
Flask Documentation (2.3.x)

Release 2.3.2

Pallets

May 28, 2023

CONTENTS

1	User's Guide	3
1.1	Installation	3
1.2	Quickstart	5
1.3	Tutorial	19
1.4	Templates	55
1.5	Testing Flask Applications	59
1.6	Handling Application Errors	63
1.7	Debugging Application Errors	71
1.8	Logging	73
1.9	Configuration Handling	76
1.10	Signals	86
1.11	Class-based Views	88
1.12	Application Structure and Lifecycle	93
1.13	The Application Context	97
1.14	The Request Context	99
1.15	Modular Applications with Blueprints	102
1.16	Extensions	107
1.17	Command Line Interface	108
1.18	Development Server	116
1.19	Working with the Shell	117
1.20	Patterns for Flask	119
1.21	Security Considerations	163
1.22	Deploying to Production	167
1.23	Using <code>async</code> and <code>await</code>	179
2	API Reference	181
2.1	API	181
3	Additional Notes	277
3.1	Design Decisions in Flask	277
3.2	Flask Extension Development	280
3.3	How to contribute to Flask	284
3.4	BSD-3-Clause License	287
3.5	Changes	288
	Python Module Index	313
	Index	315



Welcome to Flask's documentation. Get started with [Installation](#) and then get an overview with the [Quickstart](#). There is also a more detailed [Tutorial](#) that shows how to create a small but complete application with Flask. Common patterns are described in the [Patterns for Flask](#) section. The rest of the docs describe each component of Flask in detail, with a full reference in the [API](#) section.

Flask depends on the [Werkzeug](#) WSGI toolkit, the [Jinja](#) template engine, and the [Click](#) CLI toolkit. Be sure to check their documentation as well as Flask's when looking for information.

USER'S GUIDE

Flask provides configuration and conventions, with sensible defaults, to get started. This section of the documentation explains the different parts of the Flask framework and how they can be used, customized, and extended. Beyond Flask itself, look for community-maintained extensions to add even more functionality.

1.1 Installation

1.1.1 Python Version

We recommend using the latest version of Python. Flask supports Python 3.8 and newer.

1.1.2 Dependencies

These distributions will be installed automatically when installing Flask.

- [Werkzeug](#) implements WSGI, the standard Python interface between applications and servers.
- [Jinja](#) is a template language that renders the pages your application serves.
- [MarkupSafe](#) comes with Jinja. It escapes untrusted input when rendering templates to avoid injection attacks.
- [ItsDangerous](#) securely signs data to ensure its integrity. This is used to protect Flask's session cookie.
- [Click](#) is a framework for writing command line applications. It provides the `flask` command and allows adding custom management commands.
- [Blinker](#) provides support for *Signals*.

Optional dependencies

These distributions will not be installed automatically. Flask will detect and use them if you install them.

- [python-dotenv](#) enables support for *Environment Variables From dotenv* when running `flask` commands.
- [Watchdog](#) provides a faster, more efficient reloader for the development server.

greenlet

You may choose to use `gevent` or `eventlet` with your application. In this case, `greenlet` ≥ 1.0 is required. When using PyPy, PyPy $\geq 7.3.7$ is required.

These are not minimum supported versions, they only indicate the first versions that added necessary features. You should use the latest versions of each.

1.1.3 Virtual environments

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system's packages.

Python comes bundled with the `venv` module to create virtual environments.

Create an environment

Create a project folder and a `.venv` folder within:

macOS/Linux

```
$ mkdir myproject
$ cd myproject
$ python3 -m venv .venv
```

Windows

```
> mkdir myproject
> cd myproject
> py -3 -m venv .venv
```

Activate the environment

Before you work on your project, activate the corresponding environment:

macOS/Linux

```
$ . .venv/bin/activate
```

Windows

```
> . .venv\Scripts\activate
```

Your shell prompt will change to show the name of the activated environment.

1.1.4 Install Flask

Within the activated environment, use the following command to install Flask:

```
$ pip install Flask
```

Flask is now installed. Check out the [Quickstart](#) or go to the [Documentation Overview](#).

1.2 Quickstart

Eager to get started? This page gives a good introduction to Flask. Follow [Installation](#) to set up a project and install Flask first.

1.2.1 A Minimal Application

A minimal Flask application looks something like this:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"
```

So what did that code do?

1. First we imported the `Flask` class. An instance of this class will be our WSGI application.
2. Next we create an instance of this class. The first argument is the name of the application's module or package. `__name__` is a convenient shortcut for this that is appropriate for most cases. This is needed so that Flask knows where to look for resources such as templates and static files.
3. We then use the `route()` decorator to tell Flask what URL should trigger our function.
4. The function returns the message we want to display in the user's browser. The default content type is HTML, so HTML in the string will be rendered by the browser.

Save it as `hello.py` or something similar. Make sure to not call your application `flask.py` because this would conflict with Flask itself.

To run the application, use the `flask` command or `python -m flask`. You need to tell the Flask where your application is with the `--app` option.

```
$ flask --app hello run
* Serving Flask app 'hello'
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
```

Application Discovery Behavior

As a shortcut, if the file is named `app.py` or `wsgi.py`, you don't have to use `--app`. See [Command Line Interface](#) for more details.

This launches a very simple builtin server, which is good enough for testing but probably not what you want to use in production. For deployment options see *Deploying to Production*.

Now head over to <http://127.0.0.1:5000/>, and you should see your hello world greeting.

If another program is already using port 5000, you'll see `OSError: [Errno 98]` or `OSError: [WinError 10013]` when the server tries to start. See *Address already in use* for how to handle that.

Externally Visible Server

If you run the server you will notice that the server is only accessible from your own computer, not from any other in the network. This is the default because in debugging mode a user of the application can execute arbitrary Python code on your computer.

If you have the debugger disabled or trust the users on your network, you can make the server publicly available simply by adding `--host=0.0.0.0` to the command line:

```
$ flask run --host=0.0.0.0
```

This tells your operating system to listen on all public IPs.

1.2.2 Debug Mode

The `flask run` command can do more than just start the development server. By enabling debug mode, the server will automatically reload if code changes, and will show an interactive debugger in the browser if an error occurs during a request.

TypeError

TypeError: cannot concatenate 'str' and 'NoneType' objects

Traceback (most recent call last)

```
File "/Users/mitsuhiko/Development/flask/flask.py", line 650, in __call__
    return self.wsgi_app(environ, start_response)

File "/Users/mitsuhiko/Development/werkzeug-main/werkzeug/wsgi.py", line 406, in
__call__
    return self.app(environ, start_response)

File "/Users/mitsuhiko/Development/flask/flask.py", line 616, in wsgi_app
    rv = self.dispatch_request()

File "/Users/mitsuhiko/Development/flask/flask.py", line 535, in dispatch_request
    return self.view_functions[endpoint](**values)

File "/Users/mitsuhiko/Development/flask/test.py", line 8, in index
    return 'Hello ' + name

[console ready]
>>> type(name)
<type 'NoneType'>
>>>
```

Warning: The debugger allows executing arbitrary Python code from the browser. It is protected by a pin, but still represents a major security risk. Do not run the development server or debugger in a production environment.

To enable debug mode, use the `--debug` option.

```
$ flask --app hello run --debug
* Serving Flask app 'hello'
* Debug mode: on
* Running on http://127.0.0.1:5000 (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: nnn-nnn-nnn
```

See also:

- [Development Server](#) and [Command Line Interface](#) for information about running in debug mode.
- [Debugging Application Errors](#) for information about using the built-in debugger and other debuggers.
- [Logging](#) and [Handling Application Errors](#) to log errors and display nice error pages.

1.2.3 HTML Escaping

When returning HTML (the default response type in Flask), any user-provided values rendered in the output must be escaped to protect from injection attacks. HTML templates rendered with Jinja, introduced later, will do this automatically.

`escape()`, shown here, can be used manually. It is omitted in most examples for brevity, but you should always be aware of how you're using untrusted data.

```
from markupsafe import escape

@app.route("/<name>")
def hello(name):
    return f"Hello, {escape(name)}!"
```

If a user managed to submit the name `<script>alert("bad")</script>`, escaping causes it to be rendered as text, rather than running the script in the user's browser.

`<name>` in the route captures a value from the URL and passes it to the view function. These variable rules are explained below.

1.2.4 Routing

Modern web applications use meaningful URLs to help users. Users are more likely to like a page and come back if the page uses a meaningful URL they can remember and use to directly visit a page.

Use the `route()` decorator to bind a function to a URL.

```
@app.route('/')
def index():
    return 'Index Page'

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

You can do more! You can make parts of the URL dynamic and attach multiple rules to a function.

Variable Rules

You can add variable sections to a URL by marking sections with `<variable_name>`. Your function then receives the `<variable_name>` as a keyword argument. Optionally, you can use a converter to specify the type of the argument like `<converter:variable_name>`.

```
from markupsafe import escape

@app.route('/user/<username>')
def show_user_profile(username):
```

(continues on next page)

(continued from previous page)

```

# show the user profile for that user
return f'User {escape(username)}'

@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return f'Post {post_id}'

@app.route('/path/<path:subpath>')
def show_subpath(subpath):
    # show the subpath after /path/
    return f'Subpath {escape(subpath)}'

```

Converter types:

string	(default) accepts any text without a slash
int	accepts positive integers
float	accepts positive floating point values
path	like string but also accepts slashes
uuid	accepts UUID strings

Unique URLs / Redirection Behavior

The following two rules differ in their use of a trailing slash.

```

@app.route('/projects/')
def projects():
    return 'The project page'

@app.route('/about')
def about():
    return 'The about page'

```

The canonical URL for the `projects` endpoint has a trailing slash. It's similar to a folder in a file system. If you access the URL without a trailing slash (`/projects`), Flask redirects you to the canonical URL with the trailing slash (`/projects/`).

The canonical URL for the `about` endpoint does not have a trailing slash. It's similar to the pathname of a file. Accessing the URL with a trailing slash (`/about/`) produces a 404 “Not Found” error. This helps keep URLs unique for these resources, which helps search engines avoid indexing the same page twice.

URL Building

To build a URL to a specific function, use the `url_for()` function. It accepts the name of the function as its first argument and any number of keyword arguments, each corresponding to a variable part of the URL rule. Unknown variable parts are appended to the URL as query parameters.

Why would you want to build URLs using the URL reversing function `url_for()` instead of hard-coding them into your templates?

1. Reversing is often more descriptive than hard-coding the URLs.
2. You can change your URLs in one go instead of needing to remember to manually change hard-coded URLs.

3. URL building handles escaping of special characters transparently.
4. The generated paths are always absolute, avoiding unexpected behavior of relative paths in browsers.
5. If your application is placed outside the URL root, for example, in `/myapplication` instead of `/`, `url_for()` properly handles that for you.

For example, here we use the `test_request_context()` method to try out `url_for()`. `test_request_context()` tells Flask to behave as though it's handling a request even while we use a Python shell. See *Context Locals*.

```
from flask import url_for

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

@app.route('/login')
def login():
    return 'login'

@app.route('/user/<username>')
def profile(username):
    return f'{username}\s profile'

with app.test_request_context():
    print(url_for('index'))
    print(url_for('login'))
    print(url_for('login', next='/'))
    print(url_for('profile', username='John Doe'))
```

```
/
/login
/login?next=/
/user/John%20Doe
```

HTTP Methods

Web applications use different HTTP methods when accessing URLs. You should familiarize yourself with the HTTP methods as you work with Flask. By default, a route only answers to GET requests. You can use the `methods` argument of the `route()` decorator to handle different HTTP methods.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

The example above keeps all methods for the route within one function, which can be useful if each part uses some common data.

You can also separate views for different methods into different functions. Flask provides a shortcut for decorating such routes with `get()`, `post()`, etc. for each common HTTP method.

```
@app.get('/login')
def login_get():
    return show_the_login_form()

@app.post('/login')
def login_post():
    return do_the_login()
```

If GET is present, Flask automatically adds support for the HEAD method and handles HEAD requests according to the [HTTP RFC](#). Likewise, OPTIONS is automatically implemented for you.

1.2.5 Static Files

Dynamic web applications also need static files. That's usually where the CSS and JavaScript files are coming from. Ideally your web server is configured to serve them for you, but during development Flask can do that as well. Just create a folder called `static` in your package or next to your module and it will be available at `/static` on the application.

To generate URLs for static files, use the special `'static'` endpoint name:

```
url_for('static', filename='style.css')
```

The file has to be stored on the filesystem as `static/style.css`.

1.2.6 Rendering Templates

Generating HTML from within Python is not fun, and actually pretty cumbersome because you have to do the HTML escaping on your own to keep the application secure. Because of that Flask configures the [Jinja2](#) template engine for you automatically.

Templates can be used to generate any type of text file. For web applications, you'll primarily be generating HTML pages, but you can also generate markdown, plain text for emails, and anything else.

For a reference to HTML, CSS, and other web APIs, use the [MDN Web Docs](#).

To render a template you can use the `render_template()` method. All you have to do is provide the name of the template and the variables you want to pass to the template engine as keyword arguments. Here's a simple example of how to render a template:

```
from flask import render_template

@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', name=name)
```

Flask will look for templates in the `templates` folder. So if your application is a module, this folder is next to that module, if it's a package it's actually inside your package:

Case 1: a module:

```
/application.py
/templates
  /hello.html
```

Case 2: a package:

```
/application
  /__init__.py
  /templates
    /hello.html
```

For templates you can use the full power of Jinja2 templates. Head over to the official [Jinja2 Template Documentation](#) for more information.

Here is an example template:

```
<!doctype html>
<title>Hello from Flask</title>
{% if name %}
  <h1>Hello {{ name }}!</h1>
{% else %}
  <h1>Hello, World!</h1>
{% endif %}
```

Inside templates you also have access to the `config`, `request`, `session` and `g`¹ objects as well as the `url_for()` and `get_flashed_messages()` functions.

Templates are especially useful if inheritance is used. If you want to know how that works, see [Template Inheritance](#). Basically template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

Automatic escaping is enabled, so if `name` contains HTML it will be escaped automatically. If you can trust a variable and you know that it will be safe HTML (for example because it came from a module that converts wiki markup to HTML) you can mark it as safe by using the Markup class or by using the `|safe` filter in the template. Head over to the Jinja 2 documentation for more examples.

Here is a basic introduction to how the Markup class works:

```
>>> from markupsafe import Markup
>>> Markup('<strong>Hello %s!</strong>') % '<blink>hacker</blink>'
Markup('<strong>Hello &lt;blink&gt;hacker&lt;/blink&gt;!</strong>')
>>> Markup.escape('<blink>hacker</blink>')
Markup('&lt;blink&gt;hacker&lt;/blink&gt;')
>>> Markup('<em>Marked up</em> &raquo; HTML').striptags()
'Marked up >> HTML'
```

Changed in version 0.5: Autoescaping is no longer enabled for all templates. The following extensions for templates trigger autoescaping: `.html`, `.htm`, `.xml`, `.xhtml`. Templates loaded from a string will have autoescaping disabled.

1.2.7 Accessing Request Data

For web applications it's crucial to react to the data a client sends to the server. In Flask this information is provided by the global `request` object. If you have some experience with Python you might be wondering how that object can be global and how Flask manages to still be threadsafe. The answer is context locals:

¹ Unsure what that `g` object is? It's something in which you can store information for your own needs. See the documentation for [flask.g](#) and [Using SQLite 3 with Flask](#).

Context Locals

Insider Information

If you want to understand how that works and how you can implement tests with context locals, read this section, otherwise just skip it.

Certain objects in Flask are global objects, but not of the usual kind. These objects are actually proxies to objects that are local to a specific context. What a mouthful. But that is actually quite easy to understand.

Imagine the context being the handling thread. A request comes in and the web server decides to spawn a new thread (or something else, the underlying object is capable of dealing with concurrency systems other than threads). When Flask starts its internal request handling it figures out that the current thread is the active context and binds the current application and the WSGI environments to that context (thread). It does that in an intelligent way so that one application can invoke another application without breaking.

So what does this mean to you? Basically you can completely ignore that this is the case unless you are doing something like unit testing. You will notice that code which depends on a request object will suddenly break because there is no request object. The solution is creating a request object yourself and binding it to the context. The easiest solution for unit testing is to use the `test_request_context()` context manager. In combination with the `with` statement it will bind a test request so that you can interact with it. Here is an example:

```
from flask import request

with app.test_request_context('/hello', method='POST'):
    # now you can do something with the request until the
    # end of the with block, such as basic assertions:
    assert request.path == '/hello'
    assert request.method == 'POST'
```

The other possibility is passing a whole WSGI environment to the `request_context()` method:

```
with app.request_context(environ):
    assert request.method == 'POST'
```

The Request Object

The request object is documented in the API section and we will not cover it here in detail (see [Request](#)). Here is a broad overview of some of the most common operations. First of all you have to import it from the `flask` module:

```
from flask import request
```

The current request method is available by using the `method` attribute. To access form data (data transmitted in a POST or PUT request) you can use the `form` attribute. Here is a full example of the two attributes mentioned above:

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    error = None
    if request.method == 'POST':
        if valid_login(request.form['username'],
                       request.form['password']):
            return log_the_user_in(request.form['username'])
        else:
```

(continues on next page)

(continued from previous page)

```
error = 'Invalid username/password'
# the code below is executed if the request method
# was GET or the credentials were invalid
return render_template('login.html', error=error)
```

What happens if the key does not exist in the `form` attribute? In that case a special `KeyError` is raised. You can catch it like a standard `KeyError` but if you don't do that, a HTTP 400 Bad Request error page is shown instead. So for many situations you don't have to deal with that problem.

To access parameters submitted in the URL (`?key=value`) you can use the `args` attribute:

```
searchword = request.args.get('key', '')
```

We recommend accessing URL parameters with `get` or by catching the `KeyError` because users might change the URL and presenting them a 400 bad request page in that case is not user friendly.

For a full list of methods and attributes of the request object, head over to the [Request](#) documentation.

File Uploads

You can handle uploaded files with Flask easily. Just make sure not to forget to set the `enctype="multipart/form-data"` attribute on your HTML form, otherwise the browser will not transmit your files at all.

Uploaded files are stored in memory or at a temporary location on the filesystem. You can access those files by looking at the `files` attribute on the request object. Each uploaded file is stored in that dictionary. It behaves just like a standard Python file object, but it also has a `save()` method that allows you to store that file on the filesystem of the server. Here is a simple example showing how that works:

```
from flask import request

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        f = request.files['the_file']
        f.save('/var/www/uploads/uploaded_file.txt')
    ...
```

If you want to know how the file was named on the client before it was uploaded to your application, you can access the `filename` attribute. However please keep in mind that this value can be forged so never ever trust that value. If you want to use the filename of the client to store the file on the server, pass it through the `secure_filename()` function that Werkzeug provides for you:

```
from werkzeug.utils import secure_filename

@app.route('/upload', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['the_file']
        file.save(f"/var/www/uploads/{secure_filename(file.filename)}")
    ...
```

For some better examples, see [Uploading Files](#).

Cookies

To access cookies you can use the `cookies` attribute. To set cookies you can use the `set_cookie` method of response objects. The `cookies` attribute of request objects is a dictionary with all the cookies the client transmits. If you want to use sessions, do not use the cookies directly but instead use the `Sessions` in Flask that add some security on top of cookies for you.

Reading cookies:

```
from flask import request

@app.route('/')
def index():
    username = request.cookies.get('username')
    # use cookies.get(key) instead of cookies[key] to not get a
    # KeyError if the cookie is missing.
```

Storing cookies:

```
from flask import make_response

@app.route('/')
def index():
    resp = make_response(render_template(...))
    resp.set_cookie('username', 'the username')
    return resp
```

Note that cookies are set on response objects. Since you normally just return strings from the view functions Flask will convert them into response objects for you. If you explicitly want to do that you can use the `make_response()` function and then modify it.

Sometimes you might want to set a cookie at a point where the response object does not exist yet. This is possible by utilizing the *Deferred Request Callbacks* pattern.

For this also see *About Responses*.

1.2.8 Redirects and Errors

To redirect a user to another endpoint, use the `redirect()` function; to abort a request early with an error code, use the `abort()` function:

```
from flask import abort, redirect, url_for

@app.route('/')
def index():
    return redirect(url_for('login'))

@app.route('/login')
def login():
    abort(401)
    this_is_never_executed()
```

This is a rather pointless example because a user will be redirected from the index to a page they cannot access (401 means access denied) but it shows how that works.

By default a black and white error page is shown for each error code. If you want to customize the error page, you can use the `errorhandler()` decorator:

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

Note the `404` after the `render_template()` call. This tells Flask that the status code of that page should be `404` which means not found. By default `200` is assumed which translates to: all went well.

See *Handling Application Errors* for more details.

1.2.9 About Responses

The return value from a view function is automatically converted into a response object for you. If the return value is a string it's converted into a response object with the string as response body, a `200 OK` status code and a `text/html` mimetype. If the return value is a dict or list, `jsonify()` is called to produce a response. The logic that Flask applies to converting return values into response objects is as follows:

1. If a response object of the correct type is returned it's directly returned from the view.
2. If it's a string, a response object is created with that data and the default parameters.
3. If it's an iterator or generator returning strings or bytes, it is treated as a streaming response.
4. If it's a dict or list, a response object is created using `jsonify()`.
5. If a tuple is returned the items in the tuple can provide extra information. Such tuples have to be in the form `(response, status)`, `(response, headers)`, or `(response, status, headers)`. The `status` value will override the status code and `headers` can be a list or dictionary of additional header values.
6. If none of that works, Flask will assume the return value is a valid WSGI application and convert that into a response object.

If you want to get hold of the resulting response object inside the view you can use the `make_response()` function.

Imagine you have a view like this:

```
from flask import render_template

@app.errorhandler(404)
def not_found(error):
    return render_template('error.html'), 404
```

You just need to wrap the return expression with `make_response()` and get the response object to modify it, then return it:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    resp = make_response(render_template('error.html'), 404)
    resp.headers['X-Something'] = 'A value'
    return resp
```

APIs with JSON

A common response format when writing an API is JSON. It's easy to get started writing such an API with Flask. If you return a dict or list from a view, it will be converted to a JSON response.

```
@app.route("/me")
def me_api():
    user = get_current_user()
    return {
        "username": user.username,
        "theme": user.theme,
        "image": url_for("user_image", filename=user.image),
    }

@app.route("/users")
def users_api():
    users = get_all_users()
    return [user.to_json() for user in users]
```

This is a shortcut to passing the data to the `jsonify()` function, which will serialize any supported JSON data type. That means that all the data in the dict or list must be JSON serializable.

For complex types such as database models, you'll want to use a serialization library to convert the data to valid JSON types first. There are many serialization libraries and Flask API extensions maintained by the community that support more complex applications.

1.2.10 Sessions

In addition to the request object there is also a second object called `session` which allows you to store information specific to a user from one request to the next. This is implemented on top of cookies for you and signs the cookies cryptographically. What this means is that the user could look at the contents of your cookie but not modify it, unless they know the secret key used for signing.

In order to use sessions you have to set a secret key. Here is how sessions work:

```
from flask import session

# Set the secret key to some random bytes. Keep this really secret!
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    if 'username' in session:
        return f'Logged in as {session["username"]}'
    return 'You are not logged in'

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        session['username'] = request.form['username']
        return redirect(url_for('index'))
    return '''
    <form method="post">
        <p><input type="text" name="username">
```

(continues on next page)

(continued from previous page)

```
        <p><input type=submit value=Login>
    </form>
    '''

@app.route('/logout')
def logout():
    # remove the username from the session if it's there
    session.pop('username', None)
    return redirect(url_for('index'))
```

How to generate good secret keys

A secret key should be as random as possible. Your operating system has ways to generate pretty random data based on a cryptographic random generator. Use the following command to quickly generate a value for `Flask.secret_key` (or `SECRET_KEY`):

```
$ python -c 'import secrets; print(secrets.token_hex())'
'192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

A note on cookie-based sessions: Flask will take the values you put into the session object and serialize them into a cookie. If you are finding some values do not persist across requests, cookies are indeed enabled, and you are not getting a clear error message, check the size of the cookie in your page responses compared to the size supported by web browsers.

Besides the default client-side based sessions, if you want to handle sessions on the server-side instead, there are several Flask extensions that support this.

1.2.11 Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback they will probably end up hating the application. Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it on the next (and only the next) request. This is usually combined with a layout template to expose the message.

To flash a message use the `flash()` method, to get hold of the messages you can use `get_flashed_messages()` which is also available in the templates. See *Message Flashing* for a full example.

1.2.12 Logging

New in version 0.3.

Sometimes you might be in a situation where you deal with data that should be correct, but actually is not. For example you may have some client-side code that sends an HTTP request to the server but it's obviously malformed. This might be caused by a user tampering with the data, or the client code failing. Most of the time it's okay to reply with `400 Bad Request` in that situation, but sometimes that won't do and the code has to continue working.

You may still want to log that something fishy happened. This is where loggers come in handy. As of Flask 0.3 a logger is preconfigured for you to use.

Here are some example log calls:

```
app.logger.debug('A value for debugging')
app.logger.warning('A warning occurred (%d apples)', 42)
app.logger.error('An error occurred')
```

The attached *logger* is a standard logging `Logger`, so head over to the official [logging](#) docs for more information.

See *Handling Application Errors*.

1.2.13 Hooking in WSGI Middleware

To add WSGI middleware to your Flask application, wrap the application's `wsgi_app` attribute. For example, to apply Werkzeug's `ProxyFix` middleware for running behind Nginx:

```
from werkzeug.middleware.proxy_fix import ProxyFix
app.wsgi_app = ProxyFix(app.wsgi_app)
```

Wrapping `app.wsgi_app` instead of `app` means that `app` still points at your Flask application, not at the middleware, so you can continue to use and configure `app` directly.

1.2.14 Using Flask Extensions

Extensions are packages that help you accomplish common tasks. For example, Flask-SQLAlchemy provides SQLAlchemy support that makes it simple and easy to use with Flask.

For more on Flask extensions, see *Extensions*.

1.2.15 Deploying to a Web Server

Ready to deploy your new Flask app? See *Deploying to Production*.

1.3 Tutorial

1.3.1 Project Layout

Create a project directory and enter it:

```
$ mkdir flask-tutorial
$ cd flask-tutorial
```

Then follow the *installation instructions* to set up a Python virtual environment and install Flask for your project.

The tutorial will assume you're working from the `flask-tutorial` directory from now on. The file names at the top of each code block are relative to this directory.

A Flask application can be as simple as a single file.

Listing 1: hello.py

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'
```

However, as a project gets bigger, it becomes overwhelming to keep all the code in one file. Python projects use *packages* to organize code into multiple modules that can be imported where needed, and the tutorial will do this as well.

The project directory will contain:

- flaskr/, a Python package containing your application code and files.
- tests/, a directory containing test modules.
- .venv/, a Python virtual environment where Flask and other dependencies are installed.
- Installation files telling Python how to install your project.
- Version control config, such as [git](#). You should make a habit of using some type of version control for all your projects, no matter the size.
- Any other project files you might add in the future.

By the end, your project layout will look like this:

```
/home/user/Projects/flask-tutorial
├── flaskr/
│   ├── __init__.py
│   ├── db.py
│   ├── schema.sql
│   ├── auth.py
│   ├── blog.py
│   └── templates/
│       ├── base.html
│       ├── auth/
│       │   ├── login.html
│       │   └── register.html
│       └── blog/
│           ├── create.html
│           ├── index.html
│           └── update.html
├── static/
│   └── style.css
├── tests/
│   ├── conftest.py
│   ├── data.sql
│   ├── test_factory.py
│   ├── test_db.py
│   ├── test_auth.py
│   └── test_blog.py
└── .venv/
```

(continues on next page)

(continued from previous page)

```
├─ pyproject.toml
├─ MANIFEST.in
```

If you're using version control, the following files that are generated while running your project should be ignored. There may be other files based on the editor you use. In general, ignore files that you didn't write. For example, with git:

Listing 2: .gitignore

```
.venv/

*.pyc
__pycache__/

instance/

.pytest_cache/
.coverage
htmlcov/

dist/
build/
*.egg-info/
```

Continue to [Application Setup](#).

1.3.2 Application Setup

A Flask application is an instance of the *Flask* class. Everything about the application, such as configuration and URLs, will be registered with this class.

The most straightforward way to create a Flask application is to create a global *Flask* instance directly at the top of your code, like how the “Hello, World!” example did on the previous page. While this is simple and useful in some cases, it can cause some tricky issues as the project grows.

Instead of creating a *Flask* instance globally, you will create it inside a function. This function is known as the *application factory*. Any configuration, registration, and other setup the application needs will happen inside the function, then the application will be returned.

The Application Factory

It's time to start coding! Create the `flaskr` directory and add the `__init__.py` file. The `__init__.py` serves double duty: it will contain the application factory, and it tells Python that the `flaskr` directory should be treated as a package.

```
$ mkdir flaskr
```

Listing 3: flaskr/__init__.py

```
import os

from flask import Flask
```

(continues on next page)

```
def create_app(test_config=None):
    # create and configure the app
    app = Flask(__name__, instance_relative_config=True)
    app.config.from_mapping(
        SECRET_KEY='dev',
        DATABASE=os.path.join(app.instance_path, 'flaskr.sqlite'),
    )

    if test_config is None:
        # load the instance config, if it exists, when not testing
        app.config.from_pyfile('config.py', silent=True)
    else:
        # load the test config if passed in
        app.config.from_mapping(test_config)

    # ensure the instance folder exists
    try:
        os.makedirs(app.instance_path)
    except OSError:
        pass

    # a simple page that says hello
    @app.route('/hello')
    def hello():
        return 'Hello, World!'

    return app
```

`create_app` is the application factory function. You'll add to it later in the tutorial, but it already does a lot.

1. `app = Flask(__name__, instance_relative_config=True)` creates the *Flask* instance.
 - `__name__` is the name of the current Python module. The app needs to know where it's located to set up some paths, and `__name__` is a convenient way to tell it that.
 - `instance_relative_config=True` tells the app that configuration files are relative to the *instance folder*. The instance folder is located outside the `flaskr` package and can hold local data that shouldn't be committed to version control, such as configuration secrets and the database file.
2. `app.config.from_mapping()` sets some default configuration that the app will use:
 - `SECRET_KEY` is used by Flask and extensions to keep data safe. It's set to `'dev'` to provide a convenient value during development, but it should be overridden with a random value when deploying.
 - `DATABASE` is the path where the SQLite database file will be saved. It's under `app.instance_path`, which is the path that Flask has chosen for the instance folder. You'll learn more about the database in the next section.
3. `app.config.from_pyfile()` overrides the default configuration with values taken from the `config.py` file in the instance folder if it exists. For example, when deploying, this can be used to set a real `SECRET_KEY`.
 - `test_config` can also be passed to the factory, and will be used instead of the instance configuration. This is so the tests you'll write later in the tutorial can be configured independently of any development values you have configured.

4. `os.makedirs()` ensures that `app.instance_path` exists. Flask doesn't create the instance folder automatically, but it needs to be created because your project will create the SQLite database file there.
5. `@app.route()` creates a simple route so you can see the application working before getting into the rest of the tutorial. It creates a connection between the URL `/hello` and a function that returns a response, the string `'Hello, World!'` in this case.

Run The Application

Now you can run your application using the `flask` command. From the terminal, tell Flask where to find your application, then run it in debug mode. Remember, you should still be in the top-level `flask-tutorial` directory, not the `flaskr` package.

Debug mode shows an interactive debugger whenever a page raises an exception, and restarts the server whenever you make changes to the code. You can leave it running and just reload the browser page as you follow the tutorial.

```
$ flask --app flaskr run --debug
```

You'll see output similar to this:

```
* Serving Flask app "flaskr"  
* Debug mode: on  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: nnn-nnn-nnn
```

Visit <http://127.0.0.1:5000/hello> in a browser and you should see the “Hello, World!” message. Congratulations, you're now running your Flask web application!

If another program is already using port 5000, you'll see `OSError: [Errno 98]` or `OSError: [WinError 10013]` when the server tries to start. See *Address already in use* for how to handle that.

Continue to *Define and Access the Database*.

1.3.3 Define and Access the Database

The application will use a `SQLite` database to store users and posts. Python comes with built-in support for `SQLite` in the `sqlite3` module.

`SQLite` is convenient because it doesn't require setting up a separate database server and is built-in to Python. However, if concurrent requests try to write to the database at the same time, they will slow down as each write happens sequentially. Small applications won't notice this. Once you become big, you may want to switch to a different database.

The tutorial doesn't go into detail about SQL. If you are not familiar with it, the `SQLite` docs describe the [language](#).

Connect to the Database

The first thing to do when working with a SQLite database (and most other Python database libraries) is to create a connection to it. Any queries and operations are performed using the connection, which is closed after the work is finished.

In web applications this connection is typically tied to the request. It is created at some point when handling a request, and closed before the response is sent.

Listing 4: flaskr/db.py

```
import sqlite3

import click
from flask import current_app, g

def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(
            current_app.config['DATABASE'],
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row

    return g.db

def close_db(e=None):
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

`g` is a special object that is unique for each request. It is used to store data that might be accessed by multiple functions during the request. The connection is stored and reused instead of creating a new connection if `get_db` is called a second time in the same request.

`current_app` is another special object that points to the Flask application handling the request. Since you used an application factory, there is no application object when writing the rest of your code. `get_db` will be called when the application has been created and is handling a request, so `current_app` can be used.

`sqlite3.connect()` establishes a connection to the file pointed at by the `DATABASE` configuration key. This file doesn't have to exist yet, and won't until you initialize the database later.

`sqlite3.Row` tells the connection to return rows that behave like dicts. This allows accessing the columns by name.

`close_db` checks if a connection was created by checking if `g.db` was set. If the connection exists, it is closed. Further down you will tell your application about the `close_db` function in the application factory so that it is called after each request.

Create the Tables

In SQLite, data is stored in *tables* and *columns*. These need to be created before you can store and retrieve data. Flaskr will store users in the `user` table, and posts in the `post` table. Create a file with the SQL commands needed to create empty tables:

Listing 5: flaskr/schema.sql

```
DROP TABLE IF EXISTS user;
DROP TABLE IF EXISTS post;

CREATE TABLE user (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT UNIQUE NOT NULL,
  password TEXT NOT NULL
);

CREATE TABLE post (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  author_id INTEGER NOT NULL,
  created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  title TEXT NOT NULL,
  body TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES user (id)
);
```

Add the Python functions that will run these SQL commands to the `db.py` file:

Listing 6: flaskr/db.py

```
def init_db():
    db = get_db()

    with current_app.open_resource('schema.sql') as f:
        db.executescript(f.read().decode('utf8'))

@click.command('init-db')
def init_db_command():
    """Clear the existing data and create new tables."""
    init_db()
    click.echo('Initialized the database.')
```

`open_resource()` opens a file relative to the flaskr package, which is useful since you won't necessarily know where that location is when deploying the application later. `get_db` returns a database connection, which is used to execute the commands read from the file.

`click.command()` defines a command line command called `init-db` that calls the `init_db` function and shows a success message to the user. You can read *Command Line Interface* to learn more about writing commands.

Register with the Application

The `close_db` and `init_db_command` functions need to be registered with the application instance; otherwise, they won't be used by the application. However, since you're using a factory function, that instance isn't available when writing the functions. Instead, write a function that takes an application and does the registration.

Listing 7: `flaskr/db.py`

```
def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)
```

`app.teardown_appcontext()` tells Flask to call that function when cleaning up after returning the response.

`app.cli.add_command()` adds a new command that can be called with the `flask` command.

Import and call this function from the factory. Place the new code at the end of the factory function before returning the app.

Listing 8: `flaskr/__init__.py`

```
def create_app():
    app = ...
    # existing code omitted

    from . import db
    db.init_app(app)

    return app
```

Initialize the Database File

Now that `init-db` has been registered with the app, it can be called using the `flask` command, similar to the `run` command from the previous page.

Note: If you're still running the server from the previous page, you can either stop the server, or run this command in a new terminal. If you use a new terminal, remember to change to your project directory and activate the env as described in *Installation*.

Run the `init-db` command:

```
$ flask --app flaskr init-db
Initialized the database.
```

There will now be a `flaskr.sqlite` file in the instance folder in your project.

Continue to *Blueprints and Views*.

1.3.4 Blueprints and Views

A view function is the code you write to respond to requests to your application. Flask uses patterns to match the incoming request URL to the view that should handle it. The view returns data that Flask turns into an outgoing response. Flask can also go the other direction and generate a URL to a view based on its name and arguments.

Create a Blueprint

A *Blueprint* is a way to organize a group of related views and other code. Rather than registering views and other code directly with an application, they are registered with a blueprint. Then the blueprint is registered with the application when it is available in the factory function.

Flaskr will have two blueprints, one for authentication functions and one for the blog posts functions. The code for each blueprint will go in a separate module. Since the blog needs to know about authentication, you'll write the authentication one first.

Listing 9: flaskr/auth.py

```
import functools

from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_for
)
from werkzeug.security import check_password_hash, generate_password_hash

from flaskr.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

This creates a *Blueprint* named 'auth'. Like the application object, the blueprint needs to know where it's defined, so `__name__` is passed as the second argument. The `url_prefix` will be prepended to all the URLs associated with the blueprint.

Import and register the blueprint from the factory using `app.register_blueprint()`. Place the new code at the end of the factory function before returning the app.

Listing 10: flaskr/___init___.py

```
def create_app():
    app = ...
    # existing code omitted

    from . import auth
    app.register_blueprint(auth.bp)

    return app
```

The authentication blueprint will have views to register new users and to log in and log out.

The First View: Register

When the user visits the `/auth/register` URL, the `register` view will return `HTML` with a form for them to fill out. When they submit the form, it will validate their input and either show the form again with an error message or create the new user and go to the login page.

For now you will just write the view code. On the next page, you'll write templates to generate the `HTML` form.

Listing 11: flaskr/auth.py

```
@bp.route('/register', methods=('GET', 'POST'))
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'

        if error is None:
            try:
                db.execute(
                    "INSERT INTO user (username, password) VALUES (?, ?)",
                    (username, generate_password_hash(password)),
                )
                db.commit()
            except db.IntegrityError:
                error = f"User {username} is already registered."
            else:
                return redirect(url_for("auth.login"))

        flash(error)

    return render_template('auth/register.html')
```

Here's what the `register` view function is doing:

1. `@bp.route` associates the URL `/register` with the `register` view function. When Flask receives a request to `/auth/register`, it will call the `register` view and use the return value as the response.
2. If the user submitted the form, `request.method` will be `'POST'`. In this case, start validating the input.
3. `request.form` is a special type of `dict` mapping submitted form keys and values. The user will input their username and password.
4. Validate that username and password are not empty.
5. If validation succeeds, insert the new user data into the database.
 - `db.execute` takes a SQL query with `?` placeholders for any user input, and a tuple of values to replace the placeholders with. The database library will take care of escaping the values so you are not vulnerable to a *SQL injection attack*.
 - For security, passwords should never be stored in the database directly. Instead, `generate_password_hash()` is used to securely hash the password, and that hash is stored. Since this query modifies data, `db.commit()` needs to be called afterwards to save the changes.
 - An `sqlite3.IntegrityError` will occur if the username already exists, which should be shown to the user as another validation error.
6. After storing the user, they are redirected to the login page. `url_for()` generates the URL for the login view based on its name. This is preferable to writing the URL directly as it allows you to change the URL later without changing all code that links to it. `redirect()` generates a redirect response to the generated URL.
7. If validation fails, the error is shown to the user. `flash()` stores messages that can be retrieved when rendering the template.
8. When the user initially navigates to `auth/register`, or there was a validation error, an HTML page with the registration form should be shown. `render_template()` will render a template containing the HTML, which you'll write in the next step of the tutorial.

Login

This view follows the same pattern as the `register` view above.

Listing 12: `flaskr/auth.py`

```
@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.execute(
            'SELECT * FROM user WHERE username = ?', (username,)
        ).fetchone()

        if user is None:
            error = 'Incorrect username.'
        elif not check_password_hash(user['password'], password):
            error = 'Incorrect password.'

        if error is None:
            session.clear()
```

(continues on next page)

(continued from previous page)

```
    session['user_id'] = user['id']
    return redirect(url_for('index'))

flash(error)

return render_template('auth/login.html')
```

There are a few differences from the `register` view:

1. The user is queried first and stored in a variable for later use. `fetchone()` returns one row from the query. If the query returned no results, it returns `None`. Later, `fetchall()` will be used, which returns a list of all results.
2. `check_password_hash()` hashes the submitted password in the same way as the stored hash and securely compares them. If they match, the password is valid.
3. `session` is a `dict` that stores data across requests. When validation succeeds, the user's `id` is stored in a new session. The data is stored in a `cookie` that is sent to the browser, and the browser then sends it back with subsequent requests. Flask securely *signs* the data so that it can't be tampered with.

Now that the user's `id` is stored in the `session`, it will be available on subsequent requests. At the beginning of each request, if a user is logged in their information should be loaded and made available to other views.

Listing 13: flaskr/auth.py

```
@bp.before_app_request
def load_logged_in_user():
    user_id = session.get('user_id')

    if user_id is None:
        g.user = None
    else:
        g.user = get_db().execute(
            'SELECT * FROM user WHERE id = ?', (user_id,)
        ).fetchone()
```

`bp.before_app_request()` registers a function that runs before the view function, no matter what URL is requested. `load_logged_in_user` checks if a user `id` is stored in the `session` and gets that user's data from the database, storing it on `g.user`, which lasts for the length of the request. If there is no user `id`, or if the `id` doesn't exist, `g.user` will be `None`.

Logout

To log out, you need to remove the user `id` from the `session`. Then `load_logged_in_user` won't load a user on subsequent requests.

Listing 14: flaskr/auth.py

```
@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

Require Authentication in Other Views

Creating, editing, and deleting blog posts will require a user to be logged in. A *decorator* can be used to check this for each view it's applied to.

Listing 15: flaskr/auth.py

```
def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))

        return view(**kwargs)

    return wrapped_view
```

This decorator returns a new view function that wraps the original view it's applied to. The new function checks if a user is loaded and redirects to the login page otherwise. If a user is loaded the original view is called and continues normally. You'll use this decorator when writing the blog views.

Endpoints and URLs

The `url_for()` function generates the URL to a view based on a name and arguments. The name associated with a view is also called the *endpoint*, and by default it's the same as the name of the view function.

For example, the `hello()` view that was added to the app factory earlier in the tutorial has the name 'hello' and can be linked to with `url_for('hello')`. If it took an argument, which you'll see later, it would be linked to using `url_for('hello', who='World')`.

When using a blueprint, the name of the blueprint is prepended to the name of the function, so the endpoint for the login function you wrote above is 'auth.login' because you added it to the 'auth' blueprint.

Continue to *Templates*.

1.3.5 Templates

You've written the authentication views for your application, but if you're running the server and try to go to any of the URLs, you'll see a `TemplateNotFound` error. That's because the views are calling `render_template()`, but you haven't written the templates yet. The template files will be stored in the `templates` directory inside the `flaskr` package.

Templates are files that contain static data as well as placeholders for dynamic data. A template is rendered with specific data to produce a final document. Flask uses the `Jinja` template library to render templates.

In your application, you will use templates to render `HTML` which will display in the user's browser. In Flask, Jinja is configured to *autoescape* any data that is rendered in `HTML` templates. This means that it's safe to render user input; any characters they've entered that could mess with the `HTML`, such as `<` and `>` will be *escaped* with *safe* values that look the same in the browser but don't cause unwanted effects.

Jinja looks and behaves mostly like Python. Special delimiters are used to distinguish Jinja syntax from the static data in the template. Anything between `{{` and `}}` is an expression that will be output to the final document. `{%` and `%}` denotes a control flow statement like `if` and `for`. Unlike Python, blocks are denoted by start and end tags rather than indentation since static text within a block could change indentation.

The Base Layout

Each page in the application will have the same basic layout around a different body. Instead of writing the entire HTML structure in each template, each template will *extend* a base template and override specific sections.

Listing 16: flaskr/templates/base.html

```
<!doctype html>
<title>{% block title %}{% endblock %} - Flaskr</title>
<link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
<nav>
  <h1>Flaskr</h1>
  <ul>
    {% if g.user %}
      <li><span>{{ g.user['username'] }}</span>
      <li><a href="{{ url_for('auth.logout') }}">Log Out</a>
    {% else %}
      <li><a href="{{ url_for('auth.register') }}">Register</a>
      <li><a href="{{ url_for('auth.login') }}">Log In</a>
    {% endif %}
  </ul>
</nav>
<section class="content">
  <header>
    {% block header %}{% endblock %}
  </header>
  {% for message in get_flashed_messages() %}
    <div class="flash">{{ message }}</div>
  {% endfor %}
  {% block content %}{% endblock %}
</section>
```

`g` is automatically available in templates. Based on if `g.user` is set (from `load_logged_in_user`), either the username and a log out link are displayed, or links to register and log in are displayed. `url_for()` is also automatically available, and is used to generate URLs to views instead of writing them out manually.

After the page title, and before the content, the template loops over each message returned by `get_flashed_messages()`. You used `flash()` in the views to show error messages, and this is the code that will display them.

There are three blocks defined here that will be overridden in the other templates:

1. `{% block title %}` will change the title displayed in the browser's tab and window title.
2. `{% block header %}` is similar to `title` but will change the title displayed on the page.
3. `{% block content %}` is where the content of each page goes, such as the login form or a blog post.

The base template is directly in the `templates` directory. To keep the others organized, the templates for a blueprint will be placed in a directory with the same name as the blueprint.

Register

Listing 17: flaskr/templates/auth/register.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Register{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Register">
  </form>
{% endblock %}
```

`{% extends 'base.html' %}` tells Jinja that this template should replace the blocks from the base template. All the rendered content must appear inside `{% block %}` tags that override blocks from the base template.

A useful pattern used here is to place `{% block title %}` inside `{% block header %}`. This will set the title block and then output the value of it into the header block, so that both the window and page share the same title without writing it twice.

The input tags are using the `required` attribute here. This tells the browser not to submit the form until those fields are filled in. If the user is using an older browser that doesn't support that attribute, or if they are using something besides a browser to make requests, you still want to validate the data in the Flask view. It's important to always fully validate the data on the server, even if the client does some validation as well.

Log In

This is identical to the register template except for the title and submit button.

Listing 18: flaskr/templates/auth/login.html

```
{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Log In{% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="username">Username</label>
    <input name="username" id="username" required>
    <label for="password">Password</label>
    <input type="password" name="password" id="password" required>
    <input type="submit" value="Log In">
  </form>
{% endblock %}
```

Register A User

Now that the authentication templates are written, you can register a user. Make sure the server is still running (flask run if it's not), then go to <http://127.0.0.1:5000/auth/register>.

Try clicking the “Register” button without filling out the form and see that the browser shows an error message. Try removing the required attributes from the `register.html` template and click “Register” again. Instead of the browser showing an error, the page will reload and the error from `flash()` in the view will be shown.

Fill out a username and password and you'll be redirected to the login page. Try entering an incorrect username, or the correct username and incorrect password. If you log in you'll get an error because there's no `index` view to redirect to yet.

Continue to [Static Files](#).

1.3.6 Static Files

The authentication views and templates work, but they look very plain right now. Some CSS can be added to add style to the HTML layout you constructed. The style won't change, so it's a *static* file rather than a template.

Flask automatically adds a `static` view that takes a path relative to the `flaskr/static` directory and serves it. The `base.html` template already has a link to the `style.css` file:

```
{{ url_for('static', filename='style.css') }}
```

Besides CSS, other types of static files might be files with JavaScript functions, or a logo image. They are all placed under the `flaskr/static` directory and referenced with `url_for('static', filename='...')`.

This tutorial isn't focused on how to write CSS, so you can just copy the following into the `flaskr/static/style.css` file:

Listing 19: `flaskr/static/style.css`

```
html { font-family: sans-serif; background: #eee; padding: 1rem; }
body { max-width: 960px; margin: 0 auto; background: white; }
h1 { font-family: serif; color: #377ba8; margin: 1rem 0; }
a { color: #377ba8; }
hr { border: none; border-top: 1px solid lightgray; }
nav { background: lightgray; display: flex; align-items: center; padding: 0 0.5rem; }
nav h1 { flex: auto; margin: 0; }
nav h1 a { text-decoration: none; padding: 0.25rem 0.5rem; }
nav ul { display: flex; list-style: none; margin: 0; padding: 0; }
nav ul li a, nav ul li span, header .action { display: block; padding: 0.5rem; }
.content { padding: 0 1rem 1rem; }
.content > header { border-bottom: 1px solid lightgray; display: flex; align-items: flex-
  end; }
.content > header h1 { flex: auto; margin: 1rem 0 0.25rem 0; }
.flash { margin: 1em 0; padding: 1em; background: #cae6f6; border: 1px solid #377ba8; }
.post > header { display: flex; align-items: flex-end; font-size: 0.85em; }
.post > header > div:first-of-type { flex: auto; }
.post > header h1 { font-size: 1.5em; margin-bottom: 0; }
.post .about { color: slategray; font-style: italic; }
.post .body { white-space: pre-line; }
.content:last-child { margin-bottom: 0; }
.content form { margin: 1em 0; display: flex; flex-direction: column; }
```

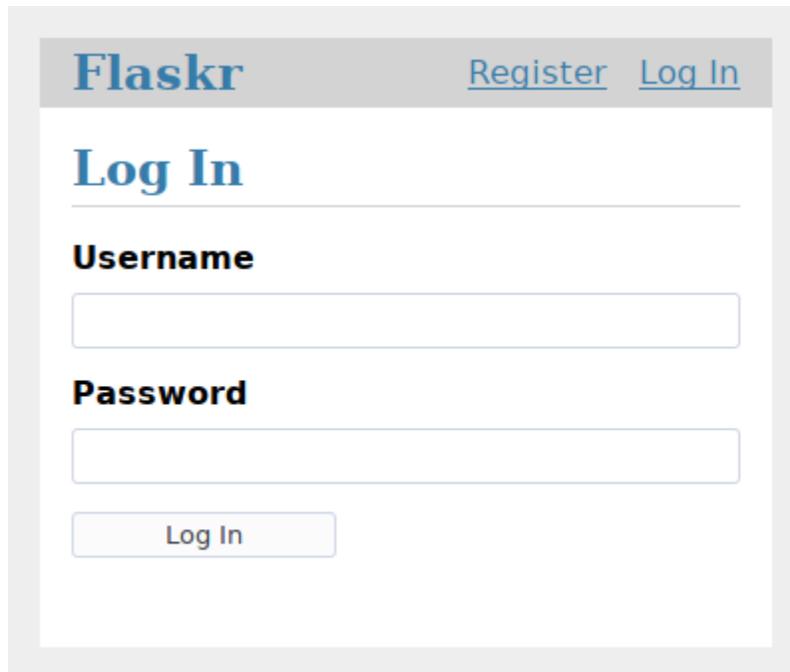
(continues on next page)

(continued from previous page)

```
.content label { font-weight: bold; margin-bottom: 0.5em; }
.content input, .content textarea { margin-bottom: 1em; }
.content textarea { min-height: 12em; resize: vertical; }
input.danger { color: #cc2f2e; }
input[type=submit] { align-self: start; min-width: 10em; }
```

You can find a less compact version of `style.css` in the [example code](#).

Go to <http://127.0.0.1:5000/auth/login> and the page should look like the screenshot below.



You can read more about CSS from [Mozilla's documentation](#). If you change a static file, refresh the browser page. If the change doesn't show up, try clearing your browser's cache.

Continue to [Blog Blueprint](#).

1.3.7 Blog Blueprint

You'll use the same techniques you learned about when writing the authentication blueprint to write the blog blueprint. The blog should list all posts, allow logged in users to create posts, and allow the author of a post to edit or delete it.

As you implement each view, keep the development server running. As you save your changes, try going to the URL in your browser and testing them out.

The Blueprint

Define the blueprint and register it in the application factory.

Listing 20: flaskr/blog.py

```
from flask import (
    Blueprint, flash, g, redirect, render_template, request, url_for
)
from werkzeug.exceptions import abort

from flaskr.auth import login_required
from flaskr.db import get_db

bp = Blueprint('blog', __name__)
```

Import and register the blueprint from the factory using `app.register_blueprint()`. Place the new code at the end of the factory function before returning the app.

Listing 21: flaskr/__init__.py

```
def create_app():
    app = ...
    # existing code omitted

    from . import blog
    app.register_blueprint(blog.bp)
    app.add_url_rule('/', endpoint='index')

    return app
```

Unlike the auth blueprint, the blog blueprint does not have a `url_prefix`. So the index view will be at `/`, the create view at `/create`, and so on. The blog is the main feature of Flaskr, so it makes sense that the blog index will be the main index.

However, the endpoint for the index view defined below will be `blog.index`. Some of the authentication views referred to a plain index endpoint. `app.add_url_rule()` associates the endpoint name 'index' with the `/` url so that `url_for('index')` or `url_for('blog.index')` will both work, generating the same `/` URL either way.

In another application you might give the blog blueprint a `url_prefix` and define a separate index view in the application factory, similar to the hello view. Then the index and `blog.index` endpoints and URLs would be different.

Index

The index will show all of the posts, most recent first. A JOIN is used so that the author information from the user table is available in the result.

Listing 22: flaskr/blog.py

```
@bp.route('/')
def index():
    db = get_db()
    posts = db.execute(
        'SELECT p.id, title, body, created, author_id, username'
```

(continues on next page)

(continued from previous page)

```

    ' FROM post p JOIN user u ON p.author_id = u.id'
    ' ORDER BY created DESC'
).fetchall()
return render_template('blog/index.html', posts=posts)

```

Listing 23: flaskr/templates/blog/index.html

```

{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}Posts{% endblock %}</h1>
{% if g.user %}
  <a class="action" href="{{ url_for('blog.create') }}">New</a>
{% endif %}
{% endblock %}

{% block content %}
{% for post in posts %}
  <article class="post">
    <header>
      <div>
        <h1>{{ post['title'] }}</h1>
        <div class="about">by {{ post['username'] }} on {{ post['created'].strftime('
->%Y-%m-%d') }}</div>
      </div>
      {% if g.user['id'] == post['author_id'] %}
        <a class="action" href="{{ url_for('blog.update', id=post['id']) }}">Edit</a>
      {% endif %}
    </header>
    <p class="body">{{ post['body'] }}</p>
  </article>
  {% if not loop.last %}
    <hr>
  {% endif %}
{% endfor %}
{% endblock %}

```

When a user is logged in, the header block adds a link to the create view. When the user is the author of a post, they'll see an "Edit" link to the update view for that post. `loop.last` is a special variable available inside Jinja for loops. It's used to display a line after each post except the last one, to visually separate them.

Create

The create view works the same as the auth `register` view. Either the form is displayed, or the posted data is validated and the post is added to the database or an error is shown.

The `login_required` decorator you wrote earlier is used on the blog views. A user must be logged in to visit these views, otherwise they will be redirected to the login page.

