
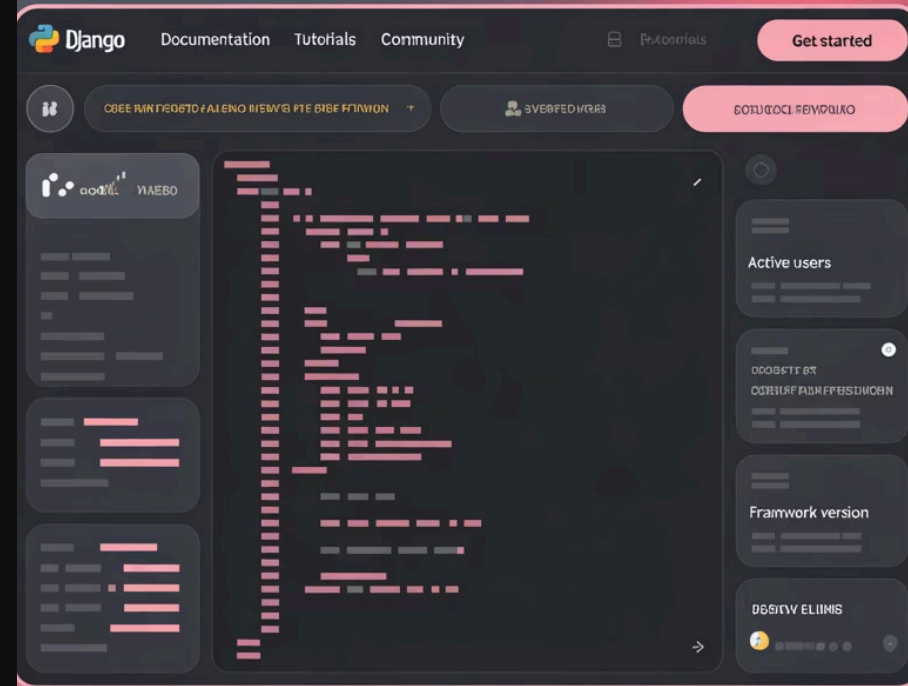


Apprendre Django pas à pas : de l'installation au déploiement

Ce guide complet vous permet d'apprendre Django, le framework web Python qui vous permettra de développer rapidement des applications web robustes et sécurisées.

 by Salsabil Zaghdoudi



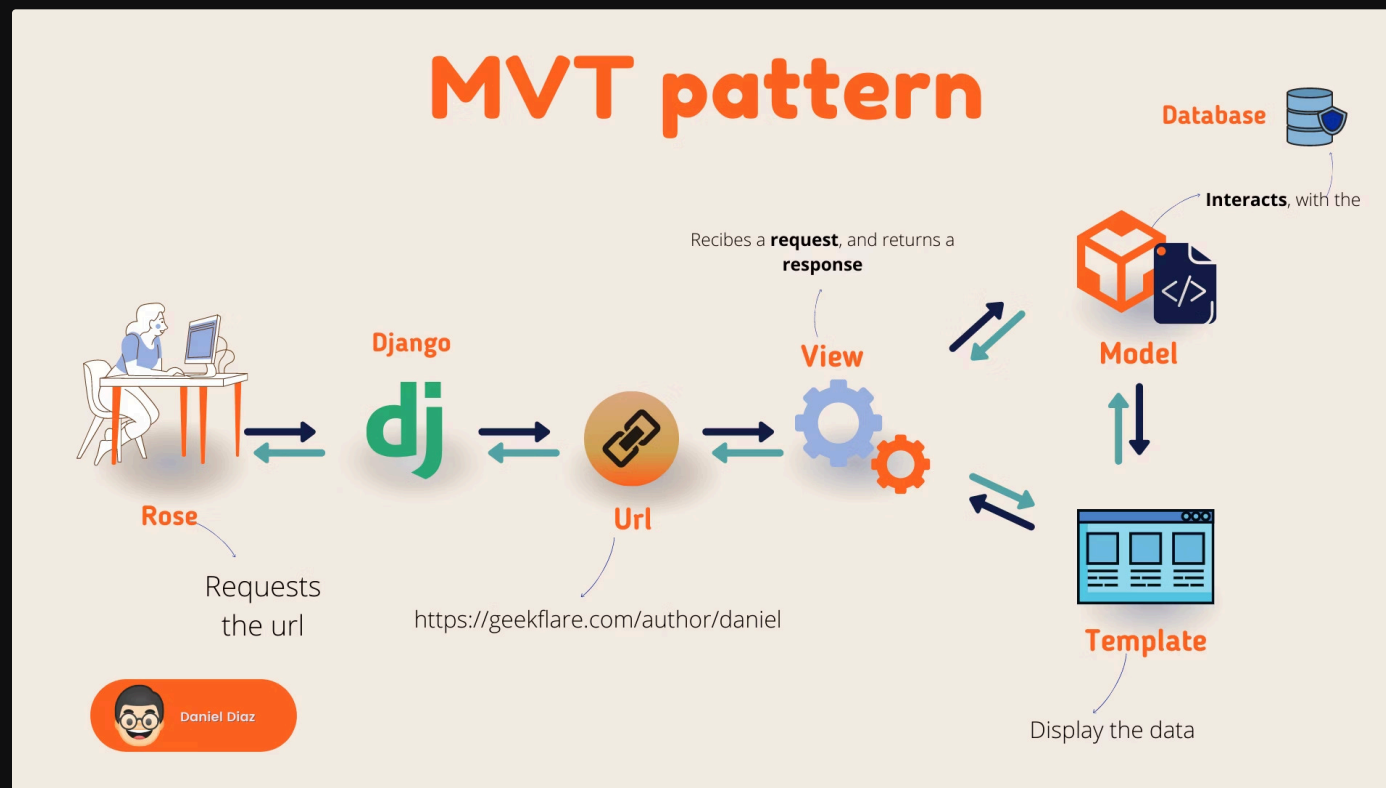
Introduction à Django

Django est un framework web Python de haut niveau qui permet de développer rapidement des sites web sécurisés en Python.

Il fournit de nombreux composants prêts à l'emploi, en pratique, Django prend en charge les tâches courantes du développement web (gestion de base de données, formulaires, authentification, interface d'administration, etc.), ce qui vous permet de vous concentrer sur les fonctionnalités spécifiques de votre application.

