.

Signal handlers receive:

- *namespace* Spinach namespace
- *job* `Job` being executed

spinach.signals.**job_finished = SafeNamedSignal "job_finished"**
Sent by a worker when a job finishes execution.

The signal is sent no matter the outcome, even if the job fails or gets rescheduled for retry.

Signal handlers receive:

- *namespace* Spinach namespace
- *job* `Job` being executed

spinach.signals.**job_schedule_retry = SafeNamedSignal "job_schedule_retry"**
Sent by a worker when a job gets rescheduled for retry.

Signal handlers receive:

- *namespace* Spinach namespace
- *job* `Job` being executed
- *err* exception that made the job retry

spinach.signals.**job_failed = SafeNamedSignal "job_failed"**
Sent by a worker when a job failed.

A failed job will not be retried.

Signal handlers receive:

- *namespace* Spinach namespace
- *job* `Job` being executed
- *err* exception that made the job fail

spinach.signals.**worker_started = SafeNamedSignal "worker_started"**
Sent by a worker when it starts.

Signal handlers receive:

- *namespace* Spinach namespace
- *worker_name* name of the worker starting

spinach.signals.**worker_terminated = SafeNamedSignal "worker_terminated"**
Sent by a worker when it shutdowns.

Signal handlers receive:

- *namespace* Spinach namespace
- *worker_name* name of the worker shutting down

7.3 Tips

7.3.1 Received objects

Objects received via signals should not be modified in handlers as it could break something in Spinach internals.

7.3.2 Exceptions

If your receiving function raises an exception while processing a signal, this exception will be logged in the `spinach.signals` logger.

7.3.3 Going further

Have a look at the blinker documentation for other ways using signals.

CHAPTER 8

Running in Production

Advices to read before deploying an application using Spinach to production.

8.1 Spinach

Since Spinach relies heavily on threads the user's code MUST be thread-safe. This is usually quite easy to achieve on a traditional web application because frameworks like Flask or Django make that obvious.

Tasks should not store state in the process between invocations. Instead all state must be stored in an external system, like a database or a cache. This advice also applies to *views* in a web application.

8.2 Redis

Most Spinach features are implemented as Lua scripts running inside Redis. Having a solid installation of Redis is the key to Spinach reliability.

To ensure that no tasks are lost or duplicated, Redis must be configured with persistence enabled. It is recommended to use AOF persistence (`appendonly yes`) instead of periodic RDB dumps. The default of fsync every second (`appendfsync everysec`) is a good trade-off between performance and security against sudden power failures.

Using Redis as a task queue is very different from using it as a cache. If an application uses Redis for both task queue and cache, it is recommended to have two separated Redis servers. One would be configured with persistence and without eviction while the other would have no persistence but would evict keys when running low on memory.

Finally standard security practices apply: Redis should not accept connections from the Internet and it should require a password even when connecting locally.

8.3 System

If the application is deployed on multiple servers it is important that their clocks be approximately synchronized. This is because Spinach uses the system time to know when a job should start. Running an *ntp* daemon is highly recommended.

Workers should be started by an init system that will restart them if they get killed or if the host reboots.

8.4 Production Checklist

Spinach:

- Tasks that are NOT safe to be retried have their *max_retries* set to *0*
- Tasks that are safe to be retried have their *max_retries* set to a positive number
- Retries happen after an exponential delay with randomized jitter (the default)
- Task *args* and *kwargs* are JSON serializable and small in size
- Jobs are sent in `Batch` to the broker when multiple jobs are to be scheduled at once
- The user's code is thread-safe
- Tasks do not store state in the process between invocations
- Logging is configured and exceptions are sent to Sentry, see *Integrations*
- Different queues are used if tasks have different usage pattens, see *Queues*
- Different namespaces are used if multiple Spinach applications share the same Redis server, see *Engine*

Redis:

- Redis uses AOF persistence
- Redis does not evict keys when running low on memory
- The Redis server used by Spinach is not also used as a cache
- Connections are secured by a long password
- Connections are encrypted if they go through the public Internet

System:

- Servers have their clock synchronized by ntp
- Workers get restarted by an init system if they get killed

CHAPTER 9

Design choices

I have used the Celery task queue for a long time and while it is a rock solid piece of software, there are some design decisions that just drive me crazy.