Listing 24: flaskr/blog.py

```

@bp.route('/create', methods=('GET', 'POST'))
@login_required
def create():
    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'INSERT INTO post (title, body, author_id)'
                ' VALUES (?, ?, ?)',
                (title, body, g.user['id'])
            )
            db.commit()
            return redirect(url_for('blog.index'))

    return render_template('blog/create.html')

```

Listing 25: flaskr/templates/blog/create.html

```

{% extends 'base.html' %}

{% block header %}
<h1>{% block title %}New Post{% endblock %}</h1>
{% endblock %}

{% block content %}
<form method="post">
  <label for="title">Title</label>
  <input name="title" id="title" value="{{ request.form['title'] }}" required>
  <label for="body">Body</label>
  <textarea name="body" id="body">{{ request.form['body'] }}</textarea>
  <input type="submit" value="Save">
</form>
{% endblock %}

```

Update

Both the update and delete views will need to fetch a post by id and check if the author matches the logged in user. To avoid duplicating code, you can write a function to get the post and call it from each view.

Listing 26: flaskr/blog.py

```
def get_post(id, check_author=True):
    post = get_db().execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' WHERE p.id = ?',
        (id,)
    ).fetchone()

    if post is None:
        abort(404, f"Post id {id} doesn't exist.")

    if check_author and post['author_id'] != g.user['id']:
        abort(403)

    return post
```

`abort()` will raise a special exception that returns an HTTP status code. It takes an optional message to show with the error, otherwise a default message is used. 404 means “Not Found”, and 403 means “Forbidden”. (401 means “Unauthorized”, but you redirect to the login page instead of returning that status.)

The `check_author` argument is defined so that the function can be used to get a post without checking the author. This would be useful if you wrote a view to show an individual post on a page, where the user doesn’t matter because they’re not modifying the post.

Listing 27: flaskr/blog.py

```
@bp.route('/<int:id>/update', methods=('GET', 'POST'))
@login_required
def update(id):
    post = get_post(id)

    if request.method == 'POST':
        title = request.form['title']
        body = request.form['body']
        error = None

        if not title:
            error = 'Title is required.'

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'UPDATE post SET title = ?, body = ?'
                ' WHERE id = ?',
                (title, body, id)
```

(continues on next page)

```

    )
    db.commit()
    return redirect(url_for('blog.index'))

return render_template('blog/update.html', post=post)

```

Unlike the views you've written so far, the `update` function takes an argument, `id`. That corresponds to the `<int:id>` in the route. A real URL will look like `/1/update`. Flask will capture the `1`, ensure it's an `int`, and pass it as the `id` argument. If you don't specify `int:` and instead do `<id>`, it will be a string. To generate a URL to the update page, `url_for()` needs to be passed the `id` so it knows what to fill in: `url_for('blog.update', id=post['id'])`. This is also in the `index.html` file above.

The `create` and `update` views look very similar. The main difference is that the `update` view uses a `post` object and an `UPDATE` query instead of an `INSERT`. With some clever refactoring, you could use one view and template for both actions, but for the tutorial it's clearer to keep them separate.

Listing 28: `flaskr/templates/blog/update.html`

```

{% extends 'base.html' %}

{% block header %}
  <h1>{% block title %}Edit "{{ post['title'] }}" {% endblock %}</h1>
{% endblock %}

{% block content %}
  <form method="post">
    <label for="title">Title</label>
    <input name="title" id="title"
      value="{{ request.form['title'] or post['title'] }}" required>
    <label for="body">Body</label>
    <textarea name="body" id="body">{{ request.form['body'] or post['body'] }}</textarea>
    <input type="submit" value="Save">
  </form>
  <hr>
  <form action="{{ url_for('blog.delete', id=post['id']) }}" method="post">
    <input class="danger" type="submit" value="Delete" onclick="return confirm('Are you
↪ sure?');">
  </form>
{% endblock %}

```

This template has two forms. The first posts the edited data to the current page (`/<id>/update`). The other form contains only a button and specifies an `action` attribute that posts to the delete view instead. The button uses some JavaScript to show a confirmation dialog before submitting.

The pattern `{{ request.form['title'] or post['title'] }}` is used to choose what data appears in the form. When the form hasn't been submitted, the original `post` data appears, but if invalid form data was posted you want to display that so the user can fix the error, so `request.form` is used instead. `request` is another variable that's automatically available in templates.

Delete

The delete view doesn't have its own template, the delete button is part of `update.html` and posts to the `/<id>/delete` URL. Since there is no template, it will only handle the POST method and then redirect to the `index` view.

Listing 29: `flaskr/blog.py`

```
@bp.route('/<int:id>/delete', methods=('POST',))
@login_required
def delete(id):
    get_post(id)
    db = get_db()
    db.execute('DELETE FROM post WHERE id = ?', (id,))
    db.commit()
    return redirect(url_for('blog.index'))
```

Congratulations, you've now finished writing your application! Take some time to try out everything in the browser. However, there's still more to do before the project is complete.

Continue to *Make the Project Installable*.

1.3.8 Make the Project Installable

Making your project installable means that you can build a *wheel* file and install that in another environment, just like you installed Flask in your project's environment. This makes deploying your project the same as installing any other library, so you're using all the standard Python tools to manage everything.

Installing also comes with other benefits that might not be obvious from the tutorial or as a new Python user, including:

- Currently, Python and Flask understand how to use the `flaskr` package only because you're running from your project's directory. Installing means you can import it no matter where you run from.
- You can manage your project's dependencies just like other packages do, so `pip install yourproject.whl` installs them.
- Test tools can isolate your test environment from your development environment.

Note: This is being introduced late in the tutorial, but in your future projects you should always start with this.

Describe the Project

The `pyproject.toml` file describes your project and how to build it.

Listing 30: `pyproject.toml`

```
[project]
name = "flaskr"
version = "1.0.0"
dependencies = [
    "flask",
]

[build-system]
```

(continues on next page)

(continued from previous page)

```
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

The setuptools build backend needs another file named `MANIFEST.in` to tell it about non-Python files to include.

Listing 31: `MANIFEST.in`

```
include flaskr/schema.sql
graft flaskr/static
graft flaskr/templates
global-exclude *.pyc
```

This tells the build to copy everything in the `static` and `templates` directories, and the `schema.sql` file, but to exclude all bytecode files.

See the official [Packaging tutorial](#) and [detailed guide](#) for more explanation of the files and options used.

Install the Project

Use `pip` to install your project in the virtual environment.

```
$ pip install -e .
```

This tells `pip` to find `pyproject.toml` in the current directory and install the project in *editable* or *development* mode. Editable mode means that as you make changes to your local code, you'll only need to re-install if you change the metadata about the project, such as its dependencies.

You can observe that the project is now installed with `pip list`.

```
$ pip list
```

Package	Version	Location
click	6.7	
Flask	1.0	
flaskr	1.0.0	/home/user/Projects/flask-tutorial
itsdangerous	0.24	
Jinja2	2.10	
MarkupSafe	1.0	
pip	9.0.3	
setuptools	39.0.1	
Werkzeug	0.14.1	
wheel	0.30.0	

Nothing changes from how you've been running your project so far. `--app` is still set to `flaskr` and `flask run` still runs the application, but you can call it from anywhere, not just the `flask-tutorial` directory.

Continue to [Test Coverage](#).

1.3.9 Test Coverage

Writing unit tests for your application lets you check that the code you wrote works the way you expect. Flask provides a test client that simulates requests to the application and returns the response data.

You should test as much of your code as possible. Code in functions only runs when the function is called, and code in branches, such as `if` blocks, only runs when the condition is met. You want to make sure that each function is tested with data that covers each branch.

The closer you get to 100% coverage, the more comfortable you can be that making a change won't unexpectedly change other behavior. However, 100% coverage doesn't guarantee that your application doesn't have bugs. In particular, it doesn't test how the user interacts with the application in the browser. Despite this, test coverage is an important tool to use during development.

Note: This is being introduced late in the tutorial, but in your future projects you should test as you develop.

You'll use `pytest` and `coverage` to test and measure your code. Install them both:

```
$ pip install pytest coverage
```

Setup and Fixtures

The test code is located in the `tests` directory. This directory is *next to* the `flaskr` package, not inside it. The `tests/conftest.py` file contains setup functions called *fixtures* that each test will use. Tests are in Python modules that start with `test_`, and each test function in those modules also starts with `test_`.

Each test will create a new temporary database file and populate some data that will be used in the tests. Write a SQL file to insert that data.

Listing 32: `tests/data.sql`

```
INSERT INTO user (username, password)
VALUES
  ('test', 'pbkdf2:sha256:50000$TCI4GzcX
↪$0de171a4f4dac32e3364c7ddc7c14f3e2fa61f2d17574483f7ffbb431b4acb2f'),
  ('other', 'pbkdf2:sha256:50000$kJPKsz6N
↪$d2d4784f1b030a9761f5cceaeca413f27f2ecb76d6168407af962ddce849f79');

INSERT INTO post (title, body, author_id, created)
VALUES
  ('test title', 'test' || x'0a' || 'body', 1, '2018-01-01 00:00:00');
```

The app fixture will call the factory and pass `test_config` to configure the application and database for testing instead of using your local development configuration.

Listing 33: `tests/conftest.py`

```
import os
import tempfile

import pytest
from flask import create_app
from flaskr.db import get_db, init_db
```

(continues on next page)

```
with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as f:
    _data_sql = f.read().decode('utf8')

@pytest.fixture
def app():
    db_fd, db_path = tempfile.mkstemp()

    app = create_app({
        'TESTING': True,
        'DATABASE': db_path,
    })

    with app.app_context():
        init_db()
        get_db().executescript(_data_sql)

    yield app

    os.close(db_fd)
    os.unlink(db_path)

@pytest.fixture
def client(app):
    return app.test_client()

@pytest.fixture
def runner(app):
    return app.test_cli_runner()
```

`tempfile.mkstemp()` creates and opens a temporary file, returning the file descriptor and the path to it. The `DATABASE` path is overridden so it points to this temporary path instead of the instance folder. After setting the path, the database tables are created and the test data is inserted. After the test is over, the temporary file is closed and removed.

`TESTING` tells Flask that the app is in test mode. Flask changes some internal behavior so it's easier to test, and other extensions can also use the flag to make testing them easier.

The `client` fixture calls `app.test_client()` with the application object created by the `app` fixture. Tests will use the client to make requests to the application without running the server.

The `runner` fixture is similar to `client`. `app.test_cli_runner()` creates a runner that can call the Click commands registered with the application.

Pytest uses fixtures by matching their function names with the names of arguments in the test functions. For example, the `test_hello` function you'll write next takes a `client` argument. Pytest matches that with the `client` fixture function, calls it, and passes the returned value to the test function.

Factory

There's not much to test about the factory itself. Most of the code will be executed for each test already, so if something fails the other tests will notice.

The only behavior that can change is passing test config. If config is not passed, there should be some default configuration, otherwise the configuration should be overridden.

Listing 34: tests/test_factory.py

```
from flaskr import create_app

def test_config():
    assert not create_app().testing
    assert create_app({'TESTING': True}).testing

def test_hello(client):
    response = client.get('/hello')
    assert response.data == b'Hello, World!'
```

You added the hello route as an example when writing the factory at the beginning of the tutorial. It returns “Hello, World!”, so the test checks that the response data matches.

Database

Within an application context, `get_db` should return the same connection each time it's called. After the context, the connection should be closed.

Listing 35: tests/test_db.py

```
import sqlite3

import pytest
from flaskr.db import get_db

def test_get_close_db(app):
    with app.app_context():
        db = get_db()
        assert db is get_db()

    with pytest.raises(sqlite3.ProgrammingError) as e:
        db.execute('SELECT 1')

    assert 'closed' in str(e.value)
```

The `init-db` command should call the `init_db` function and output a message.

Listing 36: tests/test_db.py

```
def test_init_db_command(runner, monkeypatch):
    class Recorder(object):
```

(continues on next page)

(continued from previous page)

```

        called = False

    def fake_init_db():
        Recorder.called = True

monkeypatch.setattr('flaskr.db.init_db', fake_init_db)
result = runner.invoke(args=['init-db'])
assert 'Initialized' in result.output
assert Recorder.called

```

This test uses Pytest’s monkeypatch fixture to replace the `init_db` function with one that records that it’s been called. The runner fixture you wrote above is used to call the `init-db` command by name.

Authentication

For most of the views, a user needs to be logged in. The easiest way to do this in tests is to make a POST request to the login view with the client. Rather than writing that out every time, you can write a class with methods to do that, and use a fixture to pass it the client for each test.

Listing 37: tests/conftest.py

```

class AuthActions(object):
    def __init__(self, client):
        self._client = client

    def login(self, username='test', password='test'):
        return self._client.post(
            '/auth/login',
            data={'username': username, 'password': password}
        )

    def logout(self):
        return self._client.get('/auth/logout')

@pytest.fixture
def auth(client):
    return AuthActions(client)

```

With the `auth` fixture, you can call `auth.login()` in a test to log in as the `test` user, which was inserted as part of the test data in the `app` fixture.

The `register` view should render successfully on GET. On POST with valid form data, it should redirect to the login URL and the user’s data should be in the database. Invalid data should display error messages.

Listing 38: tests/test_auth.py

```

import pytest
from flask import g, session
from flaskr.db import get_db

```

(continues on next page)

(continued from previous page)

```

def test_register(client, app):
    assert client.get('/auth/register').status_code == 200
    response = client.post(
        '/auth/register', data={'username': 'a', 'password': 'a'}
    )
    assert response.headers["Location"] == "/auth/login"

    with app.app_context():
        assert get_db().execute(
            "SELECT * FROM user WHERE username = 'a'",
        ).fetchone() is not None

@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('', '', b'Username is required.'),
    ('a', '', b'Password is required.'),
    ('test', 'test', b'already registered'),
))
def test_register_validate_input(client, username, password, message):
    response = client.post(
        '/auth/register',
        data={'username': username, 'password': password}
    )
    assert message in response.data

```

`client.get()` makes a GET request and returns the *Response* object returned by Flask. Similarly, `client.post()` makes a POST request, converting the data dict into form data.

To test that the page renders successfully, a simple request is made and checked for a 200 OK *status_code*. If rendering failed, Flask would return a 500 Internal Server Error code.

headers will have a `Location` header with the login URL when the register view redirects to the login view.

data contains the body of the response as bytes. If you expect a certain value to render on the page, check that it's in data. Bytes must be compared to bytes. If you want to compare text, use `get_data(as_text=True)` instead.

`pytest.mark.parametrize` tells Pytest to run the same test function with different arguments. You use it here to test different invalid input and error messages without writing the same code three times.

The tests for the login view are very similar to those for `register`. Rather than testing the data in the database, *session* should have `user_id` set after logging in.

Listing 39: tests/test_auth.py

```

def test_login(client, auth):
    assert client.get('/auth/login').status_code == 200
    response = auth.login()
    assert response.headers["Location"] == "/"

    with client:
        client.get('/')
        assert session['user_id'] == 1
        assert g.user['username'] == 'test'

```

(continues on next page)

(continued from previous page)

```

@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('a', 'test', b'Incorrect username.'),
    ('test', 'a', b'Incorrect password.'),
))
def test_login_validate_input(auth, username, password, message):
    response = auth.login(username, password)
    assert message in response.data

```

Using `client` in a `with` block allows accessing context variables such as `session` after the response is returned. Normally, accessing `session` outside of a request would raise an error.

Testing logout is the opposite of login. `session` should not contain `user_id` after logging out.

Listing 40: tests/test_auth.py

```

def test_logout(client, auth):
    auth.login()

    with client:
        auth.logout()
        assert 'user_id' not in session

```

Blog

All the blog views use the `auth` fixture you wrote earlier. Call `auth.login()` and subsequent requests from the client will be logged in as the test user.

The `index` view should display information about the post that was added with the test data. When logged in as the author, there should be a link to edit the post.

You can also test some more authentication behavior while testing the `index` view. When not logged in, each page shows links to log in or register. When logged in, there's a link to log out.

Listing 41: tests/test_blog.py

```

import pytest
from flaskr.db import get_db

def test_index(client, auth):
    response = client.get('/')
    assert b"Log In" in response.data
    assert b"Register" in response.data

    auth.login()
    response = client.get('/')
    assert b'Log Out' in response.data
    assert b'test title' in response.data
    assert b'by test on 2018-01-01' in response.data
    assert b'test\nbody' in response.data
    assert b'href="/1/update"' in response.data

```

A user must be logged in to access the create, update, and delete views. The logged in user must be the author of the post to access update and delete, otherwise a 403 Forbidden status is returned. If a post with the given id

doesn't exist, update and delete should return 404 Not Found.

Listing 42: tests/test_blog.py

```

@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
    '/1/delete',
))
def test_login_required(client, path):
    response = client.post(path)
    assert response.headers["Location"] == "/auth/login"

def test_author_required(app, client, auth):
    # change the post author to another user
    with app.app_context():
        db = get_db()
        db.execute('UPDATE post SET author_id = 2 WHERE id = 1')
        db.commit()

    auth.login()
    # current user can't modify other user's post
    assert client.post('/1/update').status_code == 403
    assert client.post('/1/delete').status_code == 403
    # current user doesn't see edit link
    assert b'href="/1/update"' not in client.get('/').data

@pytest.mark.parametrize('path', (
    '/2/update',
    '/2/delete',
))
def test_exists_required(client, auth, path):
    auth.login()
    assert client.post(path).status_code == 404

```

The create and update views should render and return a 200 OK status for a GET request. When valid data is sent in a POST request, create should insert the new post data into the database, and update should modify the existing data. Both pages should show an error message on invalid data.

Listing 43: tests/test_blog.py

```

def test_create(client, auth, app):
    auth.login()
    assert client.get('/create').status_code == 200
    client.post('/create', data={'title': 'created', 'body': ''})

    with app.app_context():
        db = get_db()
        count = db.execute('SELECT COUNT(id) FROM post').fetchone()[0]
        assert count == 2

```

(continues on next page)

(continued from previous page)

```
def test_update(client, auth, app):
    auth.login()
    assert client.get('/1/update').status_code == 200
    client.post('/1/update', data={'title': 'updated', 'body': ''})

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post['title'] == 'updated'

@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
))
def test_create_update_validate(client, auth, path):
    auth.login()
    response = client.post(path, data={'title': '', 'body': ''})
    assert b'Title is required.' in response.data
```

The delete view should redirect to the index URL and the post should no longer exist in the database.

Listing 44: tests/test_blog.py

```
def test_delete(client, auth, app):
    auth.login()
    response = client.post('/1/delete')
    assert response.headers["Location"] == "/"

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post is None
```

Running the Tests

Some extra configuration, which is not required but makes running tests with coverage less verbose, can be added to the project's `pyproject.toml` file.

Listing 45: pyproject.toml

```
[tool.pytest.ini_options]
testpaths = ["tests"]

[tool.coverage.run]
branch = true
source = ["flaskr"]
```

To run the tests, use the `pytest` command. It will find and run all the test functions you've written.

```
$ pytest
```

(continues on next page)

(continued from previous page)

```

===== test session starts =====
platform linux -- Python 3.6.4, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /home/user/Projects/flask-tutorial
collected 23 items

tests/test_auth.py ..... [ 34%]
tests/test_blog.py ..... [ 86%]
tests/test_db.py .. [ 95%]
tests/test_factory.py .. [100%]

===== 24 passed in 0.64 seconds =====

```

If any tests fail, pytest will show the error that was raised. You can run `pytest -v` to get a list of each test function rather than dots.

To measure the code coverage of your tests, use the `coverage` command to run `pytest` instead of running it directly.

```
$ coverage run -m pytest
```

You can either view a simple coverage report in the terminal:

```
$ coverage report
```

Name	Stmts	Miss	Branch	BrPart	Cover
flaskr/__init__.py	21	0	2	0	100%
flaskr/auth.py	54	0	22	0	100%
flaskr/blog.py	54	0	16	0	100%
flaskr/db.py	24	0	4	0	100%
TOTAL	153	0	44	0	100%

An HTML report allows you to see which lines were covered in each file:

```
$ coverage html
```

This generates files in the `htmlcov` directory. Open `htmlcov/index.html` in your browser to see the report.

Continue to [Deploy to Production](#).

1.3.10 Deploy to Production

This part of the tutorial assumes you have a server that you want to deploy your application to. It gives an overview of how to create the distribution file and install it, but won't go into specifics about what server or software to use. You can set up a new environment on your development computer to try out the instructions below, but probably shouldn't use it for hosting a real public application. See [Deploying to Production](#) for a list of many different ways to host your application.

Build and Install

When you want to deploy your application elsewhere, you build a *wheel* (.whl) file. Install and use the build tool to do this.

```
$ pip install build
$ python -m build --wheel
```

You can find the file in `dist/flaskr-1.0.0-py3-none-any.whl`. The file name is in the format of {project name}-{version}-{python tag} -{abi tag}-{platform tag}.

Copy this file to another machine, *set up a new virtualenv*, then install the file with pip.

```
$ pip install flaskr-1.0.0-py3-none-any.whl
```

Pip will install your project along with its dependencies.

Since this is a different machine, you need to run `init-db` again to create the database in the instance folder.

```
$ flask --app flaskr init-db
```

When Flask detects that it's installed (not in editable mode), it uses a different directory for the instance folder. You can find it at `.venv/var/flaskr-instance` instead.

Configure the Secret Key

In the beginning of the tutorial that you gave a default value for `SECRET_KEY`. This should be changed to some random bytes in production. Otherwise, attackers could use the public 'dev' key to modify the session cookie, or anything else that uses the secret key.

You can use the following command to output a random secret key:

```
$ python -c 'import secrets; print(secrets.token_hex())'
'192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

Create the `config.py` file in the instance folder, which the factory will read from if it exists. Copy the generated value into it.

Listing 46: `.venv/var/flaskr-instance/config.py`

```
SECRET_KEY = '192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

You can also set any other necessary configuration here, although `SECRET_KEY` is the only one needed for Flaskr.

Run with a Production Server

When running publicly rather than in development, you should not use the built-in development server (`flask run`). The development server is provided by Werkzeug for convenience, but is not designed to be particularly efficient, stable, or secure.

Instead, use a production WSGI server. For example, to use *Waitress*, first install it in the virtual environment:

```
$ pip install waitress
```

You need to tell Waitress about your application, but it doesn't use `--app` like `flask run` does. You need to tell it to import and call the application factory to get an application object.

```
$ waitress-serve --call 'flaskr:create_app'
```

```
Serving on http://0.0.0.0:8080
```

See [Deploying to Production](#) for a list of many different ways to host your application. Waitress is just an example, chosen for the tutorial because it supports both Windows and Linux. There are many more WSGI servers and deployment options that you may choose for your project.

Continue to [Keep Developing!](#).

1.3.11 Keep Developing!

You've learned about quite a few Flask and Python concepts throughout the tutorial. Go back and review the tutorial and compare your code with the steps you took to get there. Compare your project to the [example project](#), which might look a bit different due to the step-by-step nature of the tutorial.

There's a lot more to Flask than what you've seen so far. Even so, you're now equipped to start developing your own web applications. Check out the [Quickstart](#) for an overview of what Flask can do, then dive into the docs to keep learning. Flask uses [Jinja](#), [Click](#), [Werkzeug](#), and [ItsDangerous](#) behind the scenes, and they all have their own documentation too. You'll also be interested in [Extensions](#) which make tasks like working with the database or validating form data easier and more powerful.

If you want to keep developing your Flaskr project, here are some ideas for what to try next:

- A detail view to show a single post. Click a post's title to go to its page.
- Like / unlike a post.
- Comments.
- Tags. Clicking a tag shows all the posts with that tag.
- A search box that filters the index page by name.
- Paged display. Only show 5 posts per page.
- Upload an image to go along with a post.
- Format posts using Markdown.
- An RSS feed of new posts.

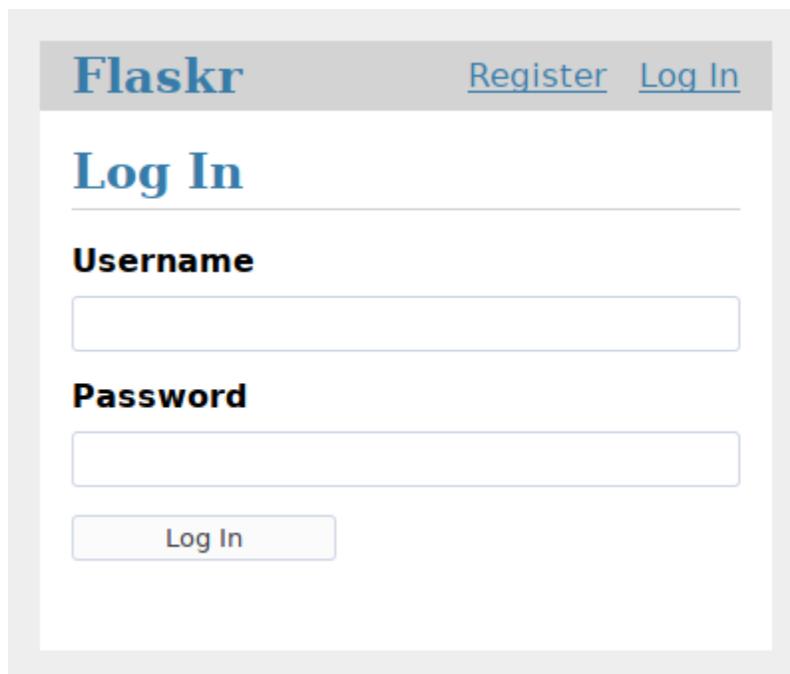
Have fun and make awesome applications!

This tutorial will walk you through creating a basic blog application called Flaskr. Users will be able to register, log in, create posts, and edit or delete their own posts. You will be able to package and install the application on other computers.

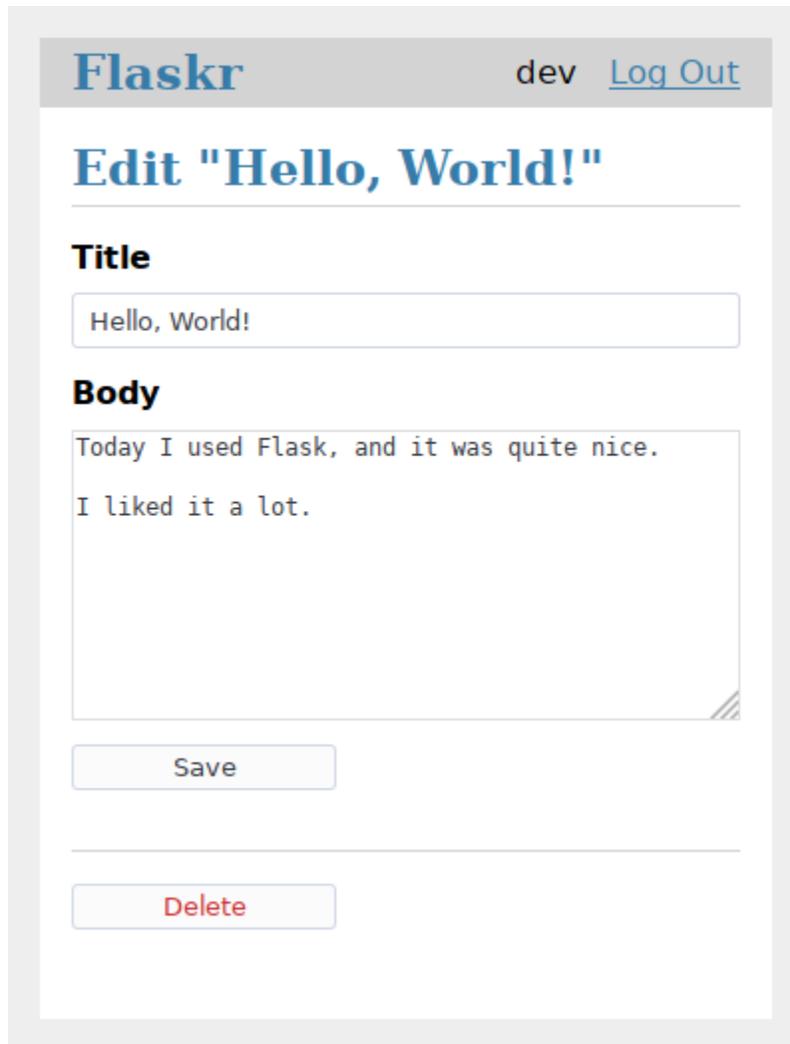


It's assumed that you're already familiar with Python. The [official tutorial](#) in the Python docs is a great way to learn or review first.

While it's designed to give a good starting point, the tutorial doesn't cover all of Flask's features. Check out the [Quickstart](#) for an overview of what Flask can do, then dive into the docs to find out more. The tutorial only uses what's provided by Flask and Python. In another project, you might decide to use [Extensions](#) or other libraries to make some tasks simpler.



Flask is flexible. It doesn't require you to use any particular project or code layout. However, when first starting, it's helpful to use a more structured approach. This means that the tutorial will require a bit of boilerplate up front, but it's done to avoid many common pitfalls that new developers encounter, and it creates a project that's easy to expand on. Once you become more comfortable with Flask, you can step out of this structure and take full advantage of Flask's flexibility.



The screenshot shows a web application interface for editing a post. At the top, there is a header with the word "Flaskr" in blue, followed by "dev" and a "Log Out" link. Below the header, the main heading is "Edit 'Hello, World!'" in blue. Underneath, there are two sections: "Title" and "Body". The "Title" section has a text input field containing "Hello, World!". The "Body" section has a larger text area containing two lines of text: "Today I used Flask, and it was quite nice." and "I liked it a lot.". Below the text area, there are two buttons: a "Save" button and a "Delete" button. The "Delete" button is highlighted in red.

The tutorial project is available as an example in the Flask repository, if you want to compare your project with the final product as you follow the tutorial.

Continue to *Project Layout*.

1.4 Templates

Flask leverages Jinja2 as its template engine. You are obviously free to use a different template engine, but you still have to install Jinja2 to run Flask itself. This requirement is necessary to enable rich extensions. An extension can depend on Jinja2 being present.

This section only gives a very quick introduction into how Jinja2 is integrated into Flask. If you want information on the template engine's syntax itself, head over to the official [Jinja2 Template Documentation](#) for more information.

1.4.1 Jinja Setup

Unless customized, Jinja2 is configured by Flask as follows:

- autoescaping is enabled for all templates ending in `.html`, `.htm`, `.xml`, `.xhtml`, as well as `.svg` when using `render_template()`.
- autoescaping is enabled for all strings when using `render_template_string()`.
- a template has the ability to opt in/out autoescaping with the `{% autoescape %}` tag.
- Flask inserts a couple of global functions and helpers into the Jinja2 context, additionally to the values that are present by default.

1.4.2 Standard Context

The following global variables are available within Jinja2 templates by default:

config

The current configuration object (*flask.Flask.config*)

Changed in version 0.10: This is now always available, even in imported templates.

New in version 0.6.

request

The current request object (*flask.request*). This variable is unavailable if the template was rendered without an active request context.

session

The current session object (*flask.session*). This variable is unavailable if the template was rendered without an active request context.

g

The request-bound object for global variables (*flask.g*). This variable is unavailable if the template was rendered without an active request context.

url_for()

The *flask.url_for()* function.

get_flashed_messages()

The *flask.get_flashed_messages()* function.

The Jinja Context Behavior

These variables are added to the context of variables, they are not global variables. The difference is that by default these will not show up in the context of imported templates. This is partially caused by performance considerations, partially to keep things explicit.

What does this mean for you? If you have a macro you want to import, that needs to access the request object you have two possibilities:

1. you explicitly pass the request to the macro as parameter, or the attribute of the request object you are interested in.
2. you import the macro “with context”.

Importing with context looks like this:

```
{% from '_helpers.html' import my_macro with context %}
```

1.4.3 Controlling Autoescaping

Autoescaping is the concept of automatically escaping special characters for you. Special characters in the sense of HTML (or XML, and thus XHTML) are &, >, <, " as well as '. Because these characters carry specific meanings in documents on their own you have to replace them by so called “entities” if you want to use them for text. Not doing so would not only cause user frustration by the inability to use these characters in text, but can also lead to security problems. (see *Cross-Site Scripting (XSS)*)

Sometimes however you will need to disable autoescaping in templates. This can be the case if you want to explicitly inject HTML into pages, for example if they come from a system that generates secure HTML like a markdown to HTML converter.

There are three ways to accomplish that:

- In the Python code, wrap the HTML string in a Markup object before passing it to the template. This is in general the recommended way.
- Inside the template, use the `|safe` filter to explicitly mark a string as safe HTML (`{{ myvariable|safe }}`)
- Temporarily disable the autoescape system altogether.

To disable the autoescape system in templates, you can use the `{% autoescape %}` block:

```
{% autoescape false %}
  <p>autoescaping is disabled here
  <p>{{ will_not_be_escaped }}
{% endautoescape %}
```

Whenever you do this, please be very cautious about the variables you are using in this block.

1.4.4 Registering Filters

If you want to register your own filters in Jinja2 you have two ways to do that. You can either put them by hand into the `jinja_env` of the application or use the `template_filter()` decorator.

The two following examples work the same and both reverse an object:

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

def reverse_filter(s):
    return s[::-1]
app.jinja_env.filters['reverse'] = reverse_filter
```

In case of the decorator the argument is optional if you want to use the function name as name of the filter. Once registered, you can use the filter in your templates in the same way as Jinja2’s builtin filters, for example if you have a Python list in context called `mylist`:

```
{% for x in mylist | reverse %}
{% endfor %}
```

1.4.5 Context Processors

To inject new variables automatically into the context of a template, context processors exist in Flask. Context processors run before the template is rendered and have the ability to inject new values into the template context. A context processor is a function that returns a dictionary. The keys and values of this dictionary are then merged with the template context, for all templates in the app:

```
@app.context_processor
def inject_user():
    return dict(user=g.user)
```

The context processor above makes a variable called *user* available in the template with the value of *g.user*. This example is not very interesting because *g* is available in templates anyways, but it gives an idea how this works.

Variables are not limited to values; a context processor can also make functions available to templates (since Python allows passing around functions):

```
@app.context_processor
def utility_processor():
    def format_price(amount, currency="€"):
        return f"{amount:.2f}{currency}"
    return dict(format_price=format_price)
```

The context processor above makes the *format_price* function available to all templates:

```
{{ format_price(0.33) }}
```

You could also build *format_price* as a template filter (see [Registering Filters](#)), but this demonstrates how to pass functions in a context processor.

1.4.6 Streaming

It can be useful to not render the whole template as one complete string, instead render it as a stream, yielding smaller incremental strings. This can be used for streaming HTML in chunks to speed up initial page load, or to save memory when rendering a very large template.

The Jinja2 template engine supports rendering a template piece by piece, returning an iterator of strings. Flask provides the *stream_template()* and *stream_template_string()* functions to make this easier to use.

```
from flask import stream_template

@app.get("/timeline")
def timeline():
    return stream_template("timeline.html")
```

These functions automatically apply the *stream_with_context()* wrapper if a request is active, so that it remains available in the template.

1.5 Testing Flask Applications

Flask provides utilities for testing an application. This documentation goes over techniques for working with different parts of the application in tests.

We will use the `pytest` framework to set up and run our tests.

```
$ pip install pytest
```

The *tutorial* goes over how to write tests for 100% coverage of the sample Flaskr blog application. See *the tutorial on tests* for a detailed explanation of specific tests for an application.

1.5.1 Identifying Tests

Tests are typically located in the `tests` folder. Tests are functions that start with `test_`, in Python modules that start with `test_`. Tests can also be further grouped in classes that start with `Test`.

It can be difficult to know what to test. Generally, try to test the code that you write, not the code of libraries that you use, since they are already tested. Try to extract complex behaviors as separate functions to test individually.

1.5.2 Fixtures

Pytest *fixtures* allow writing pieces of code that are reusable across tests. A simple fixture returns a value, but a fixture can also do setup, yield a value, then do teardown. Fixtures for the application, test client, and CLI runner are shown below, they can be placed in `tests/conftest.py`.

If you're using an *application factory*, define an app fixture to create and configure an app instance. You can add code before and after the `yield` to set up and tear down other resources, such as creating and clearing a database.

If you're not using a factory, you already have an app object you can import and configure directly. You can still use an app fixture to set up and tear down resources.

```
import pytest
from my_project import create_app

@pytest.fixture()
def app():
    app = create_app()
    app.config.update({
        "TESTING": True,
    })

    # other setup can go here

    yield app

    # clean up / reset resources here

@pytest.fixture()
def client(app):
    return app.test_client()
```

(continues on next page)

```
@pytest.fixture()
def runner(app):
    return app.test_cli_runner()
```

1.5.3 Sending Requests with the Test Client

The test client makes requests to the application without running a live server. Flask's client extends Werkzeug's client, see those docs for additional information.

The client has methods that match the common HTTP request methods, such as `client.get()` and `client.post()`. They take many arguments for building the request; you can find the full documentation in `EnvironBuilder`. Typically you'll use `path`, `query_string`, `headers`, and `data` or `json`.

To make a request, call the method the request should use with the path to the route to test. A `TestResponse` is returned to examine the response data. It has all the usual properties of a response object. You'll usually look at `response.data`, which is the bytes returned by the view. If you want to use text, Werkzeug 2.1 provides `response.text`, or use `response.get_data(as_text=True)`.

```
def test_request_example(client):
    response = client.get("/posts")
    assert b"<h2>Hello, World!</h2>" in response.data
```

Pass a dict `query_string={"key": "value", ...}` to set arguments in the query string (after the ? in the URL). Pass a dict `headers={}` to set request headers.

To send a request body in a POST or PUT request, pass a value to `data`. If raw bytes are passed, that exact body is used. Usually, you'll pass a dict to set form data.

Form Data

To send form data, pass a dict to `data`. The Content-Type header will be set to `multipart/form-data` or `application/x-www-form-urlencoded` automatically.

If a value is a file object opened for reading bytes ("rb" mode), it will be treated as an uploaded file. To change the detected filename and content type, pass a (file, filename, content_type) tuple. File objects will be closed after making the request, so they do not need to use the usual `with open() as f:` pattern.

It can be useful to store files in a `tests/resources` folder, then use `pathlib.Path` to get files relative to the current test file.

```
from pathlib import Path

# get the resources folder in the tests folder
resources = Path(__file__).parent / "resources"

def test_edit_user(client):
    response = client.post("/user/2/edit", data={
        "name": "Flask",
        "theme": "dark",
        "picture": (resources / "picture.png").open("rb"),
    })
    assert response.status_code == 200
```

JSON Data

To send JSON data, pass an object to `json`. The `Content-Type` header will be set to `application/json` automatically.

Similarly, if the response contains JSON data, the `response.json` attribute will contain the deserialized object.

```
def test_json_data(client):
    response = client.post("/graphql", json={
        "query": """
            query User($id: String!) {
              user(id: $id) {
                name
                theme
                picture_url
              }
            }
            """,
        "variables":{"id": 2},
    })
    assert response.json["data"]["user"]["name"] == "Flask"
```

1.5.4 Following Redirects

By default, the client does not make additional requests if the response is a redirect. By passing `follow_redirects=True` to a request method, the client will continue to make requests until a non-redirect response is returned.

`TestResponse.history` is a tuple of the responses that led up to the final response. Each response has a `request` attribute which records the request that produced that response.

```
def test_logout_redirect(client):
    response = client.get("/logout")
    # Check that there was one redirect response.
    assert len(response.history) == 1
    # Check that the second request was to the index page.
    assert response.request.path == "/index"
```

1.5.5 Accessing and Modifying the Session

To access Flask's context variables, mainly `session`, use the client in a `with` statement. The app and request context will remain active *after* making a request, until the `with` block ends.

```
from flask import session

def test_access_session(client):
    with client:
        client.post("/auth/login", data={"username": "flask"})
        # session is still accessible
        assert session["user_id"] == 1

    # session is no longer accessible
```

If you want to access or set a value in the session *before* making a request, use the client's `session_transaction()` method in a `with` statement. It returns a session object, and will save the session once the block ends.

```
from flask import session

def test_modify_session(client):
    with client.session_transaction() as session:
        # set a user id without going through the login route
        session["user_id"] = 1

    # session is saved now

    response = client.get("/users/me")
    assert response.json["username"] == "flask"
```

1.5.6 Running Commands with the CLI Runner

Flask provides `test_cli_runner()` to create a `FlaskCliRunner`, which runs CLI commands in isolation and captures the output in a `Result` object. Flask's runner extends Click's runner, see those docs for additional information.

Use the runner's `invoke()` method to call commands in the same way they would be called with the `flask` command from the command line.

```
import click

@app.cli.command("hello")
@click.option("--name", default="World")
def hello_command(name):
    click.echo(f"Hello, {name}!")

def test_hello_command(runner):
    result = runner.invoke(args="hello")
    assert "World" in result.output

    result = runner.invoke(args=["hello", "--name", "Flask"])
    assert "Flask" in result.output
```

1.5.7 Tests that depend on an Active Context

You may have functions that are called from views or commands, that expect an active *application context* or *request context* because they access `request`, `session`, or `current_app`. Rather than testing them by making a request or invoking the command, you can create and activate a context directly.

Use `with app.app_context()` to push an application context. For example, database extensions usually require an active app context to make queries.

```
def test_db_post_model(app):
    with app.app_context():
        post = db.session.query(Post).get(1)
```

Use `with app.test_request_context()` to push a request context. It takes the same arguments as the test client's request methods.

```
def test_validate_user_edit(app):
    with app.test_request_context(
        "/user/2/edit", method="POST", data={"name": ""}
    ):
        # call a function that accesses `request`
        messages = validate_edit_user()

    assert messages["name"][0] == "Name cannot be empty."
```

Creating a test request context doesn't run any of the Flask dispatching code, so `before_request` functions are not called. If you need to call these, usually it's better to make a full request instead. However, it's possible to call them manually.

```
def test_auth_token(app):
    with app.test_request_context("/user/2/edit", headers={"X-Auth-Token": "1"}):
        app.preprocess_request()
        assert g.user.name == "Flask"
```

1.6 Handling Application Errors

Applications fail, servers fail. Sooner or later you will see an exception in production. Even if your code is 100% correct, you will still see exceptions from time to time. Why? Because everything else involved will fail. Here are some situations where perfectly fine code can lead to server errors:

- the client terminated the request early and the application was still reading from the incoming data
- the database server was overloaded and could not handle the query
- a filesystem is full
- a harddrive crashed
- a backend server overloaded
- a programming error in a library you are using
- network connection of the server to another system failed

And that's just a small sample of issues you could be facing. So how do we deal with that sort of problem? By default if your application runs in production mode, and an exception is raised Flask will display a very simple page for you and log the exception to the *logger*.

But there is more you can do, and we will cover some better setups to deal with errors including custom exceptions and 3rd party tools.

1.6.1 Error Logging Tools

Sending error mails, even if just for critical ones, can become overwhelming if enough users are hitting the error and log files are typically never looked at. This is why we recommend using [Sentry](#) for dealing with application errors. It's available as a source-available project on [GitHub](#) and is also available as a [hosted version](#) which you can try for free. Sentry aggregates duplicate errors, captures the full stack trace and local variables for debugging, and sends you mails based on new errors or frequency thresholds.

To use Sentry you need to install the `sentry-sdk` client with extra `flask` dependencies.

```
$ pip install sentry-sdk[flask]
```

And then add this to your Flask app:

```
import sentry_sdk
from sentry_sdk.integrations.flask import FlaskIntegration

sentry_sdk.init('YOUR_DSN_HERE', integrations=[FlaskIntegration()])
```

The `YOUR_DSN_HERE` value needs to be replaced with the DSN value you get from your Sentry installation.

After installation, failures leading to an Internal Server Error are automatically reported to Sentry and from there you can receive error notifications.

See also:

- Sentry also supports catching errors from a worker queue (RQ, Celery, etc.) in a similar fashion. See the [Python SDK docs](#) for more information.
- [Flask-specific documentation](#)

1.6.2 Error Handlers

When an error occurs in Flask, an appropriate [HTTP status code](#) will be returned. 400-499 indicate errors with the client's request data, or about the data requested. 500-599 indicate errors with the server or application itself.

You might want to show custom error pages to the user when an error occurs. This can be done by registering error handlers.

An error handler is a function that returns a response when a type of error is raised, similar to how a view is a function that returns a response when a request URL is matched. It is passed the instance of the error being handled, which is most likely a [HTTPException](#).

The status code of the response will not be set to the handler's code. Make sure to provide the appropriate HTTP status code when returning a response from a handler.

Registering

Register handlers by decorating a function with [errorhandler\(\)](#). Or use [register_error_handler\(\)](#) to register the function later. Remember to set the error code when returning the response.

```
@app.errorhandler(werkzeug.exceptions.BadRequest)
def handle_bad_request(e):
    return 'bad request!', 400

# or, without the decorator
app.register_error_handler(400, handle_bad_request)
```

[werkzeug.exceptions.HTTPException](#) subclasses like [BadRequest](#) and their HTTP codes are interchangeable when registering handlers. (`BadRequest.code == 400`)

Non-standard HTTP codes cannot be registered by code because they are not known by Werkzeug. Instead, define a subclass of [HTTPException](#) with the appropriate code and register and raise that exception class.

```

class InsufficientStorage(werkzeug.exceptions.HTTPException):
    code = 507
    description = 'Not enough storage space.'

app.register_error_handler(InsufficientStorage, handle_507)

raise InsufficientStorage()

```

Handlers can be registered for any exception class, not just `HTTPException` subclasses or HTTP status codes. Handlers can be registered for a specific class, or for all subclasses of a parent class.

Handling

When building a Flask application you *will* run into exceptions. If some part of your code breaks while handling a request (and you have no error handlers registered), a “500 Internal Server Error” (`InternalServerError`) will be returned by default. Similarly, “404 Not Found” (`NotFound`) error will occur if a request is sent to an unregistered route. If a route receives an unallowed request method, a “405 Method Not Allowed” (`MethodNotAllowed`) will be raised. These are all subclasses of `HTTPException` and are provided by default in Flask.

Flask gives you the ability to raise any HTTP exception registered by Werkzeug. However, the default HTTP exceptions return simple exception pages. You might want to show custom error pages to the user when an error occurs. This can be done by registering error handlers.

When Flask catches an exception while handling a request, it is first looked up by code. If no handler is registered for the code, Flask looks up the error by its class hierarchy; the most specific handler is chosen. If no handler is registered, `HTTPException` subclasses show a generic message about their code, while other exceptions are converted to a generic “500 Internal Server Error”.

For example, if an instance of `ConnectionRefusedError` is raised, and a handler is registered for `ConnectionError` and `ConnectionRefusedError`, the more specific `ConnectionRefusedError` handler is called with the exception instance to generate the response.

Handlers registered on the blueprint take precedence over those registered globally on the application, assuming a blueprint is handling the request that raises the exception. However, the blueprint cannot handle 404 routing errors because the 404 occurs at the routing level before the blueprint can be determined.

Generic Exception Handlers

It is possible to register error handlers for very generic base classes such as `HTTPException` or even `Exception`. However, be aware that these will catch more than you might expect.

For example, an error handler for `HTTPException` might be useful for turning the default HTML errors pages into JSON. However, this handler will trigger for things you don’t cause directly, such as 404 and 405 errors during routing. Be sure to craft your handler carefully so you don’t lose information about the HTTP error.

```

from flask import json
from werkzeug.exceptions import HTTPException

@app.errorhandler(HTTPException)
def handle_exception(e):
    """Return JSON instead of HTML for HTTP errors."""
    # start with the correct headers and status code from the error
    response = e.get_response()
    # replace the body with JSON

```

(continues on next page)

(continued from previous page)

```
response.data = json.dumps({
    "code": e.code,
    "name": e.name,
    "description": e.description,
})
response.content_type = "application/json"
return response
```

An error handler for `Exception` might seem useful for changing how all errors, even unhandled ones, are presented to the user. However, this is similar to doing `except Exception:` in Python, it will capture *all* otherwise unhandled errors, including all HTTP status codes.

In most cases it will be safer to register handlers for more specific exceptions. Since `HTTPException` instances are valid WSGI responses, you could also pass them through directly.

```
from werkzeug.exceptions import HTTPException

@app.errorhandler(Exception)
def handle_exception(e):
    # pass through HTTP errors
    if isinstance(e, HTTPException):
        return e

    # now you're handling non-HTTP exceptions only
    return render_template("500_generic.html", e=e), 500
```

Error handlers still respect the exception class hierarchy. If you register handlers for both `HTTPException` and `Exception`, the `Exception` handler will not handle `HTTPException` subclasses because it the `HTTPException` handler is more specific.

Unhandled Exceptions

When there is no error handler registered for an exception, a 500 Internal Server Error will be returned instead. See `flask.Flask.handle_exception()` for information about this behavior.

If there is an error handler registered for `InternalServerError`, this will be invoked. As of Flask 1.1.0, this error handler will always be passed an instance of `InternalServerError`, not the original unhandled error.

The original error is available as `e.original_exception`.

An error handler for “500 Internal Server Error” will be passed uncaught exceptions in addition to explicit 500 errors. In debug mode, a handler for “500 Internal Server Error” will not be used. Instead, the interactive debugger will be shown.

1.6.3 Custom Error Pages

Sometimes when building a Flask application, you might want to raise a `HTTPException` to signal to the user that something is wrong with the request. Fortunately, Flask comes with a handy `abort()` function that aborts a request with a HTTP error from Werkzeug as desired. It will also provide a plain black and white error page for you with a basic description, but nothing fancy.

Depending on the error code it is less or more likely for the user to actually see such an error.

Consider the code below, we might have a user profile route, and if the user fails to pass a username we can raise a “400 Bad Request”. If the user passes a username and we can’t find it, we raise a “404 Not Found”.

```
from flask import abort, render_template, request

# a username needs to be supplied in the query args
# a successful request would be like /profile?username=jack
@app.route("/profile")
def user_profile():
    username = request.args.get("username")
    # if a username isn't supplied in the request, return a 400 bad request
    if username is None:
        abort(400)

    user = get_user(username=username)
    # if a user can't be found by their username, return 404 not found
    if user is None:
        abort(404)

    return render_template("profile.html", user=user)
```

Here is another example implementation for a “404 Page Not Found” exception:

```
from flask import render_template

@app.errorhandler(404)
def page_not_found(e):
    # note that we set the 404 status explicitly
    return render_template('404.html'), 404
```

When using *Application Factories*:

```
from flask import Flask, render_template

def page_not_found(e):
    return render_template('404.html'), 404

def create_app(config_filename):
    app = Flask(__name__)
    app.register_error_handler(404, page_not_found)
    return app
```

An example template might be this:

```
{% extends "layout.html" %}
{% block title %}Page Not Found{% endblock %}
```

(continues on next page)

(continued from previous page)

```
{% block body %}
<h1>Page Not Found</h1>
<p>What you were looking for is just not there.
<p><a href="{{ url_for('index') }}">go somewhere nice</a>
{% endblock %}
```

Further Examples

The above examples wouldn't actually be an improvement on the default exception pages. We can create a custom 500.html template like this:

```
{% extends "layout.html" %}
{% block title %}Internal Server Error{% endblock %}
{% block body %}
<h1>Internal Server Error</h1>
<p>Oops... we seem to have made a mistake, sorry!</p>
<p><a href="{{ url_for('index') }}">Go somewhere nice instead</a>
{% endblock %}
```

It can be implemented by rendering the template on "500 Internal Server Error":

```
from flask import render_template

@app.errorhandler(500)
def internal_server_error(e):
    # note that we set the 500 status explicitly
    return render_template('500.html'), 500
```

When using *Application Factories*:

```
from flask import Flask, render_template

def internal_server_error(e):
    return render_template('500.html'), 500

def create_app():
    app = Flask(__name__)
    app.register_error_handler(500, internal_server_error)
    return app
```

When using *Modular Applications with Blueprints*:

```
from flask import Blueprint

blog = Blueprint('blog', __name__)

# as a decorator
@blog.errorhandler(500)
def internal_server_error(e):
    return render_template('500.html'), 500
```

(continues on next page)

(continued from previous page)

```
# or with register_error_handler
blog.register_error_handler(500, internal_server_error)
```

1.6.4 Blueprint Error Handlers

In *Modular Applications with Blueprints*, most error handlers will work as expected. However, there is a caveat concerning handlers for 404 and 405 exceptions. These error handlers are only invoked from an appropriate `raise` statement or a call to `abort` in another of the blueprint's view functions; they are not invoked by, e.g., an invalid URL access.

This is because the blueprint does not “own” a certain URL space, so the application instance has no way of knowing which blueprint error handler it should run if given an invalid URL. If you would like to execute different handling strategies for these errors based on URL prefixes, they may be defined at the application level using the request proxy object.

```
from flask import jsonify, render_template

# at the application level
# not the blueprint level
@app.errorhandler(404)
def page_not_found(e):
    # if a request is in our blog URL space
    if request.path.startswith('/blog/'):
        # we return a custom blog 404 page
        return render_template("blog/404.html"), 404
    else:
        # otherwise we return our generic site-wide 404 page
        return render_template("404.html"), 404

@app.errorhandler(405)
def method_not_allowed(e):
    # if a request has the wrong method to our API
    if request.path.startswith('/api/'):
        # we return a json saying so
        return jsonify(message="Method Not Allowed"), 405
    else:
        # otherwise we return a generic site-wide 405 page
        return render_template("405.html"), 405
```

1.6.5 Returning API Errors as JSON

When building APIs in Flask, some developers realise that the built-in exceptions are not expressive enough for APIs and that the content type of `text/html` they are emitting is not very useful for API consumers.

Using the same techniques as above and `jsonify()` we can return JSON responses to API errors. `abort()` is called with a description parameter. The error handler will use that as the JSON error message, and set the status code to 404.

```
from flask import abort, jsonify

@app.errorhandler(404)
def resource_not_found(e):
```

(continues on next page)

```
    return jsonify(error=str(e)), 404

@app.route("/cheese")
def get_one_cheese():
    resource = get_resource()

    if resource is None:
        abort(404, description="Resource not found")

    return jsonify(resource)
```

We can also create custom exception classes. For instance, we can introduce a new custom exception for an API that can take a proper human readable message, a status code for the error and some optional payload to give more context for the error.

This is a simple example:

```
from flask import jsonify, request

class InvalidAPIUsage(Exception):
    status_code = 400

    def __init__(self, message, status_code=None, payload=None):
        super().__init__()
        self.message = message
        if status_code is not None:
            self.status_code = status_code
        self.payload = payload

    def to_dict(self):
        rv = dict(self.payload or ())
        rv['message'] = self.message
        return rv

@app.errorhandler(InvalidAPIUsage)
def invalid_api_usage(e):
    return jsonify(e.to_dict()), e.status_code

# an API app route for getting user information
# a correct request might be /api/user?user_id=420
@app.route("/api/user")
def user_api(user_id):
    user_id = request.args.get("user_id")
    if not user_id:
        raise InvalidAPIUsage("No user id provided!")

    user = get_user(user_id=user_id)
    if not user:
        raise InvalidAPIUsage("No such user!", status_code=404)

    return jsonify(user.to_dict())
```

A view can now raise that exception with an error message. Additionally some extra payload can be provided as a

dictionary through the *payload* parameter.

1.6.6 Logging

See *Logging* for information about how to log exceptions, such as by emailing them to admins.

