

# Blueprints et vues

Une fonction de vue est le code que vous écrivez pour répondre aux requêtes adressées à votre application. Flask utilise des modèles pour faire correspondre l'URL de la requête entrante à la vue qui doit la traiter. La vue renvoie des données que Flask transforme en une réponse sortante. Flask peut également aller dans l'autre sens et générer une URL vers une vue basée sur son nom et ses arguments.

## Créer un *blueprint* ¶

Un **Blueprint** est un moyen d'organiser un groupe de vues et d'autres codes connexes. Plutôt que d'enregistrer des vues et d'autres codes directement dans une application, ils sont enregistrés dans un *blueprint*. Le *blueprint* est ensuite enregistré auprès de l'application lorsqu'il est disponible dans la fonction de fabrique.

Flaskr aura deux *blueprints*, un pour les fonctions d'authentification et un pour les fonctions des articles de blog. Le code de chaque *blueprint* sera placé dans un module séparé. Puisque le blog a besoin de connaître l'authentification, vous écrirez le module d'authentification en premier.

flaskr/auth.py

```
import functools

from flask import (
    Blueprint, flash, g, redirect, render_template, request, session, url_for
)
from werkzeug.security import check_password_hash, generate_password_hash

from flaskr.db import get_db

bp = Blueprint('auth', __name__, url_prefix='/auth')
```

Ceci crée un **Blueprint** nommé 'auth'. Comme l'objet application, le *blueprint* doit savoir où il est défini, donc `__name__` est passé comme deuxième argument. Le préfixe `url_prefix` sera ajouté à toutes les URLs associées au *blueprint*.

Importez et enregistrez le *blueprint* depuis la fabrique en utilisant **`app.register_blueprint()`**. Placez le nouveau code à la fin de la fonction de la fabrique avant de retourner l'application.

flaskr/\_\_init\_\_.py

```
def create_app():
    app = ...
    # existing code omitted
```

 v: latest ▾

```
from . import auth
app.register_blueprint(auth.bp)

return app
```

Le *blueprint* d'authentification comportera des vues permettant d'enregistrer de nouveaux utilisateurs, de se connecter et de se déconnecter.

## La première vue : S'inscrire

Lorsque l'utilisateur visite l'URL `/auth/register`, la vue `register` renvoie du [HTML](#) avec un formulaire qu'il doit remplir. Lorsqu'il soumettra le formulaire, il validera son entrée et soit affichera à nouveau le formulaire avec un message d'erreur, soit créera le nouvel utilisateur et ira à la page de connexion.

Pour l'instant, vous allez juste écrire le code de la vue. À la page suivante, vous écrirez des modèles pour générer le formulaire HTML.

flaskr/auth.py

```
@bp.route('/register', methods=('GET', 'POST'))
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None

        if not username:
            error = 'Username is required.'
        elif not password:
            error = 'Password is required.'
        elif db.execute(
            'SELECT id FROM user WHERE username = ?', (username,)
        ).fetchone() is not None:
            error = f"User {username} is already registered."

        if error is None:
            db.execute(
                'INSERT INTO user (username, password) VALUES (?, ?)',
                (username, generate_password_hash(password))
            )
            db.commit()
            return redirect(url_for('auth.login'))

    flash(error)

    return render_template('auth/register.html')
```

 v: latest ▾

Voici ce que fait la fonction de vue `register` :

1. `@bp.route` associe l'URL `/register` à la fonction de vue `register`. Lorsque Flask reçoit une requête vers `/auth/register`, il appelle la vue `register` et utilise la valeur de retour comme réponse.
2. Si l'utilisateur a soumis le formulaire, `request.method` sera `'POST'`. Dans ce cas, commencer à valider l'entrée.
3. `request.form` est un type spécial de `dict` mettant en correspondance les clés et les valeurs du formulaire soumis. L'utilisateur saisira son `username` et son `password`.
4. Valider que `username` et `password` ne sont pas vides.
5. Valider que `username` n'est pas déjà enregistré en interrogeant la base de données et en vérifiant si un résultat est retourné. `db.execute` prend une requête SQL avec des espaces réservés `?` pour toute entrée utilisateur, et un tuple de valeurs pour remplacer ces espaces réservés. La bibliothèque de base de données se chargera de l'échappement des valeurs afin que vous ne soyez pas vulnérable à une *attaque par injection SQL*.

`fetchone()` renvoie une ligne de la requête. Si la requête n'a donné aucun résultat, elle renvoie `None`. Plus tard, on utilise `fetchall()`, qui renvoie une liste de tous les résultats.

