TechTreks Backend Documentation

Complete Guide to Your Flask Authentication System

Table of Contents

1. File Structure Overview
2. models.py - Database Structure
3. app.py - Application Setup
4. auth/login.py - Authentication Routes
5. init_db.py - Database Initialization
6. How Frontend Interacts with Backend
7. Next Features to Implement

File Structure Overview

```
backend/
├── app.py              # Main Flask application and configuration
├── models.py           # Database models (User table structure)
├── init_db.py          # One-time database initialization script
├── auth/
│   └── login.py        # Authentication routes (register, login)
├── instance/
│   └── users.db        # SQLite database file (created after init_db.py
runs)
└── requirements.txt    # Python dependencies
```

---

models.py - Database Structure

Complete Code

```
from flask_sqlalchemy import SQLAlchemy
```

```python
# Create db instance
db = SQLAlchemy()

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True, nullable=False)
    password_hash = db.Column(db.String(128), nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)

    def __repr__(self):
        return f'<User {self.username}>'
```

What This Does

`db = SQLAlchemy()`

- Creates a database connection object
- Created WITHOUT a Flask app to avoid circular imports
- Other files import this `db` instance
- This is called the "application factory pattern"

`class User(db.Model)`

- Defines the structure of the `user` table in the database
- `db.Model` makes this class represent a database table
- Each class attribute becomes a column in the table

Column Definitions:

Column	Type	Constraints	Purpose
`id`	Integer	primary_key=True	Unique identifier for each user
`username`	String(80)	unique=True, nullable=False	User's login name, must be unique
`password_hash`	String(128)	nullable=False	Encrypted password (never plain text!)
`email`	String(120)	unique=True, nullable=False	User's email, must be unique

Key Concepts:

- `primary_key=True` - This is the unique ID for each row
- `unique=True` - No two users can have the same value

- `nullable=False` - This field is required (can't be empty)
- Password is stored as a **hash** (one-way encryption), never plain text

Frontend Interaction:

- Frontend never directly touches this file
- Frontend sends data → backend routes → models save/retrieve from database
- This is the "data layer"

---

app.py - Application Setup

Complete Code

```python
from flask import Flask, jsonify, request
from flask_cors import CORS
from auth.login import auth as auth_bp
from models import db
import logging
import os

app = Flask(__name__)

# Database configuration
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialize database
db.init_app(app)

app.secret_key = os.environ.get("FLASK_SECRET_KEY", "dev-secret-change-in-
production")

IS_PROD = os.environ.get("FLASK_ENV") == "production"

# Session cookie settings
app.config.update(
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SECURE=False,  # False for local HTTP development
    SESSION_COOKIE_SAMESITE='Lax',
    SESSION_COOKIE_DOMAIN=None,
    PERMANENT_SESSION_LIFETIME=3600,
)
```

```
# CORS configuration
CORS(app,
     resources={r"/*": {"origins": "http://localhost:3000"}},
     supports_credentials=True,
     allow_headers=["Content-Type", "Authorization"],
     methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"])

# Logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

app.register_blueprint(auth_bp, url_prefix="/auth")

@app.route("/")
def welcome():
    return "Backend is running on port 5001!"

if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

Detailed Explanation

Imports

```
from flask import Flask, jsonify, request
```

- **Flask** - Creates the web application
- **jsonify** - Converts Python dictionaries to JSON responses
- **request** - Accesses incoming HTTP request data (JSON body, headers)

```
from flask_cors import CORS
```

- **CORS** - Handles Cross-Origin Resource Sharing
- Allows your frontend (port 3000) to talk to backend (port 5001)

```
from auth.login import auth as auth_bp
```

- Imports your authentication routes
- `auth_bp` is a Blueprint (module of related routes)

```
from models import db
```

- Imports the database instance from models.py

Application Creation

```
app = Flask(__name__)
```

- Creates the Flask application object
- `__name__` tells Flask where to look for templates/static files

Database Configuration

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
```

`SQLALCHEMY_DATABASE_URI`

- Tells Flask where the database file is
- `sqlite:///` means use SQLite database (file-based)
- `users.db` is the actual database file
- Created in the `backend/` folder

`TRACK_MODIFICATIONS = False`

- Disables a feature you don't need
- Saves memory and improves performance

```
db.init_app(app)
```

Critical! Connects your `db` object from models.py to this Flask app

- Without this line, database operations won't work
- It tells SQLAlchemy "use THIS Flask app's configuration"