1.6.7 Debugging

See *Debugging Application Errors* for information about how to debug errors in development and production.

1.7 Debugging Application Errors

1.7.1 In Production

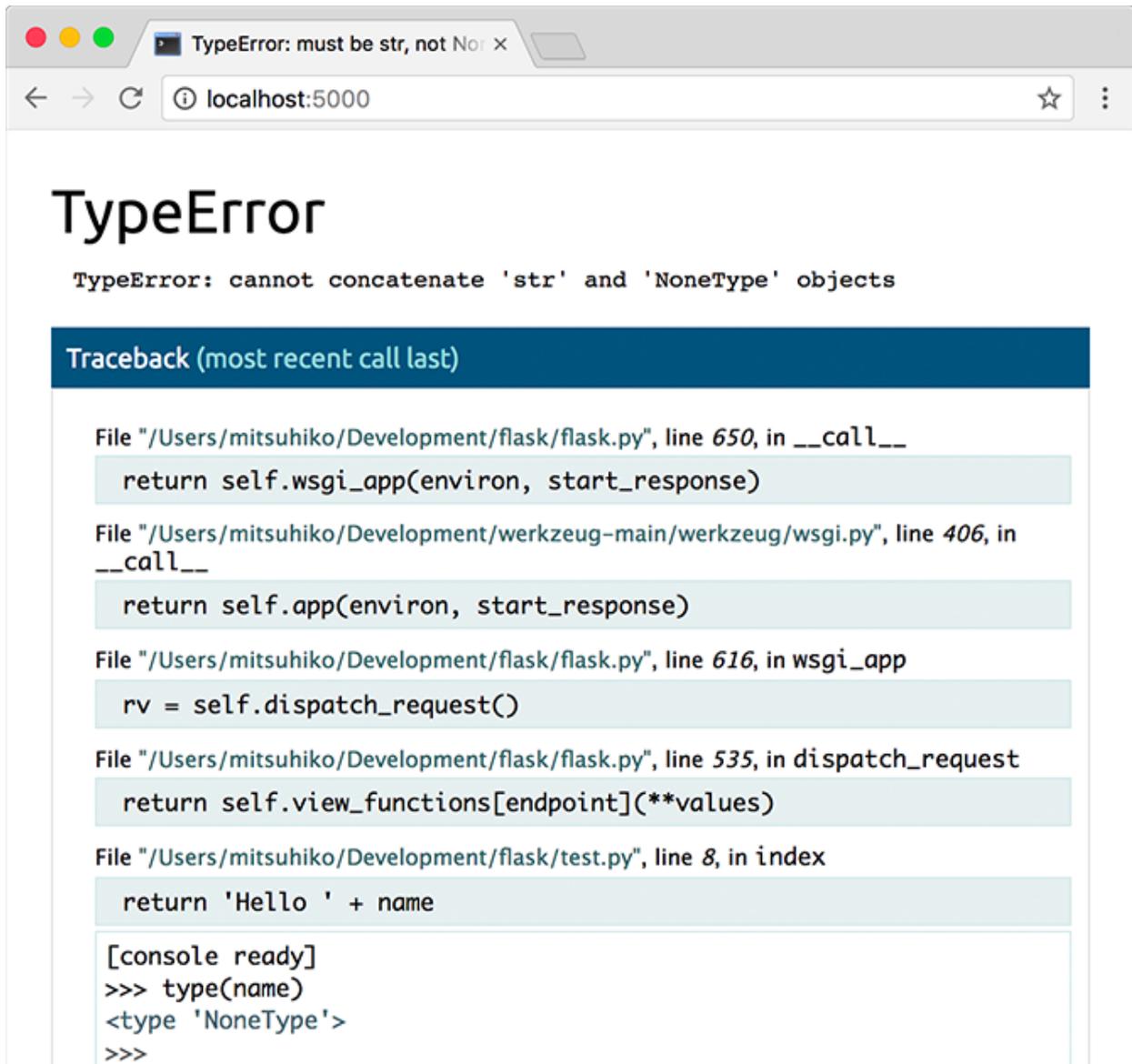
Do not run the development server, or enable the built-in debugger, in a production environment. The debugger allows executing arbitrary Python code from the browser. It's protected by a pin, but that should not be relied on for security.

Use an error logging tool, such as Sentry, as described in *Error Logging Tools*, or enable logging and notifications as described in *Logging*.

If you have access to the server, you could add some code to start an external debugger if `request.remote_addr` matches your IP. Some IDE debuggers also have a remote mode so breakpoints on the server can be interacted with locally. Only enable a debugger temporarily.

1.7.2 The Built-In Debugger

The built-in Werkzeug development server provides a debugger which shows an interactive traceback in the browser when an unhandled error occurs during a request. This debugger should only be used during development.



Warning: The debugger allows executing arbitrary Python code from the browser. It is protected by a pin, but still represents a major security risk. Do not run the development server or debugger in a production environment.

The debugger is enabled by default when the development server is run in debug mode.

```
$ flask --app hello run --debug
```

When running from Python code, passing `debug=True` enables debug mode, which is mostly equivalent.

```
app.run(debug=True)
```

Development Server and *Command Line Interface* have more information about running the debugger and debug mode. More information about the debugger can be found in the [Werkzeug documentation](#).

1.7.3 External Debuggers

External debuggers, such as those provided by IDEs, can offer a more powerful debugging experience than the built-in debugger. They can also be used to step through code during a request before an error is raised, or if no error is raised. Some even have a remote mode so you can debug code running on another machine.

When using an external debugger, the app should still be in debug mode, otherwise Flask turns unhandled errors into generic 500 error pages. However, the built-in debugger and reloader should be disabled so they don't interfere with the external debugger.

```
$ flask --app hello run --debug --no-debugger --no-reload
```

When running from Python:

```
app.run(debug=True, use_debugger=False, use_reloader=False)
```

Disabling these isn't required, an external debugger will continue to work with the following caveats.

- If the built-in debugger is not disabled, it will catch unhandled exceptions before the external debugger can.
- If the reloader is not disabled, it could cause an unexpected reload if code changes during a breakpoint.
- The development server will still catch unhandled exceptions if the built-in debugger is disabled, otherwise it would crash on any error. If you want that (and usually you don't) pass `passthrough_errors=True` to `app.run`.

```
app.run(
    debug=True, passthrough_errors=True,
    use_debugger=False, use_reloader=False
)
```

1.8 Logging

Flask uses standard Python [logging](#). Messages about your Flask application are logged with `app.logger`, which takes the same name as `app.name`. This logger can also be used to log your own messages.

```
@app.route('/login', methods=['POST'])
def login():
    user = get_user(request.form['username'])

    if user.check_password(request.form['password']):
        login_user(user)
        app.logger.info('%s logged in successfully', user.username)
        return redirect(url_for('index'))
    else:
        app.logger.info('%s failed to log in', user.username)
        abort(401)
```

If you don't configure logging, Python's default log level is usually 'warning'. Nothing below the configured level will be visible.

1.8.1 Basic Configuration

When you want to configure logging for your project, you should do it as soon as possible when the program starts. If `app.logger` is accessed before logging is configured, it will add a default handler. If possible, configure logging before creating the application object.

This example uses `dictConfig()` to create a logging configuration similar to Flask's default, except for all logs:

```
from logging.config import dictConfig

dictConfig({
    'version': 1,
    'formatters': {'default': {
        'format': '[%(asctime)s] %(levelname)s in %(module)s: %(message)s',
    }},
    'handlers': {'wsgi': {
        'class': 'logging.StreamHandler',
        'stream': 'ext://flask.logging.wsgi_errors_stream',
        'formatter': 'default'
    }},
    'root': {
        'level': 'INFO',
        'handlers': ['wsgi']
    }
})

app = Flask(__name__)
```

Default Configuration

If you do not configure logging yourself, Flask will add a `StreamHandler` to `app.logger` automatically. During requests, it will write to the stream specified by the WSGI server in `environ['wsgi.errors']` (which is usually `sys.stderr`). Outside a request, it will log to `sys.stderr`.

Removing the Default Handler

If you configured logging after accessing `app.logger`, and need to remove the default handler, you can import and remove it:

```
from flask.logging import default_handler

app.logger.removeHandler(default_handler)
```

1.8.2 Email Errors to Admins

When running the application on a remote server for production, you probably won't be looking at the log messages very often. The WSGI server will probably send log messages to a file, and you'll only check that file if a user tells you something went wrong.

To be proactive about discovering and fixing bugs, you can configure a `logging.handlers.SMTPHandler` to send an email when errors and higher are logged.

```
import logging
from logging.handlers import SMTPHandler

mail_handler = SMTPHandler(
    mailhost='127.0.0.1',
    fromaddr='server-error@example.com',
    toaddrs=['admin@example.com'],
    subject='Application Error'
)
mail_handler.setLevel(logging.ERROR)
mail_handler.setFormatter(logging.Formatter(
    '[%(asctime)s] %(levelname)s in %(module)s: %(message)s'
))

if not app.debug:
    app.logger.addHandler(mail_handler)
```

This requires that you have an SMTP server set up on the same server. See the Python docs for more information about configuring the handler.

1.8.3 Injecting Request Information

Seeing more information about the request, such as the IP address, may help debugging some errors. You can subclass `logging.Formatter` to inject your own fields that can be used in messages. You can change the formatter for Flask's default handler, the mail handler defined above, or any other handler.

```
from flask import has_request_context, request
from flask.logging import default_handler

class RequestFormatter(logging.Formatter):
    def format(self, record):
        if has_request_context():
            record.url = request.url
            record.remote_addr = request.remote_addr
        else:
            record.url = None
            record.remote_addr = None

        return super().format(record)

formatter = RequestFormatter(
    '[%(asctime)s] %(remote_addr)s requested %(url)s\n'
    '%(levelname)s in %(module)s: %(message)s'
)
```

(continues on next page)

(continued from previous page)

```
default_handler.setFormatter(formatter)
mail_handler.setFormatter(formatter)
```

1.8.4 Other Libraries

Other libraries may use logging extensively, and you want to see relevant messages from those logs too. The simplest way to do this is to add handlers to the root logger instead of only the app logger.

```
from flask.logging import default_handler

root = logging.getLogger()
root.addHandler(default_handler)
root.addHandler(mail_handler)
```

Depending on your project, it may be more useful to configure each logger you care about separately, instead of configuring only the root logger.

```
for logger in (
    app.logger,
    logging.getLogger('sqlalchemy'),
    logging.getLogger('other_package'),
):
    logger.addHandler(default_handler)
    logger.addHandler(mail_handler)
```

Werkzeug

Werkzeug logs basic request/response information to the 'werkzeug' logger. If the root logger has no handlers configured, Werkzeug adds a `StreamHandler` to its logger.

Flask Extensions

Depending on the situation, an extension may choose to log to `app.logger` or its own named logger. Consult each extension's documentation for details.

1.9 Configuration Handling

Applications need some kind of configuration. There are different settings you might want to change depending on the application environment like toggling the debug mode, setting the secret key, and other such environment-specific things.

The way Flask is designed usually requires the configuration to be available when the application starts up. You can hard code the configuration in the code, which for many small applications is not actually that bad, but there are better ways.

Independent of how you load your config, there is a config object available which holds the loaded configuration values: The `config` attribute of the `Flask` object. This is the place where Flask itself puts certain configuration values and also where extensions can put their configuration values. But this is also where you can have your own configuration.

1.9.1 Configuration Basics

The `config` is actually a subclass of a dictionary and can be modified just like any dictionary:

```
app = Flask(__name__)
app.config['TESTING'] = True
```

Certain configuration values are also forwarded to the `Flask` object so you can read and write them from there:

```
app.testing = True
```

To update multiple keys at once you can use the `dict.update()` method:

```
app.config.update(
    TESTING=True,
    SECRET_KEY='192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
)
```

1.9.2 Debug Mode

The `DEBUG` config value is special because it may behave inconsistently if changed after the app has begun setting up. In order to set debug mode reliably, use the `--debug` option on the `flask` or `flask run` command. `flask run` will use the interactive debugger and reloader by default in debug mode.

```
$ flask --app hello run --debug
```

Using the option is recommended. While it is possible to set `DEBUG` in your config or code, this is strongly discouraged. It can't be read early by the `flask run` command, and some systems or extensions may have already configured themselves based on a previous value.

1.9.3 Builtin Configuration Values

The following configuration values are used internally by Flask:

DEBUG

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

Do not enable debug mode when deploying in production.

Default: False

TESTING

Enable testing mode. Exceptions are propagated rather than handled by the the app's error handlers. Extensions may also change their behavior to facilitate easier testing. You should enable this in your own tests.

Default: False

PROPAGATE_EXCEPTIONS

Exceptions are re-raised rather than being handled by the app's error handlers. If not set, this is implicitly true if `TESTING` or `DEBUG` is enabled.

Default: None

TRAP_HTTP_EXCEPTIONS

If there is no handler for an `HTTPException`-type exception, re-raise it to be handled by the interactive debugger instead of returning it as a simple error response.

Default: `False`

TRAP_BAD_REQUEST_ERRORS

Trying to access a key that doesn't exist from request dicts like `args` and `form` will return a 400 Bad Request error page. Enable this to treat the error as an unhandled exception instead so that you get the interactive debugger. This is a more specific version of `TRAP_HTTP_EXCEPTIONS`. If unset, it is enabled in debug mode.

Default: `None`

SECRET_KEY

A secret key that will be used for securely signing the session cookie and can be used for any other security related needs by extensions or your application. It should be a long random bytes or `str`. For example, copy the output of this to your config:

```
$ python -c 'import secrets; print(secrets.token_hex())'  
'192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

Do not reveal the secret key when posting questions or committing code.

Default: `None`

SESSION_COOKIE_NAME

The name of the session cookie. Can be changed in case you already have a cookie with the same name.

Default: `'session'`

SESSION_COOKIE_DOMAIN

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.

Not setting this value is more restricted and secure than setting it.

Default: `None`

Changed in version 2.3: Not set by default, does not fall back to `SERVER_NAME`.

SESSION_COOKIE_PATH

The path that the session cookie will be valid for. If not set, the cookie will be valid underneath `APPLICATION_ROOT` or `/` if that is not set.

Default: `None`

SESSION_COOKIE_HTTPONLY

Browsers will not allow JavaScript access to cookies marked as “HTTP only” for security.

Default: `True`

SESSION_COOKIE_SECURE

Browsers will only send cookies with requests over HTTPS if the cookie is marked “secure”. The application must be served over HTTPS for this to make sense.

Default: `False`

SESSION_COOKIE_SAMESITE

Restrict how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. See *Set-Cookie options*.

Default: `None`

New in version 1.0.

PERMANENT_SESSION_LIFETIME

If `session.permanent` is true, the cookie's expiration will be set this number of seconds in the future. Can either be a `datetime.timedelta` or an `int`.

Flask's default cookie implementation validates that the cryptographic signature is not older than this value.

Default: `timedelta(days=31)` (2678400 seconds)

SESSION_REFRESH_EACH_REQUEST

Control whether the cookie is sent with every response when `session.permanent` is true. Sending the cookie every time (the default) can more reliably keep the session from expiring, but uses more bandwidth. Non-permanent sessions are not affected.

Default: True

USE_X_SENDFILE

When serving files, set the X-Sendfile header instead of serving the data with Flask. Some web servers, such as Apache, recognize this and serve the data more efficiently. This only makes sense when using such a server.

Default: False

SEND_FILE_MAX_AGE_DEFAULT

When serving files, set the cache control max age to this number of seconds. Can be a `datetime.timedelta` or an `int`. Override this value on a per-file basis using `get_send_file_max_age()` on the application or blueprint.

If None, `send_file` tells the browser to use conditional requests will be used instead of a timed cache, which is usually preferable.

Default: None

SERVER_NAME

Inform the application what host and port it is bound to. Required for subdomain route matching support.

If set, `url_for` can generate external URLs with only an application context instead of a request context.

Default: None

Changed in version 2.3: Does not affect `SESSION_COOKIE_DOMAIN`.

APPLICATION_ROOT

Inform the application what path it is mounted under by the application / web server. This is used for generating URLs outside the context of a request (inside a request, the dispatcher is responsible for setting `SCRIPT_NAME` instead; see *Application Dispatching* for examples of dispatch configuration).

Will be used for the session cookie path if `SESSION_COOKIE_PATH` is not set.

Default: `'/'`

PREFERRED_URL_SCHEME

Use this scheme for generating external URLs when not in a request context.

Default: `'http'`

MAX_CONTENT_LENGTH

Don't read more than this many bytes from the incoming request data. If not set and the request does not specify a `CONTENT_LENGTH`, no data will be read for security.

Default: None

TEMPLATES_AUTO_RELOAD

Reload templates when they are changed. If not set, it will be enabled in debug mode.

Default: None

EXPLAIN_TEMPLATE_LOADING

Log debugging information tracing how a template file was loaded. This can be useful to figure out why a template was not loaded or the wrong file appears to be loaded.

Default: False

MAX_COOKIE_SIZE

Warn if cookie headers are larger than this many bytes. Defaults to 4093. Larger cookies may be silently ignored by browsers. Set to 0 to disable the warning.

Changed in version 2.3: JSON_AS_ASCII, JSON_SORT_KEYS, JSONIFY_MIMETYPE, and JSONIFY_PRETTYPRINT_REGULAR were removed. The default app.json provider has equivalent attributes instead.

Changed in version 2.3: ENV was removed.

Changed in version 2.2: Removed PRESERVE_CONTEXT_ON_EXCEPTION.

Changed in version 1.0: LOGGER_NAME and LOGGER_HANDLER_POLICY were removed. See *Logging* for information about configuration.

Added ENV to reflect the FLASK_ENV environment variable.

Added `SESSION_COOKIE_SAMESITE` to control the session cookie's SameSite option.

Added `MAX_COOKIE_SIZE` to control a warning from Werkzeug.

New in version 0.11: SESSION_REFRESH_EACH_REQUEST, TEMPLATES_AUTO_RELOAD, LOGGER_HANDLER_POLICY, EXPLAIN_TEMPLATE_LOADING

New in version 0.10: JSON_AS_ASCII, JSON_SORT_KEYS, JSONIFY_PRETTYPRINT_REGULAR

New in version 0.9: PREFERRED_URL_SCHEME

New in version 0.8: TRAP_BAD_REQUEST_ERRORS, TRAP_HTTP_EXCEPTIONS, APPLICATION_ROOT, SESSION_COOKIE_DOMAIN, SESSION_COOKIE_PATH, SESSION_COOKIE_HTTPONLY, SESSION_COOKIE_SECURE

New in version 0.7: PROPAGATE_EXCEPTIONS, PRESERVE_CONTEXT_ON_EXCEPTION

New in version 0.6: MAX_CONTENT_LENGTH

New in version 0.5: SERVER_NAME

New in version 0.4: LOGGER_NAME

1.9.4 Configuring from Python Files

Configuration becomes more useful if you can store it in a separate file, ideally located outside the actual application package. You can deploy your application, then separately configure it for the specific deployment.

A common pattern is this:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

This first loads the configuration from the `yourapplication.default_settings` module and then overrides the values with the contents of the file the `YOURAPPLICATION_SETTINGS` environment variable points to. This environment variable can be set in the shell before starting the server:

Bash

```
$ export YOURAPPLICATION_SETTINGS=/path/to/settings.cfg
$ flask run
* Running on http://127.0.0.1:5000/
```

Fish

```
$ set -x YOURAPPLICATION_SETTINGS /path/to/settings.cfg
$ flask run
* Running on http://127.0.0.1:5000/
```

CMD

```
> set YOURAPPLICATION_SETTINGS=\path\to\settings.cfg
> flask run
* Running on http://127.0.0.1:5000/
```

Powershell

```
> $env:YOURAPPLICATION_SETTINGS = "\path\to\settings.cfg"
> flask run
* Running on http://127.0.0.1:5000/
```

The configuration files themselves are actual Python files. Only values in uppercase are actually stored in the config object later on. So make sure to use uppercase letters for your config keys.

Here is an example of a configuration file:

```
# Example configuration
SECRET_KEY = '192b9bdd22ab9ed4d12e236c78afcb9a393ec15f71bbf5dc987d54727823bcbf'
```

Make sure to load the configuration very early on, so that extensions have the ability to access the configuration when starting up. There are other methods on the config object as well to load from individual files. For a complete reference, read the [Config](#) object's documentation.

1.9.5 Configuring from Data Files

It is also possible to load configuration from a file in a format of your choice using `from_file()`. For example to load from a TOML file:

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

Or from a JSON file:

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

1.9.6 Configuring from Environment Variables

In addition to pointing to configuration files using environment variables, you may find it useful (or necessary) to control your configuration values directly from the environment. Flask can be instructed to load all environment variables starting with a specific prefix into the config using `from_prefixed_env()`.

Environment variables can be set in the shell before starting the server:

Bash

```
$ export FLASK_SECRET_KEY="5f352379324c22463451387a0aec5d2f"
$ export FLASK_MAIL_ENABLED=false
$ flask run
* Running on http://127.0.0.1:5000/
```

Fish

```
$ set -x FLASK_SECRET_KEY "5f352379324c22463451387a0aec5d2f"
$ set -x FLASK_MAIL_ENABLED false
$ flask run
* Running on http://127.0.0.1:5000/
```

CMD

```
> set FLASK_SECRET_KEY="5f352379324c22463451387a0aec5d2f"
> set FLASK_MAIL_ENABLED=false
> flask run
* Running on http://127.0.0.1:5000/
```

Powershell

```
> $env:FLASK_SECRET_KEY = "5f352379324c22463451387a0aec5d2f"
> $env:FLASK_MAIL_ENABLED = "false"
> flask run
* Running on http://127.0.0.1:5000/
```

The variables can then be loaded and accessed via the config with a key equal to the environment variable name without the prefix i.e.

```
app.config.from_prefixed_env()
app.config["SECRET_KEY"] # Is "5f352379324c22463451387a0aec5d2f"
```

The prefix is `FLASK_` by default. This is configurable via the `prefix` argument of `from_prefixed_env()`.

Values will be parsed to attempt to convert them to a more specific type than strings. By default `json.loads()` is used, so any valid JSON value is possible, including lists and dicts. This is configurable via the `loads` argument of `from_prefixed_env()`.

When adding a boolean value with the default JSON parsing, only “true” and “false”, lowercase, are valid values. Keep in mind that any non-empty string is considered True by Python.

It is possible to set keys in nested dictionaries by separating the keys with double underscore (`__`). Any intermediate keys that don’t exist on the parent dict will be initialized to an empty dict.

```
$ export FLASK_MYAPI__credentials__username=user123
```

```
app.config["MYAPI"]["credentials"]["username"] # Is "user123"
```

On Windows, environment variable keys are always uppercase, therefore the above example would end up as `MYAPI__CREDENTIALS__USERNAME`.

For even more config loading features, including merging and case-insensitive Windows support, try a dedicated library such as [Dynaconf](#), which includes integration with Flask.

1.9.7 Configuration Best Practices

The downside with the approach mentioned earlier is that it makes testing a little harder. There is no single 100% solution for this problem in general, but there are a couple of things you can keep in mind to improve that experience:

1. Create your application in a function and register blueprints on it. That way you can create multiple instances of your application with different configurations attached which makes unit testing a lot easier. You can use this to pass in configuration as needed.
2. Do not write code that needs the configuration at import time. If you limit yourself to request-only accesses to the configuration you can reconfigure the object later on as needed.
3. Make sure to load the configuration very early on, so that extensions can access the configuration when calling `init_app`.

1.9.8 Development / Production

Most applications need more than one configuration. There should be at least separate configurations for the production server and the one used during development. The easiest way to handle this is to use a default configuration that is always loaded and part of the version control, and a separate configuration that overrides the values as necessary as mentioned in the example above:

```
app = Flask(__name__)
app.config.from_object('yourapplication.default_settings')
app.config.from_envvar('YOURAPPLICATION_SETTINGS')
```

Then you just have to add a separate `config.py` file and export `YOURAPPLICATION_SETTINGS=/path/to/config.py` and you are done. However there are alternative ways as well. For example you could use imports or subclassing.

What is very popular in the Django world is to make the import explicit in the config file by adding `from yourapplication.default_settings import *` to the top of the file and then overriding the changes by hand. You could also inspect an environment variable like `YOURAPPLICATION_MODE` and set that to *production*, *development* etc and import different hard-coded files based on that.

An interesting pattern is also to use classes and inheritance for configuration:

```
class Config(object):
    TESTING = False

class ProductionConfig(Config):
    DATABASE_URI = 'mysql://user@localhost/foo'

class DevelopmentConfig(Config):
    DATABASE_URI = "sqlite:///tmp/foo.db"

class TestingConfig(Config):
```

(continues on next page)

(continued from previous page)

```
DATABASE_URI = 'sqlite:///memory:'  
TESTING = True
```

To enable such a config you just have to call into `from_object()`:

```
app.config.from_object('configmodule.ProductionConfig')
```

Note that `from_object()` does not instantiate the class object. If you need to instantiate the class, such as to access a property, then you must do so before calling `from_object()`:

```
from configmodule import ProductionConfig  
app.config.from_object(ProductionConfig())  
  
# Alternatively, import via string:  
from werkzeug.utils import import_string  
cfg = import_string('configmodule.ProductionConfig')()  
app.config.from_object(cfg)
```

Instantiating the configuration object allows you to use `@property` in your configuration classes:

```
class Config(object):  
    """Base config, uses staging database server."""  
    TESTING = False  
    DB_SERVER = '192.168.1.56'  
  
    @property  
    def DATABASE_URI(self): # Note: all caps  
        return f"mysql://user@{self.DB_SERVER}/foo"  
  
class ProductionConfig(Config):  
    """Uses production database server."""  
    DB_SERVER = '192.168.19.32'  
  
class DevelopmentConfig(Config):  
    DB_SERVER = 'localhost'  
  
class TestingConfig(Config):  
    DB_SERVER = 'localhost'  
    DATABASE_URI = 'sqlite:///memory:'
```

There are many different ways and it's up to you how you want to manage your configuration files. However here a list of good recommendations:

- Keep a default configuration in version control. Either populate the config with this default configuration or import it in your own configuration files before overriding values.
- Use an environment variable to switch between the configurations. This can be done from outside the Python interpreter and makes development and deployment much easier because you can quickly and easily switch between different configs without having to touch the code at all. If you are working often on different projects you can even create your own script for sourcing that activates a virtualenv and exports the development configuration for you.
- Use a tool like `fabric` to push code and configuration separately to the production server(s).

1.9.9 Instance Folders

New in version 0.8.

Flask 0.8 introduces instance folders. Flask for a long time made it possible to refer to paths relative to the application's folder directly (via `Flask.root_path`). This was also how many developers loaded configurations stored next to the application. Unfortunately however this only works well if applications are not packages in which case the root path refers to the contents of the package.

With Flask 0.8 a new attribute was introduced: `Flask.instance_path`. It refers to a new concept called the “instance folder”. The instance folder is designed to not be under version control and be deployment specific. It's the perfect place to drop things that either change at runtime or configuration files.

You can either explicitly provide the path of the instance folder when creating the Flask application or you can let Flask autodetect the instance folder. For explicit configuration use the `instance_path` parameter:

```
app = Flask(__name__, instance_path='/path/to/instance/folder')
```

Please keep in mind that this path *must* be absolute when provided.

If the `instance_path` parameter is not provided the following default locations are used:

- Uninstalled module:

```
/myapp.py
/instance
```

- Uninstalled package:

```
/myapp
  /__init__.py
/instance
```

- Installed module or package:

```
$PREFIX/lib/pythonX.Y/site-packages/myapp
$PREFIX/var/myapp-instance
```

`$PREFIX` is the prefix of your Python installation. This can be `/usr` or the path to your virtualenv. You can print the value of `sys.prefix` to see what the prefix is set to.

Since the config object provided loading of configuration files from relative filenames we made it possible to change the loading via filenames to be relative to the instance path if wanted. The behavior of relative paths in config files can be flipped between “relative to the application root” (the default) to “relative to instance folder” via the `instance_relative_config` switch to the application constructor:

```
app = Flask(__name__, instance_relative_config=True)
```

Here is a full example of how to configure Flask to preload the config from a module and then override the config from a file in the instance folder if it exists:

```
app = Flask(__name__, instance_relative_config=True)
app.config.from_object('yourapplication.default_settings')
app.config.from_pyfile('application.cfg', silent=True)
```

The path to the instance folder can be found via the `Flask.instance_path`. Flask also provides a shortcut to open a file from the instance folder with `Flask.open_instance_resource()`.

Example usage for both:

```
filename = os.path.join(app.instance_path, 'application.cfg')
with open(filename) as f:
    config = f.read()

# or via open_instance_resource:
with app.open_instance_resource('application.cfg') as f:
    config = f.read()
```

1.10 Signals

Signals are a lightweight way to notify subscribers of certain events during the lifecycle of the application and each request. When an event occurs, it emits the signal, which calls each subscriber.

Signals are implemented by the [Blinker](#) library. See its documentation for detailed information. Flask provides some built-in signals. Extensions may provide their own.

Many signals mirror Flask's decorator-based callbacks with similar names. For example, the `request_started` signal is similar to the `before_request()` decorator. The advantage of signals over handlers is that they can be subscribed to temporarily, and can't directly affect the application. This is useful for testing, metrics, auditing, and more. For example, if you want to know what templates were rendered at what parts of what requests, there is a signal that will notify you of that information.

1.10.1 Core Signals

See [Signals](#) for a list of all built-in signals. The [Application Structure and Lifecycle](#) page also describes the order that signals and decorators execute.

1.10.2 Subscribing to Signals

To subscribe to a signal, you can use the `connect()` method of a signal. The first argument is the function that should be called when the signal is emitted, the optional second argument specifies a sender. To unsubscribe from a signal, you can use the `disconnect()` method.

For all core Flask signals, the sender is the application that issued the signal. When you subscribe to a signal, be sure to also provide a sender unless you really want to listen for signals from all applications. This is especially true if you are developing an extension.

For example, here is a helper context manager that can be used in a unit test to determine which templates were rendered and what variables were passed to the template:

```
from flask import template_rendered
from contextlib import contextmanager

@contextmanager
def captured_templates(app):
    recorded = []
    def record(sender, template, context, **extra):
        recorded.append((template, context))
    template_rendered.connect(record, app)
    try:
        yield recorded
```

(continues on next page)

(continued from previous page)

```
finally:
    template_rendered.disconnect(record, app)
```

This can now easily be paired with a test client:

```
with captured_templates(app) as templates:
    rv = app.test_client().get('/')
    assert rv.status_code == 200
    assert len(templates) == 1
    template, context = templates[0]
    assert template.name == 'index.html'
    assert len(context['items']) == 10
```

Make sure to subscribe with an extra `**extra` argument so that your calls don't fail if Flask introduces new arguments to the signals.

All the template rendering in the code issued by the application `app` in the body of the `with` block will now be recorded in the `templates` variable. Whenever a template is rendered, the template object as well as context are appended to it.

Additionally there is a convenient helper method (`connected_to()`) that allows you to temporarily subscribe a function to a signal with a context manager on its own. Because the return value of the context manager cannot be specified that way, you have to pass the list in as an argument:

```
from flask import template_rendered

def captured_templates(app, recorded, **extra):
    def record(sender, template, context):
        recorded.append((template, context))
    return template_rendered.connected_to(record, app)
```

The example above would then look like this:

```
templates = []
with captured_templates(app, templates, **extra):
    ...
    template, context = templates[0]
```

1.10.3 Creating Signals

If you want to use signals in your own application, you can use the blinker library directly. The most common use case are named signals in a custom `Namespace`. This is what is recommended most of the time:

```
from blinker import Namespace
my_signals = Namespace()
```

Now you can create new signals like this:

```
model_saved = my_signals.signal('model-saved')
```

The name for the signal here makes it unique and also simplifies debugging. You can access the name of the signal with the `name` attribute.

1.10.4 Sending Signals

If you want to emit a signal, you can do so by calling the `send()` method. It accepts a sender as first argument and optionally some keyword arguments that are forwarded to the signal subscribers:

```
class Model(object):
    ...

    def save(self):
        model_saved.send(self)
```

Try to always pick a good sender. If you have a class that is emitting a signal, pass `self` as sender. If you are emitting a signal from a random function, you can pass `current_app._get_current_object()` as sender.

Passing Proxies as Senders

Never pass `current_app` as sender to a signal. Use `current_app._get_current_object()` instead. The reason for this is that `current_app` is a proxy and not the real application object.

1.10.5 Signals and Flask's Request Context

Signals fully support *The Request Context* when receiving signals. Context-local variables are consistently available between `request_started` and `request_finished`, so you can rely on `flask.g` and others as needed. Note the limitations described in *Sending Signals* and the `request_tearing_down` signal.

1.10.6 Decorator Based Signal Subscriptions

You can also easily subscribe to signals by using the `connect_via()` decorator:

```
from flask import template_rendered

@template_rendered.connect_via(app)
def when_template_rendered(sender, template, context, **extra):
    print(f'Template {template.name} is rendered with {context}')
```

1.11 Class-based Views

This page introduces using the `View` and `MethodView` classes to write class-based views.

A class-based view is a class that acts as a view function. Because it is a class, different instances of the class can be created with different arguments, to change the behavior of the view. This is also known as generic, reusable, or pluggable views.

An example of where this is useful is defining a class that creates an API based on the database model it is initialized with.

For more complex API behavior and customization, look into the various API extensions for Flask.

1.11.1 Basic Reusable View

Let's walk through an example converting a view function to a view class. We start with a view function that queries a list of users then renders a template to show the list.

```
@app.route("/users/")
def user_list():
    users = User.query.all()
    return render_template("users.html", users=users)
```

This works for the user model, but let's say you also had more models that needed list pages. You'd need to write another view function for each model, even though the only thing that would change is the model and template name.

Instead, you can write a *View* subclass that will query a model and render a template. As the first step, we'll convert the view to a class without any customization.

```
from flask.views import View

class UserList(View):
    def dispatch_request(self):
        users = User.query.all()
        return render_template("users.html", objects=users)

app.add_url_rule("/users/", view_func=UserList.as_view("user_list"))
```

The *View.dispatch_request()* method is the equivalent of the view function. Calling *View.as_view()* method will create a view function that can be registered on the app with its *add_url_rule()* method. The first argument to *as_view* is the name to use to refer to the view with *url_for()*.

Note: You can't decorate the class with `@app.route()` the way you'd do with a basic view function.

Next, we need to be able to register the same view class for different models and templates, to make it more useful than the original function. The class will take two arguments, the model and template, and store them on *self*. Then *dispatch_request* can reference these instead of hard-coded values.

```
class ListView(View):
    def __init__(self, model, template):
        self.model = model
        self.template = template

    def dispatch_request(self):
        items = self.model.query.all()
        return render_template(self.template, items=items)
```

Remember, we create the view function with *View.as_view()* instead of creating the class directly. Any extra arguments passed to *as_view* are then passed when creating the class. Now we can register the same view to handle multiple models.

```
app.add_url_rule(
    "/users/",
    view_func=ListView.as_view("user_list", User, "users.html"),
)
app.add_url_rule(
```

(continues on next page)

(continued from previous page)

```
"/stories/",  
view_func=ListView.as_view("story_list", Story, "stories.html"),  
)
```

1.11.2 URL Variables

Any variables captured by the URL are passed as keyword arguments to the `dispatch_request` method, as they would be for a regular view function.

```
class DetailView(View):  
    def __init__(self, model):  
        self.model = model  
        self.template = f"{model.__name__.lower()}/detail.html"  
  
    def dispatch_request(self, id):  
        item = self.model.query.get_or_404(id)  
        return render_template(self.template, item=item)  
  
app.add_url_rule(  
    "/users/<int:id>",  
    view_func=DetailView.as_view("user_detail", User)  
)
```

1.11.3 View Lifetime and `self`

By default, a new instance of the view class is created every time a request is handled. This means that it is safe to write other data to `self` during the request, since the next request will not see it, unlike other forms of global state.

However, if your view class needs to do a lot of complex initialization, doing it for every request is unnecessary and can be inefficient. To avoid this, set `View.init_every_request` to `False`, which will only create one instance of the class and use it for every request. In this case, writing to `self` is not safe. If you need to store data during the request, use `g` instead.

In the `ListView` example, nothing writes to `self` during the request, so it is more efficient to create a single instance.

```
class ListView(View):  
    init_every_request = False  
  
    def __init__(self, model, template):  
        self.model = model  
        self.template = template  
  
    def dispatch_request(self):  
        items = self.model.query.all()  
        return render_template(self.template, items=items)
```

Different instances will still be created each for each `as_view` call, but not for each request to those views.

1.11.4 View Decorators

The view class itself is not the view function. View decorators need to be applied to the view function returned by `as_view`, not the class itself. Set `View.decorators` to a list of decorators to apply.

```
class UserList(View):
    decorators = [cache(minutes=2), login_required]

app.add_url_rule('/users/', view_func=UserList.as_view())
```

If you didn't set `decorators`, you could apply them manually instead. This is equivalent to:

```
view = UserList.as_view("users_list")
view = cache(minutes=2)(view)
view = login_required(view)
app.add_url_rule('/users/', view_func=view)
```

Keep in mind that order matters. If you're used to `@decorator` style, this is equivalent to:

```
@app.route("/users/")
@login_required
@cache(minutes=2)
def user_list():
    ...
```

1.11.5 Method Hints

A common pattern is to register a view with `methods=["GET", "POST"]`, then check `request.method == "POST"` to decide what to do. Setting `View.methods` is equivalent to passing the list of methods to `add_url_rule` or `route`.

```
class MyView(View):
    methods = ["GET", "POST"]

    def dispatch_request(self):
        if request.method == "POST":
            ...
            ...

app.add_url_rule('/my-view', view_func=MyView.as_view('my-view'))
```

This is equivalent to the following, except further subclasses can inherit or change the methods.

```
app.add_url_rule(
    "/my-view",
    view_func=MyView.as_view("my-view"),
    methods=["GET", "POST"],
)
```

1.11.6 Method Dispatching and APIs

For APIs it can be helpful to use a different function for each HTTP method. *MethodView* extends the basic *View* to dispatch to different methods of the class based on the request method. Each HTTP method maps to a method of the class with the same (lowercase) name.

MethodView automatically sets *View.methods* based on the methods defined by the class. It even knows how to handle subclasses that override or define other methods.

We can make a generic *ItemAPI* class that provides *get* (detail), *patch* (edit), and *delete* methods for a given model. A *GroupAPI* can provide *get* (list) and *post* (create) methods.

```
from flask.views import MethodView

class ItemAPI(MethodView):
    init_every_request = False

    def __init__(self, model):
        self.model = model
        self.validator = generate_validator(model)

    def _get_item(self, id):
        return self.model.query.get_or_404(id)

    def get(self, id):
        item = self._get_item(id)
        return jsonify(item.to_json())

    def patch(self, id):
        item = self._get_item(id)
        errors = self.validator.validate(item, request.json)

        if errors:
            return jsonify(errors), 400

        item.update_from_json(request.json)
        db.session.commit()
        return jsonify(item.to_json())

    def delete(self, id):
        item = self._get_item(id)
        db.session.delete(item)
        db.session.commit()
        return "", 204

class GroupAPI(MethodView):
    init_every_request = False

    def __init__(self, model):
        self.model = model
        self.validator = generate_validator(model, create=True)

    def get(self):
        items = self.model.query.all()
```

(continues on next page)

(continued from previous page)

```

    return jsonify([item.to_json() for item in items])

def post(self):
    errors = self.validator.validate(request.json)

    if errors:
        return jsonify(errors), 400

    db.session.add(self.model.from_json(request.json))
    db.session.commit()
    return jsonify(item.to_json())

def register_api(app, model, name):
    item = ItemAPI.as_view(f"{name}-item", model)
    group = GroupAPI.as_view(f"{name}-group", model)
    app.add_url_rule(f"/{name}/<int:id>", view_func=item)
    app.add_url_rule(f"/{name}/", view_func=group)

register_api(app, User, "users")
register_api(app, Story, "stories")

```

This produces the following views, a standard REST API!

URL	Method	Description
/users/	GET	List all users
/users/	POST	Create a new user
/users/<id>	GET	Show a single user
/users/<id>	PATCH	Update a user
/users/<id>	DELETE	Delete a user
/stories/	GET	List all stories
/stories/	POST	Create a new story
/stories/<id>	GET	Show a single story
/stories/<id>	PATCH	Update a story
/stories/<id>	DELETE	Delete a story

1.12 Application Structure and Lifecycle

Flask makes it pretty easy to write a web application. But there are quite a few different parts to an application and to each request it handles. Knowing what happens during application setup, serving, and handling requests will help you know what's possible in Flask and how to structure your application.

1.12.1 Application Setup

The first step in creating a Flask application is creating the application object. Each Flask application is an instance of the *Flask* class, which collects all configuration, extensions, and views.

```
from flask import Flask

app = Flask(__name__)
app.config.from_mapping(
    SECRET_KEY="dev",
)
app.config.from_prefixed_env()

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

This is known as the “application setup phase”, it’s the code you write that’s outside any view functions or other handlers. It can be split up between different modules and sub-packages, but all code that you want to be part of your application must be imported in order for it to be registered.

All application setup must be completed before you start serving your application and handling requests. This is because WSGI servers divide work between multiple workers, or can be distributed across multiple machines. If the configuration changed in one worker, there’s no way for Flask to ensure consistency between other workers.

Flask tries to help developers catch some of these setup ordering issues by showing an error if setup-related methods are called after requests are handled. In that case you’ll see this error:

The setup method ‘route’ can no longer be called on the application. It has already handled its first request, any changes will not be applied consistently. Make sure all imports, decorators, functions, etc. needed to set up the application are done before running it.

However, it is not possible for Flask to detect all cases of out-of-order setup. In general, don’t do anything to modify the Flask app object and Blueprint objects from within view functions that run during requests. This includes:

- Adding routes, view functions, and other request handlers with `@app.route`, `@app.errorhandler`, `@app.before_request`, etc.
- Registering blueprints.
- Loading configuration with `app.config`.
- Setting up the Jinja template environment with `app.jinja_env`.
- Setting a session interface, instead of the default `itsdangerous` cookie.
- Setting a JSON provider with `app.json`, instead of the default provider.
- Creating and initializing Flask extensions.

1.12.2 Serving the Application

Flask is a WSGI application framework. The other half of WSGI is the WSGI server. During development, Flask, through Werkzeug, provides a development WSGI server with the `flask run` CLI command. When you are done with development, use a production server to serve your application, see [Deploying to Production](#).

Regardless of what server you're using, it will follow the [PEP 3333](#) WSGI spec. The WSGI server will be told how to access your Flask application object, which is the WSGI application. Then it will start listening for HTTP requests, translate the request data into a WSGI environ, and call the WSGI application with that data. The WSGI application will return data that is translated into an HTTP response.

1. Browser or other client makes HTTP request.
2. WSGI server receives request.
3. WSGI server converts HTTP data to WSGI `environ` dict.
4. WSGI server calls WSGI application with the `environ`.
5. Flask, the WSGI application, does all its internal processing to route the request to a view function, handle errors, etc.
6. Flask translates View function return into WSGI response data, passes it to WSGI server.
7. WSGI server creates and send an HTTP response.
8. Client receives the HTTP response.

Middleware

The WSGI application above is a callable that behaves in a certain way. Middleware is a WSGI application that wraps another WSGI application. It's a similar concept to Python decorators. The outermost middleware will be called by the server. It can modify the data passed to it, then call the WSGI application (or further middleware) that it wraps, and so on. And it can take the return value of that call and modify it further.

From the WSGI server's perspective, there is one WSGI application, the one it calls directly. Typically, Flask is the "real" application at the end of the chain of middleware. But even Flask can call further WSGI applications, although that's an advanced, uncommon use case.

A common middleware you'll see used with Flask is Werkzeug's [ProxyFix](#), which modifies the request to look like it came directly from a client even if it passed through HTTP proxies on the way. There are other middleware that can handle serving static files, authentication, etc.

1.12.3 How a Request is Handled

For us, the interesting part of the steps above is when Flask gets called by the WSGI server (or middleware). At that point, it will do quite a lot to handle the request and generate the response. At the most basic, it will match the URL to a view function, call the view function, and pass the return value back to the server. But there are many more parts that you can use to customize its behavior.

1. WSGI server calls the Flask object, which calls `Flask.wsgi_app()`.
2. A `RequestContext` object is created. This converts the WSGI `environ` dict into a `Request` object. It also creates an `AppContext` object.
3. The `app context` is pushed, which makes `current_app` and `g` available.
4. The `appcontext_pushed` signal is sent.
5. The `request context` is pushed, which makes `request` and `session` available.

6. The session is opened, loading any existing session data using the app's `session_interface`, an instance of `SessionInterface`.
7. The URL is matched against the URL rules registered with the `route()` decorator during application setup. If there is no match, the error - usually a 404, 405, or redirect - is stored to be handled later.
8. The `request_started` signal is sent.
9. Any `url_value_preprocessor()` decorated functions are called.
10. Any `before_request()` decorated functions are called. If any of these function returns a value it is treated as the response immediately.
11. If the URL didn't match a route a few steps ago, that error is raised now.
12. The `route()` decorated view function associated with the matched URL is called and returns a value to be used as the response.
13. If any step so far raised an exception, and there is an `errorhandler()` decorated function that matches the exception class or HTTP error code, it is called to handle the error and return a response.
14. Whatever returned a response value - a before request function, the view, or an error handler, that value is converted to a `Response` object.
15. Any `after_this_request()` decorated functions are called, then cleared.
16. Any `after_request()` decorated functions are called, which can modify the response object.
17. The session is saved, persisting any modified session data using the app's `session_interface`.
18. The `request_finished` signal is sent.
19. If any step so far raised an exception, and it was not handled by an error handler function, it is handled now. HTTP exceptions are treated as responses with their corresponding status code, other exceptions are converted to a generic 500 response. The `got_request_exception` signal is sent.
20. The response object's status, headers, and body are returned to the WSGI server.
21. Any `teardown_request()` decorated functions are called.
22. The `request_tearing_down` signal is sent.
23. The request context is popped, `request` and `session` are no longer available.
24. Any `teardown_appcontext()` decorated functions are called.
25. The `appcontext_tearing_down` signal is sent.
26. The app context is popped, `current_app` and `g` are no longer available.
27. The `appcontext_popped` signal is sent.

There are even more decorators and customization points than this, but that aren't part of every request lifecycle. They're more specific to certain things you might use during a request, such as templates, building URLs, or handling JSON data. See the rest of this documentation, as well as the *API* to explore further.

1.13 The Application Context

The application context keeps track of the application-level data during a request, CLI command, or other activity. Rather than passing the application around to each function, the `current_app` and `g` proxies are accessed instead.

This is similar to *The Request Context*, which keeps track of request-level data during a request. A corresponding application context is pushed when a request context is pushed.

1.13.1 Purpose of the Context

The *Flask* application object has attributes, such as `config`, that are useful to access within views and *CLI commands*. However, importing the app instance within the modules in your project is prone to circular import issues. When using the *app factory pattern* or writing reusable *blueprints* or *extensions* there won't be an app instance to import at all.

Flask solves this issue with the *application context*. Rather than referring to an app directly, you use the `current_app` proxy, which points to the application handling the current activity.

Flask automatically *pushes* an application context when handling a request. View functions, error handlers, and other functions that run during a request will have access to `current_app`.

Flask will also automatically push an app context when running CLI commands registered with *Flask.cli* using `@app.cli.command()`.

1.13.2 Lifetime of the Context

The application context is created and destroyed as necessary. When a Flask application begins handling a request, it pushes an application context and a *request context*. When the request ends it pops the request context then the application context. Typically, an application context will have the same lifetime as a request.

See *The Request Context* for more information about how the contexts work and the full life cycle of a request.

1.13.3 Manually Push a Context

If you try to access `current_app`, or anything that uses it, outside an application context, you'll get this error message:

```
RuntimeError: Working outside of application context.
```

This typically means that you attempted to use functionality that needed to interface with the current application object in some way. To solve this, set up an application context with `app.app_context()`.

If you see that error while configuring your application, such as when initializing an extension, you can push a context manually since you have direct access to the app. Use `app_context()` in a `with` block, and everything that runs in the block will have access to `current_app`.

```
def create_app():
    app = Flask(__name__)

    with app.app_context():
        init_db()

    return app
```

If you see that error somewhere else in your code not related to configuring the application, it most likely indicates that you should move that code into a view function or CLI command.

1.13.4 Storing Data

The application context is a good place to store common data during a request or CLI command. Flask provides the *g object* for this purpose. It is a simple namespace object that has the same lifetime as an application context.

Note: The *g* name stands for “global”, but that is referring to the data being global *within a context*. The data on *g* is lost after the context ends, and it is not an appropriate place to store data between requests. Use the *session* or a database to store data across requests.

A common use for *g* is to manage resources during a request.

1. `get_X()` creates resource *X* if it does not exist, caching it as `g.X`.
2. `teardown_X()` closes or otherwise deallocates the resource if it exists. It is registered as a `teardown_appcontext()` handler.

For example, you can manage a database connection using this pattern:

```
from flask import g

def get_db():
    if 'db' not in g:
        g.db = connect_to_database()

    return g.db

@app.teardown_appcontext
def teardown_db(exception):
    db = g.pop('db', None)

    if db is not None:
        db.close()
```

During a request, every call to `get_db()` will return the same connection, and it will be closed automatically at the end of the request.

You can use `LocalProxy` to make a new context local from `get_db()`:

```
from werkzeug.local import LocalProxy
db = LocalProxy(get_db)
```

Accessing `db` will call `get_db` internally, in the same way that `current_app` works.

1.13.5 Events and Signals

The application will call functions registered with `teardown_appcontext()` when the application context is popped.

The following signals are sent: `appcontext_pushed`, `appcontext_tearing_down`, and `appcontext_popped`.

1.14 The Request Context

The request context keeps track of the request-level data during a request. Rather than passing the request object to each function that runs during a request, the `request` and `session` proxies are accessed instead.

This is similar to *The Application Context*, which keeps track of the application-level data independent of a request. A corresponding application context is pushed when a request context is pushed.

1.14.1 Purpose of the Context

When the *Flask* application handles a request, it creates a `Request` object based on the environment it received from the WSGI server. Because a *worker* (thread, process, or coroutine depending on the server) handles only one request at a time, the request data can be considered global to that worker during that request. Flask uses the term *context local* for this.

Flask automatically *pushes* a request context when handling a request. View functions, error handlers, and other functions that run during a request will have access to the `request` proxy, which points to the request object for the current request.

1.14.2 Lifetime of the Context

When a Flask application begins handling a request, it pushes a request context, which also pushes an *app context*. When the request ends it pops the request context then the application context.

The context is unique to each thread (or other worker type). `request` cannot be passed to another thread, the other thread has a different context space and will not know about the request the parent thread was pointing to.

Context locals are implemented using Python's `contextvars` and Werkzeug's `LocalProxy`. Python manages the lifetime of context vars automatically, and local proxy wraps that low-level interface to make the data easier to work with.

1.14.3 Manually Push a Context

If you try to access `request`, or anything that uses it, outside a request context, you'll get this error message:

```
RuntimeError: Working outside of request context.
```

```
This typically means that you attempted to use functionality that
needed an active HTTP request. Consult the documentation on testing
for information about how to avoid this problem.
```

This should typically only happen when testing code that expects an active request. One option is to use the `test_client` to simulate a full request. Or you can use `test_request_context()` in a with block, and everything that runs in the block will have access to `request`, populated with your test data.

```
def generate_report(year):
    format = request.args.get("format")
    ...

with app.test_request_context(
    "/make_report/2017", query_string={"format": "short"}
):
    generate_report()
```

If you see that error somewhere else in your code not related to testing, it most likely indicates that you should move that code into a view function.

For information on how to use the request context from the interactive Python shell, see [Working with the Shell](#).

1.14.4 How the Context Works

The `Flask.wsgi_app()` method is called to handle each request. It manages the contexts during the request. Internally, the request and application contexts work like stacks. When contexts are pushed, the proxies that depend on them are available and point at information from the top item.

When the request starts, a `RequestContext` is created and pushed, which creates and pushes an `AppContext` first if a context for that application is not already the top context. While these contexts are pushed, the `current_app`, `g`, `request`, and `session` proxies are available to the original thread handling the request.

Other contexts may be pushed to change the proxies during a request. While this is not a common pattern, it can be used in advanced applications to, for example, do internal redirects or chain different applications together.

After the request is dispatched and a response is generated and sent, the request context is popped, which then pops the application context. Immediately before they are popped, the `teardown_request()` and `teardown_appcontext()` functions are executed. These execute even if an unhandled exception occurred during dispatch.

1.14.5 Callbacks and Errors

Flask dispatches a request in multiple stages which can affect the request, response, and how errors are handled. The contexts are active during all of these stages.

A `Blueprint` can add handlers for these events that are specific to the blueprint. The handlers for a blueprint will run if the blueprint owns the route that matches the request.

1. Before each request, `before_request()` functions are called. If one of these functions return a value, the other functions are skipped. The return value is treated as the response and the view function is not called.
2. If the `before_request()` functions did not return a response, the view function for the matched route is called and returns a response.
3. The return value of the view is converted into an actual response object and passed to the `after_request()` functions. Each function returns a modified or new response object.
4. After the response is returned, the contexts are popped, which calls the `teardown_request()` and `teardown_appcontext()` functions. These functions are called even if an unhandled exception was raised at any point above.

If an exception is raised before the teardown functions, Flask tries to match it with an `errorhandler()` function to handle the exception and return a response. If no error handler is found, or the handler itself raises an exception, Flask returns a generic `500 Internal Server Error` response. The teardown functions are still called, and are passed the exception object.

If debug mode is enabled, unhandled exceptions are not converted to a 500 response and instead are propagated to the WSGI server. This allows the development server to present the interactive debugger with the traceback.

Teardown Callbacks

The teardown callbacks are independent of the request dispatch, and are instead called by the contexts when they are popped. The functions are called even if there is an unhandled exception during dispatch, and for manually pushed contexts. This means there is no guarantee that any other parts of the request dispatch have run first. Be sure to write these functions in a way that does not depend on other callbacks and will not fail.

During testing, it can be useful to defer popping the contexts after the request ends, so that their data can be accessed in the test function. Use the `test_client()` as a `with` block to preserve the contexts until the `with` block exits.

```
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def hello():
    print('during view')
    return 'Hello, World!'

@app.teardown_request
def show_teardown(exception):
    print('after with block')

with app.test_request_context():
    print('during with block')

# teardown functions are called after the context with block exits

with app.test_client() as client:
    client.get('/')
    # the contexts are not popped even though the request ended
    print(request.path)

# the contexts are popped and teardown functions are called after
# the client with block exits
```

Signals

The following signals are sent:

1. `request_started` is sent before the `before_request()` functions are called.
2. `request_finished` is sent after the `after_request()` functions are called.
3. `got_request_exception` is sent when an exception begins to be handled, but before an `errorhandler()` is looked up or called.
4. `request_tearing_down` is sent after the `teardown_request()` functions are called.

