Flask-Executor Documentation

Release 0.10.0

Dave Chevell

Aug 19, 2022

Contents:

Flask-Executor is a Flask extension that makes it easy to work with `concurrent.futures` in your application.

CHAPTER 1

Installation

Flask-Executor is available on PyPI and can be installed with pip:

```
$ pip install flask-executor
```

CHAPTER 2

Setup

The Executor extension can either be initialized directly:

```
from flask import Flask
from flask_executor import Executor

app = Flask(__name__)
executor = Executor(app)
```

Or through the factory method:

```
executor = Executor()
executor.init_app(app)
```

CHAPTER 3

Configuration

To specify the type of executor to initialise, set `EXECUTOR_TYPE` inside your app configuration. Valid values are `'thread'` (default) to initialise a `ThreadPoolExecutor`, or `'process'` to initialise a `ProcessPoolExecutor`:

```
app.config['EXECUTOR_TYPE'] = 'thread'
```

To define the number of worker threads for a `ThreadPoolExecutor` or the number of worker processes for a `ProcessPoolExecutor`, set `EXECUTOR_MAX_WORKERS` in your app configuration. Valid values are any integer or `None` (default) to let `concurrent.futures` pick defaults for you:

```
app.config['EXECUTOR_MAX_WORKERS'] = 5
```

If multiple executors are needed, *`flask_executor.Executor`* can be initialised with a `name` parameter. Named executors will look for configuration variables prefixed with the specified `name` value, uppercased:

app.config[‘CUSTOM_EXECUTOR_TYPE’] = ‘thread’ app.config[‘CUSTOM_EXECUTOR_MAX_WORKERS’] = 5 executor = Executor(app, name=’custom’)

CHAPTER 4

Basic Usage

Flask-Executor supports the standard `concurrent.futures.Executor` methods, `submit()` and `map()`:

```
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

@app.route('/run_fib')
def run_fib():
    executor.submit(fib, 5)
    executor.map(fib, range(1, 6))
    return 'OK'
```

Submitting a task via `submit()` returns a `flask_executor.FutureProxy` object, a subclass of `concurrent.futures.Future` object from which you can retrieve your job status or result.

CHAPTER 5

Contexts

When calling `submit()` or `map()` Flask-Executor will wrap *ThreadPoolExecutor* callables with a copy of both the current application context and current request context. Code that must be run in these contexts or that depends on information or configuration stored in `flask.current_app`, `flask.request` or `flask.g` can be submitted to the executor without modification.

Note: due to limitations in Python's default object serialisation and a lack of shared memory space between subprocesses, contexts cannot be pushed to *ProcessPoolExecutor()* workers.

CHAPTER 6

Futures

`flask_executor.FutureProxy` objects look and behave like normal `concurrent.futures.Future` objects, but allow *flask_executor* to override certain methods and add additional behaviours. When submitting a callable to `add_done_callback()`, callables are wrapped with a copy of both the current application context and current request context.

You may want to preserve access to Futures returned from the executor, so that you can retrieve the results in a different part of your application. Flask-Executor allows Futures to be stored within the executor itself and provides methods for querying and returning them in different parts of your app:

```
@app.route('/start-task')
def start_task():
    executor.submit_stored('calc_power', pow, 323, 1235)
    return jsonify({'result':'success'})

@app.route('/get-result')
def get_result():
    if not executor.futures.done('calc_power'):
        return jsonify({'status': executor.futures._state('calc_power')})
    future = executor.futures.pop('calc_power')
    return jsonify({'status': done, 'result': future.result()})
```

CHAPTER 7

Decoration

Flask-Executor lets you decorate methods in the same style as distributed task queues when using 'thread' executor type like Celery:

```
@executor.job
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

@app.route('/decorate_fib')
def decorate_fib():
    fib.submit(5)
    fib.submit_stored('fibonacci', 5)
    fib.map(range(1, 6))
    return 'OK'
```

7.1 flask_executor

7.1.1 flask_executor package

Submodules

flask_executor.executor module

class flask_executor.executor.**Executor**(*app=None*, *name=''*)

Bases: flask_executor.helpers.InstanceProxy, concurrent.futures._base.Executor

An executor interface for concurrent.futures designed for working with Flask applications.

Parameters

- **app** – A Flask application instance.
- **name** – An optional name for the executor. This can be used to configure multiple executors. Named executors will look for environment variables prefixed with the name in uppercase, e.g. `CUSTOM_EXECUTOR_TYPE`.

add_default_done_callback(*fn*)
Registers callable to be attached to all newly created futures. When a callable is submitted to the executor, `concurrent.futures.Future.add_done_callback()` is called for every default callable that has been set."

Parameters **fn** – The callable to be added to the list of default done callbacks for new Futures.

init_app(*app*)
Initialise application. This will also intialise the configured executor type:

- `concurrent.futures.ThreadPoolExecutor`
- `concurrent.futures.ProcessPoolExecutor`

job(*fn*)
Decorator. Use this to transform functions into *ExecutorJob* instances that can submit themselves directly to the executor.

Example:

```
@executor.job
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

future = fib.submit(5)
results = fib.map(range(1, 6))
```

map(*fn*, **iterables*, ***kwargs*)
Submits the callable, fn, and an iterable of arguments to the executor and returns the results inside a generator.

See also `concurrent.futures.Executor.map()`.

Callables are wrapped a copy of the current application context and the current request context. Code that depends on information or configuration stored in `flask.current_app`, `flask.request` or `flask.g` can be run without modification.

Note: Because callables only have access to *copies* of the application or request contexts any changes made to these copies will not be reflected in the original view. Further, changes in the original app or request context that occur after the callable is submitted will not be available to the callable.

Parameters

- **fn** – The callable to be executed.
- ***iterables** – An iterable of arguments the callable will apply to.
- ****kwargs** – A dict of named parameters to pass to the underlying executor's `map()` method.

submit(*fn*, **args*, ***kwargs*)
Schedules the callable, fn, to be executed as fn(*args **kwargs) and returns a *FutureProxy* object, a `Future` subclass representing the execution of the callable.

See also `concurrent.futures.Executor.submit()`.

Callables are wrapped a copy of the current application context and the current request context. Code that depends on information or configuration stored in `flask.current_app`, `flask.request` or `flask.g` can be run without modification.

Note: Because callables only have access to *copies* of the application or request contexts any changes made to these copies will not be reflected in the original view. Further, changes in the original app or request context that occur after the callable is submitted will not be available to the callable.

Example:

```
future = executor.submit(pow, 323, 1235)
print(future.result())
```

Parameters

- **fn** – The callable to be executed.
- ***args** – A list of positional parameters used with the callable.
- ****kwargs** – A dict of named parameters used with the callable.

Return type flask_executor.FutureProxy

submit_stored(*future_key*, *fn*, **args*, ***kwargs*)

Submits the callable using *`Executor.submit()`* and stores the Future in the executor via a *`FutureCollection`* object available at `Executor.futures`. These futures can be retrieved anywhere inside your application and queried for status or popped from the collection. Due to memory concerns, the maximum length of the FutureCollection is limited, and the oldest Futures will be dropped when the limit is exceeded.

See *`flask_executor.futures.FutureCollection`* for more information on how to query futures in a collection.

Example:

```
@app.route('/start-task')
def start_task():
    executor.submit_stored('calc_power', pow, 323, 1235)
    return jsonify({'result':'success'})

@app.route('/get-result')
def get_result():
    if not executor.futures.done('calc_power'):
        future_status = executor.futures._state('calc_power')
        return jsonify({'status': future_status})
    future = executor.futures.pop('calc_power')
    return jsonify({'status': done, 'result': future.result()})
```

Parameters

- **future_key** – Stores the Future for the submitted task inside the executor's `futures` object with the specified key.
- **fn** – The callable to be executed.
- ***args** – A list of positional parameters used with the callable.
- ****kwargs** – A dict of named parameters used with the callable.

Return type concurrent.futures.Future

class flask_executor.executor.**ExecutorJob**(*executor*, *fn*)
Bases: object

Wraps a function with an executor so to allow the wrapped function to submit itself directly to the executor.

map(**iterables*, ***kwargs*)

submit(**args*, ***kwargs*)

submit_stored(*future_key*, **args*, ***kwargs*)

flask_executor.executor.**get_current_app_context**()

flask_executor.executor.**propagate_exceptions_callback**(*future*)

flask_executor.executor.**push_app_context**(*fn*)

flask_executor.futures module

class flask_executor.futures.**FutureCollection**(*max_length=50*)
Bases: object

A FutureCollection is an object to store and interact with concurrent.futures.Future objects. It provides access to all attributes and methods of a Future by proxying attribute calls to the stored Future object.

To access the methods of a Future from a FutureCollection instance, include a valid future_key value as the first argument of the method call. To access attributes, call them as though they were a method with future_key as the sole argument. If future_key does not exist, the call will always return None. If future_key does exist but the referenced Future does not contain the requested attribute an AttributeError will be raised.

To prevent memory exhaustion a FutureCollection instance can be bounded by number of items using the max_length parameter. As a best practice, Futures should be popped once they are ready for use, with the proxied attribute form used to determine whether a Future is ready to be used or discarded.

Parameters **max_length** – Maximum number of Futures to store. Oldest Futures are

discarded first.

add(*future_key*, *future*)
Add a new Future. If max_length limit was defined for the FutureCollection, old Futures may be dropped to respect this limit.

Parameters

- **future_key** – Key for the Future to be added.
- **future** – Future to be added.

pop(*future_key*)
Return a Future and remove it from the collection. Futures that are ready to be used should always be popped so they do not continue to consume memory.

Returns None if the key doesn't exist.

Parameters **future_key** – Key for the Future to be returned.

class flask_executor.futures.**FutureProxy**(*future*, *executor*)
Bases: flask_executor.helpers.InstanceProxy, concurrent.futures._base.Future

A FutureProxy is an instance proxy that wraps an instance of concurrent.futures.Future. Since an executor can't be made to return a subclassed Future object, this proxy class is used to override instance

behaviours whilst providing an agnostic method of accessing the original methods and attributes. :param future: An instance of `Future` that

the proxy will provide access to.