Django repose sur une architecture MTV (Modèle – Template – Vue) :

- **Les modèles** gèrent les bases de données.
- Les vues contiennent la logique métier (le code qui décide quoi faire selon la demande de l'utilisateur) elles envoient ensuite une réponse au navigateur.
- **Les templates définissent la présentation HTML des pages affichées dans le navigateur.**
Ils permettent de structurer l'affichage des données (titres, tableaux, formulaires, etc.) sans y intégrer de logique métier.



Good to know - Logique métier

La logique métier correspond aux règles et aux actions que l'application doit effectuer selon les besoins de l'utilisateur. Par exemple, dans un site de réservation, c'est la logique métier qui vérifie les disponibilités, calcule les prix ou enregistre une réservation. Elle se trouve dans les vues (views) et fait le lien entre les données et l'affichage.

Installation de Django

Avant de commencer à coder, il faut installer Django sur votre machine de développement. Il est recommandé de le faire dans un environnement virtuel Python, afin d'isoler les dépendances du projet. Sous Linux/Mac, vous pouvez créer un environnement virtuel et l'activer avec :

```
# Installer le module venv s'il n'est pas disponible
```

```
$ python3 -m pip install --user virtualenv
```

```
# Créer un environnement virtuel nommé "env"
```

```
$ python3 -m venv env
```

```
# Activer l'environnement virtuel
```

```
$ source env/bin/activate
```

Installer Django : Une fois l'environnement virtuel activé, utilisez pip pour installer la dernière version de Django :

```
(env)$ pip install django
```

Vous pouvez vérifier que l'installation a réussi en affichant la version de Django :

```
(env)$ django-admin --version
```

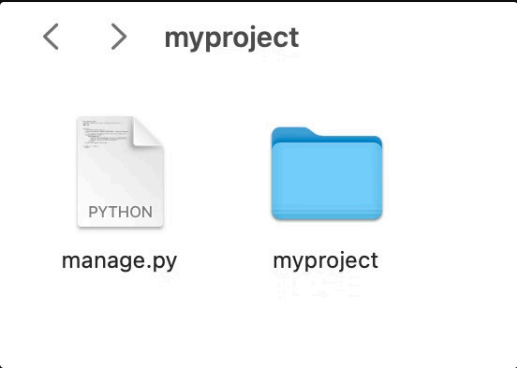
Création d'un projet Django

Un projet Django correspond à un site web Django complet, incluant sa configuration.

Pour créer un nouveau projet, placez-vous dans le répertoire de votre choix et exécutez la commande :

```
$ django-admin startproject myproject
```

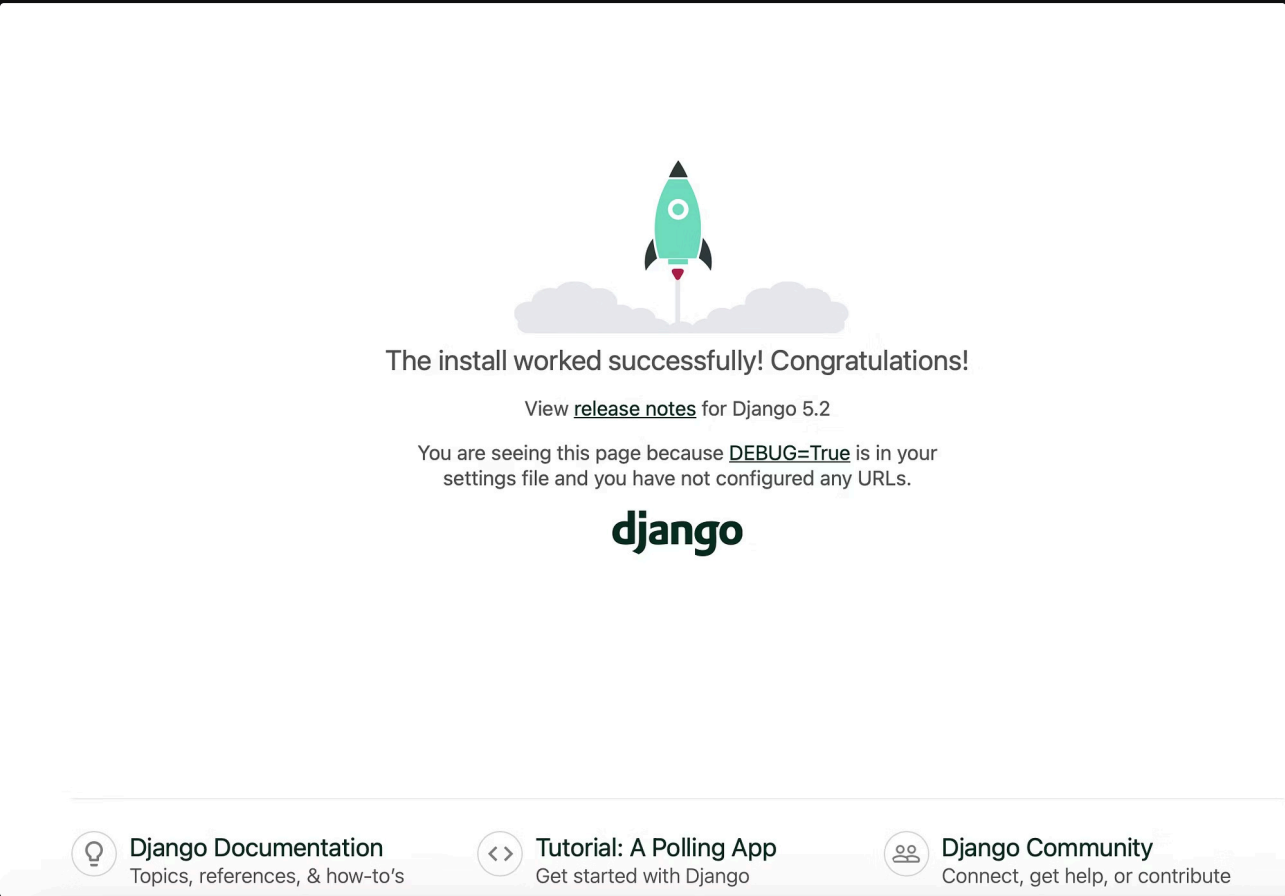
Cette commande crée un répertoire **myproject** contenant la structure de base du projet Django. Vous devriez y voir notamment un fichier **manage.py** et un sous-répertoire également nommé **myproject/** contenant **plusieurs fichiers Python** (nous détaillons leur rôle ci-dessous).



