#### // Tutorial //

# Comment créer une application web en utilisant Flask en Python 3

Published on May 13, 2020

* [Python](https://www.digitalocean.com/community/tags/python)
* [Python Frameworks](https://www.digitalocean.com/community/tags/python-frameworks)
* [Programming Project](https://www.digitalocean.com/community/tags/project)
* [Development](https://www.digitalocean.com/community/tags/development)



By [Abdelhadi Dyouri](https://www.digitalocean.com/community/users/adyouri)

Français



L’auteur a choisi le [*Free and Open Source Fund*](https://www.brightfunds.org/funds/foss-nonprofits) pour recevoir un don dans le cadre du programme [*Write for Donations*](https://do.co/w4do-cta).

### [Introduction](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#introduction)

[Flask](http://flask.pocoo.org/) est un petit framework web Python léger, qui fournit des outils et des fonctionnalités utiles qui facilitent la création d’applications web en Python.  Il offre aux développeurs une certaine flexibilité et constitue un cadre plus accessible pour les nouveaux développeurs puisque vous pouvez construire rapidement une application web en utilisant un seul fichier Python. Flask est également extensible et ne force pas une structure de répertoire particulière ou ne nécessite pas de code standard compliqué avant de commencer.

Dans le cadre de ce tutoriel, vous utiliserez la [boîte à outils Bootstrap](https://getbootstrap.com/) pour styliser votre application afin qu’elle soit plus attrayante visuellement. Bootstrap vous aidera à incorporer des pages web réactives dans votre application web afin qu’elle fonctionne également bien sur les navigateurs mobiles sans avoir à écrire votre propre code HTML, CSS et JavaScript pour atteindre ces objectifs. La boîte à outils vous permettra de vous concentrer sur l’apprentissage du fonctionnement de Flask.

Flask utilise le [moteur de modèle Jinja](http://jinja.palletsprojects.com/) pour construire dynamiquement des pages HTML en utilisant des concepts Python familiers tels que les variables, les boucles, les listes, etc. Vous utiliserez ces modèles dans le cadre de ce projet.

Dans ce tutoriel, vous allez construire un petit blog web en utilisant Flask et [SQLite](https://sqlite.org/) en Python 3. Les utilisateurs de l’application peuvent consulter tous les articles de votre base de données et cliquer sur le titre d’un article pour en voir le contenu, avec la possibilité d’ajouter un nouvel article à la base de données et de modifier ou supprimer un article existant.

## [Conditions préalables](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#conditions-prealables)

Avant de commencer à suivre ce guide, vous aurez besoin des éléments suivants :

* Un environnement de programmation Python 3 local, suivez le tutoriel pour votre distribution dans les séries [Comment installer et configurer un environnement de programmation local pour Python 3](https://www.digitalocean.com/community/tutorial_series/how-to-install-and-set-up-a-local-programming-environment-for-python-3) pour votre machine locale. Dans ce tutoriel, nous appellerons notre répertoire de projet flask\_blog.
* Une compréhension des concepts de Python 3, tels que les [types de données](https://www.digitalocean.com/community/tutorials/understanding-data-types-in-python-3), les [instructions conditionnelles](https://www.digitalocean.com/community/tutorials/how-to-write-conditional-statements-in-python-3-2), [pour les boucles](https://www.digitalocean.com/community/tutorials/how-to-construct-for-loops-in-python-3), les [fonctions](https://www.digitalocean.com/community/tutorials/how-to-define-functions-in-python-3) et autres concepts de ce type. Si vous n’êtes pas familiarisé avec Python, consultez notre série [Comment coder en Python 3](https://www.digitalocean.com/community/tutorial_series/how-to-code-in-python-3).

## [Étape 1 – Installation de Flask](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-1-installation-de-flask)

Dans cette étape, vous allez activer votre environnement Python et installer Flask en utilisant l’installateur de paquets [pip](https://pypi.org/project/pip/).

Si vous n’avez pas encore activé votre environnement de programmation, assurez-vous que vous êtes dans le répertoire de votre projet (flask\_blog) et utilisez la commande suivante pour activer l’environnement :

1. source env/bin/activate

Copy

Une fois que votre environnement de programmation est activé, votre invite aura maintenant un préfixe env qui peut ressembler à ce qui suit :

Copy

Ce préfixe est une indication que l’environnement env est actuellement actif, qui peut avoir un autre nom selon la façon dont vous l’avez nommé lors de sa création.

**Remarque :** vous pouvez utiliser [Git](https://git-scm.com/), un système de contrôle de version pour gérer et suivre efficacement le processus de développement de votre projet. Pour apprendre à utiliser Git, vous pouvez consulter notre article [Introduction à l’installation, l’utilisation et les branches de Git](https://www.digitalocean.com/community/tutorial_series/introduction-to-git-installation-usage-and-branches).

Si vous utilisez Git, il est conseillé d’ignorer le répertoire env nouvellement créé dans votre fichier .gitignore pour éviter de suivre des fichiers non liés au projet.

Vous allez maintenant installer des paquets Python et isoler le code de votre projet de l’installation principale du système Python. Vous ferez cela en utilisant pip et python.

Pour installer Flask, exécutez la commande suivante :

1. pip install flask

Copy

Une fois l’installation terminée, exécutez la commande suivante pour confirmer l’installation :

1. python -c "import flask; print(flask.\_\_version\_\_)"

Copy

Vous utilisez l’[interface en ligne de commande python](https://docs.python.org/3/using/cmdline.html) avec l’option -c pour exécuter le code Python. Ensuite, vous importez le paquet flask avec import flask; puis vous imprimez la version de Flask, qui est fournie via la variable flask.\_\_version\_\_.

La sortie sera un numéro de version semblable à ce qui suit :

Output

1.1.2

Vous avez créé le dossier du projet, un environnement virtuel et installé Flask. Vous êtes maintenant prêt à configurer votre application de base.

## [Étape 2 – Créer une application de base](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-2-creer-une-application-de-base)

Maintenant que vous avez configuré votre environnement de programmation, vous allez commencer à utiliser Flask. Dans cette étape, vous allez créer une petite application web dans un fichier Python et l’exécuter pour démarrer le serveur, qui affichera certaines informations sur le navigateur.

Dans votre répertoire flask\_blog, ouvrez un fichier nommé hello.py pour le modifier, utilisez nano ou votre éditeur de texte préféré :

1. nano hello.py

Copy

Ce fichier hello.py servira d’exemple minimal de la manière de traiter les requêtes HTTP. À l’intérieur de ce fichier, vous importerez l’[objet Flask](https://flask.palletsprojects.com/en/1.1.x/api/#flask.Flask) et créerez une fonction qui renvoie une réponse HTTP.  Écrivez le code suivant à l’intérieur hello.py :

flask\_blog/hello.py

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello():

return 'Hello, World!'

Copy

Dans le bloc de code précédent, vous importez d’abord l’objet Flask du paquet flask. Vous l’utilisez ensuite pour créer votre instance d’application Flask avec le nom app. Vous passez la variable spéciale \_\_name\_\_ qui contient le nom du module Python actuel.  Il est utilisé pour indiquer à l’instance où elle se trouve - vous en avez besoin car Flask établit certains chemins en coulisses.

Une fois que vous avez créé l’instance de l'application, vous l’utilisez pour traiter les demandes web entrantes et envoyer des réponses à l’utilisateur. @app.route est un [décorateur](https://en.wikipedia.org/wiki/Python_syntax_and_semantics#Decorators) qui transforme une fonction Python ordinaire en une fonction d’affichage Flask, qui convertit la valeur de retour de la fonction en une réponse HTTP à afficher par un client HTTP, tel qu’un navigateur web. Vous passez à la valeur '/' à @app.route() pour indiquer que cette fonction répondra aux requêtes web pour l’URL /, qui est l’URL principale.

La fonction d’affichage hello() renvoie la chaîne "Hello, World!" comme réponse.

Enregistrez et fermez le fichier.

Pour exécuter votre application web, vous devez d’abord indiquer à Flask où trouver l’application (le fichier hello.py dans votre cas) avec la variable d’environnement FLASK\_APP :

1. export FLASK\_APP=hello

Copy

Ensuite, exécutez-la en mode développement avec la variable d’environnement FLASK\_ENV :

1. export FLASK\_ENV=development

Copy

Pour terminer, exécutez l’application en utilisant la commande flask run :

1. flask run

Copy

Une fois l’application lancée, la sortie sera quelque chose comme ceci :

Output

\* Serving Flask app "hello" (lazy loading)

\* Environment: development

\* Debug mode: on

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

\* Restarting with stat

\* Debugger is active!

\* Debugger PIN: 813-894-335

La sortie précédente contient plusieurs éléments d’information, tels que :

* Le nom de l’application que vous exécutez.
* L’environnement dans lequel l’application est exécutée.
* Debug mode: onsignifie que le débogueur Flask est en cours d’exécution. Ceci est utile lors du développement car il nous donne des messages d’erreur détaillés lorsque les choses tournent mal, ce qui facilite le dépannage.
* L’application s’exécute localement sur l’URL http://127.0.0.1:5000/, 127.0.0.1 est l’IP qui représente le localhost de votre machine et :5000 est le numéro du port.

