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` v: 3.0.x
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`)

▶ *Changelog*

Parameters:

- **import_name** (*str*) – the name of the application package
- **static_url_path** (*str* | *None*) – 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** (*str* | *os.PathLike[str]* | *None*) – The folder with static files that is served at `static_url_path`. Relative to the application `root_path` or an absolute path. Defaults to `'static'`.
- **static_host** (*str* | *None*) – 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** (*bool*) – set `url_map.host_matching` attribute. Defaults to False.
- **subdomain_matching** (*bool*) – consider the subdomain relative to **`SERVER_NAME`** when matching routes. Defaults to False.
- **template_folder** (*str* | *os.PathLike[str]* | *None*) – 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** (*str* | *None*) – 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** (*bool*) – 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** (*str* | *None*) – The path to the root of the application files. This should only be set manually when it can't be detected automatically, such as for namespace packages

`request_class`

alias of **Request**

response_class

alias of **Response**

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

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

▶ *Changelog*

cli: *Group*

The Click command group for registering CLI commands for this object. The commands are available from the `flask` command once the application has been discovered and blueprints have been registered.

get_send_file_max_age(*filename*)

Used by **send_file()** to determine the `max_age` cache value for a given file path if it wasn't passed.

By default, this returns **SEND_FILE_MAX_AGE_DEFAULT** from the configuration of **current_app**. This defaults to `None`, which tells the browser to use conditional requests instead of a timed cache, which is usually preferable.

Note this is a duplicate of the same method in the Flask class.

▶ *Changelog*

Parameters: **filename** (*str* | *None*) –
Return type: int | None

send_static_file(*filename*)

The view function used to serve files from **static_folder**. A route is automatically registered for this view at **static_url_path** if **static_folder** is set.

Note this is a duplicate of the same method in the Flask class.

▶ *Changelog*

Parameters: **filename** (*str*) –
Return type: *Response*

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

Open a resource file relative to **root_path** for reading.

For example, if the file `schema.sql` is next to the file `app.py` where the `Flask` app is defined, it can be opened with:

```
with app.open_resource("schema.sql") as f:
    conn.executescript(f.read())
```

Parameters:
- **resource** (*str*) – Path to the resource relative to **root_path**.
- **mode** (*str*) – Open the file in this mode. Only reading is supported, valid values are “r” (or “rt”) and “rb”.

Return type: *IO*

Note this is a duplicate of the same method in the Flask class.

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** (*str*) – the name of the resource. To access resources within subfolders use forward slashes as separator.
- **mode** (*str*) – resource file opening mode, default is ‘rb’.

Return type: *IO*

create_jinja_environment()

Create the Jinja environment based on **jinja_options** and the various Jinja-related methods of the app. Changing **jinja_options** after this will have no effect. Also adds Flask-related globals and filters to the environment.

▶ *Changelog*

Return type: *Environment*

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.

▶ *Changelog*

Parameters: **request** (*Request* | *None*) –
Return type: *MapAdapter* | None

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** (*dict[str, Any]*) – the context as a dictionary that is updated in place to add extra variables.

Return type: None

`make_shell_context()`

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

▶ *Changelog*

Return type: dict[str, *Any*]

`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 Deploying to Production 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_reloa` v: 3.0.x
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** (*str* | *None*) – 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** (*int* | *None*) – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **debug** (*bool* | *None*) – if given, enable or disable debug mode. See **debug**.
- **load_dotenv** (*bool*) – 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** (*Any*) – the options to be forwarded to the underlying Werkzeug server. See **werkzeug.serving.run_simple()** for more information.

Return type: None

▶ *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*

Parameters:
- **use_cookies** (*bool*) –
- **kwargs** (*t.Any*) –

Return type: FlaskClient

test_cli_runner(**kwargs)

Create a CLI runner for testing CLI commands. See Running Commands with the CLI Runner.

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

▶ *Changelog*

Parameters: **kwargs** (*t.Any*) –
Return type: FlaskCliRunner

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*

Parameters: **e** (*HTTPException*) –
Return type: HTTPException | ft.ResponseReturnValue

handle_user_exception(e)

This method is called whenever an exception occurs that should be h
special case is **HTTPException** which is forwarded to the

handle_http_exception() method. This function will either return a response value or reraise the exception with the same traceback.

▶ *Changelog*

Parameters: **e** (*Exception*) –
Return type: HTTPException | ft.ResponseReturnValue

handle_exception(e)

Handle an exception that did not have an error handler associated with it, or that was raised from an error handler. This always causes a 500 `InternalServerError`.

Always sends the **got_request_exception** signal.

If **PROPAGATE_EXCEPTIONS** is `True`, such as in debug mode, the error will be re-raised so that the debugger can display it. Otherwise, the original exception is logged, and an **InternalServerError** is returned.

If an error handler is registered for `InternalServerError` or `500`, it will be used. For consistency, the handler will always receive the `InternalServerError`. The original unhandled exception is available as `e.original_exception`.

▶ *Changelog*

Parameters: **e** (*Exception*) –
Return type: *Response*

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*

Parameters: **exc_info** (*tuple[type, BaseException, TracebackType]* | *tuple[None, None, None]*) –
Return type: None

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 v: 3.0.x to convert the return value to a proper response object, call **make_response()**.

▶ *Changelog*

Return type: ft.ResponseReturnValue

`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*

Return type: *Response*

`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*

Return type: *Response*

`ensure_sync(func)`

Ensure that the function is synchronous for WSGI workers. Plain `def` functions are returned as-is. `async def` functions are wrapped to run and wait for the response.

Override this method to change how the app runs async views.

▶ *Changelog*

Parameters: **func** (*Callable[[...], Any]*) –
Return type: *Callable*[[...], *Any*]

`async_to_sync(func)`

Return a sync function that will run the coroutine function.

```
result = app.async_to_sync(func)(*args, **kwargs)
```

Override this method to change how the app converts async code to be synchronously callable.

▶ *Changelog*

Parameters: **func** (*Callable[[...], Coroutine[Any, Any, Any]]*) –

Return type: *Callable*[[...], *Any*]

url_for(*endpoint*, *, *_anchor=None*, *_method=None*, *_scheme=None*, *_external=None*, ***values*)

Generate a URL to the given endpoint with the given values.

This is called by **flask.url_for()**, and can be called directly as well.

An *endpoint* is the name of a URL rule, usually added with **@app.route()**, and usually the same name as the view function. A route defined in a **Blueprint** will prepend the blueprint's name separated by a `.` to the endpoint.

In some cases, such as email messages, you want URLs to include the scheme and domain, like `https://example.com/hello`. When not in an active request, URLs will be external by default, but this requires setting **SERVER_NAME** so Flask knows what domain to use. **APPLICATION_ROOT** and **PREFERRED_URL_SCHEME** should also be configured as needed. This config is only used when not in an active request.

Functions can be decorated with **url_defaults()** to modify keyword arguments before the URL is built.

If building fails for some reason, such as an unknown endpoint or incorrect values, the app's **handle_url_build_error()** method is called. If that returns a string, that is returned, otherwise a **BuildError** is raised.

Parameters:

- **endpoint** (*str*) – The endpoint name associated with the URL to generate. If this starts with a `.`, the current blueprint name (if any) will be used.
- **_anchor** (*str* | *None*) – If given, append this as `#anchor` to the URL.
- **_method** (*str* | *None*) – If given, generate the URL associated with this method for the endpoint.
- **_scheme** (*str* | *None*) – If given, the URL will have this scheme if it is external.
- **_external** (*bool* | *None*) – If given, prefer the URL to be internal (False) or require it to be external (True). External URLs include the scheme and domain. When not in an active request, URLs are external by default.
- **values** (*Any*) – Values to use for the variable parts of the URL rule. Unknown keys are appended as query string arguments, like `?a=b&c=d`.

Return type: str

▶ *Changelog*

`make_response(rv)`

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

Parameters: **rv** (*ft.ResponseReturnValue*) –
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`
: A response object is created with the string encoded to UTF-8 as the body.

`bytes`
: A response object is created with the bytes as the body.

`dict`
: A dictionary that will be jsonify'd before being returned.

`list`
: A list that will be jsonify'd before being returned.

`generator` or `iterator`
: A generator that returns `str` or `bytes` to be streamed as the response.

`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.

Return type: Response

v: 3.0.x

▶ *Changelog*

`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.

Return type: ft.ResponseReturnValue | None

`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** (*Response*) – a **`response_class`** object.
Returns: a new response object or the same, has to be an instance of **`response_class`**.
Return type: *Response*

`do_teardown_request(exc=_sentinel)`

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** (*BaseException* | *None*) – An unhandled exception raised while dispatching the request. Detected from the current exception information if not passed. Passed to each teardown function.
Return type: None

▶ *Changelog*

`do_teardown_appcontext(exc=_sentinel)`

v: 3.0.x

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*

Parameters: **exc** (*BaseException* | *None*) –
Return type: None

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*

Return type: *AppContext*

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** (*WSGIEnvironment*) – a WSGI environment
Return type: RequestContext

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 app.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** (*Any*) – other positional arguments passed to **EnvironBuilder**.
- **kwargs** (*Any*) – other keyword arguments passed to **EnvironBuilder**.

Return type: *RequestContext*

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

Return type: cabc.Iterable[bytes]

aborter_class

alias of **`Aborter`**

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

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

Parameters:
- **name** (*str* | *None*) – the optional name of the filter, otherwise the function name will be used.
- **f** (*Callable[[...], Any]*) –

Return type: None

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

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

▶ *Changelog*

Parameters:
- **name** (*str* | *None*) – the optional name of the global function, otherwise the function name will be used.
- **f** (*Callable[[...], Any]*) –

Return type: None

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

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

▶ *Changelog*

Parameters:
- **name** (*str* | *None*) – the optional name of the test, otherwise the function name will be used.
- **f** (*Callable[[...], bool]*) –

Return type: None

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

Register a rule for routing incoming requests and building URLs. The **route()** decorator is a shortcut to call this with the `view_func` argument. These are equivalent:

```
@app.route("/")
def index():
    ...


def index():
    ...

app.add_url_rule("/", view_func=index)
```

See URL Route Registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed. An error will be raised if a function has already been registered for the endpoint.

The `methods` parameter defaults to `["GET"]`. `HEAD` is always added automatically, and `OPTIONS` is added automatically by default.

`view_func` does not necessarily need to be passed, but if the rule should participate in routing an endpoint name must be associated with a view function at some point with the **endpoint()** decorator.

```
app.add_url_rule("/", endpoint="index")

@app.endpoint("index")
def index():
    ...
```

If `view_func` has a `required_methods` attribute, those methods are added to the passed and automatic methods. If it has a `provide_automatic_methods` attribute, it is used as the default if the parameter is not passed.

Parameters:
- **rule** (*str*) – The URL rule string.
- **endpoint** (*str* | *None*) – The endpoint name to associate with the rule and view function. Used when routing and building URLs. Defaults to `view_func.__name__`.
- **view_func** (*ft.RouteCallable* | *None*) – The view function to associate with the endpoint name.
- **provide_automatic_options** (*bool* | *None*) – Add the `OPTIONS` method and respond to `OPTIONS` requests automatically.
- **options** (*t.Any*) – Extra options passed to the **`Rule`** object.

Return type: None

`after_request(f)`

Register a function to run after each request to this object.

The function is called with the response object, and must return a response object. This allows the functions to modify or replace the response before it is sent.

If a function raises an exception, any remaining `after_request` functions will not be called. Therefore, this should not be used for actions that must execute, such as to close resources. Use **`teardown_request()`** for that.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use **`Blueprint.after_app_request()`**.

Parameters: **f** (*T_after_request*) –

Return type: *T_after_request*

`app_ctx_globals_class`

alias of **`_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*

Return type: str

before_request(*f*)

Register 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.

```
@app.before_request
def load_user():
    if "user_id" in session:
        g.user = db.session.get(session["user_id"])
```

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.

This is available on both app and blueprint objects. When used on an app, this executes before every request. When used on a blueprint, this executes before every request that the blueprint handles. To register with a blueprint and execute before every request, use **`Blueprint.before_app_request()`**.

