Débutez avec le framework Django

12 heures Moyenne

Mis à jour le 28/06/2023

Séparez la logique de l'application de la présentation avec un gabarit Django

01:26

Comprenez les gabarits

Nous allons commencer ce chapitre par la démonstration d'un anti-pattern.

En programmation, un **design pattern** (patron de conception) est un style ou une technique qui représente la meilleure pratique et conduit à de bons résultats. À l'inverse, un **anti-pattern** représente

une mauvaise pratique, quelque chose que nous devons éviter.

Pour les besoins de la démonstration, remplissons une de nos pages avec un peu plus de code HTML.

Ouvrez listings/views.py et trouvez notre vue `hello` . Nous voulons commencer à ajouter les éléments structurels d'une page web, comme `<head>` , `<body>` , `<title>` et la balise `<html>` .

Nous pourrions y parvenir en ajoutant plus de HTML à notre vue, comme ceci :

```html
def hello(request):
    bands = Band.objects.all()
    return HttpResponse(f"""
        <html>
            <head><title>Merchex</title><head>
            <body>
                <h1>Hello Django !</h1>
                <p>Mes groupes préférés sont :<p>
                <ul>
                    <li>{bands[0].name}</li>
                    <li>{bands[1].name}</li>
                    <li>{bands[2].name}</li>
                </ul>
            </body>
        </html>
    """)
```

Ce code fonctionne... mais c'est un anti-pattern.

> En fait, cette vue utilise un anti-pattern depuis le début, et là, nous aggravons la situation.

> Pourquoi, quel est le problème avec cette approche ?

Notre vue commence déjà à avoir l'air un peu chargée et la quantité de HTML ici ne fera que s'accroître au fur et à mesure de la construction de notre application.

Le problème est que notre vue a maintenant deux responsabilités :

- sélectionner tous les objets `Band` de la base de données : la **logique**;
- afficher les noms de ces groupes parmi d'autres contenus comme les titres et les paragraphes : la **présentation**.

Si une partie de notre application a trop de responsabilités, elle devient ingérable.

> Alors, comment transformer cet anti-pattern en design pattern, et utiliser les meilleures pratiques ?

Nous allons adhérer au principe de la responsabilité unique en déplaçant la responsabilité de la présentation hors de la vue et en la plaçant à sa place légitime : un gabarit.

> Consultez le cours *Écrivez du code Python maintenable* pour un rappel du **principe de responsabilité unique**.

Ajoutez un fichier gabarit HTML dans le répertoire des gabarits

Commencez par créer un nouveau fichier à l'adresse :

listings/templates/listings/hello.html

> Notez la structure de répertoire que nous utilisons ici. Nous mettons toujours un sous-répertoire dans le répertoire des gabarits (templates) qui porte le même nom que l'application (« listings »). Découvrez quelques astuces sur **l'utilisation des gabarits**.

Remplissons ce fichier avec notre HTML :

html

```
# listings/templates/listings/hello.html

<html>
    <head><title>Merchex</title></head>
    <body>
        <h1>Hello Django !</h1>
        <p>Mes groupes préférés sont :</p>
        <!-- TODO : liste des groupes -->
    </body>
</html>
```

Le code HTML n'est-il pas tellement plus beau dans un fichier .html que dans un fichier .py ? Vous bénéficiez d'une coloration syntaxique appropriée et d'une indentation automatique !

