flask

A microframework based on Werkzeug. It’s extensively documented and follows best practice patterns.

Functions

after_this_request(f)	Executes a function after this request.
copy_current_request_context(f)	A helper function that decorates a function to retain the current request context.
escape((s) -> markup)	Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences.
flash(message[, category])	Flashes a message to the next request.
get_flashed_messages([with_categories, ...])	Pulls all flashed messages from the session and returns them.
get_template_attribute(template_name, attribute)	Loads a macro (or variable) a template exports.
has_app_context()	Works like has_request_context() but for the application context.
has_request_context()	If you have code that wants to test if a request context is there or not this function can be used.
jsonify(*args, **kwargs)	This function wraps dumps() to add a few enhancements that make life easier.
make_response(*args)	Sometimes it is necessary to set additional headers in a view.
redirect(location[, code, Response])	Returns a response object (a WSGI application) that, if called, redirects the client to the target location.
render_template(template_name_or_list, **context)	Renders a template from the template folder with the given context.
render_template_string(source, **context)	Renders a template from the given template source string with the given context.
safe_join(directory, filename)	Safely join *directory* and *filename*.
send_file(filename_or_fp[, mimetype, ...])	Sends the contents of a file to the client.
send_from_directory(directory, filename, ...)	Send a file from a given directory with send_file().

stream_with_context(generator_or_function)	Request contexts disappear when the response is started on the server.
url_for(endpoint, **values)	Generates a URL to the given endpoint with the method provided.

Classes

Blueprint(name, import_name[, ...])	Represents a blueprint.
Config(root_path[, defaults])	Works exactly like a dict but provides ways to fill it from files or special dictionaries.
Flask(import_name[, static_path, ...])	The flask object implements a WSGI application and acts as the central object.
Markup	Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped.
Request(environ[, populate_request, shallow])	The request object used by default in Flask.
Response([response, status, headers, ...])	The response object that is used by default in Flask.
Session	alias of SecureCookieSession

Function Examples

flask.after_this_request

`flask.after_this_request`(*f*)[source]

Executes a function after this request. This is useful to modify response objects. The function is passed the response object and has to return the same or a new one.

Example:

```
@app.route('/')
def index():
    @after_this_request
    def add_header(response):
        response.headers['X-Foo'] = 'Parachute'
        return response
    return 'Hello World!'
```

This is more useful if a function other than the view function wants to modify a response. For instance think of a decorator that wants to add some headers without converting the return value into a response object.

New in version 0.9.

flask.copy_current_request_context

`flask.copy_current_request_context`(*f*)[source]

A helper function that decorates a function to retain the current request context. This is useful when working with greenlets. The moment the function is decorated a copy of the request context is created and then pushed when the function is called.

Example:

```
import gevent
from flask import copy_current_request_context

@app.route('/')
def index():
    @copy_current_request_context
    def do_some_work():
        # do some work here, it can access flask.request like you
        # would otherwise in the view function.
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

New in version 0.10.

flask.escape

flask.escape(*s*) → markup

Convert the characters &, <, >, ‘, and ” in string s to HTML-safe sequences. Use this if you need to display text that might contain such characters in HTML. Marks return value as markup string.

flask.flash

flask.flash(*message*, *category='message'*)[source]

Flashes a message to the next request. In order to remove the flashed message from the session and to display it to the user, the template has to call `get_flashed_messages()`.

Changed in version 0.3: *category* parameter added.

Parameters:

- **message** – the message to be flashed.
- **category** – the category for the message. The following values are recommended: `'message'` for any kind of message, `'error'` for errors, `'info'` for information messages and `'warning'` for warnings. However any kind of string can be used as category.

flask.get_flashed_messages

flask.get_flashed_messages(*with_categories=False*, *category_filter=[]*)[source]

Pulls all flashed messages from the session and returns them. Further calls in the same request to the function will return the same messages. By default just the messages are returned, but when *with_categories* is set to `True`, the return value will be a list of tuples in the form `(category, message)` instead.

Filter the flashed messages to one or more categories by providing those categories in *category_filter*. This allows rendering categories in separate html blocks. The *with_categories* and *category_filter* arguments are distinct:

- *with_categories* controls whether categories are returned with message text (`True` gives a tuple, where `False` gives just the message text).
- *category_filter* filters the messages down to only those matching the provided categories.

See Message Flashing for examples.

Changed in version 0.3: *with_categories* parameter added.

Changed in version 0.9: *category_filter* parameter added.

Parameters:
- **with_categories** – set to `True` to also receive categories.
- **category_filter** – whitelist of categories to limit return values

flask.get_template_attribute

`flask.get_template_attribute`(*template_name*, *attribute*)[source]

Loads a macro (or variable) a template exports. This can be used to invoke a macro from within Python code. If you for example have a template named `_cider.html` with the following contents:

```
{% macro hello(name) %}Hello {{ name }}!{% endmacro %}
```

You can access this from Python code like this:

```
hello = get_template_attribute('_cider.html', 'hello')
return hello('World')
```

New in version 0.2.

Parameters:
- **template_name** – the name of the template
- **attribute** – the name of the variable of macro to access

flask.has_app_context

`flask.has_app_context`()[source]

Works like `has_request_context()` but for the application context. You can also just do a boolean check on the `current_app` object instead.

New in version 0.9.

flask.has_request_context

`flask.has_request_context`()[source]

If you have code that wants to test if a request context is there or not this function can be used. For instance, you may want to take advantage of request information if the request object is available, but fail silently if it is unavailable.

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and has_request_context():
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

Alternatively you can also just test any of the context bound objects (such as `request` or `g` for truthness):

```
class User(db.Model):

    def __init__(self, username, remote_addr=None):
        self.username = username
        if remote_addr is None and request:
            remote_addr = request.remote_addr
        self.remote_addr = remote_addr
```

New in version 0.7.

---

flask.jsonify

`flask.jsonify`(**args*, ***kwargs*)[source]

This function wraps `dumps()` to add a few enhancements that make life easier. It turns the JSON output into a `Response` object with the *application/json* mimetype. For convenience, it also converts multiple arguments into an array or multiple keyword arguments into a dict. This means that both `jsonify(1,2,3)` and `jsonify([1,2,3])` serialize to `[1,2,3]`.

For clarity, the JSON serialization behavior has the following differences from `dumps()`:

1. Single argument: Passed straight through to `dumps()`.
2. Multiple arguments: Converted to an array before being passed to `dumps()`.
3. Multiple keyword arguments: Converted to a dict before being passed to `dumps()`.
4. Both args and kwargs: Behavior undefined and will throw an exception.

Example usage:

```
from flask import jsonify

@app.route('/_get_current_user')
def get_current_user():
    return jsonify(username=g.user.username,
                   email=g.user.email,
                   id=g.user.id)
```

This will send a JSON response like this to the browser:

```
{
    "username": "admin",
    "email": "admin@localhost",
    "id": 42
}
```

Changed in version 0.11: Added support for serializing top-level arrays. This introduces a security risk in ancient browsers. See JSON Security for details.

This function's response will be pretty printed if it was not requested with `X-Requested-With: XMLHttpRequest` to simplify debugging unless the `JSONIFY_PRETTYPRINT_REGULAR` config parameter is set to false. Compressed (not pretty) formatting currently means no indents and no spaces after separators.

New in version 0.2.

flask.make_response

`flask.make_response`(**args*)[source]

Sometimes it is necessary to set additional headers in a view. Because views do not have to return response objects but can return a value that is converted into a response object by Flask itself, it becomes tricky to add headers to it. This function can be called instead of using a return and you will get a response object which you can use to attach headers.

If view looked like this and you want to add a new header:

```
def index():
    return render_template('index.html', foo=42)
```

You can now do something like this:

```
def index():
    response = make_response(render_template('index.html', foo=42))
    response.headers['X-Parachutes'] = 'parachutes are cool'
    return response
```

This function accepts the very same arguments you can return from a view function. This for example creates a response with a 404 error code:

```
response = make_response(render_template('not_found.html'), 404)
```

The other use case of this function is to force the return value of a view function into a response which is helpful with view decorators:

```
response = make_response(view_function())
response.headers['X-Parachutes'] = 'parachutes are cool'
```

Internally this function does the following things:

- if no arguments are passed, it creates a new response argument
- if one argument is passed, `flask.Flask.make_response()` is invoked with it.
- if more than one argument is passed, the arguments are passed to the `flask.Flask.make_response()` function as tuple.

New in version 0.6.

flask.redirect

`flask.redirect`(*location*, *code=302*, *Response=None*)[source]

Returns a response object (a WSGI application) that, if called, redirects the client to the target location. Supported codes are 301, 302, 303, 305, and 307. 300 is not supported because it's not a real redirect and 304 because it's the answer for a request with a request with defined If-Modified-Since headers.

New in version 0.6: The location can now be a unicode string that is encoded using the `iri_to_uri()` function.

New in version 0.10: The class used for the Response object can now be passed in.

Parameters:
- **location** – the location the response should redirect to.
- **code** – the redirect status code. defaults to 302.
- **Response** (*class*) – a Response class to use when instantiating a response. The default is `werkzeug.wrappers.Response` if unspecified.

flask.render_template

`flask.render_template`(*template_name_or_list*, ***context*)[source]

Renders a template from the template folder with the given context.

Parameters:
- **template_name_or_list** – the name of the template to be rendered, or an iterable with template names the first one existing will be rendered
- **context** – the variables that should be available in the context of the template.

flask.render_template_string

`flask.render_template_string`(*source*, ***context*)[source]

Renders a template from the given template source string with the given context. Template variables will be autoescaped.

Parameters:
- **source** – the source code of the template to be rendered
- **context** – the variables that should be available in the context of the template.

flask.safe_join

`flask.safe_join`(*directory*, *filename*)[source]

Safely join *directory* and *filename*.

Example usage:

```
@app.route('/wiki/<path:filename>')
def wiki_page(filename):
    filename = safe_join(app.config['WIKI_FOLDER'], filename)
    with open(filename, 'rb') as fd:
        content = fd.read()  # Read and process the file content...
```

Parameters:
- **directory** – the base directory.
- **filename** – the untrusted filename relative to that directory.

Raises: `NotFound` if the resulting path would fall out of *directory*.

flask.send_file

`flask.send_file`(*filename_or_fp*, *mimetype=None*, *as_attachment=False*, *attachment_filename=None*, *add_etags=True*, *cache_timeout=None*, *conditional=False*)[source]

Sends the contents of a file to the client. This will use the most efficient method available and configured. By default it will try to use the WSGI server's file_wrapper support. Alternatively you can set the application's `use_x_sendfile` attribute to `True` to directly emit an `X-Sendfile` header. This however requires support of the underlying webserver for `X-Sendfile`.

By default it will try to guess the mimetype for you, but you can also explicitly provide one. For extra security you probably want to send certain files as attachment (HTML for instance). The mimetype guessing requires a *filename* or an *attachment_filename* to be provided.

Please never pass filenames to this function from user sources; you should use `send_from_directory()` instead.

New in version 0.2.

New in version 0.5: The *add_etags*, *cache_timeout* and *conditional* parameters were added. The default behavior is now to attach etags.

Changed in version 0.7: mimetype guessing and etag support for file objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself. This functionality will be removed in Flask 1.0

Changed in version 0.9: cache_timeout pulls its default from application config, when None.

Parameters :

- **filename_or_fp** – the filename of the file to send in *latin-1*. This is relative to the `root_path` if a relative path is specified. Alternatively a file object might be provided in which case `X-Sendfile` might not work and fall back to the traditional method. Make sure that the file pointer is positioned at the start of data to send before calling `send_file()`.
- **mimetype** – the mimetype of the file if provided, otherwise auto detection happens.
- **as_attachment** – set to `True` if you want to send this file with a `Content-Disposition: attachment` header.
- **attachment_filename** – the filename for the attachment if it differs from the file's filename.
- **add_etags** – set to `False` to disable attaching of etags.
- **conditional** – set to `True` to enable conditional responses.
- **cache_timeout** – the timeout in seconds for the headers. When `None` (default), this value is set by `get_send_file_max_age()` of `current_app`.

flask.send_from_directory

`flask.send_from_directory`(*directory*, *filename*, ***options*)[source]

Send a file from a given directory with `send_file()`. This is a secure way to quickly expose static files from an upload folder or something similar.

Example usage:

```
@app.route('/uploads/<path:filename>')
def download_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename, as_attachment=True)
```

Sending files and Performance

It is strongly recommended to activate either `X-Sendfile` support in your webserver or (if no authentication happens) to tell the webserver to serve files for the given path on its own without calling into the web application for improved performance.

New in version 0.5.

Parameters:
- **directory** – the directory where all the files are stored.
- **filename** – the filename relative to that directory to download.
- **options** – optional keyword arguments that are directly forwarded to `send_file()`.

flask.stream_with_context

`flask.stream_with_context`(*generator_or_function*)[source]

Request contexts disappear when the response is started on the server. This is done for efficiency reasons and to make it less likely to encounter memory leaks with badly written WSGI middlewares. The downside is that if you are using streamed responses, the generator cannot access request bound information any more.

This function however can help you keep the context around for longer:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    @stream_with_context
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(generate())
```

Alternatively it can also be used around a specific generator:

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(stream_with_context(generate()))
```

New in version 0.9.

flask.url_for

`flask.url_for`(*endpoint*, ***values*)[source]

Generates a URL to the given endpoint with the method provided.

Variable arguments that are unknown to the target endpoint are appended to the generated URL as query arguments. If the value of a query argument is `None`, the whole pair is skipped. In case blueprints are active you can shortcut references to the same blueprint by prefixing the local endpoint with a dot (`.`).

This will reference the index function local to the current blueprint:

```
url_for('.index')
```

For more information, head over to the Quickstart.

To integrate applications, `Flask` has a hook to intercept URL build errors through `Flask.url_build_error_handlers`. The *url_for* function results in a `BuildError` when the current app does not have a URL for the given endpoint and values. When it does, the `current_app` calls its `url_build_error_handlers` if it is not `None`, which can return a string to use as the result of *url_for* (instead of *url_for*'s default to raise the `BuildError` exception) or re-raise the exception. An example:

```
def external_url_handler(error, endpoint, values):
    "Looks up an external URL when `url_for` cannot build a URL."
    # This is an example of hooking the build_error_handler.
    # Here, lookup_url is some utility function you've built
    # which looks up the endpoint in some external URL registry.
    url = lookup_url(endpoint, **values)
    if url is None:
        # External lookup did not have a URL.
        # Re-raise the BuildError, in context of original traceback.
        exc_type, exc_value, tb = sys.exc_info()
        if exc_value is error:
            raise exc_type, exc_value, tb
        else:
            raise error
    # url_for will use this result, instead of raising BuildError.
    return url

app.url_build_error_handlers.append(external_url_handler)
```

Here, *error* is the instance of `BuildError`, and *endpoint* and *values* are the arguments passed into *url_for*. Note that this is for building URLs outside the current application, and not for handling 404 NotFound errors.

New in version 0.10: The *_scheme* parameter was added.

New in version 0.9: The *_anchor* and *_method* parameters were added.

New in version 0.9: Calls `Flask.handle_build_error()` on `BuildError`.

Parameters:

- **endpoint** – the endpoint of the URL (name of the function)
- **values** – the variable arguments of the URL rule
- **_external** – if set to `True`, an absolute URL is generated. Server address can be changed via `SERVER_NAME` configuration variable which defaults

to *localhost*.

- **_scheme** – a string specifying the desired URL scheme. The *_external* parameter must be set to `True` or a `ValueError` is raised. The default behavior uses the same scheme as the current request, or `PREFERRED_URL_SCHEME` from the app configuration if no request context is available. As of Werkzeug 0.10, this also can be set to an empty string to build protocol-relative URLs.
- **_anchor** – if provided this is added as anchor to the URL.
- **_method** – if provided this explicitly specifies an HTTP method.