Parameters: **f** (*T_before_request*) –
Return type: *T_before_request*

config_class

alias of **`Config`**

context_processor(*f*)

Registers a template context processor function. These functions run before rendering a template. The keys of the returned dict are added as variables available in the template.

This is available on both app and blueprint objects. When used on an app, this is called for every rendered template. When used on a blueprint, this is called for templates rendered from the blueprint's views. To register with a blueprint and affect every template, use **`Blueprint.app_context_processor()`**.

Parameters: **f** (*T_template_context_processor*) –
Return type: *T_template_context_processor*

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 loders of the application and the individual blueprints.

▶ *Changelog*

Return type: *DispatchingJinjaLoader*

property **debug**: *bool*

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. It may not behave as expected if set late.

Do not enable debug mode when deploying in production.

Default: `False`

delete(*rule*, ***options*)

Shortcut for **route()** with `methods=["DELETE"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

endpoint(*endpoint*)

Decorate a view function to register it for the given endpoint. Used if a rule is added without a `view_func` with **add_url_rule()**.

```
app.add_url_rule("/ex", endpoint="example")

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

Parameters: **endpoint** (*str*) – The endpoint name to associate with the view function.

Return type: *Callable*[[*F*], *F*]

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
```

This is available on both app and blueprint objects. When used on an app, this can handle errors from every request. When used on a blueprint, this can handle errors from requests that the blueprint handles. To register with a blueprint and affect every request, use **`Blueprint.app_errorhandler()`**.

▶ *Changelog*

Parameters: **code_or_exception** (*type[Exception]* | *int*) – the code as integer for the handler, or an arbitrary exception
Return type: *Callable*[[*T_error_handler*], *T_error_handler*]

`get(rule, **options)`

Shortcut for **`route()`** with `methods=["GET"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

`handle_url_build_error(error, endpoint, values)`

Called by **`url_for()`** if a **`BuildError`** was raised. If this returns a value, it will be returned by `url_for`, otherwise the error will be re-raised.

Each function in **`url_build_error_handlers`** is called with `error`, `endpoint` and `values`. If a function returns `None` or raises a `BuildError`, it is skipped. Otherwise, its return value is returned by `url_for`.

Parameters:
- **error** (*BuildError*) – The active `BuildError` being handled.
- **endpoint** (*str*) – The endpoint being built.
- **values** (*dict[str, Any]*) – The keyword arguments passed to `url_for`.

Return type: str

property **has_static_folder**: *bool*

True if **static_folder** is set.

▶ *Changelog*

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*

Parameters:
- **endpoint** (*str*) –
- **values** (*dict[str, Any]*) –

Return type: None

iter_blueprints()

Iterates over all blueprints by the order they were registered.

▶ *Changelog*

Return type: t.ValuesView[Blueprint]

property **jinja_env**: *Environment*

The Jinja environment used to load templates.

The environment is created the first time this property is accessed. Changing **jinja_options** after that will have no effect.

jinja_environment

alias of **Environment**

property **jinja_loader**: *BaseLoader* | *None*

The Jinja loader for this object's templates. By default this is a class **jinja2.loaders.FileSystemLoader** to **template_folder** if it is set.

▶ *Changelog*

jinja_options: *dict[str, t.Any] = {}*

Options that are passed to the Jinja environment in **create_jinja_environment()**. Changing these options after the environment is created (accessing **jinja_env**) will have no effect.

▶ *Changelog*

json_provider_class

alias of **DefaultJSONProvider**

property **logger**: *Logger*

A standard Python **Logger** for the app, with the same name as **name**.

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.

▶ *Changelog*

make_aborter()

Create the object to assign to **aborter**. That object is called by **flask.abort()** to raise HTTP errors, and can be called directly as well.

By default, this creates an instance of **aborter_class**, which defaults to **werkzeug.exceptions.Aborter**.

▶ *Changelog*

Return type: *Aborter*

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*

Parameters: **instance_relative** (*bool*) –
Return type: *Config*

property **name**: *str*

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*

patch(*rule*, ***options*)

Shortcut for **route()** with `methods=["PATCH"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

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)`

post(*rule*, ***options*)

Shortcut for **route()** with `methods=["POST"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

put(*rule*, ***options*)

Shortcut for **route()** with `methods=["PUT"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

redirect(*location*, *code=302*)

Create a redirect response object.

This is called by **flask.redirect()**, and can be called directly as well.

Parameters:
- **location** (*str*) – The URL to redirect to.
- **code** (*int*) – The status code for the redirect.

Return type: BaseResponse

▶ *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** (*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** (*t.Any*) – Additional keyword arguments are passed to **`BlueprintSetupState`**. They can be accessed in **`record()`** callbacks.

Return type: None

▶ *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*

Parameters:
- **code_or_exception** (*type[Exception]* | *int*) –
- **f** (*ft.ErrorHandlerCallable*) –

Return type: None

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

Decorate a view function to register it with the given URL rule and options. Calls **`add_url_rule()`**, which has more details about the implementation.

```
@app.route("/")
def index():
    return "Hello, World!"
```

See URL Route Registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed.

v: 3.0.x

The `methods` parameter defaults to `["GET"]`. `HEAD` and `OPTIONS` are added automatically.

Parameters:
- **rule** (*str*) – The URL rule string.
- **options** (*Any*) – Extra options passed to the `Rule` object.

Return type: *Callable*[[*T_route*], *T_route*]

`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*

Parameters: **filename** (*str*) –
Return type: bool

`shell_context_processor(f)`

Registers a shell context processor function.

▶ *Changelog*

Parameters: **f** (*T_shell_context_processor*) –
Return type: *T_shell_context_processor*

`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*

Parameters: **error** (*BaseException* | *None*) –
Return type: bool

property `static_folder: str | None`

The absolute path to the configured static folder. `None` if no static folder is set.

property `static_url_path: str | None`

v: 3.0.x

The URL prefix that the static route will be accessible from.

If it was not configured during init, it is derived from **static_folder**.

teardown_appcontext(*f*)

Registers a function to be called when the application context is popped. The application context is typically popped after the request context for each request, at the end of CLI commands, or after a manually pushed context ends.

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

When the `with` block exits (or `ctx.pop()` is called), the teardown functions are called just before the app context is made inactive. 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.

Teardown functions must avoid raising exceptions. If they execute code that might fail they must surround that code with a `try/except` block and log any errors.

The return values of teardown functions are ignored.

▶ *Changelog*

Parameters: **f** (*T_teardown*) –
Return type: *T_teardown*

teardown_request(*f*)

Register a function to be called when the request context is popped. Typically this happens at the end of each request, but contexts may be pushed manually as well during testing.

```
with app.test_request_context():
    ...
```

When the `with` block exits (or `ctx.pop()` is called), the teardown functions are called just before the request context is made inactive.

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 exception and the teardown will not receive it.

Teardown functions must avoid raising exceptions. If they execute code that might fail they must surround that code with a `try/except` block and log any errors.

The return values of teardown functions are ignored.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use **`Blueprint.teardown_app_request()`**.

Parameters: **f** (*T_teardown*) –
Return type: *T_teardown*

`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** (*str* | *None*) – the optional name of the filter, otherwise the function name will be used.
Return type: *Callable*[[*T_template_filter*], *T_template_filter*]

`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** (*str* | *None*) – the optional name of the global function, otherwise the function name will be used.
Return type: *Callable*[[*T_template_global*], *T_template_global*]

`template_test(name=None)`

A decorator that is used to register custom template test. You can sp

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** (*str* | *None*) – the optional name of the test, otherwise the function name will be used.
Return type: *Callable*[[*T_template_test*], *T_template_test*]

`test_cli_runner_class`: *type[FlaskCliRunner] | None = 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.

▶ *Changelog*

`test_client_class`: *type[FlaskClient] | None = None*

The **`test_client()`** method creates an instance of this test client class. Defaults to **`FlaskClient`**.

▶ *Changelog*

`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.

▶ *Changelog*

Parameters: **e** (*Exception*) –
Return type: bool

`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.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect every request, use **`Blueprint.app_url_defaults()`**.

Parameters: **f** (*T_url_defaults*) –
Return type: *T_url_defaults*

`url_map_class`

alias of **`Map`**

`url_rule_class`

alias of **`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.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect eve

use **`Blueprint.app_url_value_preprocessor()`**.

Parameters: **f** (*T_url_value_preprocessor*) –
Return type: *T_url_value_preprocessor*

`instance_path`

Holds the path to the instance folder.

▶ *Changelog*

`config`

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

`aborter`

An instance of **`aborter_class`** created by **`make_aborter()`**. This is called by **`flask.abort()`** to raise HTTP errors, and can be called directly as well.

▶ *Changelog*

`json`: *JSONProvider*

Provides access to JSON methods. Functions in `flask.json` will call methods on this provider when the application context is active. Used for handling JSON requests and responses.

An instance of **`json_provider_class`**. Can be customized by changing that attribute on a subclass, or by assigning to this attribute afterwards.

The default, **`DefaultJSONProvider`**, uses Python's built-in **`json`** library. A different provider can use a different JSON library.

▶ *Changelog*

`url_build_error_handlers`: *list[t.Callable[[Exception, str, dict[str, t.Any]], str]]*

A list of functions that are called by **`handle_url_build_error()`** when **`url_for()`** raises a **`BuildError`**. Each function is called with `error`, `endpoint` and `values`. If a function returns `None` or raises a `BuildError`, it is skipped. Otherwise, its return value is returned by `url_for`.

▶ *Changelog*

`teardown_appcontext_funcs`: *list[ft.TeardownCallable]*

A list of functions that are called when the application context is desl v: 3.0.x
Since the application context is also torn down if the request ends this is the place to store code that disconnects from databases.

▶ *Changelog*

`shell_context_processors`: *list[ft.ShellContextProcessorCallable]*

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

▶ *Changelog*

`blueprints`: *dict[str, Blueprint]*

Maps registered blueprint names to blueprint objects. The dict retains the order the blueprints were registered in. Blueprints can be registered multiple times, this dict does not track how often they were attached.

▶ *Changelog*

`extensions`: *dict[str, t.Any]*

a place where extensions can store application specific state. For example this is where an extension could store database engines and similar things.

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*

`url_map`

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
```

`import_name`

The name of the package or module that this object belongs to. Do n
this once it is set by the constructor.

template_folder

The path to the templates folder, relative to **root_path**, to add to the template loader. None if templates should not be added.

root_path

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

view_functions: *dict[str, ft.RouteCallable]*

A dictionary mapping endpoint names to view functions.

To register a view function, use the **route()** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

error_handler_spec: *dict[ft.AppOrBlueprintKey, dict[int | None, dict[type[Exception], ft.ErrorHandlerCallable]]]*

A data structure of registered error handlers, in the format `{scope: {code: {class: handler}}}`. The `scope` key is the name of a blueprint the handlers are active for, or `None` for all requests. The `code` key is the HTTP status code for `HTTPException`, or `None` for other exceptions. The innermost dictionary maps exception classes to handler functions.

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

This data structure is internal. It should not be modified directly and its format may change at any time.

before_request_funcs: *dict[ft.AppOrBlueprintKey, list[ft.BeforeRequestCallable]]*

A data structure of functions to call at the beginning of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **before_request()** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

after_request_funcs: *dict[ft.AppOrBlueprintKey, list[ft.AfterRequestCallable[t.Any]]]*

A data structure of functions to call at the end of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint tl… tions are active for, or `None` for all requests.

To register a function, use the **`after_request()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`teardown_request_funcs`: *dict[ft.AppOrBlueprintKey, list[ft.TeardownCallable]]*

A data structure of functions to call at the end of each request even if an exception is raised, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`teardown_request()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`template_context_processors`: *dict[ft.AppOrBlueprintKey, list[ft.TemplateContextProcessorCallable]]*

A data structure of functions to call to pass extra context values when rendering templates, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`context_processor()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`url_value_preprocessors`: *dict[ft.AppOrBlueprintKey, list[ft.URLValuePreprocessorCallable]]*

A data structure of functions to call to modify the keyword arguments passed to the view function, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`url_value_preprocessor()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`url_default_functions`: *dict[ft.AppOrBlueprintKey, list[ft.URLDefaultCallable]]*

A data structure of functions to call to modify the keyword arguments when generating URLs, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`url_defaults()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

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*, *cli_group=_sentinel*)

Parameters:
- **name** (*str*) –
- **import_name** (*str*) –
- **static_folder** (*str* | *os.PathLike[str]* | *None*) –
- **static_url_path** (*str* | *None*) –
- **template_folder** (*str* | *os.PathLike[str]* | *None*) –
- **url_prefix** (*str* | *None*) –
- **subdomain** (*str* | *None*) –
- **url_defaults** (*dict[str, t.Any]* | *None*) –
- **root_path** (*str* | *None*) –
- **cli_group** (*str* | *None*) –

cli: *Group*

The Click command group for registering CLI commands for this object. The commands are available from the `flask` command once the application has been discovered and blueprints have been registered.

get_send_file_max_age(*filename*)

Used by **send_file()** to determine the `max_age` cache value for a given file path if it wasn't passed.

By default, this returns **SEND_FILE_MAX_AGE_DEFAULT** from the configuration of **current_app**. This defaults to `None`, which tells the browser to use conditional requests instead of a timed cache, which is usually preferable.

Note this is a duplicate of the same method in the Flask class.

▶ *Changelog*

Parameters: **filename** (*str* | *None*) –
Return type: int | None

send_static_file(*filename*)

The view function used to serve files from **static_folder**. A route is automatically registered for this view at **static_url_path** if **static_folder** is set.

Note this is a duplicate of the same method in the Flask class.

▶ *Changelog*

Parameters: **filename** (*str*) –
Return type: Response

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

Open a resource file relative to **`root_path`** for reading.

For example, if the file `schema.sql` is next to the file `app.py` where the `Flask` app is defined, it can be opened with:

```
with app.open_resource("schema.sql") as f:
    conn.executescript(f.read())
```

Parameters:
- **resource** (*str*) – Path to the resource relative to **`root_path`**.
- **mode** (*str*) – Open the file in this mode. Only reading is supported, valid values are “r” (or “rt”) and “rb”.

Return type: *IO*

Note this is a duplicate of the same method in the Flask class.

`add_app_template_filter(f, name=None)`

Register a template filter, available in any template rendered by the application. Works like the **`app_template_filter()`** decorator. Equivalent to **`Flask.add_template_filter()`**.

Parameters:
- **name** (*str* | *None*) – the optional name of the filter, otherwise the function name will be used.
- **f** (*Callable[[...], Any]*) –

Return type: None

`add_app_template_global(f, name=None)`

Register a template global, available in any template rendered by the application. Works like the **`app_template_global()`** decorator. Equivalent to **`Flask.add_template_global()`**.

▶ *Changelog*

Parameters:
- **name** (*str* | *None*) – the optional name of the global, otherwise the function name will be used.
- **f** (*Callable[[...], Any]*) –

Return type: None

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

Register a template test, available in any template rendered by the application. Works like the **app_template_test()** decorator. Equivalent to **Flask.add_template_test()**.

▶ *Changelog*

Parameters:
- **name** (*str* | *None*) – the optional name of the test, otherwise the function name will be used.
- **f** (*Callable[[...], bool]*) –

Return type: None

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

Register a URL rule with the blueprint. See **Flask.add_url_rule()** for full documentation.

The URL rule is prefixed with the blueprint's URL prefix. The endpoint name, used with **url_for()**, is prefixed with the blueprint's name.

Parameters:
- **rule** (*str*) –
- **endpoint** (*str* | *None*) –
- **view_func** (*ft.RouteCallable* | *None*) –
- **provide_automatic_options** (*bool* | *None*) –
- **options** (*t.Any*) –

Return type: None

after_app_request(*f*)

Like **after_request()**, but after every request, not only those handled by the blueprint. Equivalent to **Flask.after_request()**.

Parameters: **f** (*T_after_request*) –
Return type: *T_after_request*

after_request(*f*)

Register a function to run after each request to this object.

The function is called with the response object, and must return a response object. This allows the functions to modify or replace the response before it is sent.

If a function raises an exception, any remaining `after_request` functions will not be called. Therefore, this should not be used for actions that mus such as to close resources. Use **teardown_request()** for that.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use **`Blueprint.after_app_request()`**.

Parameters: **f** (*T_after_request*) –
Return type: *T_after_request*

`app_context_processor(f)`

Like **`context_processor()`**, but for templates rendered by every view, not only by the blueprint. Equivalent to **`Flask.context_processor()`**.

Parameters: **f** (*T_template_context_processor*) –
Return type: *T_template_context_processor*

`app_errorhandler(code)`

Like **`errorhandler()`**, but for every request, not only those handled by the blueprint. Equivalent to **`Flask.errorhandler()`**.

Parameters: **code** (*type[Exception]* | *int*) –
Return type: *Callable*[[*T_error_handler*], *T_error_handler*]

`app_template_filter(name=None)`

Register a template filter, available in any template rendered by the application. Equivalent to **`Flask.template_filter()`**.

Parameters: **name** (*str* | *None*) – the optional name of the filter, otherwise the function name will be used.
Return type: *Callable*[[*T_template_filter*], *T_template_filter*]

`app_template_global(name=None)`

Register a template global, available in any template rendered by the application. Equivalent to **`Flask.template_global()`**.

▶ *Changelog*

Parameters: **name** (*str* | *None*) – the optional name of the global, otherwise the function name will be used.
Return type: *Callable*[[*T_template_global*], *T_template_global*]

`app_template_test(name=None)`

Register a template test, available in any template rendered by the application. Equivalent to **`Flask.template_test()`**.

▶ *Changelog*

Parameters: **name** (*str* | *None*) – the optional name of the test, otherwise the function name will be used.
Return type: *Callable*[[*T_template_test*], *T_template_test*]

app_url_defaults(f)

Like **url_defaults()**, but for every request, not only those handled by the blueprint. Equivalent to **Flask.url_defaults()**.

Parameters: **f** (*T_url_defaults*) –
Return type: *T_url_defaults*

app_url_value_preprocessor(f)

Like **url_value_preprocessor()**, but for every request, not only those handled by the blueprint. Equivalent to **Flask.url_value_preprocessor()**.

Parameters: **f** (*T_url_value_preprocessor*) –
Return type: *T_url_value_preprocessor*

before_app_request(f)

Like **before_request()**, but before every request, not only those handled by the blueprint. Equivalent to **Flask.before_request()**.

Parameters: **f** (*T_before_request*) –
Return type: *T_before_request*

before_request(f)

Register 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.

```
@app.before_request
def load_user():
    if "user_id" in session:
        g.user = db.session.get(session["user_id"])
```

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.

This is available on both app and blueprint objects. When used on an app, this executes before every request. When used on a blueprint, this executes before every request that the blueprint handles. To register with a blueprint cute before every request, use **Blueprint.before_app_request()**.

Parameters: **f** (*T_before_request*) –

Return type: *T_before_request*

`context_processor(f)`

Registers a template context processor function. These functions run before rendering a template. The keys of the returned dict are added as variables available in the template.

This is available on both app and blueprint objects. When used on an app, this is called for every rendered template. When used on a blueprint, this is called for templates rendered from the blueprint's views. To register with a blueprint and affect every template, use **`Blueprint.app_context_processor()`**.

Parameters: **f** (*T_template_context_processor*) –
Return type: *T_template_context_processor*

`delete(rule, **options)`

Shortcut for **`route()`** with `methods=["DELETE"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

`endpoint(endpoint)`

Decorate a view function to register it for the given endpoint. Used if a rule is added without a `view_func` with **`add_url_rule()`**.

```
app.add_url_rule("/ex", endpoint="example")

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

Parameters: **endpoint** (*str*) – The endpoint name to associate with the view function.
Return type: *Callable*[[*F*], *F*]

`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
```

This is available on both app and blueprint objects. When used on an app, this can handle errors from every request. When used on a blueprint, this can handle errors from requests that the blueprint handles. To register with a blueprint and affect every request, use **`Blueprint.app_errorhandler()`**.

▶ *Changelog*

Parameters: **code_or_exception** (*type[Exception]* | *int*) – the code as integer for the handler, or an arbitrary exception
Return type: *Callable*[[*T_error_handler*], *T_error_handler*]

`get(rule, **options)`

Shortcut for **`route()`** with `methods=["GET"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

property `has_static_folder: bool`

`True` if **`static_folder`** is set.

▶ *Changelog*

property `jinja_loader: BaseLoader | None`

The Jinja loader for this object's templates. By default this is a class **`jinja2.loaders.FileSystemLoader`** to **`template_folder`** if it is set.

▶ *Changelog*

`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

of the setup state.

Parameters:
- **app** (*App*) –
- **options** (*dict[str, t.Any]*) –
- **first_registration** (*bool*) –

Return type: BlueprintSetupState

`patch(rule, **options)`

Shortcut for **`route()`** with `methods=["PATCH"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

`post(rule, **options)`

Shortcut for **`route()`** with `methods=["POST"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

`put(rule, **options)`

Shortcut for **`route()`** with `methods=["PUT"]`.

▶ *Changelog*

Parameters:
- **rule** (*str*) –
- **options** (*Any*) –

Return type: *Callable*[[*T_route*], *T_route*]

`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.

Parameters: **func** (*Callable[[BlueprintSetupState], None]*) –

Return type: None

`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.

Parameters: **func** (*Callable[[BlueprintSetupState], None]*) –
Return type: None

`register(app, options)`

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** (*App*) – The application this blueprint is being registered with.
- **options** (*dict[str, t.Any]*) – Keyword arguments forwarded from **`register_blueprint()`**.

Return type: None

▶ *Changelog*

`register_blueprint(blueprint, **options)`

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

▶ *Changelog*

Parameters:
- **blueprint** (*Blueprint*) –
- **options** (*Any*) –

Return type: None

`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*

Parameters:
- **code_or_exception** (*type[Exception]* | *int*) –
- **f** (*ft.ErrorHandlerCallable*) –

Return type: None

`route(rule, **options)`

Decorate a view function to register it with the given URL rule and options. Calls **`add_url_rule()`**, which has more details about the implementation.

```
@app.route("/")
def index():
    return "Hello, World!"
```

See URL Route Registrations.

The endpoint name for the route defaults to the name of the view function if the `endpoint` parameter isn't passed.

The `methods` parameter defaults to `["GET"]`. `HEAD` and `OPTIONS` are added automatically.

Parameters:
- **rule** (*str*) – The URL rule string.
- **options** (*Any*) – Extra options passed to the **`Rule`** object.

Return type: *Callable*[[*T_route*], *T_route*]

property **static_folder**: *str* | *None*

The absolute path to the configured static folder. `None` if no static folder is set.

property **static_url_path**: *str* | *None*

The URL prefix that the static route will be accessible from.

If it was not configured during init, it is derived from **`static_folder`**.

teardown_app_request(*f*)

Like **`teardown_request()`**, but after every request, not only those handled by the blueprint. Equivalent to **`Flask.teardown_request()`**.

Parameters: **f** (*T_teardown*) –
Return type: *T_teardown*

teardown_request(*f*)

Register a function to be called when the request context is popped. Typically this happens at the end of each request, but contexts may be pushed manually as well during testing.

