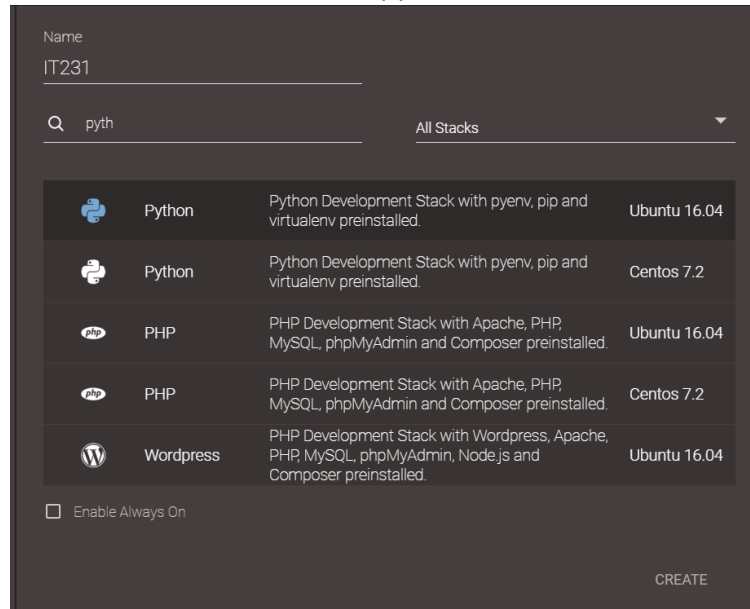


# Python/Flask Tutorial

## 1. Development Platform and Setup

- a. Create an account on Codeanywhere.com
- b. Then create a container with a python Ubuntu 16.04 instance



- c. Ensure that you have python and pip
  - i. Check your python version
    1. (command: python --version)
  - ii. Check your pip version
    1. (command: pip --version)
- d. Then in your container's terminal create a flask-tutorial directory
  - i. (command: mkdir flask-tutorial)
  - ii. Then go into that directory (command: cd flask-tutorial)
- e. Install flask
  - i. (command: pip install Flask)

## 2. Project Layout

```
/workspace/flask-tutorial
├── flaskr/
│   ├── __init__.py
│   ├── db.py
│   ├── schema.sql
│   ├── auth.py
│   ├── blog.py
│   └── templates/
│       ├── base.html
│       └── auth/
│           ├── login.html
│           └── register.html
```



### 3. Git and Github repositories

- a. If you don't already have one, create a github account on [github.com](https://github.com)
- b. Create a new repository named "Python-Flask-Tutorial"
  - i. Make it public with no README
- c. Locally in codeanywhere, in your flask-tutorial directory, type in the following commands:
  - i. `git init`
  - ii. `git commit -m "first commit"`
  - iii. `git remote add origin https://github.com/<your github username>/Python-Flask-Tutorial.git`

### 4. Flask Application Factory

- a. Create Flaskr directory
  - i. (command: `mkdir flaskr`)
  - ii. Then go into that directory (command: `cd flaskr`)
- b. Create file: `__init__.py`
  - i. This file will contain the application factory and will tell Python that the flaskr directory should be treated as a package

flaskr/\_\_init\_\_.py

```
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!'
```

- c. Run the factory with these commands in your container terminal
  - i. export LC\_ALL=C.UTF-8
  - ii. export LANG=C.UTF-8
  - iii. export FLASK\_APP=flaskr
  - iv. export FLASK\_ENV=development
  - v. flask run --host=0.0.0.0 --port=3000
- d. Then go to your container info, get the https link listening on port 3000 then add “/hello”

← → ↻ 🔒 <https://it231-cabbagebabicz577657.codeanyapp.com/hello>

Hello, World!

