Flask supports Python 3.8 and newer.

Dependencies

These distributions will be installed automatically when installing Flask.

* [Werkzeug](https://palletsprojects.com/p/werkzeug/) implements WSGI, the standard Python interface between applications and servers.
* [Jinja](https://palletsprojects.com/p/jinja/) is a template language that renders the pages your application serves.
* [MarkupSafe](https://palletsprojects.com/p/markupsafe/) comes with Jinja. It escapes untrusted input when rendering templates to avoid injection attacks.
* [ItsDangerous](https://palletsprojects.com/p/itsdangerous/) securely signs data to ensure its integrity. This is used to protect Flask’s session cookie.
* [Click](https://palletsprojects.com/p/click/) is a framework for writing command line applications. It provides the flask command and allows adding custom management commands.
* [Blinker](https://blinker.readthedocs.io/) provides support for [Signals](https://flask.palletsprojects.com/en/3.0.x/signals/).

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

* [python-dotenv](https://github.com/theskumar/python-dotenv#readme) enables support for [Environment Variables From dotenv](https://flask.palletsprojects.com/en/3.0.x/cli/#dotenv) when running flask commands.
* [Watchdog](https://pythonhosted.org/watchdog/) provides a faster, more efficient reloader for the development server.

## 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**](https://docs.python.org/3/library/venv.html#module-venv) module to create virtual environments.

### **Create an environment**

Create a project folder and a .venv folder within:

macOS/LinuxWindows

$ mkdir myproject

$ cd myproject

$ python3 -m venv .venv

### **Activate the environment**

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

macOS/LinuxWindows

$ . .venv/bin/activate

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

## Install Flask

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

$ pip install Flask

## 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>"

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](https://flask.palletsprojects.com/en/3.0.x/cli/) for more details.

If another program is already using port 5000, you’ll see OSError: [Errno 98] or OSError: [WinError 10013] when the server tries to start.

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.

## 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.

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

## 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.

## 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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.route) decorator to bind a function to a URL.

@app.route**(**'/'**)**

**def** index**():**

**return** 'Index Page'

@app.route**(**'/hello'**)**

**def** hello**():**

**return** 'Hello, World'

### **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**):**

*# 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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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.

**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'**))**

### **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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**()**

If GET is present, Flask automatically adds support for the HEAD method and handles HEAD requests according to the [HTTP RFC](https://www.ietf.org/rfc/rfc2068.txt). Likewise, OPTIONS is automatically implemented for you.

## 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.

## 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](https://palletsprojects.com/p/jinja/) 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.

**from** flask **import** render\_template

@app.route**(**'/hello/'**)**

@app.route**(**'/hello/<name>'**)**

**def** hello**(**name=**None):**

**return** render\_template**(**'hello.html'**,** name=name**)**

### **The Request Object**

The request object is documented in the API section and we will not cover it here in detail (see [**Request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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

@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:**

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**)**

### **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()**](https://werkzeug.palletsprojects.com/en/3.0.x/datastructures/#werkzeug.datastructures.FileStorage.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'**)**

### **Cookies**

To access cookies you can use the [**cookies**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Request.cookies) attribute. To set cookies you can use the [**set\_cookie**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Response.set_cookie) method of response objects. The [**cookies**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Request.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](https://flask.palletsprojects.com/en/3.0.x/quickstart/#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

## Redirects and Errors

To redirect a user to another endpoint, use the [**redirect()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.redirect) function; to abort a request early with an error code, use the [**abort()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.errorhandler) decorator:

**from** flask **import** render\_template

@app.errorhandler**(**404**)**

**def** page\_not\_found**(**error**):**

**return** render\_template**(**'page\_not\_found.html'**),** 404

## 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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.json.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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

## Sessions

In addition to the request object there is also a second object called [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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>

<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'**))**

## 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**](https://werkzeug.palletsprojects.com/en/3.0.x/middleware/proxy_fix/#werkzeug.middleware.proxy_fix.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.

# **Application Setup**

A Flask application is an instance of the [**Flask**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask) class. Everything about the application, such as configuration and URLs, will be registered with this class.