```
with app.test_request_context():
    ...
```

When the `with` block exits (or `ctx.pop()` is called), the teardown functions are called just before the request context is made inactive.

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.

Teardown functions must avoid raising exceptions. If they execute code that might fail they must surround that code with a `try/except` block an v: 3.0.x errors.

The return values of teardown functions are ignored.

This is available on both app and blueprint objects. When used on an app, this executes after every request. When used on a blueprint, this executes after every request that the blueprint handles. To register with a blueprint and execute after every request, use **`Blueprint.teardown_app_request()`**.

Parameters: **f** (*T_teardown*) –
Return type: *T_teardown*

`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.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect every request, use **`Blueprint.app_url_defaults()`**.

Parameters: **f** (*T_url_defaults*) –
Return type: *T_url_defaults*

`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.

This is available on both app and blueprint objects. When used on an app, this is called for every request. When used on a blueprint, this is called for requests that the blueprint handles. To register with a blueprint and affect every request, use **`Blueprint.app_url_value_preprocessor()`**.

Parameters: **f** (*T_url_value_preprocessor*) –
Return type: *T_url_value_preprocessor*

`import_name`

The name of the package or module that this object belongs to. Do n
this once it is set by the constructor.

template_folder

The path to the templates folder, relative to **root_path**, to add to the template loader. None if templates should not be added.

root_path

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

view_functions: *dict[str, ft.RouteCallable]*

A dictionary mapping endpoint names to view functions.

To register a view function, use the **route()** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

error_handler_spec: *dict[ft.AppOrBlueprintKey, dict[int | None, dict[type[Exception], ft.ErrorHandlerCallable]]]*

A data structure of registered error handlers, in the format `{scope: {code: {class: handler}}}`. The `scope` key is the name of a blueprint the handlers are active for, or `None` for all requests. The `code` key is the HTTP status code for `HTTPException`, or `None` for other exceptions. The innermost dictionary maps exception classes to handler functions.

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

This data structure is internal. It should not be modified directly and its format may change at any time.

before_request_funcs: *dict[ft.AppOrBlueprintKey, list[ft.BeforeRequestCallable]]*

A data structure of functions to call at the beginning of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **before_request()** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

after_request_funcs: *dict[ft.AppOrBlueprintKey, list[ft.AfterRequestCallable[t.Any]]]*

A data structure of functions to call at the end of each request, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint tions are active for, or `None` for all requests.

To register a function, use the **`after_request()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`teardown_request_funcs`: *dict[ft.AppOrBlueprintKey, list[ft.TeardownCallable]]*

A data structure of functions to call at the end of each request even if an exception is raised, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`teardown_request()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`template_context_processors`: *dict[ft.AppOrBlueprintKey, list[ft.TemplateContextProcessorCallable]]*

A data structure of functions to call to pass extra context values when rendering templates, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`context_processor()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`url_value_preprocessors`: *dict[ft.AppOrBlueprintKey, list[ft.URLValuePreprocessorCallable]]*

A data structure of functions to call to modify the keyword arguments passed to the view function, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`url_value_preprocessor()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

`url_default_functions`: *dict[ft.AppOrBlueprintKey, list[ft.URLDefaultCallable]]*

A data structure of functions to call to modify the keyword arguments when generating URLs, in the format `{scope: [functions]}`. The `scope` key is the name of a blueprint the functions are active for, or `None` for all requests.

To register a function, use the **`url_defaults()`** decorator.

This data structure is internal. It should not be modified directly and its format may change at any time.

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.

Parameters:
- **environ** (*WSGIEnvironment*) –
- **populate_request** (*bool*) –
- **shallow** (*bool*) –

url_rule: *Rule | None = 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*

view_args: *dict[str, t.Any] | None = None*

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

routing_exception: *HTTPException | None = 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.

property **max_content_length**: *int | None*

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

property **endpoint**: *str | None*

The endpoint that matched the request URL.

This will be `None` if matching failed or has not been performed yet.

This in combination with **`view_args`** can be used to reconstruct the same URL or a modified URL.

property **blueprint**: *str* | *None*

The registered name of the current blueprint.

This will be `None` if the endpoint is not part of a blueprint, or if URL matching failed or has not been performed yet.

This does not necessarily match the name the blueprint was created with. It may have been nested, or registered with a different name.

property **blueprints**: *list*[*str*]

The registered names of the current blueprint upwards through parent blueprints.

This will be an empty list if there is no current blueprint, or if URL matching failed.

▶ *Changelog*

on_json_loading_failed(*e*)

Called if **`get_json()`** 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 **`BadRequest`**.

Parameters: **e** (*ValueError* | *None*) – If parsing failed, this is the exception. It will be `None` if the content type wasn't `application/json`.

Return type: *Any*

▶ *Changelog*

property **accept_charsets**: *CharsetAccept*

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

property **accept_encodings**: *Accept*

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`**.

property **accept_languages**: *LanguageAccept* v: 3.0.x

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

property **accept_mimetypes**: *MIMEAccept*

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

access_control_request_headers

Sent with a preflight request to indicate which headers will be sent with the cross origin request. Set **access_control_allow_headers** on the response to indicate which headers are allowed.

access_control_request_method

Sent with a preflight request to indicate which method will be used for the cross origin request. Set **access_control_allow_methods** on the response to indicate which methods are allowed.

property **access_route**: *list[str]*

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 the last argument. This works like the **responder()** decorator but the function is passed the request object as the last 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** (*t.Callable[[Request], WSGIApplication]*) – the WSGI callable to decorate
Returns:	a new WSGI callable
Return type:	WSGIApplication

property **args**: *MultiDict[str, str]*

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.

▶ *Changelog*

v: 3.0.x

property **authorization**: *Authorization* | *None*

The `Authorization` header parsed into an **`Authorization`** object. `None` if the header is not present.

▶ *Changelog*

property **base_url**: *str*

Like **`url`** but without the query string.

property **cache_control**: *RequestCacheControl*

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*

Return type: None

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*

property **content_length**: *int* | *None*

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.

property **cookies**: *ImmutableMultiDict[str, str]*

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

property **data**: *bytes*

The raw data read from **stream**. Will be empty if the request represents form data.

To get the raw data even if it represents form data, use **get_data()**.

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.

▶ *Changelog*

dict_storage_class

alias of **ImmutableMultiDict**

property **files**: *ImmutableMultiDict[str, FileStorage]*

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.

property **form**: *ImmutableMultiDict[str, str]*

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 i

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 **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
Parameters:	• **args** (*Any*) –
	• **kwargs** (*Any*) –
Return type:	*Request*

property full_path: *str*

Requested path, 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 bytes object. 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 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 string.

▶ *Changelog*

Parameters:
- **cache** (*bool*) –
- **as_text** (*bool*) –
- **parse_form_data** (*bool*) –

Return type: bytes | str

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

Parse **`data`** as JSON.

If the mimetype does not indicate JSON (*application/json*, see **`is_json`**), or parsing fails, **`on_json_loading_failed()`** is called and its return value is used as the return value. By default this raises a 415 Unsupported Media Type resp.

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

Return type: *Any* | None

▶ *Changelog*

property `host`: *str*

The host name the request was made to, including the port if it's non-standard. Validated with **`trusted_hosts`**.

property `host_url`: *str*

The request URL scheme and host only.

property `if_match`: *ETags*

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

Return type: **`ETags`**

property `if_modified_since`: *datetime* | *None*

The parsed `If-Modified-Since` header as a datetime object.

▶ *Changelog*

property **if_none_match**: *ETags*

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

Return type: **ETags**

property **if_range**: *IfRange*

The parsed `If-Range` header.

▶ *Changelog*

property **if_unmodified_since**: *datetime* | *None*

The parsed `If-Unmodified-Since` header as a datetime object.

▶ *Changelog*

input_stream

The raw WSGI input stream, without any safety checks.

This is dangerous to use. It does not guard against infinite streams or reading past **content_length** or **max_content_length**.

Use **stream** instead.

property **is_json**: *bool*

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

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.

property **is_secure**: *bool*

`True` if the request was made with a secure protocol (HTTPS or WSS)

property **json**: *Any* | *None*

The parsed JSON data if **mimetype** indicates JSON (*application/json*, see **is_json**).

Calls **get_json()** with default arguments.

If the request content type is not `application/json`, this will raise a 415 Unsupported Media Type error.

▶ *Changelog*

list_storage_class

alias of **ImmutableList**

make_form_data_parser()

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

▶ *Changelog*

Return type: *FormDataParser*

max_form_memory_size: *int* | *None* = *None*

the maximum form field size. This is forwarded to the form data parsing function (**parse_form_data()**). When set and the **form** or **files** attribute is accessed and the data in memory for post data is longer than the specified value a **RequestEntityTooLarge** exception is raised.

▶ *Changelog*

max_form_parts = *1000*

The maximum number of multipart parts to parse, passed to **form_data_parser_class**. Parsing form data with more than this many parts will raise **RequestEntityTooLarge**.

▶ *Changelog*

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.

property mimetype: *str*

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'`.

property **mimetype_params**: *dict[str, str]*

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

origin

The host that the request originated from. Set **access_control_allow_origin** on the response to indicate which origins are allowed.

parameter_storage_class

alias of **ImmutableMultiDict**

property **pragma**: *HeaderSet*

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.

property **range**: *Range* | *None*

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_user

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

property **root_url**: *str*

The request URL scheme, host, and root path. This is the root that the application is accessed from.

property **script_root**: *str*

Alias for **self.root_path**. `environ["SCRIPT_ROOT"]` without a trai

property **stream**: *IO[bytes]*

The WSGI input stream, with safety checks. This stream can only be consumed once.

Use **`get_data()`** to get the full data as bytes or text. The **`data`** attribute will contain the full bytes only if they do not represent form data. The **`form`** attribute will contain the parsed form data in that case.

Unlike **`input_stream`**, this stream guards against infinite streams or reading past **`content_length`** or **`max_content_length`**.

If `max_content_length` is set, it can be enforced on streams if `wsgi.input_terminated` is set. Otherwise, an empty stream is returned.

If the limit is reached before the underlying stream is exhausted (such as a file that is too large, or an infinite stream), the remaining contents of the stream cannot be read safely. Depending on how the server handles this, clients may show a "connection reset" failure instead of seeing the 413 response.

▶ *Changelog*

`trusted_hosts`: *list[str] | None = None*

Valid host names when handling requests. By default all hosts are trusted, which means that whatever the client says the host is will be accepted.

Because `Host` and `X-Forwarded-Host` headers can be set to any value by a malicious client, it is recommended to either set this property or implement similar validation in the proxy (if the application is being run behind one).

▶ *Changelog*

property **`url`**: *str*

The full request URL with the scheme, host, root path, path, and query string.

property **`url_root`**: *str*

Alias for **`root_url`**. The URL with scheme, host, and root path. For example, `https://example.com/app/`.

property **`user_agent`**: *UserAgent*

The user agent. Use `user_agent.string` to get the header value. Set **`user_agent_class`** to a subclass of **`UserAgent`** to provide parsing for the other properties or other extended data.

▶ *Changelog*

`user_agent_class`

alias of **`UserAgent`**

property **values**: *CombinedMultiDict[str, str]*

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

For GET requests, only `args` are present, not `form`.

▶ *Changelog*

property **want_form_data_parsed**: *bool*

`True` if the request method carries content. By default this is true if a `Content-Type` is sent.

▶ *Changelog*

environ: *WSGIEnvironment*

The WSGI environment containing HTTP headers and information from the WSGI server.

shallow: *bool*

Set when creating the request object. If `True`, reading from the request body will cause a `RuntimeException`. Useful to prevent modifying the stream from middleware.

method

The method the request was made with, such as `GET`.

scheme

The URL scheme of the protocol the request used, such as `https` or `wss`.

server

The address of the server. `(host, port)`, `(path, None)` for unix sockets, or `None` if not known.

root_path

The prefix that the application is mounted under, without a trailing slash. **`path`** comes after this.

path

The path part of the URL after **`root_path`**. This is the path used for routing within the application.

query_string

The part of the URL after the "?". This is the raw value, use **args** for the parsed values.

headers

The headers received with the request.

remote_addr

The address of the client sending the request.

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**.

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.

▶ *Changelog*

Parameters:
- **response** (*Iterable[str]* | *Iterable[bytes]*) –
- **status** (*int* | *str* | *HTTPStatus* | *None*) –
- **headers** (*Headers*) –
- **mimetype** (*str* | *None*) –
- **content_type** (*str* | *None*) –
- **direct_passthrough** (*bool*) –

default_mimetype: *str* | *None* = *'text/html'*

the default mimetype if none is provided.

autocorrect_location_header = *False*

If a redirect `Location` header is a relative URL, make it an absolute URL, including scheme and domain.

▶ *Changelog*

property **max_cookie_size**: *int*

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

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

accept_ranges

The `Accept-Ranges` header. Even though the name would indicate that multiple values are supported, it must be one string token only.

The values `'bytes'` and `'none'` are common.

▶ *Changelog*

property **access_control_allow_credentials**: *bool*

Whether credentials can be shared by the browser to JavaScript code. As part of the preflight request it indicates whether credentials can be used on the cross origin request.

access_control_allow_headers

Which headers can be sent with the cross origin request.

access_control_allow_methods

Which methods can be used for the cross origin request.

access_control_allow_origin

The origin or '*' for any origin that may make cross origin requests.

access_control_expose_headers

Which headers can be shared by the browser to JavaScript code.

access_control_max_age

The maximum age in seconds the access control settings can be cached for.

add_etag(*overwrite=False, weak=False*)

Add an etag for the current response if there is none yet.

▶ *Changelog*

Parameters: • **overwrite** (*bool*) –

- **weak** (*bool*) –

Return type: None

`age`

The Age response-header field conveys the sender's estimate of the amount of time since the response (or its revalidation) was generated at the origin server.

Age values are non-negative decimal integers, representing time in seconds.

property `allow`*:* *HeaderSet*

The Allow entity-header field lists the set of methods supported by the resource identified by the Request-URI. The purpose of this field is strictly to inform the recipient of valid methods associated with the resource. An Allow header field MUST be present in a 405 (Method Not Allowed) response.

`automatically_set_content_length` = *True*

Should this response object automatically set the content-length header if possible? This is true by default.

▶ *Changelog*

property `cache_control`*:* *ResponseCacheControl*

The Cache-Control general-header field is used to specify directives that MUST be obeyed by all caching mechanisms along the request/response chain.

`calculate_content_length`()

Returns the content length if available or `None` otherwise.

Return type: int | None

`call_on_close`(*func*)

Adds a function to the internal list of functions that should be called as part of closing down the response. Since 0.7 this function also returns the function that was passed so that this can be used as a decorator.

▶ *Changelog*

Parameters: **func** (*Callable[[], Any]*) –
Return type: *Callable*[[], *Any*]

`close`()

Close the wrapped response if possible. You can also use the object i… statement which will automatically close it.

▶ *Changelog*

Return type: None

`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.

property `content_language`*:* *HeaderSet*

The Content-Language entity-header field describes the natural language(s) of the intended audience for the enclosed entity. Note that this might not be equivalent to all the languages used within the entity-body.

`content_length`

The Content-Length entity-header field indicates the size of the entity-body, in decimal number of OCTETs, sent to the recipient 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_location`

The Content-Location entity-header field MAY be used to supply the resource location for the entity enclosed in the message when that entity is accessible from a location separate from the requested resource's URI.

`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.)

property `content_range`*:* *ContentRange*

The `Content-Range` header as a **`ContentRange`** object. Available even if the header is not set.

▶ *Changelog*

property `content_security_policy`*:* *ContentSecurityPolicy*

The `Content-Security-Policy` header as a **`ContentSecurityPolicy`** object. Available even if the header is not set.

The Content-Security-Policy header adds an additional layer of security to help detect and mitigate certain types of attacks.

property **content_security_policy_report_only**: *ContentSecurityPolicy*

The `Content-Security-policy-report-only` header as a **`ContentSecurityPolicy`** object. Available even if the header is not set.

The Content-Security-Policy-Report-Only header adds a csp policy that is not enforced but is reported thereby helping detect certain types of attacks.

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.

cross_origin_embedder_policy

Prevents a document from loading any cross-origin resources that do not explicitly grant the document permission. Values must be a member of the **`werkzeug.http.COEP`** enum.

cross_origin_opener_policy

Allows control over sharing of browsing context group with cross-origin documents. Values must be a member of the **`werkzeug.http.COOP`** enum.

property **data**: *bytes* | *str*

A descriptor that calls **`get_data()`** and **`set_data()`**.

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.

▶ *Changelog*

default_status = *200*

the default status if none is provided.

delete_cookie(*key*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *samesite=None*)

Delete a cookie. Fails silently if key doesn't exist.

Parameters:
- **key** (*str*) – the key (name) of the cookie to be de[illegible]
- **path** (*str* | *None*) – if the cookie that should be [illegible] was limited to a path, the path has to be defined here.

- **domain** (*str* | *None*) – if the cookie that should be deleted was limited to a domain, that domain has to be defined here.
- **secure** (*bool*) – If `True`, the cookie will only be available via HTTPS.
- **httponly** (*bool*) – Disallow JavaScript access to the cookie.
- **samesite** (*str* | *None*) – Limit the scope of the cookie to only be attached to requests that are "same-site".

Return type: None

expires

The Expires entity-header field gives the date/time after which the response is considered stale. A stale cache entry may not normally be returned by a cache.

▶ *Changelog*

classmethod force_type(*response*, *environ=None*)

Enforce that the WSGI response is a response object of the current type. Werkzeug will use the **Response** internally in many situations like the exceptions. If you call **get_response()** on an exception you will get back a regular **Response** object, even if you are using a custom subclass.

This method can enforce a given response type, and it will also convert arbitrary WSGI callables into response objects if an environ is provided:

```
# convert a Werkzeug response object into an instance of the
# MyResponseClass subclass.
response = MyResponseClass.force_type(response)

# convert any WSGI application into a response object
response = MyResponseClass.force_type(response, environ)
```

This is especially useful if you want to post-process responses in the main dispatcher and use functionality provided by your subclass.

Keep in mind that this will modify response objects in place if possible!

Parameters:
- **response** (*Response*) – a response object or wsgi application.
- **environ** (*WSGIEnvironment* | *None*) – a WSGI environment object.

Returns: a response object.

Return type: Response

freeze()

Make the response object ready to be pickled. Does the following:

- Buffer the response into a list, ignoring **implicity_sequence_conversion** and **direct_passthrough**.
- Set the `Content-Length` header.
- Generate an `ETag` header if one is not already set.

▶ *Changelog*

Return type: None

classmethod **from_app**(*app*, *environ*, *buffered=False*)

Create a new response object from an application output. This works best if you pass it an application that returns a generator all the time. Sometimes applications may use the `write()` callable returned by the `start_response` function. This tries to resolve such edge cases automatically. But if you don't get the expected output you should set `buffered` to `True` which enforces buffering.

Parameters:
- **app** (*WSGIApplication*) – the WSGI application to execute.
- **environ** (*WSGIEnvironment*) – the WSGI environment to execute against.
- **buffered** (*bool*) – set to `True` to enforce buffering.

Returns: a response object.

Return type: Response

get_app_iter(*environ*)

Returns the application iterator for the given environ. Depending on the request method and the current status code the return value might be an empty response rather than the one from the response.

If the request method is `HEAD` or the status code is in a range where the HTTP specification requires an empty response, an empty iterable is returned.

▶ *Changelog*

Parameters: **environ** (*WSGIEnvironment*) – the WSGI environment of the request.

Returns: a response iterable.

Return type: t.Iterable[bytes]

get_data(*as_text=False*)

The string representation of the response body. Whenever you call this property the response iterable is encoded and flattened. This can lead to unwa

havior if you stream big data.

This behavior can be disabled by setting **`implicit_sequence_conversion`** to `False`.

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

▶ *Changelog*

Parameters: **as_text** (*bool*) –
Return type: bytes | str

`get_etag()`

Return a tuple in the form `(etag, is_weak)`. If there is no ETag the return value is `(None, None)`.

Return type: tuple[str, bool] | tuple[None, None]

`get_json(force=False, silent=False)`

Parse **`data`** as JSON. Useful during testing.

If the mimetype does not indicate JSON (*application/json*, see **`is_json`**), this returns `None`.

Unlike **`Request.get_json()`**, the result is not cached.

Parameters:
- **force** (*bool*) – Ignore the mimetype and always try to parse JSON.
- **silent** (*bool*) – Silence parsing errors and return `None` instead.

Return type: *Any* | None

`get_wsgi_headers(environ)`

This is automatically called right before the response is started and returns headers modified for the given environment. It returns a copy of the headers from the response with some modifications applied if necessary.

For example the location header (if present) is joined with the root URL of the environment. Also the content length is automatically set to zero here for certain status codes.

▶ *Changelog*

Parameters: **environ** (*WSGIEnvironment*) – the WSGI environment of the request.
Returns: returns a new **`Headers`** object.
Return type: Headers

`get_wsgi_response(environ)`

Returns the final WSGI response as tuple. The first item in the tuple is the application iterator, the second the status and the third the list of headers. The response returned is created specially for the given environment. For example if the request method in the WSGI environment is `'HEAD'` the response will be empty and only the headers and status code will be present.

▶ *Changelog*

Parameters:	**environ** (*WSGIEnvironment*) – the WSGI environment of the request.
Returns:	an `(app_iter, status, headers)` tuple.
Return type:	tuple[t.Iterable[bytes], str, list[tuple[str, str]]]

`implicit_sequence_conversion = True`

if set to `False` accessing properties on the response object will not try to consume the response iterator and convert it into a list.

▶ *Changelog*

property `is_json`*:* *bool*

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

property `is_sequence`*:* *bool*

If the iterator is buffered, this property will be `True`. A response object will consider an iterator to be buffered if the response attribute is a list or tuple.

▶ *Changelog*

property `is_streamed`*:* *bool*

If the response is streamed (the response is not an iterable with a length information) this property is `True`. In this case streamed means that there is no information about the number of iterations. This is usually `True` if a generator is passed to the response object.

This is useful for checking before applying some sort of post filtering that should not take place for streamed responses.

`iter_encoded()`

Iter the response encoded with the encoding of the response. If the response object is invoked as WSGI application the return value of this metho

application iterator unless **`direct_passthrough`** was activated.

Return type: *Iterator*[bytes]

property **json**: *Any* | *None*

The parsed JSON data if **mimetype** indicates JSON (*application/json*, see **is_json**).

Calls **get_json()** with default arguments.

last_modified

The Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified.

▶ *Changelog*

location

The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.

make_conditional(*request_or_environ, accept_ranges=False, complete_length=None*)

Make the response conditional to the request. This method works best if an etag was defined for the response already. The `add_etag` method can be used to do that. If called without etag just the date header is set.

This does nothing if the request method in the request or environ is anything but GET or HEAD.

For optimal performance when handling range requests, it's recommended that your response data object implements `seekable`, `seek` and `tell` methods as described by **io.IOBase**. Objects returned by **wrap_file()** automatically implement those methods.

It does not remove the body of the response because that's something the **__call__()** function does for us automatically.

Returns self so that you can do `return resp.make_conditional(req)` but modifies the object in-place.

Parameters:
- **request_or_environ** (*WSGIEnvironment* | *Request*) – a request object or WSGI environment to be used to make the response conditional against.
- **accept_ranges** (*bool* | *str*) – This parameter di… value of `Accept-Ranges` header. If `False` (defaul…), the

header is not set. If `True`, it will be set to `"bytes"`. If it's a string, it will use this value.

- **complete_length** (*int* | *None*) – Will be used only in valid Range Requests. It will set `Content-Range` complete length value and compute `Content-Length` real value. This parameter is mandatory for successful Range Requests completion.

Raises: **`RequestedRangeNotSatisfiable`** if `Range` header could not be parsed or satisfied.

Return type: Response

▶ *Changelog*

`make_sequence()`

Converts the response iterator in a list. By default this happens automatically if required. If `implicit_sequence_conversion` is disabled, this method is not automatically called and some properties might raise exceptions. This also encodes all the items.

▶ *Changelog*

Return type: None

property **mimetype**: *str* | *None*

The mimetype (content type without charset etc.)

property **mimetype_params**: *dict[str, str]*

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

▶ *Changelog*

property **retry_after**: *datetime* | *None*

The Retry-After response-header field can be used with a 503 (Service Unavailable) response to indicate how long the service is expected to be unavailable to the requesting client.

Time in seconds until expiration or date.

▶ *Changelog*

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

Sets a cookie.

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

Parameters:
- **key** (*str*) – the key (name) of the cookie to be set.
- **value** (*str*) – the value of the cookie.
- **max_age** (*timedelta* | *int* | *None*) – should be a number of seconds, or `None` (default) if the cookie should last only as long as the client's browser session.
- **expires** (*str* | *datetime* | *int* | *float* | *None*) – should be a `datetime` object or UNIX timestamp.
- **path** (*str* | *None*) – limits the cookie to a given path, per default it will span the whole domain.
- **domain** (*str* | *None*) – 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** (*bool*) – If `True`, the cookie will only be available via HTTPS.
- **httponly** (*bool*) – Disallow JavaScript access to the cookie.
- **samesite** (*str* | *None*) – Limit the scope of the cookie to only be attached to requests that are "same-site".

Return type: None

`set_data(value)`

Sets a new string as response. The value must be a string or bytes. If a string is set it's encoded to the charset of the response (utf-8 by default).

▶ *Changelog*

Parameters: **value** (*bytes* | *str*) –
Return type: None

`set_etag(etag, weak=False)`

Set the etag, and override the old one if there was one.

Parameters:
- **etag** (*str*) –
- **weak** (*bool*) –

Return type: None

property `status: str`

v: 3.0.x

The HTTP status code as a string.

property **status_code**: *int*

The HTTP status code as a number.

property **stream**: *ResponseStream*

The response iterable as write-only stream.

property **vary**: *HeaderSet*

The Vary field value indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.

property **www_authenticate**: *WWWAuthenticate*

The `WWW-Authenticate` header parsed into a **`WWWAuthenticate`** object. Modifying the object will modify the header value.

This header is not set by default. To set this header, assign an instance of **`WWWAuthenticate`** to this attribute.