- e. In your container terminal use (command: Ctrl C) to kill the application
5. Application sqlite3 database
- a. Create db.py in your flaskr directory

- i. This will allow you application to create a connection to the database

flaskr/db.py

```
import sqlite3

import click
from flask import current_app, g
from flask.cli import with_appcontext

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

    return g.db

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

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

- b. Create schema of table and columns in your flaskr folder with filename: schema.sql

flaskr/schema.sql

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

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

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

- i. Then add these functions in you db.py file to have your sqlite database run these SQL commands

flaskr/db.py

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

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

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

c. Register with the Application

- i. The close\_db and init\_db\_command functions must be registered with an application instance. Thus, we need to write a function that takes an application instance and does the proper registration so the database will be used by the registration.
- ii. Insert this code in your db.py

flaskr/db.py

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

- iii. Next, we need to import and call this function from the factory. Add this to your \_\_init\_\_.py

flaskr/\_\_init\_\_.py

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

    from . import db
    db.init_app(app)

    return app
```

d. Initialize the database file

- i. In your container terminal run the following command. This command will create a flaskr.sqlite file in your instance folder in your project.
  1. (command: flask init-db)

6. Security program controller (auth.py)

- a. Create a blueprint
  - i. A blueprint is a way to organize a group of related views and other code.
  - ii. Flaskr will have 2 blueprints: one for authentication functions and one blog posts functions
  - iii. Since the blog needs to know about authentication, we'll implement authentication first
  - iv. In your flaskr folder, create the file auth.py

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')
```

- v. Then, import and register the blueprint in \_\_init\_\_.py

flaskr/\_\_init\_\_.py

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

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

    return app
```

- b. First View: Register
  - i. Upon visiting the auth/register URL, the register view will return HTML with a form for the user to fill out in order to “register” for our application.
  - ii. Add this code to your auth.py file
  - iii. **\*\*Also, in order to support https, we need to change return redirect(url\_for('auth.login')) to return redirect(url\_for('auth.login', \_scheme='https', \_external=True))\*\***

flaskr/auth.py

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

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'
        elif db.execute(
            'SELECT id FROM user WHERE username = ?', (username,)
        ).fetchone() is not None:
            error = 'User {} is already registered.'.format(username)

        if error is None:
            db.execute(
                'INSERT INTO user (username, password) VALUES (?, ?)',
                (username, generate_password_hash(password))
            )
            db.commit()
            return redirect(url_for('auth.login'))

        flash(error)

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

c. Login

- i. If a user is already registered, we then want them to login to an existing account
- ii. Add the following code to your auth.py file
- iii. **\*\*Also, in order to support https, we need to change return redirect(url\_for('index')) to return redirect(url\_for('index', \_scheme='https', \_external=True))\*\***

flaskr/auth.py

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

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

        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))

        flash(error)

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

- iv. The user's id is stored in a session, and will be available in future requests. At the beginning of each request, the logged in user's information will be loaded and made available to other views. Add the following to your auth.py file to accomplish this.

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

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

d. Logout

- i. In order to have the user logout, we need to remove the user id from the session. Add the following code to your auth.py file to do this



flaskr/auth.py

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

- e. Require authentication in other views
  - i. Creating, editing, and deleting blog posts will require a user to be logged in. A decorator can be used to check this for each view it's applied to. Add this code to your auth.py to create the decorator
  - ii. **\*\*Also, in order to support https, we need to change return redirect(url\_for('auth.login')) to return redirect(url\_for('auth.login', \_scheme='https', \_external=True))\*\***

flaskr/auth.py

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

        return view(**kwargs)

    return wrapped_view
```

- 7. Blog interface controller (blog.py)
  - a. In your flaskr directory create a blog.py file
    - i. This will define a blueprint for your blog posts. Add the code below to your blog.py file

flaskr/blog.py

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

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

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

- b. Import and register the blueprint in your \_\_init\_\_.py file. Add the following code to your \_\_init\_\_.py file

flaskr/\_\_init\_\_.py

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

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

    return app
```

c. Index

- i. The index will show all of the posts from most recent to oldest. Insert the following code in your blog.py file to create index route

