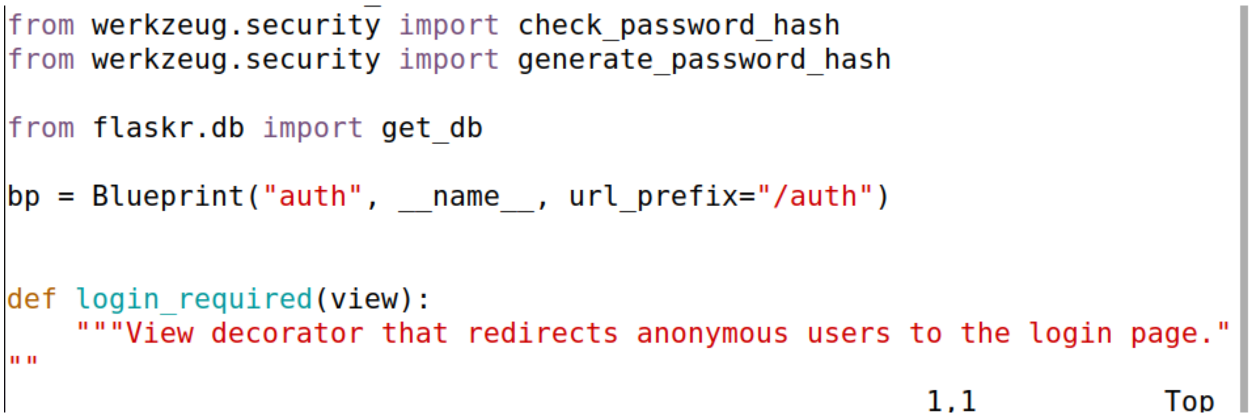
Erik O. Kriz
CSE 4400
Project 3

Task 1: Password: This system uses the werkzeug.security library functions to check/generate passwords. Password hashes are generated with the werkzeug.security.generate_password_hash() function. This function generates sha1 hashes using a number of salt characters, whose length is set by default to 8. This salt is generated with the _sys_rng.choice(SALT_CHARS) function. The system checks the validity of passwords in the check_password_hash function, which generates the hash of an input password, and checks that hash to the one stored on the database. This function is not vulnerable to timing attacks. The passwords are stored using flask's default database system (sqlite3).

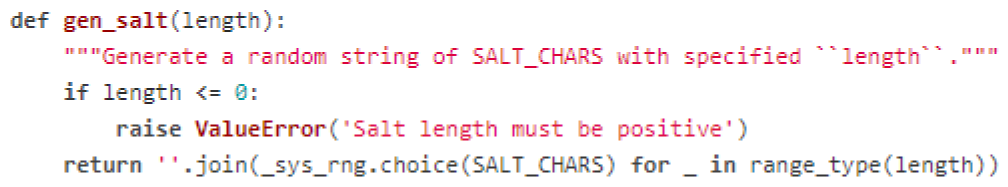


```
from werkzeug.security import check_password_hash
from werkzeug.security import generate_password_hash

from flaskr.db import get_db

bp = Blueprint("auth", __name__, url_prefix="/auth")


def login_required(view):
    """View decorator that redirects anonymous users to the login page."
""
                                                  1,1            Top
```

- Here in auth.py, you can see that this imports werkzeug.security.check_password_hash, and generate_password_hash.

```python
def generate_password_hash(password, method='pbkdf2:sha1', salt_length=8):
    """Hash a password with the given method and salt with with a string of
    the given length.  The format of the string returned includes the method
    that was used so that :func:`check_password_hash` can check the hash.

    The format for the hashed string looks like this::

        method$salt$hash

    This method can **not** generate unsalted passwords but it is possible
    to set the method to plain to enforce plaintext passwords.  If a salt
    is used, hmac is used internally to salt the password.

    If PBKDF2 is wanted it can be enabled by setting the method to
    ``pbkdf2:method:iterations`` where iterations is optional::

        pbkdf2:sha1:2000$salt$hash
        pbkdf2:sha1$salt$hash

    :param password: the password to hash.
    :param method: the hash method to use (one that hashlib supports). Can
                   optionally be in the format ``pbkdf2:<method>[:iterations]``
                   to enable PBKDF2.
    :param salt_length: the length of the salt in letters.
    """
    salt = method != 'plain' and gen_salt(salt_length) or ''
    h, actual_method = _hash_internal(method, salt, password)
    return '%s$%s$%s' % (actual_method, salt, h)



def check_password_hash(pwhash, password):
    """check a password against a given salted and hashed password value.
    In order to support unsalted legacy passwords this method supports
    plain text passwords, md5 and sha1 hashes (both salted and unsalted).

    Returns `True` if the password matched, `False` otherwise.

    :param pwhash: a hashed string like returned by
                   :func:`generate_password_hash`.
    :param password: the plaintext password to compare against the hash.
    """
    if pwhash.count('$') < 2:
        return False
    method, salt, hashval = pwhash.split('$', 2)
    return safe_str_cmp(_hash_internal(method, salt, password)[0], hashval)
```