Secret Key

```
app.secret_key = os.environ.get("FLASK_SECRET_KEY", "dev-secret-change-in-
production")
```

Why You Need This:

- Flask uses this to encrypt session cookies

- Without it, sessions won't work
- The cookie data is signed with this key so users can't tamper with it

How It Works:

- `os.environ.get("FLASK_SECRET_KEY", default)` looks for environment variable first
- Falls back to "dev-secret..." if not found
- **In production:** Set `FLASK_SECRET_KEY` as environment variable with random 32+ character string

Environment Detection

```
IS_PROD = os.environ.get("FLASK_ENV") == "production"
```

- Checks if you're running in production or development
- Used to change security settings based on environment

Session Cookie Configuration

```
app.config.update(
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SECURE=False,
    SESSION_COOKIE_SAMESITE='Lax',
    SESSION_COOKIE_DOMAIN=None,
    PERMANENT_SESSION_LIFETIME=3600,
)
```

`SESSION_COOKIE_HTTPONLY=True`

- Prevents JavaScript from reading the cookie
- Protects against XSS (cross-site scripting) attacks
- Cookie can only be sent by browser, not accessed via `document.cookie`

`SESSION_COOKIE_SECURE=False`

- When `True`, cookie only sent over HTTPS
- Set to `False` for local development (HTTP)
- **Must be `True` in production with HTTPS**

`SESSION_COOKIE_SAMESITE='Lax'`

- Controls when cookies are sent with cross-origin requests
- **`Lax`** = sent with GET requests from other sites, not POST (your current setting)

- `Strict` = only sent from same exact site
- `None` = always sent (requires HTTPS and SECURE=True)
- You use `Lax` because frontend (port 3000) and backend (port 5001) are different origins

`SESSION_COOKIE_DOMAIN=None`

- Restricts which domain can use the cookie
- `None` = only the exact domain that set it (localhost:5001)
- In production: might use `.yourdomain.com` to share across subdomains

`PERMANENT_SESSION_LIFETIME=3600`

- Session expires after 3600 seconds (1 hour)
- After this time, user must log in again

CORS Configuration

```
CORS(app,
     resources={r"/*": {"origins": "http://localhost:3000"}},
     supports_credentials=True,
     allow_headers=["Content-Type", "Authorization"],
     methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"])
```

Why You Need CORS:

- Your frontend runs on `http://localhost:3000`
- Your backend runs on `http://localhost:5001`
- Browsers block requests between different origins (security feature)
- CORS tells the browser "it's okay, these two can talk"

Breaking Down Each Part:

`resources={r"/*": {"origins": "http://localhost:3000"}}`

- `r"/*"` = apply CORS rules to all routes
- Only allow requests from `http://localhost:3000` (your React app)
- In production, change to your actual domain(s)

`supports_credentials=True`

- Allows cookies to be sent with requests
- **Critical for session authentication**
- Without this, session cookie won't be sent from frontend to backend

- Frontend must also use `credentials: "include"` in fetch

`allow_headers=["Content-Type", "Authorization"]`

- Allows these specific headers in requests
- `Content-Type: application/json` needed to send JSON
- `Authorization` for future token-based auth

`methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"]`

- Allows these HTTP methods
- **`OPTIONS`** is the "preflight request" - browser sends this before POST/PUT/DELETE
- Without `OPTIONS`, CORS will fail

What Happens Without Proper CORS:

```
Access to fetch at 'http://localhost:5001/auth/login' from origin
'http://localhost:3000'
has been blocked by CORS policy
```

Blueprint Registration

```
app.register_blueprint(auth_bp, url_prefix="/auth")
```

Blueprints Explained:

- A Blueprint is a way to organize related routes
- All routes in `auth.login` will have `/auth` prefix
- Example: `@auth.route("/login")` becomes `/auth/login`
- Helps organize large applications (auth routes, API routes, admin routes)

Application Run

```
if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

- Only runs if you execute `python app.py` directly
- `debug=True` - auto-reloads when you change code, shows detailed errors
- `port=5001` - runs on this port (5000 was taken by macOS ControlCenter)

---

auth/login.py - Authentication Routes

Complete Code

```
from flask import Blueprint, request, jsonify, session
from werkzeug.security import check_password_hash, generate_password_hash
from models import db, User

auth = Blueprint("auth", __name__)