Mettez à jour la vue pour générer ce gabarit

Mettons maintenant à jour notre vue, de sorte qu'au lieu de définir son propre HTML, elle génère notre modèle à la place :

python

```
# listings/views.py

...
from django.shortcuts import render
...

def hello(request):
    bands = Band.objects.all()
    return render(request, 'listings/hello.html')
```

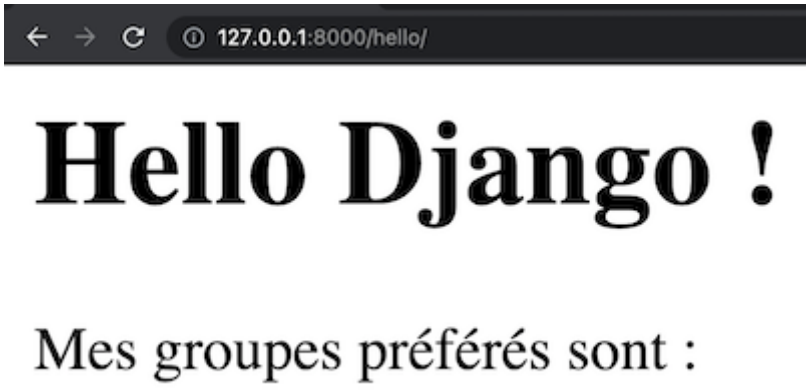
Tout d'abord, nous importons la fonction `render` . Cet élément est probablement déjà présent puisqu'il est inclus dans le code de base, mais ajoutez-le si nécessaire.

Dans la déclaration de retour, nous n'appelons plus le constructeur `HttpResponse` . Au lieu de cela, nous appelons la fonction `render` avec 2 arguments :

- l'objet `request` qui est passé dans la fonction `hello` ;
- une chaîne de caractères contenant le chemin d'accès au fichier gabarit que nous avons créé.

> Sous le capot, la fonction `render` crée un objet HttpResponse avec le HTML de notre modèle et le renvoie. Notre vue renvoie donc toujours une HttpResponse (ce qu'elle doit faire, pour être une vue).

Jetons un coup d'œil à notre page dans le navigateur :



La page « hello ».

C'est un bon début, mais les noms de nos groupes n'apparaissent plus sur la page.

Nous avons besoin d'un moyen d'injecter nos données de modèle dans notre gabarit.

Passez un objet contextuel au gabarit contenant une liste d'objets

Revenons à notre vue et ajoutons un troisième argument à notre appel de la méthode `render` . Cet argument doit être un `dict` Python.

python

```python
# listings/views.py

...
return render(request,
        'bands/hello.html',
        {'first_band': bands[0]})
```

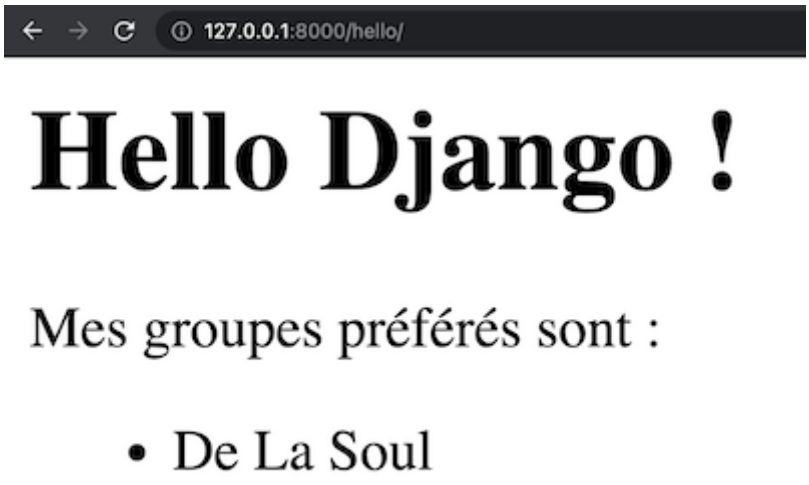
Ce dictionnaire est appelé **dictionnaire contextuel**. Chaque clé du dictionnaire devient une variable que nous pouvons utiliser dans notre modèle, comme ceci :

html

```
# listings/templates/listings/hello.html

...
<p>Mes groupes préférés sont :</p>
    <ul>
        <li>{{ first_band.name }}</li>
    </ul>
...
```

Jetons un coup d'œil à la page dans notre navigateur pour voir cela en action :



La page « hello ».

> C'est quoi ces accolades ? Ce n'est pas du HTML !

