

# Couverture des tests

L'écriture de tests unitaires pour votre application vous permet de vérifier que le code que vous avez écrit fonctionne comme vous l'attendez. Flask fournit un client de test qui simule les requêtes adressées à l'application et renvoie les données de réponse.

Vous devez tester votre code autant que possible. Le code des fonctions ne s'exécute que lorsque la fonction est appelée, et le code des embranchements, comme les blocs `if`, ne s'exécute que lorsque la condition est remplie. Vous devez vous assurer que chaque fonction est testée avec des données qui couvrent chaque branche.

Plus vous vous rapprochez d'une couverture de 100 %, plus vous pouvez être sûr qu'un changement ne modifiera pas de manière inattendue d'autres comportements. Cependant, une couverture à 100 % ne garantit pas que votre application ne comporte pas de bogues. En particulier, elle ne permet pas de tester la manière dont l'utilisateur interagit avec l'application dans le navigateur. Malgré cela, la couverture des tests est un outil important à utiliser pendant le développement.

---

## Note:

Ce point est introduit tardivement dans le tutoriel, mais dans vos futurs projets, vous devriez tester au fur et à mesure que vous développez.

---

Vous utiliserez [pytest](#) et [coverage](#) pour tester et mesurer votre code. Installez-les tous les deux :

```
$ pip install pytest coverage
```

## Installation et *fixtures*

Le code de test est situé dans le répertoire `tests`. Ce répertoire se trouve à côté du paquet `flaskr`, pas à l'intérieur. Le fichier `tests/conftest.py` contient des fonctions de configuration appelées *fixtures* que chaque test utilisera. Les tests sont dans des modules Python qui commencent par `test_`, et chaque fonction de test dans ces modules commence aussi par `test_`.

Chaque test créera un nouveau fichier de base de données temporaire et alimentera certaines données qui seront utilisées dans les tests. Écrivez un fichier SQL pour insérer ces données.

```
tests/data.sql
```

```
INSERT INTO user (username, password)  
VALUES
```

```
    ('test', 'pbkdf2:sha256:50000$TCI4GzcX$0de171a4f4dac32e3364c  v: latest ▾  
    ('other', 'pbkdf2:sha256:50000$kJPKsz6N$d2d4784f1b030a9761f5ccaeeaca4.
```

```
INSERT INTO post (title, body, author_id, created)
VALUES
  ('test title', 'test' || x'0a' || 'body', 1, '2018-01-01 00:00:00');
```

La *fixture* `app` appelle la fabrique et passe `test_config` pour configurer l'application et la base de données pour les tests au lieu d'utiliser votre configuration de développement locale.

tests/conftest.py

```
import os
import tempfile

import pytest
from flaskr import create_app
from flaskr.db import get_db, init_db

with open(os.path.join(os.path.dirname(__file__), 'data.sql'), 'rb') as
    _data_sql = f.read().decode('utf8')

@pytest.fixture
def app():
    db_fd, db_path = tempfile.mkstemp()

    app = create_app({
        'TESTING': True,
        'DATABASE': db_path,
    })

    with app.app_context():
        init_db()
        get_db().executescript(_data_sql)

    yield app

    os.close(db_fd)
    os.unlink(db_path)

@pytest.fixture
def client(app):
    return app.test_client()

@pytest.fixture
def runner(app):
    return app.test_cli_runner()
```

 v: latest ▾

`tempfile.mkstemp()` crée et ouvre un fichier temporaire, renvoyant le descripteur de fichier et le chemin d'accès à celui-ci. Le chemin d'accès à la `DATABASE` est remplacé par le chemin

d'accès temporaire au lieu du dossier de l'instance. Après avoir défini le chemin, les tables de la base de données sont créées et les données du test sont insérées. Une fois le test terminé, le fichier temporaire est fermé et supprimé.

**TESTING** indique à Flask que l'application est en mode test. Flask modifie certains comportements internes pour faciliter les tests, et d'autres extensions peuvent également utiliser ce *flag* pour faciliter leurs tests.

La *fixture* `client` appelle `app.test_client()` avec l'objet application créé par la *fixture* `app`. Les tests utiliseront le client pour faire des requêtes à l'application sans démarrer le serveur.

La *fixture* `runner` est similaire à `client`. `app.test_cli_runner()` crée un runner qui peut appeler les commandes Click enregistrées avec l'application.