@auth.route("/register", methods=["POST"])
def register():
    data = request.get_json() or {}
    username = data.get("username")
    password = data.get("password")
    email = data.get("email")

    if not username or not password or not email:
        return jsonify({"ok": False, "error": "All fields required"}), 400

    if User.query.filter_by(username=username).first():
        return jsonify({"ok": False, "error": "Username already taken"}),
409

    if User.query.filter_by(email=email).first():
        return jsonify({"ok": False, "error": "Email already registered"}),
409

    password_hash = generate_password_hash(password)
    new_user = User(username=username, email=email,
password_hash=password_hash)

    db.session.add(new_user)
    db.session.commit()

    return jsonify({"ok": True, "msg": "User registered"}), 201

@auth.route("/login", methods=["POST"])
def login():
    data = request.get_json() or {}
    username = data.get("username")
    password = data.get("password")

    if not username or not password:
        return jsonify({"error": "missing credentials"}), 400
    user = User.query.filter_by(username=username).first()

    if not user or not check_password_hash(user.password_hash, password):
        return jsonify({"error": "invalid credentials"}), 401

    session.clear()
    session['user_id'] = user.id
    session.permanent = True

    return jsonify({"ok": True, "msg": "login successful"}), 200
```

Detailed Explanation

Imports

```
from flask import Blueprint, request, jsonify, session
```

- **Blueprint** - Creates a module of related routes
- **request** - Accesses incoming HTTP request data
- **jsonify** - Converts Python dict to JSON response
- **session** - Stores user data between requests (who's logged in)

```
from werkzeug.security import check_password_hash, generate_password_hash
```

- **check_password_hash** - Verifies passwords securely
- **generate_password_hash** - Encrypts passwords before storing

```
from models import db, User
```

- Imports database instance and User model

Blueprint Creation

```
auth = Blueprint("auth", __name__)
```

- Creates a Blueprint named "auth"
- Groups all authentication-related routes together
- Registered in app.py with `/auth` prefix

Register Route

```
@auth.route("/register", methods=["POST"])
def register():
```

- `@auth.route("/register")` defines the URL endpoint
- Full URL: `http://localhost:5001/auth/register`
- `methods=["POST"]` - only accepts POST requests

```
    data = request.get_json() or {}
```

- Gets JSON data from request body
- `or {}` defaults to empty dict if no JSON

```
    username = data.get("username")
    password = data.get("password")
    email = data.get("email")
```

- Extracts fields from JSON
- Frontend sends: `{"username": "john", "password": "secret123", "email": "john@example.com"}`

```
    if not username or not password or not email:
        return jsonify({"ok": False, "error": "All fields required"}), 400
```

- **Validation:** Checks if all fields are present
- Returns error with HTTP status 400 (Bad Request)

```
    if User.query.filter_by(username=username).first():
        return jsonify({"ok": False, "error": "Username already taken"}),
409
```

- `User.query.filter_by(username=username)` - searches database for existing username
- `.first()` - gets first result (or None if not found)
- Returns 409 (Conflict) if user already exists

```
    password_hash = generate_password_hash(password)
```

Password Hashing Explained:

- **NEVER store plain passwords in database!**
- `generate_password_hash()` uses bcrypt/pbkdf2 to encrypt
- One-way encryption - you can't "decrypt" it back to original
- Example: `"password123"` → `"pbkdf2:sha256:260000$xyz...abc"`
- Even if database is stolen, attackers can't read passwords

```
    new_user = User(username=username, email=email,
password_hash=password_hash)
    db.session.add(new_user)
    db.session.commit()
```

Database Operations:

1. `User(...)` - Creates a new User object in memory (not saved yet)
2. `db.session.add(new_user)` - Stages it for saving
3. `db.session.commit()` - Actually saves to database
4. Like "Create file" → "Save As" → "Confirm" workflow

```
    return jsonify({"ok": True, "msg": "User registered"}), 201
```

- Returns success with 201 (Created) status code

Login Route

```
@auth.route("/login", methods=["POST"])
def login():
    data = request.get_json() or {}
    username = data.get("username")
    password = data.get("password")
```

- Same pattern as register - get JSON data from request

```
    if not username or not password:
        return jsonify({"error": "missing credentials"}), 400
```

- Basic validation

```
    user = User.query.filter_by(username=username).first()
```

- Look up user in database by username

```
    if not user or not check_password_hash(user.password_hash, password):
        return jsonify({"error": "invalid credentials"}), 401
```

Security Note:

- Returns same error for "user not found" and "wrong password"
- Prevents attackers from discovering which usernames are valid
- `check_password_hash()` compares entered password with stored hash
- Returns 401 (Unauthorized) for invalid credentials

