Flask User Authentication for APIs

15 min to complete · By Brandon Gigous

Contents

Right now, users of your site must log in to interact with the app and access all of its content. They do so using the login form of your app, which then gets delegated to Flask-Login through the user session. That's great and everything, but your API will have to verify and keep track of user credentials a little differently.

A REST API like the one you're making is stateless, after all, but then again it can't allow clients access to otherwise restricted information. To be able to keep the "REST" title, the server can't "remember" anything about the client or their requests, which includes user credentials. So, anyone who wants to interact with the API must provide the credentials to the server for every request.

In this lesson, you'll program two ways for your clients to authenticate through the API: HTTP authentication and token-based authentication.

HTTP Authentication

To authenticate a user via HTTP, they would need to send their credentials through the Authorization request header. Sounds complicated, but with the help of Flask-HTTPAuth, you don't have to worry about the details. This wrapper can be installed like so:

```
(env) $ pip install flask-httpauth
```

And with that, you're off to the races! Er, I mean your close to ready to authenticate your users, whether equine or human.

Using Flask-HTTPAuth

Verifying user credentials with Flask-HTTPAuth is done, like much of Flask, via a convenient decorator. The `verify_password` decorator of the `HTTPBasicAuth` object is applied to a callback function that returns `True` if user verification, given credentials, is successful.

```
# app/api/authentication.py
from flask_httpauth import HTTPBasicAuth
```

```
    auth = HTTPBasicAuth()

    @auth.verify_password
    def verify_password(email_or_token, password):
        if email_or_token == '':
            return False
        user = User.query.filter_by(email=email_or_token).first()
        if not user:
            return False
        g.current_user = user
        return user.verify_password(password)
```

You'll see authentication via tokens in a bit, but for now, `email_or_token` is just an email. The email given by the client is used to find the user, if they exist, and then the provided password is verified just like you've seen before with `User.verify_password`. This code doesn't throw any errors like a 404 if a user isn't found: otherwise that might trigger the HTML error handler, which is no bueno for an API. Instead, when authentication fails, Flask-HTTPAuth will send a 401 status code to the client, indicating that authentication wasn't successful. It will do this automatically without any other work on your part, but it doesn't hurt to help it out a little in order to keep your error messages consistent. In this case, you can override the `error_handler` decorator with another callback:

```
# app/api/authentication.py
from .errors import unauthorized

@auth.error_handler
def auth_error():
    return unauthorized('Invalid credentials')
```

And one more thing: in the `verify_password` callback, the authenticated user can be accessed later by your API with Flask's `g` context variable. That's why `verify_password` assigns `g.current_user` to `user`. You'll see later how it's used.

Guess what: you've just finished implementing basic HTTP authentication for your API! Now onto token-based authentication.

Token Authentication

Ah yes, tokens. You've already learned about tokens from this Python + Flask course section on emails. You used them to verify newly registered user accounts: an email is sent to the user with a link back to your website, and the link contains a token created by your app to ensure the user is legit. It turns out tokens are a great way to verify user accounts for an API as well, and it's not much different either. Here's how it works:

1. The client requests an access token by sending a request to the API with their login credentials
2. If verified successfully, the client receives a unique token from the API
3. Next time the client makes a request to the API, they use the token in place of user credentials

This has a couple advantages. First, it is safer as the user doesn't have to pass around their sensitive login information every time they want to make an API request. Second, along with the convenience for the user comes security for you and your webapp as tokens have an expiration date. Eventually the user will have to re-authenticate, preventing them from a "lifetime membership" to your app's data.

Changes to User Model

Before your app can pass out tokens to users of your API, you have to generate them. This ain't any different than the confirmation tokens you made before, but two new methods in your User model must be made nonetheless:

```
class User(UserMixin, db.Model):
    # ...
    def generate_auth_token(self, expiration_sec):
        s = WebSerializer(current_app.config['SECRET_KEY'],
                          expires_in=expiration_sec)
        return s.dumps({'id': self.id}).decode('utf-8')

    @staticmethod
    def verify_auth_token(token):
        s = WebSerializer(current_app.config['SECRET_KEY'])
        try:
            data = s.loads(token)
        except:
            return None
        return User.query.get(data['id'])
```

If you don't remember the details of how tokens work, feel free to refresh yourself here. The main difference here in `verify_auth_token()` is that if a user token is successfully verified, the User corresponding to the token is returned.

