[Introduction 3](#_Toc156728677)

[Installation 3](#_Toc156728678)

[QuickStart 3](#_Toc156728679)

[A Minimal Application 3](#_Toc156728680)

[Debug Mode 4](#_Toc156728681)

[HTML Escaping 4](#_Toc156728682)

[Routing 6](#_Toc156728683)

[Variable Rules 6](#_Toc156728684)

[Converter types 6](#_Toc156728685)

[Unique URLS / Redirection Behavior 7](#_Toc156728686)

[URL Building 7](#_Toc156728687)

[HTTP Methods 8](#_Toc156728688)

[Static Files 9](#_Toc156728689)

[Rendering Templates 10](#_Toc156728690)

[File Uploads 11](#_Toc156728691)

[Cookies 12](#_Toc156728692)

[Redirects and Errors 13](#_Toc156728693)

[About Responses 14](#_Toc156728694)

[APIs with JSON 15](#_Toc156728695)

[Sessions 16](#_Toc156728696)

[Logging 18](#_Toc156728697)

[Using Flask Extensions 19](#_Toc156728698)

# Introduction

Flask is an open-source lightweight web framework written in Python. It does not require particular tools or libraries, it has no database abstraction layer, form validation or any other components.

However, it supports extensions that can add application features as if they were written in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling and more.

Applications that use the Flask framework include Pinterest and LinkedIn.

# Installation

To install Flask, you need to have Python and a Python package manager (e.g. pip) installed already installed. To verify whether Python is installed open a terminal and run python –version; to verify whether pip is installed, run pip –version (or python -m pip –version).

Once you’ve verified that these 2 are installed, run the command pip install flask, to install flask. To check if flask has been installed successfully, open a terminal and run flask –version (or python -m flask –version). The latest version of Flask at the time of writing this is 3.0.1.

Voila! You have successfully installed Flask.

# QuickStart

Create a new folder on your computer named Web with Flask. In this folder create another folder called getting\_started. In this getting\_started folder create a file called app.py.

## A Minimal Application

Copy and paste the following code in the app.py file we created earlier.

****

What does the code do?

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

To run the application open a terminal in the directory where the app.py file is located (Web with Flask), use the flask run command or python -m flask run.

When you run the server, you will notice that it is only accessible from your own computer, not from any other in the network. To make the server publicly available add the –host=0.0.0.0 argument to the command i.e. run python -m flask run –host=0.0.0.0 instead of python -m flask run.

### Debug Mode

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.

**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.

## HTML Escaping

When returning HTML, any user-provided values rendered in the output must be escaped to protect from injection attacks. HTML templates rendered with Jinja will do this automatically.

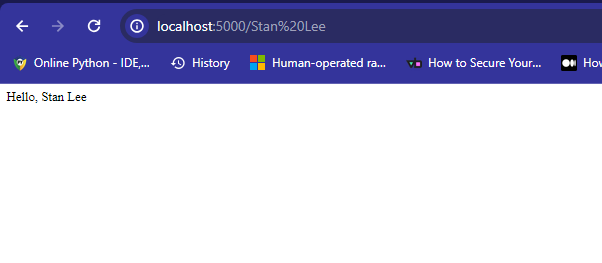
**escape()**, shown here, can be used manually. It is omitted in most examples for brevity, but you should always be aware of how you’re using untrusted data.

In the app.py file created earlier, copy and paste the following code



The <name> in the route captures a value from the URL and passes it to the view function.

You can run the server if it wasn’t already running and open a browser to localhost:5000 and try inputting a path /Stan Lee and observe the output.



Tip: Try removing the escape call from the return statement, (return f”Hello {name}”) and inputting some JavaScript code like <script>alert(“This could be dangerous”)</script> instead of Stan Lee in the URL section of the browser.

Put the escape call back in the return statement and reload the web page. What’s the difference?

## Routing

Use the route() decorator to bind a function to a URL like we’ve seen in previous examples.

You can also 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>.



## 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.



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.

Why would you want to build URLs using the URL reversing function [**url\_for()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.url_for) properly handles that for you.

For example, here we use the [**test\_request\_context()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.test_request_context) method to try out [**url\_for()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.url_for). [**test\_request\_context()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.test_request_context) tells Flask to behave as though it’s handling a request even while we use a Python shell. Copy the following code and paste at the bottom of the app.py file created previously and run the file with Python (i.e. python app.py)



## 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.



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()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.get), [**post()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.post), etc. for each common HTTP method.



## 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.

Go to our Web with Flask folder and inside the getting\_started folder, create another folder called static. This is where we will store images, CSS and JavaScript files. Our folder structure should look like this:



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.

To render a template you can use the [**render\_template()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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:



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 (i.e. a Python file):



**Case 2**: a package:



For templates you can use the full power of Jinja2 templates. Head over to the official [Jinja2 Template Documentation](https://jinja.palletsprojects.com/templates/) for more information.

Here is an example template:



Inside templates you also have access to the [**config**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.config), [**request**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.request), [**session**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.session) and [**g**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.g) [[1]](https://flask.palletsprojects.com/en/3.0.x/quickstart/#id3) objects as well as the [**url\_for()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.url_for) and [**get\_flashed\_messages()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.get_flashed_messages) functions.

Templates are especially useful if inheritance is use. Basically, template inheritance makes it possible to keep certain elements on each page (like header, navigation and footer).

## 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:



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**](https://werkzeug.palletsprojects.com/en/3.0.x/datastructures/#werkzeug.datastructures.FileStorage.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()**](https://werkzeug.palletsprojects.com/en/3.0.x/utils/#werkzeug.utils.secure_filename) function that Werkzeug provides for you:



## 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:



Storing cookies:



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

For this also see [About Responses](https://flask.palletsprojects.com/en/3.0.x/quickstart/#about-responses).

## 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:



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:



Note the 404 after the [**render\_template()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.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.

## 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:



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:



## 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.



This is a shortcut to passing the data to the [**jsonify()**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.json.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**](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:



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



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.

## Logging

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:



The attached [**logger**](https://flask.palletsprojects.com/en/3.0.x/api/#flask.Flask.logger) is a standard logging [**Logger**](https://docs.python.org/3/library/logging.html#logging.Logger), so head over to the official [**logging**](https://docs.python.org/3/library/logging.html#module-logging) docs for more information.

See [Handling Application Errors](https://flask.palletsprojects.com/en/3.0.x/errorhandling/).

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