```
response.www_authenticate = WWWAuthenticate(
    "basic", {"realm": "Authentication Required"}
)
```

Multiple values for this header can be sent to give the client multiple options. Assign a list to set multiple headers. However, modifying the items in the list will not automatically update the header values, and accessing this attribute will only ever return the first value.

To unset this header, assign `None` or use `del`.

▶ *Changelog*

response: *t.Iterable[str] | t.Iterable[bytes]*

The response body to send as the WSGI iterable. A list of strings or bytes represents a fixed-length response, any other iterable is a streaming response. Strings are encoded to bytes as UTF-8.

Do not set to a plain string or bytes, that will cause sending the response to be very inefficient as it will iterate one byte at a time.

direct_passthrough

Pass the response body directly through as the WSGI iterable. This can be used when the body is a binary file or other iterator of bytes, to skip some unnecessary checks. Use **`send_file()`** instead of setting this manually.

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 of 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.

v: 3.0.x

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()
```

Multiple requests with the same session may be sent and handled concurrently. When implementing a new session interface, consider whether reads or writes to the backing store must be synchronized. There is no guarantee on the order in which the session for each request is opened or saved, it will occur in the order that requests begin and end processing.

▶ *Changelog*

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**

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.

v: 3.0.x

▶ *Changelog*

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.

Parameters: **app** (*Flask*) –
Return type: NullSession

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.

Parameters: **obj** (*object*) –
Return type: bool

get_cookie_name(*app*)

The name of the session cookie.
Uses``app.config["SESSION_COOKIE_NAME"]``.

Parameters: **app** (*Flask*) –
Return type: str

get_cookie_domain(*app*)

The value of the `Domain` parameter on the session cookie. If not set, browsers will only send the cookie to the exact domain it was set from. Otherwise, they will send it to any subdomain of the given value as well.

Uses the **SESSION_COOKIE_DOMAIN** config.

▶ *Changelog*

Parameters: **app** (*Flask*) –
Return type: str | None

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`.

Parameters: **app** (*Flask*) –

Return type: str

`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.

Parameters: **app** (*Flask*) –
Return type: bool

`get_cookie_secure(app)`

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

Parameters: **app** (*Flask*) –
Return type: bool

`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.

Parameters: **app** (*Flask*) –
Return type: str | None

`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.

Parameters:
- **app** (*Flask*) –
- **session** (*SessionMixin*) –

Return type: datetime | None

`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*

Parameters:
- **app** (*Flask*) –
- **session** (*SessionMixin*) –

Return type: bool

open_session(*app*, *request*)

This is called at the beginning of each request, after pushing the request context, before matching the URL.

This must return an object which implements a dictionary-like interface as well as the **SessionMixin** interface.

This will return `None` to indicate that loading failed in some way that is not immediately an error. The request context will fall back to using **make_null_session()** in this case.

Parameters:
- **app** (*Flask*) –
- **request** (*Request*) –

Return type: SessionMixin | None

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

This is called at the end of each request, after generating a response, before removing the request context. It is skipped if **is_null_session()** returns `True`.

Parameters:
- **app** (*Flask*) –
- **session** (*SessionMixin*) –
- **response** (*Response*) –

Return type: None

class flask.sessions.SecureCookieSessionInterface

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

salt = '*cookie-session*'

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

static digest_method(*string=b''*)

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

Parameters: **string** (*bytes*) –

Return type: *Any*

key_derivation = '*hmac*'

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

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

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

session_class

alias of **SecureCookieSession**

open_session(*app*, *request*)

This is called at the beginning of each request, after pushing the request context, before matching the URL.

This must return an object which implements a dictionary-like interface as well as the **SessionMixin** interface.

This will return `None` to indicate that loading failed in some way that is not immediately an error. The request context will fall back to using **make_null_session()** in this case.

Parameters:
- **app** (*Flask*) –
- **request** (*Request*) –

Return type: SecureCookieSession | None

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

This is called at the end of each request, after generating a response, before removing the request context. It is skipped if **is_null_session()** returns `True`.

Parameters:
- **app** (*Flask*) –
- **session** (*SessionMixin*) –
- **response** (*Response*) –

Return type: None

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`.

Parameters: **initial** (*t.Any*) –

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`.

accessed = *False*

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

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

Return the value for key if key is in the dictionary, else default.

Parameters:
- **key** (*str*) –
- **default** (*Any*) –

Return type: *Any*

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

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

Parameters:
- **key** (*str*) –
- **default** (*Any*) –

Return type: *Any*

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.

Parameters: **initial** (*t.Any*) –

clear() → None. Remove all items from D.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *NoReturn*

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *NoReturn*

popitem(**args*, ***kwargs*)

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *NoReturn*

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *NoReturn*

setdefault(**args*, ***kwargs*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *NoReturn*

class flask.sessions.**SessionMixin**

Expands a basic dictionary with session attributes.

property **permanent**: *bool*

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

modified = *True*

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

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`.

Notice:

The **`PERMANENT_SESSION_LIFETIME`** config can be an integer or `timedelta`. The **`permanent_session_lifetime`** attribute is always a `timedelta`.

Test Client

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

Works like a regular Werkzeug test client but has knowledge about Flask to defer the cleanup of the request context until the end of a `with` block. 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.

Parameters:
- **args** (*t.Any*) –
- **kwargs** (*t.Any*) –

`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.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *Iterator*[*SessionMixin*]

`open(*args, buffered=False, follow_redirects=False, **kwargs)`

Generate an environ dict from the given arguments, make a request to the application using it, and return the response.

Parameters:
- **args** (*t.Any*) – Passed to **`EnvironBuilder`** to create the environ for the request. If a single arg is passed, it can be an existing **`EnvironBuilder`** or an environ dict.
- **buffered** (*bool*) – Convert the iterator returned by the app into a list. If the iterator has a `close()` method, it is called automatically.
- **follow_redirects** (*bool*) – Make additional requests to follow HTTP redirects until a non-redirect status is returned. **`TestResponse.history`** lists the intermediate responses.
- **kwargs** (*t.Any*) –

Return type: TestResponse

▶ *Changelog*

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 Running Commands with the CLI Runner.

Parameters:
- **app** (*Flask*) –
- **kwargs** (*t.Any*) –

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

Invokes a CLI command in an isolated environment. See **CliRunner.invoke** for full method documentation. See Running Commands with the CLI Runner 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** (*Any*) – Command object to invoke. Default is the app's **cli** group.
- **args** (*Any*) – List of strings to invoke the command with.
- **kwargs** (*Any*) –

Returns: a **Result** object.

Return type: *Any*

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. For example, a `before_request` function could load a user object from a session id, then set `g.user` to be used in the view function.

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** (*str*) – Name of attribute to get.
- **default** (*Any* | *None*) – Value to return if the attribute is not present.

Return type: *Any*

▶ *Changelog*

pop(*name*, *default=_sentinel*)

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

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

Return type: *Any*

▶ *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** (*str*) – Name of attribute to get.
- **default** (*Any*) – Value to set and return if the attribute is not present.

Return type: *Any*

▶ *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*

Return type: bool

`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. The current session is also included in the copied request context.

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 or
        # flask.session like you would otherwise in the view functio
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

▶ *Changelog*

Parameters: **f** (*F*) –
Return type: *F*

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*

Return type: bool

flask.**url_for**(*endpoint*, *, *_anchor=None*, *_method=None*, *_scheme=None*, *_external=None*, ***values*)

Generate a URL to the given endpoint with the given values.

This requires an active request or application context, and calls **current_app.url_for()**. See that method for full documentation.

Parameters:
- **endpoint** (*str*) – The endpoint name associated with the URL to generate. If this starts with a `.`, the current blueprint name (if any) will be used.
- **_anchor** (*str* | *None*) – If given, append this as `#anchor` to the URL.

- **_method** (*str* | *None*) – If given, generate the URL associated with this method for the endpoint.
- **_scheme** (*str* | *None*) – If given, the URL will have this scheme if it is external.
- **_external** (*bool* | *None*) – If given, prefer the URL to be internal (False) or require it to be external (True). External URLs include the scheme and domain. When not in an active request, URLs are external by default.
- **values** (*Any*) – Values to use for the variable parts of the URL rule. Unknown keys are appended as query string arguments, like `?a=b&c=d`.

Return type: str

▶ *Changelog*

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

Raise an **`HTTPException`** for the given status code.

If **`current_app`** is available, it will call its **`aborter`** object, otherwise it will use **`werkzeug.exceptions.abort()`**.

Parameters:
- **code** (*int* | *Response*) – The status code for the exception, which must be registered in `app.aborter`.
- **args** (*Any*) – Passed to the exception.
- **kwargs** (*Any*) – Passed to the exception.

Return type: *NoReturn*

▶ *Changelog*

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

Create a redirect response object.

If **`current_app`** is available, it will use its **`redirect()`** method, otherwise it will use **`werkzeug.utils.redirect()`**.

Parameters:
- **location** (*str*) – The URL to redirect to.
- **code** (*int*) – The status code for the redirect.
- **Response** (*type[Response]* | *None*) – The response class to use. Not used when `current_app` is active, which uses `app.response_class`.

Return type: *Response*

▶ *Changelog*

`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*

Parameters: **args** (*t.Any*) –
Return type: Response

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

Executes a function after this request. This is useful to modify response c… function is passed the response object and has to return the same or a ne… 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*

Parameters: **f** (*Callable[[Any], Any] | Callable[[Any], Awaitable[Any]]*) –
Return type: *Callable*[[*Any*], *Any*] | *Callable*[[*Any*], *Awaitable*[*Any*]]

`flask.`**`send_file`**`(`*`path_or_file, mimetype=None, as_attachment=False, download_name=None, conditional=True, etag=True, last_modified=None, max_age=None`*`)`

Send the contents of a file to the client.

The first argument can be a file path or a file-like object. Paths are preferred in most cases because Werkzeug can manage the file and get extra information from the path. Passing a file-like object requires that the file is opened in binary mode, and is mostly useful when building a file in memory with **`io.BytesIO`**.

Never pass file paths provided by a user. The path is assumed to be trusted, so a user could craft a path to access a file you didn't intend. Use **`send_from_directory()`** to safely serve user-requested paths from within a directory.

If the WSGI server sets a `file_wrapper` in `environ`, it is used, otherwise Werkzeug's built-in wrapper is used. Alternatively, if the HTTP server supports `X-Sendfile`, configuring Flask with `USE_X_SENDFILE = True` will tell the server to send the given path, which is much more efficient than reading it in Python.

Parameters:

- **path_or_file** (*os.PathLike[t.AnyStr] | str | t.BinaryIO*) – The path to the file to send, relative to the current working directory if a relative path is given. Alternatively, a file-like object opened in binary mode. Make sure the file pointer is seeked to the start of the data.
- **mimetype** (*str* | *None*) – The MIME type to send for the file. If not provided, it will try to detect it from the file name.