Pytest utilise les *fixtures* en faisant correspondre leurs noms de fonctions avec les noms des arguments dans les fonctions de test. Par exemple, la fonction `test_hello` que vous allez écrire ensuite prend un argument `client`. Pytest fait correspondre cet argument avec la fonction *fixture* `client`, l'appelle et passe la valeur retournée à la fonction de test.

## Fabrique

Il n'y a pas grand chose à tester sur l'usine elle-même. La plupart du code sera déjà exécuté pour chaque test, donc si quelque chose échoue, les autres tests le remarqueront.


Le seul comportement qui peut changer est le passage du test config. Si la configuration n'est pas passée, il doit y avoir une configuration par défaut, sinon la configuration doit être remplacée.

```
tests/test_factory.py
```

```
from flaskr import create_app

def test_config():
    assert not create_app().testing
    assert create_app({'TESTING': True}).testing

def test_hello(client):
    response = client.get('/hello')
    assert response.data == b'Hello, World!'
```

Vous avez ajouté la route `hello` comme exemple lors de l'écriture de la fabrique au début du tutoriel. Il retourne « Hello, World ! », donc le test vérifie que les données de la réponse  `v: latest` ▼ correspondent.

# Base de données

Dans un contexte d'application, `get_db` doit retourner la même connexion à chaque fois qu'il est appelé. Après le contexte, la connexion doit être fermée.

tests/test\_db.py

```
import sqlite3

import pytest
from flaskr.db import get_db

def test_get_close_db(app):
    with app.app_context():
        db = get_db()
        assert db is get_db()

    with pytest.raises(sqlite3.ProgrammingError) as e:
        db.execute('SELECT 1')

    assert 'closed' in str(e.value)
```

La commande `init-db` devrait appeler la fonction `init_db` et produire un message.

tests/test\_db.py


```
def test_init_db_command(runner, monkeypatch):
    class Recorder(object):
        called = False

    def fake_init_db():
        Recorder.called = True

    monkeypatch.setattr('flaskr.db.init_db', fake_init_db)
    result = runner.invoke(args=['init-db'])
    assert 'Initialized' in result.output
    assert Recorder.called
```

Ce test utilise la *fixture* `monkeypatch` de Pytest pour remplacer la fonction `init_db` par une fonction qui enregistre qu'elle a été appelée. La *fixture* `runner` que vous avez écrite ci-dessus est utilisée pour appeler la commande `init-db` par son nom.

# Authentification

Pour la plupart des vues, un utilisateur doit être connecté. La façon la plus simple d'  `v: latest` dans les tests est de faire une requête POST vers la vue `login` avec le client. Plutôt que d'écrire

cela à chaque fois, vous pouvez écrire une classe avec des méthodes pour le faire, et utiliser une *fixture* pour lui passer le client pour chaque test.

tests/conftest.py

```
class AuthActions(object):
    def __init__(self, client):
        self._client = client

    def login(self, username='test', password='test'):
        return self._client.post(
            '/auth/login',
            data={'username': username, 'password': password}
        )

    def logout(self):
        return self._client.get('/auth/logout')
```

```
@pytest.fixture
def auth(client):
    return AuthActions(client)
```

Avec la *fixture* `auth`, vous pouvez appeler `auth.login()` dans un test pour vous connecter en tant qu'utilisateur `test`, qui a été inséré comme partie des données de test dans l'interface `app`.

La vue `register` doit être rendue avec succès sur `GET`. Sur `POST` avec des données de formulaire valides, elle devrait rediriger vers l'URL de connexion et les données de l'utilisateur devraient être dans la base de données. Les données non valides doivent afficher des messages d'erreur.

tests/test\_auth.py

```
import pytest
from flask import g, session
from flaskr.db import get_db

def test_register(client, app):
    assert client.get('/auth/register').status_code == 200
    response = client.post(
        '/auth/register', data={'username': 'a', 'password': 'a'}
    )
    assert 'http://localhost/auth/login' == response.headers['Location']

    with app.app_context():
        assert get_db().execute(
            "select * from user where username = 'a'",
        ).fetchone() is not None
```

 v: latest ▾

```
@pytest.mark.parametrize(('username', 'password', 'message'), (
    ('', '', b'Username is required.'),
    ('a', '', b'Password is required.'),
    ('test', 'test', b'already registered'),
))
def test_register_validate_input(client, username, password, message):
    response = client.post(
        '/auth/register',
        data={'username': username, 'password': password}
    )
    assert message in response.data
```

`client.get()` fait une requête GET et renvoie l'objet `Response` retourné par Flask. De même, `client.post()` fait une requête POST, convertissant le dictionnaire `data` en données de formulaire.