```
    session.clear()
    session['user_id'] = user.id
    session.permanent = True
```

Session Management - This is Key:

1. `session.clear()`

- Removes any old session data
- Prevents "session fixation" attacks
- Example attack: Attacker sets a session ID, you log in, they hijack that session

2. `session['user_id'] = user.id`

- Stores the user's ID in the session
- Flask encrypts this data and sends it as a cookie to the browser
- On next request, Flask decrypts the cookie and `session['user_id']` is available again

3. `session.permanent = True`

- Makes session last for `PERMANENT_SESSION_LIFETIME` (1 hour in your config)
- Without this, session ends when browser closes

How Sessions Work:

```
1. User logs in → Flask stores {'user_id': 5} in session
2. Flask encrypts session data with app.secret_key
3. Sends encrypted data as "session" cookie to browser
```

```
4. Browser stores cookie
5. On next request, browser automatically sends cookie
6. Flask decrypts cookie → session['user_id'] is 5 again
7. You can check if user is logged in: if 'user_id' in session
```

```
    return jsonify({"ok": True, "msg": "login successful"}), 200
```

- Returns success with 200 (OK) status

---

init_db.py - Database Initialization

Complete Code

```
from app import app, db
from models import User
import os

def init_db():
    os.makedirs('instance', exist_ok=True)

    with app.app_context():
        db.create_all()
        print("Database initialized successfully!")

if __name__ == "__main__":
    init_db()
```

What This Does

```
os.makedirs('instance', exist_ok=True)
```

- Creates `instance` folder if it doesn't exist
- `exist_ok=True` means "don't error if folder already exists"

```
with app.app_context():
    db.create_all()
```

- `app.app_context()` - Needed because SQLAlchemy requires Flask app context
- `db.create_all()` - Reads your models and creates actual database tables

- Creates `users.db` with a `user` table matching your User model

When to Run This

Run ONCE when setting up:

```
cd backend
python3 init_db.py
```

Also run when:

- First time setting up the project
- After deleting the database
- When you want to reset the database
- When testing

Why Separate File?

- You don't want to recreate tables every time the app starts
- That would delete all user data!
- Run manually when needed

How Frontend Interacts with Backend

Registration Flow

```
┌─────────────┐
│   User      │
│  fills out  │
│   form in   │
│ Register.jsx│
└──────┬──────┘
       │
       │
       │ Clicks "Create Account"
       ▼
┌─────────────────────────────────────────────────┐
│  handleSubmit() in Register.jsx                 │
│                                                 │
│  fetch("http://localhost:5001/auth/register", {│
│    method: "POST",                              │
│    headers: {"Content-Type": "application/json"}│
│    credentials: "include", ◄── Allows cookies  │
│    body: JSON.stringify({username, email, pwd}) │
│  })                                             │
└──────────────┬──────────────────────────────────┘
               │
               │ HTTP POST Request
               ▼
┌─────────────────────────────────────────────┐
│  Backend: /auth/register                    │
│                                             │
│  1. Validates data (all fields present?)    │
│  2. Checks if username exists               │
│  3. Checks if email exists                  │
│  4. Hashes password                         │
│  5. Creates new User object                 │
│  6. Saves to database (db.session.commit())│
│  7. Returns success JSON                    │
└──────────────┬──────────────────────────────┘
               │
               │ Returns: {"ok": true, "msg": "User registered"}
               ▼
┌──────────────────────────────────────────────┐
│  Frontend: Receives success response        │
│                                              │
│  Automatically logs user in:                 │
│  fetch("http://localhost:5001/auth/login")   │
└──────────────┬───────────────────────────────┘
               │
               ▼
┌─────────────────────────────────────────────┐
│  Backend: /auth/login                       │
│                                             │
│  1. Checks credentials                      │
│  2. Sets session['user_id'] = user.id       │
│  3. Sends encrypted session cookie          │
└──────────────┬──────────────────────────────┘
               │
               │ Returns with Set-Cookie header
               ▼
┌──────────────────────────────────────────────┐
│  Browser receives and stores session cookie │
└──────────────┬───────────────────────────────┘
               │
               ▼
┌──────────────────────────────────────────────┐
│  Frontend: Redirects to home page            │
│  navigate("/")                             │
└────────────────────────────────────────────┘
```

Login Flow