- **as_attachment** (*bool*) – Indicate to a browser that it should offer to save the file instead of displaying it.
- **download_name** (*str* | *None*) – The default name browsers will use when saving the file. Defaults to the passed file name.
- **conditional** (*bool*) – Enable conditional and range responses based on request headers. Requires passing a file path and `environ`.
- **etag** (*bool* | *str*) – Calculate an ETag for the file, which requires passing a file path. Can also be a string to use instead.
- **last_modified** (*datetime* | *int* | *float* | *None*) – The last modified time to send for the file, in seconds. If not provided, it will try to detect it from the file path.
- **max_age** (*None* | (*int* | *t.Callable[[str | None], int | None]*)) – How long the client should cache the file, in seconds. If set, `Cache-Control` will be `public`, otherwise it will be `no-cache` to prefer conditional caching.

Return type: Response

▶ *Changelog*

flask.**send_from_directory**(*directory*, *path*, ***kwargs*)

Send a file from within a directory using **`send_file()`**.

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

This is a secure way to serve files from a folder, such as static files or uploads. Uses **`safe_join()`** to ensure the path coming from the client is not maliciously crafted to point outside the specified directory.

If the final path does not point to an existing regular file, raises a 404 **`NotFound`** error.

Parameters:
- **directory** (*os.PathLike[str]* | *str*) – The directory that `path` must be located under, relative to the current application's root path.
- **path** (*os.PathLike[str]* | *str*) – The path to the file to send, relative to `directory`.
- **kwargs** (*t.Any*) – Arguments to pass to **`send_file()`**.

Return type: Response

▶ *Changelog*

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** (*str*) – the message to be flashed.
- **category** (*str*) – 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.

Return type: None

`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** (*bool*) – set to `True` to also receive categories.
- **category_filter** (*Iterable[str]*) – filter of categories to limit return values. Only categories in the list will be returned.

Return type: list[str] | list[tuple[str, str]]

JSON Support

Flask uses Python's built-in **json** module for handling JSON by default. The JSON implementation can be changed by assigning a different provider to **flask.Flask.json_provider_class** or **flask.Flask.json**. The functions provided by `flask.json` will use methods on `app.json` if an app context is active.

Jinja's `|tojson` filter is configured to use the app's JSON provider. The filter marks the output with `|safe`. Use it to render data inside HTML `<script>` tags.

```
<script>
    const names = {{ names|tojson }};
    renderChart(names, {{ axis_data|tojson }});
</script>
```

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

Serialize the given arguments as JSON, and return a **Response** object with the `application/json` mimetype. A dict or list returned from a view will be converted to a JSON response automatically without needing to call this.

This requires an active request or application context, and calls **app.json.response()**.

In debug mode, the output is formatted with indentation to make it easier to read. This may also be controlled by the provider.

Either positional or keyword arguments can be given, not both. If no arguments are given, `None` is serialized.

Parameters:
- **args** (*t.Any*) – A single value to serialize, or multiple values to treat as a list to serialize.
- **kwargs** (*t.Any*) – Treat as a dict to serialize.

Return type: Response

▶ *Changelog*

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

Serialize data as JSON.

If **current_app** is available, it will use its **app.json.dumps()** method, otherwise it will use **json.dumps()**.

Parameters:
- **obj** (*Any*) – The data to serialize.
- **kwargs** (*Any*) – Arguments passed to the `dumps` implementation.

Return type: str

▶ *Changelog*

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

Serialize data as JSON and write to a file.

If **current_app** is available, it will use its **app.json.dump()** method, otherwise it will use **json.dump()**.

Parameters:
- **obj** (*Any*) – The data to serialize.
- **fp** (*IO[str]*) – A file opened for writing text. Should use the UTF-8 encoding to be valid JSON.
- **kwargs** (*Any*) – Arguments passed to the `dump` implementation.

Return type: None

▶ *Changelog*

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

Deserialize data as JSON.

If **current_app** is available, it will use its **app.json.loads()** method, otherwise it will use **json.loads()**.

Parameters:
- **s** (*str* | *bytes*) – Text or UTF-8 bytes.
- **kwargs** (*Any*) – Arguments passed to the `loads` implementation.

Return type: *Any*

▶ *Changelog*

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

Deserialize data as JSON read from a file.

If **current_app** is available, it will use its **app.json.load()** method, otherwise it will use **json.load()**.

Parameters:
- **fp** (*IO*) – A file opened for reading text or UTF-8 bytes.
- **kwargs** (*Any*) – Arguments passed to the `load` implementation.

Return type: *Any*

▶ *Changelog*

class flask.json.provider.**JSONProvider**(*app*)

A standard set of JSON operations for an application. Subclasses of this can be used to customize JSON behavior or use different JSON libraries.

To implement a provider for a specific library, subclass this base class and implement at least **`dumps()`** and **`loads()`**. All other methods have default implementations.

To use a different provider, either subclass `Flask` and set **`json_provider_class`** to a provider class, or set **`app.json`** to an instance of the class.

Parameters: **app** (*App*) – An application instance. This will be stored as a **`weakref.proxy`** on the **`_app`** attribute.

▶ *Changelog*

`dumps(obj, **kwargs)`

Serialize data as JSON.

Parameters:
- **obj** (*Any*) – The data to serialize.
- **kwargs** (*Any*) – May be passed to the underlying JSON library.

Return type: str

`dump(obj, fp, **kwargs)`

Serialize data as JSON and write to a file.

Parameters:
- **obj** (*Any*) – The data to serialize.
- **fp** (*IO[str]*) – A file opened for writing text. Should use the UTF-8 encoding to be valid JSON.
- **kwargs** (*Any*) – May be passed to the underlying JSON library.

Return type: None

`loads(s, **kwargs)`

Deserialize data as JSON.

Parameters:
- **s** (*str* | *bytes*) – Text or UTF-8 bytes.
- **kwargs** (*Any*) – May be passed to the underlying JSON library.

Return type: *Any*

`load(fp, **kwargs)`

Deserialize data as JSON read from a file.

Parameters:
- **fp** (*IO*) – A file opened for reading text or UTF-8 bytes.
- **kwargs** (*Any*) – May be passed to the underlyin
library.

Return type: *Any*

response(*args, **kwargs)

Serialize the given arguments as JSON, and return a **Response** object with the `application/json` mimetype.

The **jsonify()** function calls this method for the current application.

Either positional or keyword arguments can be given, not both. If no arguments are given, `None` is serialized.

Parameters:
- **args** (*t.Any*) – A single value to serialize, or multiple values to treat as a list to serialize.
- **kwargs** (*t.Any*) – Treat as a dict to serialize.

Return type: Response

class flask.json.provider.DefaultJSONProvider(*app*)

Provide JSON operations using Python's built-in **json** library. Serializes the following additional data types:

- **datetime.datetime** and **datetime.date** are serialized to **RFC 822** strings. This is the same as the HTTP date format.
- **uuid.UUID** is serialized to a string.
- **dataclasses.dataclass** is passed to **dataclasses.asdict()**.
- **Markup** (or any object with a `__html__` method) will call the `__html__` method to get a string.

Parameters: **app** (*App*) –

static default(*o*)

Apply this function to any object that **json.dumps()** does not know how to serialize. It should return a valid JSON type or raise a `TypeError`.

Parameters: **o** (*Any*) –
Return type: *Any*

ensure_ascii = *True*

Replace non-ASCII characters with escape sequences. This may be more compatible with some clients, but can be disabled for better performance and size.

sort_keys = *True*

Sort the keys in any serialized dicts. This may be useful for some caching situations, but can be disabled for better performance. When enabled, keys must all be strings, they are not converted before sorting.

compact: *bool* | *None* = *None*

If `True`, or `None` out of debug mode, the **`response()`** output will not add indentation, newlines, or spaces. If `False`, or `None` in debug mode, it will use a non-compact representation.

`mimetype` = *'application/json'*

The mimetype set in **`response()`**.

`dumps`(*obj, **kwargs*)

Serialize data as JSON to a string.

Keyword arguments are passed to **`json.dumps()`**. Sets some parameter defaults from the **`default`**, **`ensure_ascii`**, and **`sort_keys`** attributes.

Parameters:
- **obj** (*Any*) – The data to serialize.
- **kwargs** (*Any*) – Passed to **`json.dumps()`**.

Return type: str

`loads`(*s, **kwargs*)

Deserialize data as JSON from a string or bytes.

Parameters:
- **s** (*str* | *bytes*) – Text or UTF-8 bytes.
- **kwargs** (*Any*) – Passed to **`json.loads()`**.

Return type: *Any*

`response`(**args, **kwargs*)

Serialize the given arguments as JSON, and return a **`Response`** object with it. The response mimetype will be “application/json” and can be changed with **`mimetype`**.

If **`compact`** is `False` or debug mode is enabled, the output will be formatted to be easier to read.

Either positional or keyword arguments can be given, not both. If no arguments are given, `None` is serialized.

Parameters:
- **args** (*t.Any*) – A single value to serialize, or multiple values to treat as a list to serialize.
- **kwargs** (*t.Any*) – Treat as a dict to serialize.

Return type: Response

Tagged JSON

A compact representation for lossless serialization of non-standard JSON ty
`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()**.

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

Register a new tag with this serializer.

Parameters:
- **tag_class** (*type[JSONTag]*) – tag class to register. Will be instantiated with this serializer instance.
- **force** (*bool*) – overwrite an existing tag. If false (default), a **KeyError** is raised.
- **index** (*int* | *None*) – 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.

Return type: None

tag(*value*)

Convert a value to a tagged representation if necessary.

Parameters: **value** (*Any*) –

Return type: *Any*

untag(*value*)

Convert a tagged representation back to the original type.

Parameters: **value** (*dict[str, Any]*) –
Return type: *Any*

dumps(*value*)

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

Parameters: **value** (*Any*) –
Return type: str

loads(*value*)

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

Parameters: **value** (*str*) –
Return type: *Any*

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

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

Parameters: **serializer** (*TaggedJSONSerializer*) –

key: *str* = ''

The tag to mark the serialized object with. If empty, this tag is only used as an intermediate step during tagging.

check(*value*)

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

Parameters: **value** (*Any*) –
Return type: bool

to_json(*value*)

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

Parameters: **value** (*Any*) –
Return type: *Any*

to_python(*value*)

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

Parameters: **value** (*Any*) –

Return type: *Any*

`tag`(*value*)

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

Parameters: **value** (*Any*) –
Return type: dict[str, *Any*]

Let's see an example that adds support for **`OrderedDict`**. Dicts don't have an order in 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*)

Render a template by name with the given context.

Parameters:
- **template_name_or_list** (*str* | *Template* | *list*[*str* | *Template*]) – The name of the template to render. If a list is given, the first name to exist will be rendered.
- **context** (*Any*) – The variables to make available in the template.

Return type: str

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

Render a template from the given source string with the given context.

Parameters:
- **source** (*str*) – The source code of the template to render.
- **context** (*Any*) – The variables to make available in the template.

Return type: str

`flask.`**`stream_template`**`(template_name_or_list, **context)`

Render a template by name with the given context as a stream. This returns an iterator of strings, which can be used as a streaming response from a view.

Parameters:
- **template_name_or_list** (*str* | *Template* | *list*[*str* | *Template*]) – The name of the template to render. If a list is given, the first name to exist will be rendered.
- **context** (*Any*) – The variables to make available in the template.

Return type: *Iterator*[str]

▶ *Changelog*

`flask.`**`stream_template_string`**`(source, **context)`

Render a template from the given source string with the given context as a stream. This returns an iterator of strings, which can be used as a streaming response from a view.

Parameters:
- **source** (*str*) – The source code of the template to render.
- **context** (*Any*) – The variables to make available in the template.

Return type: *Iterator*[str]

▶ *Changelog*

`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** (*str*) – the name of the template
- **attribute** (*str*) – the name of the variable of macro to access

Return type: *Any*

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** (*str* | *os.PathLike[str]*) – path to which files are read relative from. When the config object is created v: 3.0.x plication, this is the application's **root_path**.

- **defaults** (*dict[str, t.Any] | None*) – 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** (*str*) – name of the environment variable
- **silent** (*bool*) – set to `True` if you want silent failure for missing files.

Returns: `True` if the file was loaded successfully.