Bien vu ! Dans un code HTML valide, les gabarits de Django peuvent inclure cette syntaxe avec des accolades, également connue sous le nom de **langage de gabarits Django**. Chaque fois que vous voyez des doubles accolades contenant un nom de variable, la valeur de cette variable sera insérée. Elles sont appelées **variables de gabarits**.

Nous pourrions continuer à ajouter chaque groupe individuellement au dictionnaire contextuel, mais gagnons du temps et passons-les tous en une seule fois :

python

```
# merchex/listings/views.py

    ...
    return render(request,
    'bands/hello.html',
    {'bands': bands})
```

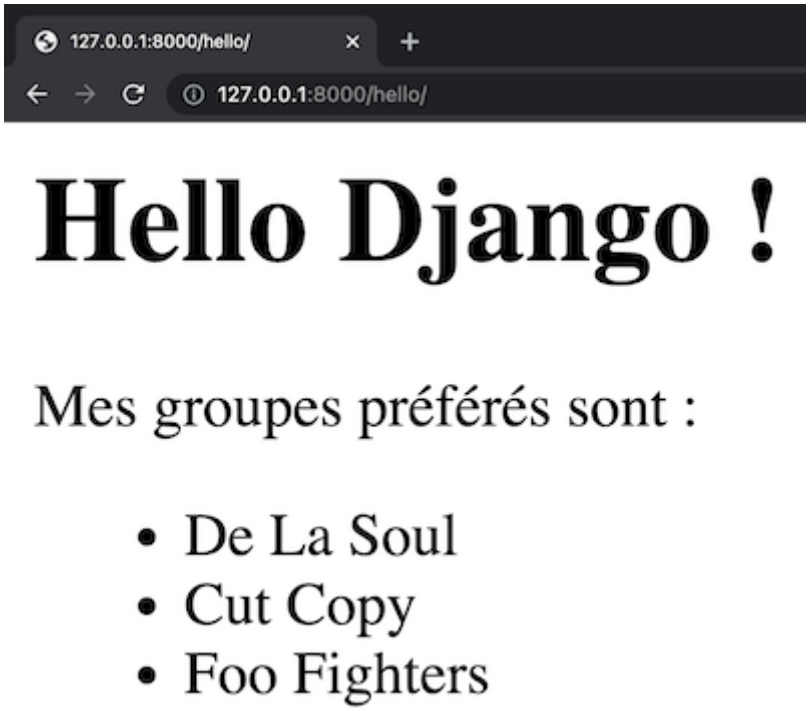
Puis, dans notre modèle :

html

```
# listings/templates/listings/hello.html

...
<p>Mes groupes préférés sont :</p>
    <ul>
        <li>{{ bands.0.name }}</li>
        <li>{{ bands.1.name }}</li>
        <li>{{ bands.2.name }}</li>
    </ul>
...
```

Dans le code du gabarit, pour accéder à un élément d'une liste, on utilise `bands.0` , au lieu de `bands[0]` comme on le fait dans le code Python.



La page « hello ».

En résumé, les gabarits sont un moyen pour définir le contenu d'une page qui ***ne change pas***. À l'intérieur de ces gabarits, nous insérons des variables de gabarits, qui servent d'espaces réservés pour le contenu qui ***change***.

00:32

Lorsque nous générons un gabarit dans une vue, nous passons un dictionnaire de contexte au gabarit et les variables de contexte sont injectées dans leurs espaces respectifs.

En gardant la vue libre de tout code de présentation (HTML), nous pouvons limiter la responsabilité de la vue à une seule chose : la logique pour récupérer les données correctes de la base de données, et les injecter dans la page.

Maintenant que nous avons compris les principes de base des gabarits, examinons certaines de leurs fonctionnalités plus utiles.