1.14.6 Notes On Proxies

Some of the objects provided by Flask are proxies to other objects. The proxies are accessed in the same way for each worker thread, but point to the unique object bound to each worker behind the scenes as described on this page.

Most of the time you don't have to care about that, but there are some exceptions where it is good to know that this object is actually a proxy:

- The proxy objects cannot fake their type as the actual object types. If you want to perform instance checks, you have to do that on the object being proxied.
- The reference to the proxied object is needed in some situations, such as sending *Signals* or passing data to a background thread.

If you need to access the underlying object that is proxied, use the `_get_current_object()` method:

```
app = current_app._get_current_object()
my_signal.send(app)
```

1.15 Modular Applications with Blueprints

New in version 0.7.

Flask uses a concept of *blueprints* for making application components and supporting common patterns within an application or across applications. Blueprints can greatly simplify how large applications work and provide a central means for Flask extensions to register operations on applications. A *Blueprint* object works similarly to a *Flask* application object, but it is not actually an application. Rather it is a *blueprint* of how to construct or extend an application.

1.15.1 Why Blueprints?

Blueprints in Flask are intended for these cases:

- Factor an application into a set of blueprints. This is ideal for larger applications; a project could instantiate an application object, initialize several extensions, and register a collection of blueprints.
- Register a blueprint on an application at a URL prefix and/or subdomain. Parameters in the URL prefix/subdomain become common view arguments (with defaults) across all view functions in the blueprint.
- Register a blueprint multiple times on an application with different URL rules.
- Provide template filters, static files, templates, and other utilities through blueprints. A blueprint does not have to implement applications or view functions.
- Register a blueprint on an application for any of these cases when initializing a Flask extension.

A blueprint in Flask is not a pluggable app because it is not actually an application – it's a set of operations which can be registered on an application, even multiple times. Why not have multiple application objects? You can do that (see *Application Dispatching*), but your applications will have separate configs and will be managed at the WSGI layer.

Blueprints instead provide separation at the Flask level, share application config, and can change an application object as necessary with being registered. The downside is that you cannot unregister a blueprint once an application was created without having to destroy the whole application object.

1.15.2 The Concept of Blueprints

The basic concept of blueprints is that they record operations to execute when registered on an application. Flask associates view functions with blueprints when dispatching requests and generating URLs from one endpoint to another.

1.15.3 My First Blueprint

This is what a very basic blueprint looks like. In this case we want to implement a blueprint that does simple rendering of static templates:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                       template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template(f'pages/{page}.html')
    except TemplateNotFound:
        abort(404)
```

When you bind a function with the help of the `@simple_page.route` decorator, the blueprint will record the intention of registering the function `show` on the application when it's later registered. Additionally it will prefix the endpoint of the function with the name of the blueprint which was given to the *Blueprint* constructor (in this case also `simple_page`). The blueprint's name does not modify the URL, only the endpoint.

1.15.4 Registering Blueprints

So how do you register that blueprint? Like this:

```
from flask import Flask
from yourapplication.simple_page import simple_page

app = Flask(__name__)
app.register_blueprint(simple_page)
```

If you check the rules registered on the application, you will find these:

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
     <Rule '/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
     <Rule '/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

The first one is obviously from the application itself for the static files. The other two are for the `show` function of the `simple_page` blueprint. As you can see, they are also prefixed with the name of the blueprint and separated by a dot (`.`).

Blueprints however can also be mounted at different locations:

```
app.register_blueprint(simple_page, url_prefix='/pages')
```

And sure enough, these are the generated rules:

```
>>> app.url_map
Map([<Rule '/static/<filename>' (HEAD, OPTIONS, GET) -> static>,
 <Rule '/pages/<page>' (HEAD, OPTIONS, GET) -> simple_page.show>,
 <Rule '/pages/' (HEAD, OPTIONS, GET) -> simple_page.show>])
```

On top of that you can register blueprints multiple times though not every blueprint might respond properly to that. In fact it depends on how the blueprint is implemented if it can be mounted more than once.

1.15.5 Nesting Blueprints

It is possible to register a blueprint on another blueprint.

```
parent = Blueprint('parent', __name__, url_prefix='/parent')
child = Blueprint('child', __name__, url_prefix='/child')
parent.register_blueprint(child)
app.register_blueprint(parent)
```

The child blueprint will gain the parent's name as a prefix to its name, and child URLs will be prefixed with the parent's URL prefix.

```
url_for('parent.child.create')
/parent/child/create
```

In addition a child blueprint's will gain their parent's subdomain, with their subdomain as prefix if present i.e.

```
parent = Blueprint('parent', __name__, subdomain='parent')
child = Blueprint('child', __name__, subdomain='child')
parent.register_blueprint(child)
app.register_blueprint(parent)

url_for('parent.child.create', _external=True)
"child.parent.domain.tld"
```

Blueprint-specific before request functions, etc. registered with the parent will trigger for the child. If a child does not have an error handler that can handle a given exception, the parent's will be tried.

1.15.6 Blueprint Resources

Blueprints can provide resources as well. Sometimes you might want to introduce a blueprint only for the resources it provides.

Blueprint Resource Folder

Like for regular applications, blueprints are considered to be contained in a folder. While multiple blueprints can originate from the same folder, it does not have to be the case and it's usually not recommended.

The folder is inferred from the second argument to `Blueprint` which is usually `__name__`. This argument specifies what logical Python module or package corresponds to the blueprint. If it points to an actual Python package that package (which is a folder on the filesystem) is the resource folder. If it's a module, the package the module is contained in will be the resource folder. You can access the `Blueprint.root_path` property to see what the resource folder is:

```
>>> simple_page.root_path
'/Users/username/TestProject/yourapplication'
```

To quickly open sources from this folder you can use the `open_resource()` function:

```
with simple_page.open_resource('static/style.css') as f:
    code = f.read()
```

Static Files

A blueprint can expose a folder with static files by providing the path to the folder on the filesystem with the `static_folder` argument. It is either an absolute path or relative to the blueprint's location:

```
admin = Blueprint('admin', __name__, static_folder='static')
```

By default the rightmost part of the path is where it is exposed on the web. This can be changed with the `static_url_path` argument. Because the folder is called `static` here it will be available at the `url_prefix` of the blueprint + `/static`. If the blueprint has the prefix `/admin`, the static URL will be `/admin/static`.

The endpoint is named `blueprint_name.static`. You can generate URLs to it with `url_for()` like you would with the static folder of the application:

```
url_for('admin.static', filename='style.css')
```

However, if the blueprint does not have a `url_prefix`, it is not possible to access the blueprint's static folder. This is because the URL would be `/static` in this case, and the application's `/static` route takes precedence. Unlike template folders, blueprint static folders are not searched if the file does not exist in the application static folder.

Templates

If you want the blueprint to expose templates you can do that by providing the `template_folder` parameter to the `Blueprint` constructor:

```
admin = Blueprint('admin', __name__, template_folder='templates')
```

For static files, the path can be absolute or relative to the blueprint resource folder.

The template folder is added to the search path of templates but with a lower priority than the actual application's template folder. That way you can easily override templates that a blueprint provides in the actual application. This also means that if you don't want a blueprint template to be accidentally overridden, make sure that no other blueprint or actual application template has the same relative path. When multiple blueprints provide the same relative template path the first blueprint registered takes precedence over the others.

So if you have a blueprint in the folder `yourapplication/admin` and you want to render the template `'admin/index.html'` and you have provided `templates` as a `template_folder` you will have to create a file like this:

`yourapplication/admin/templates/admin/index.html`. The reason for the extra `admin` folder is to avoid getting our template overridden by a template named `index.html` in the actual application template folder.

To further reiterate this: if you have a blueprint named `admin` and you want to render a template called `index.html` which is specific to this blueprint, the best idea is to lay out your templates like this:

```
yourpackage/  
  blueprints/  
    admin/  
      templates/  
        admin/  
          index.html  
      __init__.py
```

And then when you want to render the template, use `admin/index.html` as the name to look up the template by. If you encounter problems loading the correct templates enable the `EXPLAIN_TEMPLATE_LOADING` config variable which will instruct Flask to print out the steps it goes through to locate templates on every `render_template` call.

1.15.7 Building URLs

If you want to link from one page to another you can use the `url_for()` function just like you normally would do just that you prefix the URL endpoint with the name of the blueprint and a dot (`.`):

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

Additionally if you are in a view function of a blueprint or a rendered template and you want to link to another endpoint of the same blueprint, you can use relative redirects by prefixing the endpoint with a dot only:

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

This will link to `admin.index` for instance in case the current request was dispatched to any other `admin` blueprint endpoint.

1.15.8 Blueprint Error Handlers

Blueprints support the `errorhandler` decorator just like the `Flask` application object, so it is easy to make Blueprint-specific custom error pages.

Here is an example for a “404 Page Not Found” exception:

```
@simple_page.errorhandler(404)  
def page_not_found(e):  
    return render_template('pages/404.html')
```

Most errorhandlers will simply work as expected; however, there is a caveat concerning handlers for 404 and 405 exceptions. These errorhandlers are only invoked from an appropriate `raise` statement or a call to `abort` in another of the blueprint’s view functions; they are not invoked by, e.g., an invalid URL access. This is because the blueprint does not “own” a certain URL space, so the application instance has no way of knowing which blueprint error handler it should run if given an invalid URL. If you would like to execute different handling strategies for these errors based on URL prefixes, they may be defined at the application level using the `request proxy` object:

```
@app.errorhandler(404)  
@app.errorhandler(405)
```

(continues on next page)

(continued from previous page)

```
def _handle_api_error(ex):
    if request.path.startswith('/api/'):
        return jsonify(error=str(ex)), ex.code
    else:
        return ex
```

See *Handling Application Errors*.

1.16 Extensions

Extensions are extra packages that add functionality to a Flask application. For example, an extension might add support for sending email or connecting to a database. Some extensions add entire new frameworks to help build certain types of applications, like a REST API.

1.16.1 Finding Extensions

Flask extensions are usually named “Flask-Foo” or “Foo-Flask”. You can search PyPI for packages tagged with `Framework :: Flask`.

1.16.2 Using Extensions

Consult each extension’s documentation for installation, configuration, and usage instructions. Generally, extensions pull their own configuration from `app.config` and are passed an application instance during initialization. For example, an extension called “Flask-Foo” might be used like this:

```
from flask_foo import Foo

foo = Foo()

app = Flask(__name__)
app.config.update(
    FOO_BAR='baz',
    FOO_SPAM='eggs',
)

foo.init_app(app)
```

1.16.3 Building Extensions

While PyPI contains many Flask extensions, you may not find an extension that fits your need. If this is the case, you can create your own, and publish it for others to use as well. Read *Flask Extension Development* to develop your own Flask extension.

1.17 Command Line Interface

Installing Flask installs the `flask` script, a [Click](#) command line interface, in your virtualenv. Executed from the terminal, this script gives access to built-in, extension, and application-defined commands. The `--help` option will give more information about any commands and options.

1.17.1 Application Discovery

The `flask` command is installed by Flask, not your application; it must be told where to find your application in order to use it. The `--app` option is used to specify how to load the application.

While `--app` supports a variety of options for specifying your application, most use cases should be simple. Here are the typical values:

(nothing) The name “app” or “wsgi” is imported (as a “.py” file, or package), automatically detecting an app (app or application) or factory (create_app or make_app).

--app hello The given name is imported, automatically detecting an app (app or application) or factory (create_app or make_app).

`--app` has three parts: an optional path that sets the current working directory, a Python file or dotted import path, and an optional variable name of the instance or factory. If the name is a factory, it can optionally be followed by arguments in parentheses. The following values demonstrate these parts:

--app src/hello Sets the current working directory to `src` then imports `hello`.

--app hello.web Imports the path `hello.web`.

--app hello:app2 Uses the `app2` Flask instance in `hello`.

--app 'hello:create_app("dev")' The `create_app` factory in `hello` is called with the string `'dev'` as the argument.

If `--app` is not set, the command will try to import “app” or “wsgi” (as a “.py” file, or package) and try to detect an application instance or factory.

Within the given import, the command looks for an application instance named `app` or `application`, then any application instance. If no instance is found, the command looks for a factory function named `create_app` or `make_app` that returns an instance.

If parentheses follow the factory name, their contents are parsed as Python literals and passed as arguments and keyword arguments to the function. This means that strings must still be in quotes.

1.17.2 Run the Development Server

The `run` command will start the development server. It replaces the `Flask.run()` method in most cases.

```
$ flask --app hello run
* Serving Flask app "hello"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Warning: Do not use this command to run your application in production. Only use the development server during development. The development server is provided for convenience, but is not designed to be particularly secure, stable, or efficient. See [Deploying to Production](#) for how to run in production.

If another program is already using port 5000, you'll see `OSError: [Errno 98]` or `OSError: [WinError 10013]` when the server tries to start. See [Address already in use](#) for how to handle that.

Debug Mode

In debug mode, the `flask run` command will enable the interactive debugger and the reloader by default, and make errors easier to see and debug. To enable debug mode, use the `--debug` option.

```
$ flask --app hello run --debug
* Serving Flask app "hello"
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with inotify reloader
* Debugger is active!
* Debugger PIN: 223-456-919
```

The `--debug` option can also be passed to the top level `flask` command to enable debug mode for any command. The following two run calls are equivalent.

```
$ flask --app hello --debug run
$ flask --app hello run --debug
```

Watch and Ignore Files with the Reloader

When using debug mode, the reloader will trigger whenever your Python code or imported modules change. The reloader can watch additional files with the `--extra-files` option. Multiple paths are separated with `:`, or `;` on Windows.

```
$ flask run --extra-files file1:dirA/file2:dirB/
* Running on http://127.0.0.1:8000/
* Detected change in '/path/to/file1', reloading
```

The reloader can also ignore files using `fnmatch` patterns with the `--exclude-patterns` option. Multiple patterns are separated with `:`, or `;` on Windows.

1.17.3 Open a Shell

To explore the data in your application, you can start an interactive Python shell with the `shell` command. An application context will be active, and the app instance will be imported.

```
$ flask shell
Python 3.10.0 (default, Oct 27 2021, 06:59:51) [GCC 11.1.0] on linux
App: example [production]
Instance: /home/david/Projects/pallets/flask/instance
>>>
```

Use `shell_context_processor()` to add other automatic imports.

1.17.4 Environment Variables From dotenv

The `flask` command supports setting any option for any command with environment variables. The variables are named like `FLASK_OPTION` or `FLASK_COMMAND_OPTION`, for example `FLASK_APP` or `FLASK_RUN_PORT`.

Rather than passing options every time you run a command, or environment variables every time you open a new terminal, you can use Flask's `dotenv` support to set environment variables automatically.

If `python-dotenv` is installed, running the `flask` command will set environment variables defined in the files `.env` and `.flaskenv`. You can also specify an extra file to load with the `--env-file` option. Dotenv files can be used to avoid having to set `--app` or `FLASK_APP` manually, and to set configuration using environment variables similar to how some deployment services work.

Variables set on the command line are used over those set in `.env`, which are used over those set in `.flaskenv`. `.flaskenv` should be used for public variables, such as `FLASK_APP`, while `.env` should not be committed to your repository so that it can set private variables.

Directories are scanned upwards from the directory you call `flask` from to locate the files.

The files are only loaded by the `flask` command or calling `run()`. If you would like to load these files when running in production, you should call `load_dotenv()` manually.

Setting Command Options

Click is configured to load default values for command options from environment variables. The variables use the pattern `FLASK_COMMAND_OPTION`. For example, to set the port for the run command, instead of `flask run --port 8000`:

Bash

```
$ export FLASK_RUN_PORT=8000
$ flask run
* Running on http://127.0.0.1:8000/
```

Fish

```
$ set -x FLASK_RUN_PORT 8000
$ flask run
* Running on http://127.0.0.1:8000/
```

CMD

```
> set FLASK_RUN_PORT=8000
> flask run
* Running on http://127.0.0.1:8000/
```

Powershell

```
> $env:FLASK_RUN_PORT = 8000
> flask run
* Running on http://127.0.0.1:8000/
```

These can be added to the `.flaskenv` file just like `FLASK_APP` to control default command options.

Disable dotenv

The flask command will show a message if it detects dotenv files but python-dotenv is not installed.

```
$ flask run
* Tip: There are .env files present. Do "pip install python-dotenv" to use them.
```

You can tell Flask not to load dotenv files even when python-dotenv is installed by setting the `FLASK_SKIP_DOTENV` environment variable. This can be useful if you want to load them manually, or if you're using a project runner that loads them already. Keep in mind that the environment variables must be set before the app loads or it won't configure as expected.

Bash

```
$ export FLASK_SKIP_DOTENV=1
$ flask run
```

Fish

```
$ set -x FLASK_SKIP_DOTENV 1
$ flask run
```

CMD

```
> set FLASK_SKIP_DOTENV=1
> flask run
```

Powershell

```
> $env:FLASK_SKIP_DOTENV = 1
> flask run
```

1.17.5 Environment Variables From virtualenv

If you do not want to install dotenv support, you can still set environment variables by adding them to the end of the virtualenv's `activate` script. Activating the virtualenv will set the variables.

Bash

Unix Bash, `.venv/bin/activate`:

```
$ export FLASK_APP=hello
```

Fish

Fish, `.venv/bin/activate.fish`:

```
$ set -x FLASK_APP hello
```

CMD

Windows CMD, `.venv\Scripts\activate.bat`:

```
> set FLASK_APP=hello
```

Powershell

Windows Powershell, `.venv\Scripts\activate.ps1`:

```
> $env:FLASK_APP = "hello"
```

It is preferred to use dotenv support over this, since `.flaskenv` can be committed to the repository so that it works automatically wherever the project is checked out.

1.17.6 Custom Commands

The flask command is implemented using [Click](#). See that project's documentation for full information about writing commands.

This example adds the command `create-user` that takes the argument `name`.

```
import click
from flask import Flask

app = Flask(__name__)

@app.cli.command("create-user")
@click.argument("name")
def create_user(name):
    ...
```

```
$ flask create-user admin
```

This example adds the same command, but as `user create`, a command in a group. This is useful if you want to organize multiple related commands.

```
import click
from flask import Flask
from flask.cli import AppGroup

app = Flask(__name__)
user_cli = AppGroup('user')

@user_cli.command('create')
@click.argument('name')
def create_user(name):
    ...

app.cli.add_command(user_cli)
```

```
$ flask user create demo
```

See *Running Commands with the CLI Runner* for an overview of how to test your custom commands.

Registering Commands with Blueprints

If your application uses blueprints, you can optionally register CLI commands directly onto them. When your blueprint is registered onto your application, the associated commands will be available to the `flask` command. By default, those commands will be nested in a group matching the name of the blueprint.

```
from flask import Blueprint

bp = Blueprint('students', __name__)

@bp.cli.command('create')
@click.argument('name')
def create(name):
    ...

app.register_blueprint(bp)
```

```
$ flask students create alice
```

You can alter the group name by specifying the `cli_group` parameter when creating the *Blueprint* object, or later with `app.register_blueprint(bp, cli_group='...')`. The following are equivalent:

```
bp = Blueprint('students', __name__, cli_group='other')
# or
app.register_blueprint(bp, cli_group='other')
```

```
$ flask other create alice
```

Specifying `cli_group=None` will remove the nesting and merge the commands directly to the application's level:

```
bp = Blueprint('students', __name__, cli_group=None)
# or
app.register_blueprint(bp, cli_group=None)
```

```
$ flask create alice
```

Application Context

Commands added using the Flask app's `cli` or *FlaskGroup* `command()` decorator will be executed with an application context pushed, so your custom commands and parameters have access to the app and its configuration. The `with_appcontext()` decorator can be used to get the same behavior, but is not needed in most cases.

```
import click
from flask.cli import with_appcontext

@click.command()
@with_appcontext
def do_work():
    ...

app.cli.add_command(do_work)
```

1.17.7 Plugins

Flask will automatically load commands specified in the `flask.commands` entry point. This is useful for extensions that want to add commands when they are installed. Entry points are specified in `pyproject.toml`:

```
[project.entry-points."flask.commands"]
my-command = "my_extension.commands:cli"
```

Inside `my_extension/commands.py` you can then export a Click object:

```
import click

@click.command()
def cli():
    ...
```

Once that package is installed in the same virtualenv as your Flask project, you can run `flask my-command` to invoke the command.

1.17.8 Custom Scripts

When you are using the app factory pattern, it may be more convenient to define your own Click script. Instead of using `--app` and letting Flask load your application, you can create your own Click object and export it as a `console script` entry point.

Create an instance of `FlaskGroup` and pass it the factory:

```
import click
from flask import Flask
from flask.cli import FlaskGroup

def create_app():
    app = Flask('wiki')
    # other setup
    return app

@click.group(cls=FlaskGroup, create_app=create_app)
def cli():
    """Management script for the Wiki application."""
```

Define the entry point in `pyproject.toml`:

```
[project.scripts]
wiki = "wiki:cli"
```

Install the application in the virtualenv in editable mode and the custom script is available. Note that you don't need to set `--app`.

```
$ pip install -e .
$ wiki run
```

Errors in Custom Scripts

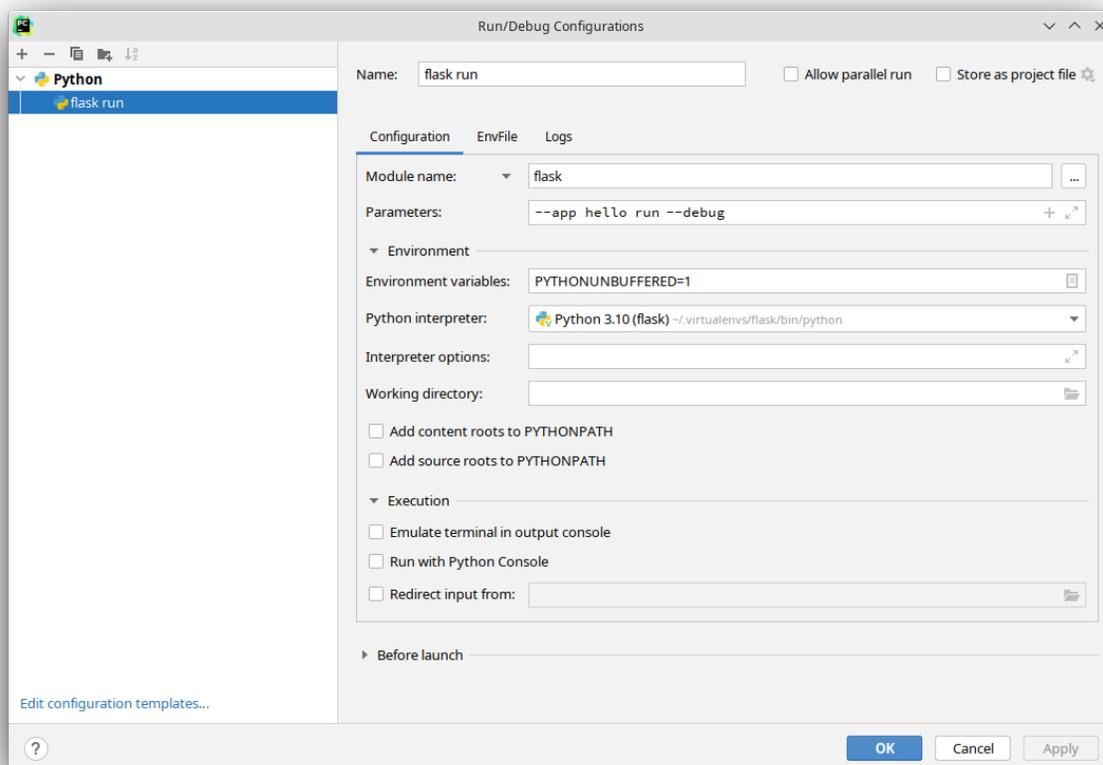
When using a custom script, if you introduce an error in your module-level code, the reloader will fail because it can no longer load the entry point.

The `flask` command, being separate from your code, does not have this issue and is recommended in most cases.

1.17.9 PyCharm Integration

PyCharm Professional provides a special Flask run configuration to run the development server. For the Community Edition, and for other commands besides `run`, you need to create a custom run configuration. These instructions should be similar for any other IDE you use.

In PyCharm, with your project open, click on *Run* from the menu bar and go to *Edit Configurations*. You'll see a screen similar to this:



Once you create a configuration for the `flask run`, you can copy and change it to call any other command.

Click the `+` (*Add New Configuration*) button and select *Python*. Give the configuration a name such as “flask run”.

Click the *Script path* dropdown and change it to *Module name*, then input `flask`.

The *Parameters* field is set to the CLI command to execute along with any arguments. This example uses `--app hello run --debug`, which will run the development server in debug mode. `--app hello` should be the import or file with your Flask app.

If you installed your project as a package in your virtualenv, you may uncheck the *PYTHONPATH* options. This will more accurately match how you deploy later.

Click *OK* to save and close the configuration. Select the configuration in the main PyCharm window and click the play button next to it to run the server.

Now that you have a configuration for `flask run`, you can copy that configuration and change the *Parameters* argument to run a different CLI command.

1.18 Development Server

Flask provides a `run` command to run the application with a development server. In debug mode, this server provides an interactive debugger and will reload when code is changed.

Warning: Do not use the development server when deploying to production. It is intended for use only during local development. It is not designed to be particularly efficient, stable, or secure.

See *Deploying to Production* for deployment options.

1.18.1 Command Line

The `flask run` CLI command is the recommended way to run the development server. Use the `--app` option to point to your application, and the `--debug` option to enable debug mode.

```
$ flask --app hello run --debug
```

This enables debug mode, including the interactive debugger and reloader, and then starts the server on <http://localhost:5000/>. Use `flask run --help` to see the available options, and *Command Line Interface* for detailed instructions about configuring and using the CLI.

Address already in use

If another program is already using port 5000, you'll see an `OSError` when the server tries to start. It may have one of the following messages:

- `OSError: [Errno 98] Address already in use`
- `OSError: [WinError 10013] An attempt was made to access a socket in a way forbidden by its access permissions`

Either identify and stop the other program, or use `flask run --port 5001` to pick a different port.

You can use `netstat` or `lsof` to identify what process id is using a port, then use other operating system tools stop that process. The following example shows that process id 6847 is using port 5000.

`netstat` (Linux)

```
$ netstat -nlp | grep 5000
tcp 0 0 127.0.0.1:5000 0.0.0.0:* LISTEN 6847/python
```

`lsof` (macOS / Linux)

```
$ lsof -P -i :5000
Python 6847 IPv4 TCP localhost:5000 (LISTEN)
```

`netstat` (Windows)

```
> netstat -ano | findstr 5000
TCP 127.0.0.1:5000 0.0.0.0:0 LISTENING 6847
```

macOS Monterey and later automatically starts a service that uses port 5000. To disable the service, go to System Preferences, Sharing, and disable “AirPlay Receiver”.

Deferred Errors on Reload

When using the `flask run` command with the reloader, the server will continue to run even if you introduce syntax errors or other initialization errors into the code. Accessing the site will show the interactive debugger for the error, rather than crashing the server.

If a syntax error is already present when calling `flask run`, it will fail immediately and show the traceback rather than waiting until the site is accessed. This is intended to make errors more visible initially while still allowing the server to handle errors on reload.

1.18.2 In Code

The development server can also be started from Python with the `Flask.run()` method. This method takes arguments similar to the CLI options to control the server. The main difference from the CLI command is that the server will crash if there are errors when reloading. `debug=True` can be passed to enable debug mode.

Place the call in a main block, otherwise it will interfere when trying to import and run the application with a production server later.

```
if __name__ == "__main__":
    app.run(debug=True)
```

```
$ python hello.py
```

1.19 Working with the Shell

New in version 0.3.

One of the reasons everybody loves Python is the interactive shell. It basically allows you to execute Python commands in real time and immediately get results back. Flask itself does not come with an interactive shell, because it does not require any specific setup upfront, just import your application and start playing around.

There are however some handy helpers to make playing around in the shell a more pleasant experience. The main issue with interactive console sessions is that you’re not triggering a request like a browser does which means that `g`, `request` and others are not available. But the code you want to test might depend on them, so what can you do?

This is where some helper functions come in handy. Keep in mind however that these functions are not only there for interactive shell usage, but also for unit testing and other situations that require a faked request context.

Generally it’s recommended that you read *The Request Context* first.

1.19.1 Command Line Interface

Starting with Flask 0.11 the recommended way to work with the shell is the `flask shell` command which does a lot of this automatically for you. For instance the shell is automatically initialized with a loaded application context.

For more information see *Command Line Interface*.

1.19.2 Creating a Request Context

The easiest way to create a proper request context from the shell is by using the `test_request_context` method which creates us a *RequestContext*:

```
>>> ctx = app.test_request_context()
```

Normally you would use the `with` statement to make this request object active, but in the shell it's easier to use the `push()` and `pop()` methods by hand:

```
>>> ctx.push()
```

From that point onwards you can work with the request object until you call `pop`:

```
>>> ctx.pop()
```

1.19.3 Firing Before/After Request

By just creating a request context, you still don't have run the code that is normally run before a request. This might result in your database being unavailable if you are connecting to the database in a before-request callback or the current user not being stored on the `g` object etc.

This however can easily be done yourself. Just call `preprocess_request()`:

```
>>> ctx = app.test_request_context()
>>> ctx.push()
>>> app.preprocess_request()
```

Keep in mind that the `preprocess_request()` function might return a response object, in that case just ignore it.

To shutdown a request, you need to trick a bit before the after request functions (triggered by `process_response()`) operate on a response object:

```
>>> app.process_response(app.response_class())
<Response 0 bytes [200 OK]>
>>> ctx.pop()
```

The functions registered as `teardown_request()` are automatically called when the context is popped. So this is the perfect place to automatically tear down resources that were needed by the request context (such as database connections).

1.19.4 Further Improving the Shell Experience

If you like the idea of experimenting in a shell, create yourself a module with stuff you want to star import into your interactive session. There you could also define some more helper methods for common things such as initializing the database, dropping tables etc.

Just put them into a module (like *shelltools*) and import from there:

```
>>> from shelltools import *
```

1.20 Patterns for Flask

Certain features and interactions are common enough that you will find them in most web applications. For example, many applications use a relational database and user authentication. They will open a database connection at the beginning of the request and get the information for the logged in user. At the end of the request, the database connection is closed.

These types of patterns may be a bit outside the scope of Flask itself, but Flask makes it easy to implement them. Some common patterns are collected in the following pages.

1.20.1 Large Applications as Packages

Imagine a simple flask application structure that looks like this:

```
/yourapplication
  yourapplication.py
  /static
    style.css
  /templates
    layout.html
    index.html
    login.html
    ...
```

While this is fine for small applications, for larger applications it's a good idea to use a package instead of a module. The *Tutorial* is structured to use the package pattern, see the [example code](#).

Simple Packages

To convert that into a larger one, just create a new folder `yourapplication` inside the existing one and move everything below it. Then rename `yourapplication.py` to `__init__.py`. (Make sure to delete all `.pyc` files first, otherwise things would most likely break)

You should then end up with something like that:

```
/yourapplication
  /yourapplication
    __init__.py
  /static
    style.css
  /templates
```

(continues on next page)

(continued from previous page)

```
layout.html
index.html
login.html
...
```

But how do you run your application now? The naive `python yourapplication/__init__.py` will not work. Let's just say that Python does not want modules in packages to be the startup file. But that is not a big problem, just add a new file called `pyproject.toml` next to the inner `yourapplication` folder with the following contents:

```
[project]
name = "yourapplication"
dependencies = [
    "flask",
]

[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

Install your application so it is importable:

```
$ pip install -e .
```

To use the `flask` command and run your application you need to set the `--app` option that tells Flask where to find the application instance:

```
$ flask --app yourapplication run
```

What did we gain from this? Now we can restructure the application a bit into multiple modules. The only thing you have to remember is the following quick checklist:

1. the *Flask* application object creation has to be in the `__init__.py` file. That way each module can import it safely and the `__name__` variable will resolve to the correct package.
2. all the view functions (the ones with a `route()` decorator on top) have to be imported in the `__init__.py` file. Not the object itself, but the module it is in. Import the view module **after the application object is created**.

Here's an example `__init__.py`:

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

import yourapplication.views
```

And this is what `views.py` would look like:

```
from yourapplication import app

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

You should then end up with something like that:

```
/yourapplication
  pyproject.toml
  /yourapplication
    __init__.py
    views.py
    /static
      style.css
    /templates
      layout.html
      index.html
      login.html
      ...
```

Circular Imports

Every Python programmer hates them, and yet we just added some: circular imports (That's when two modules depend on each other. In this case `views.py` depends on `__init__.py`). Be advised that this is a bad idea in general but here it is actually fine. The reason for this is that we are not actually using the views in `__init__.py` and just ensuring the module is imported and we are doing that at the bottom of the file.

Working with Blueprints

If you have larger applications it's recommended to divide them into smaller groups where each group is implemented with the help of a blueprint. For a gentle introduction into this topic refer to the *Modular Applications with Blueprints* chapter of the documentation.

1.20.2 Application Factories

If you are already using packages and blueprints for your application (*Modular Applications with Blueprints*) there are a couple of really nice ways to further improve the experience. A common pattern is creating the application object when the blueprint is imported. But if you move the creation of this object into a function, you can then create multiple instances of this app later.

So why would you want to do this?

1. Testing. You can have instances of the application with different settings to test every case.
2. Multiple instances. Imagine you want to run different versions of the same application. Of course you could have multiple instances with different configs set up in your webserver, but if you use factories, you can have multiple instances of the same application running in the same application process which can be handy.

So how would you then actually implement that?

Basic Factories

The idea is to set up the application in a function. Like this:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    from yourapplication.model import db
    db.init_app(app)

    from yourapplication.views.admin import admin
    from yourapplication.views.frontend import frontend
    app.register_blueprint(admin)
    app.register_blueprint(frontend)

    return app
```

The downside is that you cannot use the application object in the blueprints at import time. You can however use it from within a request. How do you get access to the application with the config? Use `current_app`:

```
from flask import current_app, Blueprint, render_template
admin = Blueprint('admin', __name__, url_prefix='/admin')

@admin.route('/')
def index():
    return render_template(current_app.config['INDEX_TEMPLATE'])
```

Here we look up the name of a template in the config.

Factories & Extensions

It's preferable to create your extensions and app factories so that the extension object does not initially get bound to the application.

Using `Flask-SQLAlchemy`, as an example, you should not do something along those lines:

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)

    db = SQLAlchemy(app)
```

But, rather, in `model.py` (or equivalent):

```
db = SQLAlchemy()
```

and in your `application.py` (or equivalent):

```
def create_app(config_filename):
    app = Flask(__name__)
    app.config.from_pyfile(config_filename)
```

(continues on next page)

(continued from previous page)

```
from yourapplication.model import db
db.init_app(app)
```

Using this design pattern, no application-specific state is stored on the extension object, so one extension object can be used for multiple apps. For more information about the design of extensions refer to *Flask Extension Development*.

Using Applications

To run such an application, you can use the **flask** command:

```
$ flask --app hello run
```

Flask will automatically detect the factory if it is named `create_app` or `make_app` in `hello`. You can also pass arguments to the factory like this:

```
$ flask --app hello:create_app(local_auth=True) run
```

Then the `create_app` factory in `myapp` is called with the keyword argument `local_auth=True`. See *Command Line Interface* for more detail.

Factory Improvements

The factory function above is not very clever, but you can improve it. The following changes are straightforward to implement:

1. Make it possible to pass in configuration values for unit tests so that you don't have to create config files on the filesystem.
2. Call a function from a blueprint when the application is setting up so that you have a place to modify attributes of the application (like hooking in before/after request handlers etc.)
3. Add in WSGI middlewares when the application is being created if necessary.

1.20.3 Application Dispatching

Application dispatching is the process of combining multiple Flask applications on the WSGI level. You can combine not only Flask applications but any WSGI application. This would allow you to run a Django and a Flask application in the same interpreter side by side if you want. The usefulness of this depends on how the applications work internally.

The fundamental difference from *Large Applications as Packages* is that in this case you are running the same or different Flask applications that are entirely isolated from each other. They run different configurations and are dispatched on the WSGI level.

Working with this Document

Each of the techniques and examples below results in an application object that can be run with any WSGI server. For production, see *Deploying to Production*. For development, Werkzeug provides a server through `werkzeug.serving.run_simple()`:

```
from werkzeug.serving import run_simple
run_simple('localhost', 5000, application, use_reloader=True)
```

Note that `run_simple` is not intended for use in production. Use a production WSGI server. See *Deploying to Production*.

In order to use the interactive debugger, debugging must be enabled both on the application and the simple server. Here is the “hello world” example with debugging and `run_simple`:

```
from flask import Flask
from werkzeug.serving import run_simple

app = Flask(__name__)
app.debug = True

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    run_simple('localhost', 5000, app,
              use_reloader=True, use_debugger=True, use_eval_ex=True)
```

Combining Applications

If you have entirely separated applications and you want them to work next to each other in the same Python interpreter process you can take advantage of the `werkzeug.wsgi.DispatcherMiddleware`. The idea here is that each Flask application is a valid WSGI application and they are combined by the dispatcher middleware into a larger one that is dispatched based on prefix.

For example you could have your main application run on `/` and your backend interface on `/backend`:

```
from werkzeug.middleware.dispatcher import DispatcherMiddleware
from frontend_app import application as frontend
from backend_app import application as backend

application = DispatcherMiddleware(frontend, {
    '/backend': backend
})
```

Dispatch by Subdomain

Sometimes you might want to use multiple instances of the same application with different configurations. Assuming the application is created inside a function and you can call that function to instantiate it, that is really easy to implement. In order to develop your application to support creating new instances in functions have a look at the *Application Factories* pattern.

A very common example would be creating applications per subdomain. For instance you configure your webserver to dispatch all requests for all subdomains to your application and you then use the subdomain information to create user-specific instances. Once you have your server set up to listen on all subdomains you can use a very simple WSGI application to do the dynamic application creation.

The perfect level for abstraction in that regard is the WSGI layer. You write your own WSGI application that looks at the request that comes and delegates it to your Flask application. If that application does not exist yet, it is dynamically created and remembered:

```
from threading import Lock

class SubdomainDispatcher:

    def __init__(self, domain, create_app):
        self.domain = domain
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, host):
        host = host.split(':')[0]
        assert host.endswith(self.domain), 'Configuration error'
        subdomain = host[:-len(self.domain)].rstrip('.')
        with self.lock:
            app = self.instances.get(subdomain)
            if app is None:
                app = self.create_app(subdomain)
                self.instances[subdomain] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(environ['HTTP_HOST'])
        return app(environ, start_response)
```

This dispatcher can then be used like this:

```
from myapplication import create_app, get_user_for_subdomain
from werkzeug.exceptions import NotFound

def make_app(subdomain):
    user = get_user_for_subdomain(subdomain)
    if user is None:
        # if there is no user for that subdomain we still have
        # to return a WSGI application that handles that request.
        # We can then just return the NotFound() exception as
        # application which will render a default 404 page.
        # You might also redirect the user to the main page then
        return NotFound()
```

(continues on next page)

(continued from previous page)

```
# otherwise create the application for the specific user
return create_app(user)

application = SubdomainDispatcher('example.com', make_app)
```

Dispatch by Path

Dispatching by a path on the URL is very similar. Instead of looking at the Host header to figure out the subdomain one simply looks at the request path up to the first slash:

```
from threading import Lock
from werkzeug.wsgi import pop_path_info, peek_path_info

class PathDispatcher:

    def __init__(self, default_app, create_app):
        self.default_app = default_app
        self.create_app = create_app
        self.lock = Lock()
        self.instances = {}

    def get_application(self, prefix):
        with self.lock:
            app = self.instances.get(prefix)
            if app is None:
                app = self.create_app(prefix)
                if app is not None:
                    self.instances[prefix] = app
            return app

    def __call__(self, environ, start_response):
        app = self.get_application(peek_path_info(environ))
        if app is not None:
            pop_path_info(environ)
        else:
            app = self.default_app
        return app(environ, start_response)
```

The big difference between this and the subdomain one is that this one falls back to another application if the creator function returns None:

```
from myapplication import create_app, default_app, get_user_for_prefix

def make_app(prefix):
    user = get_user_for_prefix(prefix)
    if user is not None:
        return create_app(user)

application = PathDispatcher(default_app, make_app)
```

1.20.4 Using URL Processors

New in version 0.7.

Flask 0.7 introduces the concept of URL processors. The idea is that you might have a bunch of resources with common parts in the URL that you don't always explicitly want to provide. For instance you might have a bunch of URLs that have the language code in it but you don't want to have to handle it in every single function yourself.

URL processors are especially helpful when combined with blueprints. We will handle both application specific URL processors here as well as blueprint specifics.

Internationalized Application URLs

Consider an application like this:

```
from flask import Flask, g

app = Flask(__name__)

@app.route('/<lang_code>/')
def index(lang_code):
    g.lang_code = lang_code
    ...

@app.route('/<lang_code>/about')
def about(lang_code):
    g.lang_code = lang_code
    ...
```

This is an awful lot of repetition as you have to handle the language code setting on the `g` object yourself in every single function. Sure, a decorator could be used to simplify this, but if you want to generate URLs from one function to another you would have to still provide the language code explicitly which can be annoying.

For the latter, this is where `url_defaults()` functions come in. They can automatically inject values into a call to `url_for()`. The code below checks if the language code is not yet in the dictionary of URL values and if the endpoint wants a value named 'lang_code':

```
@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint_expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code
```

The method `is_endpoint_expecting()` of the URL map can be used to figure out if it would make sense to provide a language code for the given endpoint.

The reverse of that function are `url_value_preprocessor()`s. They are executed right after the request was matched and can execute code based on the URL values. The idea is that they pull information out of the values dictionary and put it somewhere else:

```
@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)
```

That way you no longer have to do the *lang_code* assignment to *g* in every function. You can further improve that by writing your own decorator that prefixes URLs with the language code, but the more beautiful solution is using a blueprint. Once the 'lang_code' is popped from the values dictionary and it will no longer be forwarded to the view function reducing the code to this:

```

from flask import Flask, g

app = Flask(__name__)

@app.url_defaults
def add_language_code(endpoint, values):
    if 'lang_code' in values or not g.lang_code:
        return
    if app.url_map.is_endpoint expecting(endpoint, 'lang_code'):
        values['lang_code'] = g.lang_code

@app.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code', None)

@app.route('/<lang_code>/')
def index():
    ...

@app.route('/<lang_code>/about')
def about():
    ...

```

Internationalized Blueprint URLs

Because blueprints can automatically prefix all URLs with a common string it's easy to automatically do that for every function. Furthermore blueprints can have per-blueprint URL processors which removes a whole lot of logic from the *url_defaults()* function because it no longer has to check if the URL is really interested in a 'lang_code' parameter:

```

from flask import Blueprint, g

bp = Blueprint('frontend', __name__, url_prefix='/<lang_code>')

@bp.url_defaults
def add_language_code(endpoint, values):
    values.setdefault('lang_code', g.lang_code)

@bp.url_value_preprocessor
def pull_lang_code(endpoint, values):
    g.lang_code = values.pop('lang_code')

@bp.route('/')
def index():
    ...

@bp.route('/about')

```

(continues on next page)

(continued from previous page)

```
def about():  
    ...
```

1.20.5 Using SQLite 3 with Flask

In Flask you can easily implement the opening of database connections on demand and closing them when the context dies (usually at the end of the request).

Here is a simple example of how you can use SQLite 3 with Flask:

```
import sqlite3  
from flask import g  
  
DATABASE = '/path/to/database.db'  
  
def get_db():  
    db = getattr(g, '_database', None)  
    if db is None:  
        db = g._database = sqlite3.connect(DATABASE)  
    return db  
  
@app.teardown_appcontext  
def close_connection(exception):  
    db = getattr(g, '_database', None)  
    if db is not None:  
        db.close()
```

Now, to use the database, the application must either have an active application context (which is always true if there is a request in flight) or create an application context itself. At that point the `get_db` function can be used to get the current database connection. Whenever the context is destroyed the database connection will be terminated.

Example:

```
@app.route('/')  
def index():  
    cur = get_db().cursor()  
    ...
```

Note: Please keep in mind that the `teardown_request` and `appcontext` functions are always executed, even if a before-request handler failed or was never executed. Because of this we have to make sure here that the database is there before we close it.

Connect on Demand

The upside of this approach (connecting on first use) is that this will only open the connection if truly necessary. If you want to use this code outside a request context you can use it in a Python shell by opening the application context by hand:

```
with app.app_context():
    # now you can use get_db()
```

Easy Querying

Now in each request handling function you can access `get_db()` to get the current open database connection. To simplify working with SQLite, a row factory function is useful. It is executed for every result returned from the database to convert the result. For instance, in order to get dictionaries instead of tuples, this could be inserted into the `get_db` function we created above:

```
def make_dicts(cursor, row):
    return dict((cursor.description[idx][0], value)
                for idx, value in enumerate(row))

db.row_factory = make_dicts
```

This will make the `sqlite3` module return dicts for this database connection, which are much nicer to deal with. Even more simply, we could place this in `get_db` instead:

```
db.row_factory = sqlite3.Row
```

This would use `Row` objects rather than dicts to return the results of queries. These are `namedtuple`s, so we can access them either by index or by key. For example, assuming we have a `sqlite3.Row` called `r` for the rows `id`, `FirstName`, `LastName`, and `MiddleInitial`:

```
>>> # You can get values based on the row's name
>>> r['FirstName']
John
>>> # Or, you can get them based on index
>>> r[1]
John
# Row objects are also iterable:
>>> for value in r:
...     print(value)
1
John
Doe
M
```

Additionally, it is a good idea to provide a query function that combines getting the cursor, executing and fetching the results:

```
def query_db(query, args=(), one=False):
    cur = get_db().execute(query, args)
    rv = cur.fetchall()
    cur.close()
    return (rv[0] if rv else None) if one else rv
```

This handy little function, in combination with a row factory, makes working with the database much more pleasant than it is by just using the raw cursor and connection objects.

Here is how you can use it:

```
for user in query_db('select * from users'):
    print(user['username'], 'has the id', user['user_id'])
```

Or if you just want a single result:

```
user = query_db('select * from users where username = ?',
               [the_username], one=True)
if user is None:
    print('No such user')
else:
    print(the_username, 'has the id', user['user_id'])
```

To pass variable parts to the SQL statement, use a question mark in the statement and pass in the arguments as a list. Never directly add them to the SQL statement with string formatting because this makes it possible to attack the application using [SQL Injections](#).

Initial Schemas

Relational databases need schemas, so applications often ship a *schema.sql* file that creates the database. It's a good idea to provide a function that creates the database based on that schema. This function can do that for you:

```
def init_db():
    with app.app_context():
        db = get_db()
        with app.open_resource('schema.sql', mode='r') as f:
            db.cursor().executescript(f.read())
        db.commit()
```

You can then create such a database from the Python shell:

```
>>> from yourapplication import init_db
>>> init_db()
```

1.20.6 SQLAlchemy in Flask

Many people prefer [SQLAlchemy](#) for database access. In this case it's encouraged to use a package instead of a module for your flask application and drop the models into a separate module (*Large Applications as Packages*). While that is not necessary, it makes a lot of sense.

There are four very common ways to use SQLAlchemy. I will outline each of them here:

Flask-SQLAlchemy Extension

Because SQLAlchemy is a common database abstraction layer and object relational mapper that requires a little bit of configuration effort, there is a Flask extension that handles that for you. This is recommended if you want to get started quickly.

You can download `Flask-SQLAlchemy` from `PyPI`.

Declarative

The declarative extension in SQLAlchemy is the most recent method of using SQLAlchemy. It allows you to define tables and models in one go, similar to how Django works. In addition to the following text I recommend the official documentation on the `declarative` extension.

Here's the example `database.py` module for your application:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///tmp/test.db')
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

Base = declarative_base()
Base.query = db_session.query_property()

def init_db():
    # import all modules here that might define models so that
    # they will be registered properly on the metadata. Otherwise
    # you will have to import them first before calling init_db()
    import yourapplication.models
    Base.metadata.create_all(bind=engine)
```

To define your models, just subclass the `Base` class that was created by the code above. If you are wondering why we don't have to care about threads here (like we did in the `SQLite3` example above with the `g` object): that's because SQLAlchemy does that for us already with the `scoped_session`.

To use SQLAlchemy in a declarative way with your application, you just have to put the following code into your application module. Flask will automatically remove database sessions at the end of the request or when the application shuts down:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example model (put this into `models.py`, e.g.):

```
from sqlalchemy import Column, Integer, String
from yourapplication.database import Base

class User(Base):
```

(continues on next page)

(continued from previous page)

```

__tablename__ = 'users'
id = Column(Integer, primary_key=True)
name = Column(String(50), unique=True)
email = Column(String(120), unique=True)

def __init__(self, name=None, email=None):
    self.name = name
    self.email = email

def __repr__(self):
    return f'<User {self.name!r}>'

```

To create the database you can use the `init_db` function:

```

>>> from yourapplication.database import init_db
>>> init_db()

```

You can insert entries into the database like this:

```

>>> from yourapplication.database import db_session
>>> from yourapplication.models import User
>>> u = User('admin', 'admin@localhost')
>>> db_session.add(u)
>>> db_session.commit()

```

Querying is simple as well:

```

>>> User.query.all()
[<User 'admin'>]
>>> User.query.filter(User.name == 'admin').first()
<User 'admin'>

```

Manual Object Relational Mapping

Manual object relational mapping has a few upsides and a few downsides versus the declarative approach from above. The main difference is that you define tables and classes separately and map them together. It's more flexible but a little more to type. In general it works like the declarative approach, so make sure to also split up your application into multiple modules in a package.

Here is an example `database.py` module for your application:

```

from sqlalchemy import create_engine, MetaData
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine('sqlite:///tmp/test.db')
metadata = MetaData()
db_session = scoped_session(sessionmaker(autocommit=False,
                                         autoflush=False,
                                         bind=engine))

def init_db():
    metadata.create_all(bind=engine)

```

As in the declarative approach, you need to close the session after each request or application context shutdown. Put this into your application module:

```
from yourapplication.database import db_session

@app.teardown_appcontext
def shutdown_session(exception=None):
    db_session.remove()
```

Here is an example table and model (put this into `models.py`):

```
from sqlalchemy import Table, Column, Integer, String
from sqlalchemy.orm import mapper
from yourapplication.database import metadata, db_session

class User(object):
    query = db_session.query_property()

    def __init__(self, name=None, email=None):
        self.name = name
        self.email = email

    def __repr__(self):
        return f'<User {self.name!r}>'

users = Table('users', metadata,
              Column('id', Integer, primary_key=True),
              Column('name', String(50), unique=True),
              Column('email', String(120), unique=True)
)
mapper(User, users)
```

Querying and inserting works exactly the same as in the example above.

SQL Abstraction Layer

If you just want to use the database system (and SQL) abstraction layer you basically only need the engine:

```
from sqlalchemy import create_engine, MetaData, Table

engine = create_engine('sqlite:///tmp/test.db')
metadata = MetaData(bind=engine)
```

Then you can either declare the tables in your code like in the examples above, or automatically load them:

```
from sqlalchemy import Table

users = Table('users', metadata, autoload=True)
```

To insert data you can use the `insert` method. We have to get a connection first so that we can use a transaction:

```
>>> con = engine.connect()
>>> con.execute(users.insert(), name='admin', email='admin@localhost')
```

SQLAlchemy will automatically commit for us.

To query your database, you use the engine directly or use a connection:

```
>>> users.select(users.c.id == 1).execute().first()
(1, 'admin', 'admin@localhost')
```

These results are also dict-like tuples:

```
>>> r = users.select(users.c.id == 1).execute().first()
>>> r['name']
'admin'
```

You can also pass strings of SQL statements to the `execute()` method:

```
>>> engine.execute('select * from users where id = :1', [1]).first()
(1, 'admin', 'admin@localhost')
```

For more information about SQLAlchemy, head over to the [website](#).

1.20.7 Uploading Files

Ah yes, the good old problem of file uploads. The basic idea of file uploads is actually quite simple. It basically works like this:

1. A `<form>` tag is marked with `enctype=multipart/form-data` and an `<input type=file>` is placed in that form.
2. The application accesses the file from the `files` dictionary on the request object.
3. use the `save()` method of the file to save the file permanently somewhere on the filesystem.

A Gentle Introduction

Let's start with a very basic application that uploads a file to a specific upload folder and displays a file to the user. Let's look at the bootstrapping code for our application:

```
import os
from flask import Flask, flash, request, redirect, url_for
from werkzeug.utils import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'}

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

So first we need a couple of imports. Most should be straightforward, the `werkzeug.secure_filename()` is explained a little bit later. The `UPLOAD_FOLDER` is where we will store the uploaded files and the `ALLOWED_EXTENSIONS` is the set of allowed file extensions.

Why do we limit the extensions that are allowed? You probably don't want your users to be able to upload everything there if the server is directly sending out the data to the client. That way you can make sure that users are not able to upload HTML files that would cause XSS problems (see *Cross-Site Scripting (XSS)*). Also make sure to disallow `.php` files if the server executes them, but who has PHP installed on their server, right? :)

Next the functions that check if an extension is valid and that uploads the file and redirects the user to the URL for the uploaded file:

```
def allowed_file(filename):
    return '.' in filename and \
        filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        # check if the post request has the file part
        if 'file' not in request.files:
            flash('No file part')
            return redirect(request.url)
        file = request.files['file']
        # If the user does not select a file, the browser submits an
        # empty file without a filename.
        if file.filename == '':
            flash('No selected file')
            return redirect(request.url)
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('download_file', name=filename))

    return '''
<!doctype html>
<title>Upload new File</title>
<h1>Upload new File</h1>
<form method=post enctype=multipart/form-data>
  <input type=file name=file>
  <input type=submit value=Upload>
</form>
'''
```

So what does that `secure_filename()` function actually do? Now the problem is that there is that principle called “never trust user input”. This is also true for the filename of an uploaded file. All submitted form data can be forged, and filenames can be dangerous. For the moment just remember: always use that function to secure a filename before storing it directly on the filesystem.

Information for the Pros

So you’re interested in what that `secure_filename()` function does and what the problem is if you’re not using it? So just imagine someone would send the following information as *filename* to your application:

```
filename = "../../../home/username/.bashrc"
```

Assuming the number of `../` is correct and you would join this with the `UPLOAD_FOLDER` the user might have the ability to modify a file on the server’s filesystem he or she should not modify. This does require some knowledge about how the application looks like, but trust me, hackers are patient :)

Now let’s look how that function works:

```
>>> secure_filename('../../../home/username/.bashrc')
'home_username_.bashrc'
```

We want to be able to serve the uploaded files so they can be downloaded by users. We'll define a `download_file` view to serve files in the upload folder by name. `url_for("download_file", name=name)` generates download URLs.

```
from flask import send_from_directory

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

If you're using middleware or the HTTP server to serve files, you can register the `download_file` endpoint as `build_only` so `url_for` will work without a view function.

```
app.add_url_rule(
    "/uploads/<name>", endpoint="download_file", build_only=True
)
```

Improving Uploads

New in version 0.6.

