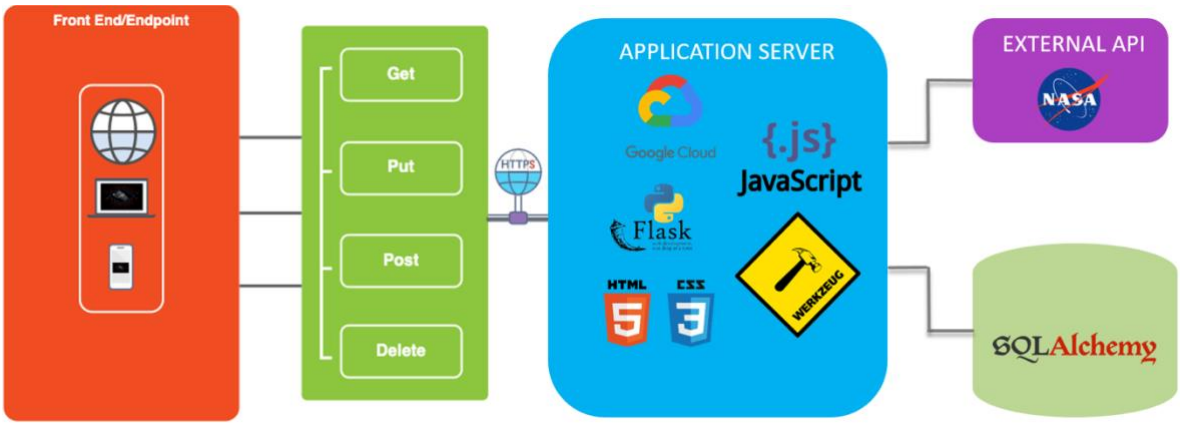
Architecture

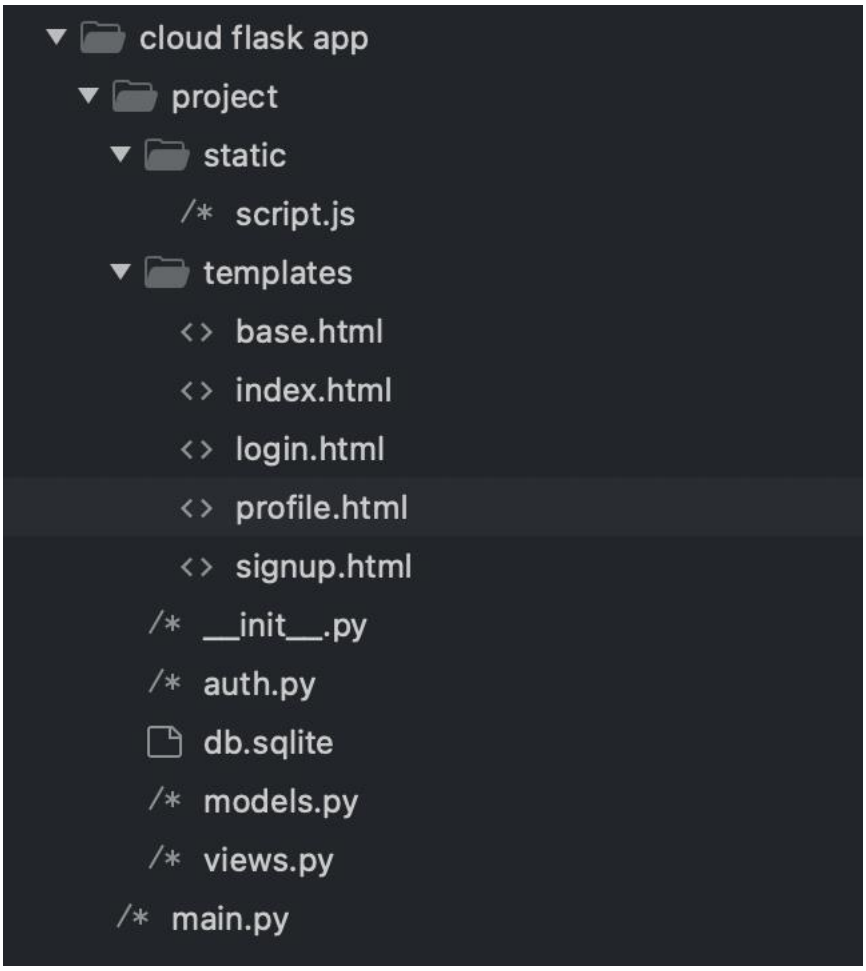


Prerequisites

To complete this project, you will need the following:

- Some familiarity with Python.
- Python installed on a local environment/ cloud environment.
- Knowledge of Basic Linux Navigation and File Management.