- Here is the man page for the functions imported from the previous figure. You can see that the default hash method is sha1, and the default salt length is 8. https://tedboy.github.io/flask/_modules/werkzeug/security.html#generate_password_hash



```
def gen_salt(length):
    """Generate a random string of SALT_CHARS with specified ``length``."""
    if length <= 0:
        raise ValueError('Salt length must be positive')
    return ''.join(_sys_rng.choice(SALT_CHARS) for _ in range_type(length))
```

  - This is from the same page as the previous figure. This is how the password functions generate salt.

Session and cookie: A session is a flask object which is built on top of the inherent cookies system in browsers. In the python code, the session is treated as a global object that all view functions can access. Any changes to the session object are therefore noticed by all view functions. The main information held in sessions is the username of the person who is currently logged in, and the secret key which is used to change attributes of the session. The only way to gain a username on your session object is to be authenticated on your machine for that username. When the user visits a page, the session is checked to see if it contains a username. If it does, that means this user has logged in and they can see the page. If there is no username in the session, then the view can simply redirect the user to the login page, or display a default message that the user is not logged in. The verification is vulnerable to timing attacks. The session is encrypted, but not securely. They are encrypted using base 64, meaning that one could decode them if they were able to get the encrypted string. Sessions are held within the sites running code, meaning that any user can inspect their page and find the encrypted session. Decoding it from there is trivial.

Updating the session

To set a key & value:

```
session["KEY"] = "VALUE"
```

In this case, we've assigned the session `USERNAME` key to the users username:

```
session["USERNAME"] = user["username"]
```

If you were to `print(session)` just after we set the `USERNAME` key, you would see (Assuming the username of "julian"):

```
<SecureCookieSession {'USERNAME': 'julian'}>
```

You'll also notice the redirect to `profile`, using:

```
return redirect(url_for("profile"))
```

Redirect takes a URL and redirects the client to it. In this case we've passed it `url_for("profile")`.

`url_for` takes arguments and builds an endpoint, in this case we've just passed the name of a function, `"profile"`, to which it builds a URL.

