

Django - TD 2

Héritage de template

On vous a annoncé plus tôt que DRY est la devise de Django. Pourtant, si l'on regarde les templates, nous pouvons voir énormément de redondance. Nous allons améliorer cela et mettre un peu de style dans notre site.

Template du projet

La première chose que nous allons faire, est de définir un gabarit de base pour le projet.


Pour cela, nous devons d'abord modifier le fichier `settings.py`


Ajoutez le code suivant :

```
TEMPLATES = [
    {
        'BACKEND':
'django.template.backends.django.DjangoTemplates',
        'DIRS': [BASE_DIR / 'templates'], ➡ new
        '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',
            ],
        },
    },
]
```

Ce bout de code permet d'ajouter une liste de chemin dans lequel Django peut trouver un gabarit. Le chemin que l'on définit ici est un dossier qui se nomme `templates` et qui se trouve dans le dossier de base. Créez ce dossier et ajoutez un fichier nommé `_base.html`.

 Le nom de ce fichier est totalement arbitraire, toutefois, nous précédons le nom par un underscore puisque ce fichier est destiné à être étendu. La communauté de Django vous remerciera de respecter cette convention.

 Petit rappel, `BASE_DIR / 'templates'` désigne la concaténation d'un chemin (`BASE_DIR`) qui est défini dans le fichier `settings.py` avec `templates`.

Gabarit de base

Dans le fichier `_base.html`, placez-y le code suivant

`BASE_DIR/templates/_base.html`

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" > content="width=device-width, initial-
scale=1.0">
  <title>{% block title %}MProject{% endblock title %}</title>
</head>
<body>
  <h1>MProject</h1>
  <h2>Un site web de gestion de tâches</h2>

  {% block content %}
  {% endblock content %}
</body>
</html>
```

Les balises `{% block %}` permettent de définir des blocs qui peuvent être surchargés par les gabarits enfants.

 Notez qu'il n'est pas obligatoire de nommer le bloc fermé (`{% endblock content %}`), mais nous préférons par soucis de lisibilité.

Extension du gabarit

Modifier maintenant le code du gabarit `detail.html`

`developer/detail.html`

```
{% extends "_base.html" %}

{% block title %} Détail - {{ developer.first_name }} {{
developer.last_name }} {% endblock title %}
{% block content %}
    <p><Strong>Prénom : {{ developer.first_name }}</Strong></p>
    <p><Strong>Nom de famille : {{ developer.last_name }}
</Strong></p>
{% endblock content %}
```

Ainsi que le fichier `index.html`

`developer/index.html`

```
{% extends "_base.html" %}

{% block title %} Liste des développeurs {% endblock title %}

{% block content %}
{% if developers %}
<ul>
    {% for dev in developers %}
    {#{<li>{{ dev.first_name }}</li>#}
    <li><a href="{% url 'developer:detail' dev.id %}"> {{
dev.first_name }}</a></li>
    {% endfor %}
</ul>
{% else %}
    <p><strong>Il n'y a aucune développeur enregistré !</strong>
</p>
{% endif %}
{% endblock content %}
```

Les gabarits ont été modifiés afin que les morceaux de code soient placés correctement au sein des blocs `title` et `content` que nous avons défini dans le gabarit de base. Nous avons ajouté la balise `{% extends '_base.html' %}` dans les deux derniers gabarit afin de notifier à Django qu'ils héritent du gabarit de base.

Vérifiez que votre site fonctionne toujours aussi bien que avant. ★

Pour plus d'information sur les balises, lisez cette [page](#).

Un peu de style avec bootstrap 4 et font-awesome

Maintenant que nous avons un gabarit de base, nous allons lui ajouter un peu de style.

Dans le head du fichier `_base.html`, nous ajoutons les liens vers un CDN `bootstrap4` ainsi que vers `fontawesome` pour agrémenter notre projet de quelques logos.