Itérez sur une liste dans un modèle

Jusqu'à présent, notre exemple de code est parti du principe que nous aurons toujours un nombre fixe de groupes : dans notre exemple, 3. Mais que se passe-t-il si nous supprimons l'un des groupes de notre base de données ? La référence à `bands[2]` entraînerait maintenant une erreur, car il n'y a plus d'élément à l'index `2` de notre liste. D'autre part, si nous avons plus de 3 groupes, nous pourrions également vouloir afficher ces groupes supplémentaires sur notre page.

En programmation, lorsque nous devons traiter une liste de longueur inconnue, nous utilisons une boucle `for` , qui itère sur chaque élément et s'arrête lorsqu'il ne reste plus d'éléments. Le langage de gabarit de Django possède sa propre syntaxe pour les boucles.

Dans notre modèle, nous allons remplacer notre liste de longueur fixe par une boucle :

html

```
# listings/templates/listings/hello.html

<p>
    Mes groupes préférés sont :
    {% for band in bands %}
```

```
    {{ band.name }},
    {% endfor %}
</p>
```

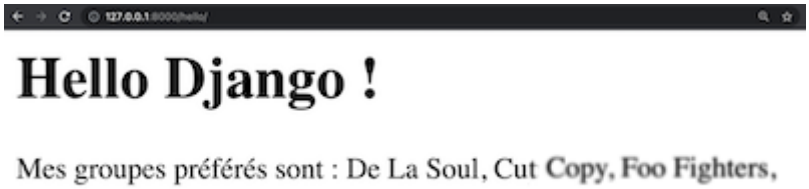
Voici quelques éléments à prendre en compte :

- Les boucles et autres instructions logiques sont entourées de crochets et de signes de pourcentage (`{% ... %}`). Il s'agit de **balises de gabarits**.
- Une instruction `for` est construite en utilisant une syntaxe similaire à celle de Python, mais sans les deux-points de fin (`:`).
- Nous avons tendance à utiliser le format conventionnel de Python `for singulier in pluriel` dans les instructions `for` .
- Une balise de gabarit `for` doit posséder une balise de fermeture `endfor` plus loin dans le gabarit.
- L'espace entre les balises `for` et `endfor` peut contenir du texte, du HTML et même des variables de gabarits Django.

> Attendez… vous avez dit que la logique appartient à la vue et la présentation au modèle, et que nous devrions garder les deux séparés. Mais on vient d'ajouter une boucle `for` à notre gabarit. N'est-ce pas… logique ?

Si, si, vous avez raison ! Les frontières sont un peu floues ici. La philosophie de conception de Django est la suivante : « un système de gabarits est un outil qui contrôle la présentation et la logique liée à la présentation » (source : Documentation Django). Django autorise donc ***certains éléments*** de logique dans ses gabarits. Mais la logique qui permet, par exemple, d'appeler la base de données, n'est pas du tout liée à la présentation et appartient strictement à la vue.

Jetons un coup d'œil à notre page dans le navigateur :



La page « hello ».

Nous avons une virgule de fin un peu gênante, mais plutôt que de passer du temps à la corriger, mettons à jour la boucle pour générer nos groupes sous forme de liste HTML non ordonnée, comme nous l'avions fait auparavant :