This page presents and explains the key design decisions behind Spinach. It can be summed up as: explicit is better than implicit. Spinach makes sure that it does not provide any convenient feature that can backfire in more complex usages.

9.1 Threaded workers

Spinach workers are threaded while other task queues like Celery or RQ rely on processes by default.

Threaded workers work best with IO bound tasks: tasks that make requests to other services, query a database or read files. If your task are CPU bound, meaning that you do heavy computations in Python, a process based worker will be more efficient.

Tasks in a typical web application are more often than not IO bound. The choice of threads as unit of concurrency is a sensible one.

Threads also have the advantage of being lighter than processes, a system can handle more threads than processes before resources get exhausted.

9.1.1 Thread safety

As Spinach workers are threads, care must be taken to make sure that the application is thread-safe. The good news is that your application is probably already thread-safe: web frameworks are often run threaded as well, so they take care of most of the heavy work for you.

You can read an article I wrote for an introduction to thread-safety.

9.1.2 Fork

Another reason why Spinach does not use processes for concurrency is because the `fork` system call used to create the workers is a very special one. It has Copy-On-Write semantics that are unfamiliar to many Python developers.

On the other hand thread-safety is a more understood problem in Python, the standard library providing most of the solutions to write thread-safe programs.

Not relying on `fork` also makes Spinach compatible with Windows.

9.2 Embeddable workers

As workers are just threads they are easily embeddable in any other Python process. This opens the door to two nice usages:

During automated tests a worker can be launched processing jobs exactly like a normal worker would do in production. What is more by using an in-memory broker there is no need for having a Redis server running during tests.

For small web projects, the task workers can be launched from the same process as the web application. As the application gets bigger the workers can be moved to a separate process very easily.

9.3 Logging

One issue I have with Celery is the way it handles logging: the framework tries to be too smart, resulting in great pain when the logging setup gets more complex.

That is why Spinach keeps it simple: as a well behaved library it uses the standard logging module and writes logs in its own loggers.

The choice of what to do with theses log records is up to the final user.

9.4 Jobs scheduled for the future

Spinach has full support for jobs that need to be executed in the future. These jobs go to a special queue until the are ready to be launched. At that time they are moved to a normal queue where they are picked by a worker.

Celery emulates this behavior by immediately sending the task to a worker and waiting there until the time has come to execute it. It means tasks cannot be scheduled much in advance without wasting resources in the worker.

9.5 Periodic jobs

One annoying thing with Celery is that you can launch as many distributed workers as you want but there must be one and only one Celery beat process running in the cluster at a time.

This approach does not work well with containerized applications that run in a cluster that often redeploys and move containers around.

All Spinach workers are part of the system that schedules periodic jobs, there is no need to have a pet in the cattle farm.

9.6 Only two brokers

Spinach lets the user pick between the in-memory broker for local development and the Redis broker for production. Both support exactly the same set of features.

Redis was chosen because it is an incredibly versatile database. With Lua scripting it becomes possible to develop entirely new patterns which are essential to create a useful and reliable task queue.

Other services like Google PubSub, Amazon SQS or AMQP are very opinionated and not as versatile as Redis, making them difficult to use within Spinach without cutting down on features.

9.7 Namespace

Multiple Spinach applications (production, staging...) can use the same Redis database without interfering with each other.

Likewise, a single interpreter can run multiple Spinach applications without them interfering with each other.

9.8 Minimize import side-effects

Spinach encourages users to write applications that have minimal side-effects when imported. There is no global state that gets created or modified when importing or using Spinach.

The user is free to use Spinach in a scoped fashion or declaring everything globally.

This makes it possible for a single interpreter to run multiple Spinach applications without them interfering with each other, which is particularly useful for running automated tests.

9.9 No worker entrypoint

Celery has this `celery worker` entrypoint that can be launched from the command line to load an application and spawn the workers.