Ouvrez un navigateur et tapez l’URL http://127.0.0.1:5000/, vous recevrez la chaîne de caractères Hello, World! en réponse, cela confirme que votre application fonctionne correctement.

**Warning** Flask utilise un simple serveur web pour servir notre application dans un environnement de développement, ce qui signifie également que le débogueur Flask est en cours d’exécution pour faciliter la capture des erreurs. Ce serveur de développement ne doit pas être utilisé dans un déploiement de production. Consultez la page des [options de déploiement](https://flask.palletsprojects.com/en/1.1.x/deploying/) dans la documentation Flask pour plus d’informations, vous pouvez également consulter ce [tutoriel sur le déploiement de Flask](https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-18-04).

Vous pouvez maintenant quitter le serveur de développement en cours d’exécution dans le terminal et ouvrir une autre fenêtre du terminal. Déplacez-vous dans le dossier du projet où hello.py se trouve, activez l’environnement virtuel, définissez les variables d’environnement FLASK\_ENV et FLASK\_APP, et passez aux étapes suivantes. (Ces commandes sont énumérées plus haut dans cette étape).

**Remarque** : lors de l’ouverture d’un nouveau terminal, il est important de se souvenir d’activer l’environnement virtuel et de définir les variables d’environnement FLASK\_ENV et FLASK\_APP.

Pendant que le serveur de développement d’une application Flask est déjà en cours d’exécution, il n’est pas possible d’exécuter une autre application Flask avec la même commande flask run. En effet, flask run utilise le numéro de port 5000 par défaut, et une fois qu’il est pris, il devient indisponible pour exécuter une autre application de sorte que vous recevriez une erreur similaire à la suivante :

Output

OSError: [Errno 98] Address already in use

Pour résoudre ce problème, vous pouvez soit arrêter le serveur qui tourne actuellement via CTRL+C, puis relancer flask run, soit, si vous voulez exécuter les deux en même temps, vous pouvez passer un numéro de port différent à l’argument -p; par exemple, pour exécuter une autre application sur le port 5001, utilisez la commande suivante :

1. flask run -p 5001

Copy

Vous disposez maintenant d’une petite application web Flask. Vous avez lancé votre application et affiché des informations sur le navigateur web. Ensuite, vous allez utiliser des fichiers HTML dans votre application.

## [Étape 3 – Utilisation des modèles HTML](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-3-utilisation-des-modeles-html)

Actuellement, votre application n’affiche qu’un simple message sans aucun HTML. Les applications web utilisent principalement le HTML pour afficher des informations à l’intention du visiteur. Vous allez donc maintenant travailler à l’incorporation de fichiers HTML à votre application, qui pourront être affichés sur le navigateur web.

Flask fournit une fonction d’aide render\_template() qui permet l’utilisation du [moteur de modèle Jinja](http://jinja.pocoo.org/). Cela facilitera grandement la gestion du HTML en écrivant votre code HTML dans des fichiers .html et en utilisant la logique dans votre code HTML. Vous allez utiliser ces fichiers HTML (modèles) pour construire toutes les pages de votre application, comme la page principale où vous afficherez les articles de blog en cours, la page de l’article de blog, la page où l’utilisateur peut ajouter un nouvel article, etc.

Dans cette étape, vous allez créer votre application Flask principale dans un nouveau fichier.

Tout d’abord, dans votre répertoire flask\_blog, utilisez nano ou votre éditeur préféré pour créer et modifier votre fichier app.py. Ce fichier contiendra tout le code que vous utiliserez pour créer l’application de blog :

1. nano app.py

Copy

Dans ce nouveau fichier, vous allez importer l’objet Flask pour créer une instance d’application Flask comme vous l’avez déjà fait. Vous importerez également la fonction d’aide [render\_template()](https://flask.palletsprojects.com/en/1.1.x/api/#flask.render_template) qui vous permet de rendre les fichiers de modèles HTML qui existent dans le dossier des modèles que vous êtes en train de créer. Le fichier aura une fonction d’affichage unique qui sera responsable de la gestion des demandes vers la principale route /. Ajoutez le contenu suivant :

flask\_blog/app.py

from flask import Flask, render\_template

app = Flask(\_\_name\_\_)

@app.route('/')

def index():

return render\_template('index.html')

Copy

La fonction d’affichage index() renvoie le résultat de l’appel de render\_template() avec index.html en argument, ce qui indique à render\_template() qu’il doit rechercher un fichier appelé index.html dans le dossier des modèles. Le dossier et le fichier n’existent pas encore, vous obtiendrez une erreur si vous deviez exécuter l’application à ce stade. Vous l’exécuterez néanmoins afin de vous familiariser avec cette exception couramment rencontrée. Vous la corrigerez ensuite en créant le dossier et le fichier nécessaires.

Enregistrez et quittez le fichier.

Arrêtez le serveur de développement dans votre autre terminal qui exécute l’application hello avec CTRL+C.

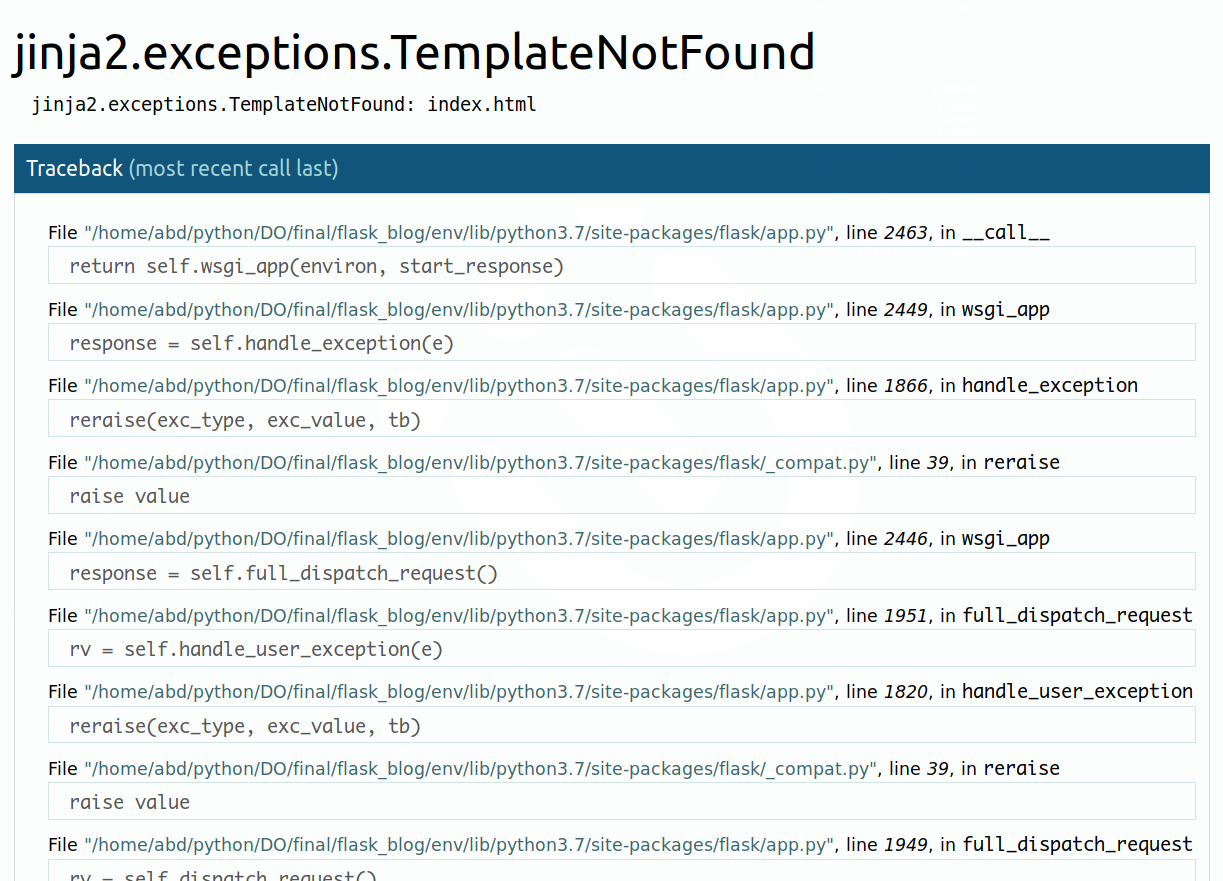
Avant d’exécuter l’application, assurez-vous de spécifier correctement la valeur de la variable d’environnement FLASK\_APP, puisque vous n’utilisez plus l’application hello :

1. export FLASK\_APP=app
2. flask run

Copy

En ouvrant l’URL http://127.0.0.1:5000/ dans votre navigateur, la page du débogueur vous informera que le modèle index.html n’a pas été trouvé. La ligne principale du code responsable de cette erreur sera mise en évidence. Dans ce cas, il s’agit de la ligne return render\_template('index.html').