Ensuite, lancez le serveur de développement intégré pour vérifier que tout fonctionne :

```
$ cd myproject
$ python manage.py runserver
```

Par défaut, le serveur écoute sur l'adresse locale 127.0.0.1:8000. Ouvrez un navigateur à l'URL <http://127.0.0.1:8000/>



Structure d'un projet Django

`manage.py`

utilitaire en ligne de commande pour interagir avec le projet (lancer le serveur, exécuter les migrations, créer des utilisateurs, etc.)

`myproject/` (répertoire du même nom que le projet)

package Python du projet contenant les fichiers de configuration.



`myproject/settings.py`

paramètres de configuration du projet Django (base de données, fuseau horaire, applications installées, etc.)

`myproject/urls.py`

définition des routes URL globales du projet.

C'est ici qu'on inclut les URLs des applications et qu'on associe des vues aux chemins de requête.

`myproject/wsgi.py`

point d'entrée standard utilisé pour le **déploiement de l'application Django** sur un serveur web.

Il repose sur la norme **WSGI** (*Web Server Gateway Interface*), qui définit comment un **serveur web** peut **communiquer avec une application Python**, notamment Django.

Ce fichier est essentiel en **production**, mais n'est généralement **pas modifié** pendant le développement.

`myproject/asgi.py`

point d'entrée ASGI (*Asynchronous Server Gateway Interface*), utilisé pour les **déploiements asynchrones** ou en **temps réel**.

*Le mode **asynchrone** signifie que l'application peut **gérer plusieurs requêtes en même temps**, sans bloquer l'exécution — contrairement au mode classique (synchrone) où chaque requête est traitée une par une.*

`myproject/init.py`

fichier vide indiquant que myproject est un package Python.

En résumé, `manage.py` est votre outil principal pendant le développement, et le dossier `myproject/` contient la configuration globale.

Création d'une application Django

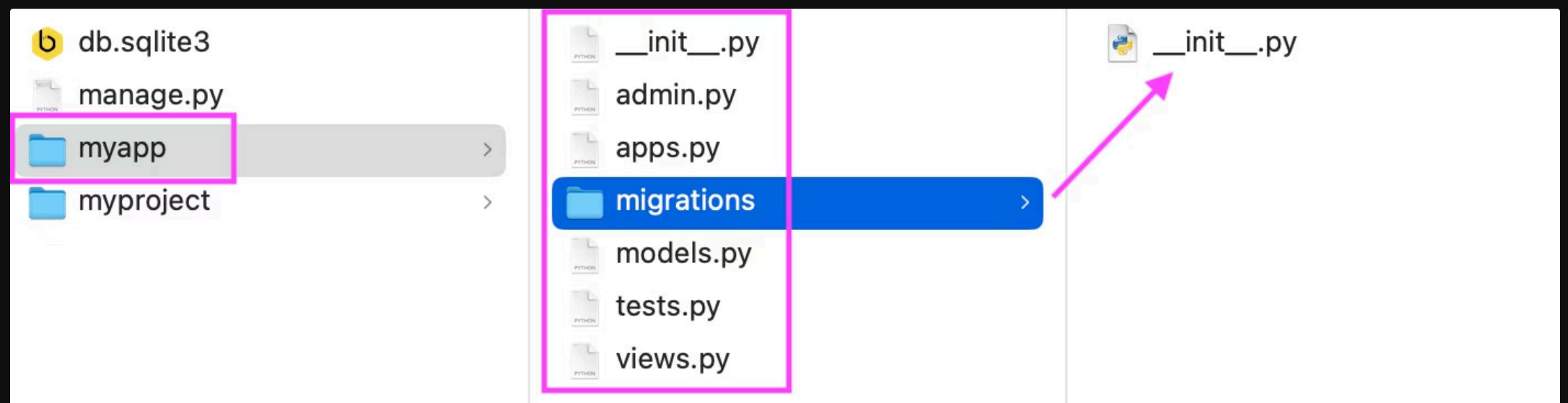
Une **application Django** est un composant modulaire du projet, correspondant à une fonctionnalité ou un module spécifique (par ex. un blog, un sondage, un forum).

Un **projet** peut contenir plusieurs applications, et une application Django peut être réutilisée dans différents projets.

Créons une première application au sein de notre projet. Dans le répertoire racine du projet (où se trouve `manage.py`), tapez :

```
$ python manage.py startapp myapp
```

Cette commande crée un dossier **myapp/** avec la structure suivante :



Django a généré les fichiers de base de l'application :



apps.py contient la configuration de l'application



models.py Sert à définir les **modèles de données**, c'est-à-dire les **structures** qui représentent les tables dans la base de données (BDD).
Exemple : un modèle **Book** va représenter une table **books** en base.



views.py Contient la logique de traitement des requêtes.

Une vue = une **fonction** ou une **classe** qui dit quoi faire quand un utilisateur demande une page (ex. afficher des données, traiter un formulaire...).



admin.py pour l'enregistrement des modèles dans l'admin.



migrations/ Dossier qui contient les **scripts de migration**, générés automatiquement par Django quand on modifie un modèle. Ces fichiers permettent de mettre à jour la structure de la base de données.



tests.py pour écrire d'éventuels tests unitaires.

Déclarer l'application dans le projet :

Avant d'aller plus loin, il faut indiquer à Django que cette application fait partie du projet.

Pour cela, ouvrez **myproject/settings.py** et ajoutez l'application dans la liste `INSTALLED_APPS` :

```
INSTALLED_APPS = [  
    "myapp.apps.MyappConfig", # Ajout de notre application  
    "django.contrib.admin",  
    "django.contrib.auth",  
    ...  
]
```

Django activera ainsi notre application et prendra en compte ses éléments (modèles, templates, etc.).