So how exactly does Flask handle uploads? Well it will store them in the webserver's memory if the files are reasonably small, otherwise in a temporary location (as returned by `tempfile.gettempdir()`). But how do you specify the maximum file size after which an upload is aborted? By default Flask will happily accept file uploads with an unlimited amount of memory, but you can limit that by setting the `MAX_CONTENT_LENGTH` config key:

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1000 * 1000
```

The code above will limit the maximum allowed payload to 16 megabytes. If a larger file is transmitted, Flask will raise a `RequestEntityTooLarge` exception.

Connection Reset Issue

When using the local development server, you may get a connection reset error instead of a 413 response. You will get the correct status response when running the app with a production WSGI server.

This feature was added in Flask 0.6 but can be achieved in older versions as well by subclassing the request object. For more information on that consult the Werkzeug documentation on file handling.

Upload Progress Bars

A while ago many developers had the idea to read the incoming file in small chunks and store the upload progress in the database to be able to poll the progress with JavaScript from the client. The client asks the server every 5 seconds how much it has transmitted, but this is something it should already know.

An Easier Solution

Now there are better solutions that work faster and are more reliable. There are JavaScript libraries like [jQuery](#) that have form plugins to ease the construction of progress bar.

Because the common pattern for file uploads exists almost unchanged in all applications dealing with uploads, there are also some Flask extensions that implement a full fledged upload mechanism that allows controlling which file extensions are allowed to be uploaded.

1.20.8 Caching

When your application runs slow, throw some caches in. Well, at least it's the easiest way to speed up things. What does a cache do? Say you have a function that takes some time to complete but the results would still be good enough if they were 5 minutes old. So then the idea is that you actually put the result of that calculation into a cache for some time.

Flask itself does not provide caching for you, but [Flask-Caching](#), an extension for Flask does. Flask-Caching supports various backends, and it is even possible to develop your own caching backend.

1.20.9 View Decorators

Python has a really interesting feature called function decorators. This allows some really neat things for web applications. Because each view in Flask is a function, decorators can be used to inject additional functionality to one or more functions. The `route()` decorator is the one you probably used already. But there are use cases for implementing your own decorator. For instance, imagine you have a view that should only be used by people that are logged in. If a user goes to the site and is not logged in, they should be redirected to the login page. This is a good example of a use case where a decorator is an excellent solution.

Login Required Decorator

So let's implement such a decorator. A decorator is a function that wraps and replaces another function. Since the original function is replaced, you need to remember to copy the original function's information to the new function. Use `functools.wraps()` to handle this for you.

This example assumes that the login page is called 'login' and that the current user is stored in `g.user` and is `None` if there is no-one logged in.

```
from functools import wraps
from flask import g, request, redirect, url_for

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if g.user is None:
            return redirect(url_for('login', next=request.url))
        return f(*args, **kwargs)
    return decorated_function
```

To use the decorator, apply it as innermost decorator to a view function. When applying further decorators, always remember that the `route()` decorator is the outermost.

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

Note: The next value will exist in `request.args` after a GET request for the login page. You'll have to pass it along when sending the POST request from the login form. You can do this with a hidden input tag, then retrieve it from `request.form` when logging the user in.

```
<input type="hidden" value="{{ request.args.get('next', '') }}" />
```

Caching Decorator

Imagine you have a view function that does an expensive calculation and because of that you would like to cache the generated results for a certain amount of time. A decorator would be nice for that. We're assuming you have set up a cache like mentioned in [Caching](#).

Here is an example cache function. It generates the cache key from a specific prefix (actually a format string) and the current path of the request. Notice that we are using a function that first creates the decorator that then decorates the function. Sounds awful? Unfortunately it is a little bit more complex, but the code should still be straightforward to read.

The decorated function will then work as follows

1. get the unique cache key for the current request based on the current path.
2. get the value for that key from the cache. If the cache returned something we will return that value.
3. otherwise the original function is called and the return value is stored in the cache for the timeout provided (by default 5 minutes).

Here the code:

```
from functools import wraps
from flask import request

def cached(timeout=5 * 60, key='view/{}'):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            cache_key = key.format(request.path)
            rv = cache.get(cache_key)
            if rv is not None:
                return rv
            rv = f(*args, **kwargs)
            cache.set(cache_key, rv, timeout=timeout)
            return rv
        return decorated_function
    return decorator
```

Notice that this assumes an instantiated cache object is available, see [Caching](#).

Templating Decorator

A common pattern invented by the TurboGears guys a while back is a templating decorator. The idea of that decorator is that you return a dictionary with the values passed to the template from the view function and the template is automatically rendered. With that, the following three examples do exactly the same:

```
@app.route('/')
def index():
    return render_template('index.html', value=42)

@app.route('/')
@templated('index.html')
def index():
    return dict(value=42)

@app.route('/')
@templated()
def index():
    return dict(value=42)
```

As you can see, if no template name is provided it will use the endpoint of the URL map with dots converted to slashes + '.html'. Otherwise the provided template name is used. When the decorated function returns, the dictionary returned is passed to the template rendering function. If `None` is returned, an empty dictionary is assumed, if something else than a dictionary is returned we return it from the function unchanged. That way you can still use the `redirect` function or return simple strings.

Here is the code for that decorator:

```
from functools import wraps
from flask import request, render_template

def templated(template=None):
    def decorator(f):
        @wraps(f)
        def decorated_function(*args, **kwargs):
            template_name = template
            if template_name is None:
                template_name = f"{request.endpoint.replace('.', '/')}.html"
            ctx = f(*args, **kwargs)
            if ctx is None:
                ctx = {}
            elif not isinstance(ctx, dict):
                return ctx
            return render_template(template_name, **ctx)
        return decorated_function
    return decorator
```

Endpoint Decorator

When you want to use the werkzeug routing system for more flexibility you need to map the endpoint as defined in the [Rule](#) to a view function. This is possible with this decorator. For example:

```
from flask import Flask
from werkzeug.routing import Rule

app = Flask(__name__)
app.url_map.add(Rule('/', endpoint='index'))

@app.endpoint('index')
def my_index():
    return "Hello world"
```

1.20.10 Form Validation with WTForms

When you have to work with form data submitted by a browser view, code quickly becomes very hard to read. There are libraries out there designed to make this process easier to manage. One of them is [WTForms](#) which we will handle here. If you find yourself in the situation of having many forms, you might want to give it a try.

When you are working with WTForms you have to define your forms as classes first. I recommend breaking up the application into multiple modules (*Large Applications as Packages*) for that and adding a separate module for the forms.

Getting the most out of WTForms with an Extension

The [Flask-WTF](#) extension expands on this pattern and adds a few little helpers that make working with forms and Flask more fun. You can get it from [PyPI](#).

The Forms

This is an example form for a typical registration page:

```
from wtforms import Form, BooleanField, StringField, PasswordField, validators

class RegistrationForm(Form):
    username = StringField('Username', [validators.Length(min=4, max=25)])
    email = StringField('Email Address', [validators.Length(min=6, max=35)])
    password = PasswordField('New Password', [
        validators.DataRequired(),
        validators.EqualTo('confirm', message='Passwords must match')
    ])
    confirm = PasswordField('Repeat Password')
    accept_tos = BooleanField('I accept the TOS', [validators.DataRequired()])
```

In the View

In the view function, the usage of this form looks like this:

```
@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm(request.form)
    if request.method == 'POST' and form.validate():
        user = User(form.username.data, form.email.data,
                    form.password.data)
        db_session.add(user)
        flash('Thanks for registering')
        return redirect(url_for('login'))
    return render_template('register.html', form=form)
```

Notice we're implying that the view is using SQLAlchemy here (*SQLAlchemy in Flask*), but that's not a requirement, of course. Adapt the code as necessary.

Things to remember:

1. create the form from the request `form` value if the data is submitted via the HTTP POST method and `args` if the data is submitted as GET.
2. to validate the data, call the `validate()` method, which will return `True` if the data validates, `False` otherwise.
3. to access individual values from the form, access `form.<NAME>.data`.

Forms in Templates

Now to the template side. When you pass the form to the templates, you can easily render them there. Look at the following example template to see how easy this is. WTForms does half the form generation for us already. To make it even nicer, we can write a macro that renders a field with label and a list of errors if there are any.

Here's an example `_formhelpers.html` template with such a macro:

```
{% macro render_field(field) %}
<dt>{{ field.label }}
<dd>{{ field(**kwargs)|safe }}
{% if field.errors %}
<ul class=errors>
  {% for error in field.errors %}
    <li>{{ error }}</li>
  {% endfor %}
</ul>
{% endif %}
</dd>
{% endmacro %}
```

This macro accepts a couple of keyword arguments that are forwarded to WTForm's field function, which renders the field for us. The keyword arguments will be inserted as HTML attributes. So, for example, you can call `render_field(form.username, class='username')` to add a class to the input element. Note that WTForms returns standard Python strings, so we have to tell Jinja2 that this data is already HTML-escaped with the `|safe` filter.

Here is the `register.html` template for the function we used above, which takes advantage of the `_formhelpers.html` template:

```
{% from "_formhelpers.html" import render_field %}
<form method=post>
  <dl>
    {{ render_field(form.username) }}
    {{ render_field(form.email) }}
    {{ render_field(form.password) }}
    {{ render_field(form.confirm) }}
    {{ render_field(form.accept_tos) }}
  </dl>
  <p><input type=submit value=Register>
</form>
```

For more information about WTForms, head over to the [WTForms website](#).

1.20.11 Template Inheritance

The most powerful part of Jinja is template inheritance. Template inheritance allows you to build a base “skeleton” template that contains all the common elements of your site and defines **blocks** that child templates can override.

Sounds complicated but is very basic. It’s easiest to understand it by starting with an example.

Base Template

This template, which we’ll call `layout.html`, defines a simple HTML skeleton document that you might use for a simple two-column page. It’s the job of “child” templates to fill the empty blocks with content:

```
<!doctype html>
<html>
  <head>
    {% block head %}
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <title>{% block title %}{% endblock %} - My Webpage</title>
    {% endblock %}
  </head>
  <body>
    <div id="content">{% block content %}{% endblock %}</div>
    <div id="footer">
      {% block footer %}
      &copy; Copyright 2010 by <a href="http://domain.invalid/">you</a>.
      {% endblock %}
    </div>
  </body>
</html>
```

In this example, the `{% block %}` tags define four blocks that child templates can fill in. All the `block` tag does is tell the template engine that a child template may override those portions of the template.

Child Template

A child template might look like this:

```
{% extends "layout.html" %}
{% block title %}Index{% endblock %}
{% block head %}
    {{ super() }}
<style type="text/css">
    .important { color: #336699; }
</style>
{% endblock %}
{% block content %}
<h1>Index</h1>
<p class="important">
    Welcome on my awesome homepage.
{% endblock %}
```

The `{% extends %}` tag is the key here. It tells the template engine that this template “extends” another template. When the template system evaluates this template, first it locates the parent. The `extends` tag must be the first tag in the template. To render the contents of a block defined in the parent template, use `{{ super() }}`.

1.20.12 Message Flashing

Good applications and user interfaces are all about feedback. If the user does not get enough feedback they will probably end up hating the application. Flask provides a really simple way to give feedback to a user with the flashing system. The flashing system basically makes it possible to record a message at the end of a request and access it next request and only next request. This is usually combined with a layout template that does this. Note that browsers and sometimes web servers enforce a limit on cookie sizes. This means that flashing messages that are too large for session cookies causes message flashing to fail silently.

Simple Flashing

So here is a full example:

```
from flask import Flask, flash, redirect, render_template, \
    request, url_for

app = Flask(__name__)
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

@app.route('/')
def index():
    return render_template('index.html')

@app.route('/login', methods=['GET', 'POST'])
def login():
    error = None
    if request.method == 'POST':
        if request.form['username'] != 'admin' or \
            request.form['password'] != 'secret':
            error = 'Invalid credentials'
```

(continues on next page)

(continued from previous page)

```

else:
    flash('You were successfully logged in')
    return redirect(url_for('index'))
return render_template('login.html', error=error)

```

And here is the layout.html template which does the magic:

```

<!doctype html>
<title>My Application</title>
{% with messages = get_flashed_messages() %}
  {% if messages %}
    <ul class=flashes>
      {% for message in messages %}
        <li>{{ message }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
{% block body %}{% endblock %}

```

Here is the index.html template which inherits from layout.html:

```

{% extends "layout.html" %}
{% block body %}
  <h1>Overview</h1>
  <p>Do you want to <a href="{{ url_for('login') }}">log in?</a>
{% endblock %}

```

And here is the login.html template which also inherits from layout.html:

```

{% extends "layout.html" %}
{% block body %}
  <h1>Login</h1>
  {% if error %}
    <p class=error><strong>Error:</strong> {{ error }}
  {% endif %}
  <form method=post>
    <dl>
      <dt>Username:
      <dd><input type=text name=username value="{{
        request.form.username }}">
      <dt>Password:
      <dd><input type=password name=password>
    </dl>
    <p><input type=submit value=Login>
  </form>
{% endblock %}

```

Flashing With Categories

New in version 0.3.

It is also possible to provide categories when flashing a message. The default category if nothing is provided is 'message'. Alternative categories can be used to give the user better feedback. For example error messages could be displayed with a red background.

To flash a message with a different category, just use the second argument to the `flash()` function:

```
flash('Invalid password provided', 'error')
```

Inside the template you then have to tell the `get_flashed_messages()` function to also return the categories. The loop looks slightly different in that situation then:

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
      {% for category, message in messages %}
        <li class="{{ category }}">{{ message }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

This is just one example of how to render these flashed messages. One might also use the category to add a prefix such as `Error:` to the message.

Filtering Flash Messages

New in version 0.9.

Optionally you can pass a list of categories which filters the results of `get_flashed_messages()`. This is useful if you wish to render each category in a separate block.

```
{% with errors = get_flashed_messages(category_filter=["error"]) %}
{% if errors %}
<div class="alert-message block-message error">
  <a class="close" href="#">x</a>
  <ul>
    {%- for msg in errors %}
      <li>{{ msg }}</li>
    {% endfor -%}
  </ul>
</div>
{% endif %}
{% endwith %}
```

1.20.13 JavaScript, fetch, and JSON

You may want to make your HTML page dynamic, by changing data without reloading the entire page. Instead of submitting an HTML `<form>` and performing a redirect to re-render the template, you can add JavaScript that calls `fetch()` and replaces content on the page.

`fetch()` is the modern, built-in JavaScript solution to making requests from a page. You may have heard of other “AJAX” methods and libraries, such as `XMLHttpRequest()` or `jQuery`. These are no longer needed in modern browsers, although you may choose to use them or another library depending on your application’s requirements. These docs will only focus on built-in JavaScript features.

Rendering Templates

It is important to understand the difference between templates and JavaScript. Templates are rendered on the server, before the response is sent to the user’s browser. JavaScript runs in the user’s browser, after the template is rendered and sent. Therefore, it is impossible to use JavaScript to affect how the Jinja template is rendered, but it is possible to render data into the JavaScript that will run.

To provide data to JavaScript when rendering the template, use the `tojson()` filter in a `<script>` block. This will convert the data to a valid JavaScript object, and ensure that any unsafe HTML characters are rendered safely. If you do not use the `tojson` filter, you will get a `SyntaxError` in the browser console.

```
data = generate_report()
return render_template("report.html", chart_data=data)
```

```
<script>
  const chart_data = {{ chart_data|tojson }}
  chartLib.makeChart(chart_data)
</script>
```

A less common pattern is to add the data to a `data-` attribute on an HTML tag. In this case, you must use single quotes around the value, not double quotes, otherwise you will produce invalid or unsafe HTML.

```
<div data-chart='{{ chart_data|tojson }}'></div>
```

Generating URLs

The other way to get data from the server to JavaScript is to make a request for it. First, you need to know the URL to request.

The simplest way to generate URLs is to continue to use `url_for()` when rendering the template. For example:

```
const user_url = {{ url_for("user", id=current_user.id)|tojson }}
fetch(user_url).then(...)
```

However, you might need to generate a URL based on information you only know in JavaScript. As discussed above, JavaScript runs in the user’s browser, not as part of the template rendering, so you can’t use `url_for` at that point.

In this case, you need to know the “root URL” under which your application is served. In simple setups, this is `/`, but it might also be something else, like `https://example.com/myapp/`.

A simple way to tell your JavaScript code about this root is to set it as a global variable when rendering the template. Then you can use it when generating URLs from JavaScript.

```
const SCRIPT_ROOT = {{ request.script_root|tojson }}
let user_id = ... // do something to get a user id from the page
let user_url = `${SCRIPT_ROOT}/user/${user_id}`
fetch(user_url).then(...)
```

Making a Request with fetch

`fetch()` takes two arguments, a URL and an object with other options, and returns a `Promise`. We won't cover all the available options, and will only use `then()` on the promise, not other callbacks or `await` syntax. Read the linked MDN docs for more information about those features.

By default, the GET method is used. If the response contains JSON, it can be used with a `then()` callback chain.

```
const room_url = {{ url_for("room_detail", id=room.id)|tojson }}
fetch(room_url)
  .then(response => response.json())
  .then(data => {
    // data is a parsed JSON object
  })
```

To send data, use a data method such as POST, and pass the `body` option. The most common types for data are form data or JSON data.

To send form data, pass a populated `FormData` object. This uses the same format as an HTML form, and would be accessed with `request.form` in a Flask view.

```
let data = new FormData()
data.append("name": "Flask Room")
data.append("description": "Talk about Flask here.")
fetch(room_url, {
  "method": "POST",
  "body": data,
}).then(...)
```

In general, prefer sending request data as form data, as would be used when submitting an HTML form. JSON can represent more complex data, but unless you need that it's better to stick with the simpler format. When sending JSON data, the `Content-Type: application/json` header must be sent as well, otherwise Flask will return a 400 error.

```
let data = {
  "name": "Flask Room",
  "description": "Talk about Flask here.",
}
fetch(room_url, {
  "method": "POST",
  "headers": {"Content-Type": "application/json"},
  "body": JSON.stringify(data),
}).then(...)
```

Following Redirects

A response might be a redirect, for example if you logged in with JavaScript instead of a traditional HTML form, and your view returned a redirect instead of JSON. JavaScript requests do follow redirects, but they don't change the page. If you want to make the page change you can inspect the response and apply the redirect manually.

```
fetch("/login", {"body": ...}).then(
  response => {
    if (response.redirected) {
      window.location = response.url
    } else {
      showLoginError()
    }
  }
)
```

Replacing Content

A response might be new HTML, either a new section of the page to add or replace, or an entirely new page. In general, if you're returning the entire page, it would be better to handle that with a redirect as shown in the previous section. The following example shows how to replace a <div> with the HTML returned by a request.

```
<div id="geology-fact">
  {{ include "geology_fact.html" }}
</div>
<script>
  const geology_url = {{ url_for("geology_fact")|tojson }}
  const geology_div = getElementById("geology-fact")
  fetch(geology_url)
    .then(response => response.text)
    .then(text => geology_div.innerHTML = text)
</script>
```

Return JSON from Views

To return a JSON object from your API view, you can directly return a dict from the view. It will be serialized to JSON automatically.

```
@app.route("/user/<int:id>")
def user_detail(id):
    user = User.query.get_or_404(id)
    return {
        "username": User.username,
        "email": User.email,
        "picture": url_for("static", filename=f"users/{id}/profile.png"),
    }
```

If you want to return another JSON type, use the `jsonify()` function, which creates a response object with the given data serialized to JSON.

```
from flask import jsonify

@app.route("/users")
def user_list():
    users = User.query.order_by(User.name).all()
    return jsonify([u.to_json() for u in users])
```

It is usually not a good idea to return file data in a JSON response. JSON cannot represent binary data directly, so it must be base64 encoded, which can be slow, takes more bandwidth to send, and is not as easy to cache. Instead, serve files using one view, and generate a URL to the desired file to include in the JSON. Then the client can make a separate request to get the linked resource after getting the JSON.

Receiving JSON in Views

Use the `json` property of the `request` object to decode the request's body as JSON. If the body is not valid JSON, or the Content-Type header is not set to `application/json`, a 400 Bad Request error will be raised.

```
from flask import request

@app.post("/user/<int:id>")
def user_update(id):
    user = User.query.get_or_404(id)
    user.update_from_json(request.json)
    db.session.commit()
    return user.to_json()
```

1.20.14 Lazily Loading Views

Flask is usually used with the decorators. Decorators are simple and you have the URL right next to the function that is called for that specific URL. However there is a downside to this approach: it means all your code that uses decorators has to be imported upfront or Flask will never actually find your function.

This can be a problem if your application has to import quick. It might have to do that on systems like Google's App Engine or other systems. So if you suddenly notice that your application outgrows this approach you can fall back to a centralized URL mapping.

The system that enables having a central URL map is the `add_url_rule()` function. Instead of using decorators, you have a file that sets up the application with all URLs.

Converting to Centralized URL Map

Imagine the current application looks somewhat like this:

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

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

@app.route('/user/<username>')
```

(continues on next page)

(continued from previous page)

```
def user(username):
    pass
```

Then, with the centralized approach you would have one file with the views (`views.py`) but without any decorator:

```
def index():
    pass

def user(username):
    pass
```

And then a file that sets up an application which maps the functions to URLs:

```
from flask import Flask
from yourapplication import views
app = Flask(__name__)
app.add_url_rule('/', view_func=views.index)
app.add_url_rule('/user/<username>', view_func=views.user)
```

Loading Late

So far we only split up the views and the routing, but the module is still loaded upfront. The trick is to actually load the view function as needed. This can be accomplished with a helper class that behaves just like a function but internally imports the real function on first use:

```
from werkzeug.utils import import_string, cached_property

class LazyView(object):

    def __init__(self, import_name):
        self.__module__, self.__name__ = import_name.rsplit('.', 1)
        self.import_name = import_name

    @cached_property
    def view(self):
        return import_string(self.import_name)

    def __call__(self, *args, **kwargs):
        return self.view(*args, **kwargs)
```

What's important here is that `__module__` and `__name__` are properly set. This is used by Flask internally to figure out how to name the URL rules in case you don't provide a name for the rule yourself.

Then you can define your central place to combine the views like this:

```
from flask import Flask
from yourapplication.helpers import LazyView
app = Flask(__name__)
app.add_url_rule('/',
                 view_func=LazyView('yourapplication.views.index'))
app.add_url_rule('/user/<username>',
                 view_func=LazyView('yourapplication.views.user'))
```

You can further optimize this in terms of amount of keystrokes needed to write this by having a function that calls into `add_url_rule()` by prefixing a string with the project name and a dot, and by wrapping `view_func` in a `LazyView` as needed.

```
def url(import_name, url_rules=[], **options):
    view = LazyView(f"yourapplication.{import_name}")
    for url_rule in url_rules:
        app.add_url_rule(url_rule, view_func=view, **options)

# add a single route to the index view
url('views.index', ['/'])

# add two routes to a single function endpoint
url_rules = ['/user/', '/user/<username>']
url('views.user', url_rules)
```

One thing to keep in mind is that before and after request handlers have to be in a file that is imported upfront to work properly on the first request. The same goes for any kind of remaining decorator.

1.20.15 MongoDB with MongoEngine

Using a document database like MongoDB is a common alternative to relational SQL databases. This pattern shows how to use `MongoEngine`, a document mapper library, to integrate with MongoDB.

A running MongoDB server and `Flask-MongoEngine` are required.

```
pip install flask-mongoengine
```

Configuration

Basic setup can be done by defining `MONGODB_SETTINGS` on `app.config` and creating a `MongoEngine` instance.

```
from flask import Flask
from flask_mongoengine import MongoEngine

app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
    "db": "myapp",
}
db = MongoEngine(app)
```

Mapping Documents

To declare a model that represents a Mongo document, create a class that inherits from `Document` and declare each of the fields.

```
import mongoengine as me

class Movie(me.Document):
    title = me.StringField(required=True)
    year = me.IntField()
```

(continues on next page)

(continued from previous page)

```

rated = me.StringField()
director = me.StringField()
actors = me.ListField()

```

If the document has nested fields, use `EmbeddedDocument` to defined the fields of the embedded document and `EmbeddedDocumentField` to declare it on the parent document.

```

class Imdb(me.EmbeddedDocument):
    imdb_id = me.StringField()
    rating = me.DecimalField()
    votes = me.IntField()

class Movie(me.Document):
    ...
    imdb = me.EmbeddedDocumentField(Imdb)

```

Creating Data

Instantiate your document class with keyword arguments for the fields. You can also assign values to the field attributes after instantiation. Then call `doc.save()`.

```

bttf = Movie(title="Back To The Future", year=1985)
bttf.actors = [
    "Michael J. Fox",
    "Christopher Lloyd"
]
bttf.imdb = Imdb(imdb_id="tt0088763", rating=8.5)
bttf.save()

```

Queries

Use the class objects attribute to make queries. A keyword argument looks for an equal value on the field.

```

bttf = Movies.objects(title="Back To The Future").get_or_404()

```

Query operators may be used by concatenating them with the field name using a double-underscore. `objects`, and queries returned by calling it, are iterable.

```

some_theron_movie = Movie.objects(actors__in=["Charlize Theron"]).first()

for recents in Movie.objects(year__gte=2017):
    print(recents.title)

```

Documentation

There are many more ways to define and query documents with MongoEngine. For more information, check out the [official documentation](#).

Flask-MongoEngine adds helpful utilities on top of MongoEngine. Check out their [documentation](#) as well.

1.20.16 Adding a favicon

A “favicon” is an icon used by browsers for tabs and bookmarks. This helps to distinguish your website and to give it a unique brand.

A common question is how to add a favicon to a Flask application. First, of course, you need an icon. It should be 16 × 16 pixels and in the ICO file format. This is not a requirement but a de-facto standard supported by all relevant browsers. Put the icon in your static directory as `favicon.ico`.

Now, to get browsers to find your icon, the correct way is to add a link tag in your HTML. So, for example:

```
<link rel="shortcut icon" href="{{ url_for('static', filename='favicon.ico') }}">
```

That’s all you need for most browsers, however some really old ones do not support this standard. The old de-facto standard is to serve this file, with this name, at the website root. If your application is not mounted at the root path of the domain you either need to configure the web server to serve the icon at the root or if you can’t do that you’re out of luck. If however your application is the root you can simply route a redirect:

```
app.add_url_rule('/favicon.ico',  
                 redirect_to=url_for('static', filename='favicon.ico'))
```

If you want to save the extra redirect request you can also write a view using `send_from_directory()`:

```
import os  
from flask import send_from_directory  
  
@app.route('/favicon.ico')  
def favicon():  
    return send_from_directory(os.path.join(app.root_path, 'static'),  
                              'favicon.ico', mimetype='image/vnd.microsoft.icon')
```

We can leave out the explicit mimetype and it will be guessed, but we may as well specify it to avoid the extra guessing, as it will always be the same.

The above will serve the icon via your application and if possible it’s better to configure your dedicated web server to serve it; refer to the web server’s documentation.

See also

- The [Favicon](#) article on Wikipedia

1.20.17 Streaming Contents

Sometimes you want to send an enormous amount of data to the client, much more than you want to keep in memory. When you are generating the data on the fly though, how do you send that back to the client without the roundtrip to the filesystem?

The answer is by using generators and direct responses.

Basic Usage

This is a basic view function that generates a lot of CSV data on the fly. The trick is to have an inner function that uses a generator to generate data and to then invoke that function and pass it to a response object:

```
@app.route('/large.csv')
def generate_large_csv():
    def generate():
        for row in iter_all_rows():
            yield f"{'','.join(row)}\n"
    return generate(), {"Content-Type": "text/csv"}
```

Each yield expression is directly sent to the browser. Note though that some WSGI middlewares might break streaming, so be careful there in debug environments with profilers and other things you might have enabled.

Streaming from Templates

The Jinja2 template engine supports rendering a template piece by piece, returning an iterator of strings. Flask provides the `stream_template()` and `stream_template_string()` functions to make this easier to use.

```
from flask import stream_template

@app.get("/timeline")
def timeline():
    return stream_template("timeline.html")
```

The parts yielded by the render stream tend to match statement blocks in the template.

Streaming with Context

The `request` will not be active while the generator is running, because the view has already returned at that point. If you try to access `request`, you'll get a `RuntimeError`.

If your generator function relies on data in `request`, use the `stream_with_context()` wrapper. This will keep the request context active during the generator.

```
from flask import stream_with_context, request
from markupsafe import escape

@app.route('/stream')
def streamed_response():
    def generate():
        yield '<p>Hello '
        yield escape(request.args['name'])
```

(continues on next page)

```

yield '!</p>'
return stream_with_context(generate())

```

It can also be used as a decorator.

```

@stream_with_context
def generate():
    ...

return generate()

```

The `stream_template()` and `stream_template_string()` functions automatically use `stream_with_context()` if a request is active.

1.20.18 Deferred Request Callbacks

One of the design principles of Flask is that response objects are created and passed down a chain of potential callbacks that can modify them or replace them. When the request handling starts, there is no response object yet. It is created as necessary either by a view function or by some other component in the system.

What happens if you want to modify the response at a point where the response does not exist yet? A common example for that would be a `before_request()` callback that wants to set a cookie on the response object.

One way is to avoid the situation. Very often that is possible. For instance you can try to move that logic into a `after_request()` callback instead. However, sometimes moving code there makes it more complicated or awkward to reason about.

As an alternative, you can use `after_this_request()` to register callbacks that will execute after only the current request. This way you can defer code execution from anywhere in the application, based on the current request.

At any time during a request, we can register a function to be called at the end of the request. For example you can remember the current language of the user in a cookie in a `before_request()` callback:

```

from flask import request, after_this_request

@app.before_request
def detect_user_language():
    language = request.cookies.get('user_lang')

    if language is None:
        language = guess_language_from_request()

    # when the response exists, set a cookie with the language
    @after_this_request
    def remember_language(response):
        response.set_cookie('user_lang', language)
        return response

g.language = language

```

1.20.19 Adding HTTP Method Overrides

Some HTTP proxies do not support arbitrary HTTP methods or newer HTTP methods (such as PATCH). In that case it's possible to “proxy” HTTP methods through another HTTP method in total violation of the protocol.

The way this works is by letting the client do an HTTP POST request and set the X-HTTP-Method-Override header. Then the method is replaced with the header value before being passed to Flask.

This can be accomplished with an HTTP middleware:

```
class HTTPMethodOverrideMiddleware(object):
    allowed_methods = frozenset([
        'GET',
        'HEAD',
        'POST',
        'DELETE',
        'PUT',
        'PATCH',
        'OPTIONS'
    ])
    bodyless_methods = frozenset(['GET', 'HEAD', 'OPTIONS', 'DELETE'])

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        method = environ.get('HTTP_X_HTTP_METHOD_OVERRIDE', '').upper()
        if method in self.allowed_methods:
            environ['REQUEST_METHOD'] = method
        if method in self.bodyless_methods:
            environ['CONTENT_LENGTH'] = '0'
        return self.app(environ, start_response)
```

To use this with Flask, wrap the app object with the middleware:

```
from flask import Flask

app = Flask(__name__)
app.wsgi_app = HTTPMethodOverrideMiddleware(app.wsgi_app)
```

1.20.20 Request Content Checksums

Various pieces of code can consume the request data and preprocess it. For instance JSON data ends up on the request object already read and processed, form data ends up there as well but goes through a different code path. This seems inconvenient when you want to calculate the checksum of the incoming request data. This is necessary sometimes for some APIs.

Fortunately this is however very simple to change by wrapping the input stream.

The following example calculates the SHA1 checksum of the incoming data as it gets read and stores it in the WSGI environment:

```
import hashlib
```

(continues on next page)

(continued from previous page)

```
class ChecksumCalcStream(object):

    def __init__(self, stream):
        self._stream = stream
        self._hash = hashlib.sha1()

    def read(self, bytes):
        rv = self._stream.read(bytes)
        self._hash.update(rv)
        return rv

    def readline(self, size_hint):
        rv = self._stream.readline(size_hint)
        self._hash.update(rv)
        return rv

def generate_checksum(request):
    env = request.environ
    stream = ChecksumCalcStream(env['wsgi.input'])
    env['wsgi.input'] = stream
    return stream._hash
```

To use this, all you need to do is to hook the calculating stream in before the request starts consuming data. (Eg: be careful accessing `request.form` or anything of that nature. `before_request_handlers` for instance should be careful not to access it).

Example usage:

```
@app.route('/special-api', methods=['POST'])
def special_api():
    hash = generate_checksum(request)
    # Accessing this parses the input stream
    files = request.files
    # At this point the hash is fully constructed.
    checksum = hash.hexdigest()
    return f"Hash was: {checksum}"
```

1.20.21 Background Tasks with Celery

If your application has a long running task, such as processing some uploaded data or sending email, you don't want to wait for it to finish during a request. Instead, use a task queue to send the necessary data to another process that will run the task in the background while the request returns immediately.

Celery is a powerful task queue that can be used for simple background tasks as well as complex multi-stage programs and schedules. This guide will show you how to configure Celery using Flask. Read Celery's [First Steps with Celery](#) guide to learn how to use Celery itself.

The Flask repository contains an [example](#) based on the information on this page, which also shows how to use JavaScript to submit tasks and poll for progress and results.

Install

Install Celery from PyPI, for example using pip:

```
$ pip install celery
```

Integrate Celery with Flask

You can use Celery without any integration with Flask, but it's convenient to configure it through Flask's config, and to let tasks access the Flask application.

Celery uses similar ideas to Flask, with a Celery app object that has configuration and registers tasks. While creating a Flask app, use the following code to create and configure a Celery app as well.

```
from celery import Celery, Task

def celery_init_app(app: Flask) -> Celery:
    class FlaskTask(Task):
        def __call__(self, *args: object, **kwargs: object) -> object:
            with app.app_context():
                return self.run(*args, **kwargs)

    celery_app = Celery(app.name, task_cls=FlaskTask)
    celery_app.config_from_object(app.config["CELERY"])
    celery_app.set_default()
    app.extensions["celery"] = celery_app
    return celery_app
```

This creates and returns a Celery app object. Celery configuration is taken from the CELERY key in the Flask configuration. The Celery app is set as the default, so that it is seen during each request. The Task subclass automatically runs task functions with a Flask app context active, so that services like your database connections are available.

Here's a basic example.py that configures Celery to use Redis for communication. We enable a result backend, but ignore results by default. This allows us to store results only for tasks where we care about the result.

```
from flask import Flask

app = Flask(__name__)
app.config.from_mapping(
    CELERY=dict(
        broker_url="redis://localhost",
        result_backend="redis://localhost",
        task_ignore_result=True,
    ),
)
celery_app = celery_init_app(app)
```

Point the `celery worker` command at this and it will find the `celery_app` object.

```
$ celery -A example worker --loglevel INFO
```

You can also run the `celery beat` command to run tasks on a schedule. See Celery's docs for more information about defining schedules.

```
$ celery -A example beat --loglevel INFO
```

Application Factory

When using the Flask application factory pattern, call the `celery_init_app` function inside the factory. It sets `app.extensions["celery"]` to the Celery app object, which can be used to get the Celery app from the Flask app returned by the factory.

```
def create_app() -> Flask:
    app = Flask(__name__)
    app.config.from_mapping(
        CELERY=dict(
            broker_url="redis://localhost",
            result_backend="redis://localhost",
            task_ignore_result=True,
        ),
    )
    app.config.from_prefixed_env()
    celery_init_app(app)
    return app
```

To use `celery` commands, Celery needs an app object, but that's no longer directly available. Create a `make_celery.py` file that calls the Flask app factory and gets the Celery app from the returned Flask app.

```
from example import create_app

flask_app = create_app()
celery_app = flask_app.extensions["celery"]
```

Point the `celery` command to this file.

```
$ celery -A make_celery worker --loglevel INFO
$ celery -A make_celery beat --loglevel INFO
```

Defining Tasks

Using `@celery_app.task` to decorate task functions requires access to the `celery_app` object, which won't be available when using the factory pattern. It also means that the decorated tasks are tied to the specific Flask and Celery app instances, which could be an issue during testing if you change configuration for a test.

Instead, use Celery's `@shared_task` decorator. This creates task objects that will access whatever the "current app" is, which is a similar concept to Flask's blueprints and app context. This is why we called `celery_app.set_default()` above.

Here's an example task that adds two numbers together and returns the result.

```
from celery import shared_task

@shared_task(ignore_result=False)
def add_together(a: int, b: int) -> int:
    return a + b
```

Earlier, we configured Celery to ignore task results by default. Since we want to know the return value of this task, we set `ignore_result=False`. On the other hand, a task that didn't need a result, such as sending an email, wouldn't set this.

Calling Tasks

The decorated function becomes a task object with methods to call it in the background. The simplest way is to use the `delay(*args, **kwargs)` method. See Celery's docs for more methods.

A Celery worker must be running to run the task. Starting a worker is shown in the previous sections.

```
from flask import request

@app.post("/add")
def start_add() -> dict[str, object]:
    a = request.form.get("a", type=int)
    b = request.form.get("b", type=int)
    result = add_together.delay(a, b)
    return {"result_id": result.id}
```

The route doesn't get the task's result immediately. That would defeat the purpose by blocking the response. Instead, we return the running task's result id, which we can use later to get the result.

Getting Results

To fetch the result of the task we started above, we'll add another route that takes the result id we returned before. We return whether the task is finished (ready), whether it finished successfully, and what the return value (or error) was if it is finished.

```
from celery.result import AsyncResult

@app.get("/result/<id>")
def task_result(id: str) -> dict[str, object]:
    result = AsyncResult(id)
    return {
        "ready": result.ready(),
        "successful": result.successful(),
        "value": result.result if result.ready() else None,
    }
```

Now you can start the task using the first route, then poll for the result using the second route. This keeps the Flask request workers from being blocked waiting for tasks to finish.

The Flask repository contains [an example](#) using JavaScript to submit tasks and poll for progress and results.

Passing Data to Tasks

The “add” task above took two integers as arguments. To pass arguments to tasks, Celery has to serialize them to a format that it can pass to other processes. Therefore, passing complex objects is not recommended. For example, it would be impossible to pass a SQLAlchemy model object, since that object is probably not serializable and is tied to the session that queried it.

Pass the minimal amount of data necessary to fetch or recreate any complex data within the task. Consider a task that will run when the logged in user asks for an archive of their data. The Flask request knows the logged in user, and has the user object queried from the database. It got that by querying the database for a given id, so the task can do the same thing. Pass the user’s id rather than the user object.

```
@shared_task
def generate_user_archive(user_id: str) -> None:
    user = db.session.get(User, user_id)
    ...

generate_user_archive.delay(current_user.id)
```

1.20.22 Subclassing Flask

The *Flask* class is designed for subclassing.

For example, you may want to override how request parameters are handled to preserve their order:

```
from flask import Flask, Request
from werkzeug.datastructures import ImmutableOrderedMultiDict
class MyRequest(Request):
    """Request subclass to override request parameter storage"""
    parameter_storage_class = ImmutableOrderedMultiDict
class MyFlask(Flask):
    """Flask subclass using the custom request class"""
    request_class = MyRequest
```

This is the recommended approach for overriding or augmenting Flask’s internal functionality.

1.20.23 Single-Page Applications

Flask can be used to serve Single-Page Applications (SPA) by placing static files produced by your frontend framework in a subfolder inside of your project. You will also need to create a catch-all endpoint that routes all requests to your SPA.

The following example demonstrates how to serve an SPA along with an API:

```
from flask import Flask, jsonify

app = Flask(__name__, static_folder='app', static_url_path="/app")

@app.route("/heartbeat")
def heartbeat():
    return jsonify({"status": "healthy"})
```

(continues on next page)

(continued from previous page)

```
@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def catch_all(path):
    return app.send_static_file("index.html")
```

1.21 Security Considerations

Web applications usually face all kinds of security problems and it's very hard to get everything right. Flask tries to solve a few of these things for you, but there are a couple more you have to take care of yourself.

1.21.1 Cross-Site Scripting (XSS)

Cross site scripting is the concept of injecting arbitrary HTML (and with it JavaScript) into the context of a website. To remedy this, developers have to properly escape text so that it cannot include arbitrary HTML tags. For more information on that have a look at the Wikipedia article on [Cross-Site Scripting](#).

Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. This should rule out all XSS problems caused in templates, but there are still other places where you have to be careful:

- generating HTML without the help of Jinja2
- calling Markup on data submitted by users
- sending out HTML from uploaded files, never do that, use the `Content-Disposition: attachment` header to prevent that problem.
- sending out textfiles from uploaded files. Some browsers are using content-type guessing based on the first few bytes so users could trick a browser to execute HTML.

Another thing that is very important are unquoted attributes. While Jinja2 can protect you from XSS issues by escaping HTML, there is one thing it cannot protect you from: XSS by attribute injection. To counter this possible attack vector, be sure to always quote your attributes with either double or single quotes when using Jinja expressions in them:

```
<input value="{{ value }}">
```

Why is this necessary? Because if you would not be doing that, an attacker could easily inject custom JavaScript handlers. For example an attacker could inject this piece of HTML+JavaScript:

```
onmouseover=alert(document.cookie)
```

When the user would then move with the mouse over the input, the cookie would be presented to the user in an alert window. But instead of showing the cookie to the user, a good attacker might also execute any other JavaScript code. In combination with CSS injections the attacker might even make the element fill out the entire page so that the user would just have to have the mouse anywhere on the page to trigger the attack.

There is one class of XSS issues that Jinja's escaping does not protect against. The `a` tag's `href` attribute can contain a *javascript:* URI, which the browser will execute when clicked if not secured properly.

```
<a href="{{ value }}">click here</a>
<a href="javascript:alert('unsafe');">click here</a>
```

To prevent this, you'll need to set the *Content Security Policy (CSP)* response header.

1.21.2 Cross-Site Request Forgery (CSRF)

Another big problem is CSRF. This is a very complex topic and I won't outline it here in detail just mention what it is and how to theoretically prevent it.

If your authentication information is stored in cookies, you have implicit state management. The state of "being logged in" is controlled by a cookie, and that cookie is sent with each request to a page. Unfortunately that includes requests triggered by 3rd party sites. If you don't keep that in mind, some people might be able to trick your application's users with social engineering to do stupid things without them knowing.

Say you have a specific URL that, when you sent POST requests to will delete a user's profile (say `http://example.com/user/delete`). If an attacker now creates a page that sends a post request to that page with some JavaScript they just have to trick some users to load that page and their profiles will end up being deleted.

Imagine you were to run Facebook with millions of concurrent users and someone would send out links to images of little kittens. When users would go to that page, their profiles would get deleted while they are looking at images of fluffy cats.

How can you prevent that? Basically for each request that modifies content on the server you would have to either use a one-time token and store that in the cookie **and** also transmit it with the form data. After receiving the data on the server again, you would then have to compare the two tokens and ensure they are equal.

Why does Flask not do that for you? The ideal place for this to happen is the form validation framework, which does not exist in Flask.

1.21.3 JSON Security

In Flask 0.10 and lower, `jsonify()` did not serialize top-level arrays to JSON. This was because of a security vulnerability in ECMAScript 4.

ECMAScript 5 closed this vulnerability, so only extremely old browsers are still vulnerable. All of these browsers have [other more serious vulnerabilities](#), so this behavior was changed and `jsonify()` now supports serializing arrays.

1.21.4 Security Headers

Browsers recognize various response headers in order to control security. We recommend reviewing each of the headers below for use in your application. The [Flask-Talisman](#) extension can be used to manage HTTPS and the security headers for you.

HTTP Strict Transport Security (HSTS)

Tells the browser to convert all HTTP requests to HTTPS, preventing man-in-the-middle (MITM) attacks.

```
response.headers['Strict-Transport-Security'] = 'max-age=31536000; includeSubDomains'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security>

Content Security Policy (CSP)

Tell the browser where it can load various types of resource from. This header should be used whenever possible, but requires some work to define the correct policy for your site. A very strict policy would be:

```
response.headers['Content-Security-Policy'] = "default-src 'self'"
```

- <https://csp.withgoogle.com/docs/index.html>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy>

X-Content-Type-Options

Forces the browser to honor the response content type instead of trying to detect it, which can be abused to generate a cross-site scripting (XSS) attack.

```
response.headers['X-Content-Type-Options'] = 'nosniff'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options>

X-Frame-Options

Prevents external sites from embedding your site in an `iframe`. This prevents a class of attacks where clicks in the outer frame can be translated invisibly to clicks on your page's elements. This is also known as “clickjacking”.

```
response.headers['X-Frame-Options'] = 'SAMEORIGIN'
```

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options>

Set-Cookie options

These options can be added to a `Set-Cookie` header to improve their security. Flask has configuration options to set these on the session cookie. They can be set on other cookies too.

- `Secure` limits cookies to HTTPS traffic only.
- `HttpOnly` protects the contents of cookies from being read with JavaScript.
- `SameSite` restricts how cookies are sent with requests from external sites. Can be set to `'Lax'` (recommended) or `'Strict'`. `Lax` prevents sending cookies with CSRF-prone requests from external sites, such as submitting a form. `Strict` prevents sending cookies with all external requests, including following regular links.

```
app.config.update(
    SESSION_COOKIE_SECURE=True,
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SAMESITE='Lax',
)

response.set_cookie('username', 'flask', secure=True, httponly=True, samesite='Lax')
```

Specifying `Expires` or `Max-Age` options, will remove the cookie after the given time, or the current time plus the age, respectively. If neither option is set, the cookie will be removed when the browser is closed.

```
# cookie expires after 10 minutes
response.set_cookie('snakes', '3', max_age=600)
```

For the session cookie, if `session.permanent` is set, then `PERMANENT_SESSION_LIFETIME` is used to set the expiration. Flask's default cookie implementation validates that the cryptographic signature is not older than this value. Lowering this value may help mitigate replay attacks, where intercepted cookies can be sent at a later time.

```
app.config.update(
    PERMANENT_SESSION_LIFETIME=600
)

@app.route('/login', methods=['POST'])
def login():
    ...
    session.clear()
    session['user_id'] = user.id
    session.permanent = True
    ...
```

Use `itsdangerous.TimedSerializer` to sign and validate other cookie values (or any values that need secure signatures).

- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie>

HTTP Public Key Pinning (HPKP)

This tells the browser to authenticate with the server using only the specific certificate key to prevent MITM attacks.

Warning: Be careful when enabling this, as it is very difficult to undo if you set up or upgrade your key incorrectly.

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning

1.21.5 Copy/Paste to Terminal

Hidden characters such as the backspace character (`\b`, `^H`) can cause text to render differently in HTML than how it is interpreted if pasted into a terminal.

For example, `import y\bose\bm\bi\bt\be\b` renders as `import yosemite` in HTML, but the backspaces are applied when pasted into a terminal, and it becomes `import os`.

If you expect users to copy and paste untrusted code from your site, such as from comments posted by users on a technical blog, consider applying extra filtering, such as replacing all `\b` characters.

```
body = body.replace("\b", "")
```

Most modern terminals will warn about and remove hidden characters when pasting, so this isn't strictly necessary. It's also possible to craft dangerous commands in other ways that aren't possible to filter. Depending on your site's use case, it may be good to show a warning about copying code in general.

1.22 Deploying to Production

After developing your application, you'll want to make it available publicly to other users. When you're developing locally, you're probably using the built-in development server, debugger, and reloader. These should not be used in production. Instead, you should use a dedicated WSGI server or hosting platform, some of which will be described here.

“Production” means “not development”, which applies whether you're serving your application publicly to millions of users or privately / locally to a single user. **Do not use the development server when deploying to production. It is intended for use only during local development. It is not designed to be particularly secure, stable, or efficient.**

1.22.1 Self-Hosted Options

Flask is a WSGI *application*. A WSGI *server* is used to run the application, converting incoming HTTP requests to the standard WSGI environ, and converting outgoing WSGI responses to HTTP responses.

The primary goal of these docs is to familiarize you with the concepts involved in running a WSGI application using a production WSGI server and HTTP server. There are many WSGI servers and HTTP servers, with many configuration possibilities. The pages below discuss the most common servers, and show the basics of running each one. The next section discusses platforms that can manage this for you.

Gunicorn

Gunicorn is a pure Python WSGI server with simple configuration and multiple worker implementations for performance tuning.

- It tends to integrate easily with hosting platforms.
- It does not support Windows (but does run on WSL).
- It is easy to install as it does not require additional dependencies or compilation.
- It has built-in async worker support using `gevent` or `eventlet`.

This page outlines the basics of running Gunicorn. Be sure to read its [documentation](#) and use `gunicorn --help` to understand what features are available.

Installing

Gunicorn is easy to install, as it does not require external dependencies or compilation. It runs on Windows only under WSL.

Create a virtualenv, install your application, then install `gunicorn`.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install gunicorn
```

Running

The only required argument to Gunicorn tells it how to load your Flask application. The syntax is `{module_import}:{app_variable}`. `module_import` is the dotted import name to the module with your application. `app_variable` is the variable with the application. It can also be a function call (with any arguments) if you're using the app factory pattern.

```
# equivalent to 'from hello import app'
$ gunicorn -w 4 'hello:app'

# equivalent to 'from hello import create_app; create_app()'
$ gunicorn -w 4 'hello:create_app()'

Starting gunicorn 20.1.0
Listening at: http://127.0.0.1:8000 (x)
Using worker: sync
Booting worker with pid: x
```

The `-w` option specifies the number of processes to run; a starting value could be `CPU * 2`. The default is only 1 worker, which is probably not what you want for the default worker type.

Logs for each request aren't shown by default, only worker info and errors are shown. To show access logs on stdout, use the `--access-logfile=-` option.

Binding Externally

Gunicorn should not be run as root because it would cause your application code to run as root, which is not secure. However, this means it will not be possible to bind to port 80 or 443. Instead, a reverse proxy such as *nginx* or *Apache httpd* should be used in front of Gunicorn.

You can bind to all external IPs on a non-privileged port using the `-b 0.0.0.0` option. Don't do this when using a reverse proxy setup, otherwise it will be possible to bypass the proxy.

```
$ gunicorn -w 4 -b 0.0.0.0 'hello:create_app()'
Listening at: http://0.0.0.0:8000 (x)
```

`0.0.0.0` is not a valid address to navigate to, you'd use a specific IP address in your browser.

Async with gevent or eventlet

The default sync worker is appropriate for many use cases. If you need asynchronous support, Gunicorn provides workers using either *gevent* or *eventlet*. This is not the same as Python's `async/await`, or the ASGI server spec. You must actually use *gevent/eventlet* in your own code to see any benefit to using the workers.

When using either *gevent* or *eventlet*, `greenlet>=1.0` is required, otherwise context locals such as `request` will not work as expected. When using PyPy, `PyPy>=7.3.7` is required.

To use *gevent*:

```
$ gunicorn -k gevent 'hello:create_app()'
Starting gunicorn 20.1.0
Listening at: http://127.0.0.1:8000 (x)
Using worker: gevent
Booting worker with pid: x
```

To use eventlet:

```
$ gunicorn -k eventlet 'hello:create_app()'
Starting gunicorn 20.1.0
Listening at: http://127.0.0.1:8000 (x)
Using worker: eventlet
Booting worker with pid: x
```

Waitress

Waitress is a pure Python WSGI server.

- It is easy to configure.
- It supports Windows directly.
- It is easy to install as it does not require additional dependencies or compilation.
- It does not support streaming requests, full request data is always buffered.
- It uses a single process with multiple thread workers.

This page outlines the basics of running Waitress. Be sure to read its documentation and `waitress-serve --help` to understand what features are available.

Installing

Create a virtualenv, install your application, then install waitress.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install waitress
```

Running

The only required argument to `waitress-serve` tells it how to load your Flask application. The syntax is `{module}:{app}`. `module` is the dotted import name to the module with your application. `app` is the variable with the application. If you're using the app factory pattern, use `--call {module}:{factory}` instead.

```
# equivalent to 'from hello import app'
$ waitress-serve --host 127.0.0.1 hello:app

# equivalent to 'from hello import create_app; create_app()'
$ waitress-serve --host 127.0.0.1 --call hello:create_app
```

(continues on next page)

(continued from previous page)

```
Serving on http://127.0.0.1:8080
```

The `--host` option binds the server to local `127.0.0.1` only.

Logs for each request aren't shown, only errors are shown. Logging can be configured through the Python interface instead of the command line.

Binding Externally

Waitress should not be run as root because it would cause your application code to run as root, which is not secure. However, this means it will not be possible to bind to port 80 or 443. Instead, a reverse proxy such as *nginx* or *Apache httpd* should be used in front of Waitress.

You can bind to all external IPs on a non-privileged port by not specifying the `--host` option. Don't do this when using a reverse proxy setup, otherwise it will be possible to bypass the proxy.

`0.0.0.0` is not a valid address to navigate to, you'd use a specific IP address in your browser.

mod_wsgi

`mod_wsgi` is a WSGI server integrated with the *Apache httpd* server. The modern `mod_wsgi-express` command makes it easy to configure and start the server without needing to write Apache httpd configuration.

- Tightly integrated with Apache httpd.
- Supports Windows directly.
- Requires a compiler and the Apache development headers to install.
- Does not require a reverse proxy setup.

This page outlines the basics of running `mod_wsgi-express`, not the more complex installation and configuration with httpd. Be sure to read the `mod_wsgi-express`, `mod_wsgi`, and *Apache httpd* documentation to understand what features are available.

Installing

Installing `mod_wsgi` requires a compiler and the Apache server and development headers installed. You will get an error if they are not. How to install them depends on the OS and package manager that you use.

Create a virtualenv, install your application, then install `mod_wsgi`.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install mod_wsgi
```

Running

The only argument to `mod_wsgi-express` specifies a script containing your Flask application, which must be called `application`. You can write a small script to import your app with this name, or to create it if using the app factory pattern.

Listing 47: `wsgi.py`

```
from hello import app

application = app
```

Listing 48: `wsgi.py`

```
from hello import create_app

application = create_app()
```

Now run the `mod_wsgi-express start-server` command.

```
$ mod_wsgi-express start-server wsgi.py --processes 4
```

The `--processes` option specifies the number of worker processes to run; a starting value could be `CPU * 2`.

Logs for each request aren't show in the terminal. If an error occurs, its information is written to the error log file shown when starting the server.

Binding Externally

Unlike the other WSGI servers in these docs, `mod_wsgi` can be run as root to bind to privileged ports like 80 and 443. However, it must be configured to drop permissions to a different user and group for the worker processes.

For example, if you created a `hello` user and group, you should install your virtualenv and application as that user, then tell `mod_wsgi` to drop to that user after starting.

