

 eBook Gratuit

APPRENEZ

Flask

eBook gratuit non affilié créé à partir des
contributeurs de Stack Overflow.

#flask

Table des matières

À propos.....	1
Chapitre 1: Commencer avec Flask.....	2
Remarques.....	2
Versions.....	2
Exemples.....	2
Installation - Stable.....	2
Bonjour le monde.....	3
Installation - Dernières.....	3
Installation - Développement.....	3
sphinx.....	3
py.test.....	4
tox.....	4
Chapitre 2: Accès aux données de demande.....	5
Introduction.....	5
Exemples.....	5
Accéder à la chaîne de requête.....	5
Forme combinée et chaîne de requête.....	5
Accéder aux champs de formulaire.....	6
Chapitre 3: Autorisation et authentification.....	7
Exemples.....	7
Utilisation de l'extension flask-login.....	7
Idée générale.....	7
Créez un LoginManager.....	7
Spécifiez un rappel utilisé pour le chargement des utilisateurs.....	7
Une classe représentant votre utilisateur.....	8
Connecter les utilisateurs dans.....	9
Je me suis connecté à un utilisateur, et maintenant?.....	9
Déconnexion des utilisateurs.....	10
Que se passe-t-il si un utilisateur n'est pas connecté et que current_user objet current_u.....	10

Quelle prochaine	11
Délai de la session de connexion.....	11
Chapitre 4: Déploiement d'une application Flask à l'aide du serveur Web uWSGI avec Nginx	13
Exemples.....	13
Utiliser uWSGI pour exécuter une application de flacon.....	13
Installer nginx et le configurer pour uWSGI.....	14
Activer le streaming depuis flask.....	15
Configurer le modèle de chaudière Flask Application, uWGSI, Nginx - Configurations de serv.....	16
Chapitre 5: Essai	22
Exemples.....	22
Test de notre application Hello World.....	22
introduction.....	22
Définir le test.....	22
Lancer le test.....	23
Test d'une API JSON implémentée dans Flask.....	23
Test de cette API avec pytest.....	23
Accéder à et manipuler des variables de session dans vos tests en utilisant Flask-Testing.....	24
Chapitre 6: Fichiers statiques	27
Exemples.....	27
Utiliser des fichiers statiques.....	27
Fichiers statiques en production (servis par le serveur web frontal).....	28
Chapitre 7: Filtres de modèle Jinja2 personnalisés	32
Syntaxe.....	32
Paramètres.....	32
Exemples.....	32
Formatage de la date et de l'heure dans un modèle Jinja2.....	32
Chapitre 8: Flacon-WTF	33
Introduction.....	33
Exemples.....	33
Un simple formulaire.....	33
Chapitre 9: Flask sur Apache avec mod_wsgi	34

Exemples.....	34
Wrapper d'application WSGI.....	34
Configuration des sites Apache pour WSGI.....	34
Chapitre 10: Flask-SQLAlchemy.....	36
Introduction.....	36
Exemples.....	36
Installation et exemple initial.....	36
Relations: un à plusieurs.....	36
Chapitre 11: Le routage.....	38
Exemples.....	38
Itinéraires de base.....	38
Catch-all route.....	39
Méthodes de routage et HTTP.....	40
Chapitre 12: Les bleus.....	41
Introduction.....	41
Exemples.....	41
Un exemple de base de plans de flacons.....	41
Chapitre 13: Les signaux.....	43
Remarques.....	43
Exemples.....	43
Connexion aux signaux.....	43
Signaux personnalisés.....	43
Chapitre 14: Message clignotant.....	45
Introduction.....	45
Syntaxe.....	45
Paramètres.....	45
Remarques.....	45
Exemples.....	45
Message simple clignotant.....	45
Clignotant Avec Catégories.....	46
Chapitre 15: Modèles de rendu.....	47

Syntaxe.....	47
Exemples.....	47
Usage render_template.....	47
Chapitre 16: Pagination.....	49
Exemples.....	49
Exemple de route de pagination avec Pagin-sqlalchemy Paginate.....	49
Rendre la pagination à Jinja.....	49
Chapitre 17: Réorienter.....	51
Syntaxe.....	51
Paramètres.....	51
Remarques.....	51
Exemples.....	51
Exemple simple.....	51
Transmission de données.....	51
Chapitre 18: Sessions.....	53
Remarques.....	53
Exemples.....	53
Utilisation de l'objet sessions dans une vue.....	53
Chapitre 19: Téléchargement de fichier.....	55
Syntaxe.....	55
Exemples.....	55
Téléchargement de fichiers.....	55
Formulaire HTML.....	55
Demandes Python.....	55
Enregistrer les téléchargements sur le serveur.....	56
Transmission de données à WTForms et Flask-WTF.....	56
TÉLÉCHARGER LE FICHER PARSE CSV EN TANT QUE LISTE DE DICTIONNAIRES EN FLACON SANS ÉCONOMI.....	57
Chapitre 20: Travailler avec JSON.....	59
Exemples.....	59
Renvoie une réponse JSON à partir de l'API Flask.....	59

Essayez-le avec une curl.....	59
Autres façons d'utiliser jsonify().....	59
Recevoir JSON à partir d'une requête HTTP.....	59
Essayez-le avec une curl.....	60
Chapitre 21: Vues basées sur la classe.....	61
Exemples.....	61
Exemple de base.....	61
Crédits.....	62

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [flask](#)

It is an unofficial and free Flask ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Flask.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec Flask

Remarques

Flask est un micro-framework d'application web Python construit sur la bibliothèque WSGI de **Werkzeug**. Flask peut être "micro", mais il est prêt pour une utilisation de production sur une variété de besoins.

Le «micro» dans le micro-cadre signifie que Flask vise à garder le noyau simple mais extensible. Flask ne prendra pas beaucoup de décisions pour vous, comme la base de données à utiliser, et les décisions prises sont faciles à modifier. Tout est à vous, pour que Flask puisse être tout ce dont vous avez besoin et rien de ce que vous n'avez pas.

La communauté prend en charge un riche écosystème d'extensions pour rendre votre application plus puissante et plus facile à développer. À mesure que votre projet se développe, vous êtes libre de prendre les décisions de conception appropriées à vos besoins.

Versions

Version	Nom de code	Date de sortie
0,12	Punsch	2016-12-21
0,11	Absinthe	2016-05-29
0,10	Limoncello	2013-06-13

Exemples

Installation - Stable

Utilisez pip pour installer Flask dans une virtualenv.

```
pip install flask
```

Instructions pas à pas pour créer une virtualenv pour votre projet:

```
mkdir project && cd project
python3 -m venv env
# or `virtualenv env` for Python 2
source env/bin/activate
pip install flask
```

N'utilisez jamais `sudo pip install` moins que vous ne compreniez exactement ce que vous faites.

Conservez votre projet dans un virtualenv local, n'installez pas sur le système Python, sauf si vous utilisez le gestionnaire de packages système.

Bonjour le monde

Créez `hello.py` :

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'
```

Puis lancez-le avec:

```
export FLASK_APP=hello.py
flask run
* Running on http://localhost:5000/
```

L'ajout du code ci-dessous permettra de l'exécuter directement avec `python hello.py`.

```
if __name__ == '__main__':
    app.run()
```

Installation - Dernières

Si vous souhaitez utiliser le dernier code, vous pouvez l'installer depuis le référentiel. Bien que vous ayez potentiellement de nouvelles fonctionnalités et corrections, seules les versions numérotées sont officiellement prises en charge.

```
pip install https://github.com/pallets/flask/tarball/master
```

Installation - Développement

Si vous souhaitez développer et contribuer au projet Flask, clonez le référentiel et installez le code en mode développement.

```
git clone ssh://github.com/pallets/flask
cd flask
python3 -m venv env
source env/bin/activate
pip install -e .
```

Il existe également des dépendances et des outils supplémentaires à prendre en compte.

sphinx

Utilisé pour créer la documentation.

```
pip install sphinx
cd docs
make html
firefox _build/html/index.html
```

py.test

Utilisé pour exécuter la suite de tests.

```
pip install pytest
py.test tests
```

tox

Utilisé pour exécuter la suite de tests avec plusieurs versions de Python.

```
pip install tox
tox
```

Notez que tox n'utilise que des interpréteurs déjà installés, donc si Python 3.3 n'est pas installé sur votre chemin, il ne sera pas testé.

Lire Commencer avec Flask en ligne: <https://riptutorial.com/fr/flask/topic/790/commencer-avec-flask>

Chapitre 2: Accès aux données de demande

Introduction

Lorsque vous travaillez avec une application Web, il est parfois important d'accéder aux données incluses dans la demande, au-delà de l'URL.

Dans Flask, ceci est stocké sous l'objet global **request**, auquel vous pouvez accéder dans votre code via la `from flask import request`.

Exemples

Accéder à la chaîne de requête

La chaîne de requête est la partie d'une requête suivant l'URL, précédée d'un `?` marque.

Exemple: `https://encrypted.google.com/search ?hl=en&q=stack%20overflow`

Pour cet exemple, nous créons un simple serveur Web d'écho qui renvoie tout ce qui lui est soumis via le champ `echo` dans `GET` requêtes `GET`.

Exemple: `localhost:5000/echo ?echo=echo+this+back+to+me`

Exemple de flacon :

```
from flask import Flask, request

app = Flask(import_name=__name__)

@app.route("/echo")
def echo():

    to_echo = request.args.get("echo", "")
    response = "{}".format(to_echo)

    return response

if __name__ == "__main__":
    app.run()
```

Forme combinée et chaîne de requête

Flask permet également d'accéder à un `CombinedMultiDict` qui donne accès aux attributs `request.form` et `request.args` sous une seule variable.

Cet exemple extrait des données d'un `name` champ de formulaire soumis avec le champ `echo` dans la chaîne de requête.

Exemple de flacon :

```

from flask import Flask, request

app = Flask(import_name=__name__)

@app.route("/echo", methods=["POST"])
def echo():

    name = request.values.get("name", "")
    to_echo = request.values.get("echo", "")

    response = "Hey there {}! You said {}".format(name, to_echo)

    return response

app.run()

```

Accéder aux champs de formulaire

Vous pouvez accéder aux données de formulaire soumises via une `POST` ou `PUT` dans Flask via l'attribut `request.form`.