Modèles et base de données

Une des forces de Django est son ORM (Object-Relational Mapping) qui permet de définir et manipuler la base de données via des classes Python appelées Modèles.

Un modèle correspond à une table dans la base de données, et chaque attribut de la classe correspond à un champ de la table.

Django génère automatiquement les requêtes SQL nécessaires, vous évitant d'écrire du SQL à la main.

Par défaut, le projet est configuré pour utiliser une base de données SQLite (un simple fichier **.sqlite3**) qui convient bien pour le développement. Vous pouvez vérifier la configuration par défaut dans settings.py :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

Comme on le voit, le moteur est SQLite et le fichier de base de données sera créé à la racine du projet (db.sqlite3). Aucune installation supplémentaire n'est requise pour SQLite. Pour un projet en production, il est recommandé d'utiliser un SGBD plus robuste comme PostgreSQL ou MySQL, mais SQLite suffira ici.

Définir un modèle :

Éditez le fichier **myapp/models.py** de votre application pour y ajouter des classes de modèle. Par exemple, supposons que nous créions un modèle pour des livres :

```
# myapp/models.py

from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200) # Titre du livre
    author = models.CharField(max_length=100) # Nom de l'auteur
    published_date = models.DateField() # Date de publication
    isbn = models.CharField(max_length=13, unique=True) # Code ISBN, unique

    def __str__(self):
        return self.title
```

Dans ce modèle **Book** :

Élément	Type de champ	Description
title	CharField	Champ texte
author	CharField	Champ texte
published_date	DateField	Champ de type date
isbn	CharField	Champ texte limité à 13 caractères, avec contrainte d'unicité
__str__ (méthode)	Méthode spéciale	Définit la représentation textuelle de l'objet (affiche le titre du livre)

Good to know :

Django propose de nombreux types de champs (IntegerField, TextField, EmailField, ForeignKey pour les relations, etc.), ce qui permet de modéliser facilement vos données.

Migrer le modèle en base de données :

Une fois le modèle défini, il faut créer la table correspondante en base. Django gère cela via les migrations.

Exécutez :

```
python manage.py makemigrations
```

Django détecte les changements dans models.py et crée un fichier de migration (dans myapp/migrations/) décrivant la création de la table Book. **Vous verrez une sortie :**

```
db.sqlite3 manage.py myapp myproject
● (env) salsabilzaghdoudi@Salsabils-MacBook-Pro myproject % python manage.py makemigrations
Migrations for 'myapp':
  myapp/migrations/0001_initial.py
    + Create model Book
○ (env) salsabilzaghdoudi@Salsabils-MacBook-Pro myproject % █
```

Ensuite, appliquez la migration à la base de données :

```
$ python manage.py migrate
```

Cette commande exécute les migrations et crée la table Book en base de données.

À ce stade, notre modèle est synchronisé avec la base de données – on peut commencer à manipuler des instances (objets) de Book via Django (ajout, requêtes...).

Remarque : Lors de changements ultérieurs (ajout de champs, nouveaux modèles, etc.), il suffira de refaire **makemigrations** puis **migrate**. Django gère un historique des migrations pour appliquer seulement les nouvelles modifications.

🤔 **Et si on fait une erreur dans models.py et on valide la migration ?** 🤔

Pour revenir en arrière, il faut repérer le fichier `xxxx_nomdelamigration.py` dans `myapp/migrations/`. Par exemple, on fait une erreur pendant la **3^e migration**, donc il faut repérer le fichier `0003_nomdelamigration.py` :

```
python manage.py migrate myapp 0002
```

Vues (Views) et logiques métier

Les vues Django gèrent la logique de traitement des requêtes et produisent les réponses (HTML, redirection, JSON, etc.).

Une vue est typiquement une fonction Python (ou une classe) prenant en paramètre un objet **HttpRequest** et renvoyant un objet **HttpResponse**.

Le résultat de la vue, c'est ce que voit l'utilisateur dans la page web.

Ouvrez **myapp/views.py** et créez votre première vue basique :

```
from django.shortcuts import render

from django.http import HttpResponse

def index(request):
    return HttpResponse("Bonjour, bienvenue sur mon site Django !")

def detail(request, book_id):
    return HttpResponse(f"Détail du livre n°{book_id}")
```

Cette vue très simple retourne directement du texte brut dans la réponse HTTP.

Pour la rendre accessible via une URL, nous devons maintenant la lier à une route (URLconf).

Dans un projet Django, le mapping URL -> vue se fait par le biais d'un fichier URLs.

Routage (URLs) de l'application

Dans Django, **chaque application** peut avoir son propre fichier de routes `urls.py`. Ce fichier permet de **lier une URL** à une **fonction (vue)**. Cela permet de mieux organiser le code, surtout si le projet contient plusieurs applications.

Important ❌

Il y a **deux niveaux** de fichiers `urls.py` dans un projet Django :

Niveau	Fichier	Rôle principal	Détail
Projet	<code>myproject/urls.py</code>	Point d'entrée des URLs du site	Il redirige vers les fichiers <code>urls.py</code> des applications via <code>include()</code> .
Application	<code>myapp/urls.py</code>	Gestion des routes propres à l'app	Il associe des URLs à des vues précises, ex. <code>/myapp/5/</code> appelle <code>detail(request, book_id=5)</code> .

Nous créons un nouveau fichier `urls.py` dans le dossier `myapp`, et nous ajoutons le code ci-dessous :

```
from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name="index"), # ex: /myapp/
    path("<int:book_id>/", views.detail, name="detail"), # ex: /myapp/5/
]
```

Dans cet exemple, nous avons deux routes :

- La racine de l'application ("") est associée à la vue `index`.
- Une URL contenant un **entier** (`<int:book_id>/`) est reliée à la vue `detail`.
Par exemple, lorsque l'utilisateur accède à l'adresse `/myapp/5/`, Django va :
 - extraire la valeur `5` depuis l'URL,
 - l'assigner à la variable `book_id`,
 - puis **appeler la fonction** `detail` comme ceci : `detail(request, book_id=5)`.

Après avoir défini les `urlpatterns` de l'application, il faut les inclure dans le fichier **URLs** du projet pour les activer.