Si vous cliquez sur cette ligne, le débogueur vous révélera plus de code afin que vous disposiez de plus de contexte pour vous aider à résoudre le problème.



Pour corriger cette erreur, créez un répertoire appelé templates à l’intérieur de votre répertoire flask\_blog. Ensuite, à l’intérieur de ce répertoire, ouvrez un fichier appelé index.html pour l’éditer :

1. mkdir templates
2. nano templates/index.html

Copy

Ensuite, ajoutez le code HTML suivant dans index.html :

flask\_blog/templates/index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>FlaskBlog</title>

</head>

<body>

<h1>Welcome to FlaskBlog</h1>

</body>

</html>

Copy

Enregistrez le fichier et utilisez votre navigateur pour naviguer à nouveau sur http://127.0.0.1:5000/, ou actualisez la page. Cette fois, le navigateur doit afficher le texte Welcome to FlaskBlog dans une balise <h1>.

En plus du dossier templates, les applications web de Flask disposent généralement d’un dossier static pour l’hébergement des fichiers statiques, tels que les fichiers CSS, les fichiers JavaScript et les images utilisées par l’application.

Vous pouvez créer un fichier de feuille de style style.css pour ajouter du CSS à votre application. Tout d’abord, créez un répertoire appelé static à l’intérieur de votre répertoire principal flask\_blog :

1. mkdir static

Copy

Ensuite, créez un autre répertoire appelé css à l’intérieur du répertoire static pour héberger les fichiers .css. Cela est généralement fait pour organiser les fichiers statiques dans des dossiers dédiés, comme par exemple, les fichiers JavaScript se trouvent généralement dans un répertoire appelé js, les images sont placées dans un répertoire appelé images (ou img), etc. La commande suivante créera le répertoire css à l’intérieur du répertoire static :

1. mkdir static/css

Copy

Ouvrez ensuite un fichier style.css à l’intérieur du répertoire css pour l’éditer :

1. nano static/css/style.css

Copy

Ajoutez la règle CSS suivante à votre fichier style.css :

flask\_blog/static/css/style.css

h1 {

border: 2px #eee solid;

color: brown;

text-align: center;

padding: 10px;

}

Copy

Le code CSS ajoutera une bordure, changera la couleur en marron, centrera le texte et ajoutera un peu de padding aux balises <h1>.

Enregistrez et fermez le fichier.

Ensuite, ouvrez le fichier de modèle index.html pour le modifier :

1. nano templates/index.html

Copy

Vous ajouterez un lien vers le fichier style.css à l’intérieur de la section <head> du fichier modèle index.html :

flask\_blog/templates/index.html

. . .

<head>

<meta charset="UTF-8">

<link rel="stylesheet" href="{{ url\_for('static', filename= 'css/style.css') }}">

<title>FlaskBlog</title>

</head>

. . .

Copy

Ici, vous utilisez la fonction d’aide [url\_for()](https://flask.palletsprojects.com/en/1.1.x/api/#flask.url_for) pour générer l’emplacement approprié du fichier. Le premier argument spécifie que vous créez un lien vers un fichier statique et que le deuxième argument est le chemin du fichier à l’intérieur du répertoire static.

Enregistrez et fermez le fichier.

En rafraîchissant la page d’index de votre application, vous remarquerez que le texte Welcome to FlaskBlog est maintenant en brun, centré et entouré d’une bordure.

Vous pouvez utiliser le langage CSS pour styliser l’application et la rendre plus attrayante en utilisant votre propre design. Toutefois, si vous n’êtes pas un concepteur de sites web ou si vous ne connaissez pas le langage CSS, vous pouvez utiliser la [boîte à outils Bootstrap](https://getbootstrap.com/), qui fournit des composants faciles à utiliser pour styliser votre application. Dans ce projet, nous utiliserons Bootstrap.

Vous pourriez penser que la création d’un autre modèle HTML impliquerait la répétition de la plupart du code HTML que vous avez déjà écrit dans le modèle index.html. Vous pouvez éviter la répétition inutile du code à l’aide d’un fichier de modèle de base, dont tous vos fichiers HTML hériteront. Pour plus d’informations, voir la section [Héritage de modèles dans Jinja](https://jinja.palletsprojects.com/en/2.10.x/templates/#template-inheritance).

Pour créer un modèle de base, créez d’abord un fichier appelé base.html dans votre répertoire templates :

1. nano templates/base.html

Copy

Tapez le code suivant dans votre modèle base.html :

flask\_blog/templates/base.html

<!doctype html>

<html lang="en">

<head>

<!-- Required meta tags -->

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

<!-- Bootstrap CSS -->

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="anonymous">

<title>{% block title %} {% endblock %}</title>

</head>

<body>

<nav class="navbar navbar-expand-md navbar-light bg-light">

<a class="navbar-brand" href="{{ url\_for('index')}}">FlaskBlog</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item active">

<a class="nav-link" href="#">About</a>

</li>

</ul>

</div>

</nav>

<div class="container">

{% block content %} {% endblock %}

</div>

<!-- Optional JavaScript -->

<!-- jQuery first, then Popper.js, then Bootstrap JS -->

<script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

</body>

</html>

Copy

Enregistrez et fermez le fichier une fois que vous avez terminé de le modifier.

La majeure partie du code du bloc précédent est du HTML standard et du code requis pour Bootstrap. Les balises <meta> fournissent des informations pour le navigateur web, la balise <link> relie les fichiers CSS de Bootstrap, et les balises <script> sont des liens vers le code JavaScript qui permet certaines fonctionnalités supplémentaires de Bootstrap, consultez la [documentation de Bootstrap](https://getbootstrap.com/) pour en savoir plus.

Cependant, les parties surlignées suivantes sont spécifiques au moteur de template Jinja :

* {% block title %} {% endblock %}​​​ : un [bloc](https://jinja.palletsprojects.com/en/2.10.x/templates/#blocks) qui sert d’emplacement pour un titre, vous l’utiliserez plus tard dans d’autres modèles pour donner un titre personnalisé à chaque page de votre application sans réécrire à chaque fois toute la section <head>.
* {{ url\_for('index')}}: un appel de fonction qui renvoie l’URL pour la fonction d’affichage index(). Il est différent de l’ancien appel url\_for() que vous utilisiez pour lier un fichier CSS statique, car il ne prend qu’un seul argument, qui est le nom de la fonction de visualisation, et renvoie à la route associée à la fonction au lieu d’un fichier statique.
* {% block content %} {% endblock %}: un autre bloc qui sera remplacé par un contenu dépendant du modèle enfant (modèles qui héritent de base.html) qui le remplacera.

Maintenant que vous avez un modèle de base, vous pouvez en tirer profit grâce à l’héritage. Ouvrez le fichier index.html :

1. nano templates/index.html

Copy

Ensuite, remplacez son contenu par le suivant :

flask\_blog/templates/index.html

{% extends 'base.html' %}

{% block content %}

<h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>

{% endblock %}

Copy

Dans cette nouvelle version du modèle index.html, vous utilisez la balise {% extends %} pour hériter du modèle base.html. Vous l’étendez ensuite en remplaçant le bloc de contenu dans le modèle de base par ce qui se trouve à l’intérieur du bloc de contenu dans le bloc de code précédent.

Ce bloc de contenu contient une balise <h1> avec le texte Welcome to FlaskBlog à l’intérieur d’un bloc de titre, qui à son tour remplace le bloc de titre original dans le modèle base.html par le texte Welcome to FlaskBlog. De cette façon, vous pouvez éviter de répéter deux fois le même texte, car il fonctionne à la fois comme un titre pour la page et un titre qui apparaît sous la barre de navigation héritée du modèle de base.

L’héritage du modèle vous donne également la possibilité de réutiliser le code HTML que vous avez dans d’autres modèles (base.html dans ce cas) sans avoir à le répéter chaque fois que nécessaire.

Enregistrez et fermez le fichier et rafraîchissez la page d’index sur votre navigateur. Vous verrez votre page avec une barre de navigation et un titre stylisé.



Vous avez utilisé des modèles HTML et des fichiers statiques dans Flask. Vous avez également utilisé Bootstrap pour commencer à affiner l’apparence de votre page et un modèle de base pour éviter la répétition du code. Dans l’étape suivante, vous allez configurer une base de données qui va stocker les données de votre application.

## [Étape 4 – Mise en place de la base de données](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-4-mise-en-place-de-la-base-de-donnees)

Au cours de cette étape, vous allez créer une base de données pour stocker les données, c’est-à-dire les articles de blog de votre application. Vous allez également remplir la base de données avec quelques exemples d’entrées.