```

from flask import Flask, request

app = Flask(import_name=__name__)

@app.route("/echo", methods=["POST"])
def echo():

    name = request.form.get("name", "")
    age = request.form.get("age", "")

    response = "Hey there {}! You said you are {} years old.".format(name, age)

    return response

app.run()

```

Lire Accès aux données de demande en ligne: <https://riptutorial.com/fr/flask/topic/8622/acces-aux-donnees-de-demande>

Chapitre 3: Autorisation et authentification

Exemples

Utilisation de l'extension flask-login

L'une des manières les plus simples d'implémenter un système d'autorisation consiste à utiliser l'extension [flask-login](#) . Le site Web du projet contient un quickstart détaillé et bien écrit, dont une version plus courte est disponible dans cet exemple.

Idée générale

L'extension expose un ensemble de fonctions utilisées pour:

- consigner les utilisateurs dans
- déconnecter les utilisateurs
- vérifier si un utilisateur est connecté ou non et trouver quel utilisateur est ce

Ce que ça ne fait pas et ce que vous devez faire vous-même:

- ne fournit pas un moyen de stocker les utilisateurs, par exemple dans la base de données
- ne fournit pas un moyen de vérifier les informations d'identification de l'utilisateur, par exemple le nom d'utilisateur et le mot de passe

Ci-dessous, il y a un ensemble minimal d'étapes nécessaires pour que tout fonctionne.

Je recommande de placer tous les codes associés à un auth dans un module ou un package distinct, par exemple `auth.py` De cette façon, vous pouvez créer les classes, objets ou fonctions personnalisés nécessaires séparément.

Créez un `LoginManager`

L'extension utilise une classe `LoginManager` qui doit être enregistrée sur votre objet d'application `Flask` .

```
from flask_login import LoginManager
login_manager = LoginManager()
login_manager.init_app(app) # app is a Flask object
```

Comme mentionné précédemment, `LoginManager` peut par exemple être une variable globale dans un fichier ou un package distinct. Ensuite, il peut être importé dans le fichier dans lequel l'objet `Flask` est créé ou dans la fonction fabrique de l'application et initialisé.

Spécifiez un rappel utilisé pour le chargement des utilisateurs

Un utilisateur sera normalement chargé à partir d'une base de données. Le rappel doit retourner un objet qui représente un utilisateur correspondant à l'ID fourni. Il doit retourner `None` si l'ID n'est pas valide.

```
@login_manager.user_loader
def load_user(user_id):
    return User.get(user_id) # Fetch the user from the database
```

Cela peut être fait directement sous la création de votre `LoginManager` .

Une classe représentant votre utilisateur

Comme mentionné, le rappel `user_loader` doit renvoyer un objet qui représente un utilisateur. Qu'est-ce que cela signifie exactement? Cet objet peut par exemple être une enveloppe autour des objets utilisateur stockés dans votre base de données ou simplement un modèle directement depuis votre base de données. Cet objet doit implémenter les méthodes et propriétés suivantes. Cela signifie que si le rappel renvoie votre modèle de base de données, vous devez vous assurer que les propriétés et méthodes mentionnées sont ajoutées à votre modèle.

- `is_authenticated`

Cette propriété doit retourner `True` si l'utilisateur est authentifié, c'est-à-dire qu'il a fourni des informations d'identification valides. Vous souhaitez vous assurer que les objets représentant vos utilisateurs renvoyés par le rappel `user_loader` renvoient `True` pour cette méthode.

- `is_active`

Cette propriété doit renvoyer `True` s'il s'agit d'un utilisateur actif. En plus d'être authentifiée, elle a également activé son compte, n'a pas été suspendue ou toute autre condition de rejet d'un compte par votre application. Les comptes inactifs ne peuvent pas se connecter. Si vous ne disposez pas d'un tel mécanisme, retournez `True` partir de cette méthode.

- `is_anonymous`

Cette propriété doit renvoyer `True` s'il s'agit d'un utilisateur anonyme. Cela signifie que votre objet utilisateur renvoyé par le rappel `user_loader` doit retourner `True` .

- `get_id()`

Cette méthode doit renvoyer un `Unicode` qui identifie de manière unique cet utilisateur et peut être utilisé pour charger l'utilisateur à partir du rappel `user_loader` . Notez que ce doit être un `Unicode` - si l'ID est nativement un `int` ou un autre type, vous devrez le convertir en `Unicode`. Si le rappel `user_loader` renvoie des objets de la base de données, cette méthode

retournera très probablement l'ID de base de données de cet utilisateur particulier. Bien entendu, le même identifiant doit provoquer le rappel de `user_loader` ultérieurement.

Si vous voulez vous faciliter les choses (** c'est en fait recommandé), vous pouvez hériter de `UserMixin` dans l'objet renvoyé par le rappel de `user_loader` (probablement un modèle de base de données). Vous pouvez voir comment ces méthodes et propriétés sont implémentées par défaut dans ce mixin [ici](#).

Connecter les utilisateurs dans

L'extension vous laisse la validation du nom d'utilisateur et du mot de passe saisis par l'utilisateur. En fait, l'extension ne se soucie pas si vous utilisez un nom d'utilisateur et mot de passe combo ou un autre mécanisme. Ceci est un exemple de journalisation des utilisateurs utilisant le nom d'utilisateur et le mot de passe.

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    # Here we use a class of some kind to represent and validate our
    # client-side form data. For example, WTForms is a library that will
    # handle this for us, and we use a custom LoginForm to validate.
    form = LoginForm()
    if form.validate_on_submit():
        # Login and validate the user.
        # user should be an instance of your `User` class
        login_user(user)

        flask.flash('Logged in successfully.')

        next = flask.request.args.get('next')
        # is_safe_url should check if the url is safe for redirects.
        # See http://flask.pocoo.org/snippets/62/ for an example.
        if not is_safe_url(next):
            return flask.abort(400)

        return flask.redirect(next or flask.url_for('index'))
    return flask.render_template('login.html', form=form)
```

En général, la journalisation des utilisateurs est effectuée en appelant `login_user` et en lui transmettant une instance d'un objet représentant votre utilisateur. Comme indiqué, cela se produit généralement après la récupération de l'utilisateur de la base de données et la validation de ses informations d'identification, mais l'objet utilisateur apparaît comme par magie dans cet exemple.

Je me suis connecté à un utilisateur, et maintenant?

L'objet renvoyé par le rappel `user_loader` est accessible de plusieurs manières.

- Dans les modèles:

L'extension l'injecte automatiquement sous le nom `current_user` aide d'un processeur de contexte de modèle. Pour désactiver ce comportement et utiliser votre processeur personnalisé, définissez `add_context_processor=False` dans votre constructeur `LoginManager` .

```
{% if current_user.is_authenticated %}
  Hi {{ current_user.name }}!
{% endif %}
```

- En code Python:

L'extension fournit un objet lié à la requête appelé `current_user` .

```
from flask_login import current_user

@app.route("/hello")
def hello():
    # Assuming that there is a name property on your user object
    # returned by the callback
    if current_user.is_authenticated:
        return 'Hello %s!' % current_user.name
    else:
        return 'You are not logged in!'
```

- Limiter l'accès rapidement en utilisant un décorateur Un décorateur `login_required` peut être utilisé pour limiter l'accès rapidement.

```
from flask_login import login_required

@app.route("/settings")
@login_required
def settings():
    pass
```

Déconnexion des utilisateurs

Les utilisateurs peuvent être déconnectés en appelant `logout_user()` . Il semble sûr de le faire même si l'utilisateur n'est pas connecté, de sorte que le décorateur `@login_required` peut très probablement être utilisé.

```
@app.route("/logout")
@login_required
def logout():
    logout_user()
    return redirect(somewhere)
```

Que se passe-t-il si un utilisateur n'est pas connecté et que `current_user` objet `current_user` ?

Par défaut, un `AnonymousUserMixin` est renvoyé:

- `is_active` et `is_authenticated` sont `False`
- `is_anonymous` est `True`
- `get_id()` renvoie `None`

Pour utiliser un objet différent pour les utilisateurs anonymes, fournissez un callable (une fonction de classe ou de fabrique) qui crée des utilisateurs anonymes sur votre `LoginManager` avec:

```
login_manager.anonymous_user = MyAnonymousUser
```

Quelle prochaine

Ceci conclut l'introduction de base à l'extension. Pour en savoir plus sur la configuration et les options supplémentaires, il est fortement recommandé de [lire le guide officiel](#).

Délai de la session de connexion

Sa bonne pratique est d'arrêter la session après un certain temps, vous pouvez le faire avec Flask-Login.

```
from flask import Flask, session
from datetime import timedelta
from flask_login import LoginManager, login_require, login_user, logout_user

# Create Flask application

app = Flask(__name__)

# Define Flask-login configuration

login_mgr = LoginManager(app)
login_mgr.login_view = 'login'
login_mgr.refresh_view = 'relogin'
login_mgr.needs_refresh_message = (u"Session timedout, please re-login")
login_mgr.needs_refresh_message_category = "info"

@app.before_request
def before_request():
    session.permanent = True
    app.permanent_session_lifetime = timedelta(minutes=5)
```

La durée de vie par défaut de la session est de 31 jours, l'utilisateur doit spécifier la vue d'actualisation de la connexion en cas de dépassement du délai d'attente.

```
app.permanent_session_lifetime = timedelta(minutes=5)
```

La ligne ci-dessus forcera l'utilisateur à se reconnecter toutes les 5 minutes.

Lire Autorisation et authentification en ligne: <https://riptutorial.com/fr/flask/topic/9053/autorisation->

Chapitre 4: Déploiement d'une application Flask à l'aide du serveur Web uWSGI avec Nginx

Exemples

Utiliser uWSGI pour exécuter une application de flaskon

Le serveur `werkzeug` intégré ne convient certainement pas pour exécuter des serveurs de production. La raison la plus évidente est le fait que le serveur `werkzeug` est à thread unique et ne peut donc traiter qu'une seule demande à la fois.

Pour cette raison, nous souhaitons utiliser le serveur uWSGI pour servir notre application à la place. Dans cet exemple, nous allons installer uWSGI et exécuter une application de test simple.

Installer uWSGI :

```
pip install uwsgi
```

C'est aussi simple que ça. Si vous n'êtes pas sûr de la version de python utilisée par votre pip, il est explicite:

```
python3 -m pip install uwsgi # for python3
python2 -m pip install uwsgi # for python2
```

Maintenant, créons une application de test simple:

app.py

```
from flask import Flask
from sys import version

app = Flask(__name__)

@app.route("/")
def index():
    return "Hello uWSGI from python version: <br>" + version

application = app
```

Dans flask, le nom conventionnel de l'application est `app` mais uWSGI recherche l' `application` par défaut. C'est pourquoi nous créons un alias pour notre application dans la dernière ligne.