Ouvrez `myproject/urls.py` (créé par `startproject`) et modifiez-le ainsi :

```
# myproject/urls.py

from django.contrib import admin
from django.urls import include, path

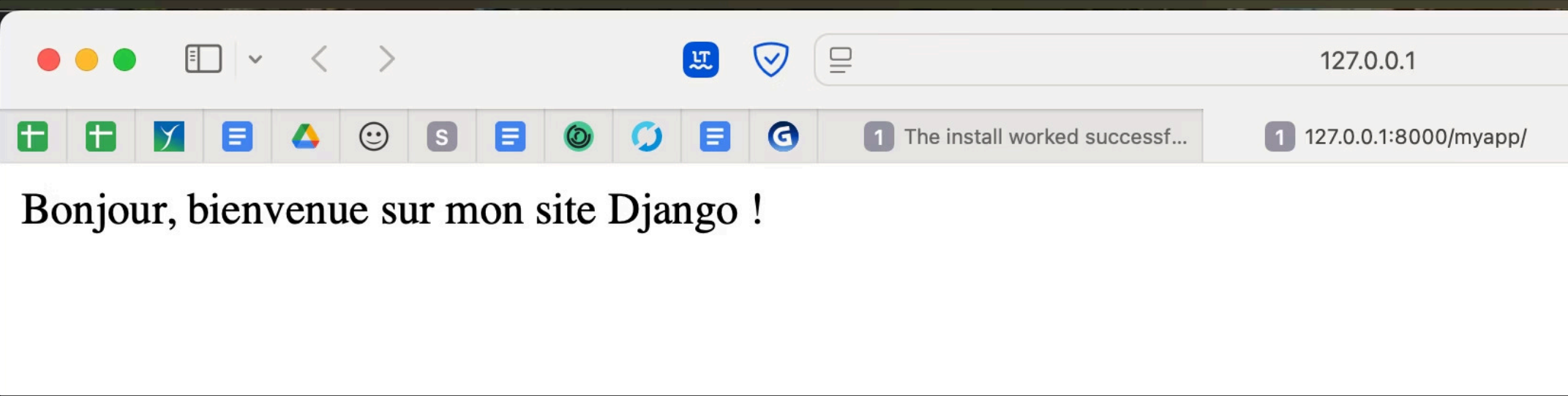
urlpatterns = [
    path('myapp/', include('myapp.urls')), # Inclure les URL de l'application
    path('admin/', admin.site.urls), # URL de l'interface d'administration
]
```

La fonction `include()` permet d'importer les routes de l'application `myapp` dans le projet principal. Grâce à cela, **toutes les URL qui commencent par `/myapp/`** seront automatiquement **redirigées vers les routes définies dans `myapp/urls.py`**.

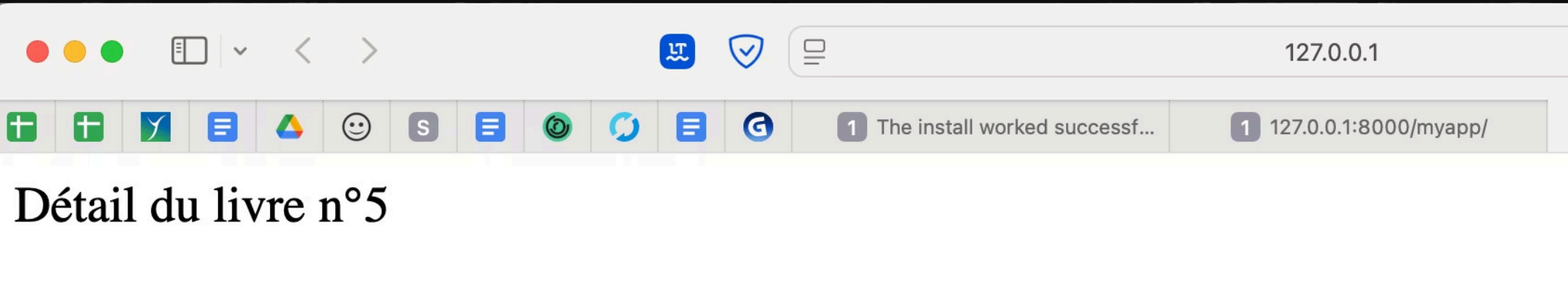
Désormais, si vous relancez le serveur :

```
python manage.py runserver
```

<http://127.0.0.1:8000/myapp/>, Django appellera la vue `index` de `myapp` et affichera la réponse "Bonjour, bienvenue..." définie plus haut:



Si vous allez sur <http://127.0.0.1:8000/myapp/5/> :



Nommage des URLs :

Notez que nous avons donné un nom à nos routes (`name="index"` et `"detail"`) dans `myapp/urls.py`. Ces attributs permettent de faire référence à ces routes dans les templates, sans taper l'URL en dur :

```
<a href="{% url 'index' %}">Accueil</a>
<a href="{% url 'detail' book_id=5 %}">Voir le livre</a>
```

Cela évitera de **casser les liens** si un jour, on change l'URL `/myapp/` en `/livres/` : il suffira de modifier le chemin **dans `urls.py`**, et non dans tous les fichiers HTML.