```
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap
.min.css">
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.
js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.16.0/umd/po
pper.min.js"></script>
<script
src="https://maxcdn.bootstrapcdn.com/bootstrap/4.5.2/js/bootstrap.m
in.js"></script>
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/4.7.0/css/font-awesome.min.css">
```

Ajoutons un peu de forme pour l'entête du site

```
<div class="p-3 bg-primary">
  <h1 class="display-1">MProject</h1>
  <h2>Un site web de gestion de tâches</h2>
</div>
```

Ajoutons un menu

```

<nav class="navbar navbar-expand-sm bg-primary navbar-dark border-top border-white">
  <ul class="navbar-nav">
    <li id="nav-home" class="nav-item">
      <a class="nav-link" href="#"><i class="fa fa-home"></i>
    </a>
    </li>
    <li id="nav-dev" class="nav-item active">
      <a class="nav-link" href="{% url 'developer:index' %}">Developers</a>
    </li>
    <li id="nav-task" class="nav-item">
      <a class="nav-link" href="#">Tasks</a>
    </li>
  </ul>
</nav>

```

Nous n'avons pas ajouté de fonctionnalité, mais notre site est maintenant un peu plus habillé. Passez de la vue `detail` à la vue `index` et profiter du site mis en style.

Question ★

Quel bout de code permet de revenir à la liste des développeurs lorsque je clique sur le menu `Developers` ?

Ajout d'un développeur

Pour le moment, nous devons passer par le `shell` afin de pouvoir ajouter un développeur. Ce n'est pas très pratique.

Nous allons ajouter au sein de l'index de développeur, la possibilité d'ajouter un développeur.

Ainsi, sur la page de l'index, nous aurons la liste des développeurs ainsi qu'un formulaire permettant d'en ajouter un.

Ajout du formulaire

Ajoutez ce morceau de code dans le gabarit `index.html`

`index.html`

```
#...
{% else %}
    <p><strong>Il n'y a aucune développeur enregistré !
</strong>/p>
{% endif %}

<form action="{% url 'developer:create' %}" method="post">
new
    {% csrf_token %}
new

    <label for="first_name">First name</label>
new
    <input type="text" name="first_name" required>
new
    <label for="last_name">Last name</label>
new
    <input type="text" name="last_name" required>
new
    <button type="submit">Create</button>
new
</form>
new
{% endblock content %}
```

Un résumé rapide :

- Ce gabarit affiche maintenant un formulaire avec deux champs texte et un bouton de création. Notez que dans ce formulaire, nous avons nommé les deux entrées `first_name` et `last_name`. Ce sont les concepts de base des formulaires HTML.
- Nous avons défini `{% url 'developer:create' %}` comme attribut action du formulaire, et nous avons précisé `method="post"`. L'utilisation de `method="post"` (par opposition à `method="get"`) est très importante, puisque le fait de valider ce formulaire va entraîner des modifications de données sur le serveur. À chaque fois qu'un formulaire modifie des données sur le serveur, vous

devez utiliser `method="post"`. Cela ne concerne pas uniquement Django ; c'est une bonne pratique à adopter en tant que développeur Web.

- Comme nous créons un formulaire POST (qui modifie potentiellement des données), il faut se préoccuper des attaques inter-sites. Heureusement, vous ne devez pas réfléchir trop longtemps car Django offre un moyen pratique à utiliser pour s'en protéger. En bref, tous les formulaires POST destinés à des URL internes doivent utiliser la balise de gabarit `{% csrf_token %}`.

Url et vue pour la création de développeur

Maintenant, nous allons créer une vue Django qui récupère les données envoyées pour nous permettre de les exploiter. D'abord, nous devons ajouter un chemin vers cette nouvelle vue.

`developer/urls.py`

```
app_name = 'developer'
urlpatterns = [
    path('', views.index, name='index'),
    path('<int:developer_id>', views.detail, name='detail'),
    path('create', views.create, name='create'),
    new
]
```

et maintenant, ajoutons la vue

`developer/views.py`