Pour tester que la page s'affiche correctement, une simple requête est effectuée et on vérifie si elle renvoie un code 200 OK `status_code``. Si le rendu échoue, Flask renvoie un code 500 Internal Server Error.

**headers** aura un en-tête `Location` avec l'URL de connexion lorsque la vue d'enregistrement redirige vers la vue de connexion.

**data** contient le corps de la réponse sous forme d'octets. Si vous vous attendez à ce qu'une certaine valeur soit rendue sur la page, vérifiez qu'elle se trouve dans `data`. Les octets doivent être comparés à des octets. Si vous voulez comparer du texte, utilisez `get_data(as_text=True)` à la place.

`pytest.mark.parametrize` indique à Pytest d'exécuter la même fonction de test avec différents arguments. Vous l'utilisez ici pour tester différentes entrées invalides et différents messages d'erreur sans écrire le même code trois fois.

Les tests pour la vue `login` sont très similaires à ceux de `register`. Plutôt que de tester les données dans la base de données, `session` devrait avoir `user_id` défini après la connexion.

tests/test\_auth.py

```
def test_login(client, auth):
    assert client.get('/auth/login').status_code == 200
    response = auth.login()
    assert response.headers['Location'] == 'http://localhost/'

    with client:
        client.get('/')
        assert session['user_id'] == 1
        assert g.user['username'] == 'test'
```

 v: latest ▾

```
@pytest.mark.parametrize(('username', 'password', 'message'), (
```

```

    ('a', 'test', b'Incorrect username.'),
    ('test', 'a', b'Incorrect password.'),
))
def test_login_validate_input(auth, username, password, message):
    response = auth.login(username, password)
    assert message in response.data

```

L'utilisation de `client` dans un bloc `with` permet d'accéder à des variables contextuelles telles que `session` après le retour de la réponse. Normalement, l'accès à `session` en dehors d'une requête soulève une erreur.

Le test `logout` est le contraire de `login`. `session` ne doit pas contenir `user_id` après la déconnexion.

tests/test\_auth.py

```

def test_logout(client, auth):
    auth.login()

    with client:
        auth.logout()
        assert 'user_id' not in session

```

## Blog

Toutes les vues du blog utilisent la *fixture* `auth` que vous avez écrit plus tôt. Appelez `auth.login()` et les requêtes suivantes du client seront connectées en tant qu'utilisateur `test`.

La vue `index` doit afficher des informations sur le message qui a été ajouté avec les données de test. Lorsque l'on est connecté en tant qu'auteur, il doit y avoir un lien pour modifier le message.

Vous pouvez également tester d'autres comportements d'authentification en testant la vue `index`. Lorsque vous n'êtes pas connecté, chaque page affiche des liens pour se connecter ou s'enregistrer. Lorsqu'on est connecté, il y a un lien pour se déconnecter.

tests/test\_blog.py

```

import pytest
from flaskr.db import get_db

def test_index(client, auth):
    response = client.get('/')
    assert b"Log In" in response.data
    assert b"Register" in response.data

    auth.login()

```

 v: latest ▼

```

response = client.get('/')
assert b'Log Out' in response.data
assert b'test title' in response.data
assert b'by test on 2018-01-01' in response.data
assert b'test\nbody' in response.data
assert b'href="/1/update"' in response.data

```

Un utilisateur doit être connecté pour accéder aux vues `create`, `update` et `delete`. L'utilisateur connecté doit être l'auteur du message pour accéder à `update` et `delete`, sinon un état 403 Forbidden est renvoyé. Si un message avec l'id donné n'existe pas, `update` et `delete` doivent retourner 404 Not Found.

