Objectifs de la feuille

- Améliorations ergonomiques
- Histoire de performances
- Un peu de décoration
- Barre de recherche
- Exercice récapitulatif

Améliorations ergonomiques

Les vues qui listent les produits (ProductListView), les déclinaisons de produits (ProductItemListView) et les attributs de produis (ProductAttributeListView) ressemblent enfin à des vues dignes de ce nom avec la possibilité de créer, éditer, modifier voire supprimer.

En revanche, je ne sais pas vous, mais mes vues restent pauvres en terme d'informations ramenées à l'écran. Concernant les déclinaisons, seul le code est visible. Concernant les attributs, seul le nom est présent. Heureusement que ma vue qui listent les produits est un peu plus fournie avec les nom, code, prix HT , prix TTC et la date de création.

Je vous propose de travailler sur les vues les moins chargées. Commençons par la vue qui liste les attributs ProductAttributeListView. Je veux afficher toutes les informations, à savoir l'attribut ainsi que ses valeurs, et pour chacune d'elles, leur position. Très simplement, nous allons modifier le queryset dans la méthode appropriée get_queryset( ) de la classe ProductAttributeListView:

```
class ProductAttributeListView(ListView):
        model = ProductAttribute
        template_name = "monapp/list_attributes.html"
        context_object_name = "productattributes"

        def get_queryset(self ):
                return ProductAttribute.objects.all()

        def get_context_data(self, **kwargs):
                context = super(ProductAttributeListView, self).get_context_data(**kwargs)
                context['titremenu'] = "Liste des attributs"
                return context
```

Quant au template list_attributes.html, il devient ceci (je vous montre ici uniquement le block contenu) :

```
{% block contenu %}
<table class="table">
	<thead>
		<th>Nom</th>
		<th>Valeurs</th>
		<th>Position</th>
		<th>Actions</th>
	</thead>
	<tbody>
		{% for attribute in productattributes %}
		<tr>
			<td>{{ attribute.name}}</td>
			<td>
				{% for value in attribute.productattributevalue_set.all %}
					<p>{{ value.value }}</p>
				{% empty %}
					<p>Aucune valeur</p>
				{% endfor %}
			</td>
			<td>
				{% for value in attribute.productattributevalue_set.all %}
					<p>{{ value.position }}</p>
				{% empty %}
					<p>Aucune valeur</p>
				{% endfor %}
			</td>
			<td>
<a href="{% url 'attribute-detail' attribute.id %}" class="btn btn-primary mt-2">Détails</a>
<a href="{% url 'attribute-update' attribute.id %}" class="btn btn-warning mt-2">Modifier</a>
<a href="{% url 'attribute-delete' attribute.id %}" class="btn btn-danger mt-2">Supprimer</a>
			</td>
		</tr>
		{% endfor %}
	</tbody>
</table>
{% endblock %}
```

DJANGO met à disposition l'ensemble des couples valeur/position d'un attribut dans la variable productattributevalue_set. Il suffit donc de boucler dessus pour obtenir les informations. Regardons vite le résultat à l'adresse http://127.0.0.1:8000/monapp/attributes/list. Vous devriez avoir le nom de l'attribut agrémenté de ses valeurs et positions.

Idéalement, nous devons retrouver ce même niveau d'informations dans la vue de détail d'un attribut. Modifions la classe ProductAttributeDetailView :

```
class ProductAttributeDetailView(DetailView):
        model = ProductAttribute
        template_name = "monapp/detail_attribute.html"
        context_object_name = "productattribute"

        def get_context_data(self, **kwargs):
                context = super(ProductAttributeDetailView, self).get_context_data(**kwargs)
                context['titremenu'] = "Détail attribut"
                context['values']=ProductAttributeValue.objects.filter(product_attribute=self.object).order_by('position')
                return context
```

Ici, nous passons par le contexte. Nous interrogeons la table ProductAttributeValue, puis nous filtrons directement sur l'attribut du produit concerné filter(product_attribute=self.object) pour obtenir les informations. Nous les trions par leur position. Enfin, tout ceci est donné au dictionnaire de contexte au travers de la variable values. Cette variable sera utilisé dans le template detail_attribute.html :

```
{% extends 'monapp/base.html' %}

{% block title %}
Mon application DJANGO
{% endblock %}

{% block menu %}
<h1>{{titremenu}}</h1>
{% endblock %}

{% block contenu %}
<h2>{{ productattribute.name }}</h2>

<ul>
{% for value in values %}
<li>{{ value.value }} (Position: {{ value.position }})</li>
{% empty %}
<li>Aucune valeur disponible pour cet attribut.</li>
{% endfor %}
</ul>

<a href="{% url 'attribute-list' %}" class="btn btn-primary mt-2">Retour</a>
<a href="{% url 'attribute-update' productattribute.id %}" class="btn btn-warning mt-2">Modifier</a>
<a href="{% url 'attribute-delete' productattribute.id %}" class="btn btn-danger mt-2">Supprimer</a>

{% endblock %}
```

Rendez vous à l'adresse http://127.0.0.1:8000/monapp/attribute/1 pour voir le résultat.

Forts de cette expérience, nous allons faire de même pour la vue qui listent les déclinaisons ProductItemListView. Je veux afficher toutes les informations, à savoir le produit concerné par la déclinaison ainsi que les attributs associés à la déclinaison (attribut/valeur). Très simplement, nous allons modifier le queryset dans la méthode appropriée get_queryset( ) de la classe ProductItemListView:

```
class ProductItemListView(ListView):
        model = ProductItem
        template_name = "monapp/list_items.html"
        context_object_name = "productitems"

        def get_queryset(self ):
                return ProductItem.objects.all()

        def get_context_data(self, **kwargs):
                context = super(ProductItemListView, self).get_context_data(**kwargs)
                context['titremenu'] = "Liste des déclinaisons"
                return context
```

Quant au template list_items.html, il devient ceci (je vous montre ici uniquement le block contenu) :

```
{% block contenu %}
<table class="table">
        <thead>
                <th>Code</th>
                <th>Produit</th>
                <th>Attributs associés</th>
                <th>Actions</th>
        </thead>
        <tbody>
                {% for item in productitems %}
                <tr>
                        <td>{{ item.code }}</a></td>
                        <td>{{ item.product.name }}</a></td>
                        <td>{% for attribute in item.attributes.all %}
                                <p>{{ attribute.product_attribute.name }} : {{attribute.value }}</p>
                                {% empty %}
                                <p>Aucune valeur</p>
                                {% endfor %}
                        </td>
                        <td>
                <a href="{% url 'item-detail' item.id %}" class="btn btn-primary mt-2">Détails</a>
                <a href="{% url 'item-update' item.id %}" class="btn btn-warning mt-2">Modifier</a>
                <a href="{% url 'item-delete' item.id %}" class="btn btn-danger mt-2">Supprimer</a>
                        </td>
                </tr>
        {% endfor %}
        </tbody>
</table>
{% endblock %}
```

Regardons vite le résultat à l'adresse http://127.0.0.1:8000/monapp/item/list. Vous devriez avoir le code de la déclinaison agrémenté du nom du produit concerné, des attributs associés avec leur nom et leur valeur. Répercutons ce changement sur la vue ProductItemDetailview en passant par une modification du contexte :

```
class ProductItemDetailView(DetailView):
	model = ProductItem
	template_name = "monapp/detail_item.html"
	context_object_name = "productitem"

	def get_context_data(self, **kwargs):
		context = super(ProductItemDetailView, self).get_context_data(**kwargs)
		context['titremenu'] = "Détail déclinaison"

		# Récupérer les attributs associés à cette déclinaison
		context['attributes'] = self.object.attributes.all()

		return context
```

Et le template detail_item.html devient :

```
{% extends 'monapp/base.html' %}

{% block title %}
Mon application DJANGO
{% endblock %}

{% block menu %}
<h1>{{titremenu}}</h1>
{% endblock %}

{% block contenu %}
<h2>{{ productitem.code }}</h2>
	<ul>
		<li>Produit : {{ productitem.product }}</li>
		{% for attribute in attributes %}
			<li>{{ attribute.product_attribute.name }} : {{ attribute.value }}</li>
		{% empty %}
			<li>Aucun attribut associé à cette déclinaison.</li>
		{% endfor %}
	</ul>

<a href="{% url 'item-list' %}" class="btn btn-primary mt-2">Retour</a>
<a href="{% url 'item-update' productitem.id %}" class="btn btn-warning mt-2">Modifier</a>
<a href="{% url 'item-delete' productitem.id %}" class="btn btn-danger mt-2">Supprimer</a>
{% endblock %}
```

Rendez vous à l'adresse http://127.0.0.1:8000/monapp/item/1 pour voir le résultat.

Et les performances dans tout ça ?

