**Python: Authenticating and Authorizing Users in Flask Applications**

[[Halim Choi](https://medium.com/@choihalim?source=post_page-----2db06fd2fcb5--------------------------------)](https://medium.com/@choihalim?source=post_page-----2db06fd2fcb5--------------------------------)

[Halim Choi](https://medium.com/@choihalim?source=post_page-----2db06fd2fcb5--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F4d3fa1a86317&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40choihalim%2Fpython-authenticating-and-authorizing-users-in-flask-applications-2db06fd2fcb5&user=Halim+Choi&userId=4d3fa1a86317&source=post_page-4d3fa1a86317----2db06fd2fcb5---------------------post_header-----------)

5 min read

·

Jun 29

2



Photo by [Jozsef Hocza](https://unsplash.com/@hocza?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

**What is Authentication/Authorization?**

Authentication and authorization are essential aspects of building secure web applications. Nearly all modern web applications rely on these features in order to authenticate or authorize users performing specific tasks. Registering an account in an application to see posts created by other users, logging in to view users’ interaction with your posts, or a user updating their profile information are some examples of when and why authentication and authorization are necessary.

Authentication is the process of verifying users attempting to access an application’s resources. It ensures that users actually own the account they are trying to log into before they are granted access. There are multiple types of authentication methods such as:

* password-based authentication
* multi-factor authentication (2FA - SMS/email verification for enhanced security)
* token-based authentication (server generates unique tokens for users)

We will be using password-based authentication in our example below.

Authorization is the process of granting or denying access to certain resources or functionalities within an application. This ensures that users have appropriate roles or permissions to perform specific actions. For example, a user in an application can be an admin who has permissions to delete other users/grant other users admin capabilities or they can be a user who can only create blog posts and comment on other users’ blog posts.

**Code**

To begin, let’s set up our ‘config.py’ in order to instantiate the Flask application and set its attributes. In order to set your secret key specific to your Flask application, run the following command in your command prompt or terminal:

$ python -c 'import os; print(os.urandom(16))'

We will use this randomly generated key to set the ‘SECRET\_KEY’ of your Flask app.

# config.py  
  
from flask import Flask  
from flask\_cors import CORS  
from flask\_migrate import Migrate  
from flask\_sqlalchemy import SQLAlchemy  
from sqlalchemy import MetaData  
  
# instantiate ap and set attributes  
app = Flask(\_\_name\_\_)  
app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///app.db'  
app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False  
app.json.compact = False  
  
# define metadata, set up Flask-Migrate for migrations, instantiate db  
metadata = MetaData(naming\_convention={  
 "fk": "fk\_%(table\_name)s\_%(column\_0\_name)s\_%(referred\_table\_name)s",  
})  
  
db = SQLAlchemy(metadata=metadata)  
migrate = Migrate(app, db)  
db.init\_app(app)  
  
# enables Cross-Origin Resource Sharing, allowing requests front-end  
CORS(app)  
  
# generate a secret key in command prompt using this command:  
# `python -c 'import os; print(os.urandom(16))'`  
app.config['SECRET\_KEY'] = ^YOUR\_SECRET\_KEY\_HERE^

Next, we must set up our ‘models.py’ which will contain a User class to store user data using SQLAlchemy.

# models.py  
from config import db  
  
class User(db.Model):  
 \_\_tablename\_\_ = "users"  
  
 id = db.Column(db.Integer, primary\_key=True)  
 username = db.Column(db.String, unique=True, nullable=False)  
 password = db.Column(db.String, nullable=False)  
 created\_at = db.Column(db.DateTime, server\_default=db.func.now())

In a file called ‘app.py’ we will set up the following routes for sign up, login, logout, and authorization. We will be utilizing Flask’s session object in order to store data associated with a specific user session across multiple requests. This information, stored on the server-side, will persist and be accessible while the user is logged into the application. In order to add a layer of encryption and security, we will be using ‘bcrypt’ to hash users’ passwords whenever an account is created. Instead of storing the user passwords in plaintext in the application’s database, we will be storing hashed passwords which will be compared to the user’s actual password during authentication.

# app.py   
  
from flask import Flask, request, make\_response, session  
import bcrypt  
  
# import configurations from config.py  
from config import app, db  
  
# importing the users table from database  
from models import User  
  
# route for sign up  
@app.route('/signup', methods=["POST"])  
def signup():  
 if request.method == "POST":  
 rq = request.get\_json()  
 username = rq['username']  
 password = rq['password']  
  
 # password hashing/salting using bcrypt  
 salt = bcrypt.gensalt()  
 hashed\_password = bcrypt.hashpw(password.encode('utf-8'), salt)  
  
 new\_user = User(  
 username=username,  
 password=hashed\_password.decode('utf-8'),  
 )  
 if new\_user:  
 db.session.add(new\_user)  
 db.session.commit()  
 session['user\_id'] = new\_user.id  
 return make\_response(new\_user.to\_dict(), 201)  
 else:  
 return {'errors': ['Invalid username or password. Please try again.']}, 401  
  
# route for login  
@app.route('/login', methods=["POST"])  
def login():  
 if request.method == "POST":  
 rq = request.get\_json()  
 # find the user with the corresponding username  
 user = User.query.filter(User.username.like(f"%{rq['username']}%")).first()  
  
 # check the password with the hashed password in our database  
 if user and bcrypt.checkpw(rq['password'].encode('utf-8'), user.password.encode('utf-8')):  
 session['user\_id'] = user.id  
 return make\_response(user.to\_dict(), 200)  
 else:  
 return {'errors': ['Invalid username or password. Please try again.']}, 401  
  
# route for logout  
@app.route('/logout', methods=["DELETE"])  
def logout():  
 if request.method == "DELETE":  
 session['user\_id'] = None  
 response = make\_response('', 204)  
 return response  
  
# route for authorization  
@app.route('/authorize')  
def authorize():  
 user\_id = session.get('user\_id')  
 if not user\_id:  
 return {'errors': 'You must be logged in to do that. Please log in or sign up.'}, 401  
 else:  
 user = User.query.filter(User.id == user\_id).first()  
 if user:  
 return make\_response(user.to\_dict(), 200)  
  
# run the server using this command: python app.py  
if \_\_name\_\_ == '\_\_main\_\_':  
 app.run(port=5555, debug=True)

Remember to make necessary migrations or revisions using these commands

# to set up Flask-Migrate  
$ flask db init migrations  
  
# create an initial empty revision  
$ flask db revision -m 'Create DB'  
  
# run these commands to track your modifications made in models.py  
$ flask db revision --autogenerate -m '<descriptive message>'  
$ flask db upgrade head

Lastly, run the server using the following command

$ python app.py

**Accessing API endpoints**

Now that you’ve set up your endpoints for authentication and authorization you can test them by using tools such as Postman or Thunder Client. The app should be running on port 5555. These are the created endpoints along with their request methods.

# signup - POST  
http://localhost:5555/signup  
  
# login - POST  
http://localhost:5555/login  
  
# logout - DELETE  
http://localhost:5555/logout  
  
# authorize - GET  
http://localhost:5555/authorize

Example: JSON Body for /signup

{  
 "username": "bob",  
 "password": "ross"  
}

Once all the endpoints have been tested, you can connect this to the front-end to create a full-stack application with user authentication and authorization!

There are many different methods for establishing user authentication and authorization. Here are some helpful resources to learn more about them: