Projet Django

Source : <https://realpython.com/get-started-with-django-1/>



# Commencez avec Django Partie 1: Construire une application Portfolio

par [Jasmine Finer](https://github.com/realpython/materials/blob/4dd5d79634efbffeb8999052a9e94b3dba4b25ba/rp-portfolio/personal_portfolio/templates/base.html#author)  01 avril 2019   [**Django**](https://realpython.com/tutorials/django/) [**intermédiaire**](https://realpython.com/tutorials/intermediate/) [**web-dev**](https://www.lipsum.com/)

Table des matières

* [Pourquoi devriez-vous apprendre Django](https://getbootstrap.com/docs/4.1/components/navbar/#why-you-should-learn-django)
* [La structure d'un site web Django](https://docs.python.org/3.7/library/datetime.html#the-structure-of-a-django-website)
* [Ce que vous allez construire](https://realpython.com/get-started-with-django-1/#what-youre-going-to-build)
* [Bonjour le monde!](https://realpython.com/get-started-with-django-1/#hello-world)
  + [Configurez votre environnement de développement](https://getbootstrap.com/docs/4.1/getting-started/introduction/#set-up-your-development-environment)
  + [Créer un projet Django](https://getbootstrap.com/#create-a-django-project)
  + [Créer une application Django](https://docs.djangoproject.com/en/2.1/ref/templates/builtins/#create-a-django-application)
  + [Créer une vue](https://realpython.com/get-started-with-django-1/#create-a-view)
  + [Ajouter Bootstrap à votre application](https://realpython.com/get-started-with-django-1/#add-bootstrap-to-your-app)
* [Présentez vos projets](https://docs.djangoproject.com/en/2.1/topics/db/queries/#showcase-your-projects)
  + [Projets App: Modèles](https://github.com/realpython/materials/tree/a639f1c2f85032334fbb4dca88f3e8dc88397f6d/rp-portfolio#projects-app-models)
  + [Projets App: Vues](https://realpython.com/get-started-with-django-1/#projects-app-views)
  + [Application Projets: Templates](https://realpython.com/get-started-with-django-1/#projects-app-templates)
* [Partagez vos connaissances avec un blog](https://realpython.com/get-started-with-django-1/#share-your-knowledge-with-a-blog)
  + [Blog App: Modèles](https://files.realpython.com/media/Screenshot_2018-12-17_at_23.50.45.fedcaa17e99d.png#blog-app-models)
  + [Application de blog: Django Admin](https://realpython.com/products/real-python-course/#blog-app-django-admin)
  + [Application Blog: Vues](https://docs.djangoproject.com/en/2.1/ref/templates/builtins/#blog-app-views)
  + [Application de blog: Templates](https://realpython.com/get-started-with-django-1/#blog-app-templates)
* [Conclusion](https://realpython.com/get-started-with-django-1/#conclusion)

[](https://srv.realpython.net/click/58257184031/?c=31941813441&p=58946116052&r=02847)

**Regarder maintenant** Ce tutoriel contient un cours vidéo connexe créé par l'équipe Real Python. Regardez-le avec le didacticiel écrit pour approfondir votre compréhension: [**Premiers pas avec Django: Construire une application Portfolio**](https://github.com/realpython/materials/blob/0091ee5421f8107e8629f1f22687ff224850b889/rp-portfolio/projects/migrations/0001_initial.py)

Django est un framework Web Python complet qui peut être utilisé pour créer des applications Web complexes. Dans ce tutoriel, vous allez apprendre et apprendre [Django](https://files.realpython.com/media/Screenshot_2018-12-17_at_23.50.51.08dadaa185fc.png) par exemple. Vous suivrez les étapes pour créer une application Web pleinement opérationnelle et, en chemin, apprendrez certaines des fonctionnalités les plus importantes du framework et leur fonctionnement commun.

Dans les articles suivants de cette série, vous découvrirez comment créer des sites Web plus complexes en utilisant encore plus de fonctionnalités de Django que ce que vous découvrirez dans ce didacticiel.

**À la fin de ce tutoriel, vous serez capable de :**

* Comprendre ce qu'est Django et pourquoi c'est un excellent framework web
* Comprendre l'architecture d'un site Django et sa comparaison avec d'autres frameworks
* Configurer un nouveau projet et une application Django
* Construire un site Web portefeuille personnel avec Django

**Bonus gratuit:**[**Cliquez ici pour accéder à un Guide de ressources pédagogiques Django gratuit (PDF)**](https://realpython.com/get-started-with-django-1/) qui présente des astuces et des astuces, ainsi que les pièges courants à éviter lors de la création d'applications Web Python + Django.

## Pourquoi devriez-vous apprendre Django

Il existe une infinité de cadres de développement Web, alors pourquoi devriez-vous apprendre Django plutôt que d’autres ? Tout d'abord, il est écrit en Python, l'un des langages de programmation les plus lisibles et les plus conviviaux pour les débutants.

**Remarque:** Ce didacticiel suppose une connaissance intermédiaire du langage Python. Si vous débutez en programmation avec Python, consultez certains de nos [tutoriels pour débutants](https://realpython.com/tutorials/basics/) ou le [cours d'introduction](https://files.realpython.com/media/Screenshot_2018-12-17_at_23.51.17.8849893e9b69.png) .

La deuxième raison pour laquelle vous devriez apprendre Django est la portée de ses fonctionnalités. Si vous devez créer un site Web, vous n'avez pas besoin de recourir à des bibliothèques ou des packages externes si vous choisissez Django. Cela signifie que vous n'avez pas besoin d'apprendre à utiliser quoi que ce soit d'autre, et la syntaxe est transparente puisque vous n'utilisez qu'un seul framework.

Il y a aussi l'avantage supplémentaire que vous n'avez pas à craindre que la mise à jour d'une bibliothèque ou d'un framework rende inutilisables les autres que vous avez installées.

Si vous avez besoin d'ajouter des fonctionnalités supplémentaires, vous pouvez utiliser une gamme de bibliothèques externes pour améliorer votre site.

L'un des points forts du framework Django est sa [documentation détaillée](https://github.com/realpython/materials/blob/1ea78b8bb71e685c476d6fd98e829b6ad0a42123/rp-portfolio/personal_portfolio/templates/base.html) . Il contient une documentation détaillée sur chaque aspect de Django, ainsi que d'excellents exemples et même un tutoriel pour vous aider à démarrer.

Il existe également une fantastique communauté de développeurs Django, donc si vous êtes bloqué, il est presque toujours possible de progresser en vérifiant la documentation ou en [demandant à la communauté](https://github.com/realpython/materials/tree/4dd5d79634efbffeb8999052a9e94b3dba4b25ba/rp-portfolio) .

Django est un cadre d'applications Web de haut niveau doté de nombreuses fonctionnalités. C'est une excellente chose pour les débutants en développement Web en raison de sa documentation fantastique, et en particulier si vous connaissez également Python.

## La structure d'un site web Django

Un site Web Django consiste en un seul **projet** divisé en **applications** séparées . L'idée est que chaque application gère une fonction autonome que le site doit exécuter. Par exemple, imaginons une application comme Instagram. Plusieurs fonctions différentes doivent être exécutées:

* **Gestion des utilisateurs** : connexion, déconnexion, enregistrement, etc.
* **Le flux d'images** : télécharger, éditer et afficher des images
* **Messagerie privée** : messages privés entre utilisateurs et notifications

Il s’agit de fonctionnalités distinctes. Ainsi, s’il s’agissait d’un site Django, chaque fonctionnalité devrait correspondre à une application Django différente au sein d’un même projet Django.

Le projet Django contient certaines configurations qui s'appliquent au projet dans son ensemble, telles que les paramètres de projet, les URL, les templates partagés et les fichiers statiques. Chaque application peut avoir sa propre base de données et ses propres fonctions pour contrôler la façon dont les données sont affichées à l'utilisateur dans des templates HTML.

Chaque application possède également ses propres URL, ainsi que ses propres templates HTML et fichiers statiques, tels que JavaScript et CSS.

Les applications Django sont structurées de manière à séparer la logique. Il prend en charge le pattern Model-View-Controller, qui est l'architecture sur laquelle la plupart des infrastructures Web sont construites. Le principe de base est que dans chaque application, il y a trois fichiers distincts qui traitent les trois éléments de logique principaux séparément :

* **Le modèle** définit la structure de données. Il s'agit généralement d'une base de données et constitue la couche de base d'une application.
* **La vue** affiche tout ou partie des données à l'utilisateur avec HTML et CSS.
* **Le contrôleur** gère les interactions entre la base de données et la vue.

Si vous souhaitez en savoir plus sur le modèle MVC, consultez le document MVC ( [Model-View-Controller) - Avec Legos](https://realpython.com/the-model-view-controller-mvc-paradigm-summarized-with-legos/) .

À Django, l'architecture est légèrement différente. Bien que basé sur le modèle MVC, Django gère la partie contrôleur elle-même. Il n'est pas nécessaire de définir les interactions entre la base de données et les vues. Tout est fait pour vous !

Le modèle utilisé par Django est appelé modèle MVT (Model-View-Template). La vue et le template du modèle MVT constituent la vue du modèle MVC. Tout ce que vous avez à faire est d’ajouter des configurations d’URL pour mapper les vues, et Django se charge du reste!

Un site Django commence par un projet et est construit avec un certain nombre d'applications qui gèrent chacune une fonctionnalité distincte. Chaque application suit le modèle modèle-vue-modèle. Maintenant que vous connaissez la structure d'un site Django, voyons ce que vous allez construire!

## Ce que vous allez construire

Avant de vous lancer dans un projet de développement Web, il est judicieux de définir un plan de ce que vous allez construire. Dans ce tutoriel, nous allons construire une application avec les fonctionnalités suivantes :

* **Un blog entièrement fonctionnel** : si vous souhaitez démontrer votre capacité de codage, un blog est un excellent moyen de le faire. Dans cette application, vous pourrez créer, mettre à jour et supprimer des articles de blog. Les messages auront des catégories qui peuvent être utilisées pour les trier. Enfin, les utilisateurs pourront laisser des commentaires sur les messages.
* **Un portfolio de votre travail** : Vous pouvez présenter les projets de [développement Web](https://realpython.com/tutorials/web-dev/) précédents ici. Vous allez créer une page de style galerie avec des liens cliquables vers les projets que vous avez terminés.

**Remarque:** avant de commencer, vous pouvez extraire le [code source](https://realpython.com/get-started-with-django-1/) et suivre le didacticiel.

Si vous préférez suivre en écrivant le code vous-même, ne vous inquiétez pas. J'ai référencé les parties pertinentes du code source afin que vous puissiez y revenir.

Nous n'utiliserons aucune bibliothèque externe Python dans ce tutoriel. L’un des avantages de Django est qu’il dispose de tellement de fonctionnalités qu’il n’est pas nécessaire de s’appuyer sur des bibliothèques externes. Cependant, nous ajouterons le style [Bootstrap 4](https://github.com/realpython/materials/tree/6fdccb8ae85c5792e3639c406be09abc2d4803d8/rp-portfolio) dans les templates.

En construisant ces deux applications, vous apprendrez les bases des modèles Django, des fonctions d'affichage, des formulaires, des templates et de la page d'administration de Django. Avec la connaissance de ces fonctionnalités, vous pourrez vous en aller et créer plus d'applications. Vous aurez également les outils pour apprendre encore plus et construire des sites Django sophistiqués.

## Hello, World!

Maintenant que vous connaissez la structure d'une application Django et ce que vous êtes sur le point de construire, nous allons suivre le processus de création d'une application dans Django. Vous l'étendrez plus tard dans votre application de portefeuille personnel.

### Configurez votre environnement de développement

Lorsque vous démarrez un nouveau projet de développement Web, il est judicieux de configurer d’abord votre environnement de développement. Créez un nouveau répertoire dans lequel votre projet va vivre cd :

**Shell**

$ mkdir rp-portfolio

$ cd rp-portfolio

Une fois dans le répertoire principal, il est recommandé de créer un [environnement virtuel](https://files.realpython.com/media/Screenshot_2018-12-09_at_17.57.22.f3c9ea711bd4.png) pour gérer les dépendances. Il existe de nombreuses façons de configurer des environnements virtuels, mais ici, vous allez utiliser venv :

**Shell**

$ python3 -m venv venv

Cette commande créera un dossier venv dans votre répertoire de travail. Dans ce répertoire, vous trouverez plusieurs fichiers, dont une copie de la bibliothèque standard Python. Plus tard, lorsque vous installerez de nouvelles dépendances, elles seront également stockées dans ce répertoire. Ensuite, vous devez activer l'environnement virtuel en exécutant la commande suivante :

**Shell**

$ source venv/bin/activate

**Remarque:** Si vous n'utilisez pas le shell bash, vous devrez peut-être utiliser une [commande différente](https://docs.python.org/3/library/venv.html#creating-virtual-environments) pour activer votre environnement virtuel. Par exemple, sous Windows, vous avez besoin de cette commande :

**Console Windows**

C:\> venv\Scripts\activate.bat

Vous saurez que votre environnement virtuel a été activé, car votre invite de console dans le terminal changera. Ça devrait ressembler à quelque chose comme ça :

**Shell**

(venv) $

**Remarque:** votre répertoire d'environnement virtuel ne doit pas être appelé venv. Si vous voulez en créer un sous un autre nom, par exemple my\_venv, il suffit de remplacer le second venv par my\_venv.

Ensuite, lors de l'activation de votre environnement virtuel, remplacez venv par my\_venv. Maintenant, l'invite porte également le préfixe (my\_venv).

Maintenant que vous avez créé un environnement virtuel, il est temps d'installer Django. Vous pouvez le faire en utilisant pip :

**Shell**

(venv) $ pip install Django

Une fois que vous avez configuré l'environnement virtuel et installé Django, vous pouvez maintenant vous plonger dans la création de l'application.

### Créer un projet Django

Comme vous l'avez vu dans la section précédente, une application Web Django est composée d'un projet et de ses applications constitutives. Assurez-vous que vous êtes dans le répertoire rp\_portfolio et que vous avez activé votre environnement virtuel, exécutez la commande suivante pour créer le projet :

**Shell**

$ django-admin startproject personal\_portfolio

Cela va créer un nouveau répertoire personal\_portfolio. Si vous entrez cd dans ce nouveau répertoire, vous verrez un autre répertoire appelé personal\_portfolio et un fichier appelé manage.py. Votre structure de répertoire devrait ressembler à ceci :

rp-portfolio/

│

├── personal\_portfolio/

│ ├── personal\_portfolio/

│ │ ├── \_\_init\_\_.py

│ │ ├── settings.py

│ │ ├── urls.py

│ │ └── wsgi.py

│ │

│ └── manage.py

│

└── venv/

La plupart du travail que vous faites sera dans ce premier répertoire personal\_portfolio. Pour éviter de devoir parcourir plusieurs répertoires (cd) à chaque fois que vous travaillez sur votre projet, il peut être utile de le réorganiser légèrement en déplaçant tous les fichiers d'un répertoire à un autre. Pendant que vous êtes dans le répertoire rp-portfolio, exécutez les commandes suivantes :

**Shell**

$ mv personal\_portfolio/manage.py ./

$ mv personal\_portfolio/personal\_portfolio/\* personal\_portfolio

$ rm -r personal\_portfolio/personal\_portfolio/

Vous devriez vous retrouver avec quelque chose comme ça :

rp-portfolio/

│

├── personal\_portfolio/

│ ├── \_\_init\_\_.py

│ ├── settings.py

│ ├── urls.py

│ └── wsgi.py

│

├── venv/

│

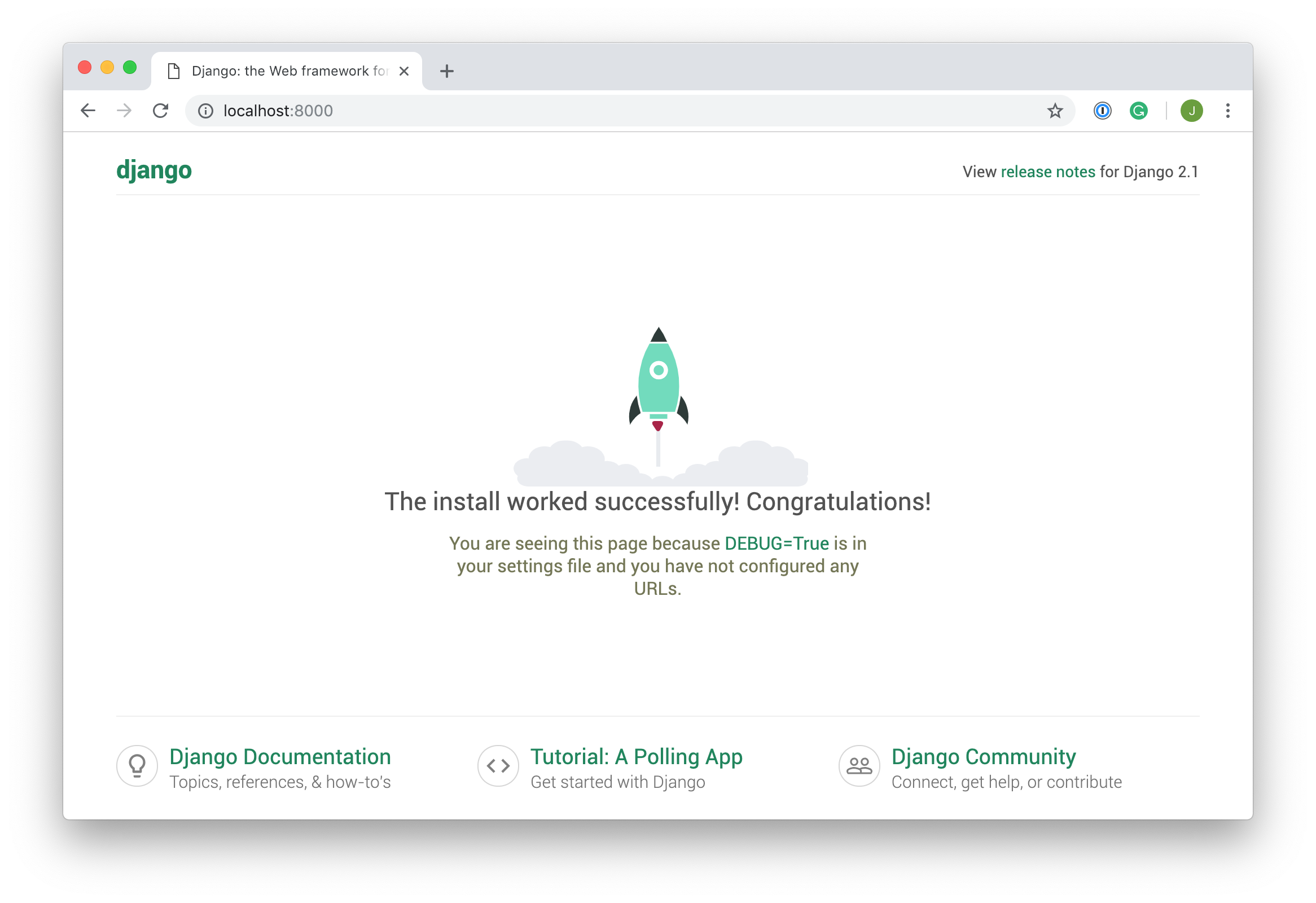
└── manage.py

Une fois la structure de votre fichier configurée, vous pouvez maintenant démarrer le serveur et vérifier que votre configuration a réussi. Dans la console, exécutez la commande suivante :

**Shell**

$ python manage.py runserver

Ensuite, dans votre navigateur, allez à localhost:8000, et vous devriez voir ce qui suit :

[](https://files.realpython.com/media/Screenshot_2018-12-09_at_17.58.16.20be0c5d3f1e.png)

Félicitations, vous avez créé un site Django ! Le code source de cette partie du tutoriel est disponible sur [GitHub](https://docs.djangoproject.com/en/2.1/). L'étape suivante consiste à créer des applications afin que vous puissiez ajouter des vues et des fonctionnalités à votre site.

### Créer une application Django

Pour cette partie du didacticiel, nous allons créer une application appelée hello\_world, que vous supprimerez par la suite car elle n'est pas nécessaire pour notre site de portfolio personnel.

Pour créer l'application, exécutez la commande suivante :

**Shell**

$ python manage.py startapp hello\_world

Cela créera un autre répertoire appelé hello\_world avec plusieurs fichiers :

* **\_\_init\_\_.py** indique à Python de traiter le répertoire comme un paquet Python.
* **admin.py** contient les paramètres pour les pages d’administration Django.
* **apps.py** contient les paramètres pour la configuration de l'application.
* **models.py** contient une série de classes que l'ORM de Django convertit en tables de base de données.
* **tests.py** contient des classes de test.
* **views.py** contient des fonctions et des classes qui gèrent les données affichées dans les templates HTML.

Une fois l'application créée, vous devez l'installer dans votre projet. Dans rp-portfolio/settings.py, ajoutez la ligne de code suivante sous INSTALLED\_APPS :

**Python**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'hello\_world',

]

Cette ligne de code signifie que votre projet sait maintenant que l'application que vous venez de créer existe. L'étape suivante consiste à créer une vue afin de pouvoir afficher quelque chose à un utilisateur.

### Créer une vue

Les vues dans Django sont un ensemble de fonctions ou de classes contenues dans le fichier views.py de votre répertoire d'application. Chaque fonction ou classe gère la logique qui est traitée chaque fois qu'une URL différente est visitée.

Accédez au fichier views.py dans le répertoire hello\_world. Il existe déjà une ligne de code qui importe render(). Ajoutez le code suivant :

**Python**

from django.shortcuts import render

def hello\_world(request):

return render(request, 'hello\_world.html', {})

Dans ce morceau de code, vous avez défini une **fonction de vue** appelée hello\_world(). Lorsque cette fonction est appelée, un fichier HTML sera rendu appelé hello\_world.html. Ce fichier n'existe pas encore, mais nous le créerons bientôt.

La fonction d'affichage prend un argument, request. Cet objet est un objet créé HttpRequestObject chaque fois qu'une page est chargée. Il contient des informations sur la requête, telles que la méthode, pouvant prendre plusieurs valeurs, notamment : GET et POST.

Maintenant que vous avez créé la fonction d'affichage, vous devez créer le template HTML à afficher pour l'utilisateur. render() recherche les templates HTML dans un répertoire appelé templates dans votre répertoire app. Créez ce répertoire et ensuite un fichier nommé à l'intérieur hello\_world.html :

**Shell**

$ mkdir hello\_world/templates/

$ touch hello\_world/templates/hello\_world.html

Ajoutez les lignes HTML suivantes à votre fichier :

**HTML**

<h1>Hello, World!</h1>

Vous avez maintenant créé une fonction pour gérer vos vues et les templates à afficher pour l'utilisateur. La dernière étape consiste à connecter vos URL pour que vous puissiez visiter la page que vous venez de créer. Votre projet comporte un module appelé urls.py dans lequel vous devez inclure une configuration d’URL pour l’application hello\_world. À l'intérieur de personal\_portfolio/urls.py, ajoutez ce qui suit :

**Python**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('hello\_world.urls')),

]

Ceci recherche un module appelé urls.py dans l'application hello\_world et enregistre toutes les URL définies dans cette application. Chaque fois que vous visitez le chemin racine de votre URL ( localhost:8000), les URL de l' application hello\_world sont enregistrées. Le module hello\_world.urls n'existe pas encore, vous devrez donc le créer :

**Shell**

$ touch hello\_world/urls.py

Dans ce module, nous devons importer l'objet chemin ainsi que le module views de notre application. Ensuite, nous voulons créer une liste de patterns d'URL correspondant aux différentes fonctions d'affichage. Pour le moment, nous n’avons créé qu’une seule fonction de vue. Il n’est donc nécessaire de créer qu’une seule URL :

**Python**

from django.urls import path

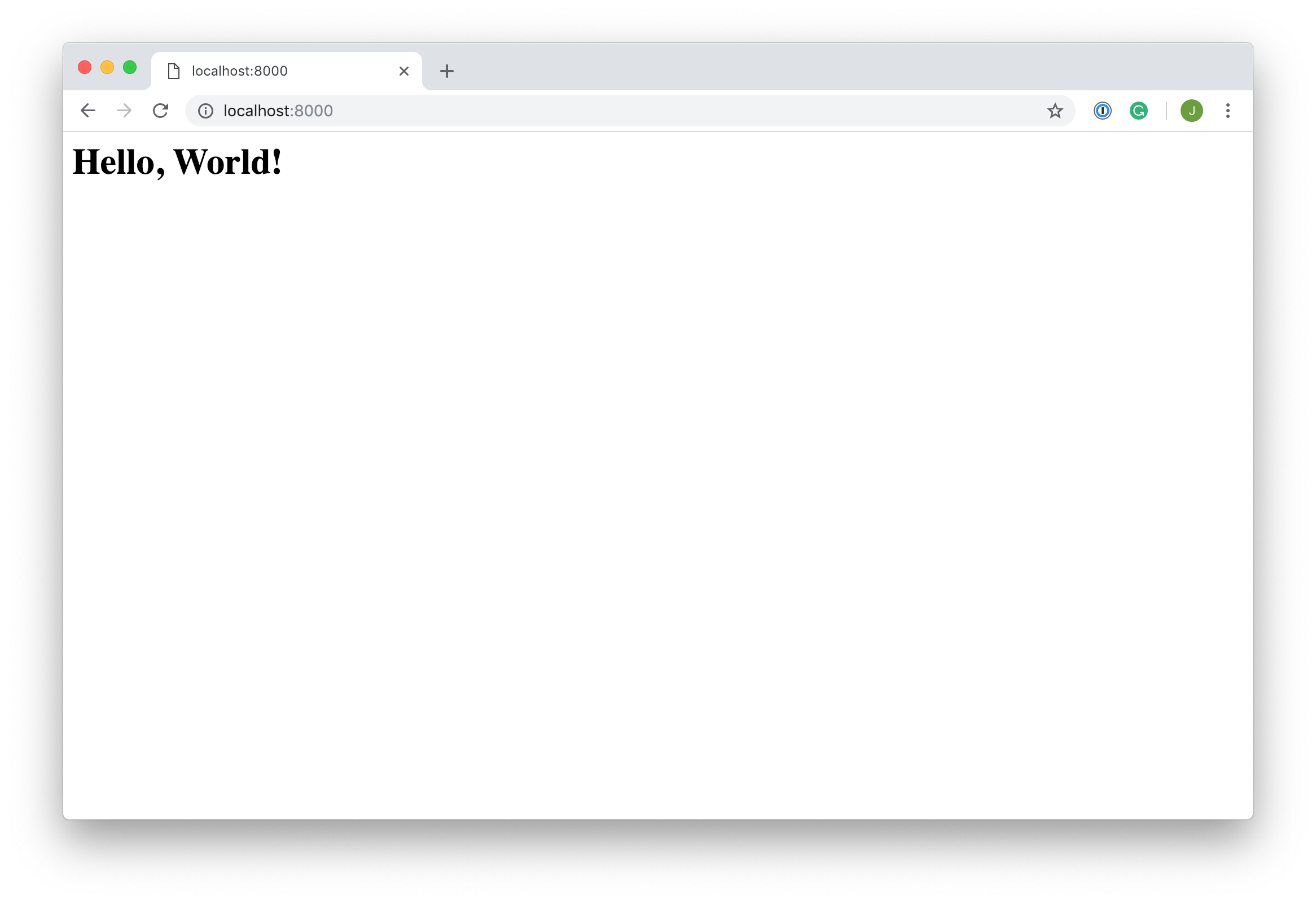
from hello\_world import views

urlpatterns = [

path('', views.hello\_world, name='hello\_world'),

]

Maintenant, lorsque vous redémarrez le serveur et que vous visitez localhost:8000, vous devriez pouvoir voir le modèle HTML que vous avez créé :

[](https://realpython.com/courses/django-portfolio-project/)

Félicitations encore ! Vous avez créé votre première application Django et l'avez connectée à votre projet. N'oubliez pas de consulter le [code source](https://github.com/realpython/materials/tree/754bf43a150b818f73e6508fd937eefee997cc6c/rp-portfolio) de cette section et de la précédente. Le seul problème maintenant, c'est que ça n'a pas l'air très beau. Dans la section suivante, nous allons ajouter des styles bootstrap à votre projet pour le rendre plus joli !

### Ajouter Bootstrap à votre application

Si vous n’ajoutez pas de style, l’application que vous créez ne sera pas très belle. Au lieu d'entrer dans le style CSS avec ce didacticiel, nous expliquerons simplement comment ajouter des styles d'amorçage à votre projet. Cela nous permettra d'améliorer l'apparence du site sans trop d'effort.