Contents vous êtes ? Alors comme cela, pour les vues de listes de déclinaisons et des attributs, vous utilisez dans le queryset :

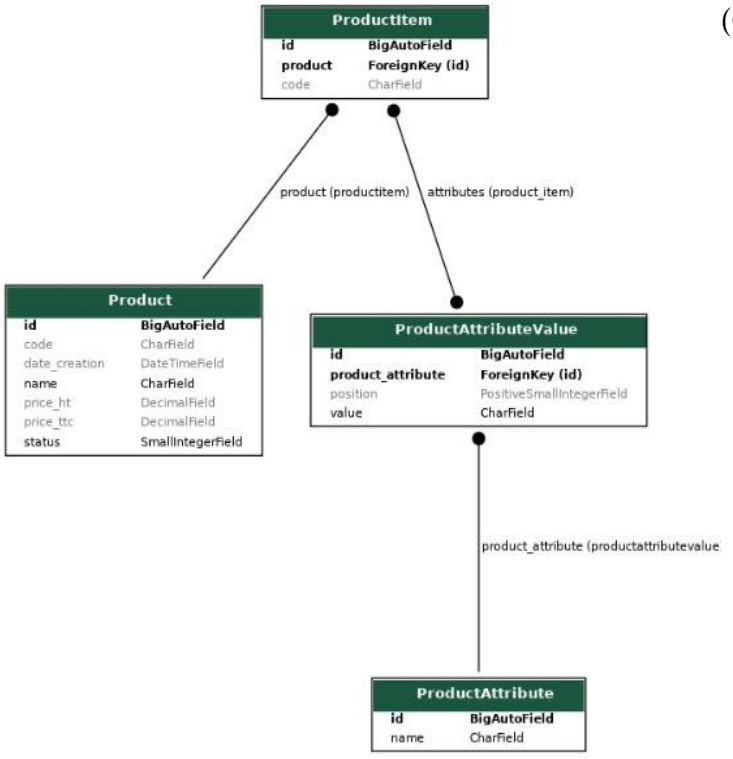
- ProductAttribute.objects.all( ) : DJANGO exécutera des requêtes supplémentaires pour récupérer les relations, en particulier lorsqu'il s'agit d'accéder aux valeurs d'attributs via productattributevalue_set. Si vous avez un grand nombre d'attributs et de valeurs, vous pourriez rencontrer des problèmes de performance...
- ProductItems.objects.all( ) : DJANGO va exécuter une requête SQL distincte pour chaque relation (par exemple, lorsqu'on accède au produit parent ou aux attributs associés). Ce n'est pas idéal pour les performances si vous avez un grand nombre d'objets…

Vous ne me croyez pas ? Allez donc faire un tour du coté de la DJANGO Toolbar :

- dans le premier cas, à l'adresse http://127.0.0.1:8000/monapp/attributes/list, pas moins de 7 requêtes dont 6 requêtes similaires et dupliquées 2 fois (2 x 3 = 6)
- dans le second cas, à l'adresse http://127.0.0.1:8000/monapp/item/list, 21 requêtes !! Je vous laisse regarder en détail mais c'est une catastrophe !

Et nous n'avons que 3 attributs et 4 déclinaisons pour ma part. Inconscients vous êtes ?
Dans notre cas, il serait recommandé d'optimiser les requêtes avec select_related ou prefetch_related.
Qu'est ce que c'est ? Comment ça marche ?

Avant d'aller plus loin, nous avons besoin de nous rappeler notre modèle.



(Génération du MCD au passage)

- pip install django-extensions
- Dans le fichier monprojet/settings.py, ajoute django_extensions dans la liste des INSTALLED_APPS
- pip install pygraphviz
- pip install pydotplus
- ./manage.py graph_models -a -o mcd.png

Revenons à nos deux nouvelles notions.

- select_related est une méthode de l'ORM de DJANGO qui permet de faire des requêtes plus efficaces en préchargeant (ou "prefetching") des relations de type clé étrangère (ForeignKey) ou clé unique (OneToOneField). Cela permet de réduire le nombre de requêtes SQL effectuées lors de l'accès à des objets liés dans des relations de base de données. En d'autres termes, au lieu de faire une requête supplémentaire chaque fois que vous accédez à un objet lié via une clé étrangère, select_related effectue une jointure SQL pour obtenir toutes les données en une seule requête.

