Quickstart

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

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?

web server gateway interface

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:

use --app

As a shortcut, if the file is named `app.py` or `wsgi.py`, you don't have
Command Line Interface for more details.

en stable

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:

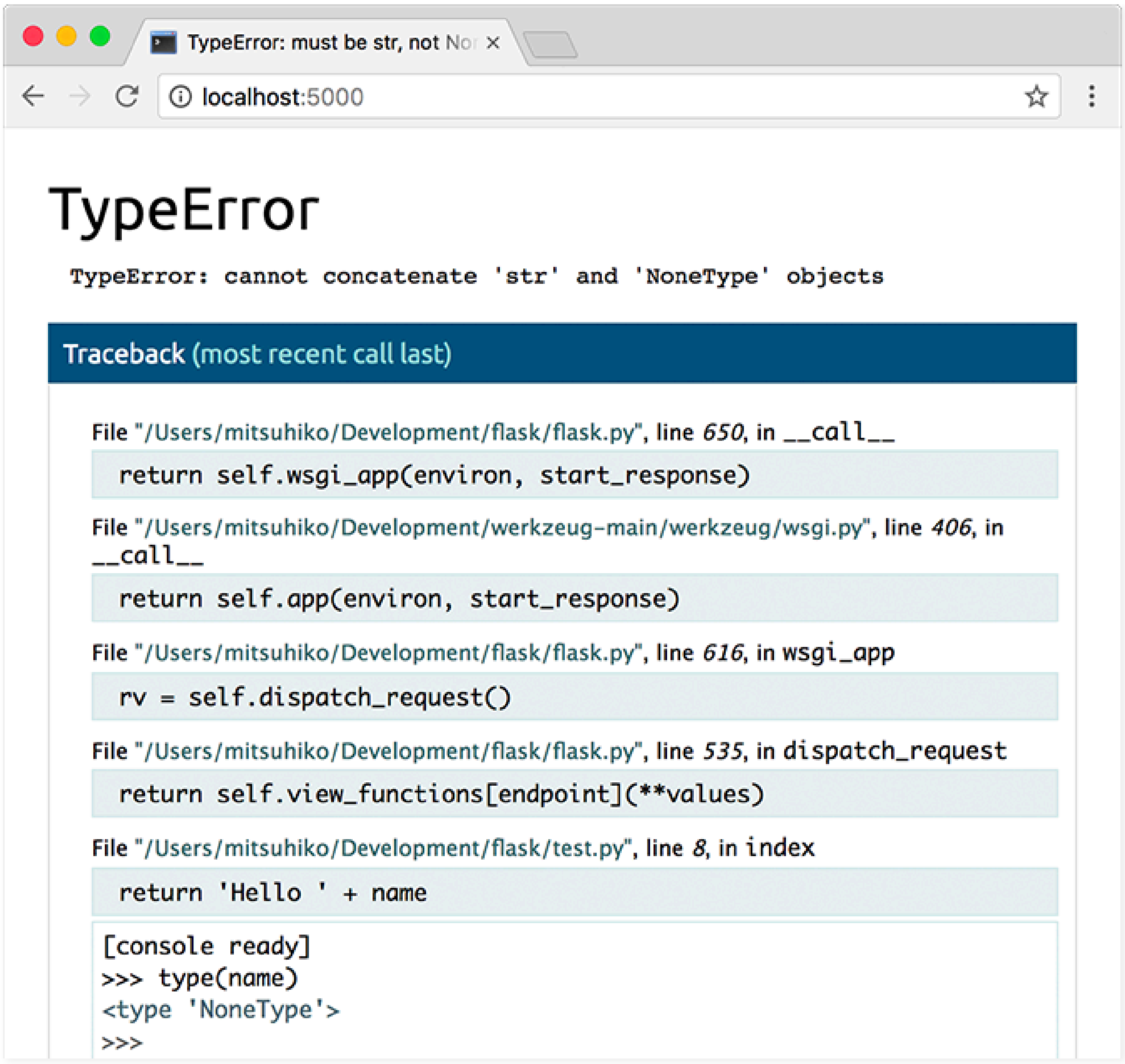
```
$ flask run --host=0.0.0.0
```

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

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.

en stable


TypeError: must be str, not No ×

localhost:5000

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
```

en stable

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.

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.

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):
    # 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

en stable

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 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', person=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 person %}
  <h1>Hello {{ person }}!</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 `person` 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'
```

▶ *Changelog*

[1] Unsure what that `g` object is? It's something in which you can sto en stable needs. See the documentation for `flask.g` and Using SQLite 3 with Flask.

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:

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:

en stable

```
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:

```python
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:

```python
@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)
```

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:

```python
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 f
`enctype="multipart/form-data"` attribute on your HTML form, otherwise the browser will not transmit your files at all.

en stable

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.

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.

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.

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
app.secret_key = b'_5#y2L"F4Q8z\n\xec]/'

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

```python
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'))
```

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.

Message Flashing

Good applications and user interfaces are all about feedback. If the us
back 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.

Logging

▶ *Changelog*

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.

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.

en stable

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.

Deploying to a Web Server

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