Vous utiliserez un fichier de [base de données SQLite](https://sqlite.org/) pour stocker vos données, car le module [sqlite3](https://docs.python.org/3/library/sqlite3.html), que nous utiliserons pour interagir avec la base de données, est disponible dans la bibliothèque standard Python. Pour plus d’informations sur SQLite, consultez [ce tutoriel](https://www.digitalocean.com/community/tutorials/how-and-when-to-use-sqlite).

Tout d’abord, comme les données dans SQLite sont stockées dans des tableaux et des colonnes, et comme vos données consistent principalement en des articles de blog, vous devez d’abord créer un tableau appelé posts (articles) avec les colonnes nécessaires. Vous allez créer un fichier .sql qui contient des commandes SQL pour créer le tableau posts avec quelques colonnes. Vous utiliserez ensuite ce fichier pour créer la base de données.

Ouvrez un fichier appelé schema.sql à l’intérieur de votre répertoire flask\_blog :

1. nano schema.sql

Copy

Tapez les commandes SQL suivantes dans ce fichier :

flask\_blog/schema.sql

DROP TABLE IF EXISTS posts;

CREATE TABLE posts (

id INTEGER PRIMARY KEY AUTOINCREMENT,

created TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP,

title TEXT NOT NULL,

content TEXT NOT NULL

);

Enregistrez et fermez le fichier.

La première commande SQL est DROP TABLE IF EXISTS posts; elle supprime toutes les tables déjà existantes nommées posts afin de ne pas créer de confusion. Notez que cela supprimera tout le contenu que vous avez dans la base de données chaque fois que vous utiliserez ces commandes SQL, donc assurez-vous de ne pas écrire de contenu important dans l’application web avant d’avoir terminé ce tutoriel et d’avoir expérimenté le résultat final. Ensuite, CREATE TABLE posts est utilisé pour créer la table posts avec les colonnes suivantes :

* id : nombre entier qui représente une clé primaire, à laquelle la base de données attribuera une valeur unique pour chaque entrée (c’est-à-dire un billet de blog).
* created : l’heure où le billet de blog a été créé. NOT NULL signifie que cette colonne ne doit pas être vide et la valeur DEFAULT est la valeur CURRENT\_TIMESTAMP, qui est le moment où l’article a été ajouté à la base de données. Tout comme id, vous n’avez pas besoin de spécifier une valeur pour cette colonne, car elle sera automatiquement remplie.
* title : le titre de l’article.
* content : le contenu de l’article.

Maintenant que vous avez un schéma SQL dans le fichier schema.sql, vous allez l’utiliserez pour créer la base de données à l’aide d’un fichier Python qui génèrera un fichier de base de données SQLite .db. Ouvrez un fichier nommé init\_db.py dans le répertoire flask\_blog en utilisant votre éditeur préféré :

1. nano init\_db.py

Copy

Ensuite, ajoutez le code suivant.

flask\_blog/init\_db.py

import sqlite3

connection = sqlite3.connect('database.db')

with open('schema.sql') as f:

connection.executescript(f.read())

cur = connection.cursor()

cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",

('First Post', 'Content for the first post')

)

cur.execute("INSERT INTO posts (title, content) VALUES (?, ?)",

('Second Post', 'Content for the second post')

)

connection.commit()

connection.close()

Copy

Vous importez d’abord le module sqlite3 et ouvrez ensuite une connexion à un fichier de base de données nommé base de database.db, qui sera créé une fois que vous aurez lancé le fichier Python. Ensuite, vous utilisez la fonction open() pour ouvrir le fichier schema.sql. Ensuite, vous exécutez son contenu en utilisant la méthode [executescript()](https://docs.python.org/3/library/sqlite3.html#sqlite3.Connection.executescript) qui exécute plusieurs instructions SQL en une seule fois, ce qui créera la table posts. Vous créez un [objet Cursor](https://docs.python.org/3/library/sqlite3.html#cursor-objects) qui vous permet d’utiliser sa méthode [execute()](https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.execute) pour exécuter deux instructions SQL INSERT afin d’ajouter deux articles de blog à votre table posts.  Enfin, vous validez les modifications et coupez la connexion.

Enregistrez et fermez le fichier et exécutez-le dans le terminal en utilisant la commande python :

1. python init\_db.py

Copy

Une fois l’exécution du fichier terminée, un nouveau fichier appelé database.db apparaîtra dans votre répertoire flask\_blog. Cela signifie que vous avez configuré avec succès votre base de données.

Dans l’étape suivante, vous récupérerez les articles de blog que vous avez insérés dans votre base de données et les afficherez sur la page d’accueil de votre application.

## [Étape 5 – Afficher tous les articles](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-5-afficher-tous-les-articles)

Maintenant que vous avez configuré votre base de données, vous pouvez modifier la fonction d’affichage index() pour afficher tous les articles que vous avez dans votre base de données.

Ouvrez le fichier app.py pour apporter les modifications suivantes :

1. nano app.py

Copy

Pour votre première modification, vous importerez le module sqlite3 en haut du fichier :

flask\_blog/app.py

import sqlite3

from flask import Flask, render\_template

. . .

Copy

Ensuite, vous allez créer une fonction qui crée une connexion à la base de données et la renvoie. Ajoutez-la directement après l’importation :

flask\_blog/app.py

. . .

from flask import Flask, render\_template

def get\_db\_connection():

conn = sqlite3.connect('database.db')

conn.row\_factory = sqlite3.Row

return conn

. . .

Copy

Cette fonction get\_db\_connection() ouvre une connexion au fichier de base de données database.db, et définit ensuite l’attribut [row\_factory](https://docs.python.org/3/library/sqlite3.html#sqlite3.Connection.row_factory) à sqlite3. Row afin que vous puissiez avoir un accès par nom aux colonnes. Cela signifie que la connexion à la base de données renverra des lignes qui se comporteront comme des dictionnaires Python normaux. Enfin, la fonction renvoie l’objet de connexion conn que vous utiliserez pour accéder à la base de données.

Après avoir défini la fonction get\_db\_connection(), modifier la fonction index() pour qu’elle ressemble à ce qui suit :

flask\_blog/app.py

. . .

@app.route('/')

def index():

conn = get\_db\_connection()

posts = conn.execute('SELECT \* FROM posts').fetchall()

conn.close()

return render\_template('index.html', posts=posts)

Copy

Dans cette nouvelle version de la fonction index(), vous ouvrez d’abord une connexion à la base de données en utilisant la fonction get\_db\_connection() que vous avez définie précédemment. Ensuite, vous exécutez une requête SQL pour sélectionner toutes les entrées de la table posts. Vous implémentez la méthode [fetchall()](https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.fetchall) pour récupérer toutes les lignes du résultat de la requête, celle-ci renvoie une liste des articles que vous avez insérés dans la base de données à l’étape précédente.

Vous fermez la connexion à la base de données en utilisant la méthode close() et vous retournez le résultat du rendu du modèle index.html. Vous passez également l’objet posts en argument, lequel contient les résultats que vous avez obtenus de la base de données, ce qui vous permettra d’accéder aux articles de blog dans le modèle index.html.

Une fois ces modifications apportées, enregistrez et fermez le fichier app.py.

Maintenant que vous avez passé les articles que vous avez extraits de la base de données au modèle index.html, vous pouvez utiliser une [boucle for](https://jinja.palletsprojects.com/en/2.10.x/templates/#for) pour afficher chaque article sur votre page d’index.

Ouvrez le fichier index.html :

1. nano templates/index.html

Copy

Ensuite, modifiez-le pour qu’il ressemble à :

flask\_blog/templates/index.html

{% extends 'base.html' %}

{% block content %}

<h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>

{% for post in posts %}

<a href="#">

<h2>{{ post['title'] }}</h2>

</a>

<span class="badge badge-primary">{{ post['created'] }}</span>

<hr>

{% endfor %}

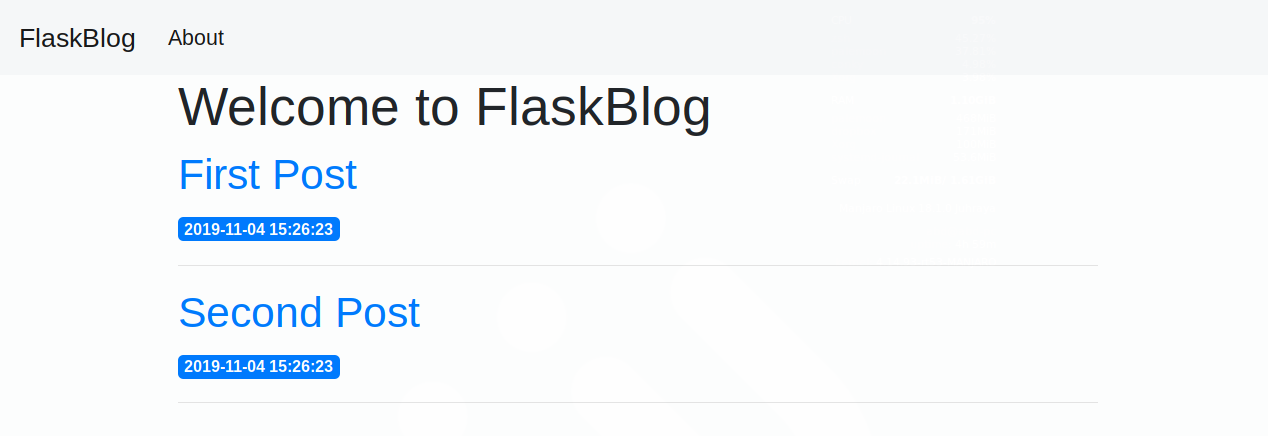
{% endblock %}

Copy

Ici, la syntaxe {% for post in posts %} est un Jinja pour boucle, qui est similaire à un Python pour boucle sauf qu’il doit être fermé plus tard avec la syntaxe {% endfor %}. Vous utilisez cette syntaxe pour boucler sur chaque élément de la liste des messages qui a été passé par la fonction index() dans la ligne return render\_template('index.html', posts=posts). Dans cette boucle for, vous affichez le titre de l’article dans un titre <h2> à l’intérieur d’une balise <a> (vous utiliserez plus tard cette balise pour faire un lien vers chaque article individuellement).