- prefetch_related est une méthode de l'ORM DJANGO qui permet de précharger (ou prefetch) des objets liés dans des relations de type Many-to-Many (ManyToManyField) ou des ForeignKey inversés. Il est utile lorsque les relations renvoient plusieurs objets ou lorsqu'on travaille avec des relations qui ne peuvent pas être résolues avec une jointure SQL simple, comme c'est le cas pour select_related. En d'autres termes, là où select_related fait une jointure SQL pour optimiser une relation de type ForeignKey ou OneToOneField, prefetch_related exécute des requêtes séparées pour les objets liés et minimise la duplication de requêtes en les récupérant à l'avance.

Donc si nous faisons le rapprochement avec notre modèle :

- pour aller de la table ProductAttribute à la table ProductAttributeValue, il faut utiliser prefetch_related (ForeignKey inversée = clé primaire de la table source dans la table destination)
- pour aller de la table ProductItem à la table ProductAttributeValue, il faut utiliser prefetch_related (ManyToMany)
- pour aller de la table ProductItem à la table Product, il faut utiliser select_related (ForeignKey classique = clé primaire de la table destination dans la table source)

Soit. Appliquons donc ces règles à notre vue ProductAttributeListView :

```
class ProductAttributeListView(ListView):
    model = ProductAttribute
    template_name = "monapp/list_attributes.html"
    context_object_name = "productattributes"

    def get_queryset(self ):
        return ProductAttribute.objects.all().prefetch_related('productattributevalue_set')

    def get_context_data(self, **kwargs):
        context = super(ProductAttributeListView, self).get_context_data(**kwargs)
        context['titremenu'] = "Liste des attributs"
        return context
```

La méthode prefetch_related('productattributevalue_set') optimise la récupération des valeurs associées en une seule requête, pour chaque attribut. Cela permet d'éviter une requête supplémentaire à chaque fois que l'on accède à productattributevalue_set dans le template. En effet, notre bonne vieille amie DJANGO Toolbar nous indique plus que 2 requêtes au lieu de 7 !!!

Quant à notre vue ProductItemListView

```
class ProductItemListView(ListView):
    model = ProductItem
    template_name = "monapp/list_items.html"
    context_object_name = "productitems"

    def get_queryset(self ):
        return ProductItem.objects.select_related('product').prefetch_related('attributes')

    def get_context_data(self, **kwargs):
        context = super(ProductItemListView, self).get_context_data(**kwargs)
        context['titremenu'] = "Liste des déclinaisons"
        return context
```

On utilise donc

- select_related('product') pour optimiser la requête en récupérant les informations du produit parent en une seule requête SQL.
- prefetch_related('attributes') pour charger efficacement les attributs associés via la relation ManyToManyField.

Nous passons à 14 requêtes.

Un peu de décoration

Oh non, encore du Bootstrap...Non non calmez-vous !

Nous allons parler décorateur au sens DJANGO.

Le décorateur @login_required est utilisé pour restreindre l'accès à certaines vues aux utilisateurs authentifiés dans DJANGO. Voici comment l'utiliser correctement sur une fonction (vue basée sur une fonction) ou sur une classe (vue basée sur une classe).

Lorsqu'il s'agit d'une vue basée sur une fonction (FBV, Function-Based View), le décorateur @login_required est simplement appliqué au-dessus de la fonction :

```
from django.contrib.auth.decorators import login_required
from django.shortcuts import render

@login_required
def my_view(request):
        return render(request, 'my_template.html')
```

Ce décorateur vérifie si l'utilisateur est authentifié avant d'exécuter la vue. Si l'utilisateur n'est pas connecté, il est redirigé vers la page de connexion par défaut. Si vous souhaitez personnaliser l'URL de redirection pour les utilisateurs non authentifiés, vous pouvez ajouter l'argument login_url

```
@login_required(login_url='/monapp/login/')
def my_view(request):
        return render(request, 'my_template.html')
```

Pour les vues basées sur les classes (CBV, Class-Based View), ce qui est notre cas, vous ne pouvez pas utiliser directement le décorateur @login_required. Cependant, DJANGO propose une méthode pour appliquer ce type de décorateur à des vues basées sur des classes en utilisant la fonction method_decorator( )

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator
from django.views.generic import TemplateView