Avant de commencer à utiliser les styles Bootstrap, nous allons créer un template de base que nous pourrons importer dans chaque vue suivante. Ce template est l'endroit où nous ajouterons ensuite les importations de style Bootstrap.

Créez un autre répertoire appelé templates, cette fois à l'intérieur de personal\_portfolio, et un fichier appelé base.html à l'intérieur du nouveau répertoire :

**Shell**

$ mkdir personal\_portfolio/templates/

$ touch personal\_portfolio/templates/base.html

Nous créons ce répertoire de templates supplémentaire pour stocker les templates HTML qui seront utilisés dans chaque application Django du projet. Comme vous l'avez vu précédemment, chaque projet Django peut être constitué de plusieurs applications gérant une logique séparée, et chaque application contient son propre répertoire templates pour stocker les templates HTML associés à l'application.

Cette structure d’application fonctionne bien pour la logique de l’arrière-plan, mais nous souhaitons que l’ensemble de notre site soit cohérent sur le front-end. Au lieu d'avoir à importer des styles Bootstrap dans chaque application, nous pouvons créer un template ou un ensemble de templates partagés par toutes les applications. Tant que Django sait rechercher des templates dans ce nouveau répertoire partagé, il peut enregistrer de nombreux styles répétés.

Dans ce nouveau fichier ( personal\_portfolio/templates/base.html ), ajoutez les lignes de code suivantes :

**HTML**

{% block page\_content %} {% endblock %}

Maintenant, dans hello\_world/templates/hello\_world.html, nous pouvons étendre ce template de base :

**HTML**

{% extends "base.html" %}

{% block page\_content %}

<h1>Hello, World!</h1>

{% endblock %}

Ce qui se passe ici, c’est que tout le code HTML contenu dans le bloc page\_content est ajouté dans le même bloc base.html.

Pour installer Bootstrap dans votre application, vous utiliserez le [CDN Bootstrap](https://realpython.com/get-started-with-django-1/#quick-start) . C'est un moyen très simple d'installer Bootstrap qui consiste simplement à ajouter quelques lignes de code à base.html. Consultez le [code source](https://realpython.com/courses/django-portfolio-project/) pour voir comment ajouter les liens CDN à votre projet.

Tous les templates que nous créerons ultérieurement s’étendront à base.html afin que nous puissions inclure un style Bootstrap sur chaque page sans avoir à importer à nouveau les styles.

Avant de voir notre nouvelle application stylée, nous devons dire à notre projet Django qu'il existe base.html. Les paramètres par défaut enregistrent des répertoires template dans chaque application, mais pas dans le répertoire du projet lui-même. Dans personal\_portfolio/settings.py, mise à jour TEMPLATES :

**Python**

TEMPLATES = [

{

"BACKEND": "django.template.backends.django.DjangoTemplates",

"DIRS": ["personal\_portfolio/templates/"],

"APP\_DIRS": True,

"OPTIONS": {

"context\_processors": [

"django.template.context\_processors.debug",

"django.template.context\_processors.request",

"django.contrib.auth.context\_processors.auth",

"django.contrib.messages.context\_processors.messages",

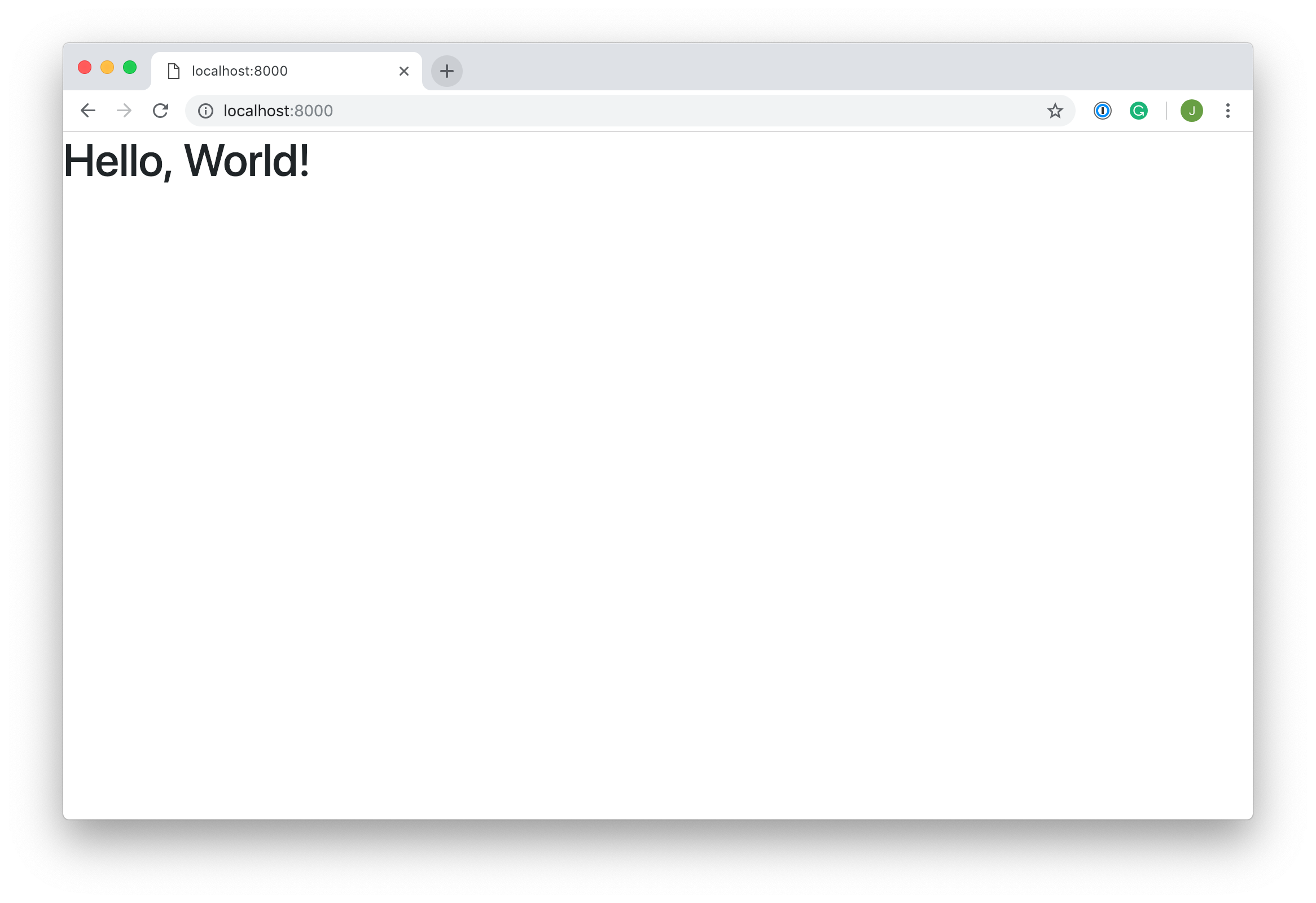
]

},

}

]

Maintenant, lors de votre visite localhost:8000, vous devriez voir que la page a été formatée avec un style légèrement différent :

[](https://github.com/realpython/materials/tree/86c5009b84ef207c6dd1873ec9dc25afbe725347/rp-portfolio)

Chaque fois que vous souhaitez créer des templates ou importer des scripts que vous souhaitez utiliser dans toutes vos applications Django au sein d'un projet, vous pouvez les ajouter à ce répertoire au niveau du projet et les étendre à l'intérieur de vos templates d'application.

L'ajout de templates est la dernière étape pour créer votre Site Django Hello, World! . Vous avez appris le fonctionnement du moteur de templates Django et comment créer des templates au niveau du projet pouvant être partagés par toutes les applications de votre projet Django.

Dans cette section, vous avez appris à créer un simple Site Django Hello, World!  en créant un projet avec une seule application. Dans la section suivante, vous créerez une autre application pour présenter les projets de développement Web et vous apprendrez tout sur les modèles dans Django!

Le code source de cette section est disponible sur [GitHub](https://github.com/realpython/materials/tree/1ea78b8bb71e685c476d6fd98e829b6ad0a42123/rp-portfolio/hello_world) .

## Présentez vos projets

Tout développeur Web cherchant à créer un portfolio a besoin d'un moyen de montrer les projets sur lesquels il a travaillé. C'est ce que vous allez construire maintenant. Vous allez créer une autre application Django appelée projects qui contiendra une série d'exemples de projets qui seront affichés à l'utilisateur. Les utilisateurs peuvent cliquer sur des projets et voir plus d'informations sur votre travail.

Avant de créer l'application projects, supprimons d'abord l'application hello\_world. Tout ce que vous devez faire est de supprimer le répertoire hello\_world et supprimez la ligne "hello\_world", de INSTALLED\_APPS en settings.py :

**Python**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'hello\_world', # Supprimer cette ligne

]

Enfin, vous devez supprimer le chemin d’URL créé dans personal\_portfolio/urls.py :

**Python**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('hello\_world.urls')), # Supprimer cette ligne

]

Maintenant que vous avez supprimé l'application hello\_world, nous pouvons créer l'application projects. Pour vous assurer que vous êtes dans le répertoire rp-portfolio, exécutez la commande suivante dans votre console :

**Shell**

$ python manage.py startapp projects

Cela va créer un répertoire nommé projects. Les fichiers créés sont les mêmes que ceux créés lors de la configuration de l'application hello\_world. Afin de connecter notre application, nous devons l'ajouter INSTALLED\_APPS dans settings.py :

**Shell**

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'projects',

]

Découvrez le [code source](https://github.com/realpython/materials/tree/4de6ebd8baebec950357d4f479fcc0de962e6ae4/rp-portfolio) de cette section sur GitHub. Nous n'allons pas nous inquiéter des URL pour cette application pour l'instant. Au lieu de cela, nous allons nous concentrer sur la construction d'un modèle Project.

### Projets App : Modèles

Si vous souhaitez stocker des données à afficher sur un site Web, vous aurez besoin d'une base de données. En règle générale, si vous souhaitez créer une base de données contenant des tables et des colonnes, vous devez utiliser SQL pour gérer la base de données. Mais lorsque vous utilisez Django, vous n'avez pas besoin d'apprendre un nouveau langage, car celui-ci est doté d'un ORM (Object Relational Mapper) intégré.

Un ORM est un programme qui vous permet de créer des classes qui correspondent à des tables de base de données. Les attributs de classe correspondent aux colonnes et les instances de classes correspondent aux lignes de la base de données. Ainsi, au lieu d'apprendre un tout nouveau langage pour créer notre base de données et ses tables, nous pouvons simplement écrire quelques classes Python.

Lorsque vous utilisez un ORM, les classes que vous construisez et qui représentent des tables de base de données sont appelées **modèles**. Dans Django, ils résident dans le module models.py de chaque application Django.

Dans votre application de projets, vous aurez besoin d'un seul tableau pour stocker les différents projets que vous afficherez à l'utilisateur. Cela signifie que vous n’avez besoin de créer qu’un seul modèle models.py.

Le modèle que vous allez créer sera appelé Project et comportera les champs suivants :

* **title** sera un champ de chaîne courte pour contenir le nom de votre projet.
* **description** sera un champ de chaîne plus grand pour contenir un morceau de texte plus long.
* **technology** sera un champ de chaîne, mais son contenu sera limité à un nombre restreint de choix.
* **image** sera un champ d’image contenant le chemin du fichier où l’image est stockée.

Pour créer ce modèle, nous allons créer une nouvelle classe models.py et ajouter les éléments suivants dans nos champs :

**Python**

from django.db import models

class Project(models.Model):

title = models.CharField(max\_length=100)

description = models.TextField()

technology = models.CharField(max\_length=20)

image = models.FilePathField(path="/img")

Les modèles Django sont livrés avec de nombreux [types de champs de modèles intégrés](https://realpython.com/get-started-with-django-1/) . Nous n'en avons utilisé que trois dans ce modèle. CharField est utilisé pour les chaînes courtes et spécifie une longueur maximale.

TextField est similaire à CharField mais peut être utilisé pour un texte de formulaire plus long car il n’a pas de limite de longueur maximale. Enfin, FilePathField contient également une chaîne mais doit pointer sur un nom de chemin de fichier.

Maintenant que nous avons créé notre classe Project, nous avons besoin de Django pour créer la base de données. Par défaut, l'ORM de Django crée des bases de données en SQLite, mais vous pouvez utiliser d'autres bases de données utilisant le langage SQL, telles que PostgreSQL ou MySQL, avec l'ORM de Django.

Pour démarrer le processus de création de notre base de données, nous devons créer une **migration** . Une migration est un fichier contenant une classe Migration avec des règles qui indiquent à Django les modifications à apporter à la base de données. Pour créer la migration, tapez la commande suivante dans la console, en vous assurant que vous vous trouvez bien dans le répertoire rp-portfolio :

**Shell**

$ python manage.py makemigrations projects

Migrations for 'projects':

projects/migrations/0001\_initial.py

- Create model Project

Vous devriez voir qu'un fichier projects/migrations/0001\_initial.py a été créé dans l'application de projets. Consultez ce fichier dans le [code source](https://realpython.com/get-started-with-django-1/) pour vous assurer que votre migration est correcte.

Maintenant que vous avez créé un fichier de migration, vous devez appliquer les migrations définies dans le fichier de migrations et créer votre base de données à l'aide de la commande migrate :

**Shell**

$ python manage.py migrate projects

Operations to perform:

Apply all migrations: projects

Running migrations:

Applying projects.0001\_initial... OK

**Remarque :** lors de l'exécution des commandes makemigrations et migrate, nous avons ajouté projects à notre commande. Cela indique à Django de ne regarder que les modèles et les migrations de l'application projects. Django est livré avec plusieurs modèles déjà créés.

Si vous exécutez makemigrations et migrate sans l'indicateur projects, toutes les migrations pour tous les modèles par défaut de vos projets Django seront créées et appliquées. Ce n'est pas un problème, mais pour les besoins de cette section, ils ne sont pas nécessaires.

Vous devriez également voir qu'un fichier appelé db.sqlite3 a été créé à la racine de votre projet. Maintenant, votre base de données est configurée et prête à fonctionner. Vous pouvez maintenant créer dans votre table des lignes contenant les différents projets que vous souhaitez afficher sur votre site portfolio.