```
$ sudo /home/hello/.venv/bin/mod_wsgi-express start-server \
/home/hello/wsgi.py \
--user hello --group hello --port 80 --processes 4
```

uWSGI

`uWSGI` is a fast, compiled server suite with extensive configuration and capabilities beyond a basic server.

- It can be very performant due to being a compiled program.
- It is complex to configure beyond the basic application, and has so many options that it can be difficult for beginners to understand.
- It does not support Windows (but does run on WSL).
- It requires a compiler to install in some cases.

This page outlines the basics of running `uWSGI`. Be sure to read its documentation to understand what features are available.

Installing

uWSGI has multiple ways to install it. The most straightforward is to install the `pyuwsgi` package, which provides precompiled wheels for common platforms. However, it does not provide SSL support, which can be provided with a reverse proxy instead.

Create a virtualenv, install your application, then install `pyuwsgi`.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install pyuwsgi
```

If you have a compiler available, you can install the `uwsgi` package instead. Or install the `pyuwsgi` package from `sdist` instead of wheel. Either method will include SSL support.

```
$ pip install uwsgi

# or
$ pip install --no-binary pyuwsgi pyuwsgi
```

Running

The most basic way to run uWSGI is to tell it to start an HTTP server and import your application.

```
$ uwsgi --http 127.0.0.1:8000 --master -p 4 -w hello:app

*** Starting uWSGI 2.0.20 (64bit) on [x] ***
*** Operational MODE: preforking ***
mounting hello:app on /
spawned uWSGI master process (pid: x)
spawned uWSGI worker 1 (pid: x, cores: 1)
spawned uWSGI worker 2 (pid: x, cores: 1)
spawned uWSGI worker 3 (pid: x, cores: 1)
spawned uWSGI worker 4 (pid: x, cores: 1)
spawned uWSGI http 1 (pid: x)
```

If you're using the app factory pattern, you'll need to create a small Python file to create the app, then point uWSGI at that.

Listing 49: `wsgi.py`

```
from hello import create_app

app = create_app()
```

```
$ uwsgi --http 127.0.0.1:8000 --master -p 4 -w wsgi:app
```

The `--http` option starts an HTTP server at 127.0.0.1 port 8000. The `--master` option specifies the standard worker manager. The `-p` option starts 4 worker processes; a starting value could be `CPU * 2`. The `-w` option tells uWSGI how to import your application

Binding Externally

uWSGI should not be run as root with the configuration shown in this doc because it would cause your application code to run as root, which is not secure. However, this means it will not be possible to bind to port 80 or 443. Instead, a reverse proxy such as *nginx* or *Apache httpd* should be used in front of uWSGI. It is possible to run uWSGI as root securely, but that is beyond the scope of this doc.

uWSGI has optimized integration with *Nginx uWSGI* and *Apache mod_proxy_uwsgi*, and possibly other servers, instead of using a standard HTTP proxy. That configuration is beyond the scope of this doc, see the links for more information.

You can bind to all external IPs on a non-privileged port using the `--http 0.0.0.0:8000` option. Don't do this when using a reverse proxy setup, otherwise it will be possible to bypass the proxy.

```
$ uwsgi --http 0.0.0.0:8000 --master -p 4 -w wsgi:app
```

0.0.0.0 is not a valid address to navigate to, you'd use a specific IP address in your browser.

Async with gevent

The default sync worker is appropriate for many use cases. If you need asynchronous support, uWSGI provides a *gevent* worker. This is not the same as Python's `async/await`, or the ASGI server spec. You must actually use *gevent* in your own code to see any benefit to using the worker.

When using *gevent*, `greenlet>=1.0` is required, otherwise context locals such as `request` will not work as expected. When using *PyPy*, `PyPy>=7.3.7` is required.

```
$ uwsgi --http 127.0.0.1:8000 --master --gevent 100 -w wsgi:app
```

```
*** Starting uWSGI 2.0.20 (64bit) on [x] ***
*** Operational MODE: async ***
mounting hello:app on /
spawned uWSGI master process (pid: x)
spawned uWSGI worker 1 (pid: x, cores: 100)
spawned uWSGI http 1 (pid: x)
*** running gevent loop engine [addr:x] ***
```

gevent

Prefer using *Gunicorn* or *uWSGI* with *gevent* workers rather than using *gevent* directly. *Gunicorn* and *uWSGI* provide much more configurable and production-tested servers.

gevent allows writing asynchronous, coroutine-based code that looks like standard synchronous Python. It uses *greenlet* to enable task switching without writing `async/await` or using *asyncio*.

eventlet is another library that does the same thing. Certain dependencies you have, or other considerations, may affect which of the two you choose to use.

gevent provides a WSGI server that can handle many connections at once instead of one per worker process. You must actually use *gevent* in your own code to see any benefit to using the server.

Installing

When using `gevent`, `greenlet`>=1.0 is required, otherwise context locals such as `request` will not work as expected. When using PyPy, PyPy>=7.3.7 is required.

Create a virtualenv, install your application, then install `gevent`.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install gevent
```

Running

To use `gevent` to serve your application, write a script that imports its `WSGIServer`, as well as your app or app factory.

Listing 50: `wsgi.py`

```
from gevent.pywsgi import WSGIServer
from hello import create_app

app = create_app()
http_server = WSGIServer(("127.0.0.1", 8000), app)
http_server.serve_forever()
```

```
$ python wsgi.py
```

No output is shown when the server starts.

Binding Externally

`gevent` should not be run as root because it would cause your application code to run as root, which is not secure. However, this means it will not be possible to bind to port 80 or 443. Instead, a reverse proxy such as *nginx* or *Apache httpd* should be used in front of `gevent`.

You can bind to all external IPs on a non-privileged port by using `0.0.0.0` in the server arguments shown in the previous section. Don't do this when using a reverse proxy setup, otherwise it will be possible to bypass the proxy.

`0.0.0.0` is not a valid address to navigate to, you'd use a specific IP address in your browser.

eventlet

Prefer using *Gunicorn* with `eventlet` workers rather than using `eventlet` directly. *Gunicorn* provides a much more configurable and production-tested server.

`eventlet` allows writing asynchronous, coroutine-based code that looks like standard synchronous Python. It uses `greenlet` to enable task switching without writing `async/await` or using `asyncio`.

gevent is another library that does the same thing. Certain dependencies you have, or other considerations, may affect which of the two you choose to use.

`eventlet` provides a WSGI server that can handle many connections at once instead of one per worker process. You must actually use `eventlet` in your own code to see any benefit to using the server.

Installing

When using eventlet, greenlet \geq 1.0 is required, otherwise context locals such as `request` will not work as expected. When using PyPy, PyPy \geq 7.3.7 is required.

Create a virtualenv, install your application, then install eventlet.

```
$ cd hello-app
$ python -m venv .venv
$ . .venv/bin/activate
$ pip install . # install your application
$ pip install eventlet
```

Running

To use eventlet to serve your application, write a script that imports its `wsgi.server`, as well as your app or app factory.

Listing 51: `wsgi.py`

```
import eventlet
from eventlet import wsgi
from hello import create_app

app = create_app()
wsgi.server(eventlet.listen(("127.0.0.1", 8000)), app)
```

```
$ python wsgi.py
(x) wsgi starting up on http://127.0.0.1:8000
```

Binding Externally

eventlet should not be run as root because it would cause your application code to run as root, which is not secure. However, this means it will not be possible to bind to port 80 or 443. Instead, a reverse proxy such as *nginx* or *Apache httpd* should be used in front of eventlet.

You can bind to all external IPs on a non-privileged port by using `0.0.0.0` in the server arguments shown in the previous section. Don't do this when using a reverse proxy setup, otherwise it will be possible to bypass the proxy.

`0.0.0.0` is not a valid address to navigate to, you'd use a specific IP address in your browser.

ASGI

If you'd like to use an ASGI server you will need to utilise WSGI to ASGI middleware. The asgiref `WsgiToAsgi` adapter is recommended as it integrates with the event loop used for Flask's *Using async and await* support. You can use the adapter by wrapping the Flask app,

```
from asgiref.wsgi import WsgiToAsgi
from flask import Flask

app = Flask(__name__)
```

(continues on next page)

(continued from previous page)

```
...  
asgi_app = WsgiToAsgi(app)
```

and then serving the `asgi_app` with the ASGI server, e.g. using [Hypercorn](#),

```
$ hypercorn module:asgi_app
```

WSGI servers have HTTP servers built-in. However, a dedicated HTTP server may be safer, more efficient, or more capable. Putting an HTTP server in front of the WSGI server is called a “reverse proxy.”

Tell Flask it is Behind a Proxy

When using a reverse proxy, or many Python hosting platforms, the proxy will intercept and forward all external requests to the local WSGI server.

From the WSGI server and Flask application’s perspectives, requests are now coming from the HTTP server to the local address, rather than from the remote address to the external server address.

HTTP servers should set `X-Forwarded-` headers to pass on the real values to the application. The application can then be told to trust and use those values by wrapping it with the `X-Forwarded-For Proxy Fix` middleware provided by Werkzeug.

This middleware should only be used if the application is actually behind a proxy, and should be configured with the number of proxies that are chained in front of it. Not all proxies set all the headers. Since incoming headers can be faked, you must set how many proxies are setting each header so the middleware knows what to trust.

```
from werkzeug.middleware.proxy_fix import ProxyFix  
  
app.wsgi_app = ProxyFix(  
    app.wsgi_app, x_for=1, x_proto=1, x_host=1, x_prefix=1  
)
```

Remember, only apply this middleware if you are behind a proxy, and set the correct number of proxies that set each header. It can be a security issue if you get this configuration wrong.

nginx

`nginx` is a fast, production level HTTP server. When serving your application with one of the WSGI servers listed in *Deploying to Production*, it is often good or necessary to put a dedicated HTTP server in front of it. This “reverse proxy” can handle incoming requests, TLS, and other security and performance concerns better than the WSGI server.

Nginx can be installed using your system package manager, or a pre-built executable for Windows. Installing and running Nginx itself is outside the scope of this doc. This page outlines the basics of configuring Nginx to proxy your application. Be sure to read its documentation to understand what features are available.

Domain Name

Acquiring and configuring a domain name is outside the scope of this doc. In general, you will buy a domain name from a registrar, pay for server space with a hosting provider, and then point your registrar at the hosting provider's name servers.

To simulate this, you can also edit your `hosts` file, located at `/etc/hosts` on Linux. Add a line that associates a name with the local IP.

Modern Linux systems may be configured to treat any domain name that ends with `.localhost` like this without adding it to the `hosts` file.

Listing 52: `/etc/hosts`

```
127.0.0.1 hello.localhost
```

Configuration

The `nginx` configuration is located at `/etc/nginx/nginx.conf` on Linux. It may be different depending on your operating system. Check the docs and look for `nginx.conf`.

Remove or comment out any existing `server` section. Add a `server` section and use the `proxy_pass` directive to point to the address the WSGI server is listening on. We'll assume the WSGI server is listening locally at `http://127.0.0.1:8000`.

Listing 53: `/etc/nginx.conf`

```
server {
    listen 80;
    server_name _;

    location / {
        proxy_pass http://127.0.0.1:8000/;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-Host $host;
        proxy_set_header X-Forwarded-Prefix /;
    }
}
```

Then *Tell Flask it is Behind a Proxy* so that your application uses these headers.

Apache httpd

`Apache httpd` is a fast, production level HTTP server. When serving your application with one of the WSGI servers listed in *Deploying to Production*, it is often good or necessary to put a dedicated HTTP server in front of it. This “reverse proxy” can handle incoming requests, TLS, and other security and performance concerns better than the WSGI server.

`httpd` can be installed using your system package manager, or a pre-built executable for Windows. Installing and running `httpd` itself is outside the scope of this doc. This page outlines the basics of configuring `httpd` to proxy your application. Be sure to read its documentation to understand what features are available.

Domain Name

Acquiring and configuring a domain name is outside the scope of this doc. In general, you will buy a domain name from a registrar, pay for server space with a hosting provider, and then point your registrar at the hosting provider's name servers.

To simulate this, you can also edit your `hosts` file, located at `/etc/hosts` on Linux. Add a line that associates a name with the local IP.

Modern Linux systems may be configured to treat any domain name that ends with `.localhost` like this without adding it to the `hosts` file.

Listing 54: `/etc/hosts`

```
127.0.0.1 hello.localhost
```

Configuration

The `httpd` configuration is located at `/etc/httpd/conf/httpd.conf` on Linux. It may be different depending on your operating system. Check the docs and look for `httpd.conf`.

Remove or comment out any existing `DocumentRoot` directive. Add the config lines below. We'll assume the WSGI server is listening locally at `http://127.0.0.1:8000`.

Listing 55: `/etc/httpd/conf/httpd.conf`

```
LoadModule proxy_module modules/mod_proxy.so
LoadModule proxy_http_module modules/mod_proxy_http.so
ProxyPass / http://127.0.0.1:8000/
RequestHeader set X-Forwarded-Proto http
RequestHeader set X-Forwarded-Prefix /
```

The `LoadModule` lines might already exist. If so, make sure they are uncommented instead of adding them manually.

Then *Tell Flask it is Behind a Proxy* so that your application uses the `X-Forwarded` headers. `X-Forwarded-For` and `X-Forwarded-Host` are automatically set by `ProxyPass`.

This list is not exhaustive, and you should evaluate these and other servers based on your application's needs. Different servers will have different capabilities, configuration, and support.

1.22.2 Hosting Platforms

There are many services available for hosting web applications without needing to maintain your own server, networking, domain, etc. Some services may have a free tier up to a certain time or bandwidth. Many of these services use one of the WSGI servers described above, or a similar interface. The links below are for some of the most common platforms, which have instructions for Flask, WSGI, or Python.

- [PythonAnywhere](#)
- [Google App Engine](#)
- [Google Cloud Run](#)
- [AWS Elastic Beanstalk](#)
- [Microsoft Azure](#)

This list is not exhaustive, and you should evaluate these and other services based on your application's needs. Different services will have different capabilities, configuration, pricing, and support.

You'll probably need to *Tell Flask it is Behind a Proxy* when using most hosting platforms.

1.23 Using async and await

New in version 2.0.

Routes, error handlers, before request, after request, and teardown functions can all be coroutine functions if Flask is installed with the `async extra` (`pip install flask[async]`). This allows views to be defined with `async def` and use `await`.

```
@app.route("/get-data")
async def get_data():
    data = await async_db_query(...)
    return jsonify(data)
```

Pluggable class-based views also support handlers that are implemented as coroutines. This applies to the `dispatch_request()` method in views that inherit from the `flask.views.View` class, as well as all the HTTP method handlers in views that inherit from the `flask.views.MethodView` class.

Using async on Windows on Python 3.8

Python 3.8 has a bug related to `asyncio` on Windows. If you encounter something like `ValueError: set_wakeup_fd only works in main thread`, please upgrade to Python 3.9.

Using async with greenlet

When using `gevent` or `eventlet` to serve an application or patch the runtime, `greenlet>=1.0` is required. When using `PyPy`, `PyPy>=7.3.7` is required.

1.23.1 Performance

Async functions require an event loop to run. Flask, as a WSGI application, uses one worker to handle one request/response cycle. When a request comes in to an async view, Flask will start an event loop in a thread, run the view function there, then return the result.

Each request still ties up one worker, even for async views. The upside is that you can run async code within a view, for example to make multiple concurrent database queries, HTTP requests to an external API, etc. However, the number of requests your application can handle at one time will remain the same.

Async is not inherently faster than sync code. Async is beneficial when performing concurrent IO-bound tasks, but will probably not improve CPU-bound tasks. Traditional Flask views will still be appropriate for most use cases, but Flask's async support enables writing and using code that wasn't possible natively before.

1.23.2 Background tasks

Async functions will run in an event loop until they complete, at which stage the event loop will stop. This means any additional spawned tasks that haven't completed when the async function completes will be cancelled. Therefore you cannot spawn background tasks, for example via `asyncio.create_task`.

If you wish to use background tasks it is best to use a task queue to trigger background work, rather than spawn tasks in a view function. With that in mind you can spawn `asyncio` tasks by serving Flask with an ASGI server and utilising the `asgiref WsgiToAsgi` adapter as described in [ASGI](#). This works as the adapter creates an event loop that runs continually.

1.23.3 When to use Quart instead

Flask's async support is less performant than async-first frameworks due to the way it is implemented. If you have a mainly async codebase it would make sense to consider [Quart](#). Quart is a reimplementation of Flask based on the [ASGI](#) standard instead of WSGI. This allows it to handle many concurrent requests, long running requests, and websockets without requiring multiple worker processes or threads.

It has also already been possible to run Flask with `Gevent` or `Eventlet` to get many of the benefits of async request handling. These libraries patch low-level Python functions to accomplish this, whereas `async/await` and ASGI use standard, modern Python capabilities. Deciding whether you should use Flask, Quart, or something else is ultimately up to understanding the specific needs of your project.

1.23.4 Extensions

Flask extensions predating Flask's async support do not expect async views. If they provide decorators to add functionality to views, those will probably not work with async views because they will not await the function or be awaitable. Other functions they provide will not be awaitable either and will probably be blocking if called within an async view.

Extension authors can support async functions by utilising the `flask.Flask.ensure_sync()` method. For example, if the extension provides a view function decorator add `ensure_sync` before calling the decorated function,

```
def extension(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ... # Extension logic
        return current_app.ensure_sync(func)(*args, **kwargs)

    return wrapper
```

Check the changelog of the extension you want to use to see if they've implemented async support, or make a feature request or PR to them.

1.23.5 Other event loops

At the moment Flask only supports `asyncio`. It's possible to override `flask.Flask.ensure_sync()` to change how async functions are wrapped to use a different library.

API REFERENCE

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 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.

2.1.1 Application Object

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

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

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

For more information about resource loading, see [open_resource\(\)](#).

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

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

About the First Parameter

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

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

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

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

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

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

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

New in version 0.11: The `root_path` parameter was added.

New in version 0.8: The `instance_path` and `instance_relative_config` parameters were added.

New in version 0.7: The `static_url_path`, `static_folder`, and `template_folder` parameters were added.

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* | *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* | *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.

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.

New in version 2.2: Moved from `flask.abort`, which calls this object.

aborter_class

alias of `werkzeug.exceptions.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.

New in version 0.10.

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.

New in version 0.10.

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")
```

(continues on next page)

(continued from previous page)

```
@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` (*flask.scaffold.T_after_request*) –

Return type `flask.scaffold.T_after_request`

`after_request_funcs: dict[ft.AppOrBlueprintKey, list[ft.AfterRequestCallable]]`

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` the functions 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.

`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*.

New in version 0.9.

Return type `flask.ctx.AppContext`

`app_ctx_globals_class`

alias of `flask.ctx._AppCtxGlobals`

`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.

New in version 2.0.

Parameters `func` (`Callable[[...], Coroutine]`) –

Return type `Callable[[...], Any]`

`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.

New in version 0.8.

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` (`flask.scaffold.T_before_request`) –

Return type `flask.scaffold.T_before_request`

`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.

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.

New in version 0.7.

cli

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.

config

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

config_class

alias of `flask.config.Config`

context_processor(*f*)

Registers a template context processor function. 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* (`flask.scaffold.T_template_context_processor`) –

Return type `flask.scaffold.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 loaders of the application and the individual blueprints.

New in version 0.7.

Return type `flask.templating.DispatchingJinjaLoader`

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.

Changed in version 0.11: `Environment.auto_reload` set in accordance with `TEMPLATES_AUTO_RELOAD` configuration option.

New in version 0.5.

Return type `flask.templating.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.

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

Changed in version 0.9: This can now also be called without a request object when the URL adapter is created for the application context.

New in version 0.6.

Parameters `request` (`flask.wrappers.Request` | `None`) –

Return type `werkzeug.routing.map.MapAdapter` | `None`

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`

```
default_config = {'APPLICATION_ROOT': '/', 'DEBUG': None,
'EXPLAIN_TEMPLATE_LOADING': False, 'MAX_CONTENT_LENGTH': None, 'MAX_COOKIE_SIZE':
4093, 'PERMANENT_SESSION_LIFETIME': datetime.timedelta(days=31),
'PREFERRED_URL_SCHEME': 'http', 'PROPAGATE_EXCEPTIONS': None, 'SECRET_KEY': None,
'SEND_FILE_MAX_AGE_DEFAULT': None, 'SERVER_NAME': None, 'SESSION_COOKIE_DOMAIN':
None, 'SESSION_COOKIE_HTTPONLY': True, 'SESSION_COOKIE_NAME': 'session',
'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_SAMESITE': None,
'SESSION_COOKIE_SECURE': False, 'SESSION_REFRESH_EACH_REQUEST': True,
'TEMPLATES_AUTO_RELOAD': None, 'TESTING': False, 'TRAP_BAD_REQUEST_ERRORS': None,
'TRAP_HTTP_EXCEPTIONS': False, 'USE_X_SENDFILE': False}
```

Default configuration parameters.

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

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

New in version 2.0.

Parameters

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

Return type `Callable`[[`flask.scaffold.T_route`], `flask.scaffold.T_route`]

dispatch_request()

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

Changed in version 0.7: This no longer does the exception handling, this code was moved to the new `full_dispatch_request()`.

Return type `ft.ResponseReturnValue`

do_teardown_appcontext(*exc=<object object>*)

Called right before the application context is popped.

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

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

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

New in version 0.9.

Parameters `exc` (*BaseException* | *None*) –

Return type *None*

do_teardown_request (*exc*=<object object>)

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

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

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

Parameters `exc` (*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*

Changed in version 0.9: Added the `exc` argument.

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*[[*flask.scaffold.F*], *flask.scaffold.F*]

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.

New in version 2.0.

Parameters `func` (*Callable*) –

Return type *Callable*

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.

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()`.

New in version 0.7: Use `register_error_handler()` instead of modifying `error_handler_spec` directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

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

Return type `Callable`[[`flask.scaffold.T_error_handler`], `flask.scaffold.T_error_handler`]

extensions: `dict`

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'`.

New in version 0.7.

finalize_request(*rv*, *from_error_handler=False*)

Given the return value from a view function this finalizes the request by converting it into a response and invoking the postprocessing functions. This is invoked for both normal request dispatching as well as error handlers.

Because this means that it might be called as a result of a failure a special safe mode is available which can be enabled with the `from_error_handler` flag. If enabled, failures in response processing will be logged and otherwise ignored.

Internal**Parameters**

- `rv` (*ft.ResponseReturnValue* | `HTTPException`) –
- `from_error_handler` (*bool*) –

Return type `Response`

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.

New in version 0.7.

Return type *flask.wrappers.Response*

get(*rule*, ***options*)

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

New in version 2.0.

Parameters

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

Return type *Callable*[[*flask.scaffold.T_route*], *flask.scaffold.T_route*]

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.

Changed in version 2.0: The default configuration is `None` instead of 12 hours.

New in version 0.9.

Parameters **filename** (*str* | *None*) –

Return type *int* | *None*

property `got_first_request`: **bool**

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

Deprecated since version 2.3: Will be removed in Flask 2.4.

New in version 0.8.

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`.

Changed in version 1.1.0: Always passes the `InternalServerError` instance to the handler, setting `original_exception` to the unhandled error.

Changed in version 1.1.0: `after_request` functions and other finalization is done even for the default 500 response when there is no handler.

New in version 0.3.

Parameters **e** (*Exception*) –

Return type *flask.wrappers.Response*

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.

Changed in version 1.0.3: `RoutingException`, used internally for actions such as slash redirects during routing, is not passed to error handlers.

Changed in version 1.0: Exceptions are looked up by code *and* by MRO, so `HTTPException` subclasses can be handled with a catch-all handler for the base `HTTPException`.

New in version 0.3.

Parameters *e* (`HTTPException`) –

Return type `HTTPException` | `ft.ResponseReturnValue`

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** (`werkzeug.routing.exceptions.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`

handle_user_exception(*e*)

This method is called whenever an exception occurs that should be handled. A 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.

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

New in version 0.7.

Parameters *e* (`Exception`) –

Return type `HTTPException` | `ft.ResponseReturnValue`

property has_static_folder: bool

True if `static_folder` is set.

New in version 0.5.

import_name

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

inject_url_defaults(*endpoint*, *values*)

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

New in version 0.7.

Parameters

- **endpoint** (*str*) –
- **values** (*dict*) –

Return type None

instance_path

Holds the path to the instance folder.

New in version 0.8.

iter_blueprints()

Iterates over all blueprints by the order they were registered.

New in version 0.11.

Return type `t.ValuesView[Blueprint]`

property jinja_env: flask.templating.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 `flask.templating.Environment`

property jinja_loader: jinja2.loaders.FileSystemLoader | 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.

New in version 0.5.

jinja_options: dict = {}

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.

Changed in version 1.1.0: This is a `dict` instead of an `ImmutableDict` to allow easier configuration.

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.

New in version 2.2.

json_provider_class

alias of `flask.json.provider.DefaultJSONProvider`

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*.

New in version 0.8.

Parameters exc_info (`tuple[type, BaseException, traceback] | tuple[None, None, None]`) –

Return type None

property logger: `logging.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.

Changed in version 1.1.0: The logger takes the same name as *name* rather than hard-coding `"flask.app"`.

Changed in version 1.0.0: Behavior was simplified. The logger is always named `"flask.app"`. The level is only set during configuration, it doesn't check `app.debug` each time. Only one format is used, not different ones depending on `app.debug`. No handlers are removed, and a handler is only added if no handlers are already configured.

New in version 0.3.

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`.

New in version 2.2.

Return type `werkzeug.exceptions.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.

New in version 0.8.

Parameters *instance_relative* (*bool*) –

Return type `flask.config.Config`

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.

New in version 0.7.

Return type `flask.wrappers.Response`

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 existing 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*

Changed in version 2.2: A generator will be converted to a streaming response. A list will be converted to a JSON response.

Changed in version 1.1: A dict will be converted to a JSON response.

Changed in version 0.9: Previously a tuple was interpreted as the arguments for the response object.

make_shell_context()

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

New in version 0.11.

Return type *dict*

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.

New in version 0.8.

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*

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*

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

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

New in version 2.0.

Parameters

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

Return type *Callable*[[`flask.scaffold.T_route`], `flask.scaffold.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"]`.

New in version 2.0.

Parameters

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

Return type *Callable*[[`flask.scaffold.T_route`], `flask.scaffold.T_route`]

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.

Changed in version 0.5: As of Flask 0.5 the functions registered for after request execution are called in reverse order of registration.

Parameters **response** (`flask.wrappers.Response`) – a `response_class` object.

Returns a new response object or the same, has to be an instance of `response_class`.

Return type `flask.wrappers.Response`

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

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

New in version 2.0.

Parameters

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

Return type *Callable*[[flask.scaffold.T_route], flask.scaffold.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 *werkzeug.wrappers.response.Response*

New in version 2.2: Moved from *flask.redirect*, which calls this method.

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*

Changed in version 2.0.1: The *name* option can be used to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for *url_for*.

New in version 0.7.

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.

New in version 0.7.

Parameters

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

Return type *None*

request_class

alias of *flask.wrappers.Request*

request_context(*environ*)

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

See *The Request Context*.

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

Parameters *environ* (*dict*) – a WSGI environment

Return type *flask.ctx.RequestContext*

response_class

alias of *flask.wrappers.Response*

root_path

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

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*[[*flask.scaffold.T_route*], *flask.scaffold.T_route*]

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_eval=False* as parameter. This will keep the debugger's traceback screen active, but disable code execution.

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

Keep in Mind

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

Parameters

- **host** (*Optional[str]*) – 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** (*Optional[int]*) – the port of the webserver. Defaults to `5000` or the port defined in the `SERVER_NAME` config variable if present.
- **debug** (*Optional[bool]*) – 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`

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

The `FLASK_DEBUG` environment variable will override [debug](#).

Threaded mode is enabled by default.

Changed in version 0.10: The default port is now picked from the `SERVER_NAME` variable.

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`.

Changed in version 2.2: Autoescaping is now enabled by default for `.svg` files.

New in version 0.5.

Parameters `filename` (*str*) –

Return type `bool`

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.

New in version 0.5.

Parameters `filename` (*str*) –

Return type `Response`

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

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

New in version 0.8.

shell_context_processor(*f*)

Registers a shell context processor function.

New in version 0.11.

Parameters *f* (`flask.app.T_shell_context_processor`) –

Return type `flask.app.T_shell_context_processor`

shell_context_processors: `list[ft.ShellContextProcessorCallable]`

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

New in version 0.11.

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.

New in version 0.10.

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`

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.

New in version 0.9.

Parameters *f* (`flask.app.T_teardown`) –

Return type `flask.app.T_teardown`

teardown_appcontext_funcs: `list[ft.TearDownCallable]`

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

New in version 0.9.

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 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* (`flask.scaffold.T_teardown`) –

Return type `flask.scaffold.T_teardown`

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.

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*[[*flask.app.T_template_filter*], *flask.app.T_template_filter*]

`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.

`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
```

New in version 0.10.

Parameters `name` (*str* / *None*) – the optional name of the global function, otherwise the function name will be used.

Return type *Callable*[[*flask.app.T_template_global*], *flask.app.T_template_global*]

`template_test` (*name=None*)

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

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

New in version 0.10.

Parameters `name` (*str* / *None*) – the optional name of the test, otherwise the function name will be used.

Return type *Callable*[[*flask.app.T_template_test*], *flask.app.T_template_test*]

`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.

New in version 1.0.

Parameters `kwargs` (*t.Any*) –

Return type *FlaskCliRunner*

`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.

New in version 1.0.

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.

Changed in version 0.11: Added ***kwargs* to support passing additional keyword arguments to the constructor of *test_client_class*.

New in version 0.7: The *use_cookies* parameter was added as well as the ability to override the client to be used by setting the *test_client_class* attribute.

Changed in version 0.4: added support for `with` block usage for the client.

Parameters

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

Return type *FlaskClient*

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

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

New in version 0.7.

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 `flask.ctx.RequestContext`

testing

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

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

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

`trap_http_exception(e)`

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

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

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

New in version 0.8.

Parameters *e* (*Exception*) –

Return type `bool`

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

Return type `None`

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`.

New in version 0.9.

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.

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* (`flask.scaffold.T_url_defaults`) –

Return type `flask.scaffold.T_url_defaults`

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** (*Optional[str]*) – If given, append this as `#anchor` to the URL.
- **_method** (*Optional[str]*) – If given, generate the URL associated with this method for the endpoint.
- **_scheme** (*Optional[str]*) – If given, the URL will have this scheme if it is external.
- **_external** (*Optional[bool]*) – 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`

New in version 2.2: Moved from `flask.url_for`, which calls this method.

`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
```

`url_map_class`

alias of `werkzeug.routing.map.Map`

`url_rule_class`

alias of `werkzeug.routing.rules.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 every request, use `Blueprint.app_url_value_preprocessor()`.

Parameters `f` (`flask.scaffold.T_url_value_preprocessor`) –

Return type `flask.scaffold.T_url_value_preprocessor`

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.

view_functions: `dict[str, t.Callable]`

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.

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.

Changed in version 0.7: Teardown events for the request and app contexts are called even if an unhandled error occurs. Other events may not be called depending on when an error occurs during dispatch. See [Callbacks and Errors](#).

Parameters

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

Return type `Any`

2.1.2 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=<object object>`)

Represents a blueprint, a collection of routes and other app-related functions that can be registered on a real application later.

A blueprint is an object that allows defining application functions without requiring an application object ahead of time. It uses the same decorators as *Flask*, but defers the need for an application by recording them for later registration.

Decorating a function with a blueprint creates a deferred function that is called with *BlueprintSetupState* when the blueprint is registered on an application.

See *Modular Applications with Blueprints* for more information.

Parameters

- **name** (*str*) – The name of the blueprint. Will be prepended to each endpoint name.
- **import_name** (*str*) – The name of the blueprint package, usually `__name__`. This helps locate the `root_path` for the blueprint.
- **static_folder** (*str* | *os.PathLike* | *None*) – A folder with static files that should be served by the blueprint’s static route. The path is relative to the blueprint’s root path. Blueprint static files are disabled by default.
- **static_url_path** (*str* | *None*) – The url to serve static files from. Defaults to `static_folder`. If the blueprint does not have a `url_prefix`, the app’s static route will take precedence, and the blueprint’s static files won’t be accessible.
- **template_folder** (*str* | *os.PathLike* | *None*) – A folder with templates that should be added to the app’s template search path. The path is relative to the blueprint’s root path. Blueprint templates are disabled by default. Blueprint templates have a lower precedence than those in the app’s templates folder.
- **url_prefix** (*str* | *None*) – A path to prepend to all of the blueprint’s URLs, to make them distinct from the rest of the app’s routes.
- **subdomain** (*str* | *None*) – A subdomain that blueprint routes will match on by default.
- **url_defaults** (*dict* | *None*) – A dict of default values that blueprint routes will receive by default.
- **root_path** (*str* | *None*) – By default, the blueprint will automatically set this based on `import_name`. In certain situations this automatic detection can fail, so the path can be specified manually instead.
- **cli_group** (*str* | *None*) –

Changed in version 1.1.0: Blueprints have a `cli_group` to register nested CLI commands. The `cli_group` parameter controls the name of the group under the `flask` command.

New in version 0.7.

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()`.

New in version 0.10.

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()`.

New in version 0.10.

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** (*flask.blueprints.T_after_request*) –

Return type *flask.blueprints.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 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` (`flask.scaffold.T_after_request`) –

Return type `flask.scaffold.T_after_request`

after_request_funcs: `dict[ft.AppOrBlueprintKey, list[ft.AfterRequestCallable]]`

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 the functions 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.

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` (`flask.blueprints.T_template_context_processor`) –

Return type `flask.blueprints.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[[flask.blueprints.T_error_handler], flask.blueprints.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[[flask.blueprints.T_template_filter], flask.blueprints.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()`.

New in version 0.10.

Parameters `name` (`str | None`) – the optional name of the global, otherwise the function name will be used.

Return type `Callable[[flask.blueprints.T_template_global], flask.blueprints.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()`.

New in version 0.10.

Parameters `name` (`str | None`) – the optional name of the test, otherwise the function name will be used.

Return type *Callable*[[flask.blueprints.T_template_test], flask.blueprints.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* (*flask.blueprints.T_url_defaults*) –

Return type flask.blueprints.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* (*flask.blueprints.T_url_value_preprocessor*) –

Return type flask.blueprints.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* (*flask.blueprints.T_before_request*) –

Return type flask.blueprints.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 and execute before every request, use *Blueprint.before_app_request()*.

Parameters *f* (*flask.scaffold.T_before_request*) –

Return type flask.scaffold.T_before_request

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.

cli

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.

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* (`flask.scaffold.T_template_context_processor`) –

Return type `flask.scaffold.T_template_context_processor`

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

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

New in version 2.0.

Parameters

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

Return type `Callable[[flask.scaffold.T_route], flask.scaffold.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[[flask.scaffold.F], flask.scaffold.F]`

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.

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()`.

New in version 0.7: Use `register_error_handler()` instead of modifying `error_handler_spec` directly, for application wide error handlers.

New in version 0.7: One can now additionally also register custom exception types that do not necessarily have to be a subclass of the `HTTPException` class.

Parameters `code_or_exception` (`type[Exception]` | `int`) – the code as integer for the handler, or an arbitrary exception

Return type `Callable`[[`flask.scaffold.T_error_handler`], `flask.scaffold.T_error_handler`]

get(`rule`, `**options`)

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

New in version 2.0.

Parameters

- **rule** (`str`) –
- **options** (`Any`) –

Return type `Callable`[[`flask.scaffold.T_route`], `flask.scaffold.T_route`]

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.

Changed in version 2.0: The default configuration is `None` instead of 12 hours.

New in version 0.9.

Parameters `filename` (`str` | `None`) –

Return type `int` | `None`

property `has_static_folder`: `bool`

True if `static_folder` is set.

New in version 0.5.

import_name

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

property `jinj_loader`: `jinj2.loaders.FileSystemLoader` | `None`

The Jinja loader for this object's templates. By default this is a class `jinj2.loaders.FileSystemLoader` to `template_folder` if it is set.

New in version 0.5.

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

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

Parameters

- **app** (`Flask`) –
- **options** (`dict`) –
- **first_registration** (`bool`) –

Return type `BlueprintSetupState`

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`

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

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

New in version 2.0.

Parameters

- **rule** (`str`) –
- **options** (`Any`) –

Return type `Callable[[flask.scaffold.T_route], flask.scaffold.T_route]`

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

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

New in version 2.0.

Parameters

- **rule** (`str`) –
- **options** (`Any`) –

Return type `Callable[[flask.scaffold.T_route], flask.scaffold.T_route]`

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

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

New in version 2.0.

Parameters

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

Return type *Callable*[[flask.scaffold.T_route], flask.scaffold.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*) –

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*) –

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** (*Flask*) – The application this blueprint is being registered with.
- **options** (*dict*) – Keyword arguments forwarded from *register_blueprint()*.

Return type None

Changed in version 2.3: Nested blueprints now correctly apply subdomains.

Changed in version 2.1: Registering the same blueprint with the same name multiple times is an error.

Changed in version 2.0.1: Nested blueprints are registered with their dotted name. This allows different blueprints with the same name to be nested at different locations.

Changed in version 2.0.1: The *name* option can be used to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for *url_for*.

register_blueprint(*blueprint, **options*)

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

Changed in version 2.0.1: The *name* option can be used to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for *url_for*.

New in version 2.0.

Parameters

- **blueprint** (*flask.blueprints.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.

New in version 0.7.

Parameters

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

Return type None

root_path

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

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*[[`flask.scaffold.T_route`], `flask.scaffold.T_route`]

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.

New in version 0.5.

Parameters **filename** (*str*) –

Return type *Response*

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** (*flask.blueprints.T_teardown*) –

Return type flask.blueprints.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 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* (*flask.scaffold.T_teardown*) –

Return type flask.scaffold.T_teardown

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.

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.

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.

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* (`flask.scaffold.T_url_defaults`) –

Return type `flask.scaffold.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* (`flask.scaffold.T_url_value_preprocessor`) –

Return type `flask.scaffold.T_url_value_preprocessor`

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.

view_functions: `dict[str, t.Callable]`

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.

2.1.3 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*) –

Return type None

property `accept_charsets`: `werkzeug.datastructures.accept.CharsetAccept`

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

property `accept_encodings`: `werkzeug.datastructures.accept.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`: `werkzeug.datastructures.accept.LanguageAccept`

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

property `accept_mimetypes`: `werkzeug.datastructures.accept.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`: `werkzeug.datastructures.structures.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.

Changed in version 2.3: Invalid bytes remain percent encoded.

property authorization: `werkzeug.datastructures.auth.Authorization` | `None`

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

Changed in version 2.3: Authorization is no longer a dict. The token attribute was added for auth schemes that use a token instead of parameters.

property base_url: `str`

Like `url` but without the query string.

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.

New in version 2.0.1.

property cache_control: `werkzeug.datastructures.cache_control.RequestCacheControl`

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

property charset: `str`

The charset used to decode body, form, and cookie data. Defaults to UTF-8.

Deprecated since version 2.3: Will be removed in Werkzeug 3.0. Request data must always be UTF-8.

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.

New in version 0.9.

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.

New in version 0.9.

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.)

New in version 0.9.

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: `werkzeug.datastructures.structures.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.

Changed in version 2.0: The datetime object is timezone-aware.

dict_storage_class

alias of `werkzeug.datastructures.structures.ImmutableMultiDict`

property encoding_errors: `str`

How errors when decoding bytes are handled. Defaults to "replace".

Deprecated since version 2.3: Will be removed in Werkzeug 3.0.

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.

environ: `WSGIEnvironment`

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

property files: `werkzeug.datastructures.structures.ImmutableMultiDict[str, werkzeug.datastructures.file_storage.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: `werkzeug.datastructures.structures.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 important.

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

Changed in version 0.9: Previous to Werkzeug 0.9 this would only contain form data for POST and PUT requests.

form_data_parser_class

alias of `werkzeug.formparser.FormDataParser`

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

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

This accepts the same options as the `EnvironBuilder`.

Changed in version 0.5: This method now accepts the same arguments as `EnvironBuilder`. Because of this the `environ` parameter is now called `environ_overrides`.

Returns request object

Parameters

- `args` (*Any*) –
- `kwargs` (*Any*) –

Return type `werkzeug.wrappers.request.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 whole data is cached (which is the default) the form parser will use 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.

New in version 0.9.

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 *Optional[Any]*

Changed in version 2.3: Raise a 415 error instead of 400.

Changed in version 2.1: Raise a 400 error if the content type is incorrect.

headers

The headers received with the request.

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: **werkzeug.datastructures.etag.ETags**

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

Return type *ETags*

property if_modified_since: **datetime.datetime | None**

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

Changed in version 2.0: The datetime object is timezone-aware.

property if_none_match: **werkzeug.datastructures.etag.ETags**

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

Return type *ETags*

property if_range: **werkzeug.datastructures.range.IfRange**

The parsed *If-Range* header.

Changed in version 2.0: `IfRange.date` is timezone-aware.

New in version 0.7.

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

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

Changed in version 2.0: The datetime object is timezone-aware.

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: Optional[Any]

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.

Changed in version 2.3: Raise a 415 error instead of 400.

Changed in version 2.1: Raise a 400 error if the content type is incorrect.

list_storage_class

alias of `werkzeug.datastructures.structures.ImmutableList`

make_form_data_parser()

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

New in version 0.8.

Return type `werkzeug.formparser.FormDataParser`

property max_content_length: int | None

Read-only view of the MAX_CONTENT_LENGTH config key.

max_forwards

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

method

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

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'}*.

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*

Changed in version 2.3: Raise a 415 error instead of 400.

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 `werkzeug.datastructures.structures.ImmutableMultiDict`

path

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

property pragma: `werkzeug.datastructures.structures.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.

query_string

The part of the URL after the “?”. This is the raw value, use `args` for the parsed values.

property range: `werkzeug.datastructures.range.Range` | `None`

The parsed *Range* header.

New in version 0.7.

Return type `Range`

referrer

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

remote_addr

The address of the client sending the request.

remote_user

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

root_path

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

property root_url: `str`

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

routing_exception: `Exception` | `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.

scheme

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

property script_root: `str`

Alias for `self.root_path`. `environ["SCRIPT_ROOT"]` without a trailing slash.

server

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

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.

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.

Changed in version 2.3: Check `max_content_length` preemptively and while reading.

Changed in version 0.9: The stream is always set (but may be consumed) even if form parsing was accessed first.

property url: `str`

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

property url_charset: `str`

The charset to use when decoding percent-encoded bytes in `args`. Defaults to the value of `charset`, which defaults to UTF-8.

Deprecated since version 2.3: Will be removed in Werkzeug 3.0. Percent-encoded bytes must always be UTF-8.

New in version 0.6.

property url_root: `str`

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

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.

New in version 0.6.

property user_agent: `werkzeug.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.

Changed in version 2.1: The built-in parser was removed. Set `user_agent_class` to a `UserAgent` subclass to parse data from the string.

user_agent_class

alias of `werkzeug.user_agent.UserAgent`

property values: `werkzeug.datastructures.structures.CombinedMultiDict[str, str]`

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

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

Changed in version 2.0: For GET requests, only *args* are present, not *form*.

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`.

property want_form_data_parsed: `bool`

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

New in version 0.8.

`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*.

2.1.4 Response Objects

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

The response object that is used by default in Flask. Works like the response object from Werkzeug but is set to have an HTML mimetype by default. Quite often you don't have to create this object yourself because *make_response()* will take care of that for you.

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

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

Changed in version 1.0: Added *max_cookie_size*.

Parameters

- **response** (`Union[Iterable[str], Iterable[bytes]]`) –
- **status** (`int | str | HTTPStatus | None`) –
- **headers** (`werkzeug.datastructures.headers.Headers`) –
- **mimetype** (`str | None`) –
- **content_type** (`str | None`) –
- **direct_passthrough** (`bool`) –

Return type `None`

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.

New in version 0.7.

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.

Changed in version 2.0: SHA-1 is used to generate the value. MD5 may not be available in some environments.

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: `werkzeug.datastructures.structures.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.

property cache_control: `werkzeug.datastructures.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.

New in version 0.6.

Parameters **func** (*Callable[[], Any]*) –

Return type *Callable[[], Any]*

property charset: `str`

The charset used to encode body and cookie data. Defaults to UTF-8.

Deprecated since version 2.3: Will be removed in Werkzeug 3.0. Response data must always be UTF-8.

close()

Close the wrapped response if possible. You can also use the object in a with statement which will automatically close it.

New in version 0.9: Can now be used in a with statement.

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: `werkzeug.datastructures.structures.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: `werkzeug.datastructures.range.ContentRange`

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

New in version 0.7.

property content_security_policy: `werkzeug.datastructures.csp.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:
`werkzeug.datastructures.csp.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.

Changed in version 2.0: The datetime object is timezone-aware.

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 deleted.
- **path** (*str* / *None*) – if the cookie that should be deleted was limited to a path, the path has to be defined here.
- **domain** (*Optional[str]*) – 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** (*Optional[str]*) – Limit the scope of the cookie to only be attached to requests that are “same-site”.

Return type None**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.

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.

Changed in version 2.0: The datetime object is timezone-aware.

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 *implicit_sequence_conversion* and *direct_passthrough*.
- Set the Content-Length header.
- Generate an ETag header if one is not already set.

Changed in version 2.1: Removed the *no_etag* parameter.

Changed in version 2.0: An ETag header is always added.

Changed in version 0.6: The Content-Length header is set.

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.

New in version 0.6.

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 unwanted behavior 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.

New in version 0.9.

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 `Optional[Any]`

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.

Changed in version 0.6: Previously that function was called *fix_headers* and modified the response object in place. Also since 0.6, IRIs in location and content-location headers are handled properly.

Also starting with 0.6, Werkzeug will attempt to set the content length if it is able to figure it out on its own. This is the case if all the strings in the response iterable are already encoded and the iterable is buffered.

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.

New in version 0.6.

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]]]

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.

New in version 0.6.

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 method is used as application iterator unless *direct_passthrough* was activated.

Return type *Iterator*[bytes]

property json: Optional[Any]

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.

Changed in version 2.0: The datetime object is timezone-aware.

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 dictates the value of *Accept-Ranges* header. If `False` (default), 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*

Changed in version 2.0: Range processing is skipped if length is 0 instead of raising a 416 Range Not Satisfiable error.

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.

New in version 0.6.

Return type `None`

property max_cookie_size: int

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

See `max_cookie_size` in Werkzeug's docs.

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'}`.

New in version 0.5.

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.

property retry_after: datetime.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.

Changed in version 2.0: The datetime object is timezone-aware.

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** (*Optional[Union[datetime.timedelta, int]]*) – should be a number of seconds, or *None* (default) if the cookie should last only as long as the client’s browser session.
- **expires** (*Optional[Union[str, datetime.datetime, int, float]]*) – 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** (*Optional[str]*) – 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** (*Optional[str]*) – 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).

New in version 0.9.

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**

The HTTP status code as a string.

property status_code: **int**

The HTTP status code as a number.

property stream: `werkzeug.wrappers.response.ResponseStream`

The response iterable as write-only stream.

property vary: `werkzeug.datastructures.structures.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: `werkzeug.datastructures.auth.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`.

Changed in version 2.3: This attribute can be assigned to to set the header. A list can be assigned to set multiple header values. Use `del` to unset the header.

Changed in version 2.3: `WWWAuthenticate` is no longer a `dict`. The `token` attribute was added for auth challenges that use a token instead of parameters.

2.1.5 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)
```

(continues on next page)

(continued from previous page)

```
# 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.

2.1.6 Session Interface

New in version 0.8.

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

class flask.sessions.SessionInterface

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

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

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

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

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

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

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.

New in version 0.8.

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.

Changed in version 2.3: Not set by default, does not fall back to `SERVER_NAME`.

Parameters `app` (`Flask`) –

Return type `str` | `None`

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_name(`app`)

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

Parameters `app` (`Flask`) –

Return type `str`

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_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`

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_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`

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`

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`

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 `flask.sessions.NullSession`

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`

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.

New in version 0.10.

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`

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.

New in version 0.11.

Parameters

- `app` (`Flask`) –
- `session` (`SessionMixin`) –

Return type `bool`

class flask.sessions.SecureCookieSessionInterface

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

static digest_method(*string=b"*, *, *usedforsecurity=True*)

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

key_derivation = `'hmac'`

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

open_session(*app, request*)

This 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`

salt = `'cookie-session'`

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

save_session(*app, session, response*)

This is called 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`

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

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

session_class

alias of [flask.sessions.SecureCookieSession](#)

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

Base class for sessions based on signed cookies.

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

Parameters **initial** (*t.Any*) –

Return type `None`

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** (*Optional[Any]*) –

Return type *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.

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** (*Optional[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*) –

Return type *None*

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*

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*

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*

class flask.sessions.**SessionMixin**

Expands a basic dictionary with session attributes.

accessed = True

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

modified = True

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

property permanent: bool

This reflects the '_permanent' key in the dict.

Notice

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

2.1.7 Test Client

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

Works like a regular Werkzeug test client but has knowledge about Flask's contexts 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`.

Changed in version 0.12: `app.test_client()` includes preset default environment, which can be set after instantiation of the `app.test_client()` object in `client.environ_base`.

Basic usage is outlined in the *Testing Flask Applications* chapter.

Parameters

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

Return type None**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`

Changed in version 2.1: Removed the `as_tuple` parameter.

Changed in version 2.0: The request input stream is closed when calling `response.close()`. Input streams for redirects are automatically closed.

Changed in version 0.5: If a dict is provided as file in the dict for the `data` parameter the content type has to be called `content_type` instead of `mimetype`. This change was made for consistency with `werkzeug.FileWrapper`.

Changed in version 0.5: Added the `follow_redirects` parameter.

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 `Generator[flask.sessions.SessionMixin, None, None]`

2.1.8 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`) –

Return type `None`

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** (`Optional[Any]`) – Command object to invoke. Default is the app's `cli` group.
- **args** (`Optional[Any]`) – List of strings to invoke the command with.
- **kwargs** (`Any`) –

Returns a `Result` object.

Return type `Any`

2.1.9 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.

Changed in version 0.10: Bound to the application context instead of the request context.

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.

New in version 0.10.

iter(g)

Return an iterator over the attribute names.

New in version 0.10.

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** (*Optional [Any]*) – Value to return if the attribute is not present.

Return type *Any*

New in version 0.10.

pop(name, default=<object object>)

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*

New in version 0.11.

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** (*Optional [Any]*) – Value to set and return if the attribute is not present.

Return type *Any*

New in version 0.11.

2.1.10 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
```

New in version 0.7.

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 function.
        ...
    gevent.spawn(do_some_work)
    return 'Regular response'
```

New in version 0.10.

Parameters `f` (*Callable*) –

Return type *Callable*

`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.

New in version 0.9.

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** (*Optional[str]*) – If given, append this as `#anchor` to the URL.
- **_method** (*Optional[str]*) – If given, generate the URL associated with this method for the endpoint.
- **_scheme** (*Optional[str]*) – If given, the URL will have this scheme if it is external.
- **_external** (*Optional[bool]*) – 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*

Changed in version 2.2: Calls `current_app.url_for`, allowing an app to override the behavior.

Changed in version 0.10: The `_scheme` parameter was added.

Changed in version 0.9: The `_anchor` and `_method` parameters were added.

Changed in version 0.9: Calls `app.handle_url_build_error` on build errors.

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

Return type *t.NoReturn*

New in version 2.2: Calls `current_app.aborter` if available instead of always using Werkzeug's default `abort`.

`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*[*BaseResponse*] / *None*) – The response class to use. Not used when `current_app` is active, which uses `app.response_class`.

Return type `BaseResponse`

New in version 2.2: Calls `current_app.redirect` if available instead of always using Werkzeug's default `redirect`.

`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.

New in version 0.6.

Parameters `args` (*t.Any*) –

Return type `Response`

`flask.after_this_request(f)`

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

Example:

```

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

```

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

New in version 0.9.

Parameters *f* (*Union*[*Callable*[[*flask.typing.ResponseClass*], *flask.typing.ResponseClass*], *Callable*[[*flask.typing.ResponseClass*], *Awaitable*[*flask.typing.ResponseClass*]]) –

Return type *Union*[*Callable*[[*flask.typing.ResponseClass*], *flask.typing.ResponseClass*], *Callable*[[*flask.typing.ResponseClass*], *Awaitable*[*flask.typing.ResponseClass*]]

`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* | *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*

Changed in version 2.0: `download_name` replaces the `attachment_filename` parameter. If `as_attachment=False`, it is passed with `Content-Disposition: inline` instead.

Changed in version 2.0: `max_age` replaces the `cache_timeout` parameter. `conditional` is enabled and `max_age` is not set by default.

Changed in version 2.0: `etag` replaces the `add_etags` parameter. It can be a string to use instead of generating one.

Changed in version 2.0: Passing a file-like object that inherits from `TextIOBase` will raise a `ValueError` rather than sending an empty file.

New in version 2.0: Moved the implementation to Werkzeug. This is now a wrapper to pass some Flask-specific arguments.

Changed in version 1.1: `filename` may be a `PathLike` object.

Changed in version 1.1: Passing a `BytesIO` object supports range requests.

Changed in version 1.0.3: Filenames are encoded with ASCII instead of Latin-1 for broader compatibility with WSGI servers.

Changed in version 1.0: UTF-8 filenames as specified in [RFC 2231](#) are supported.

Changed in version 0.12: The filename is no longer automatically inferred from file objects. If you want to use automatic MIME and etag support, pass a filename via `filename_or_fp` or `attachment_filename`.

Changed in version 0.12: `attachment_filename` is preferred over `filename` for MIME detection.

Changed in version 0.9: `cache_timeout` defaults to `Flask.get_send_file_max_age()`.

Changed in version 0.7: MIME guessing and etag support for file-like objects was deprecated because it was unreliable. Pass a filename if you are able to, otherwise attach an etag yourself.

Changed in version 0.5: The `add_etags`, `cache_timeout` and `conditional` parameters were added. The default behavior is to add etags.

New in version 0.2.

`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*) – The directory that `path` must be located under, relative to the current application's root path.

- **path** (*os.PathLike* | *str*) – The path to the file to send, relative to `directory`.
- **kwargs** (*t.Any*) – Arguments to pass to `send_file()`.

Return type *Response*

Changed in version 2.0: `path` replaces the `filename` parameter.

New in version 2.0: Moved the implementation to Werkzeug. This is now a wrapper to pass some Flask-specific arguments.

New in version 0.5.

2.1.11 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()`.

Changed in version 0.3: `category` parameter added.

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.

Changed in version 0.9: `category_filter` parameter added.

Changed in version 0.3: `with_categories` parameter added.

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]]`

2.1.12 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*

Changed in version 2.2: Calls `current_app.json.response`, allowing an app to override the behavior.

Changed in version 2.0.2: `decimal.Decimal` is supported by converting to a string.

Changed in version 0.11: Added support for serializing top-level arrays. This was a security risk in ancient browsers. See [JSON Security](#).

New in version 0.2.

`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*

Changed in version 2.3: The `app` parameter was removed.

Changed in version 2.2: Calls `current_app.json.dumps`, allowing an app to override the behavior.

Changed in version 2.0.2: `decimal.Decimal` is supported by converting to a string.

Changed in version 2.0: `encoding` will be removed in Flask 2.1.

Changed in version 1.0.3: `app` can be passed directly, rather than requiring an app context for configuration.

`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

Changed in version 2.3: The `app` parameter was removed.

Changed in version 2.2: Calls `current_app.json.dump`, allowing an app to override the behavior.

Changed in version 2.0: Writing to a binary file, and the `encoding` argument, will be removed in Flask 2.1.

`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*

Changed in version 2.3: The `app` parameter was removed.

Changed in version 2.2: Calls `current_app.json.loads`, allowing an app to override the behavior.

Changed in version 2.0: `encoding` will be removed in Flask 2.1. The data must be a string or UTF-8 bytes.

Changed in version 1.0.3: `app` can be passed directly, rather than requiring an app context for configuration.

`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*

Changed in version 2.3: The `app` parameter was removed.

Changed in version 2.2: Calls `current_app.json.load`, allowing an app to override the behavior.

Changed in version 2.2: The `app` parameter will be removed in Flask 2.3.

Changed in version 2.0: `encoding` will be removed in Flask 2.1. The file must be text mode, or binary mode with UTF-8 bytes.

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 `dump()` 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` (Flask) – An application instance. This will be stored as a `weakref.proxy` on the `_app` attribute.

Return type None

New in version 2.2.

dump(*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 underlying JSON library.

Return type *Any*

response(*args, **kwargs)

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

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** (*Flask*) –

Return type `None`

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 types. [SecureCookieSessionInterface](#) uses this to serialize the session data, but it may be useful in other places. It can be extended to support other types.

class `flask.json.tag.TaggedJSONSerializer`

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

The following extra types are supported:

- `dict`
- `tuple`
- `bytes`
- Markup
- UUID
- `datetime`

Return type `None`

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

Tag classes to bind when creating the serializer. Other tags can be added later using [register\(\)](#).

dumps(*value*)

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

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*

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

Register a new tag with this serializer.

Parameters

- **tag_class** (*type*[*flask.json.tag.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** (*Optional*[*int*]) – 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 *dict*[*str*, *Any*]

untag(*value*)

Convert a tagged representation back to the original type.

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

Return type *Any*

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

Base class for defining type tags for *TaggedJSONSerializer*.

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

Return type *None*

check(*value*)

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

Parameters **value** (*Any*) –

Return type *bool*

key: *str* | *None* = *None*

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

tag(*value*)

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

Parameters **value** (*Any*) –

Return type *Any*

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*

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)
```

2.1.13 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* | *jinj2.environment.Template* | *list[str | jinj2.environment.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* | *jinjja2.environment.Template* | *list[str | jinjja2.environment.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]*

New in version 2.2.