Return type: bool

`from_prefixed_env(prefix='FLASK', *, loads=json.loads)`

Load any environment variables that start with `FLASK_`, dropping the prefix from the env key for the config key. Values are passed through a loading function to attempt to convert them to more specific types than strings.

Keys are loaded in **`sorted()`** order.

The default loading function attempts to parse values as any valid JSON type, including dicts and lists.

Specific items in nested dicts can be set by separating the keys with double underscores (__). If an intermediate key doesn't exist, it will be initialized to an empty dict.

Parameters:
- **prefix** (*str*) – Load env vars that start with this prefix, separated with an underscore (_).
- **loads** (*Callable[[str], Any]*) – Pass each string value to this function and use the returned value as the config value. If any error is raised it is ignored and the value remains a string. The default is **`json.loads()`**.

Return type: bool

▶ *Changelog*

`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** (*str* | *PathLike[str]*) – the filename of the config. This can either be an absolute filename or a filename relative to the root path.
- **silent** (*bool*) – set to `True` if you want silent failure for missing files.

Returns: `True` if the file was loaded successfully.

Return type: bool

▶ *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)
```

Nothing is done to the object before loading. If the object is a class and has `@property` attributes, it needs to be instantiated before being passed to this method.

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** (*object* | *str*) – an import name or object

Return type: None

`from_file(filename, load, silent=False, text=True)`

Update the values in the config from a file that is loaded using the `lo` v: 3.0.x meter. The loaded data is passed to the **`from_mapping()`** method.

```
import json
app.config.from_file("config.json", load=json.load)

import tomllib
app.config.from_file("config.toml", load=tomllib.load, text=False
```

Parameters:
- **filename** (*str* | *PathLike[str]*) – The path to the data file. This can be an absolute path or relative to the config root path.
- **load** (`Callable[[Reader], Mapping]` where `Reader` implements a `read` method.) – A callable that takes a file handle and returns a mapping of loaded data from the file.
- **silent** (*bool*) – Ignore the file if it doesn't exist.
- **text** (*bool*) – Open the file in text or binary mode.

Returns: `True` if the file was loaded successfully.

Return type: bool

▶ *Changelog*

`from_mapping`(*mapping=None, **kwargs*)

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

Returns: Always returns `True`.

Parameters:
- **mapping** (*Mapping[str, Any]* | *None*) –
- **kwargs** (*Any*) –

Return type: bool

▶ *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** (*str*) – a configuration namespace
- **lowercase** (*bool*) – a flag indicating if the keys of the resulting dictionary should be lowercase
- **trim_namespace** (*bool*) – a flag indicating if the keys of the resulting dictionary should not include the namespace

Return type: dict[str, *Any*]

▶ *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*

Parameters: **generator_or_function** (*Iterator* | *Callable[[...], Iterator]*) –
Return type: *Iterator*

Useful Internals

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

The request context contains per-request information. The Flask app creates and pushes it at the beginning of the request, then pops it at the end of the request. 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. When using the interactive debugger, the context will be restored so `request` is still accessible. Similarly, the test client can preserve the context after the request ends. However, teardown functions may already have closed some resources such as database connections.

Parameters:
- **app** (*Flask*) –
- **environ** (*WSGIEnvironment*) –
- **request** (*Request* | *None*) –
- **session** (*SessionMixin* | *None*) –

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*

Return type: *RequestContext*

match_request()

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

Return type: None

pop(*exc=_sentinel*)

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*

Parameters: **exc** (*BaseException* | *None*) –
Return type: None

flask.globals.**request_ctx**

The current **RequestContext**. If a request context is not active, accessing attributes on this proxy will raise a `RuntimeError`.

This is an internal object that is essential to how Flask handles requests. Accessing this should not be needed in most cases. Most likely you want **request** and **session** instead.

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

The app context contains application-specific information. An app context is created and pushed at the beginning of each request if one is not already active. An app context is also pushed when running CLI commands.

Parameters: **app** (*Flask*) –

push()

Binds the app context to the current context.

Return type: None

pop(*exc=_sentinel*)

Pops the app context.

Parameters: **exc** (*BaseException* | *None*) –
Return type: None

flask.globals.**app_ctx**

The current **AppContext**. If an app context is not active, accessing attributes on this proxy will raise a `RuntimeError`.

This is an internal object that is essential to how Flask handles requests. v: 3.0.x
this should not be needed in most cases. Most likely you want **current_app** and **g** instead.

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.

Parameters:
- **blueprint** (*Blueprint*) –
- **app** (*App*) –
- **options** (*t.Any*) –
- **first_registration** (*bool*) –

app

a reference to the current application

blueprint

a reference to the blueprint that created this setup state.

options

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

first_registration

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.

subdomain

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

url_prefix

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

url_defaults

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

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.

Parameters:
- **rule** (*str*) –
- **endpoint** (*str* | *None*) –
- **view_func** (*ft.RouteCallable* | *None*) –

- **options** (*t.Any*) –

Return type: None

Signals

Signals are provided by the Blinker library. See Signals for an introduction.

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 unhandled exception happens during request processing, including when debugging. The exception is passed to the subscriber as `exception`.

This signal is not sent for **`HTTPException`**, or other exceptions that have error handlers registered, unless the exception was raised from an error handler.

This example shows how to do some extra logging if a theoretical `SecurityException` was raised:

```
from flask import got_request_exception

def log_security_exception(sender, exception, **extra):
    if not isinstance(exception, SecurityException):
        return

    security_logger.exception(
        f"SecurityException at {request.url!r}",
        exc_info=exception,
    )

got_request_exception.connect(log_security_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-Based Views

▶ *Changelog*

`class` `flask.views.`**`View`**

Subclass this class and override **`dispatch_request()`** to create a generic class-based view. Call **`as_view()`** to create a view function that creates an instance of the class with the given arguments and calls its `dispatch_request` method with any URL variables.

See Class-based Views for a detailed guide.

```
class Hello(View):
    init_every_request = False

    def dispatch_request(self, name):
        return f"Hello, {name}!"

app.add_url_rule(
```

```
    "/hello/<name>", view_func=Hello.as_view("hello")
)
```

Set **`methods`** on the class to change what methods the view accepts.

Set **`decorators`** on the class to apply a list of decorators to the generated view function. Decorators applied to the class itself will not be applied to the generated view function!

Set **`init_every_request`** to `False` for efficiency, unless you need to store request-global data on `self`.

`methods`: *ClassVar[Collection[str] | None] = None*

The methods this view is registered for. Uses the same default (`["GET", "HEAD", "OPTIONS"]`) as `route` and `add_url_rule` by default.

`provide_automatic_options`: *ClassVar[bool | None] = None*

Control whether the `OPTIONS` method is handled automatically. Uses the same default (`True`) as `route` and `add_url_rule` by default.

`decorators`: *ClassVar[list[Callable[[F], F]]] = []*

A list of decorators to apply, in order, to the generated view function. Remember that `@decorator` syntax is applied bottom to top, so the first decorator in the list would be the bottom decorator.

▶ *Changelog*

`init_every_request`: *ClassVar[bool] = True*

Create a new instance of this view class for every request by default. If a view subclass sets this to `False`, the same instance is used for every request.

A single instance is more efficient, especially if complex setup is done during init. However, storing data on `self` is no longer safe across requests, and **`g`** should be used instead.

▶ *Changelog*

`dispatch_request``()`

The actual view function behavior. Subclasses must override this and return a valid response. Any variables from the URL rule are passed as keyword arguments.

Return type: ft.ResponseReturnValue

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

Convert the class into a view function that can be registered for a route.

By default, the generated view will create a new instance of the view class for every request and call its **`dispatch_request()`** method. If the view class sets **`init_every_request`** to `False`, the same instance will be used for every request.

Except for `name`, all other arguments passed to this method are forwarded to the view class `__init__` method.

▶ *Changelog*

Parameters:
- **name** (*str*) –
- **class_args** (*t.Any*) –
- **class_kwargs** (*t.Any*) –

Return type: ft.RouteCallable

class flask.views.**MethodView**

Dispatches request methods to the corresponding instance methods. For example, if you implement a `get` method, it will be used to handle `GET` requests.

This can be useful for defining a REST API.

`methods` is automatically set based on the methods defined on the class.

See Class-based Views for a detailed guide.

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

    def post(self):
        session["counter"] = session.get("counter", 0) + 1
        return redirect(url_for("counter"))

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

dispatch_request(***kwargs*)

The actual view function behavior. Subclasses must override this and return a valid response. Any variables from the URL rule are passed as keyword arguments.

Parameters: **kwargs** (*t.Any*) –

Return type: ft.ResponseReturnValue

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/<int: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 matc 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*, *set_debug_flag=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. see Custom Scripts.

Parameters:
- **add_default_commands** (*bool*) – if this is True then the default run and shell commands will be added.
- **add_version_option** (*bool*) – adds the `--version` option.
- **create_app** (*t.Callable[..., Flask]* | *None*) – an optional callback that is passed the script info and returns the loaded app.
- **load_dotenv** (*bool*) – 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.
- **set_debug_flag** (*bool*) – Set the app's debug flag.
- **extra** (*t.Any*) –

▶ *Changelog*

get_command(*ctx*, *name*)

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

Parameters:
- **ctx** (*Context*) –
- **name** (*str*) –

Return type: *Command* | None

list_commands(*ctx*)

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

Parameters: **ctx** (*Context*) –

Return type: list[str]

make_context(*info_name*, *args*, *parent=None*, ***extra*)

This function when given an info name and arguments will kick off the parsing and create a new **Context**. It does not invoke the actual command callback though.

To quickly customize the context class used without overriding this method, set the **context_class** attribute.

Parameters:
- **info_name** (*str* | *None*) – the info name for this invocation. Generally this is the most descriptive name for the

script or command. For the toplevel script it's usually the name of the script, for commands below it's the name of the command.
- **args** (*list[str]*) – the arguments to parse as list of strings.
- **parent** (*Context* | *None*) – the parent context if available.
- **extra** (*Any*) – extra keyword arguments forwarded to the context constructor.

Return type: *Context*

Changed in version 8.0: Added the **context_class** attribute.

parse_args(*ctx*, *args*)

Given a context and a list of arguments this creates the parser and parses the arguments, then modifies the context as necessary. This is automatically invoked by **make_context()**.

Parameters:
- **ctx** (*Context*) –
- **args** (*list[str]*) –

Return type: list[str]

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**.

Parameters:
- **name** (*str* | *None*) –
- **commands** (*MutableMapping[str, Command]* | *Sequence[Command]* | *None*) –
- **attrs** (*Any*) –

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`.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *Callable*[[*Callable*[[...], *Any*]], *Command*]

group(**args*, ***kwargs*)

This works exactly like the method of the same name on a regular **cl** v: 3.0.x but it defaults the group class to **AppGroup**.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *Callable*[[*Callable*[[...], *Any*]], *Group*]

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

Helper 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.

Parameters:
- **app_import_path** (*str* | *None*) –
- **create_app** (*t.Callable[..., Flask]* | *None*) –
- **set_debug_flag** (*bool*) –

app_import_path

Optionally the import path for the Flask application.

create_app

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

data: *dict*[*t.Any*, *t.Any*]

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.

Return type: Flask

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.

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

Parameters: **path** (*str* | *PathLike[str]* | *None*) – Load the file at this location instead of searching.

Returns: `True` if a file was loaded.

Return type: bool

▶ *Changelog*

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

Wraps a callback so that it's guaranteed to be executed with the script's application context.

Custom commands (and their options) registered under `app.cli` or `blueprint.cli` will always have an app context available, this decorator is not required in that case.

▶ *Changelog*

Parameters: **f** (*F*) –
Return type: *F*

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

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

Parameters: **f** (*t.Callable[te.Concatenate[T, P], R]*) –
Return type: t.Callable[P, R]

flask.cli.**run_command** = <*Command run*>

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 with the '–debug' option.

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *Any*

flask.cli.**shell_command** = <*Command shell*>

Run an interactive Python shell in the context of a given Flask application. The application will populate the default namespace of this shell according to its configuration.

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

Parameters:
- **args** (*Any*) –
- **kwargs** (*Any*) –

Return type: *Any*