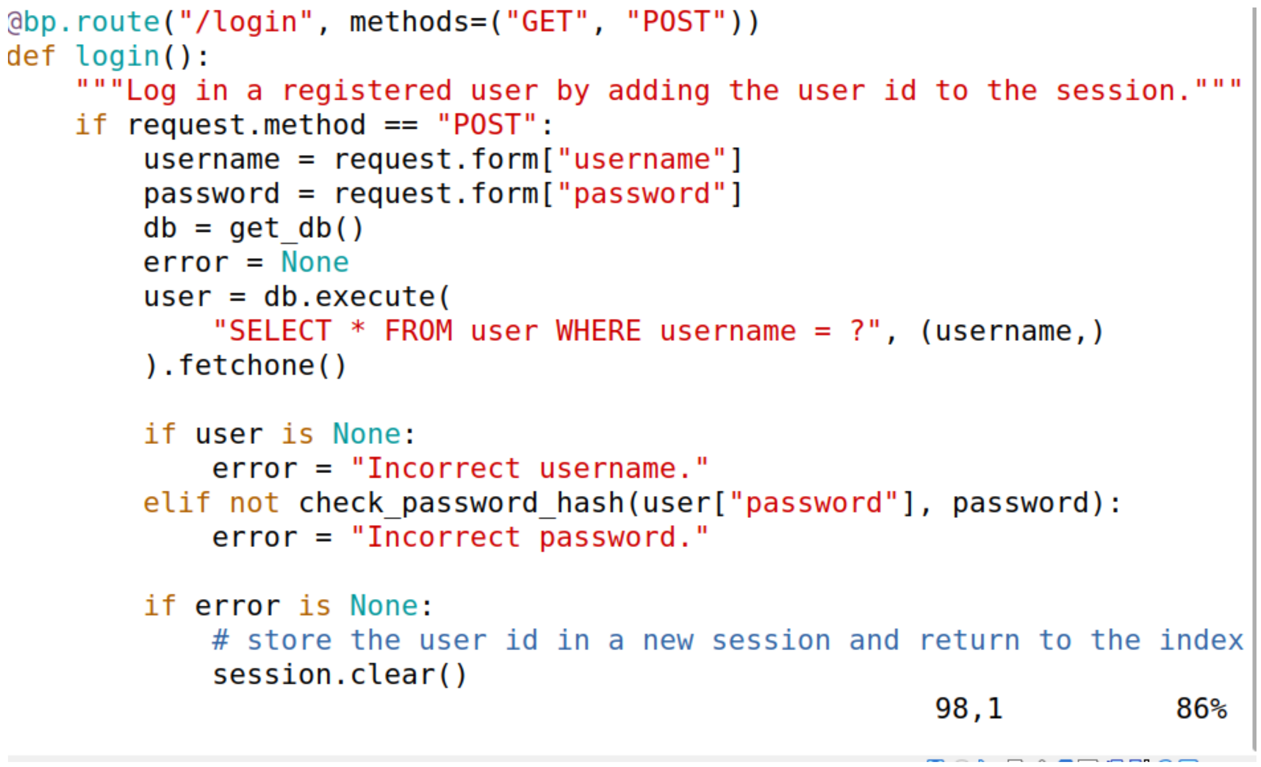
We havent's created the profile route yet so let's go ahead and do so:

Getting the session object

```
@app.route("/profile")
def profile():
```

- This snippet from the site https://pythonise.com/series/learning-flask/flask-session-object describes that the session "rides" on the back of browser cookies, and is visible to anyone with access to the browser as such. It goes on to talk about how the session is only encoded in base64 (no security properties), and as such is vulnerable to anyone looking at the data contained within.

Database: the database as defined in "db.py" is simply a sqlite3 object, and is treated as such within other functions like auth.py. The major vulnerability of sql that one has to look out for is injections attacks. The code within auth.py has been structured in a way (using execute/ ? commands) as to avoid this particular pitall of SQL.



```
@bp.route("/login", methods=("GET", "POST"))
def login():
    """Log in a registered user by adding the user id to the session."""
    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:
            # store the user id in a new session and return to the index
            session.clear()
                                                            98,1          86%
```

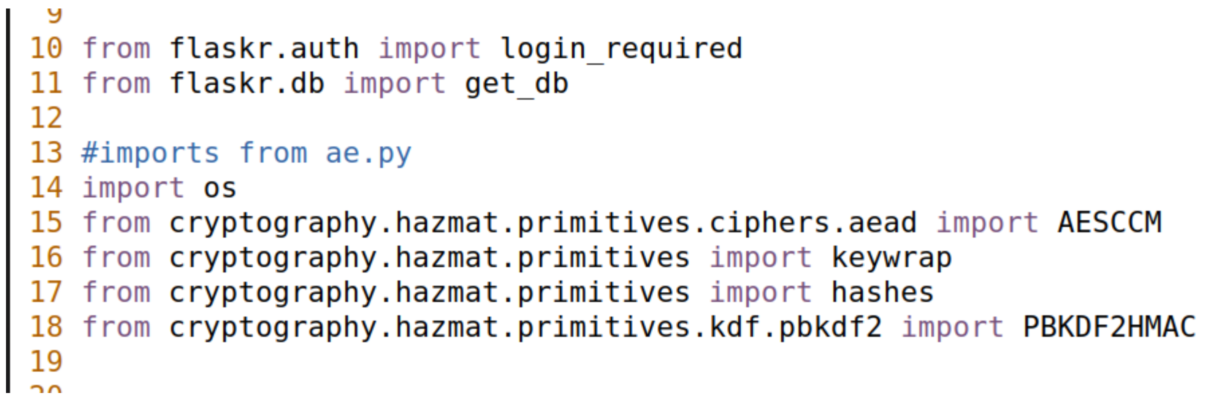
- Here you can see that the line which selects data from the SQLite3 database is formatted to be an execute function.

Output: Flask is a library which is able to use python code to render html files on users screens. When a user visits a page, flask runs a python function which corresponds to that page. This function may check things about the user through their session, and also may run calculations first, but the final return value is the output of the function render_template(). The input to this function is the path to the html template for the page that this python function corresponds to. The returned object is a rendering of that html page, with any alterations/calculations from earlier in the function shown. The function can even display different html based on the logic run before. Most commonly, a function will check the user's session to see if they are logged in, and based on that answer, display different html or possibly redirect the user to a different page.

```python
def index():
    """Show all the posts, most recent first."""
    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)
```

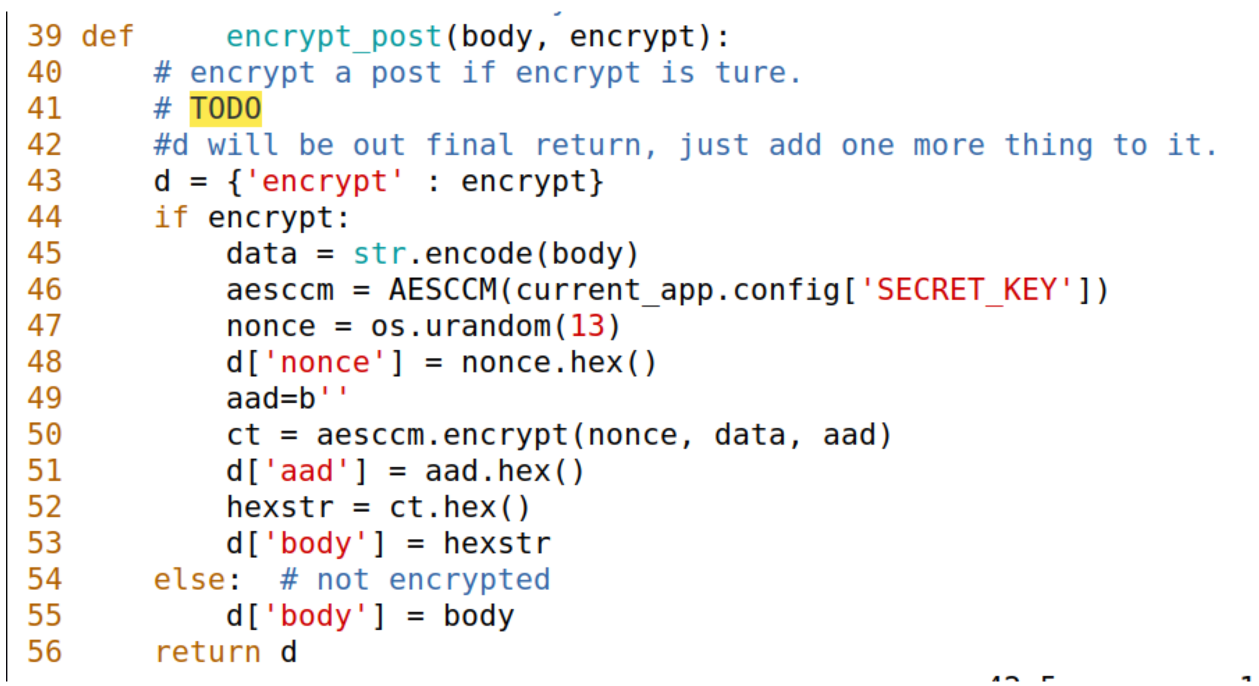
- Here you can see the python functions which execute at the main page of the website (known as the index). Notice how it returns the result of render_template, also notice that the input of render_template is a path to an html template.

Task 2: Below I have shown my updated functions for each of encrypt_post, decrypt_post, and index within the provided blog.py file. Within them, I either encrypt or decrypt the body of a post in the same fashion as ae.py encrypts strings. Any data needed for the decrypt process is simply added to the post as a member of its dictionary.