Here is a diagram to provide a sense of what the file structure of the project will look like once you have completed the tutorial:



Step 1 — Installing Packages

There are three main packages you need for your project:

- Flask
- Flask-Login: to handle the user sessions after authentication
- Flask-SQLAlchemy: to represent the user model and interface with the database

You will be using SQLite to avoid having to install any extra dependencies for the database.
First, start with creating the project directory:

```
mkdir cloud_flask_app
```

Next, navigate to the project directory:

```
cd cloud_flask_app
```

You will want to create a Python environment if you don't have one.
Depending on how Python was installed on your machine, your command will look like:

```
sudo apt-get install python3-venv
Python3 -m venv nasa
```

The -m flag is for module-name. This command will execute the module venv to create a new virtual environment named nasa. This will create a new directory containing bin, include, and lib subdirectories. And a pyvenv.cfg file.

Next, run the following command:

```
source nasa/bin/activate
```

This command will activate the virtual environment.
Run the following command from your virtual environment to install the needed packages:

```
nano requirements.txt
```

copy paste the contents of requirement.txt files and save.

```
pip install -r requirements.txt
```

Now that you have installed the packages, you are ready to create the app.

Step 2 — Creating the Main App File

Let's start by creating a project directory:

```
mkdir project
```

The first file will be the __init__.py file for the project:
This file will have the function to create the app, which will initialize the database and register the blueprints. Now, this will not do much, but it will be needed for the rest of the app.
You will need to initialize SQLAlchemy, set some configuration values, and register the blueprints here:

nano project/__init__.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager

# init SQLAlchemy so we can use it later in our models
db = SQLAlchemy()

def create_app():
    app = Flask(__name__)

    app.config['SECRET_KEY'] = 'secret-key-goes-here'
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///db.sqlite'

    db.init_app(app)

    login_manager = LoginManager()
    login_manager.login_view = 'auth.login'
    login_manager.init_app(app)

    from .models import User

    @login_manager.user_loader
    def load_user(user_id):
        # since the user_id is just the primary key of our user table, use it in the query for the user
        return User.query.get(int(user_id))

    # blueprint for auth routes in our app
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint)

    # blueprint for non-auth parts of app
    from .views import views as main_blueprint
    app.register_blueprint(main_blueprint)

    return app

```

Step 3 — Adding Routes

For the main_blueprint, the main blueprint will be used to run the application.
First, create main.py:

nano main.py

```python
from project import create_app

app= create_app()

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port = 5000)

```

Next, create views.py:
For the routes, you will use two blueprints.
For the views_blueprint, you will have a home page (/) and a profile page (/profile).

nano project/views.py

```
from flask import Blueprint, render_template
from flask_login import login_required,current_user

views = Blueprint('views', __name__)

@views.route('/')
def index():
    return render_template('index.html')

@views.route('/profile')
@login_required
def profile():
    name= current_user.name
    return render_template('profile.html',name=current_user.name)

```

For the auth_blueprint, you will have routes to retrieve both the login page (/login) and the sign-up page (/signup). Finally, you will have a logout route (/logout) to log out an active user.
Next, create auth.py:

nano project/auth.py

Then add your auth_blueprint:

```
from flask import Blueprint, render_template, redirect, url_for, request,flash
from werkzeug.security import generate_password_hash, check_password_hash
from flask_login import login_user, logout_user,login_required
from .models import User
from . import db

auth = Blueprint('auth', __name__)

@auth.route('/login')
def login():
    return render_template('login.html')

@auth.route('/login', methods=['POST'])
def login_post():
    # login code goes here
    email = request.form.get('email')
    password = request.form.get('password')
    remember = True if request.form.get('remember') else False

    user = User.query.filter_by(email=email).first()

    # check if the user actually exists
    # take the user-supplied password, hash it, and compare it to the hashed password in the database
    if not user or not check_password_hash(user.password, password):
        flash('Please check your login details and try again.')
        return redirect(url_for('auth.login')) # if the user doesn't exist or password is wrong, reload the page

    login_user(user,remember=remember)
    # if the above check passes, then we know the user has the right credentials
    return redirect(url_for('views.profile'))

```

```python
@auth.route('/signup')
def signup():
    return render_template('signup.html')

@auth.route('/signup', methods=['POST'])
def signup_post():
    # code to validate and add user to database goes here
    email = request.form.get('email')
    name = request.form.get('name')
    password = request.form.get('password')

    user = User.query.filter_by(email=email).first() # if this returns a user, then the email already exists in database

    if user: # if a user is found, we want to redirect back to signup page so user can try again
        flash('Email address already exists')
        return redirect(url_for('auth.signup'))

    # create a new user with the form data. Hash the password so the plaintext version isn't saved.
    new_user = User(email=email, name=name, password=generate_password_hash(password, method='sha256'))

    # add the new user to the database
    db.session.add(new_user)
    db.session.commit()

    return redirect(url_for('auth.login'))

@auth.route('/logout')
@login_required
def logout():
    logout_user()
    return render_template('index.html')
```

Step 4 Creating Templates —

Next, create the templates that are used in the app. This is the first step before you can implement the actual login functionality.

The app will use four templates:

- index.html
- profile.html
- login.html
- signup.html

You will also have a base template that will have code common to each of the pages. In this case, the base template will have navigation links and the general layout of the page.

First, create a templates directory under the project directory:

```
mkdir -p project/templates
```

Then create base.html:

```
nano project/templates/base.html
```

Next, add the following code to the base.html file:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Cloud Flask App</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/bulma/0.9.3/css/bulma-rtl.css" />

<style>
.navbar-end {
    font-family: "Cookie", cursive;
}
</style>


</head>

<body>
    <section class="hero is-primary is-fullheight">

        <div class="hero-head">
            <nav class="navbar">
                <div class="container">
                    <div id="navbarMenuHeroA" class="navbar-menu">
                        <div class="navbar-end">
                            <a href="{{ url_for('views.index') }}" class="navbar-item">
                                Home
                            </a>
                            {% if current_user.is_authenticated %}
                            <a href="{{ url_for('views.profile') }}" class="navbar-item">

                                Profile
                            </a>
                            {% endif %}
                            {% if not current_user.is_authenticated %}
                            <a href="{{ url_for('auth.login') }}" class="navbar-item">
                                Login
                            </a>
                            <a href="{{ url_for('auth.signup') }}" class="navbar-item">
                                Sign Up
                            </a>
                            {% endif %}
                            {% if current_user.is_authenticated %}
                            <a href="{{ url_for('auth.logout') }}" class="navbar-item">
                                Logout
                            </a>
                            {% endif %}
                        </div>
                    </div>
                </div>
            </nav>
        </div>


        <div class="hero-body">
            <div class="container has-text-centered">
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </section>
</body>

</html>

```

This code will create a series of menu links to each page of the application. It also establishes a block for content that can be overwritten by child templates.

Next, create templates/index.html:

nano project/templates/index.html

Add the following code to the newly created file to add content to the page:

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

{% block content %}
<h1 class="title">
  Flask Login Example
</h1>
<h2 class="subtitle">
  Easy authentication and authorization in Flask.
</h2>
{% endblock %}
```

This code will create a basic index page with a title and subtitle.

Next, create templates/login.html:

nano project/templates/login.html

This code generates a login page with fields for Email and Password. There is also a checkbox to “remember” a logged in session.

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

{% block content %}
<div class="column is-4 is-offset-4">
    <h3 class="title">Login</h3>
    <div class="box">
        {% with messages = get_flashed_messages() %}
        {% if messages %}
            <div class="notification is-danger">
                {{ messages[0] }}
            </div>
        {% endif %}
        {% endwith %}
        <form method="POST" action="/login">
            <div class="field">
                <div class="control">
                    <input class="input is-large" type="email" name="email" placeholder="Your Email" autofocus="">
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="password" name="password" placeholder="Your Password">
                </div>
            </div>
            <div class="field">
                <label class="checkbox">
                    <input type="checkbox" name="remember">
                    Remember me
                </label>
            </div>
            <button class="button is-block is-info is-large is-fullwidth">Login</button>
        </form>
    </div>
</div>
{% endblock %}
```

Next, create templates/signup.html:

nano project/templates/signup.html

Add the following code to create a sign-up page with fields for email, name, and password:

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

{% block content %}
<div class="column is-4 is-offset-4">
    <h3 class="title">Sign Up</h3>
    <div class="box">
        {% with messages = get_flashed_messages() %}
        {% if messages %}
            <div class="notification is-danger">
            {{ messages[0] }}. Go to <a href="{{ url_for('auth.login') }}">login page</a>.
            </div>
        {% endif %}
        {% endwith %}
        <form method="POST" action="/signup">
            <div class="field">
                <div class="control">
                    <input class="input is-large" type="email" name="email" placeholder="Email" autofocus="">
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="text" name="name" placeholder="Name" autofocus="">
                </div>
            </div>