Pour créer des instances de notre classe Project, nous allons devoir utiliser le shell Django. Le shell Django est similaire au shell Python, mais vous permet d'accéder à la base de données et de créer des entrées. Pour accéder au shell Django, nous utilisons une autre commande de gestion Django :

**Shell**

$ python manage.py shell

Une fois que vous avez accédé au shell, vous remarquerez que l'invite de commande passera de $ à >>>. Vous pouvez ensuite importer vos modèles :

**Python** >>>

>>> from projects.models import Project

Nous allons d’abord créer un nouveau projet avec les attributs suivants :

* **name:** My First Project
* **description:** A web development project.
* **technology:** Django
* **image:** img/project1.png

Pour ce faire, nous créons une instance de la classe Project dans le shell Django :

**Python** >>>

>>> p1 = Project(

... title='My First Project',

... description='A web development project.',

... technology='Django',

... image='img/project1.png'

... )

>>> p1.save()

Cela crée une nouvelle entrée dans votre table de projets et l’enregistre dans la base de données. Vous avez maintenant créé un projet que vous pouvez afficher sur le site de votre portfolio.

La dernière étape de cette section consiste à créer deux exemples de projets supplémentaires :

**Python** >>>

>>> p2 = Project(

... title='My Second Project',

... description='Another web development project.',

... technology='Flask',

... image='img/project2.png'

... )

>>> p2.save()

>>> p3 = Project(

... title='My Third Project',

... description='A final development project.',

... technology='Django',

... image='img/project3.png'

... )

>>> p3.save()

Bravo d'avoir atteint la fin de cette section ! Vous savez maintenant comment créer des modèles dans Django et construire des fichiers de migration afin de pouvoir traduire ces classes de modèles en tables de base de données. Vous avez également utilisé le shell Django pour créer trois instances de votre classe de modèle.

Dans la section suivante, nous allons prendre ces trois projets que vous avez créés et créer une fonction d'affichage pour les afficher aux utilisateurs sur une page Web. Vous pouvez trouver le [code source](https://realpython.com/get-started-with-django-1/) de cette section du tutoriel sur GitHub.

### Projets App : Vues

Maintenant que vous avez créé les projets à afficher sur votre site portfolio, vous devez créer des fonctions d'affichage pour envoyer les données de la base de données aux templates HTML.

Dans l'application projects, vous créez deux vues différentes :

1. Une vue index contenant un extrait d'informations sur chaque projet
2. Une vue détaillée qui montre plus d'informations sur un sujet particulier

Commençons par la vue index car la logique est légèrement plus simple. À l'intérieur de views.py, vous devrez importer la classe Project depuis models.py et créer une fonction project\_index() permettant de rendre un template appelé project\_index.html. Dans le corps de cette fonction, vous allez créer une requête Django ORM pour sélectionner tous les objets de la table projects :

**Python**

1 from django.shortcuts import render

2 from projects.models import Project

3

4 def project\_index(request):

5 projects = Project.objects.all()

6 context = {

7 'projects': projects

8 }

9 return render(request, 'project\_index.html', context)

Il se passe beaucoup de choses dans ce bloc de code, alors décomposons-le.

À la **ligne 5**, vous effectuez une **requête**. Une requête est simplement une commande qui vous permet de créer, d'extraire, de mettre à jour ou de supprimer des objets (ou des lignes) dans votre base de données. Dans ce cas, vous récupérez tous les objets de la table projects.

Une requête de base de données renvoie une collection de tous les objets correspondant à la requête, appelée ensemble de **requêtes**. Dans ce cas, vous souhaitez que tous les objets de la table soient renvoyés, ce qui renverra une collection de tous les projets.

À la **ligne 6** du bloc de code ci-dessus, nous définissons un dictionnaire context. Le dictionnaire n'a qu'une entrée projects à laquelle nous affectons notre ensemble de requêtes contenant tous les projets. Le dictionnaire de contexte est utilisé pour envoyer des informations à notre template. Chaque fonction de vue que vous créez doit avoir un dictionnaire de contexte.

À la **ligne 9**, context est ajouté en tant qu’argument de render(). Toutes les entrées du dictionnaire context sont disponibles dans le template, tant que l'argument context est passé à render(). Vous devrez créer un dictionnaire de contexte et le transmettre à render dans chaque fonction de vue que vous créez.

Nous rendons également un template nommé project\_index.html, qui n'existe pas encore. Ne vous inquiétez pas pour ça pour l'instant. Vous allez créer les templates pour ces vues dans la section suivante.

Ensuite, vous devrez créer la fonction d'affichage project\_detail(). Cette fonction nécessitera un argument supplémentaire : l'id du projet en cours de visualisation.

Sinon, la logique est similaire :

**Python**

13 def project\_detail(request, pk):

14 project = Project.objects.get(pk=pk)

15 context = {

16 'project' : project

17 }

18 return render(request, 'project\_detail.html', context)

À la **ligne 14**, nous effectuons une autre requête. Cette requête récupère le projet avec la clé primaire pk, égale à celle de l'argument de la fonction. Nous assignons ensuite ce projet dans notre dictionnaire context, auquel nous passons render(). Encore une fois, il y a un template project\_detail.html que nous n'avons pas encore créé.

Une fois vos fonctions de vue créées, nous devons les relier à des URL. Nous allons commencer par créer un fichier projects/urls.py contenant la configuration de l'URL pour l'application. Ce fichier doit contenir le code suivant :

**Python**

1 from django.urls import path

2 from . import views

3

4 urlpatterns = [

5 path("", views.project\_index, name="project\_index"),

6 path("<int:pk>/", views.project\_detail, name="project\_detail"),

7 ]

À la **ligne 5**, nous connectons l'URL racine de notre application à la vue project\_index. Il est légèrement plus compliqué de brancher la vue project\_detail. Pour ce faire, nous voulons que l'URL soit /1, ou /2, et ainsi de suite, en fonction de pk du projet.

La valeur de pk dans l'URL est identique à celle de pk transmise à la fonction d'affichage. Vous devez donc générer dynamiquement ces URL en fonction du projet que vous souhaitez afficher. Pour ce faire, nous avons utilisé la notation <int:pk>. Cela indique simplement à Django que la valeur transmise dans l'URL est un entier et que son nom de variable est pk.

Avec ceux qui sont maintenant configurés, nous devons relier ces URL aux URL du projet. Dans personal\_portfolio/urls.py, ajoutez la ligne de code en surbrillance suivante :

**Python**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("admin/", admin.site.urls),

path("projects/", include("projects.urls")),

]

Cette ligne de code inclut toutes les URL de l’application de projets, mais signifie qu’elles sont accessibles par le préfixe projects/. Il existe maintenant deux URL complètes accessibles avec notre projet :

* **localhost:8000/projects:** La page index du projet
* **localhost:8000/projects/3:** La vue détaillée du projet avec pk=3

Ces URL ne fonctionnent toujours pas correctement car nous n’avons aucun template HTML. Mais nos vues et notre logique sont opérationnels et il ne reste plus qu'à créer ces templates. Si vous voulez vérifier votre code, jetez un coup d'œil au [code source](https://github.com/realpython/materials/tree/d49c624500e4470107bb258642d2744eeeeefc5a/rp-portfolio) de cette section.

### Application Projets : Templates

Enfin ! Vous êtes presque au bout avec cette application. Notre dernière étape consiste à créer deux templates :

1. Le template project\_index
2. Le template project\_detail

Comme nous avons ajouté des styles Bootstrap à notre application, nous pouvons utiliser certains composants pré-stylés pour rendre les vues plus belles. Commençons par le template project\_index.

Pour le template project\_index, vous allez créer une grille de [cartes Bootstrap](https://getbootstrap.com/docs/4.1/components/card/), chaque carte affichant les détails du projet. Bien sûr, nous ne savons pas combien de projets il va y avoir. En théorie, il pourrait y en avoir des centaines à afficher.

Nous ne voulons pas avoir à créer 100 cartes Bootstrap différentes et à coder en dur toutes les informations relatives à chaque projet. Au lieu de cela, nous allons utiliser une fonctionnalité du moteur de gabarit Django: les [**boucles for**](https://realpython.com/tutorials/django/) .

En utilisant cette fonctionnalité, vous pourrez parcourir tous les projets et créer une carte pour chacun. La syntaxe de la boucle for dans le moteur de template Django est la suivante :

**HTML**

{% for project in projects %}

{# Do something... #}

{% endfor %}

Maintenant que vous savez comment fonctionnent les boucles for, vous pouvez ajouter le code suivant à un fichier nommé projects/templates/project\_index.html :

**HTML**

1 {% extends "base.html" %}

2 {% load static %}

3 {% block page\_content %}

4 <h1>Projects</h1>

5 <div class="row">

6 {% for project in projects %}

7 <div class="col-md-4">

8 <div class="card mb-2">

9 <img class="card-img-top" src="{% static project.image %}">

10 <div class="card-body">

11 <h5 class="card-title">{{ project.title }}</h5>

12 <p class="card-text">{{ project.description }}</p>

13 <a href="{% url 'project\_detail' project.pk %}"

14 class="btn btn-primary">

15 Read More

16 </a>

17 </div>

18 </div>

19 </div>

20 {% endfor %}

21 </div>

22 {% endblock %}

Il y a beaucoup de code HTML Bootstrap ici, ce qui n'est pas l'objet de ce tutoriel. N'hésitez pas à copier-coller et à jeter un coup d'œil aux documents Bootstrap si vous souhaitez en apprendre davantage. Au lieu de vous concentrer sur le Bootstrap, il y a quelques points à souligner dans ce bloc de code.

En **ligne 1**, nous étendons base.html comme dans le tutoriel d'application Hello, World! . J'ai ajouté un peu plus de style à ce fichier pour inclure une [barre de navigation](https://github.com/realpython/materials/blob/4dd5d79634efbffeb8999052a9e94b3dba4b25ba/rp-portfolio/personal_portfolio/templates/base.html) et pour que tout le contenu soit dans un [conteneur Bootstrap](https://files.realpython.com/media/Screenshot_2018-12-16_at_16.50.59.43297d79452f.png#containers) . Les modifications de base.html peuvent être vues dans le [code source](https://github.com/realpython/materials/blob/get-started-with-django-part-1/rp-portfolio/personal_portfolio/templates/base.html) sur GitHub.

Sur la **ligne 2**, nous incluons une balise {% load static %} pour inclure des fichiers statiques tels que des images. Rappelez-vous dans la section sur les modèles Django, lorsque vous avez créé le modèle Project. Un de ses attributs était un chemin de fichier. Ce chemin de fichier est l'endroit où nous allons stocker les images réelles pour chaque projet.

Django enregistre automatiquement les fichiers statiques stockés dans un répertoire nommé static/ dans chaque application. Nos noms de chemin de fichier d'image sont de la structure : img/<photo\_name>.png.

Lors du chargement de fichiers statiques, Django recherche dans le répertoire static/ les fichiers correspondant à un chemin de fichier static/ donné. Nous devons donc créer un répertoire nommé static/ avec un autre répertoire nommé img/ à l'intérieur. À l'intérieur img/, vous pouvez copier les images du [code source](https://github.com/realpython/materials/tree/7909f5a682a88d8488167bc6fe9b64a5b294f99a/rp-portfolio/projects/static/img) sur GitHub.

Sur la **ligne 6**, nous commençons la boucle for en passant en revue tous les projets transmis par le dictionnaire context.

Dans cette boucle, nous pouvons accéder à chaque projet individuel. Pour accéder aux attributs du projet, vous pouvez utiliser la notation pointée entre double accolades. Par exemple, pour accéder au titre du projet, vous utilisez {{ project.title }}. La même notation peut être utilisée pour accéder à n’importe quel attribut du projet.

Sur la **ligne 9**, nous incluons notre image de projet. À l'intérieur de l'attribut src, nous ajoutons le code {% static project.image %}. Ceci dit à Django de chercher dans les fichiers statiques pour trouver un fichier correspondant à project.image.

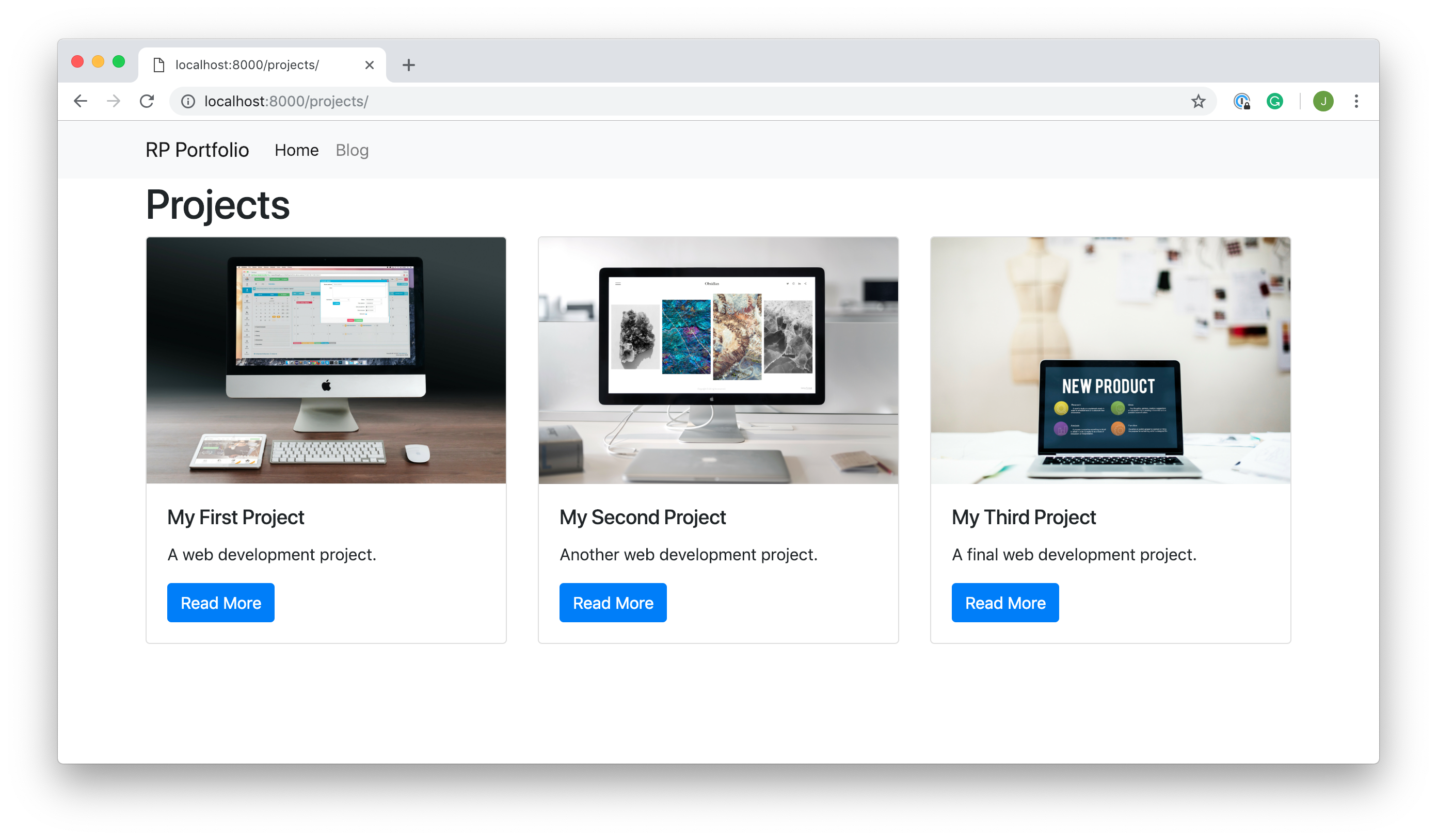
Le dernier point que nous devons souligner est le lien de la **ligne 13**. Ceci est le lien vers notre page project\_detail. Accéder aux URL dans Django revient à accéder à des fichiers statiques. Le code de l'URL a la forme suivante :

**HTML**

{% url '<url path name>' <view\_function\_arguments> %}

Dans ce cas, nous accédons à un chemin d’URL nommé project\_detail, qui prend des arguments entiers correspondant au numéro pk du projet.

Avec tout ce qui est en place, si vous démarrez le serveur Django et que vous visitez localhost:8000/projects, vous devriez voir quelque chose comme ceci :

[](https://stackoverflow.com/questions/tagged/django)

Avec le template project\_index.html en place, il est temps de créer le template project\_detail.html. Le code de ce template est ci-dessous :

**HTML**

{% extends "base.html" %}

{% load static %}

{% block page\_content %}

<h1>{{ project.title }}</h1>

<div class="row">

<div class="col-md-8">

<img src="{% static project.image %}" alt="" width="100%">

</div>

<div class="col-md-4">

<h5>About the project:</h5>

<p>{{ project.description }}</p>

<br>

<h5>Technology used:</h5>

<p>{{ project.technology }}</p>

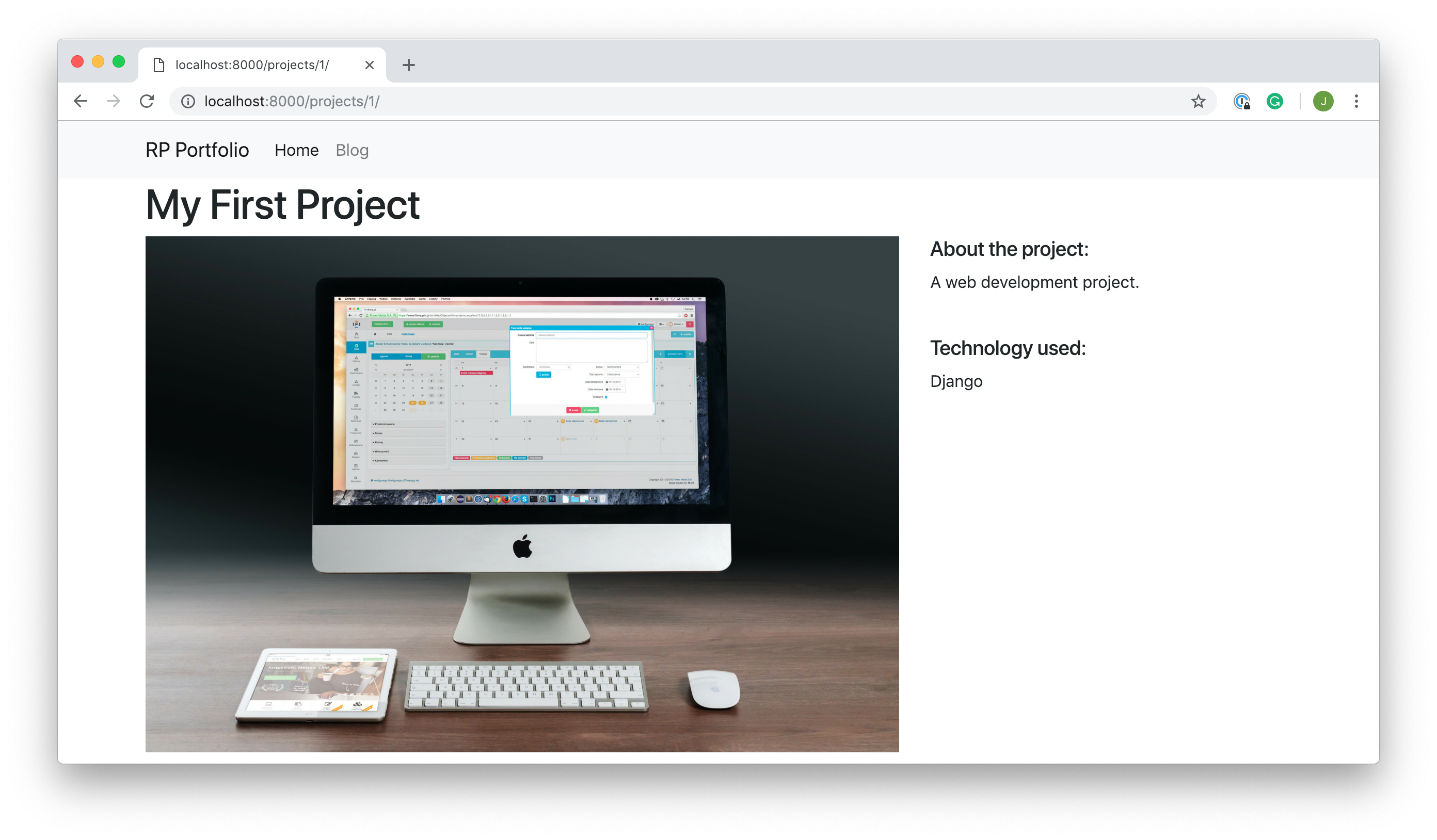
</div>

</div>

{% endblock %}

Le code de ce template a les mêmes fonctionnalités que chaque fiche de projet du template project\_index.html. La seule différence est l'introduction de certaines colonnes Bootstrap.

Si vous visitez localhost:8000/projects/1, vous devriez voir la page de détail de ce premier projet que vous avez créé :

[](https://files.realpython.com/media/Screenshot_2018-12-09_at_19.46.23.cd4783f33899.png)

Dans cette section, vous avez appris à utiliser des modèles, des vues et des templates pour créer une application entièrement fonctionnelle pour votre projet de portfolio personnel. Découvrez le [code source](https://github.com/realpython/materials/tree/7909f5a682a88d8488167bc6fe9b64a5b294f99a/rp-portfolio) de cette section sur GitHub.

Dans la section suivante, vous créerez un blog entièrement fonctionnel pour votre site et vous découvrirez également la page d'administration et les formulaires de Django.

## Partagez vos connaissances avec un blog

Un blog est un excellent ajout à tout site de portfolio personnel. Que vous le mettiez à jour tous les mois ou toutes les semaines, c'est un excellent endroit pour partager vos connaissances au fur et à mesure de votre apprentissage. Dans cette section, vous allez créer un blog entièrement fonctionnel qui vous permettra d'effectuer les tâches suivantes :

* Créer, mettre à jour et supprimer des articles de blog
* Afficher les publications destinées à l'utilisateur sous forme d'index ou de détail
* Attribuer des catégories aux articles
* Autoriser les utilisateurs à commenter les messages

Vous apprendrez également à utiliser l'interface d'administration de Django, à partir de laquelle vous pourrez créer, mettre à jour et supprimer des publications et des catégories, le cas échéant.

Avant de vous lancer dans la construction de la fonctionnalité de cette partie de votre site, créez une nouvelle application Django nommée blog. Ne supprimez pas projects. Vous voudrez les deux applications dans votre projet Django :

**Shell**

$ python manage.py startapp blog

Cela peut vous sembler familier, comme c'est la troisième fois que vous le faites. N'oubliez pas d'ajouter blog à votre INSTALLED\_APPS dans personal\_porfolio/settings.py :

**Python**

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

"projects",

"blog",

]

Attendez de brancher les URL pour l’instant. Comme avec l'application projects, vous allez commencer par ajouter vos modèles.

### Blog App : Modèles

Le fichier models.py dans cette application est beaucoup plus compliqué que dans l'application projects.

Vous aurez besoin de trois tables de base de données distinctes pour le blog :

1. Post
2. Category
3. Comment

Ces tables doivent être liées les unes aux autres. Ceci est facilité car les modèles Django sont livrés avec des champs spécialement conçus à cet effet.

Ci-dessous le code pour les modèles Category et Post :

**Python**

1 from django.db import models

2

3 class Category(models.Model):

4 name = models.CharField(max\_length=20)

5

6 class Post(models.Model):

7 title = models.CharField(max\_length=255)

8 body = models.TextField()

9 created\_on = models.DateTimeField(auto\_now\_add=True)

10 last\_modified = models.DateTimeField(auto\_now=True)

11 categories = models.ManyToManyField('Category', related\_name='posts')

Le modèle Category est très simple. Tout ce qui est nécessaire est un single CharField dans lequel nous stockons le nom de la catégorie.

Les champs title et body du modèle Post sont les mêmes types de champs que ceux que vous avez utilisés dans le modèle Project. Nous avons seulement besoin d'un CharField pour le title car nous voulons seulement une chaîne de caractère courte pour le titre du post. Le corps doit être un texte long, nous utilisons donc un TextField.

Les deux champs suivants, created\_on et last\_modified, sont en Django des DateTimeFields. Ceux-ci stockent un objet [datetime](https://getbootstrap.com/docs/4.1/getting-started/introduction/) contenant la date et l'heure de création et de modification de la publication.

À la **ligne 9**, le DateTimeField prend un argument auto\_now\_add=True. Cela affecte la date et l'heure actuelles à ce champ chaque fois qu'une instance de cette classe est créée.

Sur la **ligne 10**, le DateTimeField prend un argument auto\_now=True. Cela affecte la date et l'heure actuelles à ce champ chaque fois qu'une instance de cette classe est enregistrée. Cela signifie que chaque fois que vous éditez une instance de cette classe, la date\_modified est mis à jour.

Le dernier champ du modèle de publication est le plus intéressant. Nous souhaitons lier nos modèles de catégories et de publications de manière à ce que de nombreuses catégories puissent être affectées à de nombreuses publications. Heureusement, Django nous facilite la tâche en fournissant un type de champ ManytoManyField. Ce champ relie les modèles Post et Category et nous permet de créer une relation entre les deux tables.

Le ManyToManyField prend deux arguments. Le premier est le modèle avec lequel il y a la relation, dans ce cas sa Category. Le second nous permet d’accéder à la relation à partir d’un objet Category, même si nous n’avons pas ici ajouté de champ. En ajoutant un related\_name aux posts, nous pouvons accéder à category.posts pour nous donner une liste de messages de cette catégorie.

Le troisième et dernier modèle que nous devons ajouter est Comment. Nous allons utiliser un autre champ de relation similaire à celui de ManyToManyField qui concerne Post et Category. Cependant, nous voulons seulement que la relation aille dans un sens : un poste devrait avoir de nombreux commentaires.

Vous verrez comment cela fonctionne après avoir défini la classe Comment :

**Python**

16 class Comment(models.Model):

17 author = models.CharField(max\_length=60)

18 body = models.TextField()

19 created\_on = models.DateTimeField(auto\_now\_add=True)

20 post = models.ForeignKey('Post', on\_delete=models.CASCADE)

Les trois premiers champs de ce modèle doivent sembler familiers. Il existe un champ author permettant aux utilisateurs d'ajouter un nom ou un alias, un champ body pour le corps du commentaire et un champ created\_on identique au champ created\_on du modèle Post.

Sur la **ligne 20**, nous utilisons un autre champ relationnel, le champ ForeignKey. Ceci est similaire à la relation ManyToManyField mais à la place elle définit **une** relation **plusieurs à un**. Le raisonnement derrière cela est que **plusieurs** commentaires peuvent être assignés à **un** post. Mais vous ne pouvez pas avoir un commentaire qui correspond à plusieurs posts.

Le champ ForeignKey prend deux arguments. Le premier est l’autre modèle de la relation, dans ce cas, Post. Le second indique à Django ce qu’il faut faire lorsqu’un message est supprimé. Si un message est supprimé, nous ne voulons pas que les commentaires qui s'y rapportent traînent. Nous souhaitons donc également les supprimer, nous ajoutons donc l'argument on\_delete=models.CASCADE.

Une fois que vous avez créé les modèles, vous pouvez créer les fichiers de migration avec makemigrations :

**Shell**

$ python manage.py makemigrations blog

La dernière étape consiste à migrer les tables. Cette fois, n’ajoutez pas le flag spécifique à l’application. Plus tard, vous aurez besoin du modèle User créé par Django pour vous :

**Shell**

$ python manage.py migrate

Maintenant que vous avez créé les modèles, nous pouvons commencer à ajouter des publications et des catégories. Vous ne ferez pas cela depuis la ligne de commande comme vous l'avez fait avec les projets, car taper un article de blog entier dans la ligne de commande serait désagréable pour le moins !

Au lieu de cela, vous apprendrez à utiliser l’Admin de Django, ce qui vous permettra de créer des instances de vos classes de modèle dans une interface Web conviviale.

N'oubliez pas que vous pouvez consulter le [code source](https://realpython.com/tutorials/web-dev/) de cette section sur GitHub avant de passer à la section suivante.

### Application de blog : Admin de Django

Admin de Django est un outil fantastique et l’un des grands avantages de l’utilisation de Django. Étant donné que vous êtes la seule personne à rédiger des articles de blog et à créer des catégories, vous n'avez pas besoin de créer une interface utilisateur pour le faire.

D'autre part, vous ne voulez pas avoir à écrire des articles de blog en ligne de commande. C’est là que l’administrateur entre en jeu. Il vous permet de créer, mettre à jour et supprimer des instances de vos classes de modèle et fournit une interface agréable à cet effet.