Il est maintenant temps d'exécuter l'application:

```
uwsgi --wsgi-file app.py --http :5000
```

Vous devriez voir le message "Hello uWSGI ..." en pointant votre navigateur sur `localhost:5000`

Afin de ne pas taper la commande complète à chaque fois, nous allons créer un fichier `uwsgi.ini` pour stocker cette configuration:

uwsgi.ini

```
[uwsgi]
http = :9090
wsgi-file = app.py
single-interpreter = true
enable-threads = true
master = true
```

Les options `http` et `wsgi-file` sont les mêmes que dans la commande manuelle. Mais il y a trois autres options:

- `single-interpreter` : Il est recommandé de l'activer car cela pourrait interférer avec l'option suivante
- `enable-threads` : Cela doit être activé si vous utilisez des threads supplémentaires dans votre application. Nous ne les utilisons pas maintenant, mais maintenant nous n'avons plus à nous en préoccuper.
- `master` : le mode master doit être activé pour [diverses raisons](#)

Maintenant, nous pouvons exécuter l'application avec cette commande:

```
uwsgi --ini uwsgi.ini
```

Installer nginx et le configurer pour uWSGI

Maintenant, nous voulons installer nginx pour servir notre application.

```
sudo apt-get install nginx # on debian/ubuntu
```

Ensuite, nous créons une configuration pour notre site Web

```
cd /etc/nginx/site-available # go to the configuration for available sites
# create a file flaskconfig with your favourite editor
```

flaskconfig

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
    }
}
```

```
}
```

Cela dit à nginx d'écouter sur le port 80 (par défaut pour http) et de servir quelque chose au niveau du chemin racine (/). Là, nous demandons à nginx d'agir simplement en tant que proxy et de transmettre chaque requête à un socket appelé `flask.sock` situé dans `/tmp/`.

Activons le site:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/flaskconfig .
```

Vous souhaitez peut-être supprimer la configuration par défaut si elle est activée:

```
# inside /etc/sites-enabled
sudo rm default
```

Puis redémarrez nginx:

```
sudo service nginx restart
```

Pointez votre navigateur sur `localhost` et vous verrez une erreur: `502 Bad Gateway`.

Cela signifie que nginx est opérationnel et que le socket est absent. Alors créons cela.

Retournez dans votre fichier `uwsgi.ini` et ouvrez-le. Puis ajoutez ces lignes:

```
socket = /tmp/flask.sock
chmod-socket = 666
```

La première ligne indique à uwsgi de créer un socket à l'emplacement indiqué. Le socket sera utilisé pour recevoir des demandes et renvoyer les réponses. Dans la dernière ligne, nous permettons aux autres utilisateurs (y compris nginx) de pouvoir lire et écrire depuis cette socket.

Relancez uwsgi avec `uwsgi --ini uwsgi.ini`. Maintenant, pointez à nouveau votre navigateur vers `localhost` et vous verrez le message d'accueil "Hello uWSGI".

Notez que vous pouvez toujours voir la réponse sur `localhost:5000` car uWSGI sert maintenant l'application via http **et** le socket. Désactivons donc l'option http dans le fichier ini

```
http = :5000 # <-- remove this line and restart uwsgi
```

Maintenant, l'application n'est accessible qu'à partir de nginx (ou en lisant directement cette socket :)).

Activer le streaming depuis flask

Flask a cette fonctionnalité qui vous permet de diffuser des données depuis une vue en utilisant des générateurs.

Changeons le fichier `app.py`

- ajouter à `from flask import Response`
- ajout `from datetime import datetime`
- ajouter `from time import sleep`
- créer une nouvelle vue:

```
@app.route("/time/")
def time():
    def streamer():
        while True:
            yield "<p>{}</p>".format(datetime.now())
            sleep(1)

    return Response(streamer())
```

Ouvrez maintenant votre navigateur sur `localhost/time/` . Le site se chargera pour toujours car nginx attend que la réponse soit complète. Dans ce cas, la réponse ne sera jamais complète car elle enverra la date et l'heure actuelles pour toujours.

Pour éviter que nginx attende, nous devons ajouter une nouvelle ligne à la configuration.

Editez `/etc/nginx/sites-available/flaskconfig`

```
server {
    listen 80;
    server_name localhost;

    location / {
        include uwsgi_params;
        uwsgi_pass unix:///tmp/flask.sock;
        uwsgi_buffering off; # <-- this line is new
    }
}
```

La ligne `uwsgi_buffering off;` indique à nginx de ne pas attendre qu'une réponse soit complète.

Redémarrez nginx: `sudo service nginx restart` et regardez `localhost/time/` again.

Maintenant, vous verrez que chaque seconde une nouvelle ligne apparaît.

Configurer le modèle de chaudière Flask Application, uWSGI, Nginx - Configurations de serveur (par défaut, proxy et cache)

Ceci est un portage de configuration provenant du tutoriel de DigitalOcean sur la [façon de servir des applications Flask avec uWSGI et Nginx sur Ubuntu 14.04](#).

et quelques ressources utiles pour les serveurs nginx.

Application en flacon

Ce tutoriel suppose que vous utilisez Ubuntu.

1. Localisez `var/www/` folder.
2. Créez votre dossier d'application Web `mkdir myexample`
3. `cd myexample`

facultatif Vous souhaitez peut-être configurer un environnement virtuel pour le déploiement d'applications Web sur le serveur de production.

```
sudo pip install virtualenv
```

installer l'environnement virtuel.

```
virtualenv myexample
```

configurer l'environnement virtuel pour votre application.

```
source myprojectenv/bin/activate
```

pour activer votre environnement. Ici vous allez installer tous les paquets Python.

fin facultatif mais recommandé

Mettre en place le flacon et la passerelle uWSGI

Installez flask et la passerelle uWSGI:

```
pip install uwsgi flask
```

Exemple d'application flask dans `myexample.py`:

```
from flask import Flask
application = Flask(__name__)

@application.route("/")
def hello():
    return "<h1>Hello World</h1>"

if __name__ == "__main__":
    application.run(host='0.0.0.0')
```

Créer un fichier pour communiquer entre votre application Web et le serveur Web: interface de passerelle [https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface]

```
nano wsgi.py
```

importez ensuite votre module webapp et lancez-le à partir du point d'entrée de la passerelle.

```
from myexample import application

if __name__ == "__main__":
    application.run()
```

Pour tester uWSGI:

```
uwsgi --socket 0.0.0.0:8000 --protocol=http -w wsgi
```

Pour configurer uWSGI:

1. Créez un fichier de configuration `.ini`

```
nano myexample.ini
```

2. Configuration de base pour la passerelle uWSGI

```
# include header for using uwsgi
[uwsgi]
# point it to your python module wsgi.py
module = wsgi
# tell uWSGI to start a master node to serve requests
master = true
# spawn number of processes handling requests
processes = 5
# use a Unix socket to communicate with Nginx. Nginx will pass connections to uWSGI through a
socket, instead of using ports. This is preferable because Nginx and uWSGI stays on the same
machine.
socket = myexample.sock
# ensure file permission on socket to be readable and writable
chmod-socket = 660
# clean the socket when processes stop
vacuum = true
# use die-on-term to communicate with Ubuntu versions using Upstart initialisations: see:
# http://uwsgi-docs.readthedocs.io/en/latest/Upstart.html?highlight=die%20on%20term
die-on-term = true
```

facultatif si vous utilisez virtual env Vous pouvez `deactivate` votre environnement virtuel.

Configuration de Nginx Nous allons utiliser nginx comme:

1. serveur par défaut pour transmettre la requête au socket, en utilisant le protocole uwsgi
2. serveur proxy devant le serveur par défaut
3. serveur de cache pour mettre en cache les requêtes réussies (par exemple, vous pouvez vouloir mettre en cache les requêtes GET si votre application Web)

Recherchez le répertoire `sites-available` vos `sites-available` et créez un fichier de configuration pour votre application:

```
sudo nano /etc/nginx/sites-available/myexample
```

Ajouter le bloc suivant, dans les commentaires ce qu'il fait:

```
server {

    # setting up default server listening to port 80
    listen 8000 default_server;
    server_name myexample.com; #you can also use your IP
```



```

# specify charset encoding, optional
charset utf-8;

# specify root of your folder directory
root /var/www/myexample;

# specify locations for your web apps.
# here using /api endpoint as example
location /api {
    # include parameters of wsgi.py and pass them to socket
    include uwsgi_params;
    uwsgi_pass unix:/var/www/myexample/myexample.sock;
}

}

# Here you will specify caching zones that will be used by your virtual server
# Cache will be stored in /tmp/nginx folder
# ensure nginx have permissions to write and read there!
# See also:
# http://nginx.org/en/docs/http/nginx_http_proxy_module.html

proxy_cache_path /tmp/nginx levels=1:2 keys_zone=my_zone:10m inactive=60m;
proxy_cache_key "$scheme$request_method$host$request_uri";

# set up the virtual host!
server {
    listen 80 default_server;

    # Now www.example.com will listen to port 80 and pass request to http://example.com
    server_name www.example.com;

    # Why not caching responses

    location /api {
        # set up headers for caching
        add_header X-Proxy-Cache $upstream_cache_status;

        # use zone specified above
        proxy_cache my_zone;
        proxy_cache_use_stale updating;
        proxy_cache_lock on;

        # cache all responses ?
        # proxy_cache_valid 30d;

        # better cache only 200 responses :)
        proxy_cache_valid 200 30d;

        # ignore headers to make cache expire
        proxy_ignore_headers X-Accel-Expires Expires Cache-Control;

        # pass requests to default server on port 8000
        proxy_pass http://example.com:8000/api;
    }
}

```

Enfin, liez le fichier au répertoire `sites-enabled` avec les `sites-enabled`. Pour une explication des sites disponibles et activés, voir la réponse: [<http://serverfault.com/a/527644>]

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

Vous avez terminé avec nginx. Cependant, vous pourriez vouloir vérifier ce modèle de chaudière très précieux: [<https://github.com/h5bp/server-configs-nginx>]

Très utile pour les réglages fins.

Maintenant, testez Nginx:

```
sudo nginx -t
```

Lancez Nginx:

```
sudo service nginx restart
```

Automatiser Ubuntu pour démarrer uWSGI La dernière chose à faire est de faire qu'Ubuntu lance la passerelle wsgi en communiquant avec votre application, sinon vous devriez le faire manuellement.

1. Localisez le répertoire des scripts d'initialisation dans Ubuntu et créez un nouveau script:

```
sudo nano /etc/init/myexample.conf
```

2. Ajouter le bloc suivant, commentaires en ligne pour expliquer ce qu'il fait

```
# description for the purpose of this script
description "uWSGI server instance configured to serve myproject"

# Tell to start on system runtime 2, 3, 4, 5. Stop at any other level (0,1,6).
# Linux run levels: [http://www.debianadmin.com/debian-and-ubuntu-linux-run-levels.html]
start on runlevel [2345]
stop on runlevel [!2345]

# Set up permissions! "User" will be the username of your user account on ubuntu.
setuid user
# Allow www-data group to read and write from the socket file.
# www-data is normally the group Nginx and your web applications belong to.
# you may have all web application projects under /var/www/ that belongs to www-data
group
setgid www-data

# tell Ubuntu which environment to use.
# This is the path of your virtual environment: python will be in this path if you
installed virtualenv. Otherwise, use path of your python installation
env PATH=/var/www/myexample/myexample/bin
# then tell to Ubuntu to change and locate your web application directory
chdir /var/www/myexample
# finally execute initialisation script, that load your web app myexample.py
exec uwsgi --ini myexample.ini
```

Maintenant, vous pouvez activer votre script: `sudo start myexample`

[Lire Déploiement d'une application Flask à l'aide du serveur Web uWSGI avec Nginx en ligne:](#)

<https://riptutorial.com/fr/flask/topic/4637/deploiement-d-une-application-flask-a-l-aide-du-serveur-web-uwsgi-avec-nginx>

Chapitre 5: Essai

Exemples

Test de notre application Hello World

introduction

Dans cet exemple minimaliste, en utilisant `pytest` nous allons tester que notre application Hello World renvoie effectivement "Hello, World!" avec un code d'état HTTP OK de 200, lorsqu'il est frappé avec une requête GET sur l'URL /

D'abord, installons `pytest` dans notre virtualenv

```
pip install pytest
```

Et juste pour référence, ceci notre application du monde Bonjour:

```
# hello.py
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello, World!'
```

Définir le test

A côté de notre `hello.py`, nous définissons un module de test appelé `test_hello.py` qui sera découvert par `py.test`

```
# test_hello.py
from hello import app

def test_hello():
    response = app.test_client().get('/')

    assert response.status_code == 200
    assert response.data == b'Hello, World!'
```

Juste pour examiner, à ce stade, notre structure de projet obtenue avec la commande `tree` est la suivante:

```
.
├── hello.py
└── test_hello.py
```

Lancer le test

Maintenant, nous pouvons exécuter ce test avec la commande `py.test` qui découvrira automatiquement notre `test_hello.py` et la fonction de test qu'il `test_hello.py`.

```
$ py.test
```

Vous devriez voir une sortie et une indication que 1 test a réussi, par exemple

```
=== test session starts ===
collected 1 items
test_hello.py .
=== 1 passed in 0.13 seconds ===
```

Test d'une API JSON implémentée dans Flask

Cet exemple suppose que vous savez comment [tester une application Flask en utilisant pytest](#)

Voici une API qui prend une entrée JSON avec des valeurs entières `a` et `b` par exemple `{"a": 1, "b": 2}`, les ajoute et renvoie la somme `a + b` dans une réponse JSON, par exemple `{"sum": 3}`.

```
# hello_add.py
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add', methods=['POST'])
def add():
    data = request.get_json()
    return jsonify({'sum': data['a'] + data['b']})
```

Test de cette API avec `pytest`

Nous pouvons le tester avec `pytest`

```
# test_hello_add.py
from hello_add import app
from flask import json

def test_add():
    response = app.test_client().post(
        '/add',
        data=json.dumps({'a': 1, 'b': 2}),
        content_type='application/json',
    )

    data = json.loads(response.get_data(as_text=True))

    assert response.status_code == 200
    assert data['sum'] == 3
```

Exécutez maintenant le test avec la commande `py.test` .

Accéder à et manipuler des variables de session dans vos tests en utilisant Flask-Testing

La plupart des applications Web utilisent l'objet de session pour stocker des informations importantes. Ces exemples montrent comment tester une telle application à l'aide de Flask-Testing. L'exemple de travail complet est également disponible sur [github](#) .

Donc, installez d'abord Flask-Testing dans votre virtualenv

```
pip install flask_testing
```

Pour pouvoir utiliser l'objet de session, vous devez définir la clé secrète

```
app.secret_key = 'my-seCret_KEy'
```

Imaginons que vous ayez dans votre application la fonction de stocker des données dans des variables de session comme celle-ci

```
@app.route('/getSessionVar', methods=['GET', 'POST'])
def getSessionVariable():
    if 'GET' == request.method:
        session['sessionVar'] = 'hello'
    elif 'POST' == request.method:
        session['sessionVar'] = 'hi'
    else:
        session['sessionVar'] = 'error'

    return 'ok'
```

Pour tester cette fonction, vous pouvez importer `flask_testing` et laisser votre classe de test hériter de `flask_testing.TestCase`. Importer aussi toutes les librairies nécessaires

```
import flask
import unittest
import flask_testing
from myapp.run import app

class TestMyApp(flask_testing.TestCase):
```

Avant de commencer le test, il est très important d'implémenter la fonction **create_app** sinon il y aura des exceptions.

```
def create_app(self):
    return app
```

Pour tester votre application fonctionne comme vous le souhaitez, vous avez plusieurs possibilités. Si vous voulez simplement vous assurer que votre fonction **attribue des** valeurs particulières à une variable de session, vous pouvez simplement conserver le contexte et accéder

à flask.session

```
def testSession1(self):
    with app.test_client() as lTestClient:
        lResp= lTestClient.get('/getSessionVar')
        self.assertEqual(lResp.status_code, 200)
        self.assertEqual(flask.session['sessionVar'], 'hello')
```

Une autre astuce utile consiste à différencier les méthodes *GET* et *POST* comme dans la prochaine fonction de test.

```
def testSession2(self):
    with app.test_client() as lTestClient:
        lResp= lTestClient.post('/getSessionVar')
        self.assertEqual(lResp.status_code, 200)
        self.assertEqual(flask.session['sessionVar'], 'hi')
```

Maintenant, imaginez que votre fonction attend une variable de session à définir et réagit différemment sur des valeurs particulières comme celle-ci

```
@app.route('/changeSessionVar')
def changeSessionVariable():
    if session['existingSessionVar'] != 'hello':
        raise Exception('unexpected session value of existingSessionVar!')

    session['existingSessionVar'] = 'hello world'
    return 'ok'
```

Pour tester cette fonction, vous devez utiliser la *transaction* dite de *session* et ouvrir la session dans le contexte du client de test. Cette fonction est disponible depuis **Flask 0.8**

```
def testSession3(self):
    with app.test_client() as lTestClient:
        #keep the session
        with lTestClient.session_transaction() as lSess:
            lSess['existingSessionVar'] = 'hello'

        #here the session is stored
        lResp = lTestClient.get('/changeSessionVar')
        self.assertEqual(lResp.status_code, 200)
        self.assertEqual(flask.session['existingSessionVar'], 'hello world')
```

Exécuter les tests est comme d'habitude pour unittest

```
if __name__ == "__main__":
    unittest.main()
```

Et dans la ligne de commande

```
python tests/test_myapp.py
```

Une autre bonne façon d'exécuter vos tests consiste à utiliser la découverte la plus courante

comme celle-ci:

```
python -m unittest discover -s tests
```

Lire Essai en ligne: <https://riptutorial.com/fr/flask/topic/1260/essai>

Chapitre 6: Fichiers statiques

Exemples

Utiliser des fichiers statiques

Les applications Web nécessitent souvent des fichiers statiques tels que des fichiers CSS ou JavaScript. Pour utiliser des fichiers statiques dans une application Flask, créez un dossier appelé `static` dans votre package ou à côté de votre module et il sera disponible sur `/static` dans l'application.

Un exemple de structure de projet pour l'utilisation de modèles est le suivant:

```
MyApplication/  
  /static/  
    /style.css  
    /script.js  
  /templates/  
    /index.html  
  /app.py
```

`app.py` est un exemple de base de Flask avec un rendu de modèle.

```
from flask import Flask, render_template  
  
app = Flask(__name__)  
  
@app.route('/')  
def index():  
    return render_template('index.html')
```

Pour utiliser le fichier CSS et JavaScript statiques dans le modèle `index.html`, nous devons utiliser le nom spécial de noeud final "statique":

```
{{url_for('static', filename = 'style.css')}}}
```

Donc, **index.html** peut contenir:

```
<html>  
  <head>  
    <title>Static File</title>  
    <link href="{{url_for('static', filename = 'style.css')}}" rel="stylesheet">  
    <script src="{{url_for('static', filename = 'script.js')}}"></script>  
  </head>  
  <body>  
    <h3>Hello World!</h3>  
  </body>  
</html>
```

Après avoir exécuté `app.py`, nous verrons la page Web dans <http://localhost:5000/>.

Fichiers statiques en production (servis par le serveur web frontal)

Le serveur Web intégré de Flask peut servir des ressources statiques, et cela fonctionne très bien pour le développement. Cependant, pour les déploiements de production utilisant quelque chose comme uWSGI ou Gunicorn pour servir l'application Flask, la tâche de servir des fichiers statiques est généralement déchargée sur le serveur Web frontal (Nginx, Apache, etc.). Il s'agit d'une tâche simple / petite avec des applications plus petites, en particulier lorsque toutes les ressources statiques se trouvent dans un seul dossier. Cependant, pour les applications plus volumineuses et / ou celles utilisant des plug-ins Flask qui fournissent des ressources statiques, il peut être difficile de se souvenir des emplacements de tous ces fichiers et de les copier / collecter manuellement dans un seul répertoire. Ce document montre comment utiliser le [plug-in Flask-Collect](#) pour simplifier cette tâche.

Notez que cette documentation se concentre sur la collecte des actifs statiques. Pour illustrer cette fonctionnalité, cet exemple utilise le plug-in Flask-Bootstrap, qui fournit des ressources statiques. Il utilise également le plug-in Flask-Script, qui simplifie le processus de création de tâches en ligne de commande. Aucun de ces plug-ins n'est critique pour ce document, ils sont juste utilisés ici pour démontrer la fonctionnalité. Si vous choisissez de ne pas utiliser Flask-Script, vous devrez examiner les documents [Flask-Collect pour trouver d'autres moyens d'appeler la commande collect](#).

Notez également que la configuration de votre serveur Web frontal pour servir ces ressources statiques ne fait pas partie de la portée de cette documentation, vous devrez consulter des exemples utilisant [Nginx](#) et [Apache](#) pour plus d'informations. Disons simplement que vous allez créer des alias qui commencent par `"/static"` dans le répertoire centralisé que Flask-Collect créera pour vous dans cet exemple.