```
┌─────────────┐
│   User      │
│  fills out  │
│   form in   │
│ LoginPage   │
└─────────────┘
       │
       │
       │ Clicks "Log In"
       ▼
┌────────────────────────────────────────────────┐
│  handleSubmit() in LoginPage.jsx               │
│                                                │
│  fetch("http://localhost:5001/auth/login", {   │
│    method: "POST",                             │
│    headers: {"Content-Type": "application/json"}│
│    credentials: "include",                     │
│    body: JSON.stringify({username, password})  │
│  })                                            │
└────────────────────────────────────────────────┘
               │
               │
               │ HTTP POST Request
               ▼
┌─────────────────────────────────────────────┐
│  Backend: /auth/login                       │
│                                             │
│  1. Gets username and password from JSON    │
│  2. Queries database for user               │
│  3. Checks password hash                    │
│  4. If valid:                               │
│     - session.clear()                       │
│     - session['user_id'] = user.id          │
│     - session.permanent = True              │
│  5. Returns success                         │
└─────────────────────────────────────────────┘
               │
               │
               │ Returns: {"ok": true, "msg": "login successful"}
               │ With Set-Cookie: session=eyJfcGVybWF...
               ▼
┌──────────────────────────────────────────────┐
│  Browser stores session cookie               │
│                                              │
│  Cookie details:                             │
│  - Name: session                             │
│  - Value: eyJfcGVybWF... (encrypted)         │
│  - Domain: localhost                         │
│  - HttpOnly: true                            │
│  - SameSite: Lax                             │
│  - Expires: 1 hour                           │
└──────────────────────────────────────────────┘
               │
               │
               ▼
┌──────────────────────────────────────────────┐
│  Frontend: Redirects to home                 │
│  navigate("/")                               │
└──────────────────────────────────────────────┘

       │
       │
       ▼
┌──────────────────────────────────────────────┐
│  Future requests automatically include       │
│  session cookie                              │
│                                              │
│  Browser sends:                              │
│  Cookie: session=eyJfcGVybWF...              │
│                                              │
│  Backend can check:                          │
│  if 'user_id' in session:                    │
│      # User is logged in                     │
└──────────────────────────────────────────────┘
```

Key Points About Frontend-Backend Communication

1. CORS and Credentials

```
fetch("http://localhost:5001/auth/login", {
  credentials: "include",  // ←── MUST include this for cookies to work
  // ...
})
```

- Without `credentials: "include"`, cookies won't be sent/received
- Backend must have `supports_credentials=True` in CORS config

2. Content-Type Header

```
headers: {
  "Content-Type": "application/json"
}
```

- Tells backend "I'm sending JSON data"
- Backend can then use `request.get_json()`

3. Session Cookie Flow

```
Login → Backend sets cookie → Browser stores it →
Future requests automatically include it → Backend recognizes user
```

4. Error Handling

```
if (!res.ok) {
  // Handle error (400, 401, 409, etc.)
  const data = await res.json();
  setError(data.error);
}
```

Next Features to Implement

1. Logout Functionality (Easy - Start Here)

Backend: Add to `auth/login.py`

```
@auth.route("/logout", methods=["POST"])
def logout():
    session.clear()
    return jsonify({"ok": True, "msg": "logged out"}), 200
```

Frontend: Add logout button

```
const handleLogout = async () => {
  await fetch("http://localhost:5001/auth/logout", {
    method: "POST",
    credentials: "include"
  });
  navigate("/login");
};
```

```
// In your component:
<button onClick={handleLogout}>Logout</button>
```

---

2. Check Current User (Easy)

Backend: Add to `auth/login.py`

```
@auth.route("/current_user", methods=["GET"])
def current_user():
    user_id = session.get("user_id")
    if not user_id:
        return jsonify({"ok": False, "user": None}), 200

    user = User.query.get(user_id)
    if not user:
        return jsonify({"ok": False, "user": None}), 200

    return jsonify({
        "ok": True,
        "user": {
            "id": user.id,
            "username": user.username,
            "email": user.email
        }
    }), 200
```

Frontend: Check on page load

```
useEffect(() => {
  const checkAuth = async () => {
    const res = await fetch("http://localhost:5001/auth/current_user", {
      credentials: "include"
    });
    const data = await res.json();
    if (data.ok) {
      setUser(data.user);
    }
  };
  checkAuth();
}, []);
```

Use Cases:

- Show username in navbar
- Check if user is logged in on page load
- Protect routes (redirect to login if not authenticated)

---

3. Protected Routes (Medium)

Backend: Create decorator in `auth/login.py`

```python
from functools import wraps

def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
            return jsonify({"error": "Login required"}), 401
        return f(*args, **kwargs)
    return decorated_function