Avant de pouvoir accéder à l'admin, vous devez vous ajouter en tant que super utilisateur. C'est pourquoi, dans la section précédente, vous avez appliqué les migrations à l'ensemble du projet et non uniquement à l'application. Django est fourni avec des modèles d'utilisateur intégrés et un système de gestion des utilisateurs vous permettant de vous connecter à l'admin.

Pour commencer, vous pouvez vous ajouter en tant que super utilisateur à l'aide de la commande suivante :

**Shell**

$ python manage.py createsuperuser

Vous serez ensuite invité à entrer un nom d'utilisateur suivi de votre adresse e-mail et de votre mot de passe. Une fois que vous avez entré les détails requis, vous serez informé que le super utilisateur a été créé. Ne vous inquiétez pas si vous faites une erreur car vous pouvez simplement recommencer :

**Shell**

Username (leave blank to use 'jasmine'): jfiner

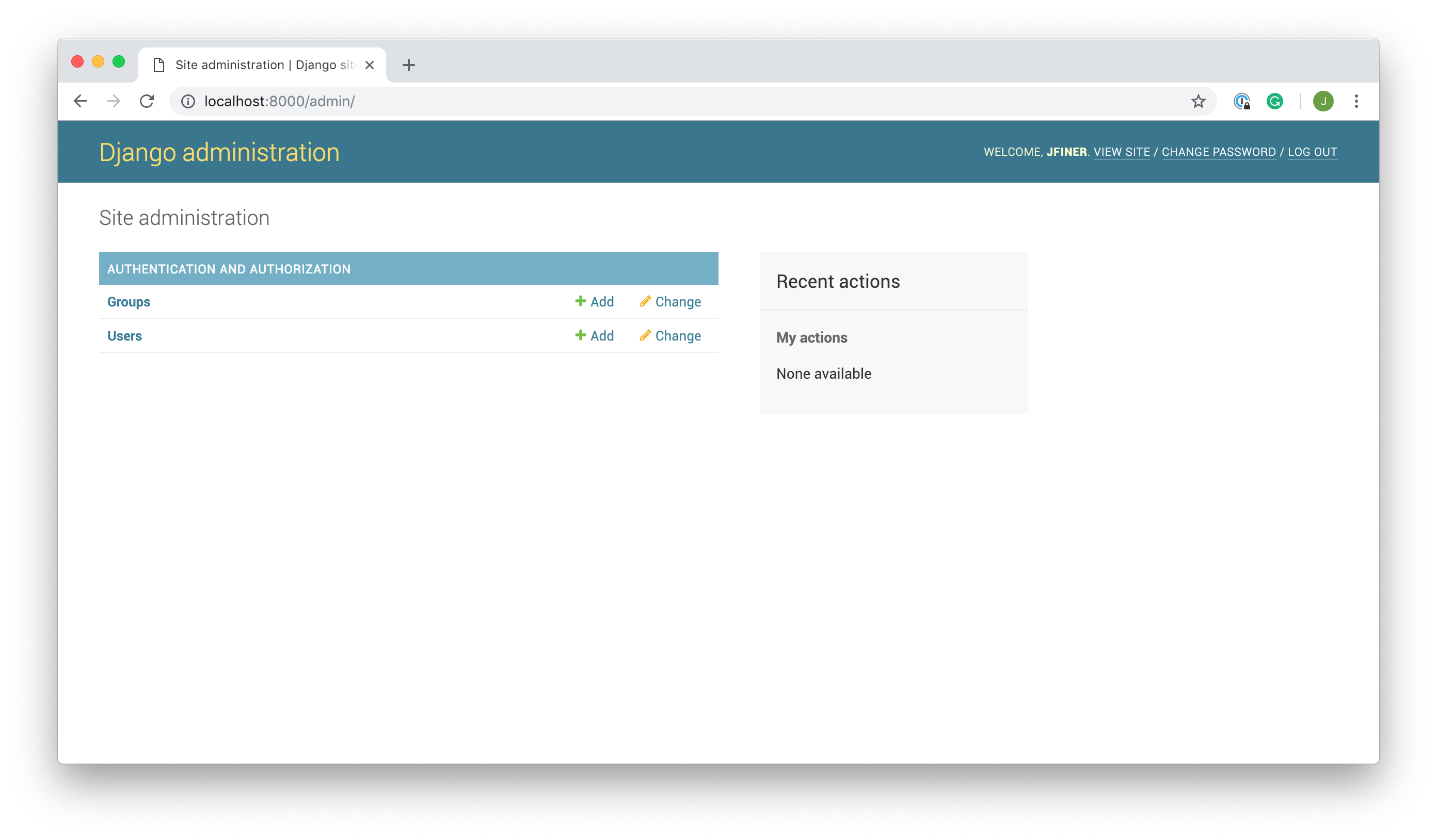
Email address: jfiner@example.com

Password:

Password (again):

Superuser created successfully.

Accédez à localhost:8000/admin et connectez-vous avec les informations d'identification que vous venez d'utiliser pour créer un super utilisateur. Vous verrez une page similaire à celle ci-dessous :

[](https://realpython.com/get-started-with-django-1/)

Les modèles d'utilisateur et de groupe doivent apparaître, mais vous remarquerez qu'il n'y a aucune référence aux modèles que vous avez créés. C'est parce que vous devez les enregistrer dans l'admin.

Dans le répertoire blog, ouvrez le fichier admin.py et tapez les lignes de code suivantes :

**Python**

1 from django.contrib import admin

2 from blog.models import Post, Category

3

4 class PostAdmin(admin.ModelAdmin):

5 pass

6

7 class CategoryAdmin(admin.ModelAdmin):

8 pass

9

10 admin.site.register(Post, PostAdmin)

11 admin.site.register(Category, CategoryAdmin)

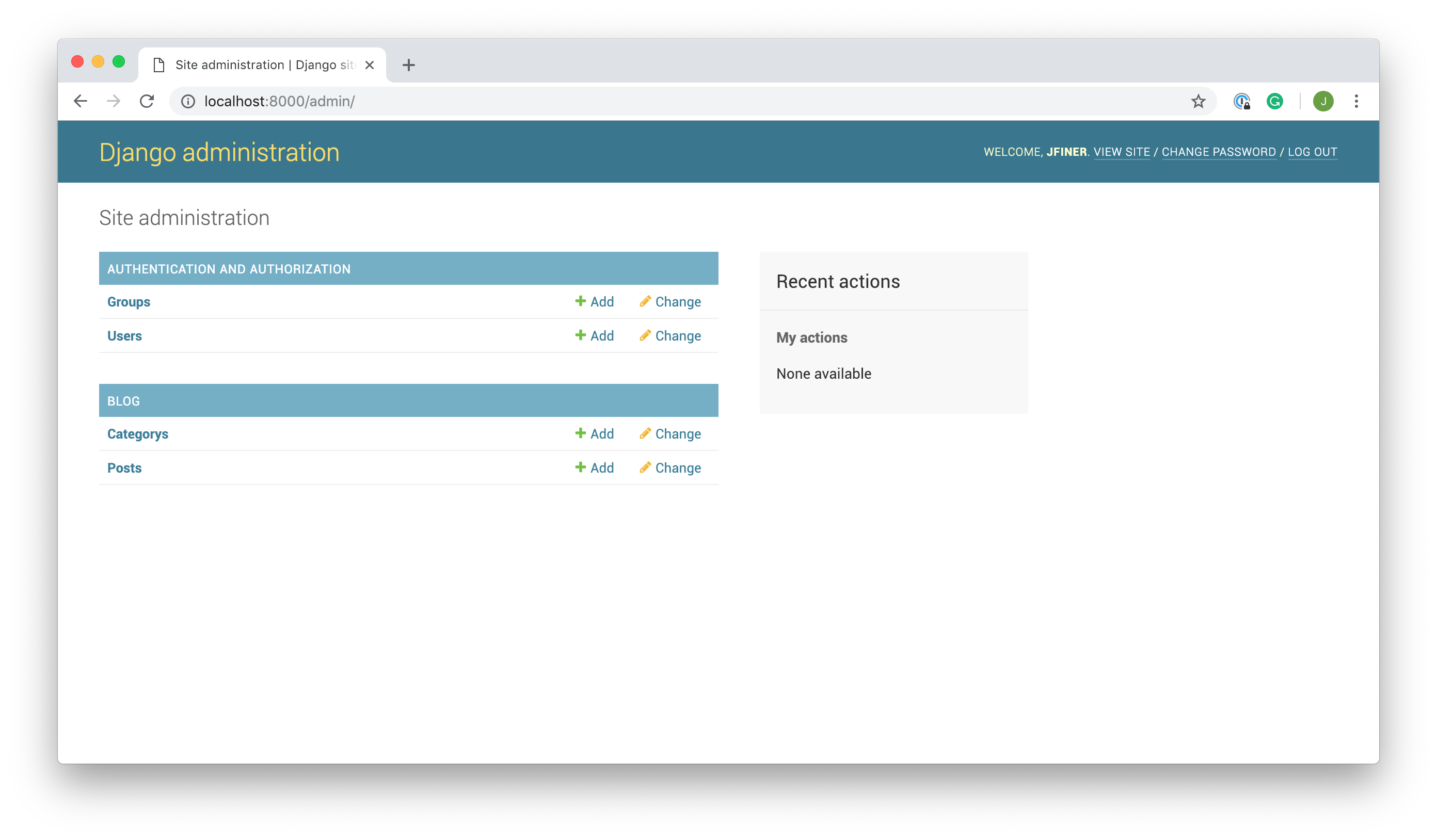
Sur la **ligne 2**, vous importez les modèles que vous souhaitez enregistrer sur la page d'administration.

**Remarque :** nous n’ajoutons pas les commentaires à l’admin. En effet, il n'est généralement pas nécessaire de modifier ou de créer des commentaires vous-même.

Si vous souhaitez ajouter une fonctionnalité dans laquelle les commentaires sont modérés, continuez et ajoutez également le modèle Commentaires. Les étapes à suivre sont exactement les mêmes !

Sur la **ligne 4** et la **ligne 7**, vous définissez les classes vides PostAdmin et CategoryAdmin. Pour les besoins de ce tutoriel, vous n'avez pas besoin d'ajouter d'attributs ou de méthodes à ces classes. Ils sont utilisés pour personnaliser ce qui est affiché sur les pages d’administration. Pour ce tutoriel, la configuration par défaut suffit.

Les deux dernières lignes sont les plus importantes. Ceux-ci enregistrent les modèles avec les classes admin. Si vous visitez maintenant localhost:8000/admin, alors vous devriez voir que les modèles Post et Category sont maintenant visibles :

[](https://files.realpython.com/media/Screenshot_2018-12-16_at_19.19.20.297b0ccfe2f3.png)

Si vous cliquez sur Articles ou Catégories, vous devriez pouvoir ajouter de nouvelles instances des deux modèles. J'aime ajouter du texte aux faux posts du blog en utilisant du texte factice en [lorem ipsum](https://realpython.com/get-started-with-django-1/) .

Créez quelques faux posts et assignez-leur de fausses catégories avant de passer à la section suivante. De cette façon, vous aurez des posts que vous pourrez visualiser lors de la création de nos templates.

N'oubliez pas de consulter le [code source](https://www.djangoproject.com/) de cette section avant de commencer à créer des vues pour notre application.

### Application Blog: Vues

Vous aurez besoin de créer trois fonctions d'affichage dans le fichier views.py du répertoire blog :

* **blog\_index** affichera une liste de tous vos posts.
* **blog\_detail** affichera l'intégralité du posts, ainsi que des commentaires et un formulaire permettant aux utilisateurs de créer de nouveaux commentaires.
* **blog\_category** sera similaire à blog\_index, mais les posts visionnés ne seront que d'une catégorie spécifique choisie par l'utilisateur.

La fonction de la vue la plus simple pour commencer est blog\_index(). Ce sera très similaire à la vue project\_index() de votre application project. Vous allez simplement interroger les modèles Post et récupérer tous ses objets :

**Python**

1 from django.shortcuts import render

2 from blog.models import Post

3

4 def blog\_index(request):

5 posts = Post.objects.all().order\_by('-created\_on')

6 context = {

7 "posts": posts,

8 }

9 return render(request, "blog\_index.html", context)

Sur la **ligne 2**, vous importez le modèle Post et sur la **ligne 5 de** la fonction d'affichage, vous obtenez un Queryset contenant tous les posts de la base de données. order\_by() ordonne le Queryset en fonction de l'argument donné. Le signe moins indique à Django de commencer par la plus grande valeur plutôt que par la plus petite. Nous l'utilisons car nous souhaitons que les posts soient ordonnés avec le post le plus récent en premier.

Enfin, vous définissez le dictionnaire context et restituez le template. Ne le créer pas pour le moment. Vous arriverez à le créer dans la section suivante.

Ensuite, vous pouvez commencer à créer la vue blog\_category(). La fonction d'affichage devra prendre un nom de catégorie en tant qu'argument et interroger la base de données Post pour tous les posts auxquels la catégorie donnée a été attribuée :

**Python**

13 def blog\_category(request, category):

14 posts = Post.objects.filter(

15 categories\_\_name\_\_contains=category

16 ).order\_by(

17 '-created\_on'

18 )

19 context = {

20 "category": category,

21 "posts": posts

22 }

23 return render(request, "blog\_category.html", context)

Sur la **ligne 14**, vous avez utilisé un [filtre Django Queryset](https://realpython.com/python-virtual-environments-a-primer/#retrieving-specific-objects-with-filters) . L'argument du filtre indique à Django quelles conditions doivent être remplies pour qu'un objet soit récupéré. Dans ce cas, nous ne voulons que les posts dont les catégories contiennent la catégorie dont le nom correspond à celui indiqué dans l'argument de la fonction vue. Encore une fois, vous utilisez order\_by() pour ordonner les posts commençant par le plus récent.

Nous ajoutons ensuite ces posts et la catégorie au dictionnaire context et restituons notre template.

La dernière fonction de vue à ajouter est blog\_detail(). Ceci est plus compliqué car nous allons inclure un formulaire. Avant d'ajouter le formulaire, configurez simplement la fonction d'affichage pour afficher un post spécifique avec un commentaire qui lui est associé. Cette fonction sera presque équivalente à la fonction de la vue project\_detail() dans l'application projects :

**Python**

21 def blog\_detail(request, pk):

22 post = Post.objects.get(pk=pk)

23 comments = Comment.objects.filter(post=post)

24 context = {

25 "post": post,

26 "comments": comments,

27 }

28

29 return render(request, "blog\_detail.html", context)

La fonction de la vue prend une valeur pk en tant qu'argument et, à la **ligne 22**, récupère l'objet avec la donnée pk.