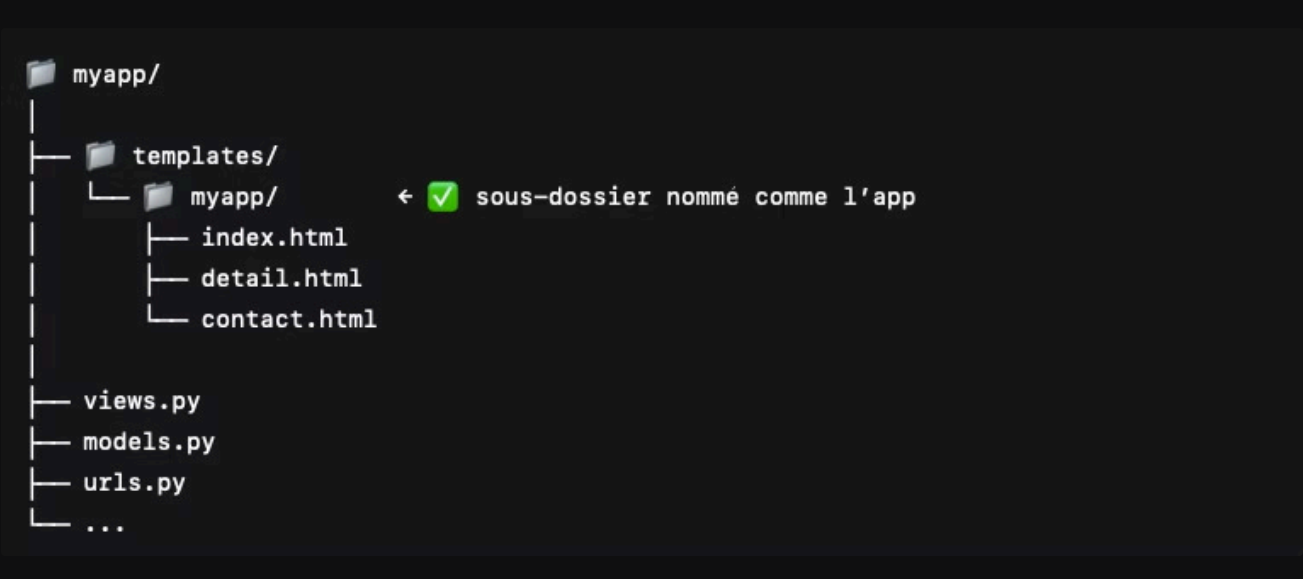
Templates (gabarits HTML)

Jusqu'ici, nos vues renvoient des réponses simples en utilisant la fonction `HttpResponse`, contenant soit du **texte brut**, soit du **HTML écrit directement dans le code Python**.

Pour des pages complètes, on utilise les templates Django, qui sont des fichiers HTML avec un langage de gabarits permettant d'injecter du contenu dynamique. Django utilise par défaut un système de templates très puissant.

Créer un template :

Par convention, on crée un répertoire templates dans chaque application Django, et on y place les fichiers HTML.



Pour éviter les conflits de noms entre applications, il est recommandé de créer un sous-dossier nommé comme l'app.

Par exemple, pour myapp, on crée : myapp/templates/myapp/, et on y place les fichiers HTML.

Pourquoi ?

- Django va chercher les templates dans tous les dossiers `templates/` déclarés.
- En créant un sous-dossier **myapp/ du même nom que l'application**, on évite que deux fichiers `index.html` (de deux apps différentes) se chevauchent.

Ainsi, notre page d'accueil de l'application pourrait être myapp/templates/myapp/index.html. Ce rangement garantit que Django trouvera le bon fichier même si d'autres apps ont des templates nommés "index.html".

Exemple de template :

Créez le fichier **myapp/templates/myapp/index.html** avec le contenu suivant :

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Liste des livres</title>
</head>
<body>
  <h1>Liste des livres</h1>

  {% if latest_books %}
    <ul>
      {% for book in latest_books %}
        <li>{{ book.title }} – {{ book.author }}</li>
      {% endfor %}
    </ul>
  {% else %}
    <p>Aucun livre disponible.</p>
  {% endif %}
</body>
</html>
```

Ce template affiche un titre suivi d'une **liste de livres, à condition que la variable latest_books (envoyée par la vue) contienne des données**.

- Les balises `{{ ... }}` permettent **d'afficher dynamiquement** les informations de chaque livre (par exemple `{{ book.title }}`).
- Les balises `{% for %} ... {% endfor %}` servent à **parcourir chaque livre** de la liste.
- Si la liste est vide, la balise `{% if %} ... {% else %}` permet **d'afficher un message alternatif**, comme : "Aucun livre disponible".

Ce système de template permet de générer des pages HTML dynamiques **sans écrire de logique Python dans le fichier HTML**, ce qui garantit une bonne séparation entre :

- Le fichier **views.py** (la vue) : contient la **logique** : récupérer les livres dans la base de données, les trier, etc.
- Le fichier **index.html** (le template) : contient uniquement **l'affichage visuel** : titres, listes, paragraphes HTML avec des balises comme `{{ book.title }}`.

Résultat : On sépare bien ce que le **backend** fait (calculer, chercher dans la base...) de ce que le **frontend** montre à l'utilisateur.

Utiliser le template dans la vue :

Au départ, notre vue `index` renvoyait simplement du **texte brut** grâce à `HttpResponse`, comme ceci :

```
def index(request):
    return HttpResponse("Bonjour !")
```

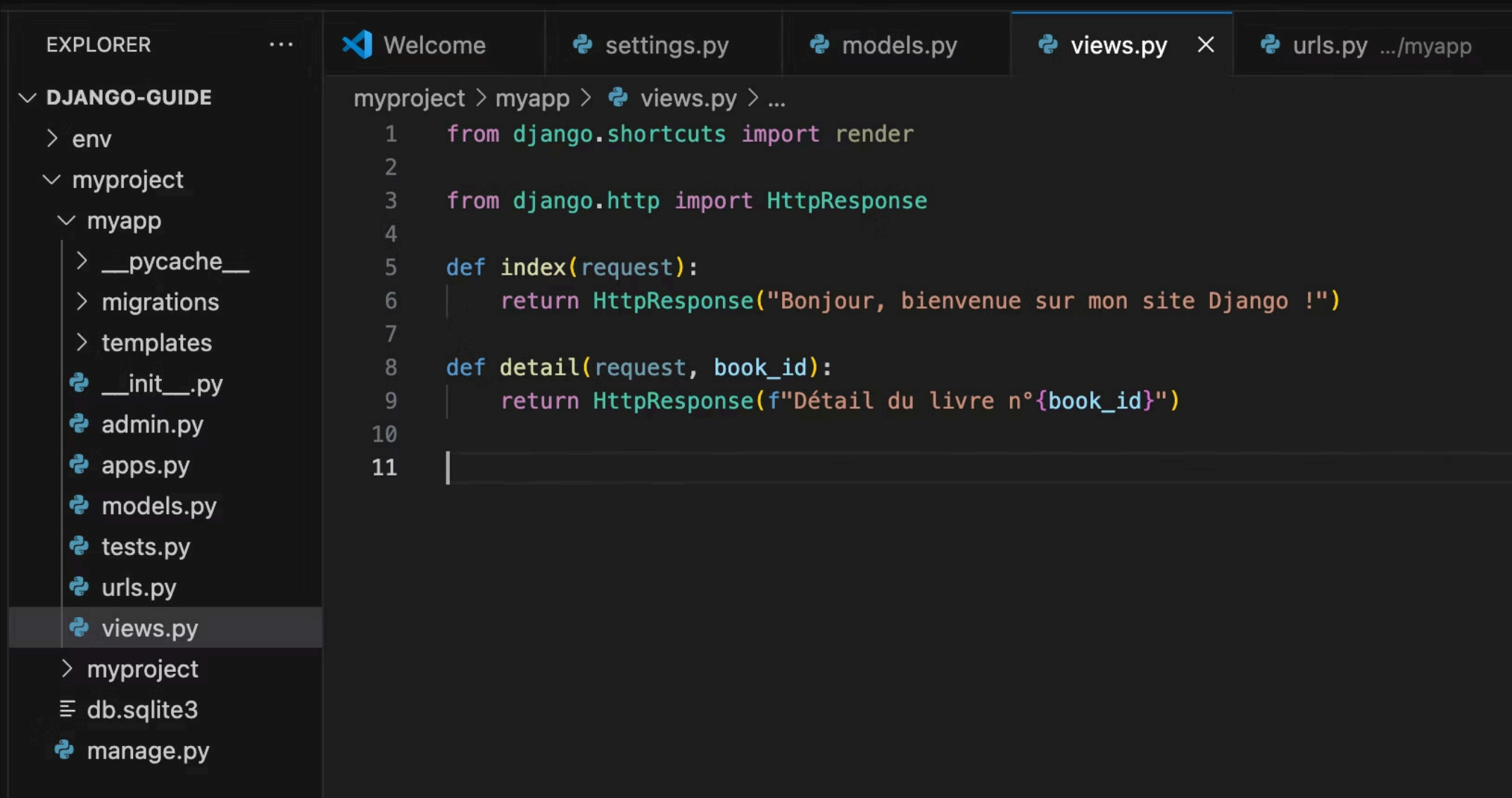
Ce fonctionnement est utile pour tester rapidement, mais **ne permet pas d'afficher une vraie page web** avec du contenu structuré.

Nouvelle version : afficher un template HTML avec des données

Nous allons maintenant **modifier la vue index** pour qu'elle :

- Récupère les données** (ex : les 5 livres les plus récents)
- Prépare un contexte** contenant ces données
- Affiche une page HTML** (`index.html`) dans laquelle ces données seront insérées automatiquement.

La vue actuelle dans **myapp/views.py** :



Maintenant on va **remplacer la vue index** pour qu'elle affiche la page HTML dynamique avec la liste des livres, tout en **gardant detail()** comme elle est pour le moment.

```
#myapp/views.py

from django.shortcuts import render
from django.http import HttpResponse
from .models import Book
def index(request):
    # Récupérer les 5 derniers livres par date de publication
    latest_books = Book.objects.order_by('-published_date')[:5]
    # Créer un dictionnaire de contexte à transmettre au template
    context = {'latest_books': latest_books}
    # Rendre la page HTML en injectant le contexte
    return render(request, 'myapp/index.html', context)

def detail(request, book_id):
    return HttpResponse(f"Détail du livre n°{book_id}")
```

Dans ce fichier **views.py**, nous avons deux vues.

La fonction **index** interagit avec le modèle **Book** pour récupérer les 5 derniers livres publiés, en les triant par date de publication décroissante (`order_by('-published_date')[:5]`).

Ces livres sont ensuite stockés dans un dictionnaire appelé **context**, qui sera transmis au fichier HTML `index.html` via la fonction **render**.

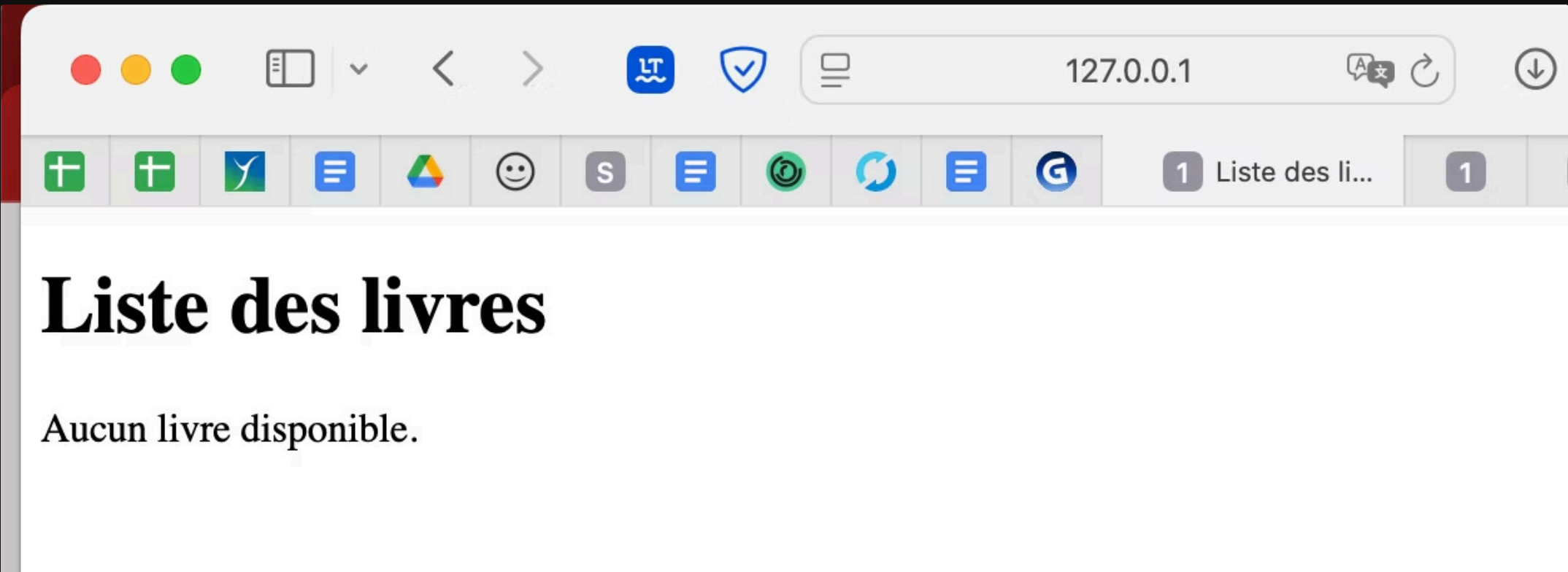
render est une fonction fournie par Django qui **permet de générer une page HTML à partir d'un template** (fichier `.html`) et d'y **injecter des données** dynamiques.

Cette méthode permet d'afficher une page web dynamique contenant les données, sans écrire de HTML dans le code Python.

La seconde vue, **detail**, est plus simple : elle renvoie une réponse texte avec l'identifiant du livre demandé, en attendant d'être améliorée pour afficher les détails du livre à partir de la base de données.

Tests :

Dans le navigateur : <http://127.0.0.1:8000/myapp/>



Focus sur les fichiers statiques

Dans une application web, les fichiers **statiques** sont les fichiers qui ne changent pas selon les données de la base. Ce sont par exemple :

- les **fichiers CSS** pour le style et la mise en page,
- les **images** (logos, icônes...),
- les **fichiers JavaScript** pour ajouter des animations ou interactions à la page.

Objectif du test

On va créer une **feuille de style CSS** qui change la couleur du titre et on va **l'afficher dans ta page Django**.

1. Créez cette arborescence :

