Test unitaires dans flask

Creer un dossier tests/ a la racine de l'app.

Formate de nommage des fichiers de test : test_*.py

client()

Creer un fichier `test_flaskr.py` qui va contenir la customisation des fixture pytest pour gerer les tests.

```
import os
import tempfile

import pytest

from flaskr import flaskr


@pytest.fixture
def client():
    db_fd, flaskr.app.config['DATABASE'] = tempfile.mkstemp()
    flaskr.app.config['TESTING'] = True
    client = flaskr.app.test_client()

    with flaskr.app.app_context():
        flaskr.init_db()

    yield client

    os.close(db_fd)
    os.unlink(flaskr.app.config['DATABASE'])
```

Le fixture client est apelle par chaque test> Client fournit une interface pour envoyer des request a l application. Client gere les cookies pour nous.

On met la variable [“TESTING”] a True pendant les tests, cela desactive l'interception des erreurs pendant le traimtement des requetes pour avoir des rapports d erreur plus complet.

Comme Sqlite3 est une base de donnees fichier, on peut facilement utiliser un tempfile pour creer une base temporaire et travailler avec. Mkstemp() crée un fichier temporaire avec un nom aleatoire, on doit juste garder l'acces a db_fd pour pouvoir fermer le fichier a la fin.

Apres le test on utilise unlink pour supprimer le fichier temporaire dont le path est stocke dans flaskr.app.config['DATABASE']

On lance les tests avec

```
pytest
```

Premier test

On va tester si l'application retourne “`No entries here so far`” si on `GET “/”`

Pour que **pytest** trouve tout seul le test on commence le nom de la fonction par **test** .

```
def test_empty_db(client):
    """Start with a blank database."""

    rv = client.get('/')
    assert b'No entries here so far' in rv.data
```

En utilisant `client.get` on peut envoyer une requete HTTP de type GET sur le PATH en argument. La valeur de retour est une response_class object. On utilise data pour verifier le contenu de la réponse. Dans notre exemple on s'assure d'avoir `No entries here so far` comme retour.

On relance en et on voit que le test passe.

```
$ pytest -v

================ test session starts ================
rootdir: ./flask/examples/flaskr, inifile: setup.cfg
collected 1 items

tests/test_flaskr.py::test_empty_db PASSED

============= 1 passed in 0.10 seconds ==============
```

Connexion et Deconnexion

La plupart des application s'utilise seulement en mode authentifiée. On a besoin de connecter notre `test client` dans l'application. Pour faire cela, on passe des arguments a la page de login avec les champs attendus par le formulaire (`username` et `password`). La page de connexion nous rediriges une fois la connexion établie, donc on utilise 'foloow_redirects' pour que le `test client` suive.

Ajouter la gestion du login dans `test_flaskr.py`

```
def login(client, username, password):
    return client.post('/login', data=dict(
        username=username,
        password=password
    ), follow_redirects=True)


def logout(client):
    return client.get('/logout', follow_redirects=True)
```

Maintenant on peut facilement tester si la connexion et login marche et echoue en cas d'identifiants invalides.

Ajoutez cette nouvelle fonction de test.

```
def test_login_logout(client):
    """Make sure login and logout works."""

    rv = login(client, flaskr.app.config['USERNAME'],
flaskr.app.config['PASSWORD'])
    assert b'You were logged in' in rv.data

    rv = logout(client)
    assert b'You were logged out' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'] + 'x',
flaskr.app.config['PASSWORD'])
    assert b'Invalid username' in rv.data

    rv = login(client, flaskr.app.config['USERNAME'],
flaskr.app.config['PASSWORD'] + 'x')
    assert b'Invalid password' in rv.data
```

Test Adding Messages

On regarde si on peut ajouter poster un message sur la page `/add` . On verifie que le html est bien present dans le contenu mais pas dans le titre.

```
def test_messages(client):
    """Test that messages work."""

    login(client, flaskr.app.config['USERNAME'],
flaskr.app.config['PASSWORD'])
    rv = client.post('/add', data=dict(
```

```
        title='<Hello>',
        text='<strong>HTML</strong> allowed here'
    ), follow_redirects=True)
    assert b'No entries here so far' not in rv.data
    assert b'&lt;Hello&gt;' in rv.data
    assert b'<strong>HTML</strong> allowed here' in rv.data
```

Autres Astuces

On peut aussi utiliser test_request_context avec with pour temporairement avoir accès a un `request context`. Ce qui permet d'avoir acces a request , g] et [http://flask.pocoo.org/docs/1.0/api/#flask.session|session dans notre fonction de test.

Voici un exemple complet :

```
import flask

app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    assert flask.request.path == '/'
    assert flask.request.args['name'] == 'Peter'
```

Tout autre objet concerne par le `context` peu être utilise de cette façon.

Si vous voulez tester votre application avec des configuration différente, ce n'est pas la bonne méthode. Preferrez l'utilisation des Applicatio]n factorie.

Attention, si vous utilisez un `test request context` les function before_request et after_request ne sont pas appelle automatiquement. Par contre les fonctions teardown_request sont exécutes quand l'exécution quitte le bloc `with` . Si vous voulez appeler before_request , vous devez appeler preprocess_request vous même.

```
app = flask.Flask(__name__)

with app.test_request_context('/?name=Peter'):
    app.preprocess_request()
    ...
```

Cela peu etre necessaire si votre application etablie une connexion a une bdd ou autre, cela depend du code de votre application.

Si vous souhaitez appeler une after_request() vous devez le faire via l'appelle de preprocess_request() auquel vous devez passer la `Response Object`

```
with app.test_request_context('/?name=Peter'):
    resp = Response('...')
    resp = app.process_response(resp)
    ...
```

Ce cas si est en général inutile car on peut directement utiliser un test client.

Simuler des Ressources ou un Context

Un tache tres commune consiste a stocker des informations d'utilisateur ou de connexion bdd dans l'application context ou le 'flask.g' . En general on charge l'objet au premier appel et apres on le supprime dans le 'teardozn'. Imaginez un code similaire pour recuperer l'utilisateur actuel :

```
def get_user():
    user = getattr(g, 'user', None)
    if user is None:
        user = fetch_current_user_from_database()
        g.user = user
    return user
```

Pour un test il serait cool de pouvoir remplacer l'utilisateur sans avoir a changer le code. On peut le faire trankilou en interceptant le signal avec flask.appcontext_pushed signal

Creation de l interception

```
from contextlib import contextmanager
from flask import appcontext_pushed, g

@contextmanager
def user_set(app, user):
    def handler(sender, **kwargs):
        g.user = user
    with appcontext_pushed.connected_to(handler, app):
        yield
```

Utilisation de l interception via `with user_set`(app, my_user)

```
from flask import json, jsonify

@app.route('/users/me')
def users_me():
```

```
    return jsonify(username=g.user.username)

with user_set(app, my_user):
    with app.test_client() as c:
        resp = c.get('/users/me')
        data = json.loads(resp.data)
        self.assert_equal(data['username'], my_user.username)
```

Conserver le Context via With et Test_client

Depuis Flask 0.4 il est possible d'utiliser `test_client()` avec un `with` block !

```
app = flask.Flask(__name__)

with app.test_client() as c:
    rv = c.get('/?tequila=42')
```

Si on utilise `test_client` sans le `with`, l'assert va echouer car le request n'est plus accessible. Si on utilise pas with, le code serait exécuté en dehors de la requête actuelle, après résolution par test_client().

Accéder et Modifier les Sessions

Si vous voulez juste vous assurer que la session contient certaines `keys` ou variables vous pouvez juste utiliser `with` avec `test_client()` pour acceder a `flask.session`.

```
with app.test_client() as c:
    rv = c.get('/')
    assert flask.session['foo'] == 42
```

Il est impossible de modifier ou accéder a la session avant que la requête soit exécutée. Depuis Flask 0.8 on peut utiliser une “session transaction” qui simule une session avec le context du `test_client` et la modifie. Au final la session est conservee. Cela fonctione sans utiliser la session backend.

```
with app.test_client() as c:
    with c.session_transaction() as sess:
        sess['a_key'] = 'a value'
```

```
    # once this is reached the session was stored
```

Dans cet exemple, nous avons utilise l'objet `sess` au lieu de `flask.session` proxy. Toutefois `sess` fournira les memes interfaces.

Tester une API JSON

Flask supporte bien le JSON.

Voici comment faire des request en JSON et vérifier des réponses JSON.

```
from flask import request, jsonify

@app.route('/api/auth')
def auth():
    json_data = request.get_json()
    email = json_data['email']
    password = json_data['password']
    return jsonify(token=generate_token(email, password))

with app.test_client() as c:
    rv = c.post('/api/auth', json={
        'username': 'flask', 'password': 'secret'
    })
    json_data = rv.get_json()
    assert verify_token(email, json_data['token'])
```

On passe l'argument `json` a `c.post` il permet d'envoyer la data en JSON-serialise avec le content-type `application/json` . On peut recuperer le JSON depuis la reponse via `get_json`

Thxer -

2018/10/15 08:33