`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]*

New in version 2.2.

`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')
```

New in version 0.2.

Parameters

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

Return type *Any*

2.1.14 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*) – path to which files are read relative from. When the config object is created by the application, this is the application's *root_path*.
- **defaults** (*dict* | *None*) – an optional dictionary of default values

Return type None

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_file(*filename*, *load*, *silent=False*, *text=True*)

Update the values in the config from a file that is loaded using the *load* parameter. The loaded data is passed to the *from_mapping()* method.

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

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

Parameters

- **filename** (*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*

Changed in version 2.3: The *text* parameter was added.

New in version 2.0.

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

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

Returns Always returns True.

Parameters

- **mapping** (*Optional[Mapping[str, Any]]*) –
- **kwargs** (*Any*) –

Return type *bool*

New in version 0.11.

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_prefixed_env(*prefix*='FLASK', *, *loads*=<function 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`

New in version 2.1.

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*) – 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`

New in version 0.7: *silent* parameter.

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]`

New in version 0.11.

2.1.15 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()))
```

New in version 0.9.

Parameters `generator_or_function` (*Union[Iterator, Callable[[...], Iterator]*)

–

Return type *Iterator*

2.1.16 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** (*dict*) –
- **request** (*Request | None*) –
- **session** (*SessionMixin | None*) –

Return type *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.

Changed in version 1.1: The current session object is used instead of reloading the original data. This prevents `flask.session` pointing to an out-of-date object.

New in version 0.10.

Return type *flask.ctx.RequestContext*

`match_request()`

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

Return type *None*

`pop(exc=<object object>)`

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

Changed in version 0.9: Added the `exc` argument.

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`) –

Return type `None`

pop(*exc*=<object object>)

Pops the app context.

Parameters `exc` (`BaseException` | `None`) –

Return type `None`

push()

Binds the app context to the current context.

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. Accessing 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** (`Flask`) –
- **options** (`t.Any`) –
- **first_registration** (`bool`) –

Return type `None`

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** (`Optional[str]`) –
- **view_func** (`Optional[Callable]`) –
- **options** (`Any`) –

Return type `None`

app

a reference to the current application

blueprint

a reference to the blueprint that created this setup state.

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.

options

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

subdomain

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

url_defaults

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

url_prefix

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

2.1.17 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()
```

(continues on next page)

(continued from previous page)

```
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'
```

New in version 0.10.

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.

New in version 0.10.

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)
```

New in version 0.10.

signals.signals_available

Deprecated since version 2.3: Will be removed in Flask 2.4. Signals are always available

2.1.18 Class-Based Views

New in version 0.7.

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*.

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.

Changed in version 2.2: Added the *init_every_request* class attribute.

Parameters

- **name** (*str*) –
- **class_args** (*t.Any*) –
- **class_kwargs** (*t.Any*) –

Return type `ft.RouteCallable`

decorators: `ClassVar[list[Callable]] = []`

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.

New in version 0.8.

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`

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.

New in version 2.2.

methods: `ClassVar[Optional[Collection[str]]] = 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.

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

2.1.19 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:

<i>string</i>	accepts any text without a slash (the default)
<i>int</i>	accepts integers
<i>float</i>	like <i>int</i> but for floating point values
<i>path</i>	like the default but also accepts slashes
<i>any</i>	matches one of the items provided
<i>uuid</i>	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.

<i>rule</i>	the URL rule as string
<i>end-point</i>	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.
<i>view_func</i>	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 <code>view_functions</code> dictionary with the endpoint as key.
<i>defaults</i>	A dictionary with defaults for this rule. See the example above for how defaults work.
<i>sub-domain</i>	specifies the rule for the subdomain in case subdomain matching is in use. If not specified the default subdomain is assumed.
<i>**options</i>	the options to be forwarded to the underlying <code>Rule</code> object. A change to Werkzeug is handling of method options. <code>methods</code> 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, <code>OPTIONS</code> is implicitly added and handled by the standard request handling. They have to be specified as keyword arguments.

2.1.20 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)
```

New in version 0.8: The `provide_automatic_options` functionality was added.

2.1.21 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*) –

Return type None

Changed in version 2.2: Added the `-A/--app`, `--debug/--no-debug`, `-e/--env-file` options.

Changed in version 2.2: An app context is pushed when running `app.cli` commands, so `@with_appcontext` is no longer required for those commands.

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

get_command(*ctx, name*)

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

list_commands(*ctx*)

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

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 it's the name of the command.
- **args** (*list[str]*) – the arguments to parse as list of strings.
- **parent** (*Optional[click.core.Context]*) – the parent context if available.
- **extra** (*Any*) – extra keyword arguments forwarded to the context constructor.

Return type `click.core.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** (*click.core.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** (*Optional[str]*) –
- **commands** (*Optional[Union[Dict[str, click.core.Command], Sequence[click.core.Command]]]*) –
- **attrs** (*Any*) –

Return type `None`

command(**args*, ***kwargs*)

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

group(**args*, ***kwargs*)

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

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

Return type *None*

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** (*Optional[Union[str, os.PathLike]]*) – Load the file at this location instead of searching.

Returns True if a file was loaded.

Return type *bool*

Changed in version 2.0: The current directory is not changed to the location of the loaded file.

Changed in version 2.0: When loading the env files, set the default encoding to UTF-8.

Changed in version 1.1.0: Returns *False* when *python-dotenv* is not installed, or when the given path isn't a file.

New in version 1.0.

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.

Changed in version 2.2: The app context is active for subcommands as well as the decorated callback. The app context is always available to *app.cli* command and parameter callbacks.

`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` (*click.decorators.F*) –

Return type `click.decorators.F`

`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*

ADDITIONAL NOTES

3.1 Design Decisions in Flask

If you are curious why Flask does certain things the way it does and not differently, this section is for you. This should give you an idea about some of the design decisions that may appear arbitrary and surprising at first, especially in direct comparison with other frameworks.

3.1.1 The Explicit Application Object

A Python web application based on WSGI has to have one central callable object that implements the actual application. In Flask this is an instance of the *Flask* class. Each Flask application has to create an instance of this class itself and pass it the name of the module, but why can't Flask do that itself?

Without such an explicit application object the following code:

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

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

Would look like this instead:

```
from hypothetical_flask import route

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

There are three major reasons for this. The most important one is that implicit application objects require that there may only be one instance at the time. There are ways to fake multiple applications with a single application object, like maintaining a stack of applications, but this causes some problems I won't outline here in detail. Now the question is: when does a microframework need more than one application at the same time? A good example for this is unit testing. When you want to test something it can be very helpful to create a minimal application to test specific behavior. When the application object is deleted everything it allocated will be freed again.

Another thing that becomes possible when you have an explicit object lying around in your code is that you can subclass the base class (*Flask*) to alter specific behavior. This would not be possible without hacks if the object were created ahead of time for you based on a class that is not exposed to you.

But there is another very important reason why Flask depends on an explicit instantiation of that class: the package name. Whenever you create a Flask instance you usually pass it `__name__` as package name. Flask depends on that information to properly load resources relative to your module. With Python's outstanding support for reflection it can then access the package to figure out where the templates and static files are stored (see [open_resource\(\)](#)). Now obviously there are frameworks around that do not need any configuration and will still be able to load templates relative to your application module. But they have to use the current working directory for that, which is a very unreliable way to determine where the application is. The current working directory is process-wide and if you are running multiple applications in one process (which could happen in a webserver without you knowing) the paths will be off. Worse: many web servers do not set the working directory to the directory of your application but to the document root which does not have to be the same folder.

The third reason is “explicit is better than implicit”. That object is your WSGI application, you don't have to remember anything else. If you want to apply a WSGI middleware, just wrap it and you're done (though there are better ways to do that so that you do not lose the reference to the application object `wsgi_app()`).

Furthermore this design makes it possible to use a factory function to create the application which is very helpful for unit testing and similar things (*Application Factories*).

3.1.2 The Routing System

Flask uses the Werkzeug routing system which was designed to automatically order routes by complexity. This means that you can declare routes in arbitrary order and they will still work as expected. This is a requirement if you want to properly implement decorator based routing since decorators could be fired in undefined order when the application is split into multiple modules.

Another design decision with the Werkzeug routing system is that routes in Werkzeug try to ensure that URLs are unique. Werkzeug will go quite far with that in that it will automatically redirect to a canonical URL if a route is ambiguous.

3.1.3 One Template Engine

Flask decides on one template engine: Jinja2. Why doesn't Flask have a pluggable template engine interface? You can obviously use a different template engine, but Flask will still configure Jinja2 for you. While that limitation that Jinja2 is *always* configured will probably go away, the decision to bundle one template engine and use that will not.

Template engines are like programming languages and each of those engines has a certain understanding about how things work. On the surface they all work the same: you tell the engine to evaluate a template with a set of variables and take the return value as string.

But that's about where similarities end. Jinja2 for example has an extensive filter system, a certain way to do template inheritance, support for reusable blocks (macros) that can be used from inside templates and also from Python code, supports iterative template rendering, configurable syntax and more. On the other hand an engine like Genshi is based on XML stream evaluation, template inheritance by taking the availability of XPath into account and more. Mako on the other hand treats templates similar to Python modules.

When it comes to connecting a template engine with an application or framework there is more than just rendering templates. For instance, Flask uses Jinja2's extensive autoescaping support. Also it provides ways to access macros from Jinja2 templates.

A template abstraction layer that would not take the unique features of the template engines away is a science on its own and a too large undertaking for a microframework like Flask.

Furthermore extensions can then easily depend on one template language being present. You can easily use your own templating language, but an extension could still depend on Jinja itself.

3.1.4 What does “micro” mean?

“Micro” does not mean that your whole web application has to fit into a single Python file (although it certainly can), nor does it mean that Flask is lacking in functionality. The “micro” in microframework means Flask aims to keep the core simple but extensible. Flask won’t make many decisions for you, such as what database to use. Those decisions that it does make, such as what templating engine to use, are easy to change. Everything else is up to you, so that Flask can be everything you need and nothing you don’t.

By default, Flask does not include a database abstraction layer, form validation or anything else where different libraries already exist that can handle that. Instead, Flask supports extensions to add such functionality to your application as if it was implemented in Flask itself. Numerous extensions provide database integration, form validation, upload handling, various open authentication technologies, and more. Flask may be “micro”, but it’s ready for production use on a variety of needs.

Why does Flask call itself a microframework and yet it depends on two libraries (namely Werkzeug and Jinja2). Why shouldn’t it? If we look over to the Ruby side of web development there we have a protocol very similar to WSGI. Just that it’s called Rack there, but besides that it looks very much like a WSGI rendition for Ruby. But nearly all applications in Ruby land do not work with Rack directly, but on top of a library with the same name. This Rack library has two equivalents in Python: WebOb (formerly Paste) and Werkzeug. Paste is still around but from my understanding it’s sort of deprecated in favour of WebOb. The development of WebOb and Werkzeug started side by side with similar ideas in mind: be a good implementation of WSGI for other applications to take advantage.

Flask is a framework that takes advantage of the work already done by Werkzeug to properly interface WSGI (which can be a complex task at times). Thanks to recent developments in the Python package infrastructure, packages with dependencies are no longer an issue and there are very few reasons against having libraries that depend on others.

3.1.5 Thread Locals

Flask uses thread local objects (context local objects in fact, they support greenlet contexts as well) for request, session and an extra object you can put your own things on (*g*). Why is that and isn’t that a bad idea?

Yes it is usually not such a bright idea to use thread locals. They cause troubles for servers that are not based on the concept of threads and make large applications harder to maintain. However Flask is just not designed for large applications or asynchronous servers. Flask wants to make it quick and easy to write a traditional web application.

3.1.6 Async/await and ASGI support

Flask supports async coroutines for view functions by executing the coroutine on a separate thread instead of using an event loop on the main thread as an async-first (ASGI) framework would. This is necessary for Flask to remain backwards compatible with extensions and code built before async was introduced into Python. This compromise introduces a performance cost compared with the ASGI frameworks, due to the overhead of the threads.

Due to how tied to WSGI Flask’s code is, it’s not clear if it’s possible to make the Flask class support ASGI and WSGI at the same time. Work is currently being done in Werkzeug to work with ASGI, which may eventually enable support in Flask as well.

See *Using async and await* for more discussion.

3.1.7 What Flask is, What Flask is Not

Flask will never have a database layer. It will not have a form library or anything else in that direction. Flask itself just bridges to Werkzeug to implement a proper WSGI application and to Jinja2 to handle templating. It also binds to a few common standard library packages such as logging. Everything else is up for extensions.

Why is this the case? Because people have different preferences and requirements and Flask could not meet those if it would force any of this into the core. The majority of web applications will need a template engine in some sort. However not every application needs a SQL database.

As your codebase grows, you are free to make the design decisions appropriate for your project. Flask will continue to provide a very simple glue layer to the best that Python has to offer. You can implement advanced patterns in SQLAlchemy or another database tool, introduce non-relational data persistence as appropriate, and take advantage of framework-agnostic tools built for WSGI, the Python web interface.

The idea of Flask is to build a good foundation for all applications. Everything else is up to you or extensions.

3.2 Flask Extension Development

Extensions are extra packages that add functionality to a Flask application. While PyPI contains many Flask extensions, you may not find one that fits your need. If this is the case, you can create your own, and publish it for others to use as well.

This guide will show how to create a Flask extension, and some of the common patterns and requirements involved. Since extensions can do anything, this guide won't be able to cover every possibility.

The best ways to learn about extensions are to look at how other extensions you use are written, and discuss with others. Discuss your design ideas with others on our [Discord Chat](#) or [GitHub Discussions](#).

The best extensions share common patterns, so that anyone familiar with using one extension won't feel completely lost with another. This can only work if collaboration happens early.

3.2.1 Naming

A Flask extension typically has `flask` in its name as a prefix or suffix. If it wraps another library, it should include the library name as well. This makes it easy to search for extensions, and makes their purpose clearer.

A general Python packaging recommendation is that the install name from the package index and the name used in `import` statements should be related. The import name is lowercase, with words separated by underscores (`_`). The install name is either lower case or title case, with words separated by dashes (`-`). If it wraps another library, prefer using the same case as that library's name.

Here are some example install and import names:

- `Flask-Name` imported as `flask_name`
- `flask-name-lower` imported as `flask_name_lower`
- `Flask-ComboName` imported as `flask_comboname`
- `Name-Flask` imported as `name_flask`

3.2.2 The Extension Class and Initialization

All extensions will need some entry point that initializes the extension with the application. The most common pattern is to create a class that represents the extension’s configuration and behavior, with an `init_app` method to apply the extension instance to the given application instance.

```
class HelloExtension:
    def __init__(self, app=None):
        if app is not None:
            self.init_app(app)

    def init_app(self, app):
        app.before_request(...)
```

It is important that the app is not stored on the extension, don’t do `self.app = app`. The only time the extension should have direct access to an app is during `init_app`, otherwise it should use `current_app`.

This allows the extension to support the application factory pattern, avoids circular import issues when importing the extension instance elsewhere in a user’s code, and makes testing with different configurations easier.

```
hello = HelloExtension()

def create_app():
    app = Flask(__name__)
    hello.init_app(app)
    return app
```

Above, the `hello` extension instance exists independently of the application. This means that other modules in a user’s project can do `from project import hello` and use the extension in blueprints before the app exists.

The `Flask.extensions` dict can be used to store a reference to the extension on the application, or some other state specific to the application. Be aware that this is a single namespace, so use a name unique to your extension, such as the extension’s name without the “flask” prefix.

3.2.3 Adding Behavior

There are many ways that an extension can add behavior. Any setup methods that are available on the `Flask` object can be used during an extension’s `init_app` method.

A common pattern is to use `before_request()` to initialize some data or a connection at the beginning of each request, then `teardown_request()` to clean it up at the end. This can be stored on `g`, discussed more below.

A more lazy approach is to provide a method that initializes and caches the data or connection. For example, a `ext.get_db` method could create a database connection the first time it’s called, so that a view that doesn’t use the database doesn’t create a connection.

Besides doing something before and after every view, your extension might want to add some specific views as well. In this case, you could define a `Blueprint`, then call `register_blueprint()` during `init_app` to add the blueprint to the app.

3.2.4 Configuration Techniques

There can be multiple levels and sources of configuration for an extension. You should consider what parts of your extension fall into each one.

- Configuration per application instance, through `app.config` values. This is configuration that could reasonably change for each deployment of an application. A common example is a URL to an external resource, such as a database. Configuration keys should start with the extension's name so that they don't interfere with other extensions.
- Configuration per extension instance, through `__init__` arguments. This configuration usually affects how the extension is used, such that it wouldn't make sense to change it per deployment.
- Configuration per extension instance, through instance attributes and decorator methods. It might be more ergonomic to assign to `ext.value`, or use a `@ext.register` decorator to register a function, after the extension instance has been created.
- Global configuration through class attributes. Changing a class attribute like `Ext.connection_class` can customize default behavior without making a subclass. This could be combined per-extension configuration to override defaults.
- Subclassing and overriding methods and attributes. Making the API of the extension itself something that can be overridden provides a very powerful tool for advanced customization.

The *Flask* object itself uses all of these techniques.

It's up to you to decide what configuration is appropriate for your extension, based on what you need and what you want to support.

Configuration should not be changed after the application setup phase is complete and the server begins handling requests. Configuration is global, any changes to it are not guaranteed to be visible to other workers.

3.2.5 Data During a Request

When writing a Flask application, the `g` object is used to store information during a request. For example the *tutorial* stores a connection to a SQLite database as `g.db`. Extensions can also use this, with some care. Since `g` is a single global namespace, extensions must use unique names that won't collide with user data. For example, use the extension name as a prefix, or as a namespace.

```
# an internal prefix with the extension name
g._hello_user_id = 2

# or an internal prefix as a namespace
from types import SimpleNamespace
g._hello = SimpleNamespace()
g._hello.user_id = 2
```

The data in `g` lasts for an application context. An application context is active when a request context is, or when a CLI command is run. If you're storing something that should be closed, use `teardown_appcontext()` to ensure that it gets closed when the application context ends. If it should only be valid during a request, or would not be used in the CLI outside a request, use `teardown_request()`.

3.2.6 Views and Models

Your extension views might want to interact with specific models in your database, or some other extension or data connected to your application. For example, let's consider a Flask-SimpleBlog extension that works with Flask-SQLAlchemy to provide a Post model and views to write and read posts.

The Post model needs to subclass the Flask-SQLAlchemy `db.Model` object, but that's only available once you've created an instance of that extension, not when your extension is defining its views. So how can the view code, defined before the model exists, access the model?

One method could be to use *Class-based Views*. During `__init__`, create the model, then create the views by passing the model to the view class's `as_view()` method.

```
class PostAPI(MethodView):
    def __init__(self, model):
        self.model = model

    def get(self, id):
        post = self.model.query.get(id)
        return jsonify(post.to_json())

class BlogExtension:
    def __init__(self, db):
        class Post(db.Model):
            id = db.Column(primary_key=True)
            title = db.Column(db.String, nullable=False)

        self.post_model = Post

    def init_app(self, app):
        api_view = PostAPI.as_view(model=self.post_model)

db = SQLAlchemy()
blog = BlogExtension(db)
db.init_app(app)
blog.init_app(app)
```

Another technique could be to use an attribute on the extension, such as `self.post_model` from above. Add the extension to `app.extensions` in `init_app`, then access `current_app.extensions["simple_blog"].post_model` from views.

You may also want to provide base classes so that users can provide their own Post model that conforms to the API your extension expects. So they could implement `class Post(blog.BasePost)`, then set it as `blog.post_model`.

As you can see, this can get a bit complex. Unfortunately, there's no perfect solution here, only different strategies and tradeoffs depending on your needs and how much customization you want to offer. Luckily, this sort of resource dependency is not a common need for most extensions. Remember, if you need help with design, ask on our [Discord Chat](#) or [GitHub Discussions](#).

3.2.7 Recommended Extension Guidelines

Flask previously had the concept of “approved extensions”, where the Flask maintainers evaluated the quality, support, and compatibility of the extensions before listing them. While the list became too difficult to maintain over time, the guidelines are still relevant to all extensions maintained and developed today, as they help the Flask ecosystem remain consistent and compatible.

1. An extension requires a maintainer. In the event an extension author would like to move beyond the project, the project should find a new maintainer and transfer access to the repository, documentation, PyPI, and any other services. The [Pallets-Eco](#) organization on GitHub allows for community maintenance with oversight from the Pallets maintainers.
2. The naming scheme is *Flask-ExtensionName* or *ExtensionName-Flask*. It must provide exactly one package or module named `flask_extension_name`.
3. The extension must use an open source license. The Python web ecosystem tends to prefer BSD or MIT. It must be open source and publicly available.
4. The extension’s API must have the following characteristics:
 - It must support multiple applications running in the same Python process. Use `current_app` instead of `self.app`, store configuration and state per application instance.
 - It must be possible to use the factory pattern for creating applications. Use the `ext.init_app()` pattern.
5. From a clone of the repository, an extension with its dependencies must be installable in editable mode with `pip install -e ..`.
6. It must ship tests that can be invoked with a common tool like `tox -e py`, `nox -s test` or `pytest`. If not using `tox`, the test dependencies should be specified in a requirements file. The tests must be part of the sdist distribution.
7. A link to the documentation or project website must be in the PyPI metadata or the readme. The documentation should use the Flask theme from the [Official Pallets Themes](#).
8. The extension’s dependencies should not use upper bounds or assume any particular version scheme, but should use lower bounds to indicate minimum compatibility support. For example, `sqlalchemy>=1.4`.
9. Indicate the versions of Python supported using `python_requires=">=version"`. Flask itself supports Python `>=3.8` as of April 2023, but this will update over time.

3.3 How to contribute to Flask

Thank you for considering contributing to Flask!

3.3.1 Support questions

Please don’t use the issue tracker for this. The issue tracker is a tool to address bugs and feature requests in Flask itself. Use one of the following resources for questions about using Flask or issues with your own code:

- The `#questions` channel on our Discord chat: <https://discord.gg/pallets>
- Ask on [Stack Overflow](#). Search with Google first using: `site:stackoverflow.com flask {search term, exception message, etc.}`
- Ask on our [GitHub Discussions](#) for long term discussion or larger questions.

3.3.2 Reporting issues

Include the following information in your post:

- Describe what you expected to happen.
- If possible, include a [minimal reproducible example](#) to help us identify the issue. This also helps check that the issue is not with your own code.
- Describe what actually happened. Include the full traceback if there was an exception.
- List your Python and Flask versions. If possible, check if this issue is already fixed in the latest releases or the latest code in the repository.

3.3.3 Submitting patches

If there is not an open issue for what you want to submit, prefer opening one for discussion before working on a PR. You can work on any issue that doesn't have an open PR linked to it or a maintainer assigned to it. These show up in the sidebar. No need to ask if you can work on an issue that interests you.

Include the following in your patch:

- Use [Black](#) to format your code. This and other tools will run automatically if you install [pre-commit](#) using the instructions below.
- Include tests if your patch adds or changes code. Make sure the test fails without your patch.
- Update any relevant docs pages and docstrings. Docs pages and docstrings should be wrapped at 72 characters.
- Add an entry in `CHANGES.rst`. Use the same style as other entries. Also include `.. versionchanged::` inline changelogs in relevant docstrings.

First time setup using GitHub Codespaces

[GitHub Codespaces](#) creates a development environment that is already set up for the project. By default it opens in Visual Studio Code for the Web, but this can be changed in your GitHub profile settings to use Visual Studio Code or JetBrains PyCharm on your local computer.

- Make sure you have a [GitHub account](#).
- From the project's repository page, click the green "Code" button and then "Create codespace on main".
- The codespace will be set up, then Visual Studio Code will open. However, you'll need to wait a bit longer for the Python extension to be installed. You'll know it's ready when the terminal at the bottom shows that the virtualenv was activated.
- Check out a branch and *start coding*.

First time setup in your local environment

- Make sure you have a [GitHub account](#).
- Download and install the [latest version of git](#).
- Configure git with your [username](#) and [email](#).

```
$ git config --global user.name 'your name'
$ git config --global user.email 'your email'
```

- Fork Flask to your GitHub account by clicking the [Fork](#) button.
- [Clone](#) your fork locally, replacing `your-username` in the command below with your actual username.

```
$ git clone https://github.com/your-username/flask
$ cd flask
```

- Create a virtualenv. Use the latest version of Python.
 - Linux/macOS

```
$ python3 -m venv .venv
$ . .venv/bin/activate
```

- Windows

```
> py -3 -m venv .venv
> .venv\Scripts\activate
```

- Install the development dependencies, then install Flask in editable mode.

```
$ python -m pip install -U pip setuptools wheel
$ pip install -r requirements/dev.txt && pip install -e .
```

- Install the pre-commit hooks.

```
$ pre-commit install --install-hooks
```

Start coding

- Create a branch to identify the issue you would like to work on. If you're submitting a bug or documentation fix, branch off of the latest `“x”` branch.

```
$ git fetch origin
$ git checkout -b your-branch-name origin/2.0.x
```

If you're submitting a feature addition or change, branch off of the `“main”` branch.

```
$ git fetch origin
$ git checkout -b your-branch-name origin/main
```

- Using your favorite editor, make your changes, [committing as you go](#).
 - If you are in a codespace, you will be prompted to [create a fork](#) the first time you make a commit. Enter `Y` to continue.
- Include tests that cover any code changes you make. Make sure the test fails without your patch. Run the tests as described below.
- Push your commits to your fork on GitHub and [create a pull request](#). Link to the issue being addressed with `fixes #123` in the pull request description.

```
$ git push --set-upstream origin your-branch-name
```

Running the tests

Run the basic test suite with `pytest`.

```
$ pytest
```

This runs the tests for the current environment, which is usually sufficient. CI will run the full suite when you submit your pull request. You can run the full test suite with `tox` if you don't want to wait.

```
$ tox
```

Running test coverage

Generating a report of lines that do not have test coverage can indicate where to start contributing. Run `pytest` using `coverage` and generate a report.

If you are using GitHub Codespaces, `coverage` is already installed so you can skip the installation command.

```
$ pip install coverage
$ coverage run -m pytest
$ coverage html
```

Open `htmlcov/index.html` in your browser to explore the report.

Read more about [coverage](#).

Building the docs

Build the docs in the `docs` directory using `Sphinx`.

```
$ cd docs
$ make html
```

Open `_build/html/index.html` in your browser to view the docs.

Read more about [Sphinx](#).

3.4 BSD-3-Clause License

Copyright 2010 Pallets

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

3.5 Changes

3.5.1 Version 2.3.2

Released 2023-05-01

- Set `Vary`: `Cookie` header when the session is accessed, modified, or refreshed.
- Update Werkzeug requirement to `>=2.3.3` to apply recent bug fixes.

3.5.2 Version 2.3.1

Released 2023-04-25

- Restore deprecated `flask import Markup`. [#5084](#)

3.5.3 Version 2.3.0

Released 2023-04-25

- Drop support for Python 3.7. [#5072](#)
- Update minimum requirements to the latest versions: Werkzeug`>=2.3.0`, Jinja2`>3.1.2`, itsdangerous`>=2.1.2`, click`>=8.1.3`.
- Remove previously deprecated code. [#4995](#)
 - The `push` and `pop` methods of the deprecated `_app_ctx_stack` and `_request_ctx_stack` objects are removed. `top` still exists to give extensions more time to update, but it will be removed.
 - The `FLASK_ENV` environment variable, `ENV` config key, and `app.env` property are removed.
 - The `session_cookie_name`, `send_file_max_age_default`, `use_x_sendfile`, `propagate_exceptions`, and `templates_auto_reload` properties on `app` are removed.
 - The `JSON_AS_ASCII`, `JSON_SORT_KEYS`, `JSONIFY_MIMETYPE`, and `JSONIFY_PRETTYPRINT_REGULAR` config keys are removed.
 - The `app.before_first_request` and `bp.before_app_first_request` decorators are removed.
 - `json_encoder` and `json_decoder` attributes on `app` and `blueprint`, and the corresponding `json.JSONEncoder` and `JSONDecoder` classes, are removed.
 - The `json.htmlsafe_dumps` and `htmlsafe_dump` functions are removed.
 - Calling setup methods on blueprints after registration is an error instead of a warning. [#4997](#)

- Importing `escape` and `Markup` from `flask` is deprecated. Import them directly from `markupsafe` instead. [#4996](#)
- The `app.get_first_request` property is deprecated. [#4997](#)
- The `locked_cached_property` decorator is deprecated. Use a lock inside the decorated function if locking is needed. [#4993](#)
- Signals are always available. `blinker>=1.6.2` is a required dependency. The `signals_available` attribute is deprecated. [#5056](#)
- Signals support `async` subscriber functions. [#5049](#)
- Remove uses of locks that could cause requests to block each other very briefly. [#4993](#)
- Use modern packaging metadata with `pyproject.toml` instead of `setup.cfg`. [#4947](#)
- Ensure subdomains are applied with nested blueprints. [#4834](#)
- `config.from_file` can use `text=False` to indicate that the parser wants a binary file instead. [#4989](#)
- If a blueprint is created with an empty name it raises a `ValueError`. [#5010](#)
- `SESSION_COOKIE_DOMAIN` does not fall back to `SERVER_NAME`. The default is not to set the domain, which modern browsers interpret as an exact match rather than a subdomain match. Warnings about `localhost` and IP addresses are also removed. [#5051](#)
- The `routes` command shows each rule's `subdomain` or `host` when domain matching is in use. [#5004](#)
- Use postponed evaluation of annotations. [#5071](#)

3.5.4 Version 2.2.5

Released 2023-05-02

- Update for compatibility with Werkzeug 2.3.3.
- Set `Vary: Cookie` header when the session is accessed, modified, or refreshed.

3.5.5 Version 2.2.4

Released 2023-04-25

- Update for compatibility with Werkzeug 2.3.

3.5.6 Version 2.2.3

Released 2023-02-15

- Autoescape is enabled by default for `.svg` template files. [#4831](#)
- Fix the type of `template_folder` to accept `pathlib.Path`. [#4892](#)
- Add `--debug` option to the `flask run` command. [#4777](#)

3.5.7 Version 2.2.2

Released 2022-08-08

- Update Werkzeug dependency to `>= 2.2.2`. This includes fixes related to the new faster router, header parsing, and the development server. [#4754](#)
- Fix the default value for `app.env` to be `"production"`. This attribute remains deprecated. [#4740](#)

3.5.8 Version 2.2.1

Released 2022-08-03

- Setting or accessing `json_encoder` or `json_decoder` raises a deprecation warning. [#4732](#)

3.5.9 Version 2.2.0

Released 2022-08-01

- Remove previously deprecated code. [#4667](#)
 - Old names for some `send_file` parameters have been removed. `download_name` replaces `attachment_filename`, `max_age` replaces `cache_timeout`, and `etag` replaces `add_etags`. Additionally, `path` replaces `filename` in `send_from_directory`.
 - The `RequestContext.g` property returning `AppContext.g` is removed.
- Update Werkzeug dependency to `>= 2.2`.
- The app and request contexts are managed using Python context vars directly rather than Werkzeug's `LocalStack`. This should result in better performance and memory use. [#4682](#)
 - Extension maintainers, be aware that `_app_ctx_stack.top` and `_request_ctx_stack.top` are deprecated. Store data on `g` instead using a unique prefix, like `g._extension_name_attr`.
- The `FLASK_ENV` environment variable and `app.env` attribute are deprecated, removing the distinction between development and debug mode. Debug mode should be controlled directly using the `--debug` option or `app.run(debug=True)`. [#4714](#)
- Some attributes that proxied config keys on `app` are deprecated: `session_cookie_name`, `send_file_max_age_default`, `use_x_sendfile`, `propagate_exceptions`, and `templates_auto_reload`. Use the relevant config keys instead. [#4716](#)
- Add new customization points to the Flask app object for many previously global behaviors.
 - `flask.url_for` will call `app.url_for`. [#4568](#)
 - `flask.abort` will call `app.aborter`. `Flask.aborter_class` and `Flask.make_aborter` can be used to customize this aborter. [#4567](#)
 - `flask.redirect` will call `app.redirect`. [#4569](#)
 - `flask.json` is an instance of `JSONProvider`. A different provider can be set to use a different JSON library. `flask.jsonify` will call `app.json.response`, other functions in `flask.json` will call corresponding functions in `app.json`. [#4692](#)
- JSON configuration is moved to attributes on the default `app.json` provider. `JSON_AS_ASCII`, `JSON_SORT_KEYS`, `JSONIFY_MIMETYPE`, and `JSONIFY_PRETTYPRINT_REGULAR` are deprecated. [#4692](#)

- Setting custom `json_encoder` and `json_decoder` classes on the app or a blueprint, and the corresponding `json.JSONEncoder` and `JSONDecoder` classes, are deprecated. JSON behavior can now be overridden using the `app.json` provider interface. #4692
- `json.htmlsafe_dumps` and `json.htmlsafe_dump` are deprecated, the function is built-in to Jinja now. #4692
- Refactor `register_error_handler` to consolidate error checking. Rewrite some error messages to be more consistent. #4559
- Use Blueprint decorators and functions intended for setup after registering the blueprint will show a warning. In the next version, this will become an error just like the application setup methods. #4571
- `before_first_request` is deprecated. Run setup code when creating the application instead. #4605
- Added the `View.init_every_request` class attribute. If a view subclass sets this to `False`, the view will not create a new instance on every request. #2520.
- A `flask.cli.FlaskGroup` Click group can be nested as a sub-command in a custom CLI. #3263
- Add `--app` and `--debug` options to the flask CLI, instead of requiring that they are set through environment variables. #2836
- Add `--env-file` option to the flask CLI. This allows specifying a dotenv file to load in addition to `.env` and `.flaskenv`. #3108
- It is no longer required to decorate custom CLI commands on `app.cli` or `blueprint.cli` with `@with_appcontext`, an app context will already be active at that point. #2410
- `SessionInterface.get_expiration_time` uses a timezone-aware value. #4645
- View functions can return generators directly instead of wrapping them in a `Response`. #4629
- Add `stream_template` and `stream_template_string` functions to render a template as a stream of pieces. #4629
- A new implementation of context preservation during debugging and testing. #4666
 - `request`, `g`, and other context-locals point to the correct data when running code in the interactive debugger console. #2836
 - Teardown functions are always run at the end of the request, even if the context is preserved. They are also run after the preserved context is popped.
 - `stream_with_context` preserves context separately from a `with client` block. It will be cleaned up when `response.get_data()` or `response.close()` is called.
- Allow returning a list from a view function, to convert it to a JSON response like a dict is. #4672
- When type checking, allow `TypedDict` to be returned from view functions. #4695
- Remove the `--eager-loading/--lazy-loading` options from the `flask run` command. The app is always eager loaded the first time, then lazily loaded in the reloader. The reloader always prints errors immediately but continues serving. Remove the internal `DispatchingApp` middleware used by the previous implementation. #4715

3.5.10 Version 2.1.3

Released 2022-07-13

- Inline some optional imports that are only used for certain CLI commands. [#4606](#)
- Relax type annotation for `after_request` functions. [#4600](#)
- `instance_path` for namespace packages uses the path closest to the imported submodule. [#4610](#)
- Clearer error message when `render_template` and `render_template_string` are used outside an application context. [#4693](#)

3.5.11 Version 2.1.2

Released 2022-04-28

- Fix type annotation for `json.loads`, it accepts str or bytes. [#4519](#)
- The `--cert` and `--key` options on `flask run` can be given in either order. [#4459](#)

3.5.12 Version 2.1.1

Released on 2022-03-30

- Set the minimum required version of `importlib_metadata` to 3.6.0, which is required on Python < 3.10. [#4502](#)

3.5.13 Version 2.1.0

Released 2022-03-28

- Drop support for Python 3.6. [#4335](#)
- Update Click dependency to `>= 8.0`. [#4008](#)
- Remove previously deprecated code. [#4337](#)
 - The CLI does not pass `script_info` to app factory functions.
 - `config.from_json` is replaced by `config.from_file(name, load=json.load)`.
 - `json` functions no longer take an `encoding` parameter.
 - `safe_join` is removed, use `werkzeug.utils.safe_join` instead.
 - `total_seconds` is removed, use `timedelta.total_seconds` instead.
 - The same blueprint cannot be registered with the same name. Use `name=` when registering to specify a unique name.
 - The test client's `as_tuple` parameter is removed. Use `response.request.environ` instead. [#4417](#)
- Some parameters in `send_file` and `send_from_directory` were renamed in 2.0. The deprecation period for the old names is extended to 2.2. Be sure to test with deprecation warnings visible.
 - `attachment_filename` is renamed to `download_name`.
 - `cache_timeout` is renamed to `max_age`.
 - `add_etags` is renamed to `etag`.
 - `filename` is renamed to `path`.

- The `RequestContext.g` property is deprecated. Use `g` directly or `AppContext.g` instead. #3898
- `copy_current_request_context` can decorate async functions. #4303
- The CLI uses `importlib.metadata` instead of `setuptools` to load command entry points. #4419
- Overriding `FlaskClient.open` will not cause an error on redirect. #3396
- Add an `--exclude-patterns` option to the `flask run` CLI command to specify patterns that will be ignored by the reloader. #4188
- When using lazy loading (the default with the debugger), the Click context from the `flask run` command remains available in the loader thread. #4460
- Deleting the session cookie uses the `httponly` flag. #4485
- Relax typing for `errorhandler` to allow the user to use more precise types and decorate the same function multiple times. #4095#4295#4297
- Fix typing for `__exit__` methods for better compatibility with `ExitStack`. #4474
- From Werkzeug, for redirect responses the `Location` header URL will remain relative, and exclude the scheme and domain, by default. #4496
- Add `Config.from_prefixed_env()` to load config values from environment variables that start with `FLASK_` or another prefix. This parses values as JSON by default, and allows setting keys in nested dicts. #4479

3.5.14 Version 2.0.3

Released 2022-02-14

- The test client's `as_tuple` parameter is deprecated and will be removed in Werkzeug 2.1. It is now also deprecated in Flask, to be removed in Flask 2.1, while remaining compatible with both in 2.0.x. Use `response.request.environ` instead. #4341
- Fix type annotation for `errorhandler` decorator. #4295
- Revert a change to the CLI that caused it to hide `ImportError` tracebacks when importing the application. #4307
- `app.json_encoder` and `json_decoder` are only passed to `dumps` and `loads` if they have custom behavior. This improves performance, mainly on PyPy. #4349
- Clearer error message when `after_this_request` is used outside a request context. #4333

3.5.15 Version 2.0.2

Released 2021-10-04

- Fix type annotation for `teardown_*` methods. #4093
- Fix type annotation for `before_request` and `before_app_request` decorators. #4104
- Fixed the issue where typing requires template global decorators to accept functions with no arguments. #4098
- Support `View` and `MethodView` instances with async handlers. #4112
- Enhance typing of `app.errorhandler` decorator. #4095
- Fix registering a blueprint twice with differing names. #4124
- Fix the type of `static_folder` to accept `pathlib.Path`. #4150
- `jsonify` handles `decimal.Decimal` by encoding to `str`. #4157

- Correctly handle raising deferred errors in CLI lazy loading. #4096
- The CLI loader handles `**kwargs` in a `create_app` function. #4170
- Fix the order of `before_request` and other callbacks that trigger before the view returns. They are called from the app down to the closest nested blueprint. #4229

3.5.16 Version 2.0.1

Released 2021-05-21

- Re-add the `filename` parameter in `send_from_directory`. The `filename` parameter has been renamed to `path`, the old name is deprecated. #4019
- Mark top-level names as exported so type checking understands imports in user projects. #4024
- Fix type annotation for `g` and inform mypy that it is a namespace object that has arbitrary attributes. #4020
- Fix some types that weren't available in Python 3.6.0. #4040
- Improve typing for `send_file`, `send_from_directory`, and `get_send_file_max_age`. #4044, #4026
- Show an error when a blueprint name contains a dot. The `.` has special meaning, it is used to separate (nested) blueprint names and the endpoint name. #4041
- Combine URL prefixes when nesting blueprints that were created with a `url_prefix` value. #4037
- Revert a change to the order that URL matching was done. The URL is again matched after the session is loaded, so the session is available in custom URL converters. #4053
- Re-add deprecated `Config.from_json`, which was accidentally removed early. #4078
- Improve typing for some functions using `Callable` in their type signatures, focusing on decorator factories. #4060
- Nested blueprints are registered with their dotted name. This allows different blueprints with the same name to be nested at different locations. #4069
- `register_blueprint` takes a `name` option to change the (pre-dotted) name the blueprint is registered with. This allows the same blueprint to be registered multiple times with unique names for `url_for`. Registering the same blueprint with the same name multiple times is deprecated. #1091
- Improve typing for `stream_with_context`. #4052

3.5.17 Version 2.0.0

Released 2021-05-11

- Drop support for Python 2 and 3.5.
- Bump minimum versions of other Pallets projects: Werkzeug ≥ 2 , Jinja2 ≥ 3 , MarkupSafe ≥ 2 , ItsDangerous ≥ 2 , Click ≥ 8 . Be sure to check the change logs for each project. For better compatibility with other applications (e.g. Celery) that still require Click 7, there is no hard dependency on Click 8 yet, but using Click 7 will trigger a `DeprecationWarning` and Flask 2.1 will depend on Click 8.
- JSON support no longer uses `simplejson`. To use another JSON module, override `app.json_encoder` and `json_decoder`. #3555
- The `encoding` option to JSON functions is deprecated. #3562
- Passing `script_info` to app factory functions is deprecated. This was not portable outside the `flask` command. Use `click.get_current_context().obj` if it's needed. #3552

- The CLI shows better error messages when the app failed to load when looking up commands. #2741
- Add `SessionInterface.get_cookie_name` to allow setting the session cookie name dynamically. #3369
- Add `Config.from_file` to load config using arbitrary file loaders, such as `toml.load` or `json.load`. `Config.from_json` is deprecated in favor of this. #3398
- The `flask run` command will only defer errors on reload. Errors present during the initial call will cause the server to exit with the traceback immediately. #3431
- `send_file` raises a `ValueError` when passed an `io` object in text mode. Previously, it would respond with 200 OK and an empty file. #3358
- When using ad-hoc certificates, check for the cryptography library instead of PyOpenSSL. #3492
- When specifying a factory function with `FLASK_APP`, keyword argument can be passed. #3553
- When loading a `.env` or `.flaskenv` file, the current working directory is no longer changed to the location of the file. #3560
- When returning a `(response, headers)` tuple from a view, the headers replace rather than extend existing headers on the response. For example, this allows setting the `Content-Type` for `jsonify()`. Use `response.headers.extend()` if extending is desired. #3628
- The `Scaffold` class provides a common API for the `Flask` and `Blueprint` classes. `Blueprint` information is stored in attributes just like `Flask`, rather than opaque lambda functions. This is intended to improve consistency and maintainability. #3215
- Include `samesite` and `secure` options when removing the session cookie. #3726
- Support passing a `pathlib.Path` to `static_folder`. #3579
- `send_file` and `send_from_directory` are wrappers around the implementations in `werkzeug.utils`. #3828
- Some `send_file` parameters have been renamed, the old names are deprecated. `attachment_filename` is renamed to `download_name`. `cache_timeout` is renamed to `max_age`. `add_etags` is renamed to `etag`. #3828#3883
- `send_file` passes `download_name` even if `as_attachment=False` by using `Content-Disposition: inline`. #3828
- `send_file` sets `conditional=True` and `max_age=None` by default. `Cache-Control` is set to `no-cache` if `max_age` is not set, otherwise `public`. This tells browsers to validate conditional requests instead of using a timed cache. #3828
- `helpers.safe_join` is deprecated. Use `werkzeug.utils.safe_join` instead. #3828
- The request context does route matching before opening the session. This could allow a session interface to change behavior based on `request.endpoint`. #3776
- Use Jinja's implementation of the `|tojson` filter. #3881
- Add route decorators for common HTTP methods. For example, `@app.post("/login")` is a shortcut for `@app.route("/login", methods=["POST"])`. #3907
- Support async views, error handlers, before and after request, and teardown functions. #3412
- Support nesting blueprints. #593#1548, #3923
- Set the default encoding to "UTF-8" when loading `.env` and `.flaskenv` files to allow to use non-ASCII characters. #3931
- `flask shell` sets up tab and history completion like the default python shell if `readline` is installed. #3941
- `helpers.total_seconds()` is deprecated. Use `timedelta.total_seconds()` instead. #3962

- Add type hinting. [#3973](#).

3.5.18 Version 1.1.4

Released 2021-05-13

- Update `static_folder` to use `_compat.fspath` instead of `os.fspath` to continue supporting Python < 3.6 [#4050](#)

3.5.19 Version 1.1.3

Released 2021-05-13

- Set maximum versions of Werkzeug, Jinja, Click, and ItsDangerous. [#4043](#)
- Re-add support for passing a `pathlib.Path` for `static_folder`. [#3579](#)

3.5.20 Version 1.1.2

Released 2020-04-03

- Work around an issue when running the `flask` command with an external debugger on Windows. [#3297](#)
- The static route will not catch all URLs if the Flask `static_folder` argument ends with a slash. [#3452](#)

3.5.21 Version 1.1.1

Released 2019-07-08

- The `flask.json_available` flag was added back for compatibility with some extensions. It will raise a deprecation warning when used, and will be removed in version 2.0.0. [#3288](#)

3.5.22 Version 1.1.0

Released 2019-07-04

- Bump minimum Werkzeug version to `>= 0.15`.
- Drop support for Python 3.4.
- Error handlers for `InternalServerError` or `500` will always be passed an instance of `InternalServerError`. If they are invoked due to an unhandled exception, that original exception is now available as `e.original_exception` rather than being passed directly to the handler. The same is true if the handler is for the base `HTTPException`. This makes error handler behavior more consistent. [#3266](#)
 - `Flask.finalize_request` is called for all unhandled exceptions even if there is no `500` error handler.
- `Flask.logger` takes the same name as `Flask.name` (the value passed as `Flask(import_name)`). This reverts 1.0's behavior of always logging to `"flask.app"`, in order to support multiple apps in the same process. A warning will be shown if old configuration is detected that needs to be moved. [#2866](#)
- `RequestContext.copy` includes the current session object in the request context copy. This prevents `session` pointing to an out-of-date object. [#2935](#)
- Using built-in `RequestContext`, unprintable Unicode characters in `Host` header will result in a HTTP 400 response and not HTTP 500 as previously. [#2994](#)

- `send_file` supports PathLike objects as described in [PEP 519](#), to support `pathlib` in Python 3. [#3059](#)
- `send_file` supports BytesIO partial content. [#2957](#)
- `open_resource` accepts the “rt” file mode. This still does the same thing as “r”. [#3163](#)
- The `MethodView.methods` attribute set in a base class is used by subclasses. [#3138](#)
- `Flask.jinja_options` is a dict instead of an `ImmutableDict` to allow easier configuration. Changes must still be made before creating the environment. [#3190](#)
- Flask’s `JSONMixin` for the request and response wrappers was moved into Werkzeug. Use Werkzeug’s version with Flask-specific support. This bumps the Werkzeug dependency to `>= 0.15`. [#3125](#)
- The `flask` command entry point is simplified to take advantage of Werkzeug 0.15’s better reloader support. This bumps the Werkzeug dependency to `>= 0.15`. [#3022](#)
- Support `static_url_path` that ends with a forward slash. [#3134](#)
- Support empty `static_folder` without requiring setting an empty `static_url_path` as well. [#3124](#)
- `jsonify` supports `dataclass` objects. [#3195](#)
- Allow customizing the `Flask.url_map_class` used for routing. [#3069](#)
- The development server port can be set to 0, which tells the OS to pick an available port. [#2926](#)
- The return value from `cli.load_dotenv` is more consistent with the documentation. It will return `False` if `python-dotenv` is not installed, or if the given path isn’t a file. [#2937](#)
- Signaling support has a stub for the `connect_via` method when the `Blinker` library is not installed. [#3208](#)
- Add an `--extra-files` option to the `flask run` CLI command to specify extra files that will trigger the reloader on change. [#2897](#)
- Allow returning a dictionary from a view function. Similar to how returning a string will produce a `text/html` response, returning a dict will call `jsonify` to produce a `application/json` response. [#3111](#)
- Blueprints have a `cli` Click group like `app.cli`. CLI commands registered with a blueprint will be available as a group under the `flask` command. [#1357](#).
- When using the test client as a context manager (`with client:`), all preserved request contexts are popped when the block exits, ensuring nested contexts are cleaned up correctly. [#3157](#)
- Show a better error message when the view return type is not supported. [#3214](#)
- `flask.testing.make_test_environ_builder()` has been deprecated in favour of a new class `flask.testing.EnvironBuilder`. [#3232](#)
- The `flask run` command no longer fails if Python is not built with SSL support. Using the `--cert` option will show an appropriate error message. [#3211](#)
- URL matching now occurs after the request context is pushed, rather than when it’s created. This allows custom URL converters to access the app and request contexts, such as to query a database for an id. [#3088](#)

3.5.23 Version 1.0.4

Released 2019-07-04

- The key information for `BadRequestKeyError` is no longer cleared outside debug mode, so error handlers can still access it. This requires upgrading to Werkzeug 0.15.5. [#3249](#)
- `send_file` url quotes the “:” and “/” characters for more compatible UTF-8 filename support in some browsers. [#3074](#)
- Fixes for [PEP 451](#) import loaders and pytest 5.x. [#3275](#)
- Show message about dotenv on stderr instead of stdout. [#3285](#)

3.5.24 Version 1.0.3

Released 2019-05-17

- `send_file` encodes filenames as ASCII instead of Latin-1 (ISO-8859-1). This fixes compatibility with Gunicorn, which is stricter about header encodings than [PEP 3333](#). [#2766](#)
- Allow custom CLIs using `FlaskGroup` to set the debug flag without it always being overwritten based on environment variables. [#2765](#)
- `flask --version` outputs Werkzeug’s version and simplifies the Python version. [#2825](#)
- `send_file` handles an `attachment_filename` that is a native Python 2 string (bytes) with UTF-8 coded bytes. [#2933](#)
- A catch-all error handler registered for `HTTPException` will not handle `RoutingException`, which is used internally during routing. This fixes the unexpected behavior that had been introduced in 1.0. [#2986](#)
- Passing the `json` argument to `app.test_client` does not push/pop an extra app context. [#2900](#)

3.5.25 Version 1.0.2

Released 2018-05-02

- Fix more backwards compatibility issues with merging slashes between a blueprint prefix and route. [#2748](#)
- Fix error with `flask routes` command when there are no routes. [#2751](#)

3.5.26 Version 1.0.1

Released 2018-04-29

- Fix registering partials (with no `__name__`) as view functions. [#2730](#)
- Don’t treat lists returned from view functions the same as tuples. Only tuples are interpreted as response data. [#2736](#)
- Extra slashes between a blueprint’s `url_prefix` and a route URL are merged. This fixes some backwards compatibility issues with the change in 1.0. [#2731](#), [#2742](#)
- Only trap `BadRequestKeyError` errors in debug mode, not all `BadRequest` errors. This allows `abort(400)` to continue working as expected. [#2735](#)
- The `FLASK_SKIP_DOTENV` environment variable can be set to 1 to skip automatically loading dotenv files. [#2722](#)

3.5.27 Version 1.0

Released 2018-04-26

- Python 2.6 and 3.3 are no longer supported.
- Bump minimum dependency versions to the latest stable versions: Werkzeug \geq 0.14, Jinja \geq 2.10, itsdangerous \geq 0.24, Click \geq 5.1. [#2586](#)
- Skip `app.run` when a Flask application is run from the command line. This avoids some behavior that was confusing to debug.
- Change the default for `JSONIFY_PRETTYPRINT_REGULAR` to `False`. `~json jsonify` returns a compact format by default, and an indented format in debug mode. [#2193](#)
- `Flask.__init__` accepts the `host_matching` argument and sets it on `Flask.url_map`. [#1559](#)
- `Flask.__init__` accepts the `static_host` argument and passes it as the `host` argument when defining the static route. [#1559](#)
- `send_file` supports Unicode in `attachment_filename`. [#2223](#)
- Pass `_scheme` argument from `url_for` to `Flask.handle_url_build_error`. [#2017](#)
- `Flask.add_url_rule` accepts the `provide_automatic_options` argument to disable adding the `OPTIONS` method. [#1489](#)
- `MethodView` subclasses inherit method handlers from base classes. [#1936](#)
- Errors caused while opening the session at the beginning of the request are handled by the app's error handlers. [#2254](#)
- Blueprints gained `Blueprint.json_encoder` and `Blueprint.json_decoder` attributes to override the app's encoder and decoder. [#1898](#)
- `Flask.make_response` raises `TypeError` instead of `ValueError` for bad response types. The error messages have been improved to describe why the type is invalid. [#2256](#)
- Add `routes CLI` command to output routes registered on the application. [#2259](#)
- Show warning when session cookie domain is a bare hostname or an IP address, as these may not behave properly in some browsers, such as Chrome. [#2282](#)
- Allow IP address as exact session cookie domain. [#2282](#)
- `SESSION_COOKIE_DOMAIN` is set if it is detected through `SERVER_NAME`. [#2282](#)
- Auto-detect zero-argument app factory called `create_app` or `make_app` from `FLASK_APP`. [#2297](#)
- Factory functions are not required to take a `script_info` parameter to work with the `flask` command. If they take a single parameter or a parameter named `script_info`, the `ScriptInfo` object will be passed. [#2319](#)
- `FLASK_APP` can be set to an app factory, with arguments if needed, for example `FLASK_APP=myproject.app:create_app('dev')`. [#2326](#)
- `FLASK_APP` can point to local packages that are not installed in editable mode, although `pip install -e` is still preferred. [#2414](#)
- The `View` class attribute `View.provide_automatic_options` is set in `View.as_view`, to be detected by `Flask.add_url_rule`. [#2316](#)
- Error handling will try handlers registered for `blueprint`, `code`, `app`, `code, blueprint`, `exception`, `app`, `exception`. [#2314](#)
- `Cookie` is added to the response's `Vary` header if the session is accessed at all during the request (and not deleted). [#2288](#)

- `Flask.test_request_context` accepts `subdomain` and `url_scheme` arguments for use when building the base URL. #1621
- Set `APPLICATION_ROOT` to `'/'` by default. This was already the implicit default when it was set to `None`.
- `TRAP_BAD_REQUEST_ERRORS` is enabled by default in debug mode. `BadRequestKeyError` has a message with the bad key in debug mode instead of the generic bad request message. #2348
- Allow registering new tags with `TaggedJSONSerializer` to support storing other types in the session cookie. #2352
- Only open the session if the request has not been pushed onto the context stack yet. This allows `stream_with_context` generators to access the same session that the containing view uses. #2354
- Add `json` keyword argument for the test client request methods. This will dump the given object as JSON and set the appropriate content type. #2358
- Extract JSON handling to a mixin applied to both the `Request` and `Response` classes. This adds the `Response.is_json` and `Response.get_json` methods to the response to make testing JSON response much easier. #2358
- Removed error handler caching because it caused unexpected results for some exception inheritance hierarchies. Register handlers explicitly for each exception if you want to avoid traversing the MRO. #2362
- Fix incorrect JSON encoding of aware, non-UTC datetimes. #2374
- Template auto reloading will honor debug mode even even if `Flask.jinja_env` was already accessed. #2373
- The following old deprecated code was removed. #2385
 - `flask.ext` - import extensions directly by their name instead of through the `flask.ext` namespace. For example, `import flask.ext.sqlalchemy` becomes `import flask_sqlalchemy`.
 - `Flask.init_jinja_globals` - extend `Flask.create_jinja_environment` instead.
 - `Flask.error_handlers` - tracked by `Flask.error_handler_spec`, use `Flask.errorhandler` to register handlers.
 - `Flask.request_globals_class` - use `Flask.app_ctx_globals_class` instead.
 - `Flask.static_path` - use `Flask.static_url_path` instead.
 - `Request.module` - use `Request.blueprint` instead.
- The `Request.json` property is no longer deprecated. #1421
- Support passing a `EnvironBuilder` or `dict` to `test_client.open`. #2412
- The `flask` command and `Flask.run` will load environment variables from `.env` and `.flaskenv` files if `python-dotenv` is installed. #2416
- When passing a full URL to the test client, the scheme in the URL is used instead of `PREFERRED_URL_SCHEME`. #2430
- `Flask.logger` has been simplified. `LOGGER_NAME` and `LOGGER_HANDLER_POLICY` config was removed. The logger is always named `flask.app`. The level is only set on first access, it doesn't check `Flask.debug` each time. Only one format is used, not different ones depending on `Flask.debug`. No handlers are removed, and a handler is only added if no handlers are already configured. #2436
- Blueprint view function names may not contain dots. #2450
- Fix a `ValueError` caused by invalid Range requests in some cases. #2526
- The development server uses threads by default. #2529
- Loading config files with `silent=True` will ignore `ENOTDIR` errors. #2581
- Pass `--cert` and `--key` options to `flask run` to run the development server over HTTPS. #2606

- Added `SESSION_COOKIE_SAMESITE` to control the SameSite attribute on the session cookie. [#2607](#)
- Added `Flask.test_cli_runner` to create a Click runner that can invoke Flask CLI commands for testing. [#2636](#)
- Subdomain matching is disabled by default and setting `SERVER_NAME` does not implicitly enable it. It can be enabled by passing `subdomain_matching=True` to the Flask constructor. [#2635](#)
- A single trailing slash is stripped from the blueprint `url_prefix` when it is registered with the app. [#2629](#)
- `Request.get_json` doesn't cache the result if parsing fails when `silent` is true. [#2651](#)
- `Request.get_json` no longer accepts arbitrary encodings. Incoming JSON should be encoded using UTF-8 per [RFC 8259](#), but Flask will autodetect UTF-8, -16, or -32. [#2691](#)
- Added `MAX_COOKIE_SIZE` and `Response.max_cookie_size` to control when Werkzeug warns about large cookies that browsers may ignore. [#2693](#)
- Updated documentation theme to make docs look better in small windows. [#2709](#)
- Rewrote the tutorial docs and example project to take a more structured approach to help new users avoid common pitfalls. [#2676](#)

3.5.28 Version 0.12.5

Released 2020-02-10

- Pin Werkzeug to < 1.0.0. [#3497](#)

3.5.29 Version 0.12.4

Released 2018-04-29

- Repackage 0.12.3 to fix package layout issue. [#2728](#)

3.5.30 Version 0.12.3

Released 2018-04-26

- `Request.get_json` no longer accepts arbitrary encodings. Incoming JSON should be encoded using UTF-8 per [RFC 8259](#), but Flask will autodetect UTF-8, -16, or -32. [#2692](#)
- Fix a Python warning about imports when using `python -m flask`. [#2666](#)
- Fix a `ValueError` caused by invalid Range requests in some cases.