# Ajout du décorateur login_required à une CBV
@method_decorator(login_required, name='dispatch')
class MyView(TemplateView):
        template_name = 'my_template.html'
```

Ce décorateur @method_decorator(login_required, name='dispatch') applique @login_required à la méthode dispatch() de la classe, qui est responsable de la gestion des requêtes HTTP (GET, POST, etc.). Cela garantit que toutes les méthodes de la vue nécessitent que l'utilisateur soit connecté.

Si vous avez plusieurs méthodes dans la classe et que vous ne voulez appliquer @login_required qu'à certaines d'entre elles, vous pouvez aussi utiliser method_decorator à l'intérieur de la classe :

```
class MyView(TemplateView):
    template_name = 'my_template.html'

    @method_decorator(login_required)
    def get(self, request, *args, **kwargs):
        return super().get(request, *args, **kwargs)
```

Si vous souhaitez personnaliser l'URL de redirection pour les utilisateurs non authentifiés, vous pouvez ajouter l'argument LOGIN_URL = '/monapp/login/' dans le fichier monapp/settings.py.

C'est à vous !
Nous souhaitons que les actions de création, modification et suppression, des produits, des déclinaisons et des attributs soit soumis à l'authentification.

Barre de recherche

Nous allons ajouter une barre de recherche sur nos différentes vues listant les produits, les déclinaisons et les attributs. Dans nos différents templates, list_products.html, list_items.html et list_atrributes.html, il nous faut rajouter un bout de code. Personnellement, je l'ai positionné entre le contenu du tableau et le footer :

```
<!-- Barre de recherche -->
<p class="lead mb-3 ml-0">Rechercher un élément...</p>
<form method="GET" action="">
        <input class="form-control" type="text" name="search" placeholder="Tapez ici votre recherche"
        value="{{ request.GET.search }}">
        <button type="submit" class="btn btn-success mt-2">Rechercher</button>
</form>
```

Le formulaire utilise la méthode GET pour soumettre la requête. Le champ de texte a pour nom search, qui correspond au paramètre GET que nous allons récupérer dans la vue.
La balise value="{{ request.GET.q }}" permet de garder le terme de recherche dans le champ après la soumission du formulaire.

Dans le fichier views.py, vous allez modifier la vue ProductListView pour afficher les produits filtrés par la recherche.

```
class ProductListView(ListView):
        model = Product
        template_name = "monapp/list_products.html"
        context_object_name = "products"

        def get_queryset(self ):
                # Surcouche pour filtrer les résultats en fonction de la recherche
                # Récupérer le terme de recherche depuis la requête GET
                query = self.request.GET.get('search')
                if query:
                        # Filtre les produits par nom (insensible à la casse)
                        return Product.objects.filter(name__icontains=query)

                # Si aucun terme de recherche, retourner tous les produits
                return Product.objects.all()

        def get_context_data(self, **kwargs):
                context = super(ProductListView, self).get_context_data(**kwargs)
                context['titremenu'] = "Liste des produits"
                return context
```

La méthode `get_queryset( )` permet de filtrer la liste des produits en fonction du terme de recherche (`search`) que l'on récupère à partir de l'URL (paramètre GET) avec `self.request.GET.get('search')`. Ici, nous utilisons `name__icontains=query`, ce qui filtre les produits dont le nom contient le terme de recherche (insensible à la casse).

C'est à vous ! Faites de même pour les vues `ProductItemListView` et `ProductAttributeListView`.
Attention, toutefois, certaines adaptations seront nécessaires, notamment au niveau du filtre...

Exercice récapitulatif

Cela ne vous aura pas échapper : il manque le système CRUD (Create, Read, Update, Delete) pour le modèle ProductAttributeValue. Pour mettre en place ce système CRUD dans DJANGO, vous devez créer des vues pour chaque action, ainsi que des templates correspondants. Vous utiliserez des vues basées sur des classes (CBV), ce qui facilite grandement le développement.
Voici un guide étape par étape pour mettre en place le CRUD pour ProductAttributeValue :

Étape 1 : Configurer les vues CRUD ☐
1.1. Vue pour lister les valeurs d'attributs (+détail) ☐
1.2. Vue pour créer une nouvelle valeur d'attribut ☐
1.3. Vue pour mettre à jour une valeur d'attribut existante ☐
1.4. Vue pour supprimer une valeur d'attribut ☐

Étape 2 : Créer un formulaire pour ProductAttributeValue ☐

Étape 3 : Créer les templates ☐
3.1. Liste des valeurs d'attributs ☐
3.2. Formulaire de création et de mise à jour ☐
3.3. Confirmation de suppression ☐

Étape 4 : Configurer les URLs ☐