flaskr/blog.py

```
@bp.route('/')
def index():
    db = get_db()
    posts = db.execute(
        'SELECT p.id, title, body, created, author_id, username'
        ' FROM post p JOIN user u ON p.author_id = u.id'
        ' ORDER BY created DESC'
    ).fetchall()
    return render_template('blog/index.html', posts=posts)
```

d. Create

- i. This view will allow a user to create a blog post. Insert the following code in your blog.py file
- ii. **\*\*Also, in order to support https, we need to change return redirect(url\_for('blog.index')) to return redirect(url\_for('blog.index', \_scheme='https', \_external=True))\*\***

```

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

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

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

    return render_template('blog/create.html')

```

e. Update

- i. Both the update and delete view will need to fetch a post by id and check if the author is the user. To avoid duplicate code, we create a function to get the post so we are able to call it from either view. Add the following code to your blog.py file.

flaskr/blog.py

```

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

    if post is None:
        abort(404, "Post id {0} doesn't exist.".format(id))

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

    return post

```

- ii. This is the update view. It will allow the user to update an existing blog post. Add the following code to your blog.py file
- iii. **\*\*Also, in order to support https, we need to change return redirect(url\_for('blog.index')) to return redirect(url\_for('blog.index', \_scheme='https', \_external=True))\*\***

flaskr/blog.py

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

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

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

        if error is not None:
            flash(error)
        else:
            db = get_db()
            db.execute(
                'UPDATE post SET title = ?, body = ?'
                ' WHERE id = ?',
                (title, body, id)
            )
            db.commit()
            return redirect(url_for('blog.index'))

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

- f. Delete
  - i. The delete view does not have template (which we will talk about next). It only has a delete button in update.html and posts to the /<id>/delete URL. Since there is no template, it will only handle the POST method and then redirect the user to the index view. Add the following code to your blog.py file to be able to delete blog posts.

flaskr/blog.py

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

## 8. View and templates

- a. First, create a templates directory under your flaskr directory. This will hold all of your html
- b. The base layout
  - i. Each page in the application will have the same basic layout around a different body. Instead of writing the entire HTML structure in each template, each template will extend a base template and override specific sections. Create a file named base.html and add the following code to your templates directory.

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

- c. Register
  - i. Create a directory under your templates directory called auth
  - ii. Create a register.html under the auth folder and add the following code

flaskr/templates/auth/register.html

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

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

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

d. Log In

- i. In your auth folder create a file called login.html with code below. You will notice that the register template is almost exactly the same except for the title and submit button.

flaskr/templates/auth/login.html

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

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

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

- e. Create a subdirectory called blog under your templates directory. Here we will store all of our blog templates.
- f. Index

flaskr/templates/blog/index.html

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

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

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

g. Create

flaskr/templates/blog/create.html

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

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

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

h. Update

flaskr/templates/blog/update.html

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

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

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

## 9. Static Files

- The views will look very plain at the moment. So we will add some CSS
- Under the flaskr directory create a subdirectory called style.css and add the following code

flaskr/static/style.css

```
html { font-family: sans-serif; background: #eee; padding: 1rem; }
body { max-width: 960px; margin: 0 auto; background: white; }
h1 { font-family: serif; color: #377ba8; margin: 1rem 0; }
a { color: #377ba8; }
hr { border: none; border-top: 1px solid lightgray; }
nav { background: lightgray; display: flex; align-items: center; padding: 0 0.5rem; }
nav h1 { flex: auto; margin: 0; }
nav h1 a { text-decoration: none; padding: 0.25rem 0.5rem; }
nav ul { display: flex; list-style: none; margin: 0; padding: 0; }
nav ul li a, nav ul li span, header .action { display: block; padding: 0.5rem; }
.content { padding: 0 1rem 1rem; }
.content > header { border-bottom: 1px solid lightgray; display: flex; align-items: flex-end; }
.content > header h1 { flex: auto; margin: 1rem 0 0.25rem 0; }
.flash { margin: 1em 0; padding: 1em; background: #cae6f6; border: 1px solid #377ba8; }
.post > header { display: flex; align-items: flex-end; font-size: 0.85em; }
.post > header > div:first-of-type { flex: auto; }
.post > header h1 { font-size: 1.5em; margin-bottom: 0; }
.post .about { color: slategray; font-style: italic; }
.post .body { white-space: pre-line; }
.content:last-child { margin-bottom: 0; }
.content form { margin: 1em 0; display: flex; flex-direction: column; }
.content label { font-weight: bold; margin-bottom: 0.5em; }
.content input, .content textarea { margin-bottom: 1em; }
.content textarea { min-height: 12em; resize: vertical; }
input.danger { color: #cc2f2e; }
input[type=submit] { align-self: start; min-width: 10em; }
```