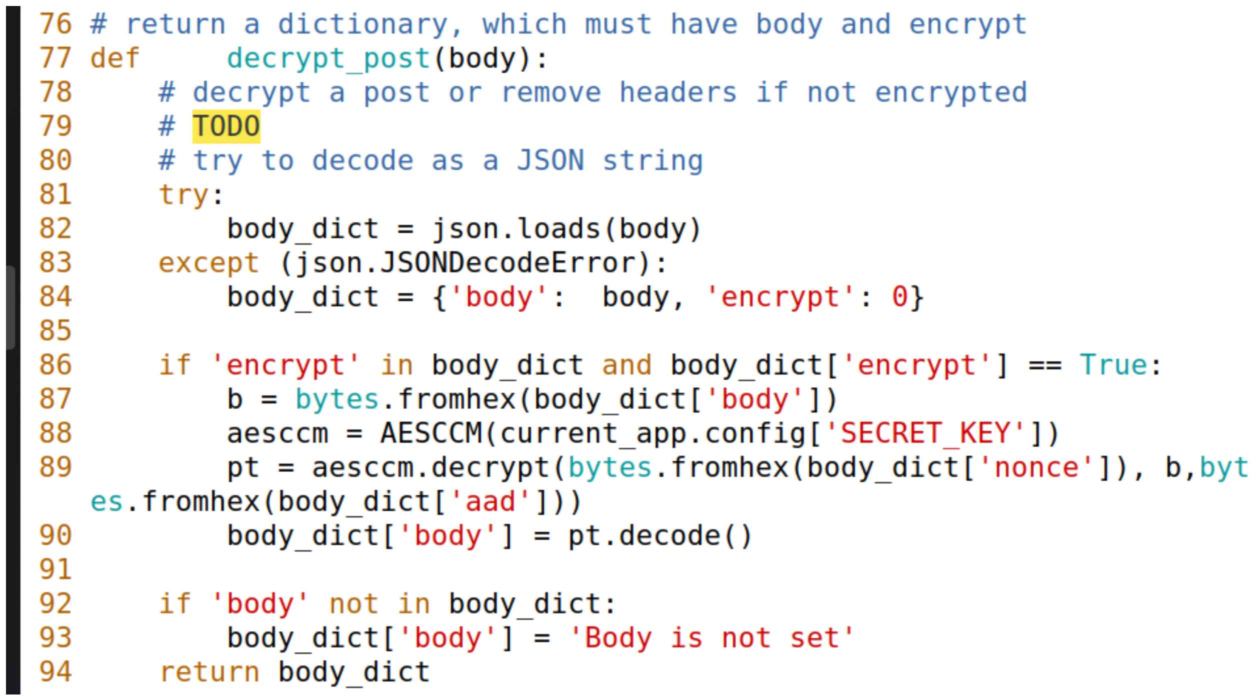
```python
from flaskr.auth import login_required
from flaskr.db import get_db

#imports from ae.py
import os
from cryptography.hazmat.primitives.ciphers.aead import AESCCM
from cryptography.hazmat.primitives import keywrap
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC

```

- Here you can see that I take the imports form ae.py and add them to blog.py



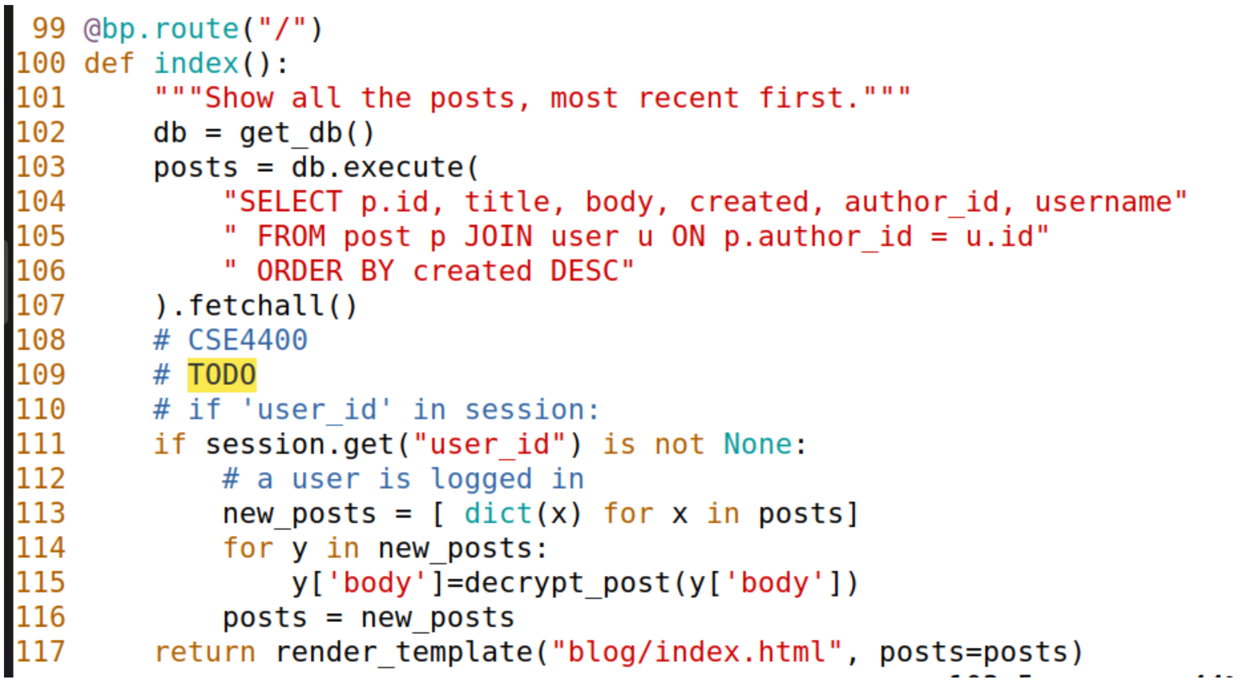
```python
def      encrypt_post(body, encrypt):
    # encrypt a post if encrypt is ture.
    # TODO
    #d will be out final return, just add one more thing to it.
    d = {'encrypt' : encrypt}
    if encrypt:
        data = str.encode(body)
        aesccm = AESCCM(current_app.config['SECRET_KEY'])
        nonce = os.urandom(13)
        d['nonce'] = nonce.hex()
        aad=b''
        ct = aesccm.encrypt(nonce, data, aad)
        d['aad'] = aad.hex()
        hexstr = ct.hex()
        d['body'] = hexstr
    else:  # not encrypted
        d['body'] = body
    return d
```

- My updated encrypt post



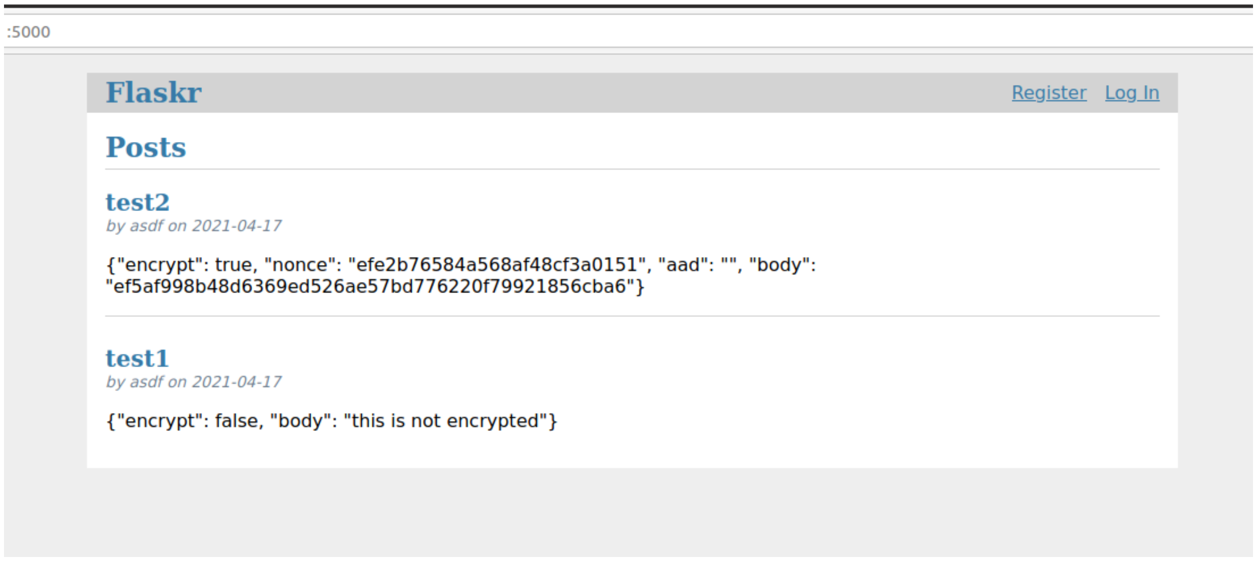
```python
# return a dictionary, which must have body and encrypt
def      decrypt_post(body):
    # decrypt a post or remove headers if not encrypted
    # TODO
    # try to decode as a JSON string
    try:
        body_dict = json.loads(body)
    except (json.JSONDecodeError):
        body_dict = {'body':  body, 'encrypt': 0}

    if 'encrypt' in body_dict and body_dict['encrypt'] == True:
        b = bytes.fromhex(body_dict['body'])
        aesccm = AESCCM(current_app.config['SECRET_KEY'])
        pt = aesccm.decrypt(bytes.fromhex(body_dict['nonce']), b,bytes.fromhex(body_dict['aad']))
        body_dict['body'] = pt.decode()

    if 'body' not in body_dict:
        body_dict['body'] = 'Body is not set'
    return body_dict
```

