Microdot

Miguel Grinberg

Jun 04, 2022

CONTENTS

Microdot is a minimalistic Python web framework inspired by Flask, and designed to run on systems with limited resources such as microcontrollers. It runs on standard Python and on MicroPython.

CHAPTER ONE

INSTALLATION

Microdot can be installed with `pip`:

```
pip install microdot
```

For MicroPython, you can install with `upip`.

On platforms where `pip` or `upip` are not viable options, you can manually copy and install the `microdot.py` and `microdot_asyncio.py` source files from the GitHub reposutory into your project directory.

CHAPTER TWO

EXAMPLES

The following is an example of a standard single or multi-threaded web server:

```python
from microdot import Microdot

app = Microdot()

@app.route('/')
def hello(request):
    return 'Hello, world!'

app.run()
```

Microdot also supports the asynchronous model and can be used under `asyncio`. The example that follows is equivalent to the one above, but uses coroutines for concurrency:

```python
from microdot_asyncio import Microdot

app = Microdot()

@app.route('/')
async def hello(request):
    return 'Hello, world!'

app.run()
```

CHAPTER

THREE

API REFERENCE

3.1 `microdot` module

The `microdot` module defines a few classes that help implement HTTP-based servers for MicroPython and standard Python, with multithreading support for Python interpreters that support it.

3.1.1 `Microdot` class

class microdot.**Microdot**

An HTTP application class.

This class implements an HTTP application instance and is heavily influenced by the `Flask` class of the Flask framework. It is typically declared near the start of the main application script.

Example:

```
from microdot import Microdot

app = Microdot()
```

route(*url_pattern*, *methods=None*)

Decorator that is used to register a function as a request handler for a given URL.

Parameters

- **url_pattern** – The URL pattern that will be compared against incoming requests.
- **methods** – The list of HTTP methods to be handled by the decorated function. If omitted, only `GET` requests are handled.

The URL pattern can be a static path (for example, `/users` or `/api/invoices/search`) or a path with dynamic components enclosed in `<` and `>` (for example, `/users/<id>` or `/invoices/<number>/products`). Dynamic path components can also include a type prefix, separated from the name with a colon (for example, `/users/<int:id>`). The type can be `string` (the default), `int`, `path` or `re:[regular-expression]`.

The first argument of the decorated function must be the request object. Any path arguments that are specified in the URL pattern are passed as keyword arguments. The return value of the function must be a *`Response`* instance, or the arguments to be passed to this class.

Example:

```python
@app.route('/')
def index(request):
    return 'Hello, world!'
```

get(*url_pattern*)

Decorator that is used to register a function as a GET request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the route decorator with methods=['GET'].

Example:

```python
@app.get('/users/<int:id>')
def get_user(request, id):
    # ...
```

post(*url_pattern*)

Decorator that is used to register a function as a POST request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the``route`` decorator with methods=['POST'].

Example:

```python
@app.post('/users')
def create_user(request):
    # ...
```

put(*url_pattern*)

Decorator that is used to register a function as a PUT request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the route decorator with methods=['PUT'].

Example:

```python
@app.put('/users/<int:id>')
def edit_user(request, id):
    # ...
```

patch(*url_pattern*)

Decorator that is used to register a function as a PATCH request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the route decorator with methods=['PATCH'].

Example:

```python
@app.patch('/users/<int:id>')
def edit_user(request, id):
    # ...
```

delete(*url_pattern*)

Decorator that is used to register a function as a DELETE request handler for a given URL.

Parameters

url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the `route` decorator with `methods=['DELETE']`.

Example:

```
@app.delete('/users/<int:id>')
def delete_user(request, id):
    # ...
```

before_request(*f*)

Decorator to register a function to run before each request is handled. The decorated function must take a single argument, the request object.

Example:

```
@app.before_request
def func(request):
    # ...
```

after_request(*f*)

Decorator to register a function to run after each request is handled. The decorated function must take two arguments, the request and response objects. The return value of the function must be an updated response object.

Example:

```
@app.before_request
def func(request, response):
    # ...
```

errorhandler(*status_code_or_exception_class*)

Decorator to register a function as an error handler. Error handler functions for numeric HTTP status codes must accept a single argument, the request object. Error handler functions for Python exceptions must accept two arguments, the request object and the exception object.

Parameters

status_code_or_exception_class – The numeric HTTP status code or Python exception class to handle.

Examples:

```
@app.errorhandler(404)
def not_found(request):
    return 'Not found'

@app.errorhandler(RuntimeError)
def runtime_error(request, exception):
    return 'Runtime error'
```

run(*host='0.0.0.0'*, *port=5000*, *debug=False*)

Start the web server. This function does not normally return, as the server enters an endless listening loop. The `shutdown()` function provides a method for terminating the server gracefully.

Parameters

- **host** – The hostname or IP address of the network interface that will be listening for requests. A value of `'0.0.0.0'` (the default) indicates that the server should listen for requests on all the available interfaces, and a value of `127.0.0.1` indicates that the server should listen for requests only on the internal networking interface of the host.
- **port** – The port number to listen for requests. The default is port 5000.
- **debug** – If `True`, the server logs debugging information. The default is `False`.

Example:

```
from microdot import Microdot

app = Microdot()

@app.route('/')
def index():
    return 'Hello, world!'

app.run(debug=True)
```

shutdown()

Request a server shutdown. The server will then exit its request listening loop and the *run()* function will return. This function can be safely called from a route handler, as it only schedules the server to terminate as soon as the request completes.

Example:

```
@app.route('/shutdown')
def shutdown(request):
    request.app.shutdown()
    return 'The server is shutting down...'
```

3.1.2 Request class

class microdot.**Request**(*app*, *client_addr*, *method*, *url*, *http_version*, *headers*, *body=None*, *stream=None*)

An HTTP request class.

Variables

- **app** – The application instance to which this request belongs.
- **client_addr** – The address of the client, as a tuple (host, port).
- **method** – The HTTP method of the request.
- **path** – The path portion of the URL.
- **query_string** – The query string portion of the URL.
- **args** – The parsed query string, as a *MultiDict* object.
- **headers** – A dictionary with the headers included in the request.
- **cookies** – A dictionary with the cookies included in the request.
- **content_length** – The parsed `Content-Length` header.

- **content_type** – The parsed `Content-Type` header.
- **stream** – The input stream, containing the request body.
- **body** – The body of the request, as bytes.
- **json** – The parsed JSON body, as a dictionary or list, or `None` if the request does not have a JSON body.
- **form** – The parsed form submission body, as a *MultiDict* object, or `None` if the request does not have a form submission.
- **g** – A general purpose container for applications to store data during the life of the request.

max_content_length = 16384

Specify the maximum payload size that is accepted. Requests with larger payloads will be rejected with a 413 status code. Applications can change this maximum as necessary.

Example:

```
Request.max_content_length = 1 * 1024 * 1024  # 1MB requests allowed
```

max_body_length = 16384

Specify the maximum payload size that can be stored in `body`. Requests with payloads that are larger than this size and up to `max_content_length` bytes will be accepted, but the application will only be able to access the body of the request by reading from `stream`. Set to 0 if you always access the body as a stream.

Example:

```
Request.max_body_length = 4 * 1024  # up to 4KB bodies read
```

max_readline = 2048

Specify the maximum length allowed for a line in the request. Requests with longer lines will not be correctly interpreted. Applications can change this maximum as necessary.

Example:

```
Request.max_readline = 16 * 1024  # 16KB lines allowed
```

static create(*app*, *client_stream*, *client_addr*)

Create a request object.

Parameters

- **app** – The Microdot application instance.
- **client_stream** – An input stream from where the request data can be read.
- **client_addr** – The address of the client, as a tuple.

This method returns a newly created `Request` object.

3.1.3 `Response` class

class `microdot.Response`(*body=''*, *status_code=200*, *headers=None*, *reason=None*)

An HTTP response class.

Parameters

- **body** – The body of the response. If a dictionary or list is given, a JSON formatter is used to generate the body. If a file-like object or a generator is given, a streaming response is used. If a string is given, it is encoded from UTF-8. Else, the body should be a byte sequence.
- **status_code** – The numeric HTTP status code of the response. The default is 200.
- **headers** – A dictionary of headers to include in the response.
- **reason** – A custom reason phrase to add after the status code. The default is "OK" for responses with a 200 status code and "N/A" for any other status codes.

`set_cookie`(*cookie*, *value*, *path=None*, *domain=None*, *expires=None*, *max_age=None*, *secure=False*, *http_only=False*)

Add a cookie to the response.

Parameters

- **cookie** – The cookie's name.
- **value** – The cookie's value.
- **path** – The cookie's path.
- **domain** – The cookie's domain.
- **expires** – The cookie expiration time, as a `datetime` object.
- **max_age** – The cookie's `Max-Age` value.
- **secure** – The cookie's `secure` flag.
- **http_only** – The cookie's `HttpOnly` flag.

classmethod `redirect`(*location*, *status_code=302*)

Return a redirect response.

Parameters

- **location** – The URL to redirect to.
- **status_code** – The 3xx status code to use for the redirect. The default is 302.

classmethod `send_file`(*filename*, *status_code=200*, *content_type=None*)

Send file contents in a response.

Parameters

- **filename** – The filename of the file.
- **status_code** – The 3xx status code to use for the redirect. The default is 302.
- **content_type** – The `Content-Type` header to use in the response. If omitted, it is generated automatically from the file extension.

Security note: The filename is assumed to be trusted. Never pass filenames provided by the user before validating and sanitizing them first.

3.1.4 `MultiDict` class

class `microdot.`**`MultiDict`**(*initial_dict=None*)

A subclass of dictionary that can hold multiple values for the same key. It is used to hold key/value pairs decoded from query strings and form submissions.

Parameters

`initial_dict` – an initial dictionary of key/value pairs to initialize this object with.

Example:

```
>>> d = MultiDict()
>>> d['sort'] = 'name'
>>> d['sort'] = 'email'
>>> print(d['sort'])
'name'
>>> print(d.getlist('sort'))
['name', 'email']
```

`get`(*key*, *default=None*, *type=None*)

Return the value for a given key.

Parameters

- **`key`** – The key to retrieve.
- **`default`** – A default value to use if the key does not exist.
- **`type`** – A type conversion callable to apply to the value.

If the multidict contains more than one value for the requested key, this method returns the first value only.

Example:

```
>>> d = MultiDict()
>>> d['age'] = '42'
>>> d.get('age')
'42'
>>> d.get('age', type=int)
42
>>> d.get('name', default='noname')
'noname'
```

`getlist`(*key*, *type=None*)

Return all the values for a given key.

Parameters

- **`key`** – The key to retrieve.
- **`type`** – A type conversion callable to apply to the values.

If the requested key does not exist in the dictionary, this method returns an empty list.

Example:

```
>>> d = MultiDict()
>>> d.getlist('items')
[]
```

(continues on next page)

(continued from previous page)

```
>>> d['items'] = '3'
>>> d.getlist('items')
['3']
>>> d['items'] = '56'
>>> d.getlist('items')
['3', '56']
>>> d.getlist('items', type=int)
[3, 56]
```

3.2 `microdot_asyncio` module

The `microdot_asyncio` module defines a few classes that help implement HTTP-based servers for MicroPython and standard Python that use `asyncio` and coroutines.

3.2.1 `Microdot` class

class `microdot_asyncio.`**`Microdot`**

async **`start_server`**(*host='0.0.0.0'*, *port=5000*, *debug=False*)

Start the Microdot web server as a coroutine. This coroutine does not normally return, as the server enters an endless listening loop. The *`shutdown()`* function provides a method for terminating the server gracefully.

Parameters

- **host** – The hostname or IP address of the network interface that will be listening for requests. A value of `'0.0.0.0'` (the default) indicates that the server should listen for requests on all the available interfaces, and a value of `127.0.0.1` indicates that the server should listen for requests only on the internal networking interface of the host.
- **port** – The port number to listen for requests. The default is port 5000.
- **debug** – If `True`, the server logs debugging information. The default is `False`.

This method is a coroutine.

Example:

```
import asyncio
from microdot_asyncio import Microdot

app = Microdot()

@app.route('/')
async def index():
    return 'Hello, world!'

async def main():
    await app.start_server(debug=True)

asyncio.run(main())
```

run(*host='0.0.0.0'*, *port=5000*, *debug=False*)

Start the web server. This function does not normally return, as the server enters an endless listening loop. The *shutdown()* function provides a method for terminating the server gracefully.

Parameters

- **host** – The hostname or IP address of the network interface that will be listening for requests. A value of `'0.0.0.0'` (the default) indicates that the server should listen for requests on all the available interfaces, and a value of `127.0.0.1` indicates that the server should listen for requests only on the internal networking interface of the host.
- **port** – The port number to listen for requests. The default is port 5000.
- **debug** – If `True`, the server logs debugging information. The default is `False`.

Example:

```
from microdot_asyncio import Microdot

app = Microdot()

@app.route('/')
async def index():
    return 'Hello, world!'

app.run(debug=True)
```

shutdown()

Request a server shutdown. The server will then exit its request listening loop and the *run()* function will return. This function can be safely called from a route handler, as it only schedules the server to terminate as soon as the request completes.

Example:

```
@app.route('/shutdown')
def shutdown(request):
    request.app.shutdown()
    return 'The server is shutting down...'
```

after_request(*f*)

Decorator to register a function to run after each request is handled. The decorated function must take two arguments, the request and response objects. The return value of the function must be an updated response object.

Example:

```
@app.before_request
def func(request, response):
    # ...
```

before_request(*f*)

Decorator to register a function to run before each request is handled. The decorated function must take a single argument, the request object.

Example:

```
@app.before_request
def func(request):
    # ...
```

delete(*url_pattern*)

Decorator that is used to register a function as a `DELETE` request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the `route` decorator with `methods=['DELETE']`.

Example:

```
@app.delete('/users/<int:id>')
def delete_user(request, id):
    # ...
```

errorhandler(*status_code_or_exception_class*)

Decorator to register a function as an error handler. Error handler functions for numeric HTTP status codes must accept a single argument, the request object. Error handler functions for Python exceptions must accept two arguments, the request object and the exception object.

Parameters
status_code_or_exception_class – The numeric HTTP status code or Python exception class to handle.

Examples:

```
@app.errorhandler(404)
def not_found(request):
    return 'Not found'

@app.errorhandler(RuntimeError)
def runtime_error(request, exception):
    return 'Runtime error'
```

get(*url_pattern*)

Decorator that is used to register a function as a `GET` request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the `route` decorator with `methods=['GET']`.

Example:

```
@app.get('/users/<int:id>')
def get_user(request, id):
    # ...
```

patch(*url_pattern*)

Decorator that is used to register a function as a `PATCH` request handler for a given URL.

Parameters
url_pattern – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the `route` decorator with `methods=['PATCH']`.

Example:

```
@app.patch('/users/<int:id>')
def edit_user(request, id):
    # ...
```

`post`(*url_pattern*)

Decorator that is used to register a function as a `POST` request handler for a given URL.

Parameters
`url_pattern` – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the``route`` decorator with `methods=['POST']`.

Example:

```
@app.post('/users')
def create_user(request):
    # ...
```

`put`(*url_pattern*)

Decorator that is used to register a function as a `PUT` request handler for a given URL.

Parameters
`url_pattern` – The URL pattern that will be compared against incoming requests.

This decorator can be used as an alias to the `route` decorator with `methods=['PUT']`.

Example:

```
@app.put('/users/<int:id>')
def edit_user(request, id):
    # ...
```

`route`(*url_pattern*, *methods=None*)

Decorator that is used to register a function as a request handler for a given URL.

Parameters

- **`url_pattern`** – The URL pattern that will be compared against incoming requests.
- **`methods`** – The list of HTTP methods to be handled by the decorated function. If omitted, only `GET` requests are handled.

The URL pattern can be a static path (for example, `/users` or `/api/invoices/search`) or a path with dynamic components enclosed in `<` and `>` (for example, `/users/<id>` or `/invoices/<number>/products`). Dynamic path components can also include a type prefix, separated from the name with a colon (for example, `/users/<int:id>`). The type can be `string` (the default), `int`, `path` or `re:[regular-expression]`.

The first argument of the decorated function must be the request object. Any path arguments that are specified in the URL pattern are passed as keyword arguments. The return value of the function must be a *`Response`* instance, or the arguments to be passed to this class.

Example:

```
@app.route('/')
def index(request):
    return 'Hello, world!'
```

3.2.2 `Request` class

class `microdot_asyncio.`**`Request`**(*app*, *client_addr*, *method*, *url*, *http_version*, *headers*, *body=None*, *stream=None*)

async static create(*app*, *client_stream*, *client_addr*)

Create a request object.

Parameters

- **app** – The Microdot application instance.
- **client_stream** – An input stream from where the request data can be read.
- **client_addr** – The address of the client, as a tuple.

This method is a coroutine. It returns a newly created `Request` object.

max_body_length = 16384

Specify the maximum payload size that can be stored in `body`. Requests with payloads that are larger than this size and up to `max_content_length` bytes will be accepted, but the application will only be able to access the body of the request by reading from `stream`. Set to 0 if you always access the body as a stream.

Example:

```
Request.max_body_length = 4 * 1024  # up to 4KB bodies read
```

max_content_length = 16384

Specify the maximum payload size that is accepted. Requests with larger payloads will be rejected with a 413 status code. Applications can change this maximum as necessary.

Example:

```
Request.max_content_length = 1 * 1024 * 1024  # 1MB requests allowed
```

max_readline = 2048

Specify the maximum length allowed for a line in the request. Requests with longer lines will not be correctly interpreted. Applications can change this maximum as necessary.

Example:

```
Request.max_readline = 16 * 1024  # 16KB lines allowed
```

3.2.3 Response class

class microdot_asyncio.Response(*body=''*, *status_code=200*, *headers=None*, *reason=None*)

An HTTP response class.

Parameters

- **body** – The body of the response. If a dictionary or list is given, a JSON formatter is used to generate the body. If a file-like object or an async generator is given, a streaming response is used. If a string is given, it is encoded from UTF-8. Else, the body should be a byte sequence.
- **status_code** – The numeric HTTP status code of the response. The default is 200.
- **headers** – A dictionary of headers to include in the response.
- **reason** – A custom reason phrase to add after the status code. The default is "OK" for responses with a 200 status code and "N/A" for any other status codes.

classmethod redirect(*location*, *status_code=302*)

Return a redirect response.

Parameters

- **location** – The URL to redirect to.
- **status_code** – The 3xx status code to use for the redirect. The default is 302.

classmethod send_file(*filename*, *status_code=200*, *content_type=None*)

Send file contents in a response.

Parameters

- **filename** – The filename of the file.
- **status_code** – The 3xx status code to use for the redirect. The default is 302.
- **content_type** – The `Content-Type` header to use in the response. If omitted, it is generated automatically from the file extension.

Security note: The filename is assumed to be trusted. Never pass filenames provided by the user before validating and sanitizing them first.

set_cookie(*cookie*, *value*, *path=None*, *domain=None*, *expires=None*, *max_age=None*, *secure=False*, *http_only=False*)

Add a cookie to the response.

Parameters

- **cookie** – The cookie's name.
- **value** – The cookie's value.
- **path** – The cookie's path.
- **domain** – The cookie's domain.
- **expires** – The cookie expiration time, as a `datetime` object.
- **max_age** – The cookie's `Max-Age` value.
- **secure** – The cookie's `secure` flag.
- **http_only** – The cookie's `HttpOnly` flag.

3.3 `microdot_wsgi` module

The `microdot_wsgi` module provides an extended `Microdot` class that implements the WSGI protocol and can be used with a complaint WSGI web server such as Gunicorn or uWSGI.

3.3.1 `Microdot` class

class `microdot_wsgi.`**`Microdot`**

`wsgi_app`(*environ*, *start_response*)

A WSGI application callable.

3.4 `microdot_asgi` module

The `microdot_asgi` module provides an extended `Microdot` class that implements the ASGI protocol and can be used with a complaint ASGI server such as Uvicorn.

3.4.1 `Microdot` class

class `microdot_asgi.`**`Microdot`**

async **`asgi_app`**(*scope*, *receive*, *send*)

An ASGI application.

- genindex
- search

INDEX