```

#...
from django.http import HttpResponse, HttpResponseRedirect ➡
ajout HttpResponseRedirect
from django.urls import reverse ➡ new
#...
def create(request):
    ➡ new
    Developer.objects.create(
new
        first_name=request.POST['first_name'], ➡
new
        last_name = request.POST['last_name'] ➡
new
    ) ➡
new
    # Toujours renvoyer une HttpResponseRedirect après avoir
    géré correctement
    # les données de la requête POST. Cela empêche les données
    d'être postées deux
    # fois si l'utilisateur clique sur le bouton précédent.
    return HttpResponseRedirect(reverse('developer:index')) ➡
new

```

Ce code contient quelques points encore non abordés dans ce tutoriel :

- `request.POST` est un objet similaire à un dictionnaire qui vous permet d'accéder aux données envoyées par leurs clés. Dans ce cas, `request.POST['first_name']` et `request.POST['last_name']` renvoient le prénom et nom du développeur sous forme d'une chaîne de caractères. Les valeurs dans `request.POST` sont toujours des chaînes de caractères. Pensez donc à réaliser une transformation si le type de votre entrée n'est pas de nature `string`.

Parenthèse Python 🐍

En Python vous pouvez convertir une chaîne de caractère en un entier grâce à la fonction `int()`. Par exemple : `int("42")`.

- Notez que Django dispose aussi de `request.GET` pour accéder aux données GET de la même manière – mais nous utilisons explicitement `request.POST` dans notre code, pour s'assurer que les données ne sont modifiées que par des requêtes POST.

- Après la création d'un développeur, le code renvoie une `HttpResponseRedirect` plutôt qu'une `HttpResponse` normale. `HttpResponseRedirect` prend un seul paramètre : l'URL vers laquelle l'utilisateur va être redirigé (voir le point suivant pour la manière de construire cette URL dans ce cas).
- Comme le commentaire Python l'indique, vous devez systématiquement renvoyer une `HttpResponseRedirect` après avoir correctement traité les données POST. Ceci n'est pas valable uniquement avec Django, c'est une bonne pratique du développement Web.
- Dans cet exemple, nous utilisons la fonction `reverse()` dans le constructeur de `HttpResponseRedirect`. Cette fonction nous évite de coder en dur une URL dans une vue. On lui donne en paramètre la vue vers laquelle nous voulons rediriger ainsi que la partie variable de l'URL qui pointe vers cette vue. Dans ce cas, en utilisant l'URLconf défini précédemment, l'appel de la fonction `reverse()` va renvoyer la chaîne de caractères `developer/`. Cette URL de redirection va ensuite appeler la vue `index` pour afficher la liste des développeurs.

Les classes formulaires

Nous allons simplifier les étapes de créations de formulaire grâce aux classes formulaires.

Dans le dossier `Developer`, ajoutez un fichier `forms.py`. Dans celui-ci ajoutez le code suivant :

Créez un formulaire

`developer/forms.py`

```
from django import forms
developer.models
from .models import Developer

class DeveloperForm(forms.Form):
    first_name = forms.CharField(label="First name",
                                max_length=100)
    last_name = forms.CharField(max_length=100)
```

Nous définissons ainsi une nouvelle classe `DeveloperForm`. Celles-ci possède les deux mêmes champs que le modèle associé.

Ajoutez le formulaire au gabarit

Nous allons maintenant modifier le gabarit afin que celui-ci affiche le formulaire. Enlevez tout ce qui a trait aux champs et ajoutez `{{ form }}`.

 Vous pouvez mettre le formulaire en forme de différente façon.

- `{{ form.as_table }}`
- `{{ form.as_p }}`
- `{{ form.as_ul }}`

`developer/index.html`

```
<form action="{% url 'developer:create' %}" method="post">
    {% csrf_token %}

    <!--<label for="first_name">First name</label>
    <input type="text" name="first_name" required>
    <label for="last_name">Last name</label>
    <input type="text" name="last_name" required>-->
    {{ form }}
    <button type="submit">Create</button>
</form>
```

Envoyez le formulaire au gabarit

Le gabarit n'est évidemment pas en mesure de deviner quel formulaire il doit afficher. Il est de la responsabilité de la vue d'ajouter le formulaire au contexte.

`developer/views.py`