Vous affichez le titre en utilisant un délimiteur de variable littérale ({{ ... }}). N’oubliez pas que post sera un objet de type dictionnaire, vous pouvez donc accéder au titre de l’article avec la balise post['title']. Vous affichez également la date de création l’article en utilisant la même méthode.

Une fois que vous avez fini d’éditer le fichier, enregistrez-le et fermez-le. Ensuite, naviguez vers la page d’index de votre navigateur. Vous verrez les deux articles que vous avez ajoutés à la base de données sur votre page.



Maintenant que vous avez modifié la fonction d’affichage index() pour afficher tous les articles que vous avez dans la base de données sur la page d’accueil de votre application, vous allez afficher chaque article sur une seule page et permettre aux utilisateurs de se relier à chaque article.

## [Étape 6 – Afficher un article unique](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-6-afficher-un-article-unique)

Au cours de cette étape, vous allez créer un nouvel itinéraire Flask avec une fonction d’affichage et un nouveau modèle HTML pour afficher un article de blog individuel par son ID.

À la fin de cette étape, l’URL http://127.0.0.1:5000/1 sera une page qui affichera le premier article (parce qu’il a l’ID 1). L’URL http://127.0.0.1:5000/ID affichera l’article avec le numéro ID associé s’il existe.

Ouvrez app.py pour le modifier :

1. nano app.py

Copy

Comme vous devrez obtenir un article de blog par son ID à partir de la base de données à plusieurs endroits plus tard dans ce projet, vous créerez une fonction autonome appelée get\_post(). Vous pouvez l’appeler en lui passant un ID et recevoir en retour le billet associé à l’ID fourni, ou faire en sorte que Flask réponde avec un message 404 Not Found si le billet n’existe pas.

Pour répondre avec une page 404, vous devez importer la fonction [abort()](https://werkzeug.palletsprojects.com/en/0.15.x/exceptions/#werkzeug.exceptions.abort) de la bibliothèque [Werkzeug](https://werkzeug.palletsprojects.com/en/0.15.x/), qui a été installée avec Flask, en haut du fichier :

flask\_blog/app.py

import sqlite3

from flask import Flask, render\_template

from werkzeug.exceptions import abort

. . .

Copy

Ensuite, ajoutez la fonction get\_post() juste après la fonction get\_db\_connection() que vous avez créée à l’étape précédente :

flask\_blog/app.py

. . .

def get\_db\_connection():

conn = sqlite3.connect('database.db')

conn.row\_factory = sqlite3.Row

return conn

def get\_post(post\_id):

conn = get\_db\_connection()

post = conn.execute('SELECT \* FROM posts WHERE id = ?',

(post\_id,)).fetchone()

conn.close()

if post is None:

abort(404)

return post

. . .

Copy

Cette nouvelle fonction a un argument post\_id qui détermine quel article de blog doit être renvoyé.

À l’intérieur de la fonction, vous utilisez la fonction get\_db\_connection() pour ouvrir une connexion à la base de données et exécuter une requête SQL pour obtenir l’article de blog associé à la valeur post\_id donnée. Vous ajoutez la méthode fetchone() pour obtenir le résultat et le stocker dans la variable post, puis fermez la connexion. Si la variable post a la valeur None, ce qui signifie qu’aucun résultat n’a été trouvé dans la base de données, vous utilisez la fonction abort() que vous avez importée précédemment pour répondre avec un code d’erreur 404 et la fonction terminera son exécution. Si toutefois, un article a été trouvé, vous retournez la valeur de la variable post.

Ensuite, ajoutez la fonction d’affichage suivante à la fin du fichier app.py :

flask\_blog/app.py

. . .

@app.route('/<int:post\_id>')

def post(post\_id):

post = get\_post(post\_id)

return render\_template('post.html', post=post)

Copy

Dans cette nouvelle fonction de visualisation, vous ajoutez une [règle de variable](https://flask.palletsprojects.com/en/1.1.x/quickstart/#variable-rules) <int:post\_id> pour spécifier que la partie après la barre oblique (/) est un entier positif (marqué avec le convertisseur int) auquel vous devez accéder dans votre fonction de visualisation. Flask reconnaît cela et passe sa valeur à l’argument du mot-clé post\_id de votre fonction de visualisation post(). Vous utilisez ensuite la fonction get\_post() pour obtenir l’article de blog associé à l’ID spécifié et stocker le résultat dans la variable post, que vous passez à un modèle post.html que vous allez bientôt créer.

Enregistrez le fichier app.py et ouvrez un nouveau fichier de modèle post.html pour le modifier :

1. nano templates/post.html

Copy

Tapez le code suivant dans ce nouveau fichier post.html. Celui-ci sera similaire au fichier index.html, sauf qu’il n’affichera qu’un seul article, en plus d’afficher également le contenu de l’article :

flask\_blog/templates/post.html

{% extends 'base.html' %}

{% block content %}

<h2>{% block title %} {{ post['title'] }} {% endblock %}</h2>

<span class="badge badge-primary">{{ post['created'] }}</span>

<p>{{ post['content'] }}</p>

{% endblock %}

Copy

Vous ajoutez le bloc title que vous avez défini dans le modèle base.html pour que le titre de la page reflète en même temps le titre de l’article qui est affiché dans un titre <h2>.

Enregistrez et fermez le fichier.

Vous pouvez maintenant naviguer vers les URL suivantes pour voir les deux articles que vous avez dans votre base de données, ainsi qu’une page qui indique à l’utilisateur que l’article de blog demandé n’a pas été trouvé (puisqu’il n’y a pas d’article avec un numéro d’identification de 3 pour l’instant) :

http://127.0.0.1:5000/1

http://127.0.0.1:5000/2

http://127.0.0.1:5000/3

En retournant à la page d’index, vous ferez en sorte que le titre de chaque article renvoie à sa page respective. Vous ferez cela en utilisant la fonction url\_for(). Tout d’abord, ouvrez le modèle index.html pour le modifier :

1. nano templates/index.html

Copy

Ensuite, changez la valeur de l’attribut href de # à {{ url\_for('post', post\_id=post['id']) }} pour que la boucle for se présente exactement comme suit :

flask\_blog/templates/index.html

{% for post in posts %}

<a href="{{ url\_for('post', post\_id=post['id']) }}">

<h2>{{ post['title'] }}</h2>

</a>

<span class="badge badge-primary">{{ post['created'] }}</span>

<hr>

{% endfor %}

Copy

Ici, vous passez « post » à la fonction url\_for() comme un premier argument. C’est le nom de la fonction d’affichage post() et comme elle accepte un argument post\_id, vous lui donnez la valeur post['id']. La fonction url\_for() renvoie l’URL appropriée pour chaque article en fonction de son ID.

Enregistrez et fermez le fichier.

Les liens de la page d’index fonctionneront désormais comme prévu. Vous avez maintenant terminé la construction de la partie de l’application responsable de l’affichage des articles de blog dans votre base de données. Ensuite, vous allez ajouter à votre application la possibilité de créer, de modifier et de supprimer des articles de blog.

## [Étape 7 – Modification d’articles](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#etape-7-modification-d-articles)

Maintenant que vous avez fini d’afficher les articles de blog présents dans la base de données sur l’application web, vous devez permettre aux utilisateurs de votre application d’écrire de nouveaux articles de blog et de les ajouter à la base de données, de modifier les articles existants et de supprimer les articles de blog inutiles.

### [Créer un nouvel article](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#creer-un-nouvel-article)

À ce stade, vous disposez d’une application qui affiche les messages dans votre base de données mais qui ne vous permet pas d’ajouter un nouveau message, sauf si vous vous connectez directement à la base de données SQLite et que vous en ajoutez un manuellement. Dans cette section, vous allez créer une page sur laquelle vous pourrez créer un article en fournissant son titre et son contenu.