L'application est structurée comme suit:

```
/manage.py - The app management script, used to run the app, and to collect static assets
/app/ - this folder contains the files that are specific to our app
| - __init__.py - Contains the create_app function
| - static/ - this folder contains the static files for our app.
|   | css/styles.css - custom styles for our app (we will leave this file empty)
|   | js/main.js - custom js for our app (we will leave this file empty)
| - templates/index.html - a simple page that extends the Flask-Bootstrap template
```

1. Tout d'abord, créez votre environnement virtuel et installez les packages requis: (your-virtualenv) `$ pip install flask-script flask-bootstrap flask-collect`

2. Établissez la structure de fichier décrite ci-dessus:

```
$ touch manage.py; mkdir -p app / {statique / {css, js}, templates}; appuyez sur app / { init
.py, static / {css / styles.css, js / main.js}}
```

3. Etablissez le contenu des `manage.py`, `app/__init__.py` et `app/templates/index.html` :

```
# manage.py
#!/usr/bin/env python
import os
```

```

from flask_script import Manager, Server
from flask import current_app
from flask_collect import Collect
from app import create_app

class Config(object):
    # CRITICAL CONFIG VALUE: This tells Flask-Collect where to put our static files!
    # Standard practice is to use a folder named "static" that resides in the top-level of the
    project directory.
    # You are not bound to this location, however; you may use basically any directory that you
    wish.
    COLLECT_STATIC_ROOT = os.path.dirname(__file__) + '/static'
    COLLECT_STORAGE = 'flask_collect.storage.file'

app = create_app(Config)

manager = Manager(app)
manager.add_command('runserver', Server(host='127.0.0.1', port=5000))

collect = Collect()
collect.init_app(app)

@manager.command
def collect():
    """Collect static from blueprints. Workaround for issue: https://github.com/klen/Flask-
    Collect/issues/22"""
    return current_app.extensions['collect'].collect()

if __name__ == "__main__":
    manager.run()

```

```

# app/__init__.py
from flask import Flask, render_template
from flask_collect import Collect
from flask_bootstrap import Bootstrap

def create_app(config):
    app = Flask(__name__)
    app.config.from_object(config)

    Bootstrap(app)
    Collect(app)

    @app.route('/')
    def home():
        return render_template('index.html')

    return app

```

```

# app/templates/index.html
{% extends "bootstrap/base.html" %}
{% block title %}This is an example page{% endblock %}

{% block navbar %}
<div class="navbar navbar-fixed-top">
  <!-- ... -->
</div>
{% endblock %}

```

```
{% block content %}
  <h1>Hello, Bootstrap</h1>
{% endblock %}
```

4. Avec ces fichiers en place, vous pouvez maintenant utiliser le script de gestion pour exécuter l'application:

```
$ ./manage.py runserver # visit http://localhost:5000 to verify that the app works correctly.
```

5. Maintenant, pour collecter vos actifs statiques pour la première fois. Avant de faire cela, notez à nouveau que vous ne devriez PAS avoir un dossier `static/` dans le niveau supérieur de votre application. C'est là que Flask-Collect va placer tous les fichiers statiques qu'il va collecter depuis votre application et les différents plugins que vous utilisez peut-être. Si vous avez un `static/` dossier dans le haut niveau de votre application, vous devez supprimer complètement avant de procéder, comme à partir d'une table rase est une partie essentielle du témoignage / comprendre ce Flask Collect fait. Notez que cette instruction n'est pas applicable pour un usage quotidien, c'est simplement pour illustrer le fait que Flask-Collect va créer ce répertoire pour vous, puis y placer un tas de fichiers.

Cela dit, vous pouvez exécuter la commande suivante pour collecter vos actifs statiques:

```
$ ./manage.py collect
```

Après cela, vous devriez voir que Flask-Collect a créé ce dossier `static/` niveau supérieur et qu'il contient les fichiers suivants:

```
$ find ./static -type f # execute this from the top-level directory of your app, same dir that
contains the manage.py script
static/bootstrap/css/bootstrap-theme.css
static/bootstrap/css/bootstrap-theme.css.map
static/bootstrap/css/bootstrap-theme.min.css
static/bootstrap/css/bootstrap.css
static/bootstrap/css/bootstrap.css.map
static/bootstrap/css/bootstrap.min.css
static/bootstrap/fonts/glyphicons-halflings-regular.eot
static/bootstrap/fonts/glyphicons-halflings-regular.svg
static/bootstrap/fonts/glyphicons-halflings-regular.ttf
static/bootstrap/fonts/glyphicons-halflings-regular.woff
static/bootstrap/fonts/glyphicons-halflings-regular.woff2
static/bootstrap/jquery.js
static/bootstrap/jquery.min.js
static/bootstrap/jquery.min.map
static/bootstrap/js/bootstrap.js
static/bootstrap/js/bootstrap.min.js
static/bootstrap/js/npm.js
static/css/styles.css
static/js/main.js
```

Et c'est tout: utilisez la commande `collect` chaque fois que vous apportez des modifications aux CSS ou JavaScript de votre application, ou lorsque vous avez mis à jour un plug-in Flask qui fournit des ressources statiques (comme Flask-Bootstrap dans cet exemple).

Lire Fichiers statiques en ligne: <https://riptutorial.com/fr/flask/topic/3678/fichiers-statiques>

Chapitre 7: Filtres de modèle Jinja2 personnalisés

Syntaxe

- `{{my_date_time | my_custom_filter}}`
- `{{my_date_time | my_custom_filter (args)}}`

Paramètres

Paramètre	Détails
valeur	La valeur transmise par Jinja, à filtrer
args	Arguments supplémentaires à transmettre à la fonction de filtrage

Exemples

Formatage de la date et de l'heure dans un modèle Jinja2

Les filtres peuvent être définis dans une méthode, puis ajoutés au dictionnaire de filtres de Jinja ou définis dans une méthode décorée avec `Flask.template_filter`.

Définir et enregistrer plus tard:

```
def format_datetime(value, format="%d %b %Y %I:%M %p"):  
    """Format a date time to (Default): d Mon YYYY HH:MM P"""  
    if value is None:  
        return ""  
    return value.strftime(format)  
  
# Register the template filter with the Jinja Environment  
app.jinja_env.filters['formatdatetime'] = format_datetime
```

Définition avec décorateur:

```
@app.template_filter('formatdatetime')  
def format_datetime(value, format="%d %b %Y %I:%M %p"):  
    """Format a date time to (Default): d Mon YYYY HH:MM P"""  
    if value is None:  
        return ""  
    return value.strftime(format)
```

Lire Filtres de modèle Jinja2 personnalisés en ligne:

<https://riptutorial.com/fr/flask/topic/1465/filtres-de-modele-jinja2-personnalises>

Chapitre 8: Flacon-WTF

Introduction

C'est une intégration simple de Flask et WTForms. Il facilite la création et la gestion des formulaires Web. Il génère automatiquement un champ masqué CSRF dans vos modèles. Il comporte également des fonctions de validation de formulaire faciles

Exemples

Un simple formulaire

```
from flask_wtf import FlaskForm
from wtforms import StringField, IntegerField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField('name', validators=[DataRequired()])
    age = IntegerField('age', validators=[DataRequired()])
```

Pour rendre le modèle, vous utiliserez quelque chose comme ceci:

```
<form method="POST" action="/">
  {{ form.hidden_tag() }}
  {{ form.name.label }} {{ form.name(size=20) }}
  <br/>
  {{ form.age.label }} {{ form.age(size=3) }}
  <input type="submit" value="Go">
</form>
```

Le code simple ci-dessus va générer notre formulaire Web flask-wtf très simple avec un champ de jeton CSRF masqué.

Lire Flacon-WTF en ligne: <https://riptutorial.com/fr/flask/topic/10579/flacon-wtf>

Chapitre 9: Flask sur Apache avec mod_wsgi

Exemples

Wrapper d'application WSGI

De nombreuses applications Flask sont développées dans *virtualenv* pour conserver des dépendances pour chaque application, distinctes de l'installation Python à l'échelle du système. Assurez-vous que *mod-wsgi* est installé dans votre *virtualenv* :

```
pip install mod-wsgi
```

Créez ensuite un wrapper wsgi pour votre application Flask. Il est généralement conservé dans le répertoire racine de votre application.

my-application.wsgi

```
activate_this = '/path/to/my-application/venv/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
import sys
sys.path.insert(0, '/path/to/my-application')

from app import app as application
```

Ce wrapper active l'environnement virtuel et tous ses modules et dépendances installés lorsqu'il est exécuté à partir d'Apache, et s'assure que le chemin de l'application est le premier dans les chemins de recherche. Par convention, les objets d'application WSGI sont appelés `application`.

Configuration des sites Apache pour WSGI

L'avantage d'utiliser Apache sur le serveur intégré Werkzeug est qu'Apache est multi-thread, ce qui signifie que plusieurs connexions à l'application peuvent être effectuées simultanément. Ceci est particulièrement utile dans les applications qui utilisent *XmlHttpRequest* (AJAX) sur le front-end.

/etc/apache2/sites-available/050-my-application.conf (ou configuration apache par défaut si elle n'est pas hébergée sur un serveur Web partagé)

```
<VirtualHost *:80>
    ServerName my-application.org

    ServerAdmin admin@my-application.org

    # Must be set, but can be anything unless you want to serve static files
    DocumentRoot /var/www/html

    # Logs for your application will go to the directory as specified:

    ErrorLog ${APACHE_LOG_DIR}/error.log
```



```
CustomLog ${APACHE_LOG_DIR}/access.log combined

# WSGI applications run as a daemon process, and need a specified user, group
# and an allocated number of thread workers. This will determine the number
# of simultaneous connections available.
WSGIDaemonProcess my-application user=username group=username threads=12

# The WSGIScriptAlias should redirect / to your application wrapper:
WSGIScriptAlias / /path/to/my-application/my-application.wsgi
# and set up Directory access permissions for the application:
<Directory /path/to/my-application>
    WSGIProcessGroup my-application
    WSGIApplicationGroup %{GLOBAL}

    AllowOverride none
    Require all granted
</Directory>
</VirtualHost>
```

Lire Flask sur Apache avec mod_wsgi en ligne: <https://riptutorial.com/fr/flask/topic/6851/flask-sur-apache-avec-mod-wsgi>

Chapitre 10: Flask-SQLAlchemy

Introduction

Flask-SQLAlchemy est une extension Flask qui ajoute la prise en charge de la populaire application de mappage d'objets relationnels (ORM) SQLAlchemy aux applications Flask. Il vise à simplifier SQLAlchemy with Flask en fournissant des implémentations par défaut pour les tâches courantes.

Exemples

Installation et exemple initial

Installation

```
pip install Flask-SQLAlchemy
```

Modèle simple

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    email = db.Column(db.String(120), unique=True)
```

L'exemple de code ci-dessus montre un simple modèle Flask-SQLAlchemy, nous pouvons ajouter un nom de tableau optionnel à la déclaration du modèle, mais ce n'est souvent pas nécessaire car Flask-SQLAlchemy utilisera automatiquement le nom de la classe comme nom de la base de données.

Notre classe héritera du modèle de base qui est une base déclarative configurée. Il n'est donc pas nécessaire de définir explicitement la base comme nous le ferions avec SQLAlchemy.

Référence

- URL Pypi: [<https://pypi.python.org/pypi/Flask-SQLAlchemy>][1]
- URL de la documentation: [<http://flask-sqlalchemy.pocoo.org/2.1/>][1]

Relations: un à plusieurs

```
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80))
    email = db.Column(db.String(120), unique=True)
    posts = db.relationship('Post', backref='user')

class Post(db.Model):
```

```
id = db.Column(db.Integer, primary_key=True)
content = db.Column(db.Text)
user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
```

Dans cet exemple, nous avons deux classes, la classe User et la classe Post, la classe User sera notre parent et la publication sera notre publication car seule la publication peut appartenir à un utilisateur, mais un utilisateur peut avoir plusieurs publications. Afin de réaliser cela, nous plaçons une clé étrangère sur l'enfant faisant référence au parent issu de notre exemple. Nous plaçons une clé étrangère sur la classe Post pour faire référence à la classe User. On utilise ensuite la `relationship()` sur le parent auquel on accède via notre objet SQLAlchemy `db`. Cela nous permet alors de référencer une collection d'objets représentés par la classe Post qui est notre enfant.

Pour créer une relation bidirectionnelle, nous utilisons `backref`, cela permettra à l'enfant de référencer le parent.

Lire Flask-SQLAlchemy en ligne: <https://riptutorial.com/fr/flask/topic/10577/flask-sqlalchemy>

Chapitre 11: Le routage

Exemples

Itinéraires de base

Les itinéraires dans Flask peuvent être définis à l'aide du décorateur de `route` de l'instance d'application Flask:

```
app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello Flask'
```

Le décorateur de `route` prend une chaîne qui correspond à l'URL. Lorsqu'une demande d'URL correspondant à cette chaîne est reçue par l'application, la fonction décorée (également appelée *fonction de vue*) est appelée. Donc, pour un itinéraire à propos, nous aurions:

```
@app.route('/about')
def about():
    return 'About page'
```

Il est important de noter que ces routes **ne** sont **pas** des expressions régulières comme dans Django.

Vous pouvez également définir *des règles de variable* pour extraire les valeurs de segment d'URL en variables:

```
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id):
    # look up the blog post with id post_id
    # return some kind of HTML
```

Ici, la règle de variable est dans le dernier segment de l'URL. Quelle que soit la valeur `get_blog_post` dans le dernier segment de l'URL, elle sera transmise à la fonction de vue (`get_blog_post`) en tant que paramètre `post_id`. Donc, une requête à `/blog/posts/42` récupérera (ou tentera de récupérer) l'article du blog avec un identifiant de 42.

Il est également courant de réutiliser les URL. Par exemple, nous souhaitons peut-être que `/blog/posts` renvoie une liste de tous les articles du blog. Nous pourrions donc avoir deux routes pour la même fonction d'affichage:

```
@app.route('/blog/posts')
@app.route('/blog/posts/<post_id>')
def get_blog_post(post_id=None):
    # get the post or list of posts
```

Notez ici que nous devons également fournir la valeur par défaut `None` pour le `post_id` dans `get_blog_post` . Lorsque le premier itinéraire est mis en correspondance, aucune valeur ne sera transmise à la fonction d'affichage.

Notez également que, par défaut, le type d'une règle de variable est une chaîne. Cependant, vous pouvez spécifier plusieurs types différents tels que `int` et `float` en préfixant la variable:

```
@app.route('/blog/post/<int:post_id>')
```

Les convertisseurs d'URL intégrés à Flask sont:

```
string | Accepts any text without a slash (the default).
int     | Accepts integers.
float   | Like int but for floating point values.
path    | Like string but accepts slashes.
any     | Matches one of the items provided
uuid    | Accepts UUID strings
```

Si nous essayons de visiter l'URL `/blog/post/foo` avec une valeur dans le dernier segment d'URL qui ne peut pas être converti en entier, l'application renvoie une erreur 404. C'est l'action correcte car il n'y a pas de règle avec `/blog/post` et une chaîne dans le dernier segment.

Enfin, les routes peuvent également être configurées pour accepter les méthodes HTTP. Le décorateur de `route` utilise un argument de mot-clé de `methods` qui est une liste de chaînes représentant les méthodes HTTP acceptables pour cette route. Comme vous l'avez peut-être supposé, la valeur par défaut est `GET` uniquement. Si nous avons un formulaire pour ajouter un nouvel article de blog et que nous voulions retourner le code HTML de la requête `GET` et analyser les données de formulaire pour la requête `POST` , la route ressemblerait à ceci:

```
@app.route('/blog/new', methods=['GET', 'POST'])
def new_post():
    if request.method == 'GET':
        # return the form
    elif request.method == 'POST':
        # get the data from the form values
```

La `request` se trouve dans le package `flask` . Notez que lorsque vous utilisez l'argument du mot-clé `methods` , vous devez être explicite sur les méthodes HTTP à accepter. Si nous n'avions listé que `POST` , la route ne répondrait plus aux requêtes `GET` et renverrait une erreur 405.

Catch-all route

Il peut être utile d'avoir une vue d'ensemble où vous manipulez vous-même une logique complexe en fonction du chemin. Cet exemple utilise deux règles: La première règle attrape spécifiquement `/` et la seconde règle des chemins arbitraires avec le convertisseur de `path` intégré. Le convertisseur de `path` correspond à n'importe quelle chaîne (y compris les barres obliques) Voir [Règles variables des flacons](#)

```
@app.route('/', defaults={'u_path': ''})
```

```
@app.route('/<path:u_path>')
def catch_all(u_path):
    print(repr(u_path))
    ...
```

```
c = app.test_client()
c.get('/') # u_path = ''
c.get('/hello') # u_path = 'hello'
c.get('/hello/stack/overflow/') # u_path = 'hello/stack/overflow/'
```

Méthodes de routage et HTTP

Par défaut, les routes ne répondent qu'aux requêtes `GET`. Vous pouvez modifier ce comportement en fournissant les `methods` argument de la `route()` décorateur.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

Vous pouvez également mapper différentes fonctions sur le même noeud final en fonction de la méthode HTTP utilisée.

```
@app.route('/endpoint', methods=['GET'])
def get_endpoint():
    #respond to GET requests for '/endpoint'

@app.route('/endpoint', methods=['POST', 'PUT', 'DELETE'])
def post_or_put():
    #respond to POST, PUT, or DELETE requests for '/endpoint'
```

Lire Le routage en ligne: <https://riptutorial.com/fr/flask/topic/2415/le-routage>

Chapitre 12: Les bleus

Introduction

Les plans directeurs sont un concept puissant dans le développement d'applications Flask qui permet aux applications de fonctionner plus modulaires et de pouvoir suivre plusieurs modèles. Ils facilitent l'administration de très grandes applications en fonctionnaires et peuvent donc être utilisés pour mettre à l'échelle les applications Flask. Vous pouvez réutiliser les applications Blueprint, mais vous ne pouvez pas exécuter un Blueprint car il doit être enregistré sur votre application principale.

Exemples

Un exemple de base de plans de fonctionnaires

Une application Flask minimale ressemble à ceci:

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def index():
    return "Hello World!"
```

Une grande application Flask peut séparer un fichier en plusieurs fichiers par des `blueprints`.

Objectif

Facilitez la maintenance de l'application pour les autres.

Structure de dossier de grande application

```
/app
  /templates
  /static
  /views
    __init__.py
    index.py
  app.py
```

views / index.py

```
from flask import Blueprint, render_template

index_blueprint = Blueprint('index', __name__)

@index_blueprint.route("/")
def index():
```

```
return "Hello World!"
```

app.py

```
from flask import Flask
from views.index import index_blueprint

application = Flask(__name__)
application.register_blueprint(index_blueprint)
```

Exécuter l'application

```
$ export FLASK_APP=app.py
$ flask run
```

Lire Les bleus en ligne: <https://riptutorial.com/fr/flask/topic/6427/les-bleus>

Chapitre 13: Les signaux

Remarques

Flask prend en charge les signaux en utilisant [Blinker](#) . Le support du signal est facultatif. ils ne seront activés que si Blinker est installé.

```
pip install blinker
```

<http://flask.pocoo.org/docs/dev/signals/>

Les signaux ne sont pas asynchrones. Lorsqu'un signal est envoyé, il exécute immédiatement chacune des fonctions connectées séquentiellement.

Exemples

Connexion aux signaux

Utilisation d'un signal de `connect` méthode pour connecter une fonction à un signal. Lorsqu'un signal est envoyé, chaque fonction connectée est appelée avec l'expéditeur et tout argument nommé fourni par le signal.

```
from flask import template_rendered

def log_template(sender, template, context, **kwargs):
    sender.logger.info(
        'Rendered template %(template)r with context %(context)r.',
        template=template, context=context
    )

template_rendered.connect(log_template)
```

Consultez la documentation sur les [signaux intégrés](#) pour obtenir des informations sur les arguments fournis. Un modèle utile consiste à ajouter un argument `**kwargs` pour intercepter des arguments inattendus.

Signaux personnalisés

Si vous souhaitez [créer et envoyer des signaux](#) dans votre propre code (par exemple, si vous écrivez une extension), créez une nouvelle instance de `Signal` et `send` appel lorsque les abonnés doivent être avertis. Les signaux sont créés à l'aide d'un [Namespace](#) .

```
from flask import current_app
from flask.signals import Namespace

namespace = Namespace()
```

```
message_sent = namespace.signal('mail_sent')

def message_response(recipient, body):
    ...
    message_sent.send(
        current_app._get_current_object(),
        recipient=recipient,
        body=body
    )

@message_sent.connect
def log_message(app, recipient, body):
    ...
```

Préférez utiliser le support de signal de Flask avec Blinker directement. Il encapsule la bibliothèque pour que les signaux restent facultatifs si les développeurs utilisant votre extension n'ont pas choisi d'installer Blinker.

Lire Les signaux en ligne: <https://riptutorial.com/fr/flask/topic/2331/les-signaux>

Chapitre 14: Message clignotant

Introduction

Message clignotant au modèle par `flash()` fonction `flash()` .