# Use it:
@auth.route("/profile", methods=["GET"])
@login_required
def profile():
    user = User.query.get(session['user_id'])
    return jsonify({
        "username": user.username,
        "email": user.email
    })
```

Frontend: Protect routes with React Router

```javascript
import { Navigate } from 'react-router-dom';

const ProtectedRoute = ({ children }) => {
  const [isAuth, setIsAuth] = useState(null);

  useEffect(() => {
    fetch("http://localhost:5001/auth/current_user", {
      credentials: "include"
    })
    .then(res => res.json())
    .then(data => setIsAuth(data.ok));
  }, []);

  if (isAuth === null) return <div>Loading...</div>;
  return isAuth ? children : <Navigate to="/login" />;
};

// In your routes:
<Route path="/dashboard" element={
  <ProtectedRoute>
    <Dashboard />
  </ProtectedRoute>
} />
```

---

4. Email Verification (Advanced)

Step 1: Add fields to User model

```
class User(db.Model):
    # ...existing fields...
    email_verified = db.Column(db.Boolean, default=False)
    verification_token = db.Column(db.String(100), unique=True)
```

Step 2: Generate token on registration

```
import secrets

@auth.route("/register", methods=["POST"])
def register():
    # ...existing code...
    verification_token = secrets.token_urlsafe(32)
    new_user = User(
        username=username,
        email=email,
        password_hash=password_hash,
        verification_token=verification_token
    )
    # ...save to database...

    # Send email with verification link
    send_verification_email(email, verification_token)
```

Step 3: Verify endpoint

```python
@auth.route("/verify/<token>", methods=["GET"])
def verify_email(token):
    user = User.query.filter_by(verification_token=token).first()
    if not user:
        return jsonify({"error": "Invalid token"}), 400

    user.email_verified = True
    user.verification_token = None
    db.session.commit()

    return jsonify({"ok": True, "msg": "Email verified!"})
```

---

5. Password Reset (Advanced)

Step 1: Request reset

```python
@auth.route("/request_reset", methods=["POST"])
def request_reset():
    email = request.get_json().get("email")
    user = User.query.filter_by(email=email).first()

    if user:
        reset_token = secrets.token_urlsafe(32)
        user.reset_token = reset_token
        user.reset_token_expiry = datetime.utcnow() + timedelta(hours=1)
        db.session.commit()

        # Send email with reset link
        send_reset_email(email, reset_token)

    # Always return success (don't reveal if email exists)
    return jsonify({"ok": True, "msg": "If email exists, reset link sent"})
```

Step 2: Reset password

```python
@auth.route("/reset_password/<token>", methods=["POST"])
def reset_password(token):
    user = User.query.filter_by(reset_token=token).first()

    if not user or user.reset_token_expiry < datetime.utcnow():
        return jsonify({"error": "Invalid or expired token"}), 400

    new_password = request.get_json().get("password")
    user.password_hash = generate_password_hash(new_password)
    user.reset_token = None
    user.reset_token_expiry = None
    db.session.commit()

    return jsonify({"ok": True, "msg": "Password reset successful"})
```

---

6. User Profile Updates (Medium)

Backend:

```
@auth.route("/update_profile", methods=["PUT"])
@login_required
def update_profile():
    data = request.get_json()
    user = User.query.get(session['user_id'])

    if data.get("email"):
        # Check if email already taken
        existing = User.query.filter_by(email=data["email"]).first()
        if existing and existing.id != user.id:
            return jsonify({"error": "Email already in use"}), 409
        user.email = data["email"]

    if data.get("username"):
        existing = User.query.filter_by(username=data["username"]).first()
        if existing and existing.id != user.id:
            return jsonify({"error": "Username already taken"}), 409
        user.username = data["username"]

    db.session.commit()
    return jsonify({"ok": True, "msg": "Profile updated"})

@auth.route("/change_password", methods=["PUT"])
@login_required
def change_password():
    data = request.get_json()
    user = User.query.get(session['user_id'])

    if not check_password_hash(user.password_hash,
data.get("current_password")):
        return jsonify({"error": "Current password incorrect"}), 401

    user.password_hash = generate_password_hash(data.get("new_password"))
    db.session.commit()

    return jsonify({"ok": True, "msg": "Password changed"})
```

---

7. Rate Limiting (Medium - Important for Security)

Install Flask-Limiter:

```
pip install Flask-Limiter
```

Add to `app.py`:

```
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address

limiter = Limiter(
    app=app,
    key_func=get_remote_address,
    default_limits=["200 per day", "50 per hour"]
)
```

Apply to login route:

```
@auth.route("/login", methods=["POST"])
@limiter.limit("5 per minute")  # Prevent brute force attacks
def login():
    # ...existing code...
```

---

8. Better Validation (Easy)

Add validation functions:

```
import re
```

```python
def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return re.match(pattern, email) is not None

def validate_password(password):
    """
    Password must:
    - Be at least 8 characters
    - Contain uppercase letter
    - Contain lowercase letter
    - Contain number
    - Contain special character
    """
    if len(password) < 8:
        return False, "Password must be at least 8 characters"
    if not re.search(r'[A-Z]', password):
        return False, "Password must contain an uppercase letter"
    if not re.search(r'[a-z]', password):
        return False, "Password must contain a lowercase letter"
    if not re.search(r'\d', password):
        return False, "Password must contain a number"
    if not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        return False, "Password must contain a special character"
    return True, ""

def validate_username(username):
    if len(username) < 3 or len(username) > 30:
        return False, "Username must be 3-30 characters"
    if not re.match(r'^[A-Za-z0-9_.-]+$', username):
        return False, "Username can only contain letters, numbers, dots,
dashes, underscores"
    return True, ""
```

Use in register route:

```python
@auth.route("/register", methods=["POST"])
def register():
    # ...get data...

    # Validate username
    valid, msg = validate_username(username)
    if not valid:
        return jsonify({"error": msg}), 400

    # Validate email
    if not validate_email(email):
        return jsonify({"error": "Invalid email format"}), 400

    # Validate password
    valid, msg = validate_password(password)
    if not valid:
        return jsonify({"error": msg}), 400

    # ...continue with registration...
```

---

9. Database Migrations (Medium)

Install Flask-Migrate:

```
pip install Flask-Migrate
```

Add to `app.py`:

```
from flask_migrate import Migrate

migrate = Migrate(app, db)
```

Initialize migrations:

```
flask db init
```

Create migration after changing models:

```
flask db migrate -m "Add email_verified column"
```

Apply migration:

```
flask db upgrade
```

Why use migrations?

- Safely change database schema without losing data
- Version control for your database
- Easy to deploy changes to production

10. User Roles/Permissions (Advanced)

Add role to User model:

```
class User(db.Model):
    # ...existing fields...
    role = db.Column(db.String(20), default="user")  # "user", "admin",
"moderator"
```

Create admin decorator:

```
def admin_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
            return jsonify({"error": "Login required"}), 401

        user = User.query.get(session['user_id'])
        if user.role != 'admin':
            return jsonify({"error": "Admin access required"}), 403

        return f(*args, **kwargs)
    return decorated_function

# Use it:
@auth.route("/admin/users", methods=["GET"])
@admin_required
def list_all_users():
    users = User.query.all()
    return jsonify({
        "users": [{"id": u.id, "username": u.username} for u in users]
    })
```

11. Logging and Monitoring (Medium)

Enhanced logging in `app.py`:

```
import logging
from logging.handlers import RotatingFileHandler
```

```
if not app.debug:
    # Production logging
    file_handler = RotatingFileHandler('app.log', maxBytes=10240,
backupCount=10)
    file_handler.setFormatter(logging.Formatter(
        '%(asctime)s %(levelname)s: %(message)s [in %(pathname)s:%
(lineno)d]'
    ))
    file_handler.setLevel(logging.INFO)
    app.logger.addHandler(file_handler)
    app.logger.setLevel(logging.INFO)
    app.logger.info('Application startup')
```

Log authentication events:

```
@auth.route("/login", methods=["POST"])
def login():
    # ...existing code...

    if not user or not check_password_hash(user.password_hash, password):
        app.logger.warning(f'Failed login attempt for username: {username}')
        return jsonify({"error": "invalid credentials"}), 401

    app.logger.info(f'Successful login: {username}')
    # ...rest of code...
```

---

12. Security Headers (Easy)

Add to `app.py`:

```
@app.after_request
def add_security_headers(response):
    response.headers['X-Content-Type-Options'] = 'nosniff'
    response.headers['X-Frame-Options'] = 'DENY'
    response.headers['X-XSS-Protection'] = '1; mode=block'

    if IS_PROD:
        response.headers['Strict-Transport-Security'] = 'max-age=31536000;
includeSubDomains'

    return response
```

What these do:

- **X-Content-Type-Options: nosniff** - Prevents MIME type sniffing
- **X-Frame-Options: DENY** - Prevents clickjacking attacks
- **X-XSS-Protection** - Enables browser XSS protection
- **Strict-Transport-Security** - Forces HTTPS (production only)

Production Deployment Checklist

Environment Variables