Ouvrez le fichier app.py pour le modifier :

1. nano app.py

Copy

Tout d’abord, vous allez importer ce qui suit depuis le framework Flask :

* L’objet [request](https://flask.palletsprojects.com/en/1.1.x/api/#flask.request) global pour accéder aux données de la requête entrante qui sera soumise via un formulaire HTML.
* La fonction [url\_for()](https://flask.palletsprojects.com/en/1.1.x/api/#flask.url_for) pour générer des URL.
* La fonction [flash()](https://flask.palletsprojects.com/en/1.1.x/api/#flask.flash) pour afficher un message lorsqu’une requête est traitée.
* La fonction [redirect()](https://flask.palletsprojects.com/en/1.1.x/api/#flask.redirect) pour rediriger le client vers un autre emplacement.

Ajoutez les importations à votre fichier comme suit :

flask\_blog/app.py

import sqlite3

from flask import Flask, render\_template, request, url\_for, flash, redirect

from werkzeug.exceptions import abort

. . .

Copy

La fonction flash() stocke les messages flashs dans la session du navigateur du client, ce qui nécessite la définition d’une clé secrète. Cette clé secrète est utilisée pour sécuriser les sessions, ce qui permet à Flask de se souvenir des informations d’une requête à l’autre, comme passer de la page du nouveau message à la page d’index. L’utilisateur peut accéder aux informations stockées dans la session, mais ne peut pas les modifier s’il ne dispose pas de la clé secrète, de sorte que vous ne devez jamais autoriser quiconque à accéder à votre clé secrète. Pour plus d’informations, consultez [la documentation Flask pour les sessions](https://flask.palletsprojects.com/en/1.1.x/api/#sessions).

Pour définir une clé secrète, vous ajouterez une configuration SECRET\_KEY à votre application via l’objet app.config. Ajoutez-la directement après la définition de l’app avant de définir la fonction de visualisation index() :

flask\_blog/app.py

. . .

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = 'your secret key'

@app.route('/')

def index():

conn = get\_db\_connection()

posts = conn.execute('SELECT \* FROM posts').fetchall()

conn.close()

return render\_template('index.html', posts=posts)

. . .

Copy

N’oubliez pas que la clé secrète doit être une longue chaîne aléatoire.

Après avoir défini une clé secrète, vous allez créer une fonction de visualisation qui rendra un modèle affichant un formulaire que vous pouvez remplir pour créer un nouveau article de blog. Ajoutez cette nouvelle fonction au bas du fichier :

flask\_blog/app.py

. . .

@app.route('/create', methods=('GET', 'POST'))

def create():

return render\_template('create.html')

Copy

Cela crée une route /create qui accepte les demandes GET et POST. Les requêtes GET sont acceptées par défaut. Pour accepter également les requêtes POST, qui sont envoyées par le navigateur lors de la soumission des formulaires, vous passerez un [tuple](https://www.digitalocean.com/community/tutorials/understanding-tuples-in-python-3) avec les types de requêtes acceptés à l’argument methods du décorateur @app.route().

Enregistrez et fermez le fichier.

Pour créer le modèle, ouvrez un fichier appelé create.html dans votre dossier templates :

1. nano templates/create.html

Copy

Ajoutez le code suivant dans ce nouveau fichier :

flask\_blog/templates/create.html

{% extends 'base.html' %}

{% block content %}

<h1>{% block title %} Create a New Post {% endblock %}</h1>

<form method="post">

<div class="form-group">

<label for="title">Title</label>

<input type="text" name="title"

placeholder="Post title" class="form-control"

value="{{ request.form['title'] }}"></input>

</div>

<div class="form-group">

<label for="content">Content</label>

<textarea name="content" placeholder="Post content"

class="form-control">{{ request.form['content'] }}</textarea>

</div>

<div class="form-group">

<button type="submit" class="btn btn-primary">Submit</button>

</div>

</form>

{% endblock %}

Copy

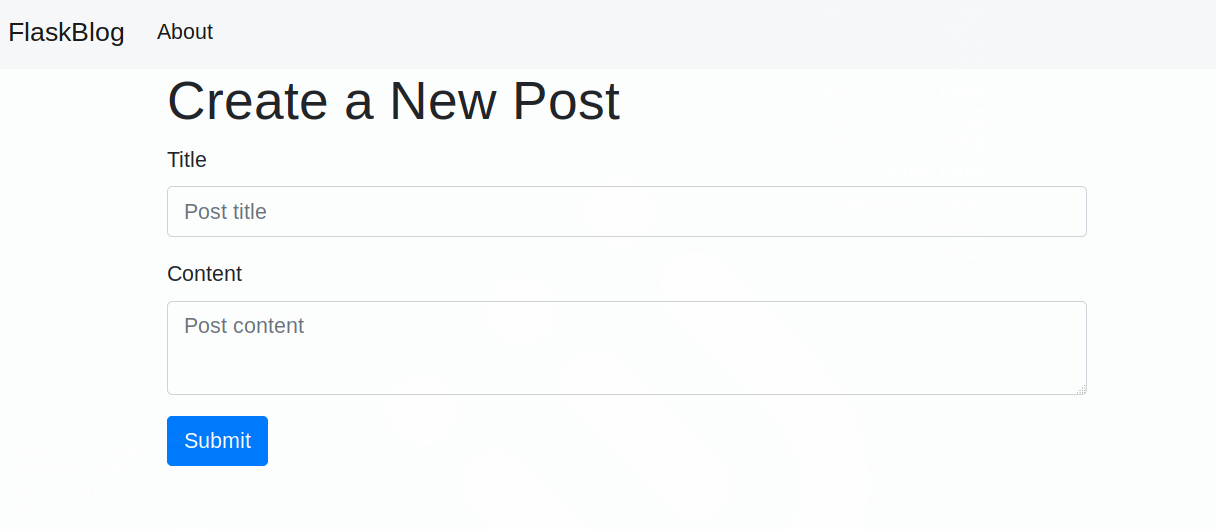
La majorité de ce code est du HTML standard. Il affichera une zone de saisie pour le titre du message, une zone de texte pour le contenu du message et un bouton pour soumettre le formulaire.

La valeur du titre de l’article est {{ request.form['title'] }} et la zone de texte a la valeur {{ request.form['content'] }}, ceci afin que les données que vous entrez ne soient pas perdues si quelque chose se passe mal.  Par exemple, si vous écrivez un long article et que vous oubliez de lui donner un titre, un message s’affiche pour vous informer que le titre est requis. Cela se produira sans perdre le message que vous avez écrit puisqu’il sera stocké dans l’objet request global auquel vous avez accès dans vos modèles.

Maintenant que le serveur de développement tourne, utilisez votre navigateur pour accéder à la route /create :

http://127.0.0.1:5000/create

Vous verrez une page **Create a New Post** avec une boîte pour un titre et un contenu.



Ce formulaire soumet une demande POST à votre fonction create() view. Cependant, il n’y a pas encore de code pour traiter une demande POST dans la fonction, donc rien ne se passe après avoir rempli le formulaire et l’avoir soumis.

Vous traiterez la requête POST entrante lorsqu’un formulaire sera soumis. Vous ferez cela à l’intérieur de la fonction de visualisation create(). Vous pouvez traiter la demande POST séparément en vérifiant la valeur de request.method. Lorsque sa valeur est réglée sur 'POST', elle signifie que la requête est une requête POST, vous allez ensuite continuer à extraire les données soumises, le valider et l’insérer dans votre base de données.

Ouvrez le fichier app.py pour le modifier :

1. nano app.py

Copy

Modifiez la fonction de visualisation create() pour qu’elle ressemble exactement à ce qui suit :

flask\_blog/app.py

. . .

@app.route('/create', methods=('GET', 'POST'))

def create():

if request.method == 'POST':

title = request.form['title']

content = request.form['content']

if not title:

flash('Title is required!')

else:

conn = get\_db\_connection()

conn.execute('INSERT INTO posts (title, content) VALUES (?, ?)',

(title, content))

conn.commit()

conn.close()

return redirect(url\_for('index'))

return render\_template('create.html')

Copy

Dans la déclaration if, vous devez vous assurer que le code qui suit n’est exécuté que lorsque la demande est une demande POST via la comparaison request.method == 'POST'.

Vous extrayez ensuite le titre et le contenu soumis de l’objet request.form qui vous donne accès aux données du formulaire dans la demande. Si le titre n’est pas fourni, la condition if not title serait remplie, en affichant un message à l’utilisateur l’informant que le titre est requis. Si, en revanche, le titre est fourni, vous ouvrez une connexion avec la fonction get\_db\_connection() et insérez le titre et le contenu que vous avez reçu dans le tableau posts.

Vous validez ensuite les modifications dans la base de données et coupez la connexion. Après avoir ajouté l’article de blog à la base de données, vous redirigez le client vers la page d’index en utilisant la fonction redirect() en lui passant l’URL générée par la fonction url\_for() avec la valeur 'index' en argument.

Enregistrez et fermez le fichier.