À la **ligne 23**, nous récupérons tous les commentaires attribués au post donné en utilisant à nouveau les filtres Django.

Enfin, ajoutez les deux, post et comments au dictionnaire context et restituez le template.

Pour ajouter un formulaire à la page, vous devez créer un autre fichier dans le répertoire blog nommé forms.py. Les formes Django ressemblent beaucoup aux modèles. Un formulaire consiste en une classe où les attributs de classe sont des champs de formulaire. Django est livré avec des champs de formulaire intégrés que vous pouvez utiliser pour créer rapidement le formulaire dont vous avez besoin.

Pour ce formulaire, les seuls champs dont vous aurez besoin sont author, qui devrait être un CharField, et body, qui peut aussi être un CharField.

**Remarque :** Si le CharField de votre formulaire correspond à un modèle CharField, assurez-vous que les deux ont la même valeur max\_length.

blog/forms.py devrait contenir le code suivant :

**Python**

from django import forms

class CommentForm(forms.Form):

author = forms.CharField(

max\_length=60,

widget=forms.TextInput(attrs={

"class": "form-control",

"placeholder": "Your Name"

})

)

body = forms.CharField(widget=forms.Textarea(

attrs={

"class": "form-control",

"placeholder": "Leave a comment!"

})

)

Vous remarquerez également qu'un argument widget a été passé aux deux champs. Le champ author a le widget forms.TextInput. Cela indique à Django de charger ce champ en tant qu'élément de saisie de texte HTML dans les templates. Le champ body utilise à la place un widget forms.TextArea, de sorte qu'il est restitué sous la forme d'un élément de zone de texte HTML.

Ces widgets prennent également un argument attrs, qui est un dictionnaire et nous permet de spécifier des classes CSS, ce qui facilitera la mise en forme du template pour cette vue ultérieurement. Cela nous permet également d’ajouter du texte de substitution.

Lorsqu'un formulaire est posté, une demande POST est envoyée au serveur. Donc, dans la fonction de vue, nous devons vérifier si une demande POST a été reçue. Nous pouvons ensuite créer un commentaire à partir des champs de formulaire. Django est livré avec un outil pratique is\_valid() sur ses formulaires, nous pouvons donc vérifier que tous les champs ont été correctement saisis.

Une fois que vous avez créé le commentaire à partir du formulaire, vous devez l'enregistrer à l'aide de save(), puis interroger la base de données pour connaître tous les commentaires affectés au postdonné. Votre fonction de vue doit contenir le code suivant :

**Python**

21 def blog\_detail(request, pk):

22 post = Post.objects.get(pk=pk)

23

24 form = CommentForm()

25 if request.method == 'POST':

26 form = CommentForm(request.POST)

27 if form.is\_valid():

28 comment = Comment(

29 author=form.cleaned\_data["author"],

30 body=form.cleaned\_data["body"],

31 post=post

32 )

33 comment.save()

34

35 comments = Comment.objects.filter(post=post)

36 context = {

37 "post": post,

38 "comments": comments,

39 "form": form,

40 }

41 return render(request, "blog\_detail.html", context)

Sur la **ligne 24**, nous créons une instance de notre classe de formulaire. N'oubliez pas d'importer votre formulaire au début du fichier :

**Python**

from .forms import CommentForm

Nous vérifions ensuite si une demande POST a été reçue. Si tel est le cas, nous créons une nouvelle instance de notre formulaire, contenant les données saisies dans le formulaire.

Le formulaire est ensuite validé avec is\_valid(). Si le formulaire est valide, une nouvelle instance de Comment est créée. Vous pouvez accéder aux données du formulaire en utilisant form.cleaned\_data, qui est un dictionnaire.

Les clés du dictionnaire correspondent aux champs du formulaire, vous pouvez donc accéder à l'auteur à l'aide de form.cleaned\_data['author']. N'oubliez pas d'ajouter le post actuel au commentaire lorsque vous le créez.

**Remarque :** le cycle de vie de la soumission d'un formulaire peut être un peu compliqué, alors voici un aperçu de son fonctionnement:

1. Lorsqu'un utilisateur visite une page contenant un formulaire, il envoie une demande GET au serveur. Dans ce cas, il n'y a pas de données entrées dans le formulaire, nous voulons donc simplement restituer le formulaire et l'afficher.
2. Lorsqu'un utilisateur entre des informations et clique sur le bouton Soumettre , une demande POST contenant les données soumises avec le formulaire est envoyée au serveur. À ce stade, les données doivent être traitées et deux choses peuvent se produire :
   * Le formulaire est valide et l'utilisateur est redirigé vers la page suivante.
   * Le formulaire n'est pas valide et le formulaire vide s'affiche à nouveau. L'utilisateur est de retour à l'étape 1 et le processus se répète.

Le module de formulaires Django générera des erreurs que vous pourrez afficher à l'utilisateur. Cela dépasse le cadre de ce didacticiel, mais vous pouvez en savoir plus sur le [restitution des messages d'erreur de formulaire](https://docs.djangoproject.com/en/2.1/topics/forms/#rendering-form-error-messages) dans la documentation de Django.

À la **ligne 33**, enregistrez le commentaire et continuez à ajouter le form au dictionnaire context afin que vous puissiez accéder au formulaire dans le template HTML.

La dernière étape avant de créer les templates et de voir ce blog en marche consiste à connecter les URL. Vous devrez créer un autre fichier urls.py à l'intérieur blog/ et ajouter les URL pour les trois vues :

**Python**

from django.urls import path

from . import views

urlpatterns = [

path("", views.blog\_index, name="blog\_index"),

path("<int:pk>/", views.blog\_detail, name="blog\_detail"),

path("<category>/", views.blog\_category, name="blog\_category"),

]

Une fois que les URL spécifiques au blog sont en place, vous devez les ajouter à la configuration de l'URL du projet à l'aide de include() :

**Python**

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path("admin/", admin.site.urls),

path("projects/", include("projects.urls")),

path("blog/", include("blog.urls")),

]

Avec cette configuration, toutes les URL du blog seront préfixées blog/ et vous aurez les chemins d’URL suivants:

* **localhost:8000/blog :** Index du blog
* **localhost:8000/blog/1 :** Vue détaillée du blog avec pk=1
* **localhost:8000/blog/python :** Vue de l'index du blog de tous les posts avec la catégorie python

Ces URL ne fonctionnent pas encore car vous devez encore créer les templates.

Dans cette section, vous avez créé toutes les vues pour votre application de blog. Vous avez appris à utiliser les filtres lors de la création de requêtes et à créer des formulaires Django. Ce ne sera pas long maintenant jusqu'à ce que vous puissiez voir votre application de blog en action !

Comme toujours, n'oubliez pas que vous pouvez consulter le [code source](https://github.com/realpython/materials/tree/8ae8693470e7126a233f5d49685ad65bce3cb537/rp-portfolio) de cette section sur GitHub.

### Application de blog : Templates

La dernière pièce de notre application de blog est les templates. À la fin de cette section, vous aurez créé un blog entièrement fonctionnel.

Vous remarquerez que des éléments bootstrap sont inclus dans les templates pour rendre l'interface plus jolie. Les éléments Bootstrap n’étant pas l’objet de ce tutoriel, j’ai donc passé en revue ce qu’ils font, mais consultez la [documentation Bootstrap](https://realpython.com/get-started-with-django-1/) pour en savoir plus.

Le premier template que vous créez concerne l'index de blog dans un nouveau fichier blog/templates/blog\_index.html. Ce sera très similaire à la vue d'index des projets.

Vous utiliserez une boucle for pour parcourir toutes les posts. Pour chaque post, vous afficherez le titre et un extrait du corps. Comme toujours, vous allez étendre le modèle de base personal\_porfolio/templates/base.html, qui contient notre barre de navigation et un formatage supplémentaire :

**HTML**

1 {% extends "base.html" %}

2 {% block page\_content %}

3 <div class="col-md-8 offset-md-2">

4 <h1>Blog Index</h1>

5 <hr>

6 {% for post in posts %}

7 <h2><a href="{% url 'blog\_detail' post.pk%}">{{ post.title }}</a></h2>

8 <small>

9 {{ post.created\_on.date }} |&nbsp;

10 Categories:&nbsp;

11 {% for category in post.categories.all %}

12 <a href="{% url 'blog\_category' category.name %}">

13 {{ category.name }}

14 </a>&nbsp;

15 {% endfor %}

16 </small>

17 <p>{{ post.body | slice:":400" }}...</p>

18 {% endfor %}

19 </div>

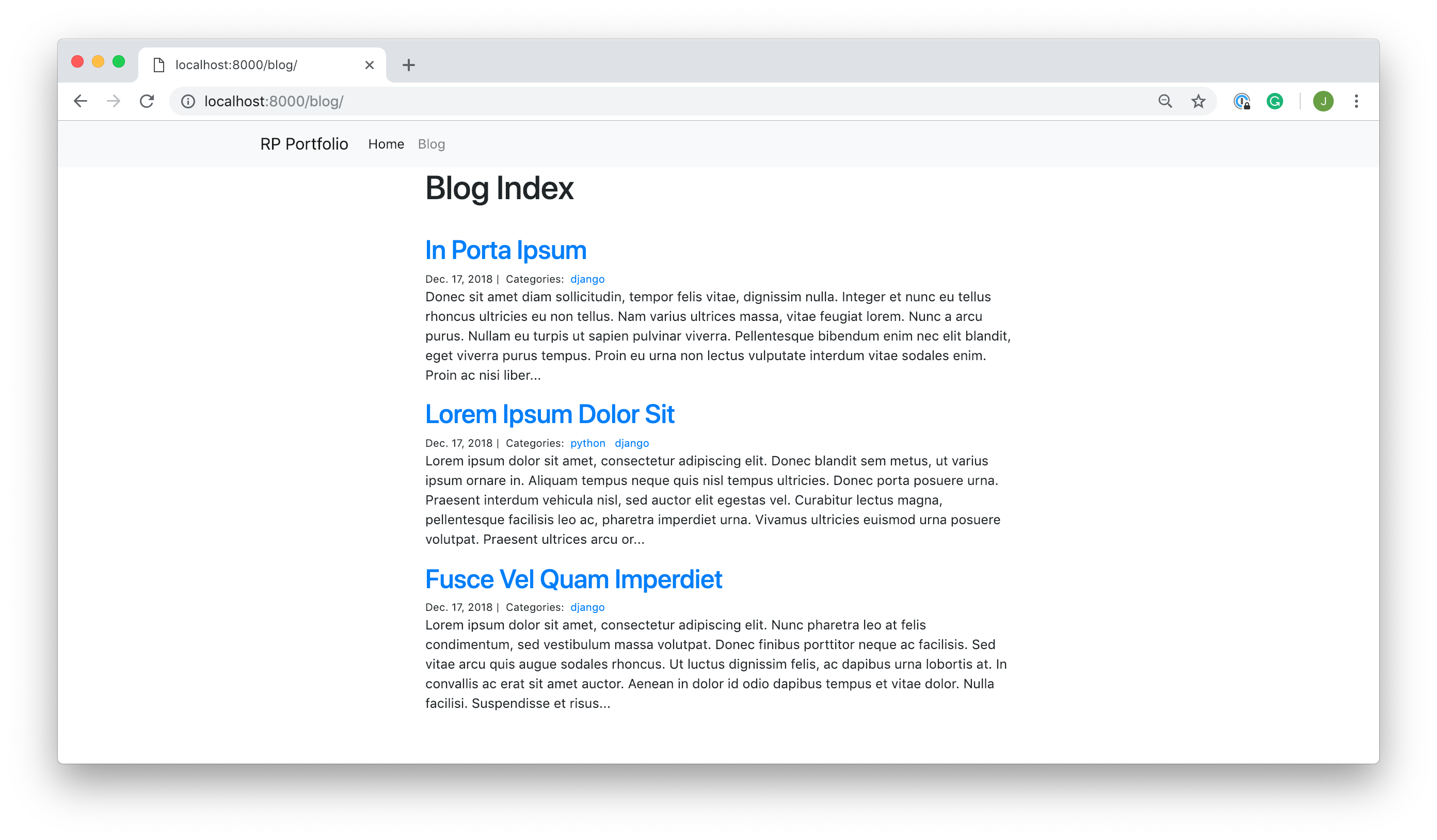
20 {% endblock %}

À la **ligne 7**, nous avons le titre du post, qui est un lien hypertexte. Le lien est un lien Django où nous pointons vers l'URL nommée blog\_detail, qui prend un entier comme argument et doit correspondre à la valeur pk du post.

Sous le titre, nous afficherons l'attribut created\_on du post ainsi que ses catégories. Sur la **ligne 11**, nous utilisons une autre boucle for pour parcourir toutes les catégories attribuées au post.

Sur la **ligne 17**, nous utilisons un **filtre de template** [slice](https://realpython.com/get-started-with-django-1/#slice) pour couper le corps du post de 400 caractères afin que l’index du blog soit plus lisible.

Une fois en place, vous devriez pouvoir accéder à cette page en visitant localhost:8000/blog :

[](https://getbootstrap.com/docs/4.1/layout/overview/)

Ensuite, créez un autre fichier HTML blog/templates/blog\_category.html où votre template blog\_category résidera. Cela devrait être identique à blog\_index.html, sauf que le nom de la catégorie à l'intérieur de la h1balise est remplacé par Blog Index :

1 {% extends "base.html" %}

2 {% block page\_content %}

3 <div class="col-md-8 offset-md-2">

4 <h1>{{ category | title }}</h1>

5 <hr>

6 {% for post in posts %}

7 <h2><a href="{% url 'blog\_detail' post.pk%}">{{ post.title }}</a></h2>

8 <small>

9 {{ post.created\_on.date }} |&nbsp;

10 Categories:&nbsp;

11 {% for category in post.categories.all %}

12 <a href="{% url 'blog\_category' category.name %}">

13 {{ category.name }}

14 </a>&nbsp;

15 {% endfor %}

16 </small>

17 <p>{{ post.body | slice:":400" }}...</p>

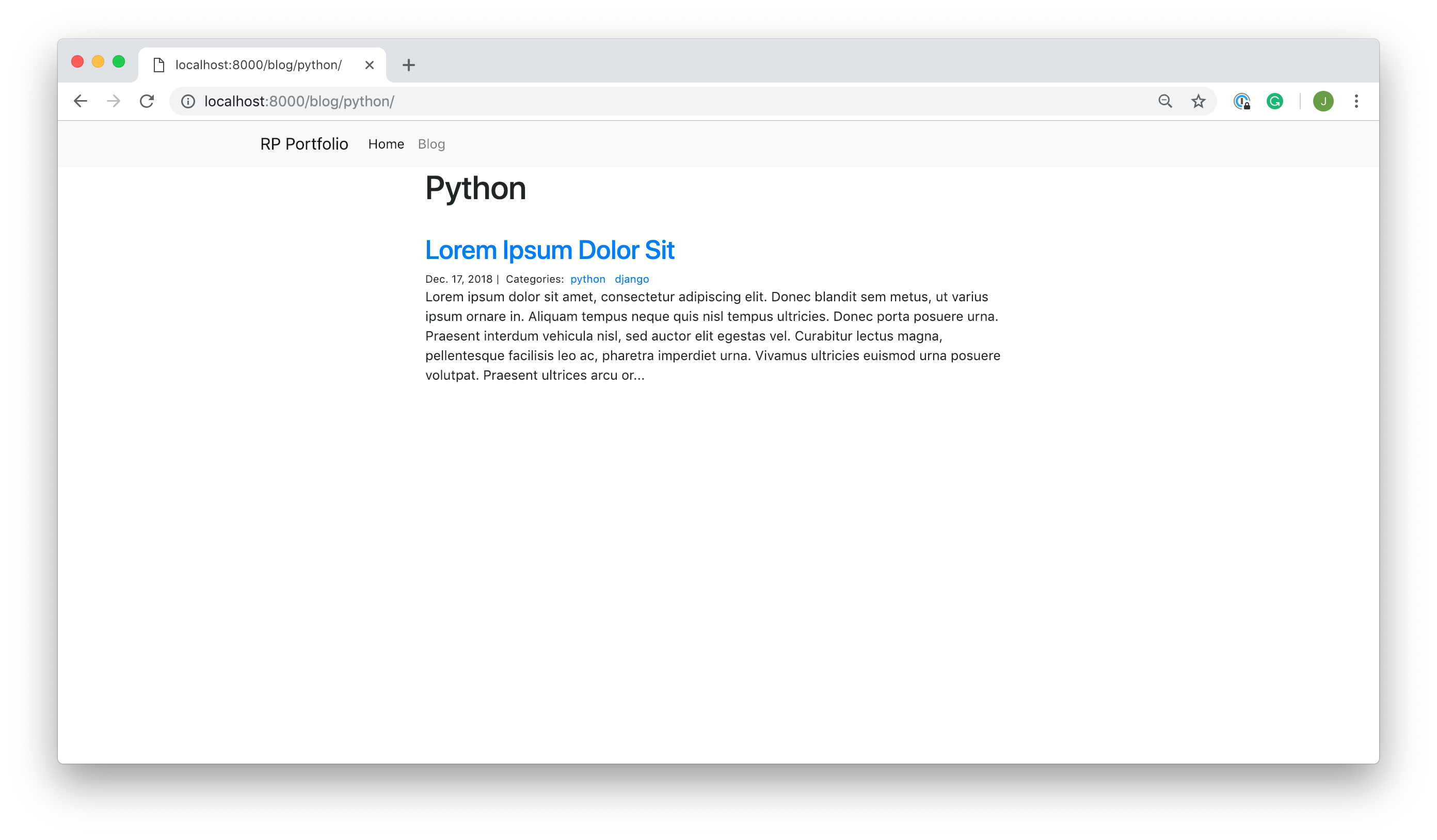
18 {% endfor %}

19 </div>

20 {% endblock %}

La plupart des lignes de ce template sont identiques au template précédent. La seule différence est sur la **ligne 4**, où nous utilisons un autre filtre de template Django [title](https://docs.djangoproject.com/en/2.1/ref/templates/builtins/#title). Cela applique titlecase à la chaîne et fait que les mots commencent par un caractère majuscule.

Une fois ce template terminé, vous pourrez accéder à votre vue Catégorie. Si vous avez défini une catégorie nommée python, vous devriez pouvoir visiter localhost:8000/blog/python et voir tous les posts de cette catégorie :

[](https://github.com/realpython/materials/tree/0091ee5421f8107e8629f1f22687ff224850b889/rp-portfolio)

Le dernier template à créer est le template post\_detail. Dans ce template, vous afficherez le titre et le corps complet d'un post.

Entre le titre et le corps du post, vous affichez la date à laquelle le post a été créé et toutes les catégories. En dessous de cela, vous allez inclure un formulaire de commentaires pour que les utilisateurs puissent ajouter un nouveau commentaire. Sous cela, il y aura une liste de commentaires qui ont déjà été laissés :

**HTML**

1 {% extends "base.html" %}

2 {% block page\_content %}

3 <div class="col-md-8 offset-md-2">

4 <h1>{{ post.title }}</h1>

5 <small>

6 {{ post.created\_on.date }} |&nbsp;

7 Categories:&nbsp;

8 {% for category in post.categories.all %}

9 <a href="{% url 'blog\_category' category.name %}">

10 {{ category.name }}

11 </a>&nbsp;

12 {% endfor %}

13 </small>

14 <p>{{ post.body | linebreaks }}</p>

15 <h3>Leave a comment:</h3>

16 <form action="/blog/{{ post.pk }}/" method="post">

17 {% csrf\_token %}

18 <div class="form-group">

19 {{ form.author }}

20 </div>

21 <div class="form-group">

22 {{ form.body }}

23 </div>

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

25 </form>

26 <h3>Comments:</h3>

27 {% for comment in comments %}

28 <p>

29 On {{comment.created\_on.date }}&nbsp;

30 <b>{{ comment.author }}</b> wrote:

31 </p>

32 <p>{{ comment.body }}</p>

33 <hr>

34 {% endfor %}

35 </div>

36 {% endblock %}

Les premières lignes du template dans lesquelles nous affichons le titre, la date et les catégories du post correspondent à la même logique que pour les templates précédents. Cette fois, lors du rendu du corps du post, utilisez un filtre de template [linebreaks](https://files.realpython.com/media/Screenshot_2018-12-16_at_19.18.42.ee6fb5185d8a.png#linebreaks). Cette balise enregistre les sauts de ligne en tant que nouveaux paragraphes, de sorte que le corps ne s'affiche pas comme un long bloc de texte.

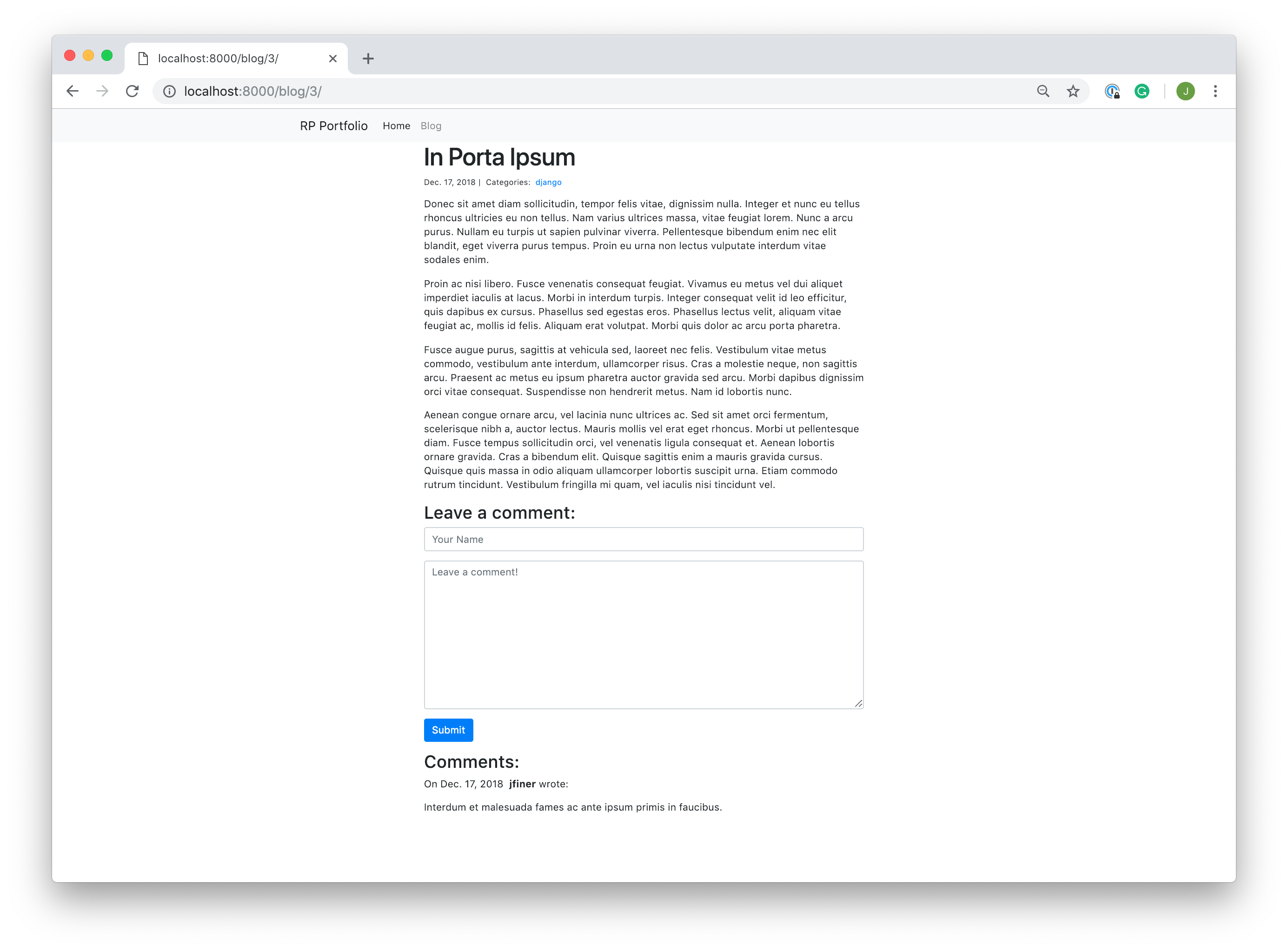
Sous le post, à la **ligne 16**, vous affichez votre formulaire. L'action de formulaire pointe vers le chemin d'accès URL de la page à laquelle vous envoyez la demande POST. Dans ce cas, c'est la même chose que la page en cours de visite. Vous ajoutez ensuite un csrf\_token, qui fournit la sécurité et restitue les champs body et author du formulaire, suivis d'un bouton d'envoi.

Pour obtenir le style bootstrap sur les champs author et body, vous devez ajouter la form-controlclasse aux inputs de texte.

Parce que Django restitue les entrées pour vous lorsque vous incluez {{ form.body }} et {{ form.author }}, vous ne pouvez pas ajouter ces classes dans le modèle. C'est pourquoi vous avez ajouté les attributs aux widgets de formulaire dans la section précédente.

Sous le formulaire, il y a une autre boucle for qui parcourt tous les commentaires du post donné. Les commentaires, les attributs body, author et created\_on, sont tous affichés.

Une fois que ce template est en place, vous devriez pouvoir visiter localhost:8000/blog/1 et afficher votre premier post :

[](https://github.com/realpython/materials/tree/master/rp-portfolio)

Vous devriez également pouvoir accéder aux pages de détail du post en cliquant sur leur titre dans la vue blog\_index.

La dernière touche finale consiste à ajouter un lien vers la barre de navigation de blog \_index dans base.html. Ainsi, lorsque vous cliquez sur Blog dans la barre de navigation, vous pourrez visiter le blog. Consultez [les mises](https://files.realpython.com/media/Screenshot_2018-12-16_at_16.46.36.a71c744f096a.png#L16) à [jour](https://realpython.com/get-started-with-django-1/#L16) de base.html dans le code source pour voir comment ajouter ce lien.

Maintenant que votre site de portfolio personnel est complet, vous avez créé votre premier site Django. La version finale du [code source](https://docs.djangoproject.com/en/2.1/) contenant toutes les fonctionnalités peut être trouvée sur GitHub, alors jetez-y un coup d'œil ! Cliquez un peu sur le site pour voir toutes les fonctionnalités et essayez de laisser des commentaires sur vos posts !

Vous trouverez peut-être ici et là quelques petites choses à améliorer. Allez-y et arrangez-les. La meilleure façon d'en apprendre davantage sur ce framework Web est de le faire par la pratique. Essayez donc d'étendre ce projet et de le rendre encore meilleur ! Si vous ne savez pas par où commencer, je vous ai laissé quelques idées dans la conclusion ci-dessous !

## Conclusion

Félicitations, vous avez atteint la fin du tutoriel ! Nous avons couvert beaucoup, alors assurez-vous de continuer à pratiquer et à construire. Plus vous en construirez, plus cela deviendra facile et moins vous devrez vous reporter à cet article ou à la [documentation](https://docs.djangoproject.com/en/2.1/ref/models/fields/) . Vous construirez des applications Web sophistiquées en un rien de temps.

Dans ce tutoriel, vous avez vu :

* Comment créer des **projets** et des **applications** Django
* Comment ajouter des pages Web avec des **vues** et des **templates**
* Comment obtenir un input utilisateur avec des **formulaires**
* Comment connecter vos vues et vos templates avec des configurations d' **URL**
* Comment ajouter des données à votre site en utilisant des bases de données relationnelles avec **Object Relational Mapper** de Django
* Comment utilizer l’**Admin** **Django** pour gérer vos modèles

En outre, vous avez découvert la structure MVT des applications Web Django et il vous a été expliqué pourquoi Django est un si bon choix pour le développement Web.

Si vous souhaitez en savoir plus sur Django, consultez la [documentation](https://realpython.com/get-started-with-django-1/) et restez à l’écoute pour la deuxième partie de cette série!

**Regarder maintenant** Ce tutoriel contient un cours vidéo connexe créé par l'équipe Real Python. Regardez-le avec le didacticiel écrit pour approfondir votre compréhension : [**Premiers pas avec Django: Construire une application Portfolio**](https://realpython.com/python-for-loop/)

Déploiement

<https://docs.djangoproject.com/fr/2.2/howto/deployment/wsgi/modwsgi/>