**import** os

**from** flask **import** Flask

**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 a Blueprint

A [**Blueprint**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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.

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'**)**

# **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.

Describe the Project

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

pyproject.toml

**[project]**

name = "flaskr"

version = "1.0.0"

description = "The basic blog app built in the Flask tutorial."

dependencies = **[**

"flask"**,**

**]**

**[build-system]**

requires = **[**"flit\_core<4"**]**

build-backend = "flit\_core.buildapi"

## Install the Project

Use pip to install your project in the virtual environment.

$ pip install -e .

## 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](https://flask.palletsprojects.com/en/3.0.x/installation/#install-create-env), 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.

## 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](https://docs.pylonsproject.org/projects/waitress/en/stable/), 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

## 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**](https://werkzeug.palletsprojects.com/en/3.0.x/exceptions/#werkzeug.exceptions.HTTPException).

### **Registering**

Register handlers by decorating a function with [**errorhandler()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.errorhandler). Or use [**register\_error\_handler()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**)**

## 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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.json.jsonify) we can return JSON responses to API errors. [**abort()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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**):**

**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**)**

app.run**(**

debug=**True,** passthrough\_errors=**True,**

use\_debugger=**False,** use\_reloader=**False**

**)**

## 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.

**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**):**

DATABASE\_URI = 'sqlite:///:memory:'

TESTING = **True**

To enable such a config you just have to call into [**from\_object()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Config.from_object):

app.config.from\_object**(**'configmodule.ProductionConfig'**)**

# **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.

## 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**](https://blinker.readthedocs.io/en/stable/#blinker.base.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'**)**

## Sending Signals

If you want to emit a signal, you can do so by calling the [**send()**](https://blinker.readthedocs.io/en/stable/#blinker.base.Signal.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.

## 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}'**)**

## 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.

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.

## 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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.wsgi_app).
2. A [**RequestContext**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.ctx.RequestContext) object is created. This converts the WSGI environ dict into a [**Request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Request) object. It also creates an **AppContext** object.
3. The [app context](https://flask.palletsprojects.com/en/3.0.x/appcontext/) is pushed, which makes [**current\_app**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.current_app) and [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g) available.
4. The [**appcontext\_pushed**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.appcontext_pushed) signal is sent.
5. The [request context](https://flask.palletsprojects.com/en/3.0.x/reqcontext/) is pushed, which makes [**request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request) and [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session) available.
6. The session is opened, loading any existing session data using the app’s [**session\_interface**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.session_interface), an instance of [**SessionInterface**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.sessions.SessionInterface).
7. The URL is matched against the URL rules registered with the [**route()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request_started) signal is sent.
9. Any [**url\_value\_preprocessor()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.url_value_preprocessor) decorated functions are called.
10. Any [**before\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Response) object.
15. Any [**after\_this\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.after_this_request) decorated functions are called, then cleared.
16. Any [**after\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.session_interface).
18. The [**request\_finished**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_request) decorated functions are called.
22. The [**request\_tearing\_down**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request_tearing_down) signal is sent.
23. The request context is popped, [**request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request) and [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session) are no longer available.
24. Any [**teardown\_appcontext()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_appcontext) decorated functions are called.
25. The [**appcontext\_tearing\_down**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.appcontext_tearing_down) signal is sent.
26. The app context is popped, [**current\_app**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.current_app) and [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g) are no longer available.
27. The [**appcontext\_popped**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.appcontext_popped) signal is sent.

Flask uses *contexts* to temporarily make certain objects globally accessible.

Contexts enable Flask to make certain variables globally accessible to a thread without interfering with the other threads.

There are two contexts in Flask: the *application context* and the *request context*. Table 2-1

shows the variables exposed by each of these contexts.

*Table 2-1. Flask context globals*

Variable name Context Description

**current\_app -> Application context -> The application instance for the active application.**

**g ->Application context-> An object that the application can use for temporary storage during the handling of**

**a request. This variable is reset with each request.**

**Request -> Request context -> The request object, which encapsulates the contents of a HTTP request sent by the**

**client.**

**session -> Request context -> The user session, a dictionary that the application can use to store values that are**

**“remembered” between requests.**

Flask activates (or *pushes*) the application and request contexts before dispatching a

request and then removes them when the request is handled. When the application

context is pushed, the current\_app and g variables become available to the thread;

likewise, when the request context is pushed, request and session become available

as well. If any of these variables are accessed without an active application or request

context, an error is generated.