Maintenant, naviguez vers la route /create en utilisant votre navigateur web :

http://127.0.0.1:5000/create

Remplissez le formulaire avec un titre de votre choix et un peu de contenu. Une fois que vous avez soumis le formulaire, vous verrez le nouveau article listé sur la page d’index.

Enfin, vous afficherez des messages clignotants et ajouterez un lien vers la barre de navigation dans le modèle base.html pour accéder facilement à cette nouvelle page. Ouvrez le fichier modèle :

1. nano templates/base.html

Copy

Modifiez le fichier en ajoutant une nouvelle balise <li> à la suite du lien “About” à l’intérieur de la balise <nav>. Ensuite, ajoutez un nouveau for loop directement au-dessus du bloc content pour afficher les messages clignotants sous la barre de navigation. Ces messages sont disponibles dans la fonction spéciale get\_flashed\_messages() que Flask fournit :

flask\_blog/templates/base.html

<nav class="navbar navbar-expand-md navbar-light bg-light">

<a class="navbar-brand" href="{{ url\_for('index')}}">FlaskBlog</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">

<span class="navbar-toggler-icon"></span>

</button>

<div class="collapse navbar-collapse" id="navbarNav">

<ul class="navbar-nav">

<li class="nav-item">

<a class="nav-link" href="#">About</a>

</li>

<li class="nav-item">

<a class="nav-link" href="{{url\_for('create')}}">New Post</a>

</li>

</ul>

</div>

</nav>

<div class="container">

{% for message in get\_flashed\_messages() %}

<div class="alert alert-danger">{{ message }}</div>

{% endfor %}

{% block content %} {% endblock %}

</div>

Copy

Enregistrez et fermez le fichier. La barre de navigation comportera désormais un élément New Post qui renvoie à la route /create.

### [Éditer un message](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#editer-un-message)

Pour qu’un blog soit à jour, vous devez être en mesure de modifier vos articles existants. Cette section vous guidera dans la création d’une nouvelle page dans votre application afin de simplifier le processus de modification d’un article.

Tout d’abord, vous allez ajouter une nouvelle route au fichier app.py. Sa fonction de visualisation recevra l’ID du message qui doit être édité, l’URL sera au format /post\_id/edit avec la variable post\_id étant l’ID du message. Ouvrez le fichier app.py pour le modifier :

1. nano app.py

Copy

Ensuite, ajoutez la fonction d’affichage edit() suivante à la fin du fichier. L’édition d’un article existant est similaire à la création d’un nouvel article, donc cette fonction de visualisation sera similaire à la fonction de visualisation create() :

flask\_blog/app.py

. . .

@app.route('/<int:id>/edit', methods=('GET', 'POST'))

def edit(id):

post = get\_post(id)

if request.method == 'POST':

title = request.form['title']

content = request.form['content']

if not title:

flash('Title is required!')

else:

conn = get\_db\_connection()

conn.execute('UPDATE posts SET title = ?, content = ?'

' WHERE id = ?',

(title, content, id))

conn.commit()

conn.close()

return redirect(url\_for('index'))

return render\_template('edit.html', post=post)

Copy

L article que vous éditez est déterminé par l’URL et Flask passera le numéro d’identification à la fonction edit() via l’argument id. Vous ajoutez cette valeur à la fonction get\_post() pour récupérer le message associé à l’ID fourni de la base de données. Les nouvelles données seront envoyées dans une requête POST, qui est traitée dans la condition if request.method == 'POST'.

Tout comme lorsque vous créez un nouveau message, vous extrayez d’abord les données de l’objet request.form, puis vous envoyez un message si le titre a une valeur vide, sinon, vous ouvrez une connexion à la base de données. Ensuite, vous mettez à jour le tableau posts en définissant un nouveau titre et un nouveau contenu où l’ID de l’article dans la base de données est égal à l’ID qui se trouvait dans l’URL.

Dans le cas d’une requête GET, vous rendez un modèle edit.html en passant dans la variable post qui contient la valeur retournée par la fonction get\_post(). Vous l’utiliserez pour afficher le titre et le contenu existants sur la page d’édition.

Enregistrez et fermez le fichier, puis créez un nouveau modèle edit.html :

1. nano templates/edit.html

Copy

Ecrivez le code suivant dans ce nouveau fichier :

flask\_blog/templates/edit.html

{% extends 'base.html' %}

{% block content %}

<h1>{% block title %} Edit "{{ post['title'] }}" {% endblock %}</h1>

<form method="post">

<div class="form-group">

<label for="title">Title</label>

<input type="text" name="title" placeholder="Post title"

class="form-control"

value="{{ request.form['title'] or post['title'] }}">

</input>

</div>

<div class="form-group">

<label for="content">Content</label>

<textarea name="content" placeholder="Post content"

class="form-control">{{ request.form['content'] or post['content'] }}</textarea>

</div>

<div class="form-group">

<button type="submit" class="btn btn-primary">Submit</button>

</div>

</form>

<hr>

{% endblock %}

Copy

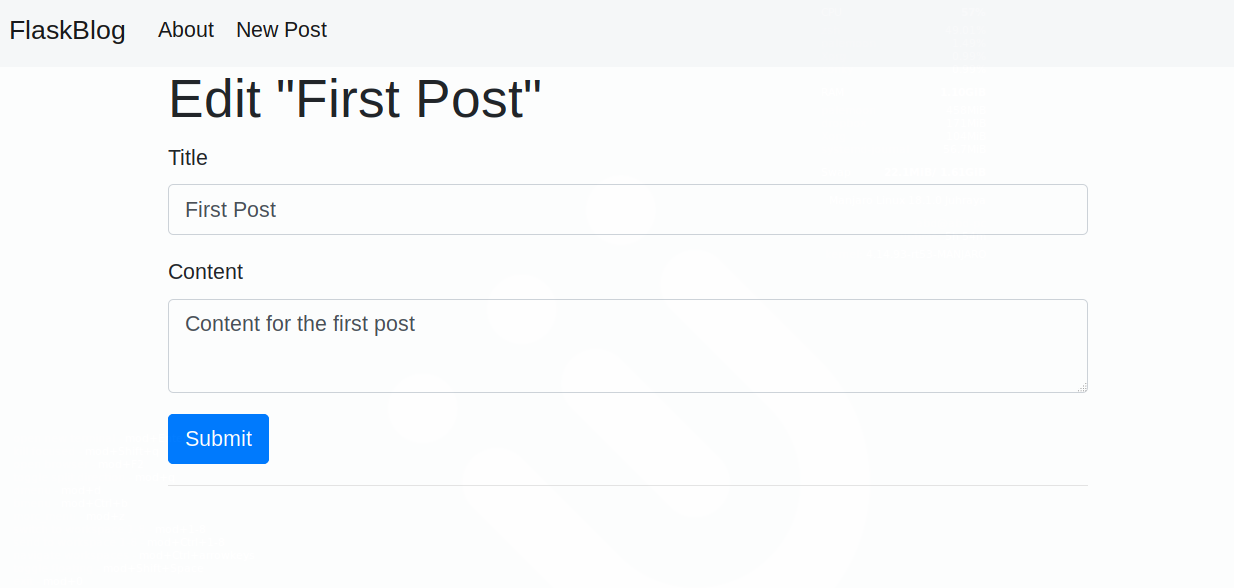
Enregistrez et fermez le fichier.

Ce code suit le même modèle, à l’exception de la syntaxe {{ request.form['title'] ou post['title'] }} et {{ request.form['content'] ou post['content'] }}. Cela affiche les données stockées dans la requête si elles existent, sinon, cela affiche les données de la variable post qui a été passée au modèle contenant les données actuelles de la base de données.

Maintenant, naviguez vers l’URL suivante pour modifier le premier article :

http://127.0.0.1:5000/1/edit

Vous verrez une page **Edit “First Post”**.



Modifiez l’article et soumettez le formulaire, puis assurez-vous que l’article a été mis à jour.

Vous devez maintenant ajouter un lien qui pointe vers la page d’édition pour chaque article sur la page d’index. Ouvrez le fichier modèle index.html :

1. nano templates/index.html

Copy

Modifiez le fichier pour qu’il ressemble exactement à ce qui suit :

flask\_blog/templates/index.html

{% extends 'base.html' %}

{% block content %}

<h1>{% block title %} Welcome to FlaskBlog {% endblock %}</h1>

{% for post in posts %}

<a href="{{ url\_for('post', post\_id=post['id']) }}">

<h2>{{ post['title'] }}</h2>

</a>

<span class="badge badge-primary">{{ post['created'] }}</span>

<a href="{{ url\_for('edit', id=post['id']) }}">

<span class="badge badge-warning">Edit</span>

</a>

<hr>

{% endfor %}

{% endblock %}

Copy

Enregistrez et fermez le fichier.

Ici, vous ajoutez une balise <a> pour faire un lien vers la fonction de visualisation edit(), en passant la valeur post['id'] pour faire un lien vers la page d’édition de chaque article avec le lien Edit.