- My updated decrypt_post



```python
@bp.route("/")
def index():
    """Show all the posts, most recent first."""
    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()
    # CSE4400
    # TODO
    # if 'user_id' in session:
    if session.get("user_id") is not None:
        # a user is logged in
        new_posts = [ dict(x) for x in posts]
        for y in new_posts:
            y['body']=decrypt_post(y['body'])
        posts = new_posts
    return render_template("blog/index.html", posts=posts)
```

- My updated index function



:5000

Flaskr Register Log In

Posts

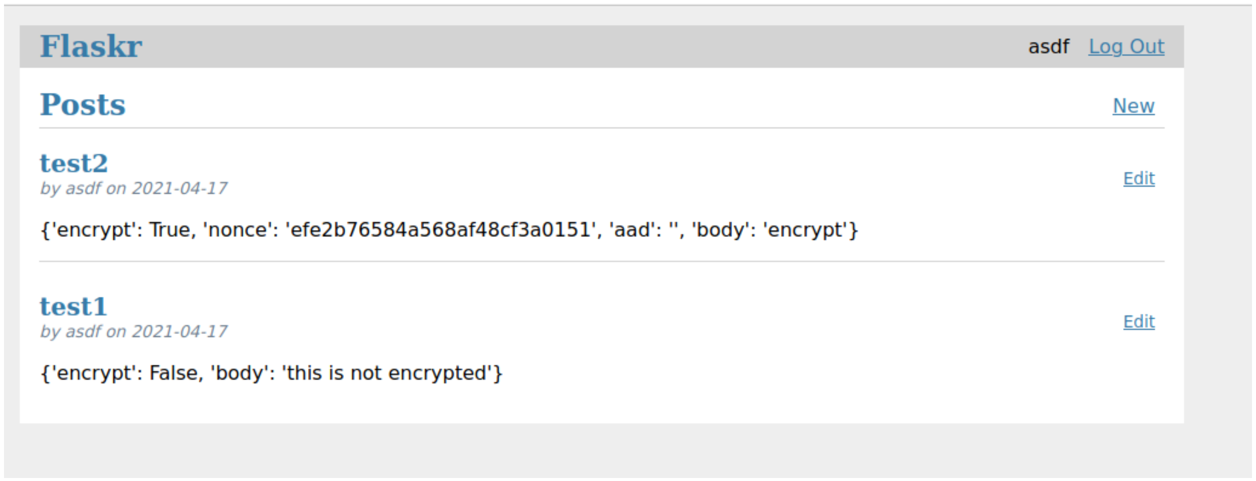
test2
by asdf on 2021-04-17

{"encrypt": true, "nonce": "efe2b76584a568af48cf3a0151", "aad": "", "body": "ef5af998b48d6369ed526ae57bd776220f79921856cba6"}

test1
by asdf on 2021-04-17

{"encrypt": false, "body": "this is not encrypted"}

- Here I have two posts on the site. You can see that one is encrypted, while the other is not. Since the current user is not logged in, the encrypted one shows its ciphertext under the “body” section.



test2
by asdf on 2021-04-17

{'encrypt': True, 'nonce': 'efe2b76584a568af48cf3a0151', 'aad': '', 'body': 'encrypt'}

test1
by asdf on 2021-04-17

{'encrypt': False, 'body': 'this is not encrypted'}

- Now that a user (named 'asdf') has logged in, the post 'test2' now has its plaintext displayed instead of its ciphertext.