Adding Token Authentication Functionality

Before your API starts handing out tokens, put in the functionality to authenticate a user with a token. To do so just means a few additions to the `verify_password` callback you made before:

```
@auth.verify_password
def verify_password(email_or_token, password):
    if email_or_token == '':
        return False
    if password == '':
        g.current_user = User.verify_auth_token(email_or_token)
        g.token_used = True
        return g.current_user is not None
    user = User.query.filter_by(email=email_or_token).first()
    if not user:
        return False
    g.current_user = user
    g.token_used = False
    return user.verify_password(password)
```

How do you tell if a user passed in a token? Well, chances are if they didn't supply a password, they used a token instead, because remember that tokens allow users to temporarily bypass having to provide that sensitive information every time. In the case of no password, the newly-created `User.verify_auth_token()` is called to grab the User associated with the token, which is stored in `g.current_user`. Then a new `token_used` attribute is set, indicating if the user was verified with a token. If the user wasn't verified successfully with a token, then `False` is returned.

Gimme My Token!

Once you can generate tokens, you have to be able to give them to users! Back in your `api/authentication.py` file, you can make a view function that returns a token to the client via JSON:

```
@api.route('/tokens/', methods=['POST'])
def get_token():
    if g.current_user.is_anonymous or g.token_used:
        return unauthorized('Invalid credentials')
    return jsonify({'token': g.current_user.generate_auth_token(
        expiration_sec=3600), 'expiration': 3600})
```

Only authenticated users can get tokens, and they do so by making a POST request to `/api/v1/tokens/`. Once your app authenticates your users' credentials via `verify_password` using HTTP, it's already taken note of which user that is in `g.current_user`. The `get_token()` function double checks that the user is authenticated, then hands them a token if all goes well, with an expiration of course.

Protecting Your App's Content

Now just like the rest of your app where you restricted access to routes to only logged in users with Flask-Login's `@login_required` decorator, you can do the same with your API. Flask-HTTPAuth has a decorator with the same name you can use from a `HTTPBasicAuth` object.

All of the API routes/endpoints in your app will need to be protected. To make your life a little less tedious, you can require that all users who make requests to endpoints in the `api` blueprint be logged in by applying the `login_required` decorator of `auth` to the `before_request` handler:

```
# app/api/authentication.py
from .errors import forbidden

@api.before_request
@auth.login_required
def before_request():
    if not g.current_user.is_anonymous and \
            not g.current_user.confirmed:
        return forbidden('Unconfirmed account')
```

Fantastic, with your app's security in place you should be all set to allow those logged in user's access to your app's data via the API. You'll learn about it in the next lesson.

- A team of mentors and advisors dedicated to your success
- Weekly 1-on-1 screen share meetings with your professional mentor
- Constant, personal, in-depth support 7 days a week via Discord
- Accountability, feedback, encouragement, and success

Get Mentorship

Summary: HTTP and Token Authentication

HTTP Authentication with Flask-HTTPAuth

- To authenticate a user via HTTP, they would need to send their credentials through the Authorization request header. With the help of Flask-HTTPAuth, you don't have to worry about the details. This wrapper can be installed like so:

```
(env) $ pip install flask-httpauth
```

- Verifying user credentials with Flask-HTTPAuth is done, like much of Flask, via a convenient decorator. The `verify_password` decorator of the `HTTPBasicAuth` object is applied to a callback function that returns `True` if user verification, given credentials, is successful.

Token Authentication

- Tokens are a great way to verify user accounts for an API as well, and it's not much different either. The client requests an access token by sending a request to the API

with their login credentials, If verified successfully, the client receives a unique token from the API, Next time the client makes a request to the API, they use the token in place of user credentials

- Tokens are safer and have an expiration date
- Before your app can pass out tokens to users of your API, you have to generate them. Two new methods in your User model must be made, `generate_auth_token` and `verify_auth_token`
- Add the functionality to authenticate a user with a token. To do so just means a few additions to the `verify_password` callback you made before
- Once you can generate tokens, you have to be able to give them to users! Back in your `api/authentication.py` file, you can make a view function that returns a token to the client via JSON
- All of the API routes/endpoints in your app will need to be protected. You can require that all users who make requests to endpoints in the `api` blueprint be logged in by applying the `login_required` decorator of `auth` to the `before_request` handler

Previous Next → Video: Basic API HTTP Authentication