            <div class="field">
                <div class="control">
                    <input class="input is-large" type="password" name="password" placeholder="Password">
                </div>
            </div>

            <button class="button is-block is-info is-large is-fullwidth">Sign Up</button>
        </form>
    </div>
</div>
{% endblock %}
```

Next, create templates/profile.html:

nano project/templates/profile.html

This is where we will display the data fetched for the NASA APOD API

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

{% block content %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>NASA's Astronomy Picture of the Day</title>
  </head>
  <body>
    <body>
  <main class="container">
    <h1>NASA's Astronomy Picture of the Day</h1>
    <h2 id="title"></h2>
    <p id="date"></p>
    <section class="picture-explanation-container">
      <img src="" id="picture" alt="astronomy image by NASA" width="40%" height="40%" />
      <p id="explanation"></p>
    </section>
  </main>

  <div class="button-container">
 <button class="random-picture-button" id="random-day-generator">Random</button>
 </div>

  <script src="{{url_for('static', filename='script.js')}}"></script>
</body>
  </body>
</html>

{% endblock %}
```

Step 5 — Creating User Models

The user model represents what it means for the app to have a user. This tutorial will require fields for an email address, password, and name. In future applications, you may decide you want much more information to be stored per user. You can add things like birthdays, profile pictures, locations, or any user preferences.

Models created in Flask-SQLAlchemy are represented by classes that then translate to tables in a database. The attributes of those classes then turn into columns for those tables.

Create the User model:

```
nano project/models.py
```

Define the User model:

```
from . import db
from flask_login import UserMixin

class User(UserMixin,db.Model):
    id = db.Column(db.Integer, primary_key=True) # primary keys are required by SQLAlchemy
    email = db.Column(db.String(100), unique=True)
    password = db.Column(db.String(100))
    name = db.Column(db.String(1000))
```

This code defines a User with columns for an id, email, password, and name.

Now that you've created a User model, you can move on to configuring your database.

Step 6 — Configuring the Database

You will be using an SQLite database. You could create an SQLite database on your own, but let's have Flask-SQLAlchemy do it for you. You already have the path of the database specified in the __init__.py file, so you will need to tell Flask-SQLAlchemy to create the database in the Python REPL.

Ensure that you are still in the virtual environment and in the flask_cloud_app directory.

If you stop your app and open a Python REPL, you can create the database using the create_all method on the db object:

```
python
from project import db, create_app, models
db.create_all(app=create_app())
exit()
```

You will now see a db.sqlite file in your project directory. This database will have the user table in it.

Step 7 — Setting Up the JavaScript for fetching data from Nasa APOD API
One of the most popular websites at NASA is the Astronomy Picture of the Day. In fact, this website is one of the most popular websites across all federal agencies.

HTTP Request
GET https://api.nasa.gov/planetary/apod
concept_tags are now disabled in this service. Also, an optional return parameter *copyright* is returned if the image is not public domain.

Query Parameters

Parameter	Type	Default	Description
date	YYYY-MM-DD	*today*	The date of the APOD image to retrieve
start_date	YYYY-MM-DD	none	The start of a date range, when requesting date for a range of dates. Cannot be used with date.
end_date	YYYY-MM-DD	*today*	The end of the date range, when used with start_date.
count	int	none	If this is specified then count randomly chosen images will be returned. Cannot be used with date or start_date and end_date.
thumbs	bool	False	Return the URL of video thumbnail. If an APOD is not a video, this parameter is ignored.
api_key	string	DEMO_KEY	api.nasa.gov key for expanded usage

Example query
https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY

```
const url = 'https://api.nasa.gov/planetary/apod?api_key='
const api_key = 'DEMO_KEY'

const fetchNASAData = async () => {
  try {
    const response = await fetch(`${url}${api_key}`)
    const data = await response.json()
    console.log('NASA APOD data', data)
    displayData(data)
  } catch (error) {
    console.log(error)
  }
}

const displayData = data => {
  document.getElementById('title').textContent = data.title
  document.getElementById('date').textContent = data.date
  document.getElementById('picture').src = data.hdurl
  document.getElementById('explanation').textContent = data.explanation
}

fetchNASAData()


document.querySelector("#random-day-generator").addEventListener("click", () => {
 const fetchData = async () => {
 //utility function to generate a random date after 2010 (before 2010 causes bugs)
 function randomDate(start, end) {
 return new Date(start.getTime() + Math.random() * (end.getTime() - start.getTime()))
 }
 let randomRolledDate = randomDate(new Date(2010, 0, 1), new Date())
 randomRolledDate = "&date=" + randomRolledDate.toISOString().slice(0, 10) + "&"

 try {
 //new API response
  const response = await fetch(url + api_key + randomRolledDate)
  const data = await response.json()
  console.log('NASA data', data)
   displayData(data)
  } catch (error) {
    console.log(error)
  }
}
 fetchData()
})
```

Step 8 — Run the application

The FLASK_DEBUG environment variable is enabled by setting it to 1. This will enable a debugger that will display application errors in the browser.