```
#...
from .forms import DeveloperForm

def index(request):
    context = {
        'developers': Developer.objects.all(),
        'form': DeveloperForm,  # new
    }

    return render(request, 'developer/index.html', context)
#...
```

Valider le formulaire

Nous allons maintenant utiliser ce formulaire afin d'obtenir les données saisies par l'utilisateur.

developer/views.py

```
#...
def create(request):
    form = DeveloperForm(request.POST)  # new

    if form.is_valid():  # new
        Developer.objects.create(  # new
            first_name=form.cleaned_data['first_name'],  # new
            last_name=form.cleaned_data['last_name']  # new
        )  # new
        # Toujours renvoyer une HttpResponseRedirect après avoir
        # géré correctement
        # les données de la requête POST. Cela empêche les données
        # d'être postées deux
        # fois si l'utilisateur clique sur le bouton précédent.
        return HttpResponseRedirect(reverse('developer:index'))
#...
```

Notez que nous n'utilisons plus l'instruction `request.POST['xxx']` pour récupérer la donnée associée à un champ, mais `form.cleaned_data['first_name']`. Cela a plusieurs impacts.

1. Il est nécessaire de demander la validité (`is_valid` du formulaire avant d'obtenir une donnée *nettoyée*.)

2. Une donnée nettoyée n'est pas nécessairement un string. Ainsi, pour un champ de type `IntegerField`, la donnée retournée sera de type entier.

Un formulaire DRY

Vous l'avez peut-être remarqué, mais dans le modèle, les champs avaient une longueur de 200 caractères. Dans le formulaire de 100. Ce type d'incohérence apparaît lorsque nous oublions le principe DRY.

Django a prévu une meilleure manière de procéder afin de créer un formulaire sur base d'un modèle.

`developer/forms.py`

```
from django import forms

from .models import Developer

class DeveloperForm(forms.ModelForm): ➡ forms.ModelForm
    plutôt que forms.Form
    # first_name = forms.CharField(label="First name",
    max_length=100) ➡ old
    # last_name = forms.CharField(label='Last name',
    max_length=100) ➡ old
    class Meta: ➡ new
        model = Developer ➡ new
        fields = ['first_name', 'last_name'] ➡ new
```

Et voilà, nous avons un formulaire basé sur le modèle `Developer`. Et surtout, nous respectons le principe DRY !

Vues génériques

En Django, il y a les vues fonctions que nous avons déjà vues, mais il y a aussi les vues génériques (vues classes). Ces dernières, plus récentes, ont pour objectif de simplifier davantage la création de vue.

De base, ces vues sont utilisées pour les cas classiques du développement Web : récupérer les données depuis la base de données suivant un paramètre contenu dans l'URL, charger un gabarit et renvoyer le gabarit interprété.

Les vues génériques permettent l'abstraction de pratiques communes, à un tel point que vous n'avez pas à écrire de code Python pour écrire une application.

Nous allons convertir notre application de gestion de projet pour qu'elle utilise le système de vues génériques. Nous pourrions ainsi supprimer une partie de notre code. Nous avons quelques pas à faire pour effectuer cette conversion. Nous allons :

- Convertir l'URLconf.
- Supprimer quelques anciennes vues désormais inutiles.
- Introduire de nouvelles vues basées sur les vues génériques de Django.

DevDetailView

Vue DevDetailView

Dans le fichier `developer.views.py`, crée la classe `DevDetailView` et supprimez la fonction `detail()`.

`developer.view.py`

```
class DevDetailView(DetailView):
    model = Developer
    template_name = 'developer/detail.html'