# **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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.current_app) and [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g) proxies are accessed instead.

## Purpose of the Context

The [**Flask**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask) application object has attributes, such as [**config**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.config), that are useful to access within views and [CLI commands](https://flask.palletsprojects.com/en/3.0.x/cli/). However, importing the app instance within the modules in your project is prone to circular import issues. When using the [app factory pattern](https://flask.palletsprojects.com/en/3.0.x/patterns/appfactories/) or writing reusable [blueprints](https://flask.palletsprojects.com/en/3.0.x/blueprints/) or [extensions](https://flask.palletsprojects.com/en/3.0.x/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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.current_app) proxy, which points to the application handling the current activity.

## 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](https://flask.palletsprojects.com/en/3.0.x/reqcontext/). 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.

## Manually Push a Context

If you try to access [**current\_app**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.app_context) in a with block, and everything that runs in the block will have access to [**current\_app**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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.

## Storing Data

The application context is a good place to store common data during a request or CLI command. Flask provides the [**g object**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g) 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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session) or a database to store data across requests.

A common use for [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**()**

# **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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request) and [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session) proxies are accessed instead.

This is similar to [The Application Context](https://flask.palletsprojects.com/en/3.0.x/appcontext/), which keeps track of the application-level data independent of a request. A corresponding application context is pushed when a request context is pushed.

## Purpose of the Context

When the [**Flask**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask) application handles a request, it creates a [**Request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request) proxy, which points to the request object for the current request.

## Lifetime of the Context

When a Flask application begins handling a request, it pushes a request context, which also pushes an [app context](https://flask.palletsprojects.com/en/3.0.x/appcontext/). 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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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**](https://docs.python.org/3/library/contextvars.html#module-contextvars) and Werkzeug’s [**LocalProxy**](https://werkzeug.palletsprojects.com/en/3.0.x/local/#werkzeug.local.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.

## How the Context Works

The [**Flask.wsgi\_app()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.ctx.RequestContext) is created and pushed, which creates and pushes an [**AppContext**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.ctx.AppContext) first if a context for that application is not already the top context. While these contexts are pushed, the [**current\_app**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.current_app), [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g), [**request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request), and [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_request) and [**teardown\_appcontext()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_appcontext) functions are executed. These execute even if an unhandled exception occurred during dispatch.

## 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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_request) and [**teardown\_appcontext()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request_started) is sent before the [**before\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.before_request) functions are called.
2. [**request\_finished**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request_finished) is sent after the [**after\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.after_request) functions are called.
3. [**got\_request\_exception**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.got_request_exception) is sent when an exception begins to be handled, but before an [**errorhandler()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.errorhandler) is looked up or called.
4. [**request\_tearing\_down**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request_tearing_down) is sent after the [**teardown\_request()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.teardown_request) functions are called.

## 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.

## 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**)**

## 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**)**

## 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

# **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.

## 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.

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.

## 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](https://github.com/pallets/flask/issues/248#issuecomment-59934857), so this behavior was changed and **jsonify()** now supports serializing arrays.

## 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](https://github.com/GoogleCloudPlatform/flask-talisman) extension can be used to manage HTTPS and the security headers for you.

### **HTTP Strict Transport Security (HSTS)**[**¶**](https://flask.palletsprojects.com/en/3.0.x/security/#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'

### **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'"

## 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](https://github.com/pallets/flask/issues/248#issuecomment-59934857), so this behavior was changed and **jsonify()** now supports serializing arrays.

## 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](https://github.com/GoogleCloudPlatform/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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session.permanent) is set, then [**PERMANENT\_SESSION\_LIFETIME**](https://flask.palletsprojects.com/en/3.0.x/config/#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>

Bottom of Form

## 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**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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'**])**

## 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](https://flask.palletsprojects.com/en/3.0.x/cli/) for more detail.