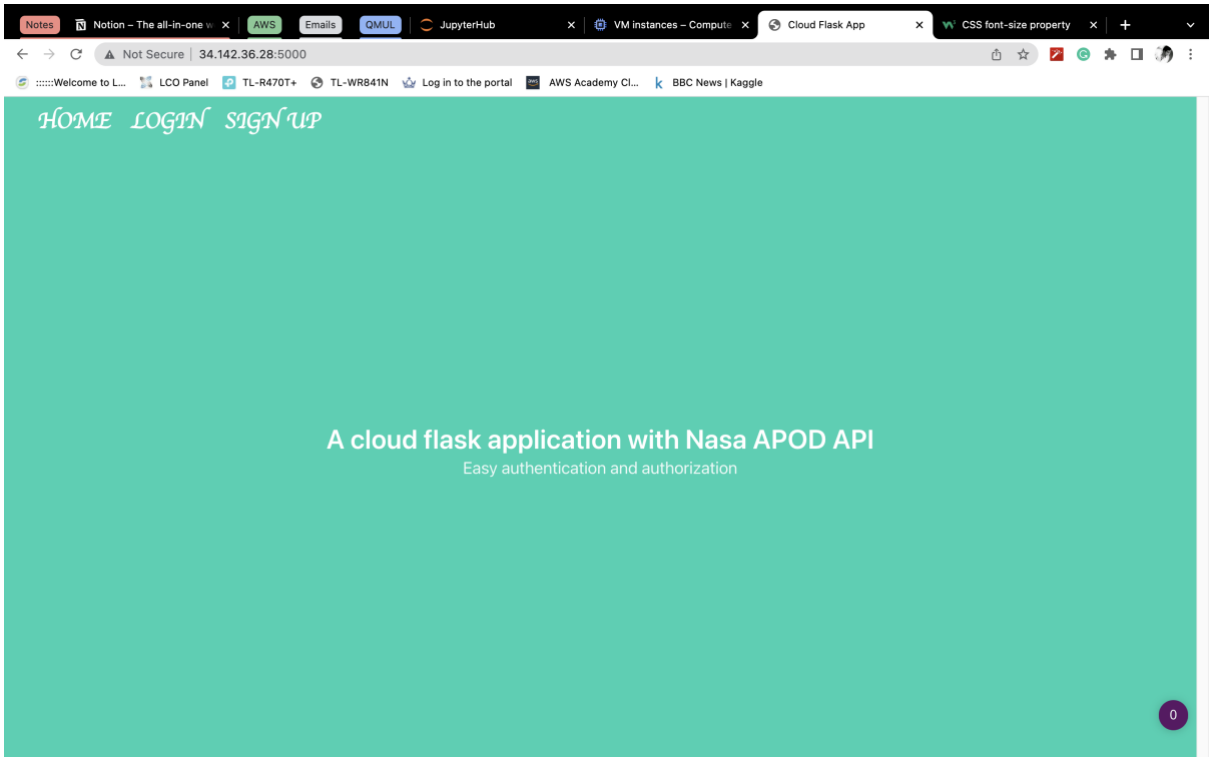
Ensure that you are in the flask_cloud_app directory and then run the project:

```
python3 main.py
```

Now, in a web browser, you can navigate to the five possible URLs and see the text returned that was defined in auth.py and views.py.

For example, visiting localhost:5000/ displays: Home:

Cloud Flask App using Nasa APOD API



Conclusion

We built a login system for an app using Flask-Login and Flask-SQLAlchemy in this app. By initially constructing a user model and saving the user's information, we have demonstrated how to authenticate a user. Then we had to check that the user's password was correct by hashing it and comparing it to the one saved in the database. Finally, we introduced authorisation to the app by using the @login required decorator on a profile page to restrict access to only logged-in users.

For simple apps, the code you wrote in this article will suffice, but if you want more functionality right away, you might consider using the Flask-User or Flask-Security libraries, which are both built on Flask. Finally, we have have demonstrated the use of Nasa APOD API to display picture of the day and randomly generated picture of data by passing a random date to the API.