# def detail(request, developer_id):
#     #developer = Developer.objects.get(pk=developer_id)
#     developer = get_object_or_404(Developer,
# pk=developer_id)
#     return render(request, 'developer/detail.html',
# {'developer': developer})
```

- Nous utilisons ici la vues générique : `DetailView`. Cette vue permet l'abstraction des concepts affichés une page détaillée pour un type particulier d'objet (ici `Developer`).
- Par défaut, la vue générique `DetailView` utilise un gabarit appelé `<nom app>/<nom modèle>_detail.html`. Dans notre cas, elle utiliserait le gabarit `"developer/developer_detail.html"`. L'attribut `template_name` est utilisé pour signifier à Django d'utiliser un nom de gabarit spécifique plutôt que le nom de gabarit par défaut. Dans notre cas, nous avons choisi de renommer le

template, mais cela n'était pas obligatoire. En revanche, cela le devient si vous devez afficher de deux manières différentes un même modèle.

- Dans les parties précédentes de ce tutoriel, le template `detail.html` a été renseigné avec un contexte qui contenait la variable de contexte `developer`. Pour `DetailView`, la variable `developer` est fournie automatiquement ; comme nous utilisons un modèle nommé `Developer`, Django sait donner un nom approprié à la variable de contexte.

La vue générique `DetailView` s'attend à ce que la clé primaire capturée dans l'URL s'appelle "pk", nous allons donc changer `developer_id` en `pk` pour la vue générique.

URL DevDetailView

Nous l'avons vu, `path` prend en deuxième paramètre une fonction vue. La transformation de la vue générique vers une vue fonction se fait grâce à l'appel à la méthode `as_view()`.

`developer.urls.py`

```
from .views import DevDetailView
#...
urlpatterns = [
    path('', views.index, name='index'),
    #path('<int:developer_id>', views.detail, name='detail'),
    ➡ old
    path('<int:pk>', DevDetailView.as_view(), name='detail'),
    ➡ new
    path('create', views.create, name='create'),
]
```

⚠ Pourquoi y a-t-il les parenthèses après `DevDetailView.as_view()` et pas après `views.detail` ?

Le deuxième paramètre de `path` est une fonction qui sera appelée lorsque l'url coïncidera avec le premier paramètre. Ainsi, dans le cas d'une vue fonction, nous donnons en paramètre la fonction `detail`. Dans le cas d'une vue générique, nous donnons en paramètre le retour de la fonction `as_view()` qui est une fonction !

IndexView

Cela va se compliquer un petit peu. En effet, nous avons une liste des développeurs un peu particulière puisqu'elle est suivie d'un formulaire de création. Nous allons procéder par étape afin de rendre l'implémentation de la classe générique aussi claire que possible.

Vue IndexView

Commençons par créer notre classe comme si nous n'avions pas de formulaire.

`developer.view.py`

```
from django.views.generic import DetailView, ListView  🖱 On
ajoute ListView

#...

class IndexView(ListView):  🖱
    new
    model = Developer  🖱 new
    template_name = "developer/index.html"  🖱 new
    context_object_name = 'developers'  🖱 new
```

- Nous créons une nouvelle classe qui hérite de `ListView`.
- Nous indiquons que la vue est faite pour le modèle `Developer`.
- À l'instar d'une `DetailView` un nom de template est généré automatiquement. Dans le cas d'une `ListView`, le nom généré automatiquement est le suivant : `<nom_app>/<nom_modèle><suffixe_gabarit>.html`. Étant donné que le suffixe par défaut est `_list`, nous aurions pour notre modèle : `developer/developer_list.html`. Nous modifions ce nom de template afin qu'il corresponde à ce qu'il y a déjà dans le gabarit (ce changement pourrait aussi se faire au niveau du gabarit).
- Nous modifions également le nom de la variable du contexte qui contient la liste des développeur. Par défaut, celle-ci aurait pour nom : `developer_list`.

Ajout du formulaire à IndexView

Nous avançons, mais nous n'avons pas ajouté notre formulaire dans le contexte. Pour cela, il est nécessaire de réécrire la méthode `get_context_data()`.

