Objectifs de la feuille

- ModelForm
- Validation formulaire
- La vue générique de création DJANGO
- La vue générique de mise à jour DJANGO
- La vue générique de suppression DJANGO

Le modèle définit la forme

Maintenant que nous savons comment créer des formulaires dans DJANGO, la prochaine étape est de créer un formulaire qui permettra aux utilisateurs de créer un nouvel objet Product. On pourrait penser qu'on commencerait par définir une autre classe de formulaire et, pour chaque champ de notre modèle, définir un champ de formulaire correspondant et vous pourriez le faire !

Même si nous construisons deux choses distinctes, un modèle et un formulaire, il semble y avoir beaucoup d'informations dupliquées entre les deux. Nous avons déjà défini chaque champ dans le modèle, en spécifiant les types de données (string, integer) et les règles (par exemple max_length). Pourquoi devrions-nous les définir également dans le formulaire ? Ne serait-il pas formidable de pouvoir laisser le modèle définir le formulaire ? C'est exactement ce à quoi servent les ModelForm.

Dans monapp/forms.py, nous allons définir une nouvelle classe qui hérite de django.forms.ModelForm :

```
class ProductForm(forms.ModelForm):
    class Meta:
        model = Product
        fields = '__all__'
```

La nouvelle classe contient une classe imbriquée, Meta, qui spécifie le modèle pour lequel ce formulaire sera utilisé, et les champs de ce modèle à inclure dans ce formulaire (dans notre cas, tous les champs). Maintenant, utilisons notre ProductForm dans la vue ProductCreateView en le passant au modèle :

```
def ProductCreate(request):

    form = ProductForm()
    Return render(request, "monapp/new_product.html", {'form': form})
```

Prenons soin de mettre à jour le fichier monapp/urls.py avec la nouvelle url :

```
path("product/add/",views.ProductCreate, name="product-add"),
```

Et ensuite nous générons le formulaire dans le template new_product.html:

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

{% block contenu %}
<h1> Création d'un nouveau produit </h1>
<form action="" method="post" novalidate>
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="btn btn-success mt-2">Créer</button>
</form>
{% endblock %}
```

Jetez un coup d'œil à notre page dans le navigateur à l'adresse http://127.0.0.1:8000/product/add/. Nous venons de générer un formulaire pour notre modèle, sans avoir à définir aucun champ de formulaire, nous avons laissé le modèle le faire pour nous ! Et voici la meilleure partie : si nous modifions les champs du modèle, le formulaire est automatiquement mis à jour. Mais nous n'avons pas encore fini : nous devons mettre à jour notre vue pour qu'elle crée un objet Product à partir des valeurs que nous soumettons dans le formulaire.

Nous allons suivre le même schéma que pour le formulaire de « Contact ». Récapitulons le cycle de vie de ce modèle de vue, en style « pseudocode » :

- La demande est transmise à la fonction de vue.
    - S'il s'agit d'une demande GET :
        - Il doit s'agir de la première requête sur cette page et l'utilisateur s'attend à voir un formulaire vierge, prêt à être rempli. Nous créons donc une nouvelle instance vide du formulaire et nous la stockons dans une variable « form ».
    - S'il s'agit d'une demande POST :
        - Il ne s'agit pas de la première demande, mais d'une demande ultérieure à cette page, accompagnée de certaines valeurs de formulaire que l'utilisateur aura soumises. Nous créons donc une instance du formulaire et nous la remplissons avec les valeurs envoyées dans la requête POST. C'est stocké dans une variable « form ».
        - Si la validation du formulaire se déroule avec succès :
            - Nous effectuons l'action, qu'il s'agisse d'envoyer un e-mail, de créer un objet ou de toute autre action.
            - Nous redirigeons l'utilisateur vers une autre vue, peut-être une page de confirmation ou une autre page qui indique que l'action a réussi. Nous arrêtons d'exécuter cette fonction de vue à ce stade. Cependant...
        - Si la validation du formulaire n'a PAS réussi :
            - Le processