API

This part of the documentation covers all the interfaces of Flask. For parts where Flask depends on external libraries, we document the most important right here and provide links to the canonical documentation.

Application Object

class `flask.`**`Flask`**(*import_name*, *static_url_path=None*, *static_folder='static'*, *static_host=None*, *host_matching=False*, *subdomain_matching=False*, *template_folder='templates'*, *instance_path=None*, *instance_relative_config=False*, *root_path=None*)

The flask object implements a WSGI application and acts as the central object. It is passed the name of the module or package of the application. Once it is created it will act as a central registry for the view functions, the URL rules, template configuration and much more.

The name of the package is used to resolve resources from inside the package or the folder the module is contained in depending on if the package parameter resolves to an actual python package (a folder with an `__init__.py` file inside) or a standard module (just a `.py` file).

For more information about resource loading, see **`open_resource()`**.

Usually you create a **`Flask`** instance in your main module or in the `__init__.py` file of your package like this:

```
from flask import Flask
app = Flask(__name__)
```

About the First Parameter:

The idea of the first parameter is to give Flask an idea of what belongs to your application. This name is used to find resources on the filesystem, can be used by extensions to improve debugging information and a lot more.

So it's important what you provide there. If you are using a single module, *__name__* is always the correct value. If you however are using a package, it's usually recommended to hardcode the name of your package there.

For example if your application is defined in `yourapplication/app.py` you should create it with one of the two versions below:

```
app = Flask('yourapplication')
app = Flask(__name__.split('.')[0])
```

Why is that? The application will work even with *__name__*, thanks to how resources are looked up. However it will make debugging more painful. Certain extensions can make assumptions based on the import name of your application. For example the Flask-SQLAlchemy extension will look for the code in your application that triggered an SQL query in debug mode. If the import name is not properly set up, that debugging information is lost. (For example it would only pick up SQL queries in *yourapplication.app* and not *yourapplication.views.frontend*)

New in version 1.0: The `host_matching` and `static_host` parameters were added.

New in version 1.0: The `subdomain_matching` parameter was added. Subdomain matching needs to be enabled manually now. Setting **`SERVER_NAME`** does not implicitly enable it.

▶ *Changelog*

Parameters:
- **import_name** – the name of the application package
- **static_url_path** – can be used to specify a different path for the static files on the web. Defaults to the name of the *static_folder* folder.
- **static_folder** – the folder with static files that should be served at *static_url_path*. Defaults to the `'static'` folder in the root path of the application.
- **static_host** – the host to use when adding the static route. Defaults to None. Required when using `host_matching=True` with a `static_folder` configured.
- **host_matching** – set `url_map.host_matching` attribute. Defaults to False.
- **subdomain_matching** – consider the subdomain relative to **`SERVER_NAME`** when matching routes. Defaults to False.
- **template_folder** – the folder that contains the templates that should be used by the application. Defaults to `'templates'` folder in the root path of the application.
- **instance_path** – An alternative instance path for the application. By default the folder `'instance'` next to the package or module is assumed to be the instance path.
- **instance_relative_config** – if set to `True` relative filenames for loading the config are assumed to be relative to the instance path instead of the application root.
- **root_path** – Flask by default will automatically calculate the path to the root of the application. In certain situations this cannot be achieved (for instance if the package is a Python 3 namespace package) and needs to be manually defined.

`add_template_filter`(*f*, *name=None*)

Register a custom template filter. Works exactly like the **`template_filter()`** decorator.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

add_template_global(*f*, *name=None*)

Register a custom template global function. Works exactly like the **template_global()** decorator.

▶ *Changelog*

Parameters: **name** – the optional name of the global function, otherwise the function name will be used.

add_template_test(*f*, *name=None*)

Register a custom template test. Works exactly like the **template_test()** decorator.

▶ *Changelog*

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, *provide_automatic_options=None*, ***options*)

Connects a URL rule. Works exactly like the **route()** decorator. If a view_func is provided it will be registered with the endpoint.

Basically this example:

```
@app.route('/')
def index():
    pass
```

Is equivalent to the following:

```
def index():
    pass
app.add_url_rule('/', 'index', index)
```

If the view_func is not provided you will need to connect the endpoint to a view function like so:

```
app.view_functions['index'] = index
```

Internally **route()** invokes **add_url_rule()** so if you want to customize the behavior via subclassing you only need to change this method.

For more information refer to URL Route Registrations.

▶ *Changelog*

Parameters:
- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **view_func** – the function to call when serving a request to the provided endpoint
- **provide_automatic_options** – controls whether the `OPTIONS` method should be added automatically. This can also be controlled by setting the `view_func.provide_automatic_options = False` before adding the rule.
- **options** – the options to be forwarded to the underlying **`Rule`** object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

`after_request`(*f*)

Register a function to be run after each request.

Your function must take one parameter, an instance of **`response_class`** and return a new response object or the same (see **`process_response()`**).

As of Flask 0.7 this function might not be executed at the end of the request in case an unhandled exception occurred.

`after_request_funcs` = *None*

A dictionary with lists of functions that should be called after each request. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. This can for example be used to close database connections. To register a function here, use the **`after_request()`** decorator.

`app_context`()

Create an **`AppContext`**. Use as a `with` block to push the context, which will make **`current_app`** point at this application.

An application context is automatically pushed by **`RequestContext.push()`** when handling a request, and when running a CLI command. Use this to manually create a context outside of these situations.

```
with app.app_context():
    init_db()
```

See The Application Context.

▶ *Changelog*

app_ctx_globals_class

alias of **flask.ctx._AppCtxGlobals**

auto_find_instance_path()

Tries to locate the instance path if it was not provided to the constructor of the application class. It will basically calculate the path to a folder named `instance` next to your main file or the package.

▶ *Changelog*

before_first_request(f)

Registers a function to be run before the first request to this instance of the application.

The function will be called without any arguments and its return value is ignored.

▶ *Changelog*

before_first_request_funcs = *None*

A list of functions that will be called at the beginning of the first request to this instance. To register a function, use the **before_first_request()** decorator.

▶ *Changelog*

before_request(f)

Registers a function to run before each request.

For example, this can be used to open a database connection, or to load the logged in user from the session.

The function will be called without any arguments. If it returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

before_request_funcs = *None*

A dictionary with lists of functions that will be called at the beginning of each request. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use the **before_request()** decorator.

blueprints = *None*

all the attached blueprints in a dictionary by name. Blueprints can be attached multiple times so this dictionary does not tell you how often they got attached.

▶ *Changelog*

cli = *None*

The click command line context for this application. Commands registered here show up in the **flask** command once the application has been discovered. The default commands are provided by Flask itself and can be overridden.

This is an instance of a **click.Group** object.

config = *None*

The configuration dictionary as **Config**. This behaves exactly like a regular dictionary but supports additional methods to load a config from files.

config_class

alias of **flask.config.Config**

context_processor(*f*)

Registers a template context processor function.

create_global_jinja_loader()

Creates the loader for the Jinja2 environment. Can be used to override just the loader and keeping the rest unchanged. It's discouraged to override this function. Instead one should override the **jinja_loader()** function instead.

The global loader dispatches between the loaders of the application and the individual blueprints.

▶ *Changelog*

create_jinja_environment()

Creates the Jinja2 environment based on **jinja_options** and **select_jinja_autoescape()**. Since 0.7 this also adds the Jinja2 globals and filters after initialization. Override this function to customize the behavior.

▶ *Changelog*

create_url_adapter(*request*)

Creates a URL adapter for the given request. The URL adapter is created at a point where the request context is not yet set up so the request is passed explicitly.

Changed in version 1.0: **SERVER_NAME** no longer implicitly enables subdomain matching. Use **subdomain_matching** instead.

▶ *Changelog*

`debug`

Whether debug mode is enabled. When using `flask run` to start the development server, an interactive debugger will be shown for unhandled exceptions, and the server will be reloaded when code changes. This maps to the **`DEBUG`** config key. This is enabled when **`env`** is `'development'` and is overridden by the `FLASK_DEBUG` environment variable. It may not behave as expected if set in code.

Do not enable debug mode when deploying in production.

Default: `True` if **`env`** is `'development'`, or `False` otherwise.

`default_config` = *{'APPLICATION_ROOT': '/', 'DEBUG': None, 'ENV': None, 'EXPLAIN_TEMPLATE_LOADING': False, 'JSONIFY_MIMETYPE': 'application/json', 'JSONIFY_PRETTYPRINT_REGULAR': False, 'JSON_AS_ASCII': True, 'JSON_SORT_KEYS': True, 'MAX_CONTENT_LENGTH': None, 'MAX_COOKIE_SIZE': 4093, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(31), 'PREFERRED_URL_SCHEME': 'http', 'PRESERVE_CONTEXT_ON_EXCEPTION': None, 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None, 'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(0, 43200), 'SERVER_NAME': None, 'SESSION_COOKIE_DOMAIN': None, 'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_SAMESITE': None, 'SESSION_COOKIE_SECURE': False, 'SESSION_REFRESH_EACH_REQUEST': True, 'TEMPLATES_AUTO_RELOAD': None, 'TESTING': False, 'TRAP_BAD_REQUEST_ERRORS': None, 'TRAP_HTTP_EXCEPTIONS': False, 'USE_X_SENDFILE': False}*

Default configuration parameters.

`dispatch_request`()

Does the request dispatching. Matches the URL and returns the return value of the view or error handler. This does not have to be a response object. In order to convert the return value to a proper response object, call **`make_response()`**.

▶ *Changelog*

`do_teardown_appcontext`(*exc=<object object>*)

Called right before the application context is popped.

When handling a request, the application context is popped after the request context. See **`do_teardown_request()`**.

This calls all functions decorated with **teardown_appcontext()**. Then the **appcontext_tearing_down** signal is sent.

This is called by **AppContext.pop()**.

▶ *Changelog*

do_teardown_request(*exc=<object object>*)

Called after the request is dispatched and the response is returned, right before the request context is popped.

This calls all functions decorated with **teardown_request()**, and **Blueprint.teardown_request()** if a blueprint handled the request. Finally, the **request_tearing_down** signal is sent.

This is called by **RequestContext.pop()**, which may be delayed during testing to maintain access to resources.

Parameters: **exc** – An unhandled exception raised while dispatching the request. Detected from the current exception information if not passed. Passed to each teardown function.

▶ *Changelog*

endpoint(*endpoint*)

A decorator to register a function as an endpoint. Example:

```
@app.endpoint('example.endpoint')
def example():
    return "example"
```

Parameters: **endpoint** – the name of the endpoint

env

What environment the app is running in. Flask and extensions may enable behaviors based on the environment, such as enabling debug mode. This maps to the **ENV** config key. This is set by the **FLASK_ENV** environment variable and may not behave as expected if set in code.

Do not enable development when deploying in production.

Default: `'production'`

error_handler_spec = *None*

A dictionary of all registered error handlers. The key is `None` for error handlers active on the application, otherwise the key is the name of the blueprint. Each key points to another

dictionary where the key is the status code of the http exception. The special key `None` points to a list of tuples where the first item is the class for the instance check and the second the error handler function.

To register an error handler, use the **`errorhandler()`** decorator.

errorhandler(*code_or_exception*)

Register a function to handle errors by code or exception class.

A decorator that is used to register a function given an error code. Example:

```
@app.errorhandler(404)
def page_not_found(error):
    return 'This page does not exist', 404
```

You can also register handlers for arbitrary exceptions:

```
@app.errorhandler(DatabaseError)
def special_exception_handler(error):
    return 'Database connection failed', 500
```

▶ *Changelog*

Parameters: **code_or_exception** – the code as integer for the handler, or an arbitrary exception

extensions = *None*

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things. For backwards compatibility extensions should register themselves like this:

```
if not hasattr(app, 'extensions'):
    app.extensions = {}
app.extensions['extensionname'] = SomeObject()
```

The key must match the name of the extension module. For example in case of a "Flask-Foo" extension in *flask_foo*, the key would be `'foo'`.

▶ *Changelog*

full_dispatch_request()

Dispatches the request and on top of that performs request pre and postprocessing as well as HTTP exception catching and error handling.

▶ *Changelog*

get_send_file_max_age(*filename*)

Provides default cache_timeout for the **send_file()** functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of **current_app**.

Static file functions such as **send_from_directory()** use this function, and **send_file()** calls this function on **current_app** when the given cache_timeout is `None`. If a cache_timeout is given in **send_file()**, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

▶ *Changelog*

got_first_request

This attribute is set to `True` if the application started handling the first request.

▶ *Changelog*

handle_exception(*e*)

Default exception handling that kicks in when an exception occurs that is not caught. In debug mode the exception will be re-raised immediately, otherwise it is logged and the handler for a 500 internal server error is used. If no such handler exists, a default 500 internal server error message is displayed.

▶ *Changelog*

handle_http_exception(*e*)

Handles an HTTP exception. By default this will invoke the registered error handlers and fall back to returning the exception as response.

▶ *Changelog*

handle_url_build_error(*error*, *endpoint*, *values*)

Handle **BuildError** on **url_for()**.

handle_user_exception(*e*)

This method is called whenever an exception occurs that should be handled. A special case are **`HTTPException`**s which are forwarded by this function to the **`handle_http_exception()`** method. This function will either return a response value or reraise the exception with the same traceback.

Changed in version 1.0: Key errors raised from request data like `form` show the the bad key in debug mode rather than a generic bad request message.

▶ *Changelog*

`has_static_folder`

This is `True` if the package bound object's container has a folder for static files.

▶ *Changelog*

`import_name` = *None*

The name of the package or module that this app belongs to. Do not change this once it is set by the constructor.

`inject_url_defaults`(*endpoint*, *values*)

Injects the URL defaults for the given endpoint directly into the values dictionary passed. This is used internally and automatically called on URL building.

▶ *Changelog*

`instance_path` = *None*

Holds the path to the instance folder.

▶ *Changelog*

`iter_blueprints`()

Iterates over all blueprints by the order they were registered.

▶ *Changelog*

`jinja_env`

The Jinja2 environment used to load templates.

`jinja_environment`

alias of **`flask.templating.Environment`**

`jinja_loader`

The Jinja loader for this package bound object.

▶ *Changelog*

`jinja_options` = {*'extensions': ['jinja2.ext.autoescape', 'jinja2.ext.with_']*}

Options that are passed directly to the Jinja2 environment.

`json_decoder`

alias of **`flask.json.JSONDecoder`**

`json_encoder`

alias of **`flask.json.JSONEncoder`**

`log_exception`(*exc_info*)

Logs an exception. This is called by **`handle_exception()`** if debugging is disabled and right before the handler is called. The default implementation logs the exception as error on the **`logger`**.

▶ *Changelog*

`logger`

The `'flask.app'` logger, a standard Python **`Logger`**.

In debug mode, the logger's **`level`** will be set to **`DEBUG`**.

If there are no handlers configured, a default handler will be added. See Logging for more information.

Changed in version 1.0: Behavior was simplified. The logger is always named `flask.app`. The level is only set during configuration, it doesn't check `app.debug` each time. Only one format is used, not different ones depending on `app.debug`. No handlers are removed, and a handler is only added if no handlers are already configured.

▶ *Changelog*

`make_config`(*instance_relative=False*)

Used to create the config attribute by the Flask constructor. The *instance_relative* parameter is passed in from the constructor of Flask (there named *instance_relative_config*) and indicates if the config should be relative to the instance path or the root path of the application.

▶ *Changelog*

`make_default_options_response`()

This method is called to create the default `OPTIONS` response. This can be changed through subclassing to change the default behavior of `OPTIONS` responses.

▶ *Changelog*

`make_null_session`()

Creates a new instance of a missing session. Instead of overriding this method we recommend replacing the **`session_interface`**.

▶ *Changelog*

`make_response`(*rv*)

Convert the return value from a view function to an instance of **`response_class`**.

Parameters: **rv** –
the return value from the view function. The view function must return a response. Returning `None`, or the view ending without returning, is not allowed. The following types are allowed for `view_rv`:

`str` (`unicode` in Python 2)
: A response object is created with the string encoded to UTF-8 as the body.

`bytes` (`str` in Python 2)
: A response object is created with the bytes as the body.

`tuple`
: Either `(body, status, headers)`, `(body, status)`, or `(body, headers)`, where `body` is any of the other types allowed here, `status` is a string or an integer, and `headers` is a dictionary or a list of `(key, value)` tuples. If `body` is a **`response_class`** instance, `status` overwrites the exiting value and `headers` are extended.

`response_class`
: The object is returned unchanged.

other **`Response`** class
: The object is coerced to **`response_class`**.

`callable()`
: The function is called as a WSGI application. The result is used to create a response object.

▶ *Changelog*

`make_shell_context`()

Returns the shell context for an interactive shell for this application. This runs all the registered shell context processors.

▶ *Changelog*

name

The name of the application. This is usually the import name with the difference that it's guessed from the run file if the import name is main. This name is used as a display name when Flask needs the name of the application. It can be set and overridden to change the value.

▶ *Changelog*

open_instance_resource(*resource*, *mode='rb'*)

Opens a resource from the application's instance folder (**instance_path**). Otherwise works like **open_resource()**. Instance resources can also be opened for writing.

Parameters:
- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

open_resource(*resource*, *mode='rb'*)

Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters:
- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

open_session(*request*)

Creates or opens a new session. Default implementation stores all session data in a signed cookie. This requires that the **secret_key** is set. Instead of overriding this method we recommend replacing the **session_interface**.

Parameters: **request** – an instance of **request_class**.

permanent_session_lifetime

A **timedelta** which is used to set the expiration date of a permanent session. The default is 31 days which makes a permanent session survive for roughly one month.

This attribute can also be configured from the config with the `PERMANENT_SESSION_LIFETIME` configuration key. Defaults to `timedelta(days=31)`

preprocess_request()

Called before the request is dispatched. Calls **url_value_preprocessors** registered with the app and the current blueprint (if any). Then calls **before_request_funcs** registered with the app and the blueprint.

If any **before_request()** handler returns a non-None value, the value is handled as if it was the return value from the view, and further request handling is stopped.

preserve_context_on_exception

Returns the value of the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration value in case it's set, otherwise a sensible default is returned.

▶ *Changelog*

process_response(*response*)

Can be overridden in order to modify the response object before it's sent to the WSGI server. By default this will call all the **after_request()** decorated functions.

▶ *Changelog*

Parameters: **response** – a **response_class** object.
Returns: a new response object or the same, has to be an instance of **response_class**.

propagate_exceptions

Returns the value of the `PROPAGATE_EXCEPTIONS` configuration value in case it's set, otherwise a sensible default is returned.

▶ *Changelog*

register_blueprint(*blueprint*, ***options*)

Register a **Blueprint** on the application. Keyword arguments passed to this method will override the defaults set on the blueprint.

Calls the blueprint's **register()** method after recording the blueprint in the application's **blueprints**.

Parameters:
- **blueprint** – The blueprint to register.
- **url_prefix** – Blueprint routes will be prefixed with this.
- **subdomain** – Blueprint routes will match on this subdomain.
- **url_defaults** – Blueprint routes will use these default values for view arguments.
- **options** – Additional keyword arguments are passed to `BlueprintSetupState`. They can be accessed in `record()` callbacks.

▶ *Changelog*

`register_error_handler`(*code_or_exception*, *f*)

Alternative error attach function to the `errorhandler()` decorator that is more straightforward to use for non decorator usage.

▶ *Changelog*

`request_class`

alias of `flask.wrappers.Request`

`request_context`(*environ*)

Create a `RequestContext` representing a WSGI environment. Use a `with` block to push the context, which will make `request` point at this request.

See The Request Context.

Typically you should not call this from your own code. A request context is automatically pushed by the `wsgi_app()` when handling a request. Use `test_request_context()` to create an environment and context instead of this method.

Parameters: **environ** – a WSGI environment

`response_class`

alias of `flask.wrappers.Response`

`root_path` = *None*

Absolute path to the package on the filesystem. Used to look up resources contained in the package.

`route`(*rule*, ***options*)

A decorator that is used to register a view function for a given URL rule. This does the same thing as `add_url_rule()` but is intended for decorator usage:

```
@app.route('/')
def index():
    return 'Hello World'
```

For more information refer to URL Route Registrations.

Parameters:
- **rule** – the URL rule as string
- **endpoint** – the endpoint for the registered URL rule. Flask itself assumes the name of the view function as endpoint
- **options** – the options to be forwarded to the underlying `Rule` object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling.

run(*host=None*, *port=None*, *debug=None*, *load_dotenv=True*, ***options*)

Runs the application on a local development server.

Do not use `run()` in a production setting. It is not intended to meet security and performance requirements for a production server. Instead, see Deployment Options for WSGI server recommendations.

If the **debug** flag is set the server will automatically reload for code changes and show a debugger in case an exception happened.

If you want to run the application in debug mode, but disable the code execution on the interactive debugger, you can pass `use_evalex=False` as parameter. This will keep the debugger's traceback screen active, but disable code execution.

It is not recommended to use this function for development with automatic reloading as this is badly supported. Instead you should be using the **flask** command line script's `run` support.

Keep in Mind:

Flask will suppress any server error with a generic error page unless it is in debug mode. As such to enable just the interactive debugger without the code reloading, you have to invoke `run()` with `debug=True` and `use_reloader=False`. Setting `use_debugger` to `True` without being in debug mode won't catch any exceptions because there won't be any to catch.

Parameters:
- **host** – the hostname to listen on. Set this to `'0.0.0.0'` to have the server available externally as well. Defaults to `'127.0.0.1'` or the host in the `SERVER_NAME` config variable if present.
- **port** – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.

- **debug** – if given, enable or disable debug mode. See **debug**.
- **load_dotenv** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.
- **options** – the options to be forwarded to the underlying Werkzeug server. See **werkzeug.serving.run_simple()** for more information.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from `.env` and `.flaskenv` files.

If set, the **FLASK_ENV** and **FLASK_DEBUG** environment variables will override **env** and **debug**.

Threaded mode is enabled by default.

▶ *Changelog*

save_session(*session*, *response*)

Saves the session if it needs updates. For the default implementation, check **open_session()**. Instead of overriding this method we recommend replacing the **session_interface**.

Parameters:
- **session** – the session to be saved (a **SecureCookie** object)
- **response** – an instance of **response_class**

secret_key

If a secret key is set, cryptographic components can use this to sign cookies and other things. Set this to a complex random value when you want to use the secure cookie for instance.

This attribute can also be configured from the config with the **SECRET_KEY** configuration key. Defaults to `None`.

select_jinja_autoescape(*filename*)

Returns `True` if autoescaping should be active for the given template name. If no template name is given, returns *True*.

▶ *Changelog*

send_file_max_age_default

A **timedelta** which is used as default cache_timeout for the **send_file()** functions. The default is 12 hours.

This attribute can also be configured from the config with the `SEND_FILE_MAX_AGE_DEFAULT` configuration key. This configuration variable can also be set with an integer value used as seconds. Defaults to `timedelta(hours=12)`

send_static_file(*filename*)

Function used internally to send static files from the static folder to the browser.

▶ *Changelog*

session_cookie_name

The secure cookie uses this for the name of the session cookie.

This attribute can also be configured from the config with the `SESSION_COOKIE_NAME` configuration key. Defaults to `'session'`

session_interface = *<flask.sessions.SecureCookieSessionInterface object>*

the session interface to use. By default an instance of **SecureCookieSessionInterface** is used here.

▶ *Changelog*

shell_context_processor(*f*)

Registers a shell context processor function.

▶ *Changelog*

shell_context_processors = *None*

A list of shell context processor functions that should be run when a shell context is created.

▶ *Changelog*

should_ignore_error(*error*)

This is called to figure out if an error should be ignored or not as far as the teardown system is concerned. If this function returns `True` then the teardown handlers will not be passed the error.

▶ *Changelog*

static_folder

The absolute path to the configured static folder.

static_url_path

The URL prefix that the static route will be registered for.

teardown_appcontext(*f*)

Registers a function to be called when the application context ends. These functions are typically also called when the request context is popped.

Example:

```
ctx = app.app_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the app context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Since a request context typically also manages an application context it would also be called when you pop a request context.

When a teardown function was called because of an unhandled exception it will be passed an error object. If an **`errorhandler()`** is registered, it will handle the exception and the teardown will not receive it.

The return values of teardown functions are ignored.

▶ *Changelog*

`teardown_appcontext_funcs` = *None*

A list of functions that are called when the application context is destroyed. Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

▶ *Changelog*

`teardown_request`(*f*)

Register a function to be run at the end of each request, regardless of whether there was an exception or not. These functions are executed when the request context is popped, even if not an actual request was performed.

Example:

```
ctx = app.test_request_context()
ctx.push()
...
ctx.pop()
```

When `ctx.pop()` is executed in the above example, the teardown functions are called just before the request context moves from the stack of active contexts. This becomes relevant if you are using such constructs in tests.

Generally teardown functions must take every necessary step to avoid that they will fail. If they do execute code that might fail they will have to surround the execution of these code by try/except statements and log occurring errors.

When a teardown function was called because of an exception it will be passed an error object.

The return values of teardown functions are ignored.

Debug Note:

In debug mode Flask will not tear down a request on an exception immediately. Instead it will keep it alive so that the interactive debugger can still access it. This behavior can be controlled by the `PRESERVE_CONTEXT_ON_EXCEPTION` configuration variable.

---

teardown_request_funcs = *None*

A dictionary with lists of functions that are called after each request, even if an exception has occurred. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. These functions are not allowed to modify the request, and their return values are ignored. If an exception occurred while processing the request, it gets passed to each teardown_request function. To register a function here, use the **teardown_request()** decorator.

▶ *Changelog*

template_context_processors = *None*

A dictionary with list of functions that are called without argument to populate the template context. The key of the dictionary is the name of the blueprint this function is active for, `None` for all requests. Each returns a dictionary that the template context is updated with. To register a function here, use the **context_processor()** decorator.

template_filter(*name=None*)

A decorator that is used to register custom template filter. You can specify a name for the filter, otherwise the function name will be used. Example:

```
@app.template_filter()
def reverse(s):
    return s[::-1]
```

Parameters: **name** – the optional name of the filter, otherwise the function name will be

used.

`template_folder` = *None*

Location of the template files to be added to the template lookup. `None` if templates should not be added.

`template_global`(*name=None*)

A decorator that is used to register a custom template global function. You can specify a name for the global function, otherwise the function name will be used. Example:

```
@app.template_global()
def double(n):
    return 2 * n
```

▶ *Changelog*

Parameters: **name** – the optional name of the global function, otherwise the function name will be used.

`template_test`(*name=None*)

A decorator that is used to register custom template test. You can specify a name for the test, otherwise the function name will be used. Example:

```
@app.template_test()
def is_prime(n):
    if n == 2:
        return True
    for i in range(2, int(math.ceil(math.sqrt(n))) + 1):
        if n % i == 0:
            return False
    return True
```

▶ *Changelog*

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

`templates_auto_reload`

Reload templates when they are changed. Used by **`create_jinja_environment()`**.

This attribute can be configured with **`TEMPLATES_AUTO_RELOAD`**. If not set, it will be enabled in debug mode.

New in version 1.0: This property was added but the underlying config and behavior already existed.

▶ *Changelog*

`test_cli_runner`(***kwargs*)

Create a CLI runner for testing CLI commands. See Testing CLI Commands.

Returns an instance of **`test_cli_runner_class`**, by default **`FlaskCliRunner`**. The Flask app object is passed as the first argument.

New in version 1.0.

▶ *Changelog*

`test_cli_runner_class` = *None*

The **`CliRunner`** subclass, by default **`FlaskCliRunner`** that is used by **`test_cli_runner()`**. Its `__init__` method should take a Flask app object as the first argument.

New in version 1.0.

▶ *Changelog*

`test_client`(*use_cookies=True*, ***kwargs*)

Creates a test client for this application. For information about unit testing head over to Testing Flask Applications.

Note that if you are testing for assertions or exceptions in your application code, you must set `app.testing = True` in order for the exceptions to propagate to the test client. Otherwise, the exception will be handled by the application (not visible to the test client) and the only indication of an AssertionError or other exception will be a 500 status code response to the test client. See the **`testing`** attribute. For example:

```
app.testing = True
client = app.test_client()
```

The test client can be used in a `with` block to defer the closing down of the context until the end of the `with` block. This is useful if you want to access the context locals for testing:

```
with app.test_client() as c:
    rv = c.get('/?vodka=42')
    assert request.args['vodka'] == '42'
```

Additionally, you may pass optional keyword arguments that will then be passed to the application's **`test_client_class`** constructor. For example:

```
from flask.testing import FlaskClient

class CustomClient(FlaskClient):
    def __init__(self, *args, **kwargs):
        self._authentication = kwargs.pop("authentication")
        super(CustomClient,self).__init__( *args, **kwargs)

app.test_client_class = CustomClient
client = app.test_client(authentication='Basic ....')
```

See **FlaskClient** for more information.

▶ *Changelog*

test_client_class = *None*

the test client that is used with when *test_client* is used.

▶ *Changelog*

test_request_context(**args*, ***kwargs*)

Create a **RequestContext** for a WSGI environment created from the given values. This is mostly useful during testing, where you may want to run a function that uses request data without dispatching a full request.

See The Request Context.

Use a `with` block to push the context, which will make **request** point at the request for the created environment.

```
with test_request_context(...):
    generate_report()
```

When using the shell, it may be easier to push and pop the context manually to avoid indentation.

```
ctx = app.test_request_context(...)
ctx.push()
...
ctx.pop()
```

Takes the same arguments as Werkzeug's **EnvironBuilder**, with some defaults from the application. See the linked Werkzeug docs for most of the available arguments. Flask-specific behavior is listed here.

Parameters: • **path** – URL path being requested.

- **base_url** – Base URL where the app is being served, which `path` is relative to. If not given, built from `PREFERRED_URL_SCHEME`, `subdomain`, `SERVER_NAME`, and `APPLICATION_ROOT`.
- **subdomain** – Subdomain name to append to `SERVER_NAME`.
- **url_scheme** – Scheme to use instead of `PREFERRED_URL_SCHEME`.
- **data** – The request body, either as a string or a dict of form keys and values.
- **json** – If given, this is serialized as JSON and passed as `data`. Also defaults `content_type` to `application/json`.
- **args** – other positional arguments passed to `EnvironBuilder`.
- **kwargs** – other keyword arguments passed to `EnvironBuilder`.

`testing`

The testing flag. Set this to `True` to enable the test mode of Flask extensions (and in the future probably also Flask itself). For example this might activate test helpers that have an additional runtime cost which should not be enabled by default.

If this is enabled and PROPAGATE_EXCEPTIONS is not changed from the default it's implicitly enabled.

This attribute can also be configured from the config with the `TESTING` configuration key. Defaults to `False`.

`trap_http_exception`(*e*)

Checks if an HTTP exception should be trapped or not. By default this will return `False` for all exceptions except for a bad request key error if `TRAP_BAD_REQUEST_ERRORS` is set to `True`. It also returns `True` if `TRAP_HTTP_EXCEPTIONS` is set to `True`.

This is called for all HTTP exceptions raised by a view function. If it returns `True` for any exception the error handler for this exception is not called and it shows up as regular exception in the traceback. This is helpful for debugging implicitly raised HTTP exceptions.

Changed in version 1.0: Bad request errors are not trapped by default in debug mode.

▶ *Changelog*

`update_template_context`(*context*)

Update the template context with some commonly used variables. This injects request, session, config and g into the template context as well as everything template context processors want to inject. Note that the as of Flask 0.6, the original values in the context will not be overridden if a context processor decides to return a value with the same key.

Parameters: **context** – the context as a dictionary that is updated in place to add extra variables.

url_build_error_handlers = *None*

A list of functions that are called when **url_for()** raises a **BuildError**. Each function registered here is called with *error*, *endpoint* and *values*. If a function returns `None` or raises a **BuildError** the next function is tried.

▶ *Changelog*

url_default_functions = *None*

A dictionary with lists of functions that can be used as URL value preprocessors. The key `None` here is used for application wide callbacks, otherwise the key is the name of the blueprint. Each of these functions has the chance to modify the dictionary of URL values before they are used as the keyword arguments of the view function. For each function registered this one should also provide a **url_defaults()** function that adds the parameters automatically again that were removed that way.

▶ *Changelog*

url_defaults(*f*)

Callback function for URL defaults for all view functions of the application. It's called with the endpoint and values and should update the values passed in place.

url_map = *None*

The **Map** for this instance. You can use this to change the routing converters after the class was created but before any routes are connected. Example:

```
from werkzeug.routing import BaseConverter

class ListConverter(BaseConverter):
    def to_python(self, value):
        return value.split(',')
    def to_url(self, values):
        return ','.join(super(ListConverter, self).to_url(value)
                        for value in values)

app = Flask(__name__)
app.url_map.converters['list'] = ListConverter
```

url_rule_class

alias of **werkzeug.routing.Rule**

url_value_preprocessor(*f*)

Register a URL value preprocessor function for all view functions in the application. These functions will be called before the **before_request()** functions.

The function can modify the values captured from the matched url before they are passed to the view. For example, this can be used to pop a common language code value and place it in `g` rather than pass it to every view.

The function is passed the endpoint name and values dict. The return value is ignored.

`url_value_preprocessors` = *None*

A dictionary with lists of functions that are called before the **`before_request_funcs`** functions. The key of the dictionary is the name of the blueprint this function is active for, or `None` for all requests. To register a function, use **`url_value_preprocessor()`**.

▶ *Changelog*

`use_x_sendfile`

Enable this if you want to use the X-Sendfile feature. Keep in mind that the server has to support this. This only affects files sent with the **`send_file()`** method.

▶ *Changelog*

This attribute can also be configured from the config with the `USE_X_SENDFILE` configuration key. Defaults to `False`.

`view_functions` = *None*

A dictionary of all view functions registered. The keys will be function names which are also used to generate URLs and the values are the function objects themselves. To register a view function, use the **`route()`** decorator.

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

The actual WSGI application. This is not implemented in **`__call__()`** so that middlewares can be applied without losing a reference to the app object. Instead of doing this:

```
app = MyMiddleware(app)
```

It's a better idea to do this instead:

```
app.wsgi_app = MyMiddleware(app.wsgi_app)
```

Then you still have the original application object around and can continue to call methods on it.

▶ *Changelog*

Parameters: • **environ** – A WSGI environment.

- **start_response** – A callable accepting a status code, a list of headers, and an optional exception context to start the response.

Blueprint Objects

class `flask.`**`Blueprint`**(*name*, *import_name*, *static_folder=None*, *static_url_path=None*, *template_folder=None*, *url_prefix=None*, *subdomain=None*, *url_defaults=None*, *root_path=None*)

Represents a blueprint. A blueprint is an object that records functions that will be called with the **`BlueprintSetupState`** later to register functions or other things on the main application. See Modular Applications with Blueprints for more information.

▶ *Changelog*

`add_app_template_filter`(*f*, *name=None*)

Register a custom template filter, available application wide. Like **`Flask.add_template_filter()`** but for a blueprint. Works exactly like the **`app_template_filter()`** decorator.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

`add_app_template_global`(*f*, *name=None*)

Register a custom template global, available application wide. Like **`Flask.add_template_global()`** but for a blueprint. Works exactly like the **`app_template_global()`** decorator.

▶ *Changelog*

Parameters: **name** – the optional name of the global, otherwise the function name will be used.

`add_app_template_test`(*f*, *name=None*)

Register a custom template test, available application wide. Like **`Flask.add_template_test()`** but for a blueprint. Works exactly like the **`app_template_test()`** decorator.

▶ *Changelog*

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

`add_url_rule`(*rule*, *endpoint=None*, *view_func=None*, ***options*)

Like `Flask.add_url_rule()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

`after_app_request`(*f*)

Like `Flask.after_request()` but for a blueprint. Such a function is executed after each request, even if outside of the blueprint.

`after_request`(*f*)

Like `Flask.after_request()` but for a blueprint. This function is only executed after each request that is handled by a function of that blueprint.

`app_context_processor`(*f*)

Like `Flask.context_processor()` but for a blueprint. Such a function is executed each request, even if outside of the blueprint.

`app_errorhandler`(*code*)

Like `Flask.errorhandler()` but for a blueprint. This handler is used for all requests, even if outside of the blueprint.

`app_template_filter`(*name=None*)

Register a custom template filter, available application wide. Like `Flask.template_filter()` but for a blueprint.

Parameters: **name** – the optional name of the filter, otherwise the function name will be used.

`app_template_global`(*name=None*)

Register a custom template global, available application wide. Like `Flask.template_global()` but for a blueprint.

▶ *Changelog*

Parameters: **name** – the optional name of the global, otherwise the function name will be used.

`app_template_test`(*name=None*)

Register a custom template test, available application wide. Like `Flask.template_test()` but for a blueprint.

▶ *Changelog*

Parameters: **name** – the optional name of the test, otherwise the function name will be used.

app_url_defaults(*f*)

Same as **url_defaults()** but application wide.

app_url_value_preprocessor(*f*)

Same as **url_value_preprocessor()** but application wide.

before_app_first_request(*f*)

Like **Flask.before_first_request()**. Such a function is executed before the first request to the application.

before_app_request(*f*)

Like **Flask.before_request()**. Such a function is executed before each request, even if outside of a blueprint.

before_request(*f*)

Like **Flask.before_request()** but for a blueprint. This function is only executed before each request that is handled by a function of that blueprint.

context_processor(*f*)

Like **Flask.context_processor()** but for a blueprint. This function is only executed for requests handled by a blueprint.

endpoint(*endpoint*)

Like **Flask.endpoint()** but for a blueprint. This does not prefix the endpoint with the blueprint name, this has to be done explicitly by the user of this method. If the endpoint is prefixed with a . it will be registered to the current blueprint, otherwise it's an application independent endpoint.

errorhandler(*code_or_exception*)

Registers an error handler that becomes active for this blueprint only. Please be aware that routing does not happen local to a blueprint so an error handler for 404 usually is not handled by a blueprint unless it is caused inside a view function. Another special case is the 500 internal server error which is always looked up from the application.

Otherwise works as the **errorhandler()** decorator of the **Flask** object.

get_send_file_max_age(*filename*)

Provides default cache_timeout for the **send_file()** functions.

By default, this function returns `SEND_FILE_MAX_AGE_DEFAULT` from the configuration of **current_app**.

Static file functions such as **send_from_directory()** use this function, and **send_file()** calls this function on **current_app** when the given cache_timeout is None. If a cache_timeout is given in **send_file()**, that timeout is used; otherwise, this method is called.

This allows subclasses to change the behavior when sending files based on the filename. For example, to set the cache timeout for .js files to 60 seconds:

```
class MyFlask(flask.Flask):
    def get_send_file_max_age(self, name):
        if name.lower().endswith('.js'):
            return 60
        return flask.Flask.get_send_file_max_age(self, name)
```

▶ *Changelog*

has_static_folder

This is True if the package bound object's container has a folder for static files.

▶ *Changelog*

import_name = *None*

The name of the package or module that this app belongs to. Do not change this once it is set by the constructor.

jinja_loader

The Jinja loader for this package bound object.

▶ *Changelog*

json_decoder = *None*

Blueprint local JSON decoder class to use. Set to None to use the app's **json_decoder**.

json_encoder = *None*

Blueprint local JSON decoder class to use. Set to None to use the app's **json_encoder**.

make_setup_state(*app*, *options*, *first_registration=False*)

Creates an instance of **BlueprintSetupState()** object that is later passed to the register callback functions. Subclasses can override this to return a subclass of the setup state.

open_resource(*resource*, *mode='rb'*)

Opens a resource from the application's resource folder. To see how this works, consider the following folder structure:

```
/myapplication.py
/schema.sql
/static
    /style.css
/templates
    /layout.html
    /index.html
```

If you want to open the `schema.sql` file you would do the following:

```
with app.open_resource('schema.sql') as f:
    contents = f.read()
    do_something_with(contents)
```

Parameters:
- **resource** – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** – resource file opening mode, default is 'rb'.

record(*func*)

Registers a function that is called when the blueprint is registered on the application. This function is called with the state as argument as returned by the **make_setup_state()** method.

record_once(*func*)

Works like **record()** but wraps the function in another function that will ensure the function is only called once. If the blueprint is registered a second time on the application, the function passed is not called.

register(*app*, *options*, *first_registration=False*)

Called by **Flask.register_blueprint()** to register all views and callbacks registered on the blueprint with the application. Creates a **BlueprintSetupState** and calls each **record()** callback with it.

Parameters:
- **app** – The application this blueprint is being registered with.
- **options** – Keyword arguments forwarded from **register_blueprint()**.
- **first_registration** – Whether this is the first time this blueprint has been registered on the application.

register_error_handler(*code_or_exception*, *f*)

Non-decorator version of the **errorhandler()** error attach function, akin to the **register_error_handler()** application-wide function of the **Flask** object but for error handlers limited to this blueprint.

▶ *Changelog*

`root_path` = *None*

Absolute path to the package on the filesystem. Used to look up resources contained in the package.

`route`(*rule*, ***options*)

Like `Flask.route()` but for a blueprint. The endpoint for the `url_for()` function is prefixed with the name of the blueprint.

`send_static_file`(*filename*)

Function used internally to send static files from the static folder to the browser.

▶ *Changelog*

`static_folder`

The absolute path to the configured static folder.

`static_url_path`

The URL prefix that the static route will be registered for.

`teardown_app_request`(*f*)

Like `Flask.teardown_request()` but for a blueprint. Such a function is executed when tearing down each request, even if outside of the blueprint.

`teardown_request`(*f*)

Like `Flask.teardown_request()` but for a blueprint. This function is only executed when tearing down requests handled by a function of that blueprint. Teardown request functions are executed when the request context is popped, even when no actual request was performed.

`template_folder` = *None*

Location of the template files to be added to the template lookup. `None` if templates should not be added.

`url_defaults`(*f*)

Callback function for URL defaults for this blueprint. It's called with the endpoint and values and should update the values passed in place.

`url_value_preprocessor`(*f*)

Registers a function as URL value preprocessor for this blueprint. It's called before the view functions are called and can modify the url values provided.

Incoming Request Data

class `flask.`**`Request`**(*environ*, *populate_request=True*, *shallow=False*)

The request object used by default in Flask. Remembers the matched endpoint and view arguments.

It is what ends up as **`request`**. If you want to replace the request object used you can subclass this and set **`request_class`** to your subclass.

The request object is a **`Request`** subclass and provides all of the attributes Werkzeug defines plus a few Flask specific ones.

`environ`

The underlying WSGI environment.

`path`

`full_path`

`script_root`

`url`

`base_url`

`url_root`

Provides different ways to look at the current IRI. Imagine your application is listening on the following application root:

```
http://www.example.com/myapplication
```

And a user requests the following URI:

```
http://www.example.com/myapplication/%CF%80/page.html?x=y
```

In this case the values of the above mentioned attributes would be the following:

path	`u'/π/page.html'`
full_path	`u'/π/page.html?x=y'`
script_root	`u'/myapplication'`
base_url	`u'http://www.example.com/myapplication/π/page.html'`

url	`u'http://www.example.com/myapplication/π/page.html?x=y'`
url_root	`u'http://www.example.com/myapplication/'`

accept_charsets

List of charsets this client supports as **CharsetAccept** object.

accept_encodings

List of encodings this client accepts. Encodings in a HTTP term are compression encodings such as gzip. For charsets have a look at **accept_charset**.

accept_languages

List of languages this client accepts as **LanguageAccept** object.

accept_mimetypes

List of mimetypes this client supports as **MIMEAccept** object.

access_route

If a forwarded header exists this is a list of all ip addresses from the client ip to the last proxy server.

classmethod application(*f*)

Decorate a function as responder that accepts the request as first argument. This works like the **responder()** decorator but the function is passed the request object as first argument and the request object will be closed automatically:

```
@Request.application
def my_wsgi_app(request):
    return Response('Hello World!')
```

As of Werkzeug 0.14 HTTP exceptions are automatically caught and converted to responses instead of failing.

Parameters: **f** – the WSGI callable to decorate
Returns: a new WSGI callable

args

The parsed URL parameters (the part in the URL after the question mark).

By default an **ImmutableMultiDict** is returned from this function. This can be changed by setting **parameter_storage_class** to a different type. This might be necessary if the order of the form data is important.

authorization

The *Authorization* object in parsed form.

base_url

Like **url** but without the querystring See also: **trusted_hosts**.

blueprint

The name of the current blueprint

cache_control

A **RequestCacheControl** object for the incoming cache control headers.

close()

Closes associated resources of this request object. This closes all file handles explicitly. You can also use the request object in a with statement which will automatically close it.

▶ *Changelog*

content_encoding

The Content-Encoding entity-header field is used as a modifier to the media-type. When present, its value indicates what additional content codings have been applied to the entity-body, and thus what decoding mechanisms must be applied in order to obtain the media-type referenced by the Content-Type header field.

▶ *Changelog*

content_length

The Content-Length entity-header field indicates the size of the entity-body in bytes or, in the case of the HEAD method, the size of the entity-body that would have been sent had the request been a GET.

content_md5

The Content-MD5 entity-header field, as defined in RFC 1864, is an MD5 digest of the entity-body for the purpose of providing an end-to-end message integrity check (MIC) of the entity-body. (Note: a MIC is good for detecting accidental modification of the entity-body in transit, but is not proof against malicious attacks.)

▶ *Changelog*

content_type

The Content-Type entity-header field indicates the media type of the entity-body sent to the recipient or, in the case of the HEAD method, the media type that would have been sent had the request been a GET.

cookies

A **dict** with the contents of all cookies transmitted with the request.

data

Contains the incoming request data as string in case it came with a mimetype Werkzeug does not handle.

date

The Date general-header field represents the date and time at which the message was originated, having the same semantics as orig-date in RFC 822.

dict_storage_class

alias of **werkzeug.datastructures.ImmutableTypeConversionDict**

endpoint

The endpoint that matched the request. This in combination with **view_args** can be used to reconstruct the same or a modified URL. If an exception happened when matching, this will be `None`.

files

MultiDict object containing all uploaded files. Each key in **files** is the name from the `<input type="file" name="">`. Each value in **files** is a Werkzeug **FileStorage** object.

It basically behaves like a standard file object you know from Python, with the difference that it also has a **save()** function that can store the file on the filesystem.

Note that **files** will only contain data if the request method was POST, PUT or PATCH and the `<form>` that posted to the request had `enctype="multipart/form-data"`. It will be empty otherwise.

See the **MultiDict** / **FileStorage** documentation for more details about the used data structure.

form

The form parameters. By default an **ImmutableMultiDict** is returned from this function. This can be changed by setting **parameter_storage_class** to a different type. This might be necessary if the order of the form data is important.

Please keep in mind that file uploads will not end up here, but instead in the **files** attribute.

▶ *Changelog*

form_data_parser_class

alias of **werkzeug.formparser.FormDataParser**

classmethod **from_values**(**args*, ***kwargs*)

Create a new request object based on the values provided. If environ is given missing values are filled from there. This method is useful for small scripts when you need to simulate a request from an URL. Do not use this method for unittesting, there is a full featured client object (**Client**) that allows to create multipart requests, support for cookies etc.

This accepts the same options as the **EnvironBuilder**.

▶ *Changelog*

Returns: request object

full_path

Requested path as unicode, including the query string.

get_data(*cache=True*, *as_text=False*, *parse_form_data=False*)

This reads the buffered incoming data from the client into one bytestring. By default this is cached but that behavior can be changed by setting *cache* to *False*.

Usually it's a bad idea to call this method without checking the content length first as a client could send dozens of megabytes or more to cause memory problems on the server.

Note that if the form data was already parsed this method will not return anything as form data parsing does not cache the data like this method does. To implicitly invoke form data parsing function set *parse_form_data* to *True*. When this is done the return value of this method will be an empty string if the form parser handles the data. This generally is not necessary as if the whole data is cached (which is the default) the form parser will used the cached data to parse the form data. Please be generally aware of checking the content length first in any case before calling this method to avoid exhausting server memory.

If *as_text* is set to *True* the return value will be a decoded unicode string.

▶ *Changelog*

get_json(*force=False*, *silent=False*, *cache=True*)

Parse and return the data as JSON. If the mimetype does not indicate JSON (*application/json*, see **is_json()**), this returns `None` unless `force` is true. If parsing fails, **on_json_loading_failed()** is called and its return value is used as the return value.

Parameters:
- **force** – Ignore the mimetype and always try to parse JSON.
- **silent** – Silence parsing errors and return `None` instead.
- **cache** – Store the parsed JSON to return for subsequent calls.

headers

The headers from the WSGI environ as immutable **EnvironHeaders**.

host

Just the host including the port if available. See also: **trusted_hosts**.

host_url

Just the host with scheme as IRI. See also: **trusted_hosts**.

if_match

An object containing all the etags in the *If-Match* header.

Return type: **ETags**

if_modified_since

The parsed *If-Modified-Since* header as datetime object.

if_none_match

An object containing all the etags in the *If-None-Match* header.

Return type: **ETags**

if_range

The parsed *If-Range* header.

▶ *Changelog*

Return type: **IfRange**

if_unmodified_since

The parsed *If-Unmodified-Since* header as datetime object.

is_json

Check if the mimetype indicates JSON data, either *application/json* or *application/*+json*.

▶ *Changelog*

is_multiprocess

boolean that is *True* if the application is served by a WSGI server that spawns multiple processes.

is_multithread

boolean that is *True* if the application is served by a multithreaded WSGI server.

is_run_once

boolean that is *True* if the application will be executed only once in a process lifetime. This is the case for CGI for example, but it's not guaranteed that the execution only happens one time.

is_secure

True if the request is secure.

is_xhr

True if the request was triggered via a JavaScript XMLHttpRequest. This only works with libraries that support the `X-Requested-With` header and set it to "XMLHttpRequest". Libraries that do that are prototype, jQuery and Mochikit and probably some more.

Deprecated since version 0.13: `X-Requested-With` is not standard and is unreliable.

▶ *Changelog*

json

This will contain the parsed JSON data if the mimetype indicates JSON (*application/json*, see **`is_json()`**), otherwise it will be `None`.

list_storage_class

alias of **`werkzeug.datastructures.ImmutableList`**

make_form_data_parser()

Creates the form data parser. Instantiates the **`form_data_parser_class`** with some parameters.

▶ *Changelog*

max_content_length

Read-only view of the `MAX_CONTENT_LENGTH` config key.

max_forwards

The Max-Forwards request-header field provides a mechanism with the TRACE and OPTIONS methods to limit the number of proxies or gateways that can forward the request to the next inbound server.

method

The request method. (For example `'GET'` or `'POST'`).

mimetype

Like **content_type**, but without parameters (eg, without charset, type etc.) and always lowercase. For example if the content type is `text/HTML; charset=utf-8` the mimetype would be `'text/html'`.

mimetype_params

The mimetype parameters as dict. For example if the content type is `text/html; charset=utf-8` the params would be `{'charset': 'utf-8'}`.

on_json_loading_failed(*e*)

Called if **get_json()** parsing fails and isn't silenced. If this method returns a value, it is used as the return value for **get_json()**. The default implementation raises a **BadRequest** exception.

▶ *Changelog*

parameter_storage_class

alias of **werkzeug.datastructures.ImmutableMultiDict**

path

Requested path as unicode. This works a bit like the regular path info in the WSGI environment but will always include a leading slash, even if the URL root is accessed.

pragma

The Pragma general-header field is used to include implementation-specific directives that might apply to any recipient along the request/response chain. All pragma directives specify optional behavior from the viewpoint of the protocol; however, some systems MAY require that behavior be consistent with the directives.

query_string

The URL parameters as raw bytestring.

range

The parsed *Range* header.

▶ *Changelog*

Return type: **Range**

referrer

The Referer[sic] request-header field allows the client to specify, for the server's benefit, the address (URI) of the resource from which the Request-URI was obtained (the "referrer", although the header field is misspelled).

remote_addr

The remote address of the client.

remote_user

If the server supports user authentication, and the script is protected, this attribute contains the username the user has authenticated as.

routing_exception = *None*

If matching the URL failed, this is the exception that will be raised / was raised as part of the request handling. This is usually a **NotFound** exception or something similar.

scheme

URL scheme (http or https).

▶ *Changelog*

script_root

The root path of the script without the trailing slash.

stream

If the incoming form data was not encoded with a known mimetype the data is stored unmodified in this stream for consumption. Most of the time it is a better idea to use **data** which will give you that data as a string. The stream only returns the data once.

Unlike **input_stream** this stream is properly guarded that you can't accidentally read past the length of the input. Werkzeug will internally always refer to this stream to read data which makes it possible to wrap this object with a stream that does filtering.

▶ *Changelog*

url

The reconstructed current URL as IRI. See also: **trusted_hosts**.

url_charset

The charset that is assumed for URLs. Defaults to the value of **charset**.

▶ *Changelog*

url_root

The full URL root (with hostname), this is the application root as IRI. See also: **trusted_hosts**.

`url_rule` = *None*

The internal URL rule that matched the request. This can be useful to inspect which methods are allowed for the URL from a before/after handler (`request.url_rule.methods`) etc. Though if the request's method was invalid for the URL rule, the valid list is available in `routing_exception.valid_methods` instead (an attribute of the Werkzeug exception **`MethodNotAllowed`**) because the request was never internally bound.

▶ *Changelog*

`user_agent`

The current user agent.

`values`

A **`werkzeug.datastructures.CombinedMultiDict`** that combines **`args`** and **`form`**.

`view_args` = *None*

A dict of view arguments that matched the request. If an exception happened when matching, this will be `None`.

`want_form_data_parsed`

Returns True if the request method carries content. As of Werkzeug 0.9 this will be the case if a content type is transmitted.

▶ *Changelog*

`flask.request`

To access incoming request data, you can use the global *request* object. Flask parses incoming request data for you and gives you access to it through that global object. Internally Flask makes sure that you always get the correct data for the active thread if you are in a multithreaded environment.

This is a proxy. See Notes On Proxies for more information.

The request object is an instance of a **`Request`** subclass and provides all of the attributes Werkzeug defines. This just shows a quick overview of the most important ones.

Response Objects

class `flask.`**`Response`**(*response=None*, *status=None*, *headers=None*, *mimetype=None*, *content_type=None*, *direct_passthrough=False*)

The response object that is used by default in Flask. Works like the response object from Werkzeug but is set to have an HTML mimetype by default. Quite often you don't have to create

this object yourself because **make_response()** will take care of that for you.

If you want to replace the response object used you can subclass this and set **response_class** to your subclass.

Changed in version 1.0: JSON support is added to the response, like the request. This is useful when testing to get the test client response data as JSON.

Changed in version 1.0: Added **max_cookie_size**.

▼ *Changelog*

headers

A **Headers** object representing the response headers.

status

A string with a response status.

status_code

The response status as integer.

data

A descriptor that calls **get_data()** and **set_data()**. This should not be used and will eventually get deprecated.

get_json(*force=False*, *silent=False*, *cache=True*)

Parse and return the data as JSON. If the mimetype does not indicate JSON (*application/json*, see **is_json()**), this returns `None` unless `force` is true. If parsing fails, **on_json_loading_failed()** is called and its return value is used as the return value.

Parameters:
- **force** – Ignore the mimetype and always try to parse JSON.
- **silent** – Silence parsing errors and return `None` instead.
- **cache** – Store the parsed JSON to return for subsequent calls.

is_json

Check if the mimetype indicates JSON data, either *application/json* or *application/*+json*.

▶ *Changelog*

max_cookie_size

Read-only view of the **MAX_COOKIE_SIZE** config key.

See **max_cookie_size** in Werkzeug's docs.

`mimetype`

The mimetype (content type without charset etc.)

`set_cookie`(*key*, *value=''*, *max_age=None*, *expires=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *samesite=None*)

Sets a cookie. The parameters are the same as in the cookie *Morsel* object in the Python standard library but it accepts unicode data, too.

A warning is raised if the size of the cookie header exceeds `max_cookie_size`, but the header will still be set.

Parameters:
- **key** – the key (name) of the cookie to be set.
- **value** – the value of the cookie.
- **max_age** – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client's browser session.
- **expires** – should be a *datetime* object or UNIX timestamp.
- **path** – limits the cookie to a given path, per default it will span the whole domain.
- **domain** – if you want to set a cross-domain cookie. For example, `domain=".example.com"` will set a cookie that is readable by the domain `www.example.com`, `foo.example.com` etc. Otherwise, a cookie will only be readable by the domain that set it.
- **secure** – If *True*, the cookie will only be available via HTTPS
- **httponly** – disallow JavaScript to access the cookie. This is an extension to the cookie standard and probably not supported by all browsers.
- **samesite** – Limits the scope of the cookie such that it will only be attached to requests if those requests are "same-site".

Sessions

If you have set `Flask.secret_key` (or configured it from `SECRET_KEY`) you can use sessions in Flask applications. A session makes it possible to remember information from one request to another. The way Flask does this is by using a signed cookie. The user can look at the session contents, but can't modify it unless they know the secret key, so make sure to set that to something complex and unguessable.

To access the current session you can use the `session` object:

class `flask.`**`session`**

The session object works pretty much like an ordinary dict, with the difference that it keeps track on modifications.

This is a proxy. See Notes On Proxies for more information.

The following attributes are interesting:

new

`True` if the session is new, `False` otherwise.

modified

`True` if the session object detected a modification. Be advised that modifications on mutable structures are not picked up automatically, in that situation you have to explicitly set the attribute to `True` yourself. Here an example:

```
# this change is not picked up because a mutable object (here
# a list) is changed.
session['objects'].append(42)
# so mark it as modified yourself
session.modified = True
```

permanent

If set to `True` the session lives for **`permanent_session_lifetime`** seconds. The default is 31 days. If set to `False` (which is the default) the session will be deleted when the user closes the browser.

Session Interface

▶ *Changelog*

The session interface provides a simple way to replace the session implementation that Flask is using.

class `flask.sessions.`**`SessionInterface`**

The basic interface you have to implement in order to replace the default session interface which uses werkzeug's securecookie implementation. The only methods you have to implement are **`open_session()`** and **`save_session()`**, the others have useful defaults which you don't need to change.

The session object returned by the **`open_session()`** method has to provide a dictionary like interface plus the properties and methods from the **`SessionMixin`**. We recommend just subclassing a dict and adding that mixin:

```
class Session(dict, SessionMixin):
    pass
```

If **open_session()** returns None Flask will call into **make_null_session()** to create a session that acts as replacement if the session support cannot work because some requirement is not fulfilled. The default **NullSession** class that is created will complain that the secret key was not set.

To replace the session interface on an application all you have to do is to assign **flask.Flask.session_interface**:

```
app = Flask(__name__)
app.session_interface = MySessionInterface()
```

▶ *Changelog*

get_cookie_domain(*app*)

Returns the domain that should be set for the session cookie.

Uses SESSION_COOKIE_DOMAIN if it is configured, otherwise falls back to detecting the domain based on SERVER_NAME.

Once detected (or if not set at all), SESSION_COOKIE_DOMAIN is updated to avoid re-running the logic.

get_cookie_httponly(*app*)

Returns True if the session cookie should be httponly. This currently just returns the value of the SESSION_COOKIE_HTTPONLY config var.

get_cookie_path(*app*)

Returns the path for which the cookie should be valid. The default implementation uses the value from the SESSION_COOKIE_PATH config var if it's set, and falls back to APPLICATION_ROOT or uses / if it's None.

get_cookie_samesite(*app*)

Return 'Strict' or 'Lax' if the cookie should use the SameSite attribute. This currently just returns the value of the **SESSION_COOKIE_SAMESITE** setting.

get_cookie_secure(*app*)

Returns True if the cookie should be secure. This currently just returns the value of the SESSION_COOKIE_SECURE setting.

get_expiration_time(*app*, *session*)

A helper method that returns an expiration date for the session or None if the session is linked to the browser session. The default implementation returns now + the permanent session

lifetime configured on the application.

is_null_session(*obj*)

Checks if a given object is a null session. Null sessions are not asked to be saved.

This checks if the object is an instance of **null_session_class** by default.

make_null_session(*app*)

Creates a null session which acts as a replacement object if the real session support could not be loaded due to a configuration error. This mainly aids the user experience because the job of the null session is to still support lookup without complaining but modifications are answered with a helpful error message of what failed.

This creates an instance of **null_session_class** by default.

null_session_class

make_null_session() will look here for the class that should be created when a null session is requested. Likewise the **is_null_session()** method will perform a typecheck against this type.

alias of **NullSession**

open_session(*app*, *request*)

This method has to be implemented and must either return `None` in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on **SessionMixin**.

pickle_based = *False*

A flag that indicates if the session interface is pickle based. This can be used by Flask extensions to make a decision in regards to how to deal with the session object.

▶ *Changelog*

save_session(*app*, *session*, *response*)

This is called for actual sessions returned by **open_session()** at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.

should_set_cookie(*app*, *session*)

Used by session backends to determine if a `Set-Cookie` header should be set for this session cookie for this response. If the session has been modified, the cookie is set. If the session is permanent and the `SESSION_REFRESH_EACH_REQUEST` config is true, the cookie is always set.

This check is usually skipped if the session was deleted.

▶ *Changelog*

class flask.sessions.**SecureCookieSessionInterface**

The default session interface that stores sessions in signed cookies through the **itsdangerous** module.

static **digest_method**()

the hash function to use for the signature. The default is sha1

key_derivation = *'hmac'*

the name of the itsdangerous supported key derivation. The default is hmac.

open_session(*app*, *request*)

This method has to be implemented and must either return `None` in case the loading failed because of a configuration error or an instance of a session object which implements a dictionary like interface + the methods and attributes on **SessionMixin**.

salt = *'cookie-session'*

the salt that should be applied on top of the secret key for the signing of cookie based sessions.

save_session(*app*, *session*, *response*)

This is called for actual sessions returned by **open_session()** at the end of the request. This is still called during a request context so if you absolutely need access to the request you can do that.

serializer = *<flask.json.tag.TaggedJSONSerializer object>*

A python serializer for the payload. The default is a compact JSON derived serializer with support for some extra Python types such as datetime objects or tuples.

session_class

alias of **SecureCookieSession**

class flask.sessions.**SecureCookieSession**(*initial=None*)

Base class for sessions based on signed cookies.

This session backend will set the **modified** and **accessed** attributes. It cannot reliably track whether a session is new (vs. empty), so **new** remains hard coded to `False`.

accessed = *False*

header, which allows caching proxies to cache different pages for different users.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

modified = *False*

When data is changed, this is set to `True`. Only the session dictionary itself is tracked; if the session contains mutable data (for example a nested dict) then this must be set to `True` manually when modifying that data. The session cookie will only be written to the response if this is `True`.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

class `flask.sessions.`**NullSession**(*initial=None*)

Class used to generate nicer error messages if sessions are not available. Will still allow read-only access to the empty session but fail on setting.

class `flask.sessions.`**SessionMixin**

Expands a basic dictionary with session attributes.

accessed = *True*

Some implementations can detect when session data is read or written and set this when that happens. The mixin default is hard coded to `True`.

modified = *True*

Some implementations can detect changes to the session and set this when that happens. The mixin default is hard coded to `True`.

permanent

This reflects the `'_permanent'` key in the dict.

Notice:

The `PERMANENT_SESSION_LIFETIME` config key can also be an integer starting with Flask 0.8. Either catch this down yourself or use the **`permanent_session_lifetime`** attribute on the app which converts the result to an integer automatically.

Test Client

class `flask.testing.`**FlaskClient**(**args*, ***kwargs*)

Works like a regular Werkzeug test client but has some knowledge about how Flask works to defer the cleanup of the request context stack to the end of a `with` body when used in a `with` statement.

For general information about how to use this class refer to **werkzeug.test.Client**.

▶ *Changelog*

Basic usage is outlined in the Testing Flask Applications chapter.

open(**args*, ***kwargs*)

Takes the same arguments as the **EnvironBuilder** class with some additions: You can provide a **EnvironBuilder** or a WSGI environment as only argument instead of the **EnvironBuilder** arguments and two optional keyword arguments (*as_tuple*, *buffered*) that change the type of the return value or the way the application is executed.

▶ *Changelog*

Additional parameters:

Parameters:
- **as_tuple** – Returns a tuple in the form `(environ, result)`
- **buffered** – Set this to True to buffer the application run. This will automatically close the application for you as well.
- **follow_redirects** – Set this to True if the *Client* should follow HTTP redirects.

session_transaction(**args*, ***kwargs*)

When used in combination with a **with** statement this opens a session transaction. This can be used to modify the session that the test client uses. Once the **with** block is left the session is stored back.

```
with client.session_transaction() as session:
    session['value'] = 42
```

Internally this is implemented by going through a temporary test request context and since session handling could depend on request variables this function accepts the same arguments as **test_request_context()** which are directly passed through.

Test CLI Runner

class flask.testing.**FlaskCliRunner**(*app*, ***kwargs*)

A **CliRunner** for testing a Flask app's CLI commands. Typically created using **test_cli_runner()**. See Testing CLI Commands.

invoke(*cli=None*, *args=None*, ***kwargs*)

Invokes a CLI command in an isolated environment. See **CliRunner.invoke** for full method documentation. See Testing CLI Commands for examples.

If the `obj` argument is not given, passes an instance of **`ScriptInfo`** that knows how to load the Flask app being tested.

Parameters:
- **cli** – Command object to invoke. Default is the app's **`cli`** group.
- **args** – List of strings to invoke the command with.

Returns: a **`Result`** object.

Application Globals

To share data that is valid for one request only from one function to another, a global variable is not good enough because it would break in threaded environments. Flask provides you with a special object that ensures it is only valid for the active request and that will return different values for each request. In a nutshell: it does the right thing, like it does for **`request`** and **`session`**.

flask.**g**

A namespace object that can store data during an application context. This is an instance of **`Flask.app_ctx_globals_class`**, which defaults to **`ctx._AppCtxGlobals`**.

This is a good place to store resources during a request. During testing, you can use the Faking Resources and Context pattern to pre-configure such resources.

This is a proxy. See Notes On Proxies for more information.

▶ *Changelog*

class flask.ctx.**_AppCtxGlobals**

A plain object. Used as a namespace for storing data during an application context.

Creating an app context automatically creates this object, which is made available as the **`g`** proxy.

'key' in g

Check whether an attribute is present.

▶ *Changelog*

iter(g)

Return an iterator over the attribute names.

▶ *Changelog*

get(*name*, *default=None*)

Get an attribute by name, or a default value. Like **`dict.get()`**.

Parameters:
- **name** – Name of attribute to get.

- **default** – Value to return if the attribute is not present.

▶ *Changelog*

pop(*name*, *default=<object object>*)

Get and remove an attribute by name. Like **dict.pop()**.

Parameters:
- **name** – Name of attribute to pop.
- **default** – Value to return if the attribute is not present, instead of raise a `KeyError`.

▶ *Changelog*

setdefault(*name*, *default=None*)

Get the value of an attribute if it is present, otherwise set and return a default value. Like **dict.setdefault()**.

Parameters: **name** – Name of attribute to get.
Param: default: Value to set and return if the attribute is not present.

▶ *Changelog*

Useful Functions and Classes

flask.**current_app**

A proxy to the application handling the current request. This is useful to access the application without needing to import it, or if it can't be imported, such as when using the application factory pattern or in blueprints and extensions.

This is only available when an application context is pushed. This happens automatically during requests and CLI commands. It can be controlled manually with **app_context()**.

This is a proxy. See Notes On Proxies for more information.

flask.**has_request_context**()

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
```

▶ *Changelog*

flask.**copy_current_request_context**(*f*)

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'
```

▶ *Changelog*

flask.**has_app_context**()

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

▶ *Changelog*

flask.**url_for**(*endpoint*, ***values*)

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.

▶ *Changelog*

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.

`flask.`**`abort`**(*status*, **args*, ***kwargs*)

Raises an **`HTTPException`** for the given status code or WSGI application:

```
abort(404)  # 404 Not Found
abort(Response('Hello World'))
```

Can be passed a WSGI application or a status code. If a status code is given it's looked up in the list of exceptions and will raise that exception, if passed a WSGI application it will wrap it in a proxy WSGI exception and raise that:

```
abort(404)
abort(Response('Hello World'))
```

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

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.

▶ *Changelog*

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.`**`make_response`**(**args*)

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.

▶ *Changelog*

flask.**after_this_request**(*f*)

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.

▶ *Changelog*

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

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.

ETags will also be attached automatically if a *filename* is provided. You can turn this off by setting *add_etags=False*.

If *conditional=True* and *filename* is provided, this method will try to upgrade the response stream to support range requests. This will allow the request to be answered with partial content response.

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

Changed in version 1.0: UTF-8 filenames, as specified in RFC 2231, are supported.

▶ *Changelog*

Parameters:

- **filename_or_fp** – the filename of the file to send. 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. If a file path is given, auto detection happens as fallback, otherwise an error will be raised.
- **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`**.
- **last_modified** – set the `Last-Modified` header to this value, a **`datetime`** or timestamp. If a file was passed, this overrides its mtime.

`flask.`**`send_from_directory`**(*directory*, *filename*, ***options*)

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.

▶ *Changelog*

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.safe_join(*directory*, **pathnames*)

Safely join *directory* and zero or more untrusted *pathnames* components.

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 trusted base directory.
- **pathnames** – the untrusted pathnames relative to that directory.

Raises: **NotFound** if one or more passed paths fall out of its boundaries.

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.

class flask.Markup

Marks a string as being safe for inclusion in HTML/XML output without needing to be escaped. This implements the *__html__* interface a couple of frameworks and web applications use. **Markup**

is a direct subclass of *unicode* and provides all the methods of *unicode* just that it escapes arguments passed and always returns *Markup*.

The *escape* function returns markup objects so that double escaping can't happen.

The constructor of the **Markup** class can be used for three different things: When passed an unicode object it's assumed to be safe, when passed an object with an HTML representation (has an *__html__* method) that representation is used, otherwise the object passed is converted into a unicode string and then assumed to be safe:

```
>>> Markup("Hello <em>World</em>!")
Markup(u'Hello <em>World</em>!')
>>> class Foo(object):
...  def __html__(self):
...   return '<a href="#">foo</a>'
...
>>> Markup(Foo())
Markup(u'<a href="#">foo</a>')
```

If you want object passed being always treated as unsafe you can use the **escape()** classmethod to create a **Markup** object:

```
>>> Markup.escape("Hello <em>World</em>!")
Markup(u'Hello &lt;em&gt;World&lt;/em&gt;!')
```

Operations on a markup string are markup aware which means that all arguments are passed through the **escape()** function:

```
>>> em = Markup("<em>%s</em>")
>>> em % "foo & bar"
Markup(u'<em>foo &amp; bar</em>')
>>> strong = Markup("<strong>%(text)s</strong>")
>>> strong % {'text': '<blink>hacker here</blink>'}
Markup(u'<strong>&lt;blink&gt;hacker here&lt;/blink&gt;</strong>')
>>> Markup("<em>Hello</em> ") + "<foo>"
Markup(u'<em>Hello</em> &lt;foo&gt;')
```

classmethod **escape**(*s*)

Escape the string. Works like **escape()** with the difference that for subclasses of **Markup** this function would return the correct subclass.

striptags()

Unescape markup into an text_type string and strip all tags. This also resolves known HTML4 and XHTML entities. Whitespace is normalized to one:

```
>>> Markup("Main &raquo;  <em>About</em>").striptags()
u'Main \xbb About'
```

unescape()

Unescape markup again into an text_type string. This also resolves known HTML4 and XHTML entities:

```
>>> Markup("Main &raquo; <em>About</em>").unescape()
u'Main \xbb <em>About</em>'
```

Message Flashing

flask.**flash**(*message*, *category='message'*)

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()**.

▶ *Changelog*

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**(*with_categories=False*, *category_filter=[]*)

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.

▶ *Changelog*

Parameters:
- **with_categories** – set to `True` to also receive categories.

- **category_filter** – whitelist of categories to limit return values

JSON Support

Flask uses `simplejson` for the JSON implementation. Since simplejson is provided by both the standard library as well as extension, Flask will try simplejson first and then fall back to the stdlib json module. On top of that it will delegate access to the current application's JSON encoders and decoders for easier customization.

So for starters instead of doing:

```
try:
    import simplejson as json
except ImportError:
    import json
```

You can instead just do this:

```
from flask import json
```

For usage examples, read the **`json`** documentation in the standard library. The following extensions are by default applied to the stdlib's JSON module:

1. `datetime` objects are serialized as **RFC 822** strings.
2. Any object with an `__html__` method (like **`Markup`**) will have that method called and then the return value is serialized as string.

The **`htmlsafe_dumps()`** function of this json module is also available as filter called `|tojson` in Jinja2. Note that inside `script` tags no escaping must take place, so make sure to disable escaping with `|safe` if you intend to use it inside `script` tags unless you are using Flask 0.10 which implies that:

```
<script type=text/javascript>
    doSomethingWith({{ user.username|tojson|safe }});
</script>
```

Auto-Sort JSON Keys:

The configuration variable `JSON_SORT_KEYS` (Configuration Handling) can be set to false to stop Flask from auto-sorting keys. By default sorting is enabled and outside of the app context sorting is turned on.

Notice that disabling key sorting can cause issues when using content based HTTP caches and Python's hash randomization feature.

flask.json.**jsonify**(**args*, ***kwargs*)

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
}
```

▶ *Changelog*

This function's response will be pretty printed if the `JSONIFY_PRETTYPRINT_REGULAR` config parameter is set to True or the Flask app is running in debug mode. Compressed (not pretty) formatting currently means no indents and no spaces after separators.

▶ *Changelog*

flask.json.**dumps**(*obj*, ***kwargs*)

Serialize `obj` to a JSON formatted `str` by using the application's configured encoder (**json_encoder**) if there is an application on the stack.

This function can return `unicode` strings or ascii-only bytestrings by default which coerce into unicode strings automatically. That behavior by default is controlled by the `JSON_AS_ASCII` configuration variable and can be overridden by the simplejson `ensure_ascii` parameter.

`flask.json.`**`dump`**(*obj*, *fp*, ***kwargs*)

Like **`dumps()`** but writes into a file object.

`flask.json.`**`loads`**(*s*, ***kwargs*)

Unserialize a JSON object from a string `s` by using the application's configured decoder (**`json_decoder`**) if there is an application on the stack.

`flask.json.`**`load`**(*fp*, ***kwargs*)

Like **`loads()`** but reads from a file object.

class `flask.json.`**`JSONEncoder`**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

The default Flask JSON encoder. This one extends the default simplejson encoder by also supporting `datetime` objects, `UUID` as well as `Markup` objects which are serialized as RFC 822 datetime strings (same as the HTTP date format). In order to support more data types override the **`default()`** method.

`default`(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a **`TypeError`**).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    return JSONEncoder.default(self, o)
```

class `flask.json.`**`JSONDecoder`**(**, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, strict=True, object_pairs_hook=None*)

The default JSON decoder. This one does not change the behavior from the default simplejson decoder. Consult the **`json`** documentation for more information. This decoder is not only used for the load functions of this module but also **`Request`**.

Tagged JSON

A compact representation for lossless serialization of non-standard JSON types. **`SecureCookieSessionInterface`** uses this to serialize the session data, but it may be useful in other places. It can be extended to support other types.

class flask.json.tag.**TaggedJSONSerializer**

Serializer that uses a tag system to compactly represent objects that are not JSON types. Passed as the intermediate serializer to **itsdangerous.Serializer**.

The following extra types are supported:

- **dict**
- **tuple**
- **bytes**
- **Markup**
- **UUID**
- **datetime**

default_tags = *[<class 'flask.json.tag.TagDict'>, <class 'flask.json.tag.PassDict'>, <class 'flask.json.tag.TagTuple'>, <class 'flask.json.tag.PassList'>, <class 'flask.json.tag.TagBytes'>, <class 'flask.json.tag.TagMarkup'>, <class 'flask.json.tag.TagUUID'>, <class 'flask.json.tag.TagDateTime'>]*

Tag classes to bind when creating the serializer. Other tags can be added later using **register()**.

dumps(*value*)

Tag the value and dump it to a compact JSON string.

loads(*value*)

Load data from a JSON string and deserialized any tagged objects.

register(*tag_class*, *force=False*, *index=None*)

Register a new tag with this serializer.

Parameters:
- **tag_class** – tag class to register. Will be instantiated with this serializer instance.
- **force** – overwrite an existing tag. If false (default), a **KeyError** is raised.
- **index** – index to insert the new tag in the tag order. Useful when the new tag is a special case of an existing tag. If `None` (default), the tag is appended to the end of the order.

Raises: **KeyError** – if the tag key is already registered and `force` is not true.

tag(*value*)

Convert a value to a tagged representation if necessary.

untag(*value*)

Convert a tagged representation back to the original type.

class flask.json.tag.**JSONTag**(*serializer*)

Base class for defining type tags for **TaggedJSONSerializer**.

check(*value*)

Check if the given value should be tagged by this tag.

key = *None*

The tag to mark the serialized object with. If `None`, this tag is only used as an intermediate step during tagging.

tag(*value*)

Convert the value to a valid JSON type and add the tag structure around it.

to_json(*value*)

Convert the Python object to an object that is a valid JSON type. The tag will be added later.

to_python(*value*)

Convert the JSON representation back to the correct type. The tag will already be removed.

Let's seen an example that adds support for **OrderedDict**. Dicts don't have an order in Python or JSON, so to handle this we will dump the items as a list of `[key, value]` pairs. Subclass **JSONTag** and give it the new key `' od'` to identify the type. The session serializer processes dicts first, so insert the new tag at the front of the order since `OrderedDict` must be processed before `dict`.

```
from flask.json.tag import JSONTag

class TagOrderedDict(JSONTag):
    __slots__ = ('serializer',)
    key = ' od'

    def check(self, value):
        return isinstance(value, OrderedDict)

    def to_json(self, value):
        return [[k, self.serializer.tag(v)] for k, v in iteritems(value)]

    def to_python(self, value):
        return OrderedDict(value)

app.session_interface.serializer.register(TagOrderedDict, index=0)
```

Template Rendering

flask.**render_template**(*template_name_or_list*, ***context*)

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**(*source*, ***context*)

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.**get_template_attribute**(*template_name*, *attribute*)

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')
```

▶ *Changelog*

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

Configuration

class flask.**Config**(*root_path*, *defaults=None*)

Works exactly like a dict but provides ways to fill it from files or special dictionaries. There are two common patterns to populate the config.

Either you can fill the config from a config file:

```
app.config.from_pyfile('yourconfig.cfg')
```

Or alternatively you can define the configuration options in the module that calls **from_object()** or provide an import path to a module that should be loaded. It is also possible to tell it to use the

same module and with that provide the configuration values just before the call:

```
DEBUG = True
SECRET_KEY = 'development key'
app.config.from_object(__name__)
```

In both cases (loading from any Python file or loading from modules), only uppercase keys are added to the config. This makes it possible to use lowercase values in the config file for temporary values that are not added to the config or to define the config keys in the same file that implements the application.

Probably the most interesting way to load configurations is from an environment variable pointing to a file:

```
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

In this case before launching the application you have to set this environment variable to the file you want to use. On Linux and OS X use the export statement:

```
export YOURAPPLICATION_SETTINGS='/path/to/config/file'
```

On windows use *set* instead.

Parameters:
- **root_path** – path to which files are read relative from. When the config object is created by the application, this is the application's **root_path**.
- **defaults** – an optional dictionary of default values

from_envvar(*variable_name*, *silent=False*)

Loads a configuration from an environment variable pointing to a configuration file. This is basically just a shortcut with nicer error messages for this line of code:

```
app.config.from_pyfile(os.environ['YOURAPPLICATION_SETTINGS'])
```

Parameters:
- **variable_name** – name of the environment variable
- **silent** – set to `True` if you want silent failure for missing files.

Returns: bool. `True` if able to load config, `False` otherwise.

from_json(*filename*, *silent=False*)

Updates the values in the config from a JSON file. This function behaves as if the JSON object was a dictionary and passed to the **from_mapping()** function.

Parameters:
- **filename** – the filename of the JSON file. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

▶ *Changelog*

`from_mapping`(**mapping*, ***kwargs*)

Updates the config like `update()` ignoring items with non-upper keys.

▶ *Changelog*

`from_object`(*obj*)

Updates the values from the given object. An object can be of one of the following two types:

- a string: in this case the object with that name will be imported
- an actual object reference: that object is used directly

Objects are usually either modules or classes. `from_object()` loads only the uppercase attributes of the module/class. A `dict` object will not work with `from_object()` because the keys of a `dict` are not attributes of the `dict` class.

Example of module-based configuration:

```
app.config.from_object('yourapplication.default_config')
from yourapplication import default_config
app.config.from_object(default_config)
```

You should not use this function to load the actual configuration but rather configuration defaults. The actual config should be loaded with `from_pyfile()` and ideally from a location not within the package because the package might be installed system wide.

See Development / Production for an example of class-based configuration using `from_object()`.

Parameters: **obj** – an import name or object

`from_pyfile`(*filename*, *silent=False*)

Updates the values in the config from a Python file. This function behaves as if the file was imported as module with the `from_object()` function.

Parameters:
- **filename** – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** – set to `True` if you want silent failure for missing files.

▶ *Changelog*

`get_namespace`(*namespace*, *lowercase=True*, *trim_namespace=True*)

Returns a dictionary containing a subset of configuration options that match the specified namespace/prefix. Example usage:

```
app.config['IMAGE_STORE_TYPE'] = 'fs'
app.config['IMAGE_STORE_PATH'] = '/var/app/images'
app.config['IMAGE_STORE_BASE_URL'] = 'http://img.website.com'
image_store_config = app.config.get_namespace('IMAGE_STORE_')
```

The resulting dictionary *image_store_config* would look like:

```
{
    'type': 'fs',
    'path': '/var/app/images',
    'base_url': 'http://img.website.com'
}
```

This is often useful when configuration options map directly to keyword arguments in functions or class constructors.

Parameters:
- **namespace** – a configuration namespace
- **lowercase** – a flag indicating if the keys of the resulting dictionary should be lowercase
- **trim_namespace** – a flag indicating if the keys of the resulting dictionary should not include the namespace

▶ *Changelog*

Stream Helpers

flask.**stream_with_context**(*generator_or_function*)

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()))
```

▶ *Changelog*

Useful Internals

class `flask.ctx.`**`RequestContext`**(*app*, *environ*, *request=None*)

The request context contains all request relevant information. It is created at the beginning of the request and pushed to the *_request_ctx_stack* and removed at the end of it. It will create the URL adapter and request object for the WSGI environment provided.

Do not attempt to use this class directly, instead use **`test_request_context()`** and **`request_context()`** to create this object.

When the request context is popped, it will evaluate all the functions registered on the application for teardown execution (**`teardown_request()`**).

The request context is automatically popped at the end of the request for you. In debug mode the request context is kept around if exceptions happen so that interactive debuggers have a chance to introspect the data. With 0.4 this can also be forced for requests that did not fail and outside of `DEBUG` mode. By setting `'flask._preserve_context'` to `True` on the WSGI environment the context will not pop itself at the end of the request. This is used by the **`test_client()`** for example to implement the deferred cleanup functionality.

You might find this helpful for unittests where you need the information from the context local around for a little longer. Make sure to properly **`pop()`** the stack yourself in that situation, otherwise your unittests will leak memory.

`copy`()

Creates a copy of this request context with the same request object. This can be used to move a request context to a different greenlet. Because the actual request object is the same this cannot be used to move a request context to a different thread unless access to the request object is locked.

▶ *Changelog*

match_request()

Can be overridden by a subclass to hook into the matching of the request.

pop(*exc=<object object>*)

Pops the request context and unbinds it by doing that. This will also trigger the execution of functions registered by the **teardown_request()** decorator.

▶ *Changelog*

push()

Binds the request context to the current context.

flask._request_ctx_stack

The internal **LocalStack** that holds **RequestContext** instances. Typically, the **request** and **session** proxies should be accessed instead of the stack. It may be useful to access the stack in extension code.

The following attributes are always present on each layer of the stack:

app
: the active Flask application.

url_adapter
: the URL adapter that was used to match the request.

request
: the current request object.

session
: the active session object.

g
: an object with all the attributes of the **flask.g** object.

flashes
: an internal cache for the flashed messages.

Example usage:

```
from flask import _request_ctx_stack

def get_session():
    ctx = _request_ctx_stack.top
```

```
    if ctx is not None:
        return ctx.session
```

class flask.ctx.**AppContext**(*app*)

The application context binds an application object implicitly to the current thread or greenlet, similar to how the **RequestContext** binds request information. The application context is also implicitly created if a request context is created but the application is not on top of the individual application context.

pop(*exc=<object object>*)

Pops the app context.

push()

Binds the app context to the current context.

flask.**_app_ctx_stack**

The internal **LocalStack** that holds **AppContext** instances. Typically, the **current_app** and **g** proxies should be accessed instead of the stack. Extensions can access the contexts on the stack as a namespace to store data.

▶ *Changelog*

class flask.blueprints.**BlueprintSetupState**(*blueprint*, *app*, *options*, *first_registration*)

Temporary holder object for registering a blueprint with the application. An instance of this class is created by the **make_setup_state()** method and later passed to all register callback functions.

add_url_rule(*rule*, *endpoint=None*, *view_func=None*, ***options*)

A helper method to register a rule (and optionally a view function) to the application. The endpoint is automatically prefixed with the blueprint's name.

app = *None*

a reference to the current application

blueprint = *None*

a reference to the blueprint that created this setup state.

first_registration = *None*

as blueprints can be registered multiple times with the application and not everything wants to be registered multiple times on it, this attribute can be used to figure out if the blueprint was registered in the past already.

options = *None*

a dictionary with all options that were passed to the **register_blueprint()** method.

subdomain = *None*

The subdomain that the blueprint should be active for, `None` otherwise.

url_defaults = *None*

A dictionary with URL defaults that is added to each and every URL that was defined with the blueprint.

url_prefix = *None*

The prefix that should be used for all URLs defined on the blueprint.

Signals

▶ *Changelog*

signals.**signals_available**

`True` if the signaling system is available. This is the case when blinker is installed.

The following signals exist in Flask:

flask.**template_rendered**

This signal is sent when a template was successfully rendered. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
                        context)

from flask import template_rendered
template_rendered.connect(log_template_renders, app)
```

flask.**before_render_template**

This signal is sent before template rendering process. The signal is invoked with the instance of the template as *template* and the context as dictionary (named *context*).

Example subscriber:

```
def log_template_renders(sender, template, context, **extra):
    sender.logger.debug('Rendering template "%s" with context %s',
                        template.name or 'string template',
```

```
                                context)

from flask import before_render_template
before_render_template.connect(log_template_renders, app)
```

flask.request_started

This signal is sent when the request context is set up, before any request processing happens. Because the request context is already bound, the subscriber can access the request with the standard global proxies such as **request**.

Example subscriber:

```
def log_request(sender, **extra):
    sender.logger.debug('Request context is set up')

from flask import request_started
request_started.connect(log_request, app)
```

flask.request_finished

This signal is sent right before the response is sent to the client. It is passed the response to be sent named *response*.

Example subscriber:

```
def log_response(sender, response, **extra):
    sender.logger.debug('Request context is about to close down.  '
                        'Response: %s', response)

from flask import request_finished
request_finished.connect(log_response, app)
```

flask.got_request_exception

This signal is sent when an exception happens during request processing. It is sent *before* the standard exception handling kicks in and even in debug mode, where no exception handling happens. The exception itself is passed to the subscriber as *exception*.

Example subscriber:

```
def log_exception(sender, exception, **extra):
    sender.logger.debug('Got exception during processing: %s', exception)

from flask import got_request_exception
got_request_exception.connect(log_exception, app)
```

flask.request_tearing_down

This signal is sent when the request is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import request_tearing_down
request_tearing_down.connect(close_db_connection, app)
```

As of Flask 0.9, this will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.appcontext_tearing_down

This signal is sent when the app context is tearing down. This is always called, even if an exception is caused. Currently functions listening to this signal are called after the regular teardown handlers, but this is not something you can rely on.