Parameters **executor** – An instance of *flask_executor.Executor* which will be used to provide access to Flask context features.

add_done_callback(*fn*)

Attaches a callable that will be called when the future finishes.

Args:

fn: A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

Module contents

class flask_executor.**Executor**(*app=None*, *name=''*)

Bases: `flask_executor.helpers.InstanceProxy`, `concurrent.futures._base.Executor`

An executor interface for `concurrent.futures` designed for working with Flask applications.

Parameters

- **app** – A Flask application instance.
- **name** – An optional name for the executor. This can be used to configure multiple executors. Named executors will look for environment variables prefixed with the name in uppercase, e.g. `CUSTOM_EXECUTOR_TYPE`.

add_default_done_callback(*fn*)

Registers callable to be attached to all newly created futures. When a callable is submitted to the executor, `concurrent.futures.Future.add_done_callback()` is called for every default callable that has been set."

Parameters **fn** – The callable to be added to the list of default done callbacks for new Futures.

init_app(*app*)

Initialise application. This will also intialise the configured executor type:

- `concurrent.futures.ThreadPoolExecutor`
- `concurrent.futures.ProcessPoolExecutor`

job(*fn*)

Decorator. Use this to transform functions into *ExecutorJob* instances that can submit themselves directly to the executor.

Example:

```
@executor.job
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

(continues on next page)

(continued from previous page)

```
future = fib.submit(5)
results = fib.map(range(1, 6))
```

map (*fn*, **iterables*, ***kwargs*)

Submits the callable, fn, and an iterable of arguments to the executor and returns the results inside a generator.

See also `concurrent.futures.Executor.map()`.

Callables are wrapped a copy of the current application context and the current request context. Code that depends on information or configuration stored in `flask.current_app`, `flask.request` or `flask.g` can be run without modification.

Note: Because callables only have access to *copies* of the application or request contexts any changes made to these copies will not be reflected in the original view. Further, changes in the original app or request context that occur after the callable is submitted will not be available to the callable.

Parameters

- **fn** – The callable to be executed.
- ***iterables** – An iterable of arguments the callable will apply to.
- ****kwargs** – A dict of named parameters to pass to the underlying executor's `map()` method.

submit (*fn*, **args*, ***kwargs*)

Schedules the callable, fn, to be executed as fn(*args **kwargs) and returns a *FutureProxy* object, a `Future` subclass representing the execution of the callable.

See also `concurrent.futures.Executor.submit()`.

Callables are wrapped a copy of the current application context and the current request context. Code that depends on information or configuration stored in `flask.current_app`, `flask.request` or `flask.g` can be run without modification.

Note: Because callables only have access to *copies* of the application or request contexts any changes made to these copies will not be reflected in the original view. Further, changes in the original app or request context that occur after the callable is submitted will not be available to the callable.

Example:

```
future = executor.submit(pow, 323, 1235)
print(future.result())
```

Parameters

- **fn** – The callable to be executed.
- ***args** – A list of positional parameters used with the callable.
- ****kwargs** – A dict of named parameters used with the callable.

Return type flask_executor.FutureProxy

submit_stored (*future_key*, *fn*, **args*, ***kwargs*)

Submits the callable using *Executor.submit()* and stores the Future in the executor via a

FutureCollection object available at `Executor.futures`. These futures can be retrieved anywhere inside your application and queried for status or popped from the collection. Due to memory concerns, the maximum length of the FutureCollection is limited, and the oldest Futures will be dropped when the limit is exceeded.

See *flask_executor.futures.FutureCollection* for more information on how to query futures in a collection.

Example:

```
@app.route('/start-task')
def start_task():
    executor.submit_stored('calc_power', pow, 323, 1235)
    return jsonify({'result':'success'})

@app.route('/get-result')
def get_result():
    if not executor.futures.done('calc_power'):
        future_status = executor.futures._state('calc_power')
        return jsonify({'status': future_status})
    future = executor.futures.pop('calc_power')
    return jsonify({'status': done, 'result': future.result()})
```

Parameters

- **future_key** – Stores the Future for the submitted task inside the executor's `futures` object with the specified key.
- **fn** – The callable to be executed.
- ***args** – A list of positional parameters used with the callable.
- ****kwargs** – A dict of named parameters used with the callable.

Return type concurrent.futures.Future

CHAPTER 8

Default Callbacks

`concurrent.futures.Future` objects can have callbacks attached by using `add_done_callback()`. Flask-Executor lets you specify default callbacks that will be applied to all new futures created by the executor:

```
def some_callback(future):
    # do something with future

executor.add_default_done_callback(some_callback)

# Callback will be added to the below task automatically
executor.submit(pow, 323, 1235)
```

CHAPTER 9

Propagate Exceptions

Normally any exceptions thrown by background threads or processes will be swallowed unless explicitly checked for. To instead surface all exceptions thrown by background tasks, Flask-Executor can add a special default callback that raises any exceptions thrown by tasks submitted to the executor:

```
app.config['EXECUTOR_PROPAGATE_EXCEPTIONS'] = True
```

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index