### [Suppression d’un article](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#suppression-d-un-article)

Parfois, un article n’a plus besoin d’être accessible au public, c’est pourquoi la fonctionnalité de suppression d’un article est cruciale. Au cours de cette étape, vous allez ajouter la fonctionnalité de suppression à votre application.

Tout d’abord, vous allez ajouter une nouvelle route /ID/delete qui accepte les requêtes POST, similaire à la fonction de visualisation edit(). Votre nouvelle fonction de visualisation delete() recevra l’ID du message à supprimer de l’URL. Ouvrez le fichier app.py :

1. nano app.py

Copy

Ajoutez la fonction de visualisation suivante au bas du fichier :

flask\_blog/app.py

# ....

@app.route('/<int:id>/delete', methods=('POST',))

def delete(id):

post = get\_post(id)

conn = get\_db\_connection()

conn.execute('DELETE FROM posts WHERE id = ?', (id,))

conn.commit()

conn.close()

flash('"{}" was successfully deleted!'.format(post['title']))

return redirect(url\_for('index'))

Copy

Cette fonction de visualisation n’accepte que les requêtes POST. Cela signifie que la navigation vers la route /ID/delete sur votre navigateur renvoie une erreur, car les navigateurs web acceptent par défaut les requêtes GET.

Cependant, vous pouvez accéder à cette route via un formulaire qui envoie une requête POST passant l’ID de l’article que vous voulez supprimer. La fonction recevra la valeur de l’ID et l’utilisera pour récupérer l’article dans la base de données avec la fonction get\_post().

Ensuite, vous ouvrez une connexion à la base de données et exécutez une commande SQL DELETE FROM pour supprimer l’article. Vous validez la modification dans la base de données et fermez la connexion tout en faisant clignoter un message pour informer l’utilisateur que le message a été supprimé avec succès et le rediriger vers la page d’index.

Notez que vous ne rendez pas un fichier modèle, ceci est dû au fait que vous allez juste ajouter un bouton Delete à la page d’édition.

Ouvrez le fichier modèle edit.html :

1. nano templates/edit.html

Copy

Ensuite, ajoutez la balise <form> suivante après la balise <hr> et directement avant la ligne {% endblock %} :

flask\_blog/templates/edit.html

<hr>

<form action="{{ url\_for('delete', id=post['id']) }}" method="POST">

<input type="submit" value="Delete Post"

class="btn btn-danger btn-sm"

onclick="return confirm('Are you sure you want to delete this post?')">

</form>

{% endblock %}

Copy

Vous utilisez la méthode [confirm()](https://developer.mozilla.org/en-US/docs/Web/API/Window/confirm) pour afficher un message de confirmation avant de soumettre la demande.

Maintenant, naviguez à nouveau vers la page d’édition d’un article de blog et essayez de le supprimer :

http://127.0.0.1:5000/1/edit

À la fin de cette étape, le code source de votre projet ressemblera au [code de cette page](https://github.com/do-community/flask_blog).

Grâce à cela, les utilisateurs de votre application peuvent maintenant écrire de nouveaux articles de blog et les ajouter à la base de données, modifier et supprimer des articles existants.

## [Conclusion](https://www.digitalocean.com/community/tutorials/how-to-make-a-web-application-using-flask-in-python-3-fr#conclusion)

Ce tutoriel a introduit les concepts essentiels du framework Flask Python. Vous avez appris à créer une petite application web, à l’exécuter dans un serveur de développement et à permettre à l’utilisateur de fournir des données personnalisées via des paramètres d’URL et des formulaires web. Vous avez également utilisé le [moteur de modèles Jinja](http://jinja.palletsprojects.com/) pour réutiliser les fichiers HTML et y utiliser la logique. À la fin de ce tutoriel, vous disposez maintenant d’un blog web entièrement fonctionnel qui interagit avec une [base de données SQLite](https://sqlite.org/) pour créer, afficher, modifier et supprimer des articles de blog en utilisant le langage Python et des requêtes SQL.

Vous pouvez développer davantage cette application en ajoutant l’authentification des utilisateurs afin que seuls les utilisateurs enregistrés puissent créer et modifier des articles de blog, vous pouvez également ajouter des commentaires et des balises pour chaque article de blog, et ajouter des téléchargements de fichiers pour donner aux utilisateurs la possibilité d’inclure des images dans l’article. Pour plus d’informations, consultez la [documentation Flask](https://flask.palletsprojects.com/en/1.1.x/).

Flask possède de nombreuses [extensions Flask](http://flask.palletsprojects.com/en/1.1.x/extensions/) créées par sa communauté. Voici une liste d’extensions que vous pourriez envisager d’utiliser pour faciliter votre processus de développement :

* [Flask-Login](https://flask-login.readthedocs.io/en/latest/) : gère la session utilisateur et gère la connexion et la déconnexion ainsi que la mémorisation des utilisateurs connectés.
* [Flask-SQLAlchemy](https://flask-sqlalchemy.palletsprojects.com/en/2.x/) : simplifie l’utilisation de Flask avec [SQLAlchemy](https://www.sqlalchemy.org/), une boîte à outils SQL Python et un ORM (mappeur relationnel d’objets) pour interagir avec les bases de données SQL.
* [Flask-Mail](https://pythonhosted.org/Flask-Mail/) : aide à l’envoi de messages électroniques dans votre application Flask.

Thanks for learning with the DigitalOcean Community. Check out our offerings for compute, storage, networking, and managed databases.

[**Learn more about us**](https://www.digitalocean.com/)

**About the authors**



[Abdelhadi Dyouri](https://www.digitalocean.com/community/users/adyouri)

Author



[Kathryn Hancox](https://www.digitalocean.com/community/users/khancox)

Editor

#### **Still looking for an answer?**

[**Ask a question**](https://www.digitalocean.com/community/questions)[**Search for more help**](https://www.digitalocean.com/community)

Was this helpful?

YesNo

Comments

**Leave a comment**

Haut du formulaire

Leave a comment...

Bas du formulaire

This textbox defaults to using **Markdown** to format your answer.

You can type **!ref** in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

[**Sign In or Sign Up to Comment**](https://www.digitalocean.com/api/dynamic-content/v1/login?success_redirect=https%3A%2F%2Fwww.digitalocean.com%2Fcommunity%2Ftutorials%2Fhow-to-make-a-web-application-using-flask-in-python-3-fr&error_redirect=https%3A%2F%2Fwww.digitalocean.com%2Fauth-error&type=register&site=community)

[Creative CommonsThis work is licensed under a Creative Commons Attribution-NonCommercial- ShareAlike 4.0 International License.](https://creativecommons.org/licenses/by-nc-sa/4.0/)

Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the [Glacier Bay National Park & Preserve](https://www.nps.gov/glba/learn/nature/soundclips.htm) and [Merrick079](https://freesound.org/people/Merrick079/sounds/567347/) for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the [Whale and Dolphin Conservation](https://www.whales.org/).

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg



##### Get our biweekly newsletter

Sign up for Infrastructure as a Newsletter.

[**Sign up**](https://www.digitalocean.com/community/newsletter)



##### Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We’d like to help.

[**Learn more**](https://www.digitalocean.com/community/pages/hollies-hub-for-good)



##### Become a contributor

You get paid; we donate to tech nonprofits.

[**Learn more**](https://www.digitalocean.com/community/pages/write-for-digitalocean)

###### **Featured on Community**

[Kubernetes Course](https://www.digitalocean.com/community/tutorials/building-optimized-containers-for-kubernetes)[Learn Python 3](https://www.digitalocean.com/community/tutorial-series/how-to-code-in-python-3)[Machine Learning in Python](https://www.digitalocean.com/community/tutorials/machine-learning-projects-python-a-digitalocean-ebook)[Getting started with Go](https://www.digitalocean.com/community/tutorials/how-to-write-your-first-program-in-go)[Intro to Kubernetes](https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes)

###### **DigitalOcean Products**

[Cloudways](https://www.cloudways.com/en)[Virtual Machines](https://www.digitalocean.com/products/droplets)[Managed Databases](https://www.digitalocean.com/products/managed-databases)[Managed Kubernetes](https://www.digitalocean.com/products/kubernetes)[Block Storage](https://www.digitalocean.com/products/block-storage)[Object Storage](https://www.digitalocean.com/products/spaces)[Marketplace](https://marketplace.digitalocean.com/)[VPC](https://www.digitalocean.com/products/vpc)[Load Balancers](https://www.digitalocean.com/products/load-balancer)

##### Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow – whether you’re running one virtual machine or ten thousand.

[**Learn more ->**](https://www.digitalocean.com/products)

##### Get started for free

Enter your email to get $200 in credit for your first 60 days with DigitalOcean.

Haut du formulaire

Send My Promo

Bas du formulaire

New accounts only. By submitting your email you agree to our [Privacy Policy](https://www.digitalocean.com/legal/privacy-policy).

###### Company

###### Products

###### Community

###### Solutions

###### Contact

© 2023 DigitalOcean, LLC.