```

from django.views.generic import DetailView, ListView

#...

class IndexView(ListView):
    model = Developer
    template_name = "developer/index.html"
    context_object_name = 'developers'

    def get_context_data(self, **kwargs):
        new
        context = super(IndexView,
self).get_context_data(**kwargs)    new
        context['form'] = DeveloperForm
        new
        return context
        new

```

Parenthèse Python 🐍

En ☕ Java, nous écrivons `super.getContextData(...)` afin d'appeler la fonction de la classe mère. En python, il est nécessaire de donner la classe en premier paramètre et `self` en second paramètre.

Ainsi,

1. Nous chargeons dans la variable `context` le contexte tel qu'il était défini précédemment, c'est-à-dire contenant la variable `developers`.
2. Ensuite, nous ajoutons une clé `form` et le formulaire à utiliser `DeveloperForm`. Nous retournons ensuite le contexte définit au sein de la variable `context`.

URL IndexView

Il est maintenant temps d'associer une url à notre nouvelle classe vue. Rien de plus simple, vous avez fait quelque chose de très similaire avec `DevDetailView`.

```

developer.urls.py

```



```

from .views import DevDetailView
#...
urlpatterns = [
    #path('', views.index, name='index'),           ➡ old
    path('', IndexView.as_view(), name='index'),     ➡ new
    path('<int:pk>', DevDetailView.as_view(), name='detail'),
    path('create', views.create, name='create'),
]

```

Vue générique et Mixin

C'est bien gentil tout ça, mais tout cela m'a l'air bien compliqué et je ne sais pas où vous avez été chercher l'information. 😞

En réalité, tout cela est relativement simple. Surtout quand on sait où chercher l'information.

Les vues génériques sont basées sur le principe de Mixin. Wikipédia le définit assez simplement ce principe de la manière suivante

Concept de Mixin

"En programmation orientée objet, un mixin ou une classe mixin est une classe destinée à être composée par héritage multiple avec une autre classe pour lui apporter des fonctionnalités. C'est un cas de réutilisation d'implémentation. Chaque mixin représente un service qu'il est possible de greffer aux classes héritières. "

[Wikipédia](#)

Maintenant que vous savez ce que c'est, il vous suffit de savoir quelle fonctionnalité est greffée à votre classe. Vous trouverez ces informations dans la documentation. Par exemple, pour `DetailView` vous trouverez les Mixin utilisés et donc les fonctionnalités [ici](#).

Quelques exemples :

- Comment je sais que le template par défaut pour une classe qui hérite de `ListView` est `<nom_app>/<nom_modèle><suffixe_gabarit>.html` ? Cela est ajouté grâce au [MultipleObjectTemplateResponseMixin](#) dont hérite la classe `ListView`.
- Et pour `get_context_data()` ? Alors là c'est dans [ContextMixin](#)

Nous vous recommandons également d'aller jeter un coup d'oeil de temps à autre dans le code des vues génériques. Il vous apprendra beaucoup sur leur fonctionnement. Il se trouve [ici](#).

Modal, include et Crispy

Le modal

On peut imaginer que pour une utilisation normale de l'application, l'ajout d'un développeur se fait de manière occasionnelle.

Nous allons donc vous proposer de mettre ce formulaire au sein d'un *Modal*.

Un modal est une sorte de boîte de dialogue qui est affichée devant la page courante lorsqu'un évènement survient ou que l'utilisateur en fait la demande.

developer/index.html

```
<!--<form action="{% url 'developer:create' %}"
method="post">    🖱️old
    {% csrf_token %}
    🖱️old
    {{ form }}
    🖱️old
    <button type="submit">Create</button>
    🖱️old
</form>
    🖱️old
-->

<!-- Ajout d'un bouton pour faire apparaître la boîte de
dialogue 🖱️ début du nouveau block -->
<button type="button" class="btn btn-primary" data-
toggle="modal" data-target="#add-dev-modal">Add user</button>

<!-- Ajout du modal contenant le formulaire -->
<div class="modal fade " id="add-dev-modal">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h4 class="modal-title">New developer</h4>
                <button type="button" class="close" data-
dismiss="modal"><i class="fa fa-close"></i></button>
            </div>
            <div class="modal-body">
```