html

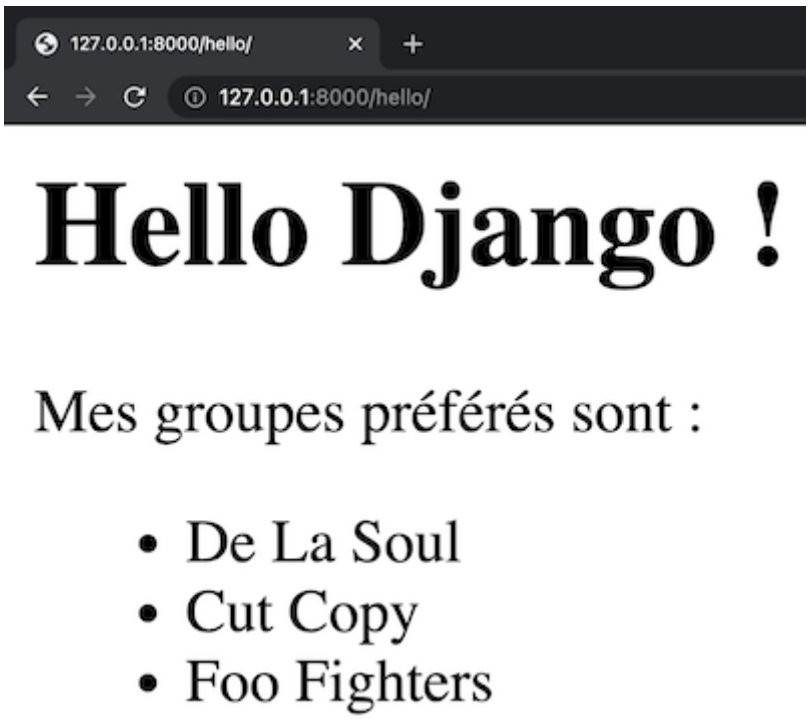
```
# listings/templates/listings/hello.html

```

```
<p>Mes groupes préférés sont :</p>
<ul>
    {% for band in bands %}
    <li>{{ band.name }}</li>
    {% endfor %}
</ul>
```

Remarquez que les balises d'ouverture et de fermeture `<ul>` sont en ***dehors*** de la boucle : nous voulons qu'elles n'apparaissent qu'une seule fois. Les balises `<li>` sont à ***l'intérieur*** de la boucle, car nous avons besoin qu'elles soient répétées : une fois pour chaque groupe de la liste.

Nous revoilà avec notre liste, mais maintenant nous la générons avec une boucle.



La page « hello ».

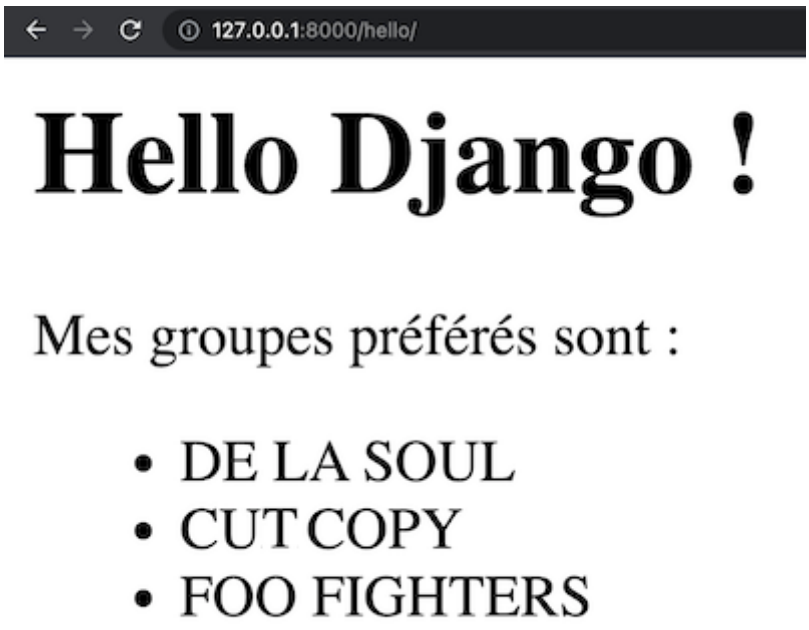
Utilisez un filtre de gabarits (par exemple : tronquer) dans un gabarit

Parfois, lorsque nous injectons une variable dans notre gabarit, nous voudrions appliquer un certain formatage. Les **filtres de gabarits** peuvent nous y aider. Imaginons que nous voulions afficher les noms de nos groupes dans une case différente. Nous pourrions utiliser le filtre `lower` (minuscules) ou `upper` (majuscules), comme ceci :

html

```
# exemple de code

<li>{{ band.name|upper }}</li>
```



La page « hello ».

Vous appliquez un filtre à une variable en utilisant la barre verticale (`|`).

Les filtres ne se limitent pas à la mise en forme du texte. Voyons comment nous pourrions afficher le nombre de groupes dans notre liste, en appliquant le filtre `length` à notre liste de groupes :

html

```
# exemple de code

<p>J'ai {{ bands|length }} groupes préférés.</p>
```

Utilisez une instruction « if » dans un gabarit

Nous pouvons également utiliser une logique d'embranchements dans nos gabarits.

Dans notre gabarit, sous notre liste de groupes, ajoutons le code suivant :

html

```
<p>
    J'ai..
    {% if bands|length < 5 %}
        peu de
    {% elif bands|length < 10 %}
        quelques
    {% else %}
        beaucoup de
    {% endif %}
        groupes préférés.
</p>
```

Les instructions `if` , `elif` et `else` sont également semblables à la façon dont nous les écrivons en Python, mais là encore, nous omettons les deux-points de fin (`:`).

Voici ce que donneraient ces deux derniers exemples dans le navigateur :

J'ai 3 groupes préférés.

J'ai quelques groupes préférés.

Dans le navigateur.

Maintenant que vous avez suivi le processus, vous pouvez vérifier que vous avez bien compris en regardant ce screencast, qui décrit l'ensemble du processus de création d'un gabarit.

02:16

C'est à vous ! Créez des gabarits Django pour vos pages

Maintenant que vous avez compris l'importance des gabarits et que vous avez vu certaines des choses utiles qu'ils permettent de faire, il est temps pour vous de créer vos propres gabarits.

Je veux que vous créiez des gabarits pour la vue `listings`, la vue `about` et la vue `contact` que vous avez créées dans le dernier chapitre. Vos gabarits doivent être des pages HTML complètes,

comprenant les balises `<html>`, `<head>`, `<title>` et `<body>`. Pour la vue `listings`, vous devez injecter vos objets `Listing` dans le gabarit.

Si vous avez besoin d'indications, rappelez-vous que vous devez effectuer les opérations suivantes pour chaque gabarit que vous créez :

- Créer un fichier gabarit dans « listings/templates/listings/ » et lui donner l'extension « .html ».
- Déplacer votre HTML hors de la vue, et dans le gabarit.
- Changer la déclaration de retour de la vue pour appeler la méthode `render` et lui passer le chemin de votre fichier de gabarit.
- Passer également un dictionnaire de contexte à la méthode `render`.
- Utiliser des variables de gabarits pour injecter des données dans votre gabarit.
- Utiliser les balises de gabarits pour utiliser les boucles dans votre gabarit si besoin.

En résumé

- Les gabarits sont l'endroit où nous définissons tous les éléments de présentation d'une page ; pour une application web, c'est le HTML.
- La vue peut ainsi se concentrer sur la logique, dont la récupération des données correctes à injecter dans la page.
- Nous injectons des données dans un gabarit à l'aide de variables de gabarits.
- Nous utilisons les balises de gabarits pour les boucles, les embranchements et le formatage dans les gabarits.

Avant de passer au chapitre suivant, je veux que vous jetiez un coup d'œil aux deux fichiers de gabarits que vous avez créés. Voyez-vous des portions de code qui se répètent ?

Maintenant que nous avons nos gabarits, le « T » pour Template de l'architecture MVT, comment éviter de répéter le code entre les gabarits ? La réponse est un gabarit de base, que nous examinerons dans le prochain chapitre.

Indiquer que ce chapitre n'est pas terminé

Sauvegardez des données dans une base de données avec un modèle et une migration

Ajoutez structure et style à votre site grâce à un gabarit de base, du CSS et des fichiers statiques

Les professeurs

Patrick Wampé

Développeur full stack et Data Scientist. Formateur dans plusieurs écoles d'informatique, il a également écrit un livre sur l'IA.

Patrick Heneghan

Software engineer in the UK, coding mostly in Python on backend systems.

Rafiq Hilali

British Software Engineer and Django expert with Lambert Labs. Currently based in BC, Canada.