tests/test\_blog.py

```



@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
    '/1/delete',
))
def test_login_required(client, path):
    response = client.post(path)
    assert response.headers['Location'] == 'http://localhost/auth/login'

def test_author_required(app, client, auth):
    # change the post author to another user
    with app.app_context():
        db = get_db()
        db.execute('UPDATE post SET author_id = 2 WHERE id = 1')
        db.commit()

    auth.login()
    # current user can't modify other user's post
    assert client.post('/1/update').status_code == 403
    assert client.post('/1/delete').status_code == 403
    # current user doesn't see edit link
    assert b'href="/1/update"' not in client.get('/').data

@pytest.mark.parametrize('path', (
    '/2/update',
    '/2/delete',
))
def test_exists_required(client, auth, path):
    auth.login()
    assert client.post(path).status_code == 404

```

Les vues `create` et `update` doivent afficher et renvoyer un état 200 OK pour une  **v: latest**  Lorsque des données valides sont envoyées dans une requête POST, la vue `create` doit insérer les nouvelles données du message dans la base de données et la vue `update` doit modifier les



données existantes. Les deux pages doivent afficher un message d'erreur en cas de données invalides.

tests/test\_blog.py

```
def test_create(client, auth, app):
    auth.login()
    assert client.get('/create').status_code == 200
    client.post('/create', data={'title': 'created', 'body': ''})

    with app.app_context():
        db = get_db()
        count = db.execute('SELECT COUNT(id) FROM post').fetchone()[0]
        assert count == 2

def test_update(client, auth, app):
    auth.login()
    assert client.get('/1/update').status_code == 200
    client.post('/1/update', data={'title': 'updated', 'body': ''})

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post['title'] == 'updated'

@pytest.mark.parametrize('path', (
    '/create',
    '/1/update',
))
def test_create_update_validate(client, auth, path):
    auth.login()
    response = client.post(path, data={'title': '', 'body': ''})
    assert b'Title is required.' in response.data
```

La vue `delete` doit rediriger vers l'URL de l'index et le message ne doit plus exister dans la base de données.

tests/test\_blog.py

```
def test_delete(client, auth, app):
    auth.login()
    response = client.post('/1/delete')
    assert response.headers['Location'] == 'http://localhost/'

    with app.app_context():
        db = get_db()
        post = db.execute('SELECT * FROM post WHERE id = 1').fetchone()
        assert post is None
```

# Exécution des tests

Une configuration supplémentaire, qui n'est pas nécessaire mais qui rend l'exécution des tests de couverture moins verbeuse, peut être ajoutée au fichier `setup.cfg` du projet.

```
setup.cfg
```

```
[tool:pytest]
testpaths = tests
```

```
[coverage:run]
branch = True
source =
    flaskr
```


Pour exécuter les tests, utilisez la commande `pytest`. Elle trouvera et exécutera toutes les fonctions de test que vous avez écrites.

```
$ pytest
```

```
===== test session starts =====
platform linux -- Python 3.6.4, pytest-3.5.0, py-1.5.3, pluggy-0.6.0
rootdir: /home/user/Projects/flask-tutorial, inifile: setup.cfg
collected 23 items

tests/test_auth.py ..... [ 34%]
tests/test_blog.py ..... [ 86%]
tests/test_db.py .. [ 95%]
tests/test_factory.py .. [100%]

===== 24 passed in 0.64 seconds =====
```



Si un test échoue, `pytest` affichera l'erreur qui a été levée. Vous pouvez lancer `pytest -v` pour obtenir une liste de chaque fonction de test plutôt que des points.

Pour mesurer la couverture de code de vos tests, utilisez la commande `coverage` pour lancer `pytest` au lieu de le lancer directement.

```
$ coverage run -m pytest
```

Vous pouvez soit afficher un simple rapport de couverture dans le terminal :

```
$ coverage report
```

Name	Stmts	Miss	Branch	BrPart	Cover
flaskr/__init__.py	21	0	2	0	100%
flaskr/auth.py	54	0	22	0	100%

 v: latest ▼

flaskr/blog.py	54	0	16	0	100%
flaskr/db.py	24	0	4	0	100%
-----					
TOTAL	153	0	44	0	100%

Un rapport HTML vous permet de voir quelles lignes ont été couvertes dans chaque fichier :

```
$ coverage html
```

Cela génère des fichiers dans le répertoire `htmlcov`. Ouvrez `htmlcov/index.html` dans votre navigateur pour voir le rapport.

Passez à [Déployer en production.](#)