3.5.31 Version 0.12.2

Released 2017-05-16

- Fix a bug in `safe_join` on Windows.

3.5.32 Version 0.12.1

Released 2017-03-31

- Prevent `flask run` from showing a `NoAppException` when an `ImportError` occurs within the imported application module.
- Fix encoding behavior of `app.config.from_pyfile` for Python 3. #2118
- Use the `SERVER_NAME` config if it is present as default values for `app.run`. #2109, #2152
- Call `ctx.auto_pop` with the exception object instead of `None`, in the event that a `BaseException` such as `KeyboardInterrupt` is raised in a request handler.

3.5.33 Version 0.12

Released 2016-12-21, codename Punsch

- The cli command now responds to `--version`.
- Mimetype guessing and ETag generation for file-like objects in `send_file` has been removed. #104, #1849
- Mimetype guessing in `send_file` now fails loudly and doesn't fall back to `application/octet-stream`. #1988
- Make `flask.safe_join` able to join multiple paths like `os.path.join` #1730
- Revert a behavior change that made the dev server crash instead of returning an Internal Server Error. #2006
- Correctly invoke response handlers for both regular request dispatching as well as error handlers.
- Disable logger propagation by default for the app logger.
- Add support for range requests in `send_file`.
- `app.test_client` includes preset default environment, which can now be directly set, instead of per `client.get`.
- Fix crash when running under PyPy3. #1814

3.5.34 Version 0.11.1

Released 2016-06-07

- Fixed a bug that prevented `FLASK_APP=foobar/__init__.py` from working. #1872

3.5.35 Version 0.11

Released 2016-05-29, codename Absinthe

- Added support to serializing top-level arrays to `jsonify`. This introduces a security risk in ancient browsers.
- Added `before_render_template` signal.
- Added `**kwargs` to `Flask.test_client` to support passing additional keyword arguments to the constructor of `Flask.test_client_class`.
- Added `SESSION_REFRESH_EACH_REQUEST` config key that controls the set-cookie behavior. If set to `True` a permanent session will be refreshed each request and get their lifetime extended, if set to `False` it will only be modified if the session actually modifies. Non permanent sessions are not affected by this and will always expire if the browser window closes.

- Made Flask support custom JSON mimetypes for incoming data.
- Added support for returning tuples in the form (`response`, `headers`) from a view function.
- Added `Config.from_json`.
- Added `Flask.config_class`.
- Added `Config.get_namespace`.
- Templates are no longer automatically reloaded outside of debug mode. This can be configured with the new `TEMPLATES_AUTO_RELOAD` config key.
- Added a workaround for a limitation in Python 3.3's namespace loader.
- Added support for explicit root paths when using Python 3.3's namespace packages.
- Added `flask` and the `flask.cli` module to start the local debug server through the click CLI system. This is recommended over the old `flask.run()` method as it works faster and more reliable due to a different design and also replaces `Flask-Script`.
- Error handlers that match specific classes are now checked first, thereby allowing catching exceptions that are subclasses of HTTP exceptions (in `werkzeug.exceptions`). This makes it possible for an extension author to create exceptions that will by default result in the HTTP error of their choosing, but may be caught with a custom error handler if desired.
- Added `Config.from_mapping`.
- Flask will now log by default even if debug is disabled. The log format is now hardcoded but the default log handling can be disabled through the `LOGGER_HANDLER_POLICY` configuration key.
- Removed deprecated module functionality.
- Added the `EXPLAIN_TEMPLATE_LOADING` config flag which when enabled will instruct Flask to explain how it locates templates. This should help users debug when the wrong templates are loaded.
- Enforce blueprint handling in the order they were registered for template loading.
- Ported test suite to `pytest`.
- Deprecated `request.json` in favour of `request.get_json()`.
- Add “pretty” and “compressed” separators definitions in `jsonify()` method. Reduces JSON response size when `JSONIFY_PRETTYPRINT_REGULAR=False` by removing unnecessary white space included by default after separators.
- JSON responses are now terminated with a newline character, because it is a convention that UNIX text files end with a newline and some clients don't deal well when this newline is missing. [#1262](#)
- The automatically provided `OPTIONS` method is now correctly disabled if the user registered an overriding rule with the lowercase-version `options`. [#1288](#)
- `flask.json.jsonify` now supports the `datetime.date` type. [#1326](#)
- Don't leak exception info of already caught exceptions to context teardown handlers. [#1393](#)
- Allow custom Jinja environment subclasses. [#1422](#)
- Updated extension dev guidelines.
- `flask.g` now has `pop()` and `setdefault` methods.
- Turn on autoescape for `flask.templating.render_template_string` by default. [#1515](#)
- `flask.ext` is now deprecated. [#1484](#)
- `send_from_directory` now raises `BadRequest` if the filename is invalid on the server OS. [#1763](#)

- Added the `JSONIFY_MIMETYPE` configuration variable. [#1728](#)
- Exceptions during teardown handling will no longer leave bad application contexts lingering around.
- Fixed broken `test_appcontext_signals()` test case.
- Raise an `AttributeError` in `helpers.find_package` with a useful message explaining why it is raised when a [PEP 302](#) import hook is used without an `is_package()` method.
- Fixed an issue causing exceptions raised before entering a request or app context to be passed to teardown handlers.
- Fixed an issue with query parameters getting removed from requests in the test client when absolute URLs were requested.
- Made `@before_first_request` into a decorator as intended.
- Fixed an etags bug when sending a file streams with a name.
- Fixed `send_from_directory` not expanding to the application root path correctly.
- Changed logic of before first request handlers to flip the flag after invoking. This will allow some uses that are potentially dangerous but should probably be permitted.
- Fixed Python 3 bug when a handler from `app.url_build_error_handlers` reraises the `BuildError`.

3.5.36 Version 0.10.1

Released 2013-06-14

- Fixed an issue where `|tojson` was not quoting single quotes which made the filter not work properly in HTML attributes. Now it's possible to use that filter in single quoted attributes. This should make using that filter with `angular.js` easier.
- Added support for byte strings back to the session system. This broke compatibility with the common case of people putting binary data for token verification into the session.
- Fixed an issue where registering the same method twice for the same endpoint would trigger an exception incorrectly.

3.5.37 Version 0.10

Released 2013-06-13, codename Limoncello

- Changed default cookie serialization format from pickle to JSON to limit the impact an attacker can do if the secret key leaks.
- Added `template_test` methods in addition to the already existing `template_filter` method family.
- Added `template_global` methods in addition to the already existing `template_filter` method family.
- Set the content-length header for x-sendfile.
- `tojson` filter now does not escape script blocks in HTML5 parsers.
- `tojson` used in templates is now safe by default. This was allowed due to the different escaping behavior.
- Flask will now raise an error if you attempt to register a new function on an already used endpoint.
- Added wrapper module around `simplejson` and added default serialization of datetime objects. This allows much easier customization of how JSON is handled by Flask or any Flask extension.

- Removed deprecated internal `flask.session` module alias. Use `flask.sessions` instead to get the session module. This is not to be confused with `flask.session` the session proxy.
- Templates can now be rendered without request context. The behavior is slightly different as the `request`, `session` and `g` objects will not be available and blueprint's context processors are not called.
- The config object is now available to the template as a real global and not through a context processor which makes it available even in imported templates by default.
- Added an option to generate non-ascii encoded JSON which should result in less bytes being transmitted over the network. It's disabled by default to not cause confusion with existing libraries that might expect `flask.json.dumps` to return bytes by default.
- `flask.g` is now stored on the app context instead of the request context.
- `flask.g` now gained a `get()` method for not erroring out on non existing items.
- `flask.g` now can be used with the `in` operator to see what's defined and it now is iterable and will yield all attributes stored.
- `flask.Flask.request_globals_class` got renamed to `flask.Flask.app_ctx_globals_class` which is a better name to what it does since 0.10.
- `request`, `session` and `g` are now also added as proxies to the template context which makes them available in imported templates. One has to be very careful with those though because usage outside of macros might cause caching.
- Flask will no longer invoke the wrong error handlers if a proxy exception is passed through.
- Added a workaround for chrome's cookies in localhost not working as intended with domain names.
- Changed logic for picking defaults for cookie values from sessions to work better with Google Chrome.
- Added `message_flashed` signal that simplifies flashing testing.
- Added support for copying of request contexts for better working with greenlets.
- Removed custom JSON HTTP exception subclasses. If you were relying on them you can reintroduce them again yourself trivially. Using them however is strongly discouraged as the interface was flawed.
- Python requirements changed: requiring Python 2.6 or 2.7 now to prepare for Python 3.3 port.
- Changed how the teardown system is informed about exceptions. This is now more reliable in case something handles an exception halfway through the error handling process.
- Request context preservation in debug mode now keeps the exception information around which means that teardown handlers are able to distinguish error from success cases.
- Added the `JSONIFY_PRETTYPRINT_REGULAR` configuration variable.
- Flask now orders JSON keys by default to not trash HTTP caches due to different hash seeds between different workers.
- Added `appcontext_pushed` and `appcontext_popped` signals.
- The builtin `run` method now takes the `SERVER_NAME` into account when picking the default port to run on.
- Added `flask.request.get_json()` as a replacement for the old `flask.request.json` property.

3.5.38 Version 0.9

Released 2012-07-01, codename Campari

- The `Request.on_json_loading_failed` now returns a JSON formatted response by default.
- The `url_for` function now can generate anchors to the generated links.
- The `url_for` function now can also explicitly generate URL rules specific to a given HTTP method.
- `Logger` now only returns the debug log setting if it was not set explicitly.
- Unregister a circular dependency between the WSGI environment and the request object when shutting down the request. This means that `environwerkzeug.request` will be `None` after the response was returned to the WSGI server but has the advantage that the garbage collector is not needed on CPython to tear down the request unless the user created circular dependencies themselves.
- `Session` is now stored after callbacks so that if the session payload is stored in the session you can still modify it in an after request callback.
- The `Flask` class will avoid importing the provided import name if it can (the required first parameter), to benefit tools which build Flask instances programmatically. The `Flask` class will fall back to using `import` on systems with custom module hooks, e.g. Google App Engine, or when the import name is inside a zip archive (usually an egg) prior to Python 2.7.
- Blueprints now have a decorator to add custom template filters application wide, `Blueprint.app_template_filter`.
- The `Flask` and `Blueprint` classes now have a non-decorator method for adding custom template filters application wide, `Flask.add_template_filter` and `Blueprint.add_app_template_filter`.
- The `get_flashed_messages` function now allows rendering flashed message categories in separate blocks, through a `category_filter` argument.
- The `Flask.run` method now accepts `None` for `host` and `port` arguments, using default values when `None`. This allows for calling run using configuration values, e.g. `app.run(app.config.get('MYHOST'), app.config.get('MYPORT'))`, with proper behavior whether or not a config file is provided.
- The `render_template` method now accepts either an iterable of template names or a single template name. Previously, it only accepted a single template name. On an iterable, the first template found is rendered.
- Added `Flask.app_context` which works very similar to the request context but only provides access to the current application. This also adds support for URL generation without an active request context.
- View functions can now return a tuple with the first instance being an instance of `Response`. This allows for returning `jsonify(error="error msg"), 400` from a view function.
- `Flask` and `Blueprint` now provide a `get_send_file_max_age` hook for subclasses to override behavior of serving static files from Flask when using `Flask.send_static_file` (used for the default static file handler) and `helpers.send_file`. This hook is provided a filename, which for example allows changing cache controls by file extension. The default max-age for `send_file` and static files can be configured through a new `SEND_FILE_MAX_AGE_DEFAULT` configuration variable, which is used in the default `get_send_file_max_age` implementation.
- Fixed an assumption in sessions implementation which could break message flashing on sessions implementations which use external storage.
- Changed the behavior of tuple return values from functions. They are no longer arguments to the response object, they now have a defined meaning.
- Added `Flask.request_globals_class` to allow a specific class to be used on creation of the `g` instance of each request.

- Added `required_methods` attribute to view functions to force-add methods on registration.
- Added `flask.after_this_request`.
- Added `flask.stream_with_context` and the ability to push contexts multiple times without producing unexpected behavior.

3.5.39 Version 0.8.1

Released 2012-07-01

- Fixed an issue with the undocumented `flask.session` module to not work properly on Python 2.5. It should not be used but did cause some problems for package managers.

3.5.40 Version 0.8

Released 2011-09-29, codename Rakija

- Refactored session support into a session interface so that the implementation of the sessions can be changed without having to override the Flask class.
- Empty session cookies are now deleted properly automatically.
- View functions can now opt out of getting the automatic `OPTIONS` implementation.
- HTTP exceptions and Bad Request errors can now be trapped so that they show up normally in the traceback.
- Flask in debug mode is now detecting some common problems and tries to warn you about them.
- Flask in debug mode will now complain with an assertion error if a view was attached after the first request was handled. This gives earlier feedback when users forget to import view code ahead of time.
- Added the ability to register callbacks that are only triggered once at the beginning of the first request with `Flask.before_first_request`.
- Malformed JSON data will now trigger a bad request HTTP exception instead of a value error which usually would result in a 500 internal server error if not handled. This is a backwards incompatible change.
- Applications now not only have a root path where the resources and modules are located but also an instance path which is the designated place to drop files that are modified at runtime (uploads etc.). Also this is conceptually only instance depending and outside version control so it's the perfect place to put configuration files etc.
- Added the `APPLICATION_ROOT` configuration variable.
- Implemented `TestClient.session_transaction` to easily modify sessions from the test environment.
- Refactored test client internally. The `APPLICATION_ROOT` configuration variable as well as `SERVER_NAME` are now properly used by the test client as defaults.
- Added `View.decorators` to support simpler decorating of pluggable (class-based) views.
- Fixed an issue where the test client if used with the “with” statement did not trigger the execution of the teardown handlers.
- Added finer control over the session cookie parameters.
- HEAD requests to a method view now automatically dispatch to the get method if no handler was implemented.
- Implemented the virtual `flask.ext` package to import extensions from.
- The context preservation on exceptions is now an integral component of Flask itself and no longer of the test client. This cleaned up some internal logic and lowers the odds of runaway request contexts in unittests.

- Fixed the Jinja2 environment's `list_templates` method not returning the correct names when blueprints or modules were involved.

3.5.41 Version 0.7.2

Released 2011-07-06

- Fixed an issue with URL processors not properly working on blueprints.

3.5.42 Version 0.7.1

Released 2011-06-29

- Added missing future import that broke 2.5 compatibility.
- Fixed an infinite redirect issue with blueprints.

3.5.43 Version 0.7

Released 2011-06-28, codename Grappa

- Added `Flask.make_default_options_response` which can be used by subclasses to alter the default behavior for `OPTIONS` responses.
- Unbound locals now raise a proper `RuntimeError` instead of an `AttributeError`.
- Mimetype guessing and etag support based on file objects is now deprecated for `send_file` because it was unreliable. Pass filenames instead or attach your own etags and provide a proper mimetype by hand.
- Static file handling for modules now requires the name of the static folder to be supplied explicitly. The previous autodetection was not reliable and caused issues on Google's App Engine. Until 1.0 the old behavior will continue to work but issue dependency warnings.
- Fixed a problem for Flask to run on jython.
- Added a `PROPAGATE_EXCEPTIONS` configuration variable that can be used to flip the setting of exception propagation which previously was linked to `DEBUG` alone and is now linked to either `DEBUG` or `TESTING`.
- Flask no longer internally depends on rules being added through the `add_url_rule` function and can now also accept regular werkzeug rules added to the url map.
- Added an `endpoint` method to the flask application object which allows one to register a callback to an arbitrary endpoint with a decorator.
- Use Last-Modified for static file sending instead of Date which was incorrectly introduced in 0.6.
- Added `create_jinja_loader` to override the loader creation process.
- Implemented a silent flag for `config.from_pyfile`.
- Added `teardown_request` decorator, for functions that should run at the end of a request regardless of whether an exception occurred. Also the behavior for `after_request` was changed. It's now no longer executed when an exception is raised.
- Implemented `has_request_context`.
- Deprecated `init_jinja_globals`. Override the `Flask.create_jinja_environment` method instead to achieve the same functionality.
- Added `safe_join`.

- The automatic JSON request data unpacking now looks at the `charset_mimetype` parameter.
- Don't modify the session on `get_flashed_messages` if there are no messages in the session.
- `before_request` handlers are now able to abort requests with errors.
- It is not possible to define user exception handlers. That way you can provide custom error messages from a central hub for certain errors that might occur during request processing (for instance database connection errors, timeouts from remote resources etc.).
- Blueprints can provide blueprint specific error handlers.
- Implemented generic class-based views.

3.5.44 Version 0.6.1

Released 2010-12-31

- Fixed an issue where the default `OPTIONS` response was not exposing all valid methods in the `Allow` header.
- Jinja2 template loading syntax now allows `./` in front of a template load path. Previously this caused issues with module setups.
- Fixed an issue where the subdomain setting for modules was ignored for the static folder.
- Fixed a security problem that allowed clients to download arbitrary files if the host server was a windows based operating system and the client uses backslashes to escape the directory the files were exposed from.

3.5.45 Version 0.6

Released 2010-07-27, codename Whisky

- After request functions are now called in reverse order of registration.
- `OPTIONS` is now automatically implemented by Flask unless the application explicitly adds `'OPTIONS'` as method to the URL rule. In this case no automatic `OPTIONS` handling kicks in.
- Static rules are now even in place if there is no static folder for the module. This was implemented to aid GAE which will remove the static folder if it's part of a mapping in the `.yaml` file.
- `Flask.config` is now available in the templates as `config`.
- Context processors will no longer override values passed directly to the render function.
- Added the ability to limit the incoming request data with the new `MAX_CONTENT_LENGTH` configuration value.
- The endpoint for the `Module.add_url_rule` method is now optional to be consistent with the function of the same name on the application object.
- Added a `make_response` function that simplifies creating response object instances in views.
- Added signalling support based on blinker. This feature is currently optional and supposed to be used by extensions and applications. If you want to use it, make sure to have `blinker` installed.
- Refactored the way URL adapters are created. This process is now fully customizable with the `Flask.create_url_adapter` method.
- Modules can now register for a subdomain instead of just an URL prefix. This makes it possible to bind a whole module to a configurable subdomain.

3.5.46 Version 0.5.2

Released 2010-07-15

- Fixed another issue with loading templates from directories when modules were used.

3.5.47 Version 0.5.1

Released 2010-07-06

- Fixes an issue with template loading from directories when modules were used.

3.5.48 Version 0.5

Released 2010-07-06, codename Calvados

- Fixed a bug with subdomains that was caused by the inability to specify the server name. The server name can now be set with the `SERVER_NAME` config key. This key is now also used to set the session cookie cross-subdomain wide.
- Autoescaping is no longer active for all templates. Instead it is only active for `.html`, `.htm`, `.xml` and `.xhtml`. Inside templates this behavior can be changed with the `autoescape` tag.
- Refactored Flask internally. It now consists of more than a single file.
- `send_file` now emits etags and has the ability to do conditional responses builtin.
- (temporarily) dropped support for zipped applications. This was a rarely used feature and led to some confusing behavior.
- Added support for per-package template and static-file directories.
- Removed support for `create_jinja_loader` which is no longer used in 0.5 due to the improved module support.
- Added a helper function to expose files from any directory.

3.5.49 Version 0.4

Released 2010-06-18, codename Rakia

- Added the ability to register application wide error handlers from modules.
- `Flask.after_request` handlers are now also invoked if the request dies with an exception and an error handling page kicks in.
- Test client has not the ability to preserve the request context for a little longer. This can also be used to trigger custom requests that do not pop the request stack for testing.
- Because the Python standard library caches loggers, the name of the logger is configurable now to better support unittests.
- Added `TESTING` switch that can activate unittesting helpers.
- The logger switches to `DEBUG` mode now if `debug` is enabled.

3.5.50 Version 0.3.1

Released 2010-05-28

- Fixed a error reporting bug with `Config.from_envvar`.
- Removed some unused code.
- Release does no longer include development leftover files (.git folder for themes, built documentation in zip and pdf file and some .pyc files)

3.5.51 Version 0.3

Released 2010-05-28, codename Schnaps

- Added support for categories for flashed messages.
- The application now configures a `logging.Handler` and will log request handling exceptions to that logger when not in debug mode. This makes it possible to receive mails on server errors for example.
- Added support for context binding that does not require the use of the with statement for playing in the console.
- The request context is now available within the with statement making it possible to further push the request context or pop it.
- Added support for configurations.

3.5.52 Version 0.2

Released 2010-05-12, codename J?germeister

- Various bugfixes
- Integrated JSON support
- Added `get_template_attribute` helper function.
- `Flask.add_url_rule` can now also register a view function.
- Refactored internal request dispatching.
- Server listens on 127.0.0.1 by default now to fix issues with chrome.
- Added external URL support.
- Added support for `send_file`.
- Module support and internal request handling refactoring to better support pluggable applications.
- Sessions can be set to be permanent now on a per-session basis.
- Better error reporting on missing secret keys.
- Added support for Google Appengine.

3.5.53 Version 0.1

Released 2010-04-16

- First public preview release.

PYTHON MODULE INDEX

f

flask, 181
flask.json, 251
flask.json.tag, 255

Symbols

`_AppCtxGlobals` (class in `flask.ctx`), 243

A

`abort()` (in module `flask`), 246

`aborter` (`flask.Flask` attribute), 182

`aborter_class` (`flask.Flask` attribute), 182

`accept_charset` (`flask.Request` property), 218

`accept_encodings` (`flask.Request` property), 218

`accept_languages` (`flask.Request` property), 218

`accept_mimetypes` (`flask.Request` property), 218

`accept_ranges` (`flask.Response` attribute), 226

`access_control_allow_credentials`
(`flask.Response` property), 226

`access_control_allow_headers` (`flask.Response` attribute), 227

`access_control_allow_methods` (`flask.Response` attribute), 227

`access_control_allow_origin` (`flask.Response` attribute), 227

`access_control_expose_headers` (`flask.Response` attribute), 227

`access_control_max_age` (`flask.Response` attribute), 227

`access_control_request_headers` (`flask.Request` attribute), 218

`access_control_request_method` (`flask.Request` attribute), 218

`access_route` (`flask.Request` property), 218

`accessed` (`flask.sessions.SecureCookieSession` attribute), 239

`accessed` (`flask.sessions.SessionMixin` attribute), 241

`add_app_template_filter()` (`flask.Blueprint` method), 207

`add_app_template_global()` (`flask.Blueprint` method), 207

`add_app_template_test()` (`flask.Blueprint` method), 208

`add_etag()` (`flask.Response` method), 227

`add_template_filter()` (`flask.Flask` method), 182

`add_template_global()` (`flask.Flask` method), 183

`add_template_test()` (`flask.Flask` method), 183

`add_url_rule()` (`flask.Blueprint` method), 208

`add_url_rule()` (`flask.blueprints.BlueprintSetupState` method), 264

`add_url_rule()` (`flask.Flask` method), 183

`after_app_request()` (`flask.Blueprint` method), 208

`after_request()` (`flask.Blueprint` method), 208

`after_request()` (`flask.Flask` method), 184

`after_request_funcs` (`flask.Blueprint` attribute), 209

`after_request_funcs` (`flask.Flask` attribute), 184

`after_this_request()` (in module `flask`), 247

`age` (`flask.Response` attribute), 227

`allow` (`flask.Response` property), 227

`app` (`flask.blueprints.BlueprintSetupState` attribute), 264

`app_context()` (`flask.Flask` method), 184

`app_context_processor()` (`flask.Blueprint` method), 209

`app_ctx_globals_class` (`flask.Flask` attribute), 185

`app_errorhandler()` (`flask.Blueprint` method), 209

`app_import_path` (`flask.cli.ScriptInfo` attribute), 274

`app_template_filter()` (`flask.Blueprint` method), 209

`app_template_global()` (`flask.Blueprint` method), 209

`app_template_test()` (`flask.Blueprint` method), 209

`app_url_defaults()` (`flask.Blueprint` method), 210

`app_url_value_preprocessor()` (`flask.Blueprint` method), 210

`AppContext` (class in `flask.ctx`), 264

`appcontext_popped` (in module `flask`), 267

`appcontext_pushed` (in module `flask`), 267

`appcontext_tearing_down` (in module `flask`), 267

`AppGroup` (class in `flask.cli`), 273

`application()` (`flask.Request` class method), 218

`APPLICATION_ROOT` (built-in variable), 79

`args` (`flask.Request` property), 218

`as_view()` (`flask.views.View` class method), 268

`async_to_sync()` (`flask.Flask` method), 185

`authorization` (`flask.Request` property), 218

`auto_find_instance_path()` (`flask.Flask` method), 185

B

`base_url` (`flask.Request` property), 219

`before_app_request()` (`flask.Blueprint` method), 210

before_request() (*flask.Blueprint method*), 210
before_request() (*flask.Flask method*), 185
before_request_funcs (*flask.Blueprint attribute*), 210
before_request_funcs (*flask.Flask attribute*), 185
Blueprint (*class in flask*), 206
blueprint (*flask.blueprints.BlueprintSetupState attribute*), 265
blueprint (*flask.Request property*), 219
blueprints (*flask.Flask attribute*), 185
blueprints (*flask.Request property*), 219
BlueprintSetupState (*class in flask.blueprints*), 264

C

cache_control (*flask.Request property*), 219
cache_control (*flask.Response property*), 227
calculate_content_length() (*flask.Response method*), 227
call_on_close() (*flask.Response method*), 227
charset (*flask.Request property*), 219
charset (*flask.Response property*), 228
check() (*flask.json.tag.JSONTag method*), 256
clear() (*flask.sessions.NullSession method*), 240
cli (*flask.Blueprint attribute*), 210
cli (*flask.Flask attribute*), 186
close() (*flask.Request method*), 219
close() (*flask.Response method*), 228
command() (*flask.cli.AppGroup method*), 273
compact (*flask.json.provider.DefaultJSONProvider attribute*), 254
Config (*class in flask*), 259
config (*flask.Flask attribute*), 186
config_class (*flask.Flask attribute*), 186
content_encoding (*flask.Request attribute*), 219
content_encoding (*flask.Response attribute*), 228
content_language (*flask.Response property*), 228
content_length (*flask.Request property*), 219
content_length (*flask.Response attribute*), 228
content_location (*flask.Response attribute*), 228
content_md5 (*flask.Request attribute*), 219
content_md5 (*flask.Response attribute*), 228
content_range (*flask.Response property*), 228
content_security_policy (*flask.Response property*), 228
content_security_policy_report_only (*flask.Response property*), 228
content_type (*flask.Request attribute*), 219
content_type (*flask.Response attribute*), 228
context_processor() (*flask.Blueprint method*), 210
context_processor() (*flask.Flask method*), 186
cookies (*flask.Request property*), 220
copy() (*flask.ctx.RequestContext method*), 263
copy_current_request_context() (*in module flask*), 245
create_app (*flask.cli.ScriptInfo attribute*), 274

create_global_jinja_loader() (*flask.Flask method*), 186
create_jinja_environment() (*flask.Flask method*), 186
create_url_adapter() (*flask.Flask method*), 186
cross_origin_embedder_policy (*flask.Response attribute*), 229
cross_origin_opener_policy (*flask.Response attribute*), 229
current_app (*in module flask*), 244

D

data (*flask.cli.ScriptInfo attribute*), 274
data (*flask.Request property*), 220
data (*flask.Response property*), 229
date (*flask.Request attribute*), 220
date (*flask.Response attribute*), 229
DEBUG (*built-in variable*), 77
debug (*flask.Flask property*), 187
decorators (*flask.views.View attribute*), 269
default() (*flask.json.provider.DefaultJSONProvider static method*), 254
default_config (*flask.Flask attribute*), 187
default_tags (*flask.json.tag.TaggedJSONSerializer attribute*), 255
DefaultJSONProvider (*class in flask.json.provider*), 254
delete() (*flask.Blueprint method*), 211
delete() (*flask.Flask method*), 187
delete_cookie() (*flask.Response method*), 229
dict_storage_class (*flask.Request attribute*), 220
digest_method() (*flask.sessions.SecureCookieSessionInterface static method*), 239
direct_passthrough (*flask.Response attribute*), 229
dispatch_request() (*flask.Flask method*), 187
dispatch_request() (*flask.views.MethodView method*), 269
dispatch_request() (*flask.views.View method*), 269
do_teardown_appcontext() (*flask.Flask method*), 187
do_teardown_request() (*flask.Flask method*), 188
dump() (*flask.json.provider.JSONProvider method*), 253
dump() (*in module flask.json*), 251
dumps() (*flask.json.provider.DefaultJSONProvider method*), 254
dumps() (*flask.json.provider.JSONProvider method*), 253
dumps() (*flask.json.tag.TaggedJSONSerializer method*), 255
dumps() (*in module flask.json*), 251

E

encoding_errors (*flask.Request property*), 220
endpoint (*flask.Request property*), 220
endpoint() (*flask.Blueprint method*), 211

endpoint() (*flask.Flask* method), 188
 ensure_ascii (*flask.json.provider.DefaultJSONProvider* attribute), 254
 ensure_sync() (*flask.Flask* method), 188
 environ (*flask.Request* attribute), 220
 environment variable
 FLASK_DEBUG, 198
 FLASK_ENV, 80
 YOURAPPLICATION_SETTINGS, 81
 error_handler_spec (*flask.Blueprint* attribute), 211
 error_handler_spec (*flask.Flask* attribute), 188
 errorhandler() (*flask.Blueprint* method), 211
 errorhandler() (*flask.Flask* method), 188
 expires (*flask.Response* attribute), 229
 EXPLAIN_TEMPLATE_LOADING (*built-in* variable), 80
 extensions (*flask.Flask* attribute), 189

F

files (*flask.Request* property), 220
 finalize_request() (*flask.Flask* method), 189
 first_registration (*flask.blueprints.BlueprintSetupState* attribute), 265
 flash() (*in module flask*), 250
 flask
 module, 181
 Flask (*class in flask*), 181
 flask.globals.app_ctx (*in module flask*), 264
 flask.globals.request_ctx (*in module flask*), 263
 flask.json
 module, 251
 flask.json.tag
 module, 255
 FLASK_DEBUG, 198
 FLASK_ENV, 80
 FlaskClient (*class in flask.testing*), 241
 FlaskCliRunner (*class in flask.testing*), 243
 FlaskGroup (*class in flask.cli*), 272
 force_type() (*flask.Response* class method), 229
 form (*flask.Request* property), 220
 form_data_parser_class (*flask.Request* attribute), 220
 freeze() (*flask.Response* method), 230
 from_app() (*flask.Response* class method), 230
 from_envvar() (*flask.Config* method), 259
 from_file() (*flask.Config* method), 259
 from_mapping() (*flask.Config* method), 260
 from_object() (*flask.Config* method), 260
 from_prefixed_env() (*flask.Config* method), 261
 from_pyfile() (*flask.Config* method), 261
 from_values() (*flask.Request* class method), 220
 full_dispatch_request() (*flask.Flask* method), 189
 full_path (*flask.Request* property), 221

G

g (*in module flask*), 243
 get() (*flask.Blueprint* method), 212
 get() (*flask.ctx._AppCtxGlobals* method), 244
 get() (*flask.Flask* method), 190
 get() (*flask.sessions.SecureCookieSession* method), 239
 get_app_iter() (*flask.Response* method), 230
 get_command() (*flask.cli.FlaskGroup* method), 272
 get_cookie_domain() (*flask.sessions.SessionInterface* method), 236
 get_cookie_httponly() (*flask.sessions.SessionInterface* method), 236
 get_cookie_name() (*flask.sessions.SessionInterface* method), 237
 get_cookie_path() (*flask.sessions.SessionInterface* method), 237
 get_cookie_samesite() (*flask.sessions.SessionInterface* method), 237
 get_cookie_secure() (*flask.sessions.SessionInterface* method), 237
 get_data() (*flask.Request* method), 221
 get_data() (*flask.Response* method), 231
 get_etag() (*flask.Response* method), 231
 get_expiration_time() (*flask.sessions.SessionInterface* method), 237
 get_flashed_messages() (*in module flask*), 250
 get_json() (*flask.Request* method), 221
 get_json() (*flask.Response* method), 231
 get_namespace() (*flask.Config* method), 261
 get_send_file_max_age() (*flask.Blueprint* method), 212
 get_send_file_max_age() (*flask.Flask* method), 190
 get_template_attribute() (*in module flask*), 258
 get_wsgi_headers() (*flask.Response* method), 231
 get_wsgi_response() (*flask.Response* method), 231
 got_first_request (*flask.Flask* property), 190
 got_request_exception (*in module flask*), 266
 group() (*flask.cli.AppGroup* method), 273

H

handle_exception() (*flask.Flask* method), 190
 handle_http_exception() (*flask.Flask* method), 190
 handle_url_build_error() (*flask.Flask* method), 191
 handle_user_exception() (*flask.Flask* method), 191
 has_app_context() (*in module flask*), 245
 has_request_context() (*in module flask*), 244
 has_static_folder (*flask.Blueprint* property), 212
 has_static_folder (*flask.Flask* property), 191
 headers (*flask.Request* attribute), 222
 host (*flask.Request* property), 222
 host_url (*flask.Request* property), 222

I

`if_match` (*flask.Request* property), 222
`if_modified_since` (*flask.Request* property), 222
`if_none_match` (*flask.Request* property), 222
`if_range` (*flask.Request* property), 222
`if_unmodified_since` (*flask.Request* property), 222
`import_name` (*flask.Blueprint* attribute), 212
`import_name` (*flask.Flask* attribute), 191
`init_every_request` (*flask.views.View* attribute), 269
`inject_url_defaults` (*flask.Flask* method), 191
`input_stream` (*flask.Request* attribute), 222
`instance_path` (*flask.Flask* attribute), 192
`invoke` (*flask.testing.FlaskCliRunner* method), 243
`is_json` (*flask.Request* property), 222
`is_json` (*flask.Response* property), 232
`is_multiprocess` (*flask.Request* attribute), 222
`is_multithread` (*flask.Request* attribute), 222
`is_null_session` (*flask.sessions.SessionInterface* method), 237
`is_run_once` (*flask.Request* attribute), 222
`is_secure` (*flask.Request* property), 223
`is_sequence` (*flask.Response* property), 232
`is_streamed` (*flask.Response* property), 232
`iter_blueprints` (*flask.Flask* method), 192
`iter_encoded` (*flask.Response* method), 232

J

`jinja_env` (*flask.Flask* property), 192
`jinja_environment` (*flask.Flask* attribute), 192
`jinja_loader` (*flask.Blueprint* property), 212
`jinja_loader` (*flask.Flask* property), 192
`jinja_options` (*flask.Flask* attribute), 192
`json` (*flask.Flask* attribute), 192
`json` (*flask.Request* property), 223
`json` (*flask.Response* property), 232
`json_provider_class` (*flask.Flask* attribute), 192
`jsonify` (*in module flask.json*), 251
`JSONProvider` (*class in flask.json.provider*), 252
`JSONTag` (*class in flask.json.tag*), 256

K

`key` (*flask.json.tag.JSONTag* attribute), 256
`key_derivation` (*flask.sessions.SecureCookieSessionInterface* attribute), 239

L

`last_modified` (*flask.Response* attribute), 232
`list_commands` (*flask.cli.FlaskGroup* method), 272
`list_storage_class` (*flask.Request* attribute), 223
`load` (*flask.json.provider.JSONProvider* method), 253
`load` (*in module flask.json*), 252
`load_app` (*flask.cli.ScriptInfo* method), 274
`load_dotenv` (*in module flask.cli*), 274

`loads` (*flask.json.provider.DefaultJSONProvider* method), 255
`loads` (*flask.json.provider.JSONProvider* method), 253
`loads` (*flask.json.tag.TaggedJSONSerializer* method), 256
`loads` (*in module flask.json*), 252
`location` (*flask.Response* attribute), 232
`log_exception` (*flask.Flask* method), 192
`logger` (*flask.Flask* property), 193

M

`make_aborter` (*flask.Flask* method), 193
`make_conditional` (*flask.Response* method), 232
`make_config` (*flask.Flask* method), 193
`make_context` (*flask.cli.FlaskGroup* method), 273
`make_default_options_response` (*flask.Flask* method), 193
`make_form_data_parser` (*flask.Request* method), 223
`make_null_session` (*flask.sessions.SessionInterface* method), 237
`make_response` (*flask.Flask* method), 193
`make_response` (*in module flask*), 247
`make_sequence` (*flask.Response* method), 233
`make_setup_state` (*flask.Blueprint* method), 212
`make_shell_context` (*flask.Flask* method), 194
`match_request` (*flask.ctx.RequestContext* method), 263
`MAX_CONTENT_LENGTH` (*built-in variable*), 79
`max_content_length` (*flask.Request* property), 223
`MAX_COOKIE_SIZE` (*built-in variable*), 80
`max_cookie_size` (*flask.Response* property), 233
`max_forwards` (*flask.Request* attribute), 223
`message_flashed` (*in module flask*), 267
`method` (*flask.Request* attribute), 223
`methods` (*flask.views.View* attribute), 269
`MethodView` (*class in flask.views*), 269
`mimetype` (*flask.json.provider.DefaultJSONProvider* attribute), 254
`mimetype` (*flask.Request* property), 223
`mimetype` (*flask.Response* property), 233
`mimetype_params` (*flask.Request* property), 223
`mimetype_params` (*flask.Response* property), 233
`modified` (*flask.session* attribute), 235
`modified` (*flask.sessions.SecureCookieSession* attribute), 240
`modified` (*flask.sessions.SessionMixin* attribute), 241
`module`
 flask, 181
 flask.json, 251
 flask.json.tag, 255

N

name (*flask.Flask* property), 194
 new (*flask.session* attribute), 235
 null_session_class (*flask.sessions.SessionInterface* attribute), 238
 NullSession (*class in flask.sessions*), 240

O

on_json_loading_failed() (*flask.Request* method), 223
 open() (*flask.testing.FlaskClient* method), 242
 open_instance_resource() (*flask.Flask* method), 194
 open_resource() (*flask.Blueprint* method), 213
 open_resource() (*flask.Flask* method), 194
 open_session() (*flask.sessions.SecureCookieSessionInterface* method), 239
 open_session() (*flask.sessions.SessionInterface* method), 238
 options (*flask.blueprints.BlueprintSetupState* attribute), 265
 origin (*flask.Request* attribute), 223

P

parameter_storage_class (*flask.Request* attribute), 224
 parse_args() (*flask.cli.FlaskGroup* method), 273
 pass_script_info() (*in module flask.cli*), 274
 patch() (*flask.Blueprint* method), 213
 patch() (*flask.Flask* method), 195
 path (*flask.Request* attribute), 224
 permanent (*flask.session* attribute), 236
 permanent (*flask.sessions.SessionMixin* property), 241
 PERMANENT_SESSION_LIFETIME (*built-in variable*), 79
 permanent_session_lifetime (*flask.Flask* attribute), 195
 pickle_based (*flask.sessions.SessionInterface* attribute), 238
 pop() (*flask.ctx.AppCtxGlobals* method), 244
 pop() (*flask.ctx.AppContext* method), 264
 pop() (*flask.ctx.RequestContext* method), 263
 pop() (*flask.sessions.NullSession* method), 240
 popitem() (*flask.sessions.NullSession* method), 240
 post() (*flask.Blueprint* method), 213
 post() (*flask.Flask* method), 195
 pragma (*flask.Request* property), 224
 PREFERRED_URL_SCHEME (*built-in variable*), 79
 preprocess_request() (*flask.Flask* method), 195
 process_response() (*flask.Flask* method), 195
 PROPAGATE_EXCEPTIONS (*built-in variable*), 77
 provide_automatic_options (*flask.views.View* attribute), 269
 push() (*flask.ctx.AppContext* method), 264
 put() (*flask.Blueprint* method), 213

put() (*flask.Flask* method), 195

Python Enhancement Proposals

PEP 302, 304
 PEP 3333, 95, 298
 PEP 451, 298
 PEP 519, 297

Q

query_string (*flask.Request* attribute), 224

R

range (*flask.Request* property), 224
 record() (*flask.Blueprint* method), 214
 record_once() (*flask.Blueprint* method), 214
 redirect() (*flask.Flask* method), 196
 redirect() (*in module flask*), 246
 referrer (*flask.Request* attribute), 224
 register() (*flask.Blueprint* method), 214
 register() (*flask.json.tag.TaggedJSONSerializer* method), 256
 register_blueprint() (*flask.Blueprint* method), 214
 register_blueprint() (*flask.Flask* method), 196
 register_error_handler() (*flask.Blueprint* method), 214
 register_error_handler() (*flask.Flask* method), 196
 remote_addr (*flask.Request* attribute), 224
 remote_user (*flask.Request* attribute), 224
 render_template() (*in module flask*), 257
 render_template_string() (*in module flask*), 257
 Request (*class in flask*), 217
 request (*in module flask*), 226
 request_class (*flask.Flask* attribute), 196
 request_context() (*flask.Flask* method), 196
 request_finished (*in module flask*), 266
 request_started (*in module flask*), 265
 request_tearing_down (*in module flask*), 266
 RequestContext (*class in flask.ctx*), 263
 Response (*class in flask*), 226
 response (*flask.Response* attribute), 233
 response() (*flask.json.provider.DefaultJSONProvider* method), 255
 response() (*flask.json.provider.JSONProvider* method), 253
 response_class (*flask.Flask* attribute), 197
 retry_after (*flask.Response* property), 233
 RFC
 RFC 2231, 249
 RFC 822, 254
 RFC 8259, 301
 root_path (*flask.Blueprint* attribute), 215
 root_path (*flask.Flask* attribute), 197
 root_path (*flask.Request* attribute), 224
 root_url (*flask.Request* property), 224
 route() (*flask.Blueprint* method), 215

route() (*flask.Flask* method), 197
 routing_exception (*flask.Request* attribute), 224
 run() (*flask.Flask* method), 197
 run_command (*in module flask.cli*), 275

S

salt (*flask.sessions.SecureCookieSessionInterface* attribute), 239
 save_session() (*flask.sessions.SecureCookieSessionInterface* method), 239
 save_session() (*flask.sessions.SessionInterface* method), 238
 scheme (*flask.Request* attribute), 224
 script_root (*flask.Request* property), 224
 ScriptInfo (*class in flask.cli*), 273
 SECRET_KEY (*built-in variable*), 78
 secret_key (*flask.Flask* attribute), 198
 SecureCookieSession (*class in flask.sessions*), 239
 SecureCookieSessionInterface (*class in flask.sessions*), 238
 select_jinja_autoescape() (*flask.Flask* method), 198
 send_file() (*in module flask*), 248
 SEND_FILE_MAX_AGE_DEFAULT (*built-in variable*), 79
 send_from_directory() (*in module flask*), 249
 send_static_file() (*flask.Blueprint* method), 215
 send_static_file() (*flask.Flask* method), 198
 serializer (*flask.sessions.SecureCookieSessionInterface* attribute), 239
 server (*flask.Request* attribute), 224
 SERVER_NAME (*built-in variable*), 79
 session (*class in flask*), 235
 session_class (*flask.sessions.SecureCookieSessionInterface* attribute), 239
 SESSION_COOKIE_DOMAIN (*built-in variable*), 78
 SESSION_COOKIE_HTTPONLY (*built-in variable*), 78
 SESSION_COOKIE_NAME (*built-in variable*), 78
 SESSION_COOKIE_PATH (*built-in variable*), 78
 SESSION_COOKIE_SAMESITE (*built-in variable*), 78
 SESSION_COOKIE_SECURE (*built-in variable*), 78
 session_interface (*flask.Flask* attribute), 198
 SESSION_REFRESH_EACH_REQUEST (*built-in variable*), 79
 session_transaction() (*flask.testing.FlaskClient* method), 242
 SessionInterface (*class in flask.sessions*), 236
 SessionMixin (*class in flask.sessions*), 241
 set_cookie() (*flask.Response* method), 233
 set_data() (*flask.Response* method), 234
 set_etag() (*flask.Response* method), 234
 setdefault() (*flask.ctx.AppCtxGlobals* method), 244
 setdefault() (*flask.sessions.NullSession* method), 241
 setdefault() (*flask.sessions.SecureCookieSession* method), 240

shallow (*flask.Request* attribute), 224
 shell_command (*in module flask.cli*), 275
 shell_context_processor() (*flask.Flask* method), 199
 shell_context_processors (*flask.Flask* attribute), 199
 should_ignore_error() (*flask.Flask* method), 199
 should_set_cookie() (*flask.sessions.SessionInterface* method), 238
 signals.signals_available (*in module flask*), 268
 sort_keys (*flask.json.provider.DefaultJSONProvider* attribute), 254
 static_folder (*flask.Blueprint* property), 215
 static_folder (*flask.Flask* property), 199
 static_url_path (*flask.Blueprint* property), 215
 static_url_path (*flask.Flask* property), 199
 status (*flask.Response* property), 234
 status_code (*flask.Response* property), 234
 stream (*flask.Request* property), 225
 stream (*flask.Response* property), 234
 stream_template() (*in module flask*), 258
 stream_template_string() (*in module flask*), 258
 stream_with_context() (*in module flask*), 262
 subdomain (*flask.blueprints.BlueprintSetupState* attribute), 265

T

tag() (*flask.json.tag.JSONTag* method), 256
 tag() (*flask.json.tag.TaggedJSONSerializer* method), 256
 TaggedJSONSerializer (*class in flask.json.tag*), 255
 teardown_app_request() (*flask.Blueprint* method), 215
 teardown_appcontext() (*flask.Flask* method), 199
 teardown_appcontext_funcs (*flask.Flask* attribute), 199
 teardown_request() (*flask.Blueprint* method), 216
 teardown_request() (*flask.Flask* method), 200
 teardown_request_funcs (*flask.Blueprint* attribute), 216
 teardown_request_funcs (*flask.Flask* attribute), 200
 template_context_processors (*flask.Blueprint* attribute), 216
 template_context_processors (*flask.Flask* attribute), 200
 template_filter() (*flask.Flask* method), 200
 template_folder (*flask.Blueprint* attribute), 216
 template_folder (*flask.Flask* attribute), 201
 template_global() (*flask.Flask* method), 201
 template_rendered (*in module flask*), 265
 template_test() (*flask.Flask* method), 201
 TEMPLATES_AUTO_RELOAD (*built-in variable*), 79
 test_cli_runner() (*flask.Flask* method), 201
 test_cli_runner_class (*flask.Flask* attribute), 201

[test_client\(\)](#) (*flask.Flask method*), 202
[test_client_class](#) (*flask.Flask attribute*), 202
[test_request_context\(\)](#) (*flask.Flask method*), 202
 TESTING (*built-in variable*), 77
[testing](#) (*flask.Flask attribute*), 203
[to_json\(\)](#) (*flask.json.tag.JSONTag method*), 257
[to_python\(\)](#) (*flask.json.tag.JSONTag method*), 257
 TRAP_BAD_REQUEST_ERRORS (*built-in variable*), 78
[trap_http_exception\(\)](#) (*flask.Flask method*), 203
 TRAP_HTTP_EXCEPTIONS (*built-in variable*), 77

U

[untag\(\)](#) (*flask.json.tag.TaggedJSONSerializer method*), 256
[update\(\)](#) (*flask.sessions.NullSession method*), 241
[update_template_context\(\)](#) (*flask.Flask method*), 204
[url](#) (*flask.Request property*), 225
[url_build_error_handlers](#) (*flask.Flask attribute*), 204
[url_charset](#) (*flask.Request property*), 225
[url_default_functions](#) (*flask.Blueprint attribute*), 216
[url_default_functions](#) (*flask.Flask attribute*), 204
[url_defaults](#) (*flask.blueprints.BlueprintSetupState attribute*), 265
[url_defaults\(\)](#) (*flask.Blueprint method*), 216
[url_defaults\(\)](#) (*flask.Flask method*), 204
[url_for\(\)](#) (*flask.Flask method*), 204
[url_for\(\)](#) (*in module flask*), 245
[url_map](#) (*flask.Flask attribute*), 205
[url_map_class](#) (*flask.Flask attribute*), 205
[url_prefix](#) (*flask.blueprints.BlueprintSetupState attribute*), 265
[url_root](#) (*flask.Request property*), 225
[url_rule](#) (*flask.Request attribute*), 225
[url_rule_class](#) (*flask.Flask attribute*), 205
[url_value_preprocessor\(\)](#) (*flask.Blueprint method*), 217
[url_value_preprocessor\(\)](#) (*flask.Flask method*), 205
[url_value_preprocessors](#) (*flask.Blueprint attribute*), 217
[url_value_preprocessors](#) (*flask.Flask attribute*), 206
 USE_X_SENDFILE (*built-in variable*), 79
[user_agent](#) (*flask.Request property*), 225
[user_agent_class](#) (*flask.Request attribute*), 225

V

[values](#) (*flask.Request property*), 225
[vary](#) (*flask.Response property*), 235
[View](#) (*class in flask.views*), 268
[view_args](#) (*flask.Request attribute*), 226
[view_functions](#) (*flask.Blueprint attribute*), 217
[view_functions](#) (*flask.Flask attribute*), 206

W

[want_form_data_parsed](#) (*flask.Request property*), 226
[with_appcontext\(\)](#) (*in module flask.cli*), 274
[wsgi_app\(\)](#) (*flask.Flask method*), 206
[www_authenticate](#) (*flask.Response property*), 235

Y

YOURAPPLICATION_SETTINGS, 81