# **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.

## 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

**})**

# **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**()**

...

## 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](https://flask-sqlalchemy.palletsprojects.com/) from [PyPI](https://pypi.org/project/Flask-SQLAlchemy/).

## 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](https://docs.sqlalchemy.org/en/latest/orm/extensions/declarative/) extension.

Here’s the example database.py module for your application:

**from** sqlalchemy **import** create\_engine

**from** sqlalchemy.orm **import** scoped\_session**,** sessionmaker**,** 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**)**

**from** sqlalchemy **import** Column**,** Integer**,** String

**from** yourapplication.database **import** Base

**class** User**(**Base**):**

\_\_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}>'

# **Class-based Views**

This page introduces using the [**View**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.views.View) and [**MethodView**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.views.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.

## 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.

# **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()**](https://werkzeug.palletsprojects.com/en/3.0.x/datastructures/#werkzeug.datastructures.FileStorage.save) method of the file to save the file permanently somewhere on the filesystem.

**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>

'''

Improving Uploads

Changelog

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()**](https://docs.python.org/3/library/tempfile.html#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

## Login Required Decorator

**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

## Caching Decorator

**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

## 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'

**else:**

flash**(**'You were successfully logged in'**)**

**return** redirect**(**url\_for**(**'index'**))**

**return** render\_template**(**'login.html'**,** error=error**)**

# **Subclassing Flask**

The [**Flask**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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.

Request Dispatching

When the application receives a request from a client, it needs to find what view function

to invoke to service it. For this task, Flask looks up the URL given in the request in the

application’s *URL map*, which contains a mapping of URLs to the view functions that

handle them. Flask builds this map using the app.route decorators or the equivalent

nondecorator version app.add\_url\_rule().

Request Hooks

Sometimes it is useful to execute code before or after each request is processed. For

example, at the start of each request it may be necessary to create a database connection,

or authenticate the user making the request. Instead of duplicating the code that does

this in every view function, Flask gives you the option to register common functions to

be invoked before or after a request is dispatched to a view function.

Request hooks are implemented as decorators. These are the four hooks supported by

Flask:

• before\_first\_request: Register a function to run before the first request is

handled.

• before\_request: Register a function to run before each request.

• after\_request: Register a function to run after each request, if no unhandled exceptions

occurred.

• teardown\_request: Register a function to run after each request, even if unhandled

exceptions occurred.

A common pattern to share data between request hook functions and view functions is

to use the g context global. For example, a before\_request handler can load the loggedin

user from the database and store it in g.user. Later, when the view function is invoked,

it can access the user from there.

Instead of returning one, two, or three values as a tuple, Flask view functions have the

option of returning a Response object. The make\_response() function takes one, two,

or three arguments, the same values that can be returned from a view function, and

returns a Response object.

A redirect is typically indicated with a 302 response status code and the URL to redirect

to given in a Location header. A redirect response can be generated using a three-value

return, or also with a Response object, but given its frequent use, Flask provides a

redirect() helper function that creates this response:

**from flask import** redirect

@app.route('/')

**def** index():

**return** redirect('http://www.example.com')

Another special response is issued with the abort function, which is used for error

handling. The following example returns status code 404 if the id dynamic argument

given in the URL does not represent a valid user:

**from flask import** abort

@app.route('/user/<id>')

**def** get\_user(id):

user = load\_user(id)

**if not** user:

abort(404)

**return** '<h1>Hello, %s</h1>' % user.name

Note that abort does not return control back to the function that calls it but gives control

back to the web server by raising an exception.

Hashing Passwords with Werkzeug

Werkzeug’s *security* module conveniently implements secure password hashing. This

functionality is exposed with just two functions, used in the registration and verification

phases, respectively:

• generate\_password\_hash(password, method=*pbkdf2:sha1*, salt\_length=8):

This function takes a plain-text password and returns the password hash as a string

that can be stored in the user database. The default values for method and

salt\_length are sufficient for most use cases.

• check\_password\_hash(hash, password): This function takes a password hash retrieved

from the database and the password entered by the user. A return value of

True indicates that the password is correct