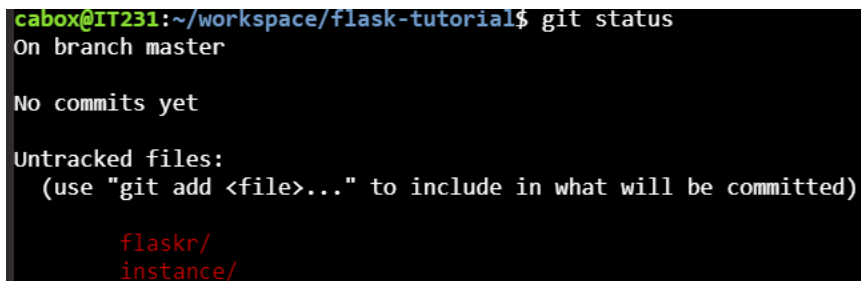


## 10. Running flaskr

- a. Go to the flaskr-tutorial directory in your terminal and run the following code to launch the application!
  - i. `export LC_ALL=C.UTF-8`
  - ii. `export LANG=C.UTF-8`
  - iii. `export FLASK_APP=flaskr`
  - iv. `export FLASK_ENV=development`
  - v. `flask run --host=0.0.0.0 --port=3000`
- b. To kill the application press Ctrl C in the terminal

## 11. Save all code in a repository

- a. Here are all the commands to save to your remote repository you created at the beginning
  - i. `git status`
    1. this will show which files need to be committed to master



```
cabox@IT231:~/workspace/flask-tutorial$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    flaskr/
    instance/
```

- ii. `git add flaskr/`
  1. (we will stage our flaskr directory for commit and will ignore instance/ since it's just a generated directory based on our editor)
- iii. `git status`
  1. (we type in this command again to see all the files we have staged for commit)

```

nothing added to commit but untracked files present (use "git add" to track)
cabox@IT231:~/workspace/flask-tutorial$ git add flaskr/
cabox@IT231:~/workspace/flask-tutorial$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   flaskr/__init__.py
        new file:   flaskr/__pycache__/__init__.cpython-37.pyc
        new file:   flaskr/__pycache__/auth.cpython-37.pyc
        new file:   flaskr/__pycache__/blog.cpython-37.pyc
        new file:   flaskr/__pycache__/db.cpython-37.pyc
        new file:   flaskr/auth.py
        new file:   flaskr/blog.py
        new file:   flaskr/db.py
        new file:   flaskr/schema.sql
        new file:   flaskr/static/style.css
        new file:   flaskr/templates/auth/login.html
        new file:   flaskr/templates/auth/register.html
        new file:   flaskr/templates/base.html
        new file:   flaskr/templates/blog/create.html
        new file:   flaskr/templates/blog/index.html
        new file:   flaskr/templates/blog/update.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        instance/

```

- iv. git commit -m "Flaskr tutorial completed"
  1. (all the file staged for commit will now be ready to be pushed to our master branch in our repository)
- v. git push origin master
  1. (We have now pushed all our files to our remote repository!)
- b. To access our remote repository elsewhere we will follow these commands
  - i. git clone https://github.com/<your github user name>/Python-Flask-Tutorial.git
    1. We now have all the files from our remote repository on our local machine!