6. Si la validation réussit, insérer les nouvelles données de l'utilisateur dans la base de données. Pour des raisons de sécurité, les mots de passe ne doivent jamais être stockés directement dans la base de données. Au lieu de cela, `generate_password_hash()` est utilisé pour hacher de manière sécurisée le mot de passe, et ce hash est stocké. Comme cette requête modifie des données, `db.commit()` doit être appelé ensuite pour enregistrer les modifications.
7. Après avoir enregistré l'utilisateur, il est redirigé vers la page de connexion. `url_for()` génère l'URL de la vue de connexion en fonction de son nom. C'est préférable à l'écriture directe de l'URL car cela vous permet de changer l'URL plus tard sans modifier tout le code qui y est lié. `redirect()` génère une réponse de redirection vers l'URL générée.
8. Si la validation échoue, l'erreur est affichée à l'utilisateur. `flash()` stocke les messages qui peuvent être récupérés lors du rendu du modèle.
9. Lorsque l'utilisateur navigue initialement vers `auth/register`, ou qu'il y a une erreur de validation, une page HTML avec le formulaire d'enregistrement devrait être affichée. `render_template()` rendra un modèle contenant le HTML, que vous écrirez dans la prochaine étape du tutoriel.

## Connexion

 v: latest ▾

Cette vue suit le même modèle que la vue `register` ci-dessus.

flaskr/auth.py

```

@bp.route('/login', methods=('GET', 'POST'))
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        db = get_db()
        error = None
        user = db.execute(
            'SELECT * FROM user WHERE username = ?', (username,)
        ).fetchone()

        if user is None:
            error = 'Incorrect username.'
        elif not check_password_hash(user['password'], password):
            error = 'Incorrect password.'

        if error is None:
            session.clear()
            session['user_id'] = user['id']
            return redirect(url_for('index'))

        flash(error)

    return render_template('auth/login.html')

```

Il y a quelques différences par rapport à la vue `register` :

1. L'utilisateur est d'abord interrogé et stocké dans une variable pour une utilisation ultérieure.
2. `check_password_hash()` hache le mot de passe soumis de la même manière que le hash stocké et les compare de manière sécurisée. S'ils correspondent, le mot de passe est valide.
3. La `session` est un `dict` qui stocke les données entre les requêtes. Lorsque la validation réussit, l'id de l'utilisateur est stocké dans une nouvelle session. Les données sont stockées dans un *cookie* qui est envoyé au navigateur, et le navigateur le renvoie ensuite avec les requêtes suivantes. Flask signe les données de manière sécurisée afin qu'elles ne puissent pas être modifiées.

Maintenant que l'identifiant de l'utilisateur est stocké dans la `session`, il sera disponible lors des requêtes suivantes. Au début de chaque requête, si un utilisateur est connecté, ses informations doivent être chargées et mises à la disposition des autres vues.

flaskr/auth.py

```

@bp.before_app_request
def load_logged_in_user():
    user_id = session.get('user_id')

    if user_id is None:
        g.user = None
    else:

```

 v: latest ▾

```
g.user = get_db().execute(
    'SELECT * FROM user WHERE id = ?', (user_id,)
).fetchone()
```

`bp.before_app_request()` enregistre une fonction qui s'exécute avant la fonction de vue, quelle que soit l'URL demandée. `load_logged_in_user` vérifie si un id d'utilisateur est stocké dans la `session` et récupère les données de cet utilisateur depuis la base de données, en les stockant sur `g.user`, qui dure le temps de la requête. S'il n'y a pas d'id utilisateur, ou si l'id n'existe pas, `g.user` sera `None`.

## Déconnexion

Pour vous déconnecter, vous devez supprimer l'identifiant de l'utilisateur de la session `session`. Ensuite, `load_logged_in_user` ne chargera pas un utilisateur lors des requêtes suivantes.

flaskr/auth.py

```
@bp.route('/logout')
def logout():
    session.clear()
    return redirect(url_for('index'))
```

## Exiger l'authentification dans d'autres vues


Pour créer, modifier et supprimer des articles de blog, l'utilisateur doit être connecté. Un *décorateur* peut être utilisé pour vérifier cela pour chaque vue à laquelle il est appliqué.

flaskr/auth.py

```
def login_required(view):
    @functools.wraps(view)
    def wrapped_view(**kwargs):
        if g.user is None:
            return redirect(url_for('auth.login'))

        return view(**kwargs)

    return wrapped_view
```

Ce décorateur renvoie une nouvelle fonction de vue qui englobe la vue originale à laquelle il est appliqué. La nouvelle fonction vérifie si l'utilisateur existe ou alors redirige vers la page de connexion. Si un utilisateur est défini, la vue originale est appelée et continue norm  v: latest ▼  
Vous utiliserez ce décorateur lors de l'écriture des vues du blog.

# Points de terminaison et URLs

La fonction `url_for()` génère l'URL d'une vue à partir d'un nom et d'arguments. Le nom associé à une vue est également appelé le *point de terminaison*, et par défaut, il est identique au nom de la fonction de vue.

Par exemple, la vue `hello()` qui a été ajoutée à la fabrique d'application plus tôt dans le tutoriel a le nom `'hello'` et peut être liée avec `url_for('hello')`. Si elle prenait un argument, ce que vous verrez plus tard, elle serait liée en utilisant `url_for('hello', who='World')`.

Lorsque vous utilisez un *blueprint*, le nom du *blueprint* est ajouté au nom de la fonction. Ainsi, le point de terminaison de la fonction `login` que vous avez écrite ci-dessus est `auth.login` car vous l'avez ajoutée au *blueprint* `auth`.

Continuer vers [Modèles](#).