Example subscriber:

```
def close_db_connection(sender, **extra):
    session.close()

from flask import appcontext_tearing_down
appcontext_tearing_down.connect(close_db_connection, app)
```

This will also be passed an *exc* keyword argument that has a reference to the exception that caused the teardown if there was one.

flask.appcontext_pushed

This signal is sent when an application context is pushed. The sender is the application. This is usually useful for unittests in order to temporarily hook in information. For instance it can be used to set a resource early onto the *g* object.

Example usage:

```
from contextlib import contextmanager
from flask import appcontext_pushed

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
```

```
    with appcontext_pushed.connected_to(handler, app):
        yield
```

And in the testcode:

```
def test_user_me(self):
    with user_set(app, 'john'):
        c = app.test_client()
        resp = c.get('/users/me')
        assert resp.data == 'username=john'
```

▶ *Changelog*

flask.appcontext_popped

This signal is sent when an application context is popped. The sender is the application. This usually falls in line with the **appcontext_tearing_down** signal.

▶ *Changelog*

flask.message_flashed

This signal is sent when the application is flashing a message. The messages is sent as *message* keyword argument and the category as *category*.

Example subscriber:

```
recorded = []
def record(sender, message, category, **extra):
    recorded.append((message, category))

from flask import message_flashed
message_flashed.connect(record, app)
```

▶ *Changelog*

class signals.Namespace

An alias for **blinker.base.Namespace** if blinker is available, otherwise a dummy class that creates fake signals. This class is available for Flask extensions that want to provide the same fallback system as Flask itself.

signal(*name*, *doc=None*)

Creates a new signal for this namespace if blinker is available, otherwise returns a fake signal that has a send method that will do nothing but will fail with a **RuntimeError** for all other operations, including connecting.

Class-Based Views

▶ *Changelog*

class flask.views.**View**

Alternative way to use view functions. A subclass has to implement **dispatch_request()** which is called with the view arguments from the URL routing system. If **methods** is provided the methods do not have to be passed to the **add_url_rule()** method explicitly:

```
class MyView(View):
    methods = ['GET']

    def dispatch_request(self, name):
        return 'Hello %s!' % name

app.add_url_rule('/hello/<name>', view_func=MyView.as_view('myview'))
```

When you want to decorate a pluggable view you will have to either do that when the view function is created (by wrapping the return value of **as_view()**) or you can use the **decorators** attribute:

```
class SecretView(View):
    methods = ['GET']
    decorators = [superuser_required]

    def dispatch_request(self):
        ...
```

The decorators stored in the decorators list are applied one after another when the view function is created. Note that you can *not* use the class based decorators since those would decorate the view class and not the generated view function!

classmethod **as_view**(*name*, **class_args*, ***class_kwargs*)

Converts the class into an actual view function that can be used with the routing system. Internally this generates a function on the fly which will instantiate the **View** on each request and call the **dispatch_request()** method on it.

The arguments passed to **as_view()** are forwarded to the constructor of the class.

decorators = *()*

The canonical way to decorate class-based views is to decorate the return value of as_view(). However since this moves parts of the logic from the class declaration to the place where it's hooked into the routing system.

You can place one or more decorators in this list and whenever the view function is created the result is automatically decorated.

▶ *Changelog*

dispatch_request()

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

methods = *None*

A list of methods this view can handle.

provide_automatic_options = *None*

Setting this disables or force-enables the automatic options handling.

class flask.views.**MethodView**

A class-based view that dispatches request methods to the corresponding class methods. For example, if you implement a `get` method, it will be used to handle `GET` requests.

```
class CounterAPI(MethodView):
    def get(self):
        return session.get('counter', 0)

    def post(self):
        session['counter'] = session.get('counter', 0) + 1
        return 'OK'

app.add_url_rule('/counter', view_func=CounterAPI.as_view('counter'))
```

dispatch_request(**args*, ***kwargs*)

Subclasses have to override this method to implement the actual view function code. This method is called with all the arguments from the URL rule.

URL Route Registrations

Generally there are three ways to define rules for the routing system:

1. You can use the **flask.Flask.route()** decorator.
2. You can use the **flask.Flask.add_url_rule()** function.
3. You can directly access the underlying Werkzeug routing system which is exposed as **flask.Flask.url_map**.

Variable parts in the route can be specified with angular brackets (`/user/<username>`). By default a variable part in the URL accepts any string without a slash however a different converter can be specified as well by using `<converter:name>`.

Variable parts are passed to the view function as keyword arguments.

The following converters are available:

string	accepts any text without a slash (the default)
int	accepts integers
float	like *int* but for floating point values
path	like the default but also accepts slashes
any	matches one of the items provided
uuid	accepts UUID strings

Custom converters can be defined using **`flask.Flask.url_map`**.

Here are some examples:

```
@app.route('/')
def index():
    pass

@app.route('/<username>')
def show_user(username):
    pass

@app.route('/post/<int:post_id>')
def show_post(post_id):
    pass
```

An important detail to keep in mind is how Flask deals with trailing slashes. The idea is to keep each URL unique so the following rules apply:

1. If a rule ends with a slash and is requested without a slash by the user, the user is automatically redirected to the same page with a trailing slash attached.
2. If a rule does not end with a trailing slash and the user requests the page with a trailing slash, a 404 not found is raised.

This is consistent with how web servers deal with static files. This also makes it possible to use relative link targets safely.

You can also define multiple rules for the same function. They have to be unique however. Defaults can also be specified. Here for example is a definition for a URL that accepts an optional page:

```
@app.route('/users/', defaults={'page': 1})
@app.route('/users/page/<int:page>')
def show_users(page):
    pass
```

This specifies that `/users/` will be the URL for page one and `/users/page/N` will be the URL for page `N`.

If a URL contains a default value, it will be redirected to its simpler form with a 301 redirect. In the above example, `/users/page/1` will be redirected to `/users/`. If your route handles `GET` and `POST` requests, make sure the default route only handles `GET`, as redirects can't preserve form data.

```
@app.route('/region/', defaults={'id': 1})
@app.route('/region/<id>', methods=['GET', 'POST'])
def region(id):
   pass
```

Here are the parameters that **`route()`** and **`add_url_rule()`** accept. The only difference is that with the route parameter the view function is defined with the decorator instead of the *view_func* parameter.

rule	the URL rule as string
endpoint	the endpoint for the registered URL rule. Flask itself assumes that the name of the view function is the name of the endpoint if not explicitly stated.
view_func	the function to call when serving a request to the provided endpoint. If this is not provided one can specify the function later by storing it in the **`view_functions`** dictionary with the endpoint as key.
defaults	A dictionary with defaults for this rule. See the example above for how defaults work.
subdomain	specifies the rule for the subdomain in case subdomain matching is in use. If not specified the default subdomain is assumed.
***options*	the options to be forwarded to the underlying **`Rule`** object. A change to Werkzeug is handling of method options. methods is a list of methods this rule should be limited to (`GET`, `POST` etc.). By default a rule just listens for `GET` (and implicitly `HEAD`). Starting with Flask 0.6, `OPTIONS` is implicitly added and handled by the standard request handling. They have to be specified as keyword arguments.

View Function Options

For internal usage the view functions can have some attributes attached to customize behavior the view function would normally not have control over. The following attributes can be provided optionally to either override some defaults to **`add_url_rule()`** or general behavior:

- *__name__*: The name of a function is by default used as endpoint. If endpoint is provided explicitly this value is used. Additionally this will be prefixed with the name of the blueprint by default which cannot be customized from the function itself.
- *methods*: If methods are not provided when the URL rule is added, Flask will look on the view function object itself if a *methods* attribute exists. If it does, it will pull the information for the methods from there.
- *provide_automatic_options*: if this attribute is set Flask will either force enable or disable the automatic implementation of the HTTP `OPTIONS` response. This can be useful when working with decorators that want to customize the `OPTIONS` response on a per-view basis.
- *required_methods*: if this attribute is set, Flask will always add these methods when registering a URL rule even if the methods were explicitly overridden in the `route()` call.

Full example:

```
def index():
    if request.method == 'OPTIONS':
        # custom options handling here
        ...
    return 'Hello World!'
index.provide_automatic_options = False
index.methods = ['GET', 'OPTIONS']

app.add_url_rule('/', index)
```

▶ *Changelog*

Command Line Interface

class `flask.cli.`**`FlaskGroup`**(*add_default_commands=True*, *create_app=None*, *add_version_option=True*, *load_dotenv=True*, ***extra*)

Special subclass of the **`AppGroup`** group that supports loading more commands from the configured Flask app. Normally a developer does not have to interface with this class but there are some very advanced use cases for which it makes sense to create an instance of this.

For information as of why this is useful see Custom Scripts.

Parameters:
- **add_default_commands** – if this is True then the default run and shell commands wil be added.
- **add_version_option** – adds the `--version` option.
- **create_app** – an optional callback that is passed the script info and returns the loaded app.
- **load_dotenv** – Load the nearest `.env` and `.flaskenv` files to set environment variables. Will also change the working directory to the directory containing the first file found.

Changed in version 1.0: If installed, python-dotenv will be used to load environment variables from `.env` and `.flaskenv` files.

▶ *Changelog*

`get_command`(*ctx*, *name*)

Given a context and a command name, this returns a **`Command`** object if it exists or returns *None*.

`list_commands`(*ctx*)

Returns a list of subcommand names in the order they should appear.

`main`(**args*, ***kwargs*)

This is the way to invoke a script with all the bells and whistles as a command line application. This will always terminate the application after a call. If this is not wanted, `SystemExit` needs to be caught.

This method is also available by directly calling the instance of a **`Command`**.

New in version 3.0: Added the *standalone_mode* flag to control the standalone mode.

▶ *Changelog*

Parameters:
- **args** – the arguments that should be used for parsing. If not provided, `sys.argv[1:]` is used.
- **prog_name** – the program name that should be used. By default the program name is constructed by taking the file name from `sys.argv[0]`.
- **complete_var** – the environment variable that controls the bash completion support. The default is `"_<prog_name>_COMPLETE"` with prog name in uppercase.
- **standalone_mode** – the default behavior is to invoke the script in standalone mode. Click will then handle exceptions and convert them into error messages and the function will never return but shut down the interpreter. If this is set to *False* they will be propagated to the caller and the return value of this function is the return value of **`invoke()`**.
- **extra** – extra keyword arguments are forwarded to the context constructor. See **`Context`** for more information.

class `flask.cli.`**`AppGroup`**(*name=None*, *commands=None*, ***attrs*)

This works similar to a regular click **`Group`** but it changes the behavior of the **`command()`** decorator so that it automatically wraps the functions in **`with_appcontext()`**.

Not to be confused with **`FlaskGroup`**.

command(*args, **kwargs)

This works exactly like the method of the same name on a regular `click.Group` but it wraps callbacks in `with_appcontext()` unless it's disabled by passing `with_appcontext=False`.

group(*args, **kwargs)

This works exactly like the method of the same name on a regular `click.Group` but it defaults the group class to `AppGroup`.

class flask.cli.**ScriptInfo**(*app_import_path=None*, *create_app=None*)

Help object to deal with Flask applications. This is usually not necessary to interface with as it's used internally in the dispatching to click. In future versions of Flask this object will most likely play a bigger role. Typically it's created automatically by the `FlaskGroup` but you can also manually create it and pass it onwards as click object.

app_import_path = *None*

Optionally the import path for the Flask application.

create_app = *None*

Optionally a function that is passed the script info to create the instance of the application.

data = *None*

A dictionary with arbitrary data that can be associated with this script info.

load_app()

Loads the Flask app (if not yet loaded) and returns it. Calling this multiple times will just result in the already loaded app to be returned.

flask.cli.**load_dotenv**(*path=None*)

Load "dotenv" files in order of precedence to set environment variables.

If an env var is already set it is not overwritten, so earlier files in the list are preferred over later files.

Changes the current working directory to the location of the first file found, with the assumption that it is in the top level project directory and will be where the Python path should import local packages from.

This is a no-op if python-dotenv is not installed.

Parameters: **path** – Load the file at this location instead of searching.
Returns: `True` if a file was loaded.

New in version 1.0.

▶ *Changelog*

flask.cli.**with_appcontext**(*f*)

Wraps a callback so that it's guaranteed to be executed with the script's application context. If callbacks are registered directly to the `app.cli` object then they are wrapped with this function by default unless it's disabled.

flask.cli.**pass_script_info**(*f*)

Marks a function so that an instance of **`ScriptInfo`** is passed as first argument to the click callback.

flask.cli.**run_command** = <*click.core.Command object*>

Run a local development server.

This server is for development purposes only. It does not provide the stability, security, or performance of production WSGI servers.

The reloader and debugger are enabled by default if FLASK_ENV=development or FLASK_DEBUG=1.

flask.cli.**shell_command** = <*click.core.Command object*>

Runs an interactive Python shell in the context of a given Flask application. The application will populate the default namespace of this shell according to it's configuration.

This is useful for executing small snippets of management code without having to manually configure the application.