```
# .env file (never commit this!)
FLASK_ENV=production
FLASK_SECRET_KEY=your-super-secret-random-32-char-string
DATABASE_URL=postgresql://user:pass@host:5432/dbname
ALLOWED_ORIGINS=https://yourdomain.com
```

Changes for Production

1. app.py changes:

```python
# Use environment variables
app.secret_key = os.environ.get("FLASK_SECRET_KEY")
if not app.secret_key:
    raise ValueError("FLASK_SECRET_KEY must be set")

IS_PROD = os.environ.get("FLASK_ENV") == "production"

# Production database (PostgreSQL instead of SQLite)
if IS_PROD:
    app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get("DATABASE_URL")
else:
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///users.db'

# Session cookies
app.config.update(
    SESSION_COOKIE_HTTPONLY=True,
    SESSION_COOKIE_SECURE=True,  # Always True in production
    SESSION_COOKIE_SAMESITE='Lax',
    SESSION_COOKIE_DOMAIN=os.environ.get("COOKIE_DOMAIN"),
    PERMANENT_SESSION_LIFETIME=3600,
)
```

```
# CORS
allowed_origins = os.environ.get("ALLOWED_ORIGINS",
"http://localhost:3000").split(",")
CORS(app,
     resources={r"/*": {"origins": allowed_origins}},
     supports_credentials=True,
     allow_headers=["Content-Type", "Authorization"],
     methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"])

# Run configuration
if __name__ == "__main__":
    if IS_PROD:
        # Use Gunicorn in production: gunicorn -w 4 app:app
        pass
    else:
        app.run(debug=True, port=5001)
```

2. Use production server (not Flask dev server):

```
pip install gunicorn
gunicorn -w 4 -b 0.0.0.0:8000 app:app
```

3. Use production database:

```
pip install psycopg2-binary  # PostgreSQL
# or
pip install PyMySQL  # MySQL
```

4. Frontend changes:

```
// config.js
const API_BASE_URL = process.env.REACT_APP_API_URL ||
"http://localhost:5001";
export default API_BASE_URL;

// In components:
import API_BASE_URL from '../config';
fetch(`${API_BASE_URL}/auth/login`, { /* ... */ });
```

5. HTTPS everywhere:

- Use Let's Encrypt for free SSL certificates

- Set `SESSION_COOKIE_SECURE=True`
- Never allow HTTP in production

6. Install requirements:

```
# requirements.txt
Flask==3.0.0
Flask-CORS==4.0.0
Flask-SQLAlchemy==3.1.1
Werkzeug==3.0.1
gunicorn==21.2.0
psycopg2-binary==2.9.9
Flask-Migrate==4.0.5
Flask-Limiter==3.5.0
python-dotenv==1.0.0
```

Summary

What You've Built

✅ **Complete authentication system** with registration and login
✅ **Secure password storage** using bcrypt hashing
✅ **Session-based authentication** with encrypted cookies
✅ **SQLite database** with User model
✅ **CORS configuration** for frontend-backend communication
✅ **Blueprint organization** for scalable code structure
✅ **Security best practices** (HttpOnly cookies, no plain passwords)

Key Concepts Learned

1. **Database Models** - Defining data structure with SQLAlchemy
2. **Password Hashing** - Never store plain passwords
3. **Sessions** - Keeping users logged in between requests
4. **CORS** - Allowing cross-origin requests
5. **Blueprints** - Organizing routes into modules
6. **HTTP Methods** - GET, POST, PUT, DELETE
7. **Status Codes** - 200 (OK), 201 (Created), 400 (Bad Request), 401 (Unauthorized), 409 (Conflict)
8. **JSON API** - Sending and receiving data as JSON

9. **Cookies** - Storing session data in browser
10. **Environment Configuration** - Different settings for dev/production

Your Stack

- **Backend:** Flask (Python)
- **Database:** SQLite (dev) → PostgreSQL (production)
- **Frontend:** React
- **Authentication:** Session-based with cookies
- **Security:** Bcrypt password hashing, HttpOnly cookies, CORS

This is a solid foundation for building full-stack web applications! 🎉

Last Updated: November 12, 2025
Version: 1.0
Project: TechTreks Authentication System