Syntaxe

- `flash (message, category = 'message')`
- `flash ('salut, monde!')`
- `flash ('Ceci est un message d'avertissement', 'warning')`

Paramètres

message	le message à clignoter.
Catégorie	la catégorie du message, la valeur par défaut est <code>message</code> .

Remarques

- [Héritage du modèle](#)
- [API](#)

Exemples

Message simple clignotant

Définissez `SECRET_KEY` , puis le message clignotant dans la fonction d'affichage:

```
from flask import Flask, flash, render_template

app = Flask(__name__)
app.secret_key = 'some_secret'

@app.route('/')
def index():
    flash('Hello, I'm a message.')
    return render_template('index.html')
```

`layout.html` ensuite les messages dans `layout.html` (que l' `index.html` étendu depuis):

```
{% with messages = get_flashed_messages() %}
{% if messages %}
<ul class=flashes>
{% for message in messages %}
```

```
    <li>{{ message }}</li>
  {% endfor %}
</ul>
{% endif %}
{% endwith %}
{% block body %}{% endblock %}
```

Clignotant Avec Catégories

Définissez le second argument lorsque vous utilisez `flash()` dans la fonction d'affichage:

```
flash('Something was wrong!', 'error')
```

Dans le modèle, définissez `with_categories=true` dans `get_flashed_messages()`, vous obtenez alors une liste de tuples sous la forme de `(message, category)`, vous pouvez donc utiliser `category` en tant que classe HTML.

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    <ul class=flashes>
      {% for category, message in messages %}
        <li class="{{ category }}">{{ message }}</li>
      {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
```

Lire Message clignotant en ligne: <https://riptutorial.com/fr/flask/topic/10756/message-clignotant>

Chapitre 15: Modèles de rendu

Syntaxe

- `render_template(template_name_or_list, **context)`

Exemples

Usage `render_template`

Flask vous permet d'utiliser des modèles pour le contenu de pages Web dynamiques. Un exemple de structure de projet pour l'utilisation de modèles est le suivant:

```
myproject/  
  /app/  
    /templates/  
      /index.html  
    /views.py
```

views.py :

```
from flask import Flask, render_template  
  
app = Flask(__name__)  
  
@app.route("/")  
def index():  
    pagetitle = "HomePage"  
    return render_template("index.html",  
                           mytitle=pagetitle,  
                           mycontent="Hello World")
```

Notez que vous pouvez transmettre du contenu dynamique de votre gestionnaire de routage au modèle en ajoutant des paires clé / valeur à la fonction `render_templates`. Dans l'exemple ci-dessus, les variables "pagetitle" et "mycontent" seront transmises au modèle pour inclusion dans la page rendue. Incluez ces variables dans le modèle en les plaçant entre accolades: `{{mytitle}}`

index.html :

```
<html>  
  <head>  
    <title>{{ mytitle }}</title>  
  </head>  
  <body>  
    <p>{{ mycontent }}</p>  
  </body>  
</html>
```

Lorsqu'il est exécuté comme le premier exemple, `http://localhost:5000/` aura le titre "HomePage"

et un paragraphe avec le contenu "Hello World".

Lire Modèles de rendu en ligne: <https://riptutorial.com/fr/flask/topic/1641/modeles-de-rendu>

Chapitre 16: Pagination

Exemples

Exemple de route de pagination avec `Pagin-sqlalchemy` `Paginate`

Dans cet exemple, nous utilisons un paramètre dans l'itinéraire pour spécifier le numéro de page. Nous définissons une valeur par défaut de 1 dans la `page=1` paramètres de la fonction `page=1` . Nous avons un objet `User` dans la base de données et nous l'interrogeons, en ordre décroissant, en indiquant d'abord les derniers utilisateurs. Nous utilisons ensuite la méthode `paginate` de l'objet de `query` dans `flask-sqlalchemy`. Nous passons ensuite ceci à `render_template` pour être rendu.

```
@app.route('/users')
@app.route('/users/page/<int:page>')
def all_users(page=1):
    try:
        users_list = User.query.order_by(
            User.id.desc()
        ).paginate(page, per_page=USERS_PER_PAGE)
    except OperationalError:
        flash("No users in the database.")
        users_list = None

    return render_template(
        'users.html',
        users_list=users_list,
        form=form
    )
```

Rendre la pagination à Jinja

Ici, nous utilisons l'objet que nous avons passé à `render_template` pour afficher les pages, la page active en cours, ainsi que les boutons précédent et suivant si vous pouvez aller à la page précédente / suivante.

```
<!-- previous page -->
{% if users_list.has_prev %}
<li>
    <a href="{{ url_for('users', page=users_list.prev_num) }}">Previous</a>
</li>
{% endif %}

<!-- all page numbers -->
{% for page_num in users_list.iter_pages() %}
    {% if page_num %}
        {% if page_num != users_list.page %}
            <li>
                <a href="{{ url_for('users', page=page_num) }}">{{ page_num }}</a>
            </li>
        {% else %}
            <li class="active">
                <a href="#">{{ page_num }}</a>
            </li>
        {% endif %}
    {% endif %}
{% endfor %}
```

```
        </li>
    {% endif %}
{% else %}
    <li>
        <span class="ellipsis" style="white-space: nowrap; overflow: hidden; text-overflow:
ellipsis">...</span>
    </li>
{% endif %}
{% endfor %}

<!-- next page -->
{% if users_list.has_next %}
<li>
    <a href="{{ url_for('users', page=users_list.next_num) }}">Next</a></li>
{% endif %}
{% endif %}
```

Lire Pagination en ligne: <https://riptutorial.com/fr/flask/topic/6460/pagination>

Chapitre 17: Réorienter

Syntaxe

- rediriger (localisation, code, réponse)

Paramètres

Paramètre	Détails
emplacement	L'emplacement auquel la réponse doit rediriger.
code	(Facultatif) Le code d'état de la redirection, 302 par défaut. Les codes pris en charge sont 301, 302, 303, 305 et 307.
Réponse	(Facultatif) Classe de réponse à utiliser lors de l'instanciation d'une réponse. La valeur par défaut est <code>werkzeug.wrappers.Response</code> si non spécifiée.

Remarques

Le paramètre d'emplacement doit être une URL. Il peut s'agir d'une entrée brute, telle que "<http://www.webpage.com>", ou être créée avec la fonction `url_for()`.

Exemples

Exemple simple

```
from flask import Flask, render_template, redirect, url_for

app = Flask(__name__)

@app.route('/')
def main_page():
    return render_template('main.html')

@app.route('/main')
def go_to_main():
    return redirect(url_for('main_page'))
```

Transmission de données

```
# ...
# same as above

@app.route('/welcome/<name>')
def welcome(name):
```

```
    return render_template('main.html', name=name)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        # ...
        # check for valid login, assign username
        if valid:
            return redirect(url_for('main_page', name=username))
        else:
            return redirect(url_for('login_error'))
    else:
        return render_template('login.html')
```

Lire Réorienter en ligne: <https://riptutorial.com/fr/flask/topic/6856/reorienter>

Chapitre 18: Sessions

Remarques

Les sessions sont dérivées de dictionnaires, ce qui signifie qu'elles fonctionneront avec les méthodes de dictionnaire les plus courantes.

Exemples

Utilisation de l'objet sessions dans une vue

Tout d'abord, assurez-vous d'avoir importé des sessions depuis le flacon

```
from flask import session
```

Pour utiliser la session, une application Flask nécessite un **SECRET_KEY** défini.

```
app = Flask(__name__)
app.secret_key = 'app secret key'
```

Les sessions sont implémentées par défaut avec un **cookie** signé avec la clé secrète. Cela garantit que les données ne sont pas modifiées, sauf par votre application, alors assurez-vous d'en choisir une sécurisée! Un navigateur renverra le cookie à votre application avec chaque demande, permettant ainsi la persistance des données entre les requêtes.

Pour utiliser une session, il suffit de référencer l'objet (il se comportera comme un dictionnaire)

```
@app.route('/')
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])
```

Pour libérer une variable de session, utilisez la méthode **pop ()** .

```
session.pop('counter', None)
```

Exemple de code:

```
from flask import Flask, session

app = Flask(__name__)
app.secret_key = 'app secret key'

@app.route('/')
```

```
def index():
    if 'counter' in session:
        session['counter'] += 1
    else:
        session['counter'] = 1
    return 'Counter: '+str(session['counter'])

if __name__ == '__main__':
    app.debug = True
    app.run()
```

Lire Sessions en ligne: <https://riptutorial.com/fr/flask/topic/2748/sessions>

Chapitre 19: Téléchargement de fichier

Syntaxe

- `request.files ['name']` # fichier unique requis
- `request.files.get ('name')` # Aucun s'il n'est pas posté
- `request.files.getlist ('name')` # liste de zéro ou plusieurs fichiers publiés
- `CombinedMultiDict ((request.files, request.form))` # combine les données de formulaire et de fichier

Exemples

Téléchargement de fichiers

Formulaire HTML

- Utilisez une [entrée de type de file](#) et le navigateur fournira un champ permettant à l'utilisateur de sélectionner un fichier à télécharger.
- Seuls les formulaires avec la méthode `post` peuvent envoyer des données de fichier.
- Assurez-vous de définir l' `enctype=multipart/form-data` . Sinon, le nom du fichier sera envoyé mais pas les données du fichier.
- Utilisez l'attribut `multiple` sur l'entrée pour permettre de sélectionner plusieurs fichiers pour le champ unique.

```
<form method=post enctype=multipart/form-data>
  <!-- single file for the "profile" field -->
  <input type=file name=profile>
  <!-- multiple files for the "charts" field -->
  <input type=file multiple name=charts>
  <input type=submit>
</form>
```

Demandes Python

[Requests](#) est une bibliothèque Python puissante pour faire des requêtes HTTP. Vous pouvez l'utiliser (ou d'autres outils) pour [publier des fichiers](#) sans navigateur.

- Ouvrez les fichiers à lire en mode binaire.
- Il existe plusieurs structures de données que les `files` prennent. Cela montre une liste de tuples `(name, data)` , qui autorise plusieurs fichiers comme le formulaire ci-dessus.

```
import requests

with open('profile.txt', 'rb') as f1, open('chart1.csv', 'rb') as f2, open('chart2.csv', 'rb')
```

```
as f3:
    files = [
        ('profile', f1),
        ('charts', f2),
        ('charts', f3)
    ]
    requests.post('http://localhost:5000/upload', files=files)
```

Ceci n'est pas censé être une liste exhaustive. Pour des exemples d'utilisation de votre outil préféré ou de scénarios plus complexes, consultez la documentation de cet outil.

Enregistrer les téléchargements sur le serveur

Les fichiers téléchargés sont disponibles dans `request.files`, un `MultiDict` champ de mappage `MultiDict` pour les objets de fichier. Utilisez `getlist` - au lieu de `[]` ou `get` - si plusieurs fichiers ont été téléchargés avec le même nom de champ.

```
request.files['profile'] # single file (even if multiple were sent)
request.files.getlist('charts') # list of files (even if one was sent)
```

Les objets dans `request.files` ont une méthode de `save` qui enregistre le fichier localement. Créez un répertoire commun pour enregistrer les fichiers.

L'attribut `filename` est le nom avec lequel le fichier a été téléchargé. Cela peut être défini de manière arbitraire par le client, alors transmettez-le via la méthode `secure_filename` pour générer un nom valide et sûr sous `secure_filename` enregistrer. Cela ne garantit pas que le nom est *unique*, de sorte que les fichiers existants seront écrasés à moins que vous ne fassiez un travail supplémentaire pour le détecter.

```
import os
from flask import render_template, request, redirect, url_for
from werkzeug import secure_filename

# Create a directory in a known location to save files to.
uploads_dir = os.path.join(app.instance_path, 'uploads')
os.makedirs(uploads_dir, exists_ok=True)

@app.route('/upload', methods=['GET', 'POST'])
def upload():
    if request.method == 'POST':
        # save the single "profile" file
        profile = request.files['profile']
        profile.save(os.path.join(uploads_dir, secure_filename(profile.filename)))

        # save each "charts" file
        for file in request.files.getlist('charts'):
            file.save(os.path.join(uploads_dir, secure_filename(file.name)))

    return redirect(url_for('upload'))

return render_template('upload.html')
```

Transmission de données à WTFORMS et Flask-WTF

WTForms fournit un `FileField` pour restituer une entrée de type de fichier. Il ne fait rien de spécial avec les données téléchargées. Toutefois, comme Flask divise les données de formulaire (`request.form`) et les données de fichier (`request.files`), vous devez vous assurer de transmettre les données correctes lors de la création du formulaire. Vous pouvez utiliser un `CombinedMultiDict` pour combiner les deux en une structure unique que WTForms comprend.

```
form = ProfileForm(CombinedMultiDict((request.files, request.form)))
```

Si vous utilisez [Flask-WTF](#) , une extension pour intégrer Flask et WTForms, le transfert des données correctes sera géré automatiquement pour vous.

En raison d'un bogue dans WTForms, un seul fichier sera présent pour chaque champ, même si plusieurs ont été téléchargés. Voir [ce numéro](#) pour plus de détails. Il sera corrigé dans 3.0.

TÉLÉCHARGER LE FICHIER PARSE CSV EN TANT QUE LISTE DE DICTIONNAIRES EN FLACON SANS ÉCONOMIE

Les développeurs doivent souvent concevoir des sites Web permettant aux utilisateurs de télécharger un fichier CSV. En règle générale, il n'y a **aucune raison** de **sauvegarder** le fichier CSV actuel car les données seront traitées et / ou stockées dans une base de données une fois téléchargées. Cependant, la plupart des méthodes d'analyse syntaxique des données CSV, si ce n'est la plupart, nécessitent que les données soient lues sous forme de fichier. Cela peut poser un problème si vous utilisez **FLASK** pour le développement Web.

Supposons que notre CSV a une ligne d'en-tête et ressemble à ceci:

```
h1,h2,h3
'yellow','orange','blue'
'green','white','black'
'orange','pink','purple'
```

Maintenant, supposons que le formulaire HTML pour télécharger un fichier est comme suit:

```
<form action="upload.html" method="post" enctype="multipart/form-data">
  <input type="file" name="fileupload" id="fileToUpload">
  <input type="submit" value="Upload File" name="submit">
</form>
```

Puisque personne ne veut réinventer la roue, vous décidez d' **importer csv** dans votre script **FLASK** . Il n'y a aucune garantie que les utilisateurs téléchargent le fichier csv avec les colonnes dans le bon ordre. Si le fichier csv a une ligne d'en-tête, alors, à l'aide de la méthode **csv.DictReader** , vous pouvez lire le fichier CSV sous la forme d'une liste de dictionnaires, en saisissant les entrées de la ligne d'en-tête. Cependant, **csv.DictReader** a besoin d'un fichier et n'accepte pas directement les chaînes. Vous pensez peut-être que vous devez utiliser les méthodes **FLASK** pour enregistrer d'abord le fichier téléchargé, obtenir le nouveau nom et l'emplacement du fichier, l'ouvrir à l'aide de **csv.DictReader** , puis supprimer le fichier. Semble un peu comme un déchet.

Heureusement, nous pouvons obtenir le contenu du fichier sous forme de chaîne, puis diviser la chaîne par des lignes terminées. La méthode `csv.DictReader` acceptera ceci comme substitut à un fichier. Le code suivant montre comment cela peut être accompli sans enregistrer temporairement le fichier.

```
@application.route('upload.html', methods = ['POST'])
def upload_route_summary():
    if request.method == 'POST':

        # Create variable for uploaded file
        f = request.files['fileupload']

        #store the file contents as a string
        fstring = f.read()

        #create list of dictionaries keyed by header row
        csv_dicts = [{k: v for k, v in row.items()} for row in
        csv.DictReader(fstring.splitlines(), skipinitialspace=True)]

        #do something list of dictionaries
        return "success"
```

La variable `csv_dicts` est maintenant la liste de dictionnaires suivante:

```
csv_dicts =
[
    {'h1':'yellow', 'h2':'orange', 'h3':'blue'},
    {'h1':'green', 'h2':'white', 'h3':'black'},
    {'h1':'orange', 'h2':'pink', 'h3':'purple'}
]
```

Si vous êtes nouveau sur PYTHON, vous pouvez accéder aux données suivantes:

```
csv_dicts[1]['h2'] = 'white'
csv_dicts[0]['h3'] = 'blue'
```

D'autres solutions impliquent l'importation du module `io` et utilisent la méthode `io.Stream`. Je pense que c'est une approche plus simple. Je crois que le code est un peu plus facile à suivre que d'utiliser la méthode `io`. Cette approche est spécifique à l'exemple d'analyse d'un fichier CSV téléchargé.

Lire Téléchargement de fichier en ligne: <https://riptutorial.com/fr/flask/topic/5459/telechargement-de-fichier>

Chapitre 20: Travailler avec JSON

Exemples

Renvoie une réponse JSON à partir de l'API Flask

Flask possède un utilitaire appelé `jsonify()` qui facilite le retour des réponses JSON

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/get-json')
def hello():
    return jsonify(hello='world') # Returns HTTP Response with {"hello": "world"}
```

Essayez-le avec une `curl`

```
curl -X GET http://127.0.0.1:5000/api/get-json
{
  "hello": "world"
}
```

Autres façons d'utiliser `jsonify()`

En utilisant un dictionnaire existant:

```
person = {'name': 'Alice', 'birth-year': 1986}
return jsonify(person)
```

En utilisant une liste:

```
people = [{'name': 'Alice', 'birth-year': 1986},
          {'name': 'Bob', 'birth-year': 1985}]
return jsonify(people)
```

Recevoir JSON à partir d'une requête HTTP

Si le type MIME de la requête HTTP est `application/json`, l'appel de `request.get_json()` renverra les données JSON analysées (sinon, il renvoie `None`).

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api/echo-json', methods=['GET', 'POST', 'DELETE', 'PUT'])
```

```
def add():  
  
    data = request.get_json()  
    # ... do your business logic, and return some response  
    # e.g. below we're just echo-ing back the received JSON data  
    return jsonify(data)
```

Essayez-le avec une `curl`

Le paramètre `-H 'Content-Type: application/json'` spécifie qu'il s'agit d'une requête JSON:

```
curl -X POST -H 'Content-Type: application/json' http://127.0.0.1:5000/api/echo-json -d  
'{"name": "Alice"}'  
{  
  "name": "Alice"  
}
```

Pour envoyer des requêtes à l'aide d'autres méthodes HTTP, remplacez `curl -X POST` par la méthode souhaitée, par exemple `curl -X GET`, `curl -X PUT`, etc.

Lire [Travailler avec JSON en ligne](https://riptutorial.com/fr/flask/topic/1789/travailler-avec-json): <https://riptutorial.com/fr/flask/topic/1789/travailler-avec-json>

Chapitre 21: Vues basées sur la classe

Exemples

Exemple de base

Avec les vues basées sur les classes, nous utilisons des classes au lieu de méthodes pour implémenter nos vues. Un exemple simple d'utilisation des vues basées sur les classes se présente comme suit:

```
from flask import Flask
from flask.views import View

app = Flask(__name__)

class HelloWorld(View):

    def dispatch_request(self):
        return 'Hello World!'

class HelloUser(View):

    def dispatch_request(self, name):
        return 'Hello {}'.format(name)

app.add_url_rule('/hello', view_func=HelloWorld.as_view('hello_world'))
app.add_url_rule('/hello/<string:name>', view_func=HelloUser.as_view('hello_user'))

if __name__ == "__main__":
    app.run(host='0.0.0.0', debug=True)
```

Lire Vues basées sur la classe en ligne: <https://riptutorial.com/fr/flask/topic/7494/vues-basees-sur-la-classe>

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec Flask	arsho , bakkal , Community , davidism , ettanany , Martijn Pieters , mmenschig , Sean Vieira , Shrike
2	Accès aux données de demande	RPi Awesomeness
3	Autorisation et authentification	boreq , Ninad Mhatre
4	Déploiement d'une application Flask à l'aide du serveur Web uWSGI avec Nginx	Gal Dreiman , Tempux , user305883 , wimkeir , Wombatz
5	Essai	bakkal , oggo
6	Fichiers statiques	arsho , davidism , MikeC , YellowShark
7	Filtres de modèle Jinja2 personnalisés	Celeo , dylanj.nz
8	Flacon-WTF	Achim Munene
9	Flask sur Apache avec mod_wsgi	Aaron D
10	Flask-SQLAlchemy	Achim Munene , arsho , Matt Davis
11	Le routage	davidism , Douglas Starnes , Grey Li , junnytony , Luke Taylor , MikeC , mmenschig , sytech
12	Les bleus	Achim Munene , Kir Chou , stamaimer
13	Les signaux	davidism
14	Message clignotant	Grey Li
15	Modèles de rendu	arsho , atayanel , Celeo , fabioqcorreia , Jon Chan , MikeC
16	Pagination	hdbuster
17	Réorienter	coralvanda

18	Sessions	arsho , PsyKzz , this-vidor
19	Téléchargement de fichier	davidism , sigmasum
20	Travailler avec JSON	bakkal , g3rv4
21	Vues basées sur la classe	ettanany