```

        <form action="{% url 'developer:create' %}"
method="post">
            {% csrf_token %}
            {{ form }}
            <div>
                <button class="btn btn-primary"
type="submit">Créer</button>
            </div>
        </form>
    </div>
</div>
</div>
</div>
</div> <!-- 👉 fin de l'ajout -->

```

Modal et respect du Dry

Il est fort probable que l'ajout d'un utilisateur puisse se faire à partir de plusieurs fenêtre. Et même si cela ne se produit pas dans ce projet, il reste de bonne pratique de le supposer.

Nous allons donc extraire celui-ci afin de pouvoir le réutilise plus tard, dans un autre template, si besoin en est.

Dans le dossier `developer/templates`, ajoutez un nouveau template `_create_dev_modal.html`. Comme discuté précédemment, le nom de notre nouveau template commence par un `_`. En effet, celui-ci ne sera jamais utilisé indépendamment d'un autre template.

Copiez-y tout le code que vous venez d'ajouter dans le fichier `developer/index.html`.

`developer/_create_dev_modal.html`

```

<!-- Ajout d'un bouton pour faire apparaître la boîte de
dialogue -->
<button type="button" class="btn btn-primary" data-
toggle="modal" data-target="#add-dev-modal">Add user</button>

<!-- Ajout du modal contenant le formulaire -->
<div class="modal fade " id="add-dev-modal">
    <div class="modal-dialog modal-dialog-centered">
        <div class="modal-content">
            <div class="modal-header">
                <h4 class="modal-title">New developer</h4>

```

```

        <button type="button" class="close" data-
dismiss="modal"><i class="fa fa-close"></i></button>
    </div>
    <div class="modal-body">
        <form action="{% url 'developer:create' %}"
method="post">
            {% csrf_token %}
            {{ form }}
            <div>
                <button class="btn btn-primary"
type="submit">Créer</button>
            </div>
        </form>
    </div>
</div>
</div>
</div>

```

Dans `developer/index.html`, remplacez tout ce code par l'inclusion du fichier `developer/_create_dev_modal.html`. L'inclusion se fait grâce à la balise `{% include '<nom du fichier>' %}`.

`developer/index.html`

```

#...
</ul>
{% else %}
    <p><strong>Il n'y a aucune développeur enregistré !
</strong></p>
{% endif %}

<!-- bloc modal --> 🖱️ old
{% include 'developer/_create_dev_modal.html' %} 🖱️ new
{% endblock content %}

```

Crispy

Si vous avez bien suivi le tutoriel jusqu'à maintenant, vous avez peut-être choisi d'utiliser `{{ form.as_p }}` ou autre pour avoir un formulaire un peu plus joli.

Dans Django, il est possible d'ajouter facilement des apps externes. Nous allons illustrer cela par l'ajout d'une app nommée Crispy. Elle permet de rendre un peu plus joli les formulaires.

1. Installez le module. Pour cela, saisissez la commande `python -m pip install django-crispy-forms`
2. Ajoutez django-crispy-forms aux applications installées

`settings.py`

```
INSTALLED_APPS = [  
    #...  
    'django.contrib.staticfiles',  
  
    #My apps  
    'developer.apps.DeveloperConfig',  
  
    #Third-party app                ➡ new  
    'crispy_forms',                ➡ new  
]
```

3. Configuré le pack à utiliser en ajoutant la variable `CRISPY_TEMPLATE_PACK` au fichier `settings.py`.

```
# CRISPY FORM CONFIGURATION  
CRISPY_TEMPLATE_PACK = 'bootstrap4'
```

4. Modifiez `{{ form }}` ou `{{ form.as_qqc }}` par `{{ form|crispy }}` et enfin, chargez le tag crispy dans votre template formulaire. Cela se fait grâce à la balise `{% load %}`.

`developer/_create_dev_modal.html`

```
{% load crispy_forms_tags %}                ➡ new  
  
<button type="button" class="btn btn-primary" data-  
toggle="modal" data-target="#add-dev-modal">Add  
user</button>  
#...  
  
        <!-- {{ form.as_p }} -->                ➡ old  
        {{ form|crispy }}                    ➡ new  
#...
```

Vous venez de terminer de rendre votre formulaire propre et réutilisable.