```
myapp/  
|  
├── static/  
|   ├── myapp/  
|   └── style.css
```

2. Dans le fichier `style.css` :

```
h1 {  
    color: red;  
    text-align: center;  
}
```

3. le nouveau `index.html` dans `myapp/templates/myapp/index.html`:

```
{% load static %}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="UTF-8">  
    <title>Liste des livres</title>  
    <link rel="stylesheet" href="{% static 'myapp/style.css' %}">  
</head>  
<body>  
    <h1>Liste des livres</h1>  
  
    {% if latest_books %}  
        <ul>  
            {% for book in latest_books %}  
                <li>{{ book.title }} – {{ book.author }}</li>  
            {% endfor %}  
        </ul>  
    {% else %}  
        <p>Aucun livre disponible.</p>  
    {% endif %}  
</body>  
</html>
```

4. Redémarrez le serveur : `python manage.py runserver`

Formulaires basés sur un modèle (ModelForm)

Un **ModelForm** est une classe Django qui permet de générer automatiquement un formulaire HTML à partir d’un modèle de base de données. Il facilite la création, la modification et la validation d’objets sans avoir à redéfinir manuellement tous les champs du formulaire.

Grâce au ModelForm, on gagne du temps et on réduit les risques d’erreur, car les champs du formulaire sont directement liés aux attributs du modèle (comme `title`, `author`, etc.). Django s’occupe également de la validation et de l’enregistrement des données en base avec une simple méthode `form.save()`.

1. Créer un ModelForm basé sur le modèle Book Dans myapp/forms.py :

```
myapp/forms.py
from django import forms
from .models import Book

class BookForm(forms.ModelForm):
    class Meta:
        model = Book
        fields = ['title', 'author', 'published_date', 'isbn']
```

2.Créer la vue add_book Dans myapp/views.py :

```
# myapp/views.py

from django.shortcuts import render, redirect
from django.http import HttpResponse
from .models import Book
from .forms import BookForm

# Vue d'accueil : affiche les 5 derniers livres
def index(request):
    latest_books = Book.objects.order_by('-published_date')[:5]
    context = {'latest_books': latest_books}
    return render(request, 'myapp/index.html', context)

# Vue de détail (non encore utilisée pleinement)
def detail(request, book_id):
    return HttpResponse(f"Détail du livre n°{book_id}")

# Vue pour ajouter un livre via un formulaire ModelForm
def add_book(request):
    if request.method == 'POST':
        form = BookForm(request.POST)
        if form.is_valid():
            form.save() # Enregistre le livre dans la base
            return redirect('index') # Redirige vers la page d'accueil
        else:
            print(form.errors) # Affiche les erreurs dans le terminal
    else:
        form = BookForm() # Affiche un formulaire vide
    return render(request, 'myapp/add_book.html', {'form': form})
```

3. Ajouter une URL pour accéder au formulaire :

```
#myapp/urls.py

from django.urls import path
from . import views

urlpatterns = [
    path("", views.index, name='index'),
    path('add/', views.add_book, name='add_book'),
    path('<int:book_id>/', views.detail, name='detail'),
]
```

4.Créer un nouveau template add_book.html dans myapp/templates/myapp

```
<!DOCTYPE html>
<html>
<head>
    <title>Ajouter un livre</title>
</head>
<body>
    <h1>Ajouter un nouveau livre</h1>

    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Enregistrer</button>
    </form>

    <p><a href="{% url 'index' %}">Retour à la liste des livres</a></p>
</body>
</html>
```

Vérifiez que myproject/urls.py contient bien :

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('myapp/', include('myapp.urls')), # Inclure les URL de l'application
    path('admin/', admin.site.urls), # URL de l'interface d'administration
]
```

Testez l'ajout des livres : <http://localhost:8000/myapp/add/>

L'interface d'administration Django

Django inclut une interface d'administration générique, permettant de gérer facilement les données de vos modèles sans avoir à coder de back-office spécifique.

Une fois activée, elle fournit un site web interne pour créer, modifier, supprimer les objets de votre base, gérer les utilisateurs, les permissions, etc., le tout avec authentification sécurisée.

Par défaut, l'admin est déjà installée dans `INSTALLED_APPS` et accessible à l'URL `/admin/` (nous l'avons laissée dans `myproject/urls.py`).

Créer un super-utilisateur

Pour vous connecter à l'admin, il faut un compte administrateur. Créez-en un avec la commande :

```
$ python manage.py createsuperuser
Username: admin
Email address: admin@example.com
Password: *****
Password (again): *****
Superuser created successfully.
```

Accéder à l'interface d'administration

Une fois cette étape faite, lancez le serveur (runserver).

Rendez-vous sur <http://127.0.0.1:8000/admin/> dans votre navigateur.

Vous arrivez sur l'écran de connexion de l'admin Django.

Identifiez-vous avec le compte superuser que vous venez de créer.