The problem I often face is that I never know if a setting should be defined in my code as part of the app setup or as a flag of this command line.

Moreover command line flags and application settings often have slightly different names, making things more confusing.

Spinach thus makes it foolproof, you are responsible for configuring the Spinach app though your Python code. You can read settings from environment variables, from a file or anything else possible in Python.

It is then easy to use it to create your own entrypoint to launch the workers.

9.10 Schedule tasks in batch

A pattern that is used frequently with task queues is to periodically scan all entities and schedule an individual task for each entity that needs further work. For instance closing user accounts of member who haven't logged in in a year.

With Celery this results in having to do as many round-trips to the broker as there are tasks to schedule. There are some workarounds but they just move the problem elsewhere.

Spinach supports sending tasks to the broker in batch to avoid this overhead.

9.11 Written for the Cloud

Latency between workers and Redis can be high, for example when they are deployed in two separate regions. Spinach leverages Lua scripting in Redis to avoid unnecessary round-trips by batching calls as much as possible.

In a cloud environment network connections can get dropped and packets get lost. Spinach retries failed actions after applying an exponential backoff with randomized jitter to avoid the thundering herd problem when the network gets back to normal.

Workers are expected to be deployed in containers, probably managed by an orchestrator like Kubernetes or Nomad that often scale and shuffle containers around. Workers can join and leave the cluster at any time without impacting the ability to process jobs.

CHAPTER 10

FAQ

10.1 Should I use Spinach?

Spinach is a very young software, if your business depends heavily on background tasks you should probably go with Celery or RQ.

That being said, Spinach relies on proven technologies (Redis, Python queues, thread pools...) and is heavily tested.

10.2 Threads are not enough, can I use Processes?

Threading is the only concurrency primitive however it is possible to run many processes each containing one worker thread. This will open more connections to Redis, but Redis is known to support thousands of concurrent connections so this should not be a problem.

The best approach to achieve this is to rely on an init system like systemd, supervisord or docker. The init system will be responsible for spawning the correct number of processes and making sure they are properly restarted if they terminate prematurely.

Writing this init system yourself in Python using the multiprocessing module is possible but it must not import your actual application using Spinach. This is because mixing threads and forks in a single interpreter is a minefield. Anyway you are probably better off using a battle tested init system.

10.3 What is the licence?

Spinach is released under the `BSD license`.

Hacking guide:

CHAPTER 11

Contributing

This page contains the few guidelines and conventions used in the code base.

11.1 Pull requests

The development of Spinach happens on GitHub, the main repository is https://github.com/NicolasLM/spinach. To contribute to Spinach:

- Fork `NicolasLM/spinach`
- Clone your fork
- Create a feature branch `git checkout -b my_feature`
- Commit your changes
- Push your changes to your fork `git push origin my_feature`
- Create a GitHub pull request against `NicolasLM/spinach`'s master branch

Note: Avoid including multiple commits in your pull request, unless it adds value to a future reader. If you need to modify a commit, `git commit --amend` is your friend. Write a meaningful commit message, see How to write a commit message.

11.2 Python sources

The code base follows pep8 guidelines with lines wrapping at the 79th character. You can verify that the code follows the conventions with:

```
$ pep8 spinach tests
```

Running tests is an invaluable help when adding a new feature or when refactoring. Try to add the proper test cases in `tests/` together with your patch. The test suite can be run with pytest:

```
$ pytest tests
```

11.3 Compatibility

Spinach runs on all versions of Python starting from 3.5. Tests are run on Travis to ensure that.

11.4 Documentation sources

Documentation is located in the `doc` directory of the repository. It is written in reStructuredText and built with Sphinx.

If you modify the docs, make sure it builds without errors:

```
$ cd doc/
$ make html
```

The generated HTML pages should land in `doc/_build/html`.

CHAPTER 12

Internals

This page provides the basic information needed to start reading and modifying the source code of Spinach. It presents how it works inside and how the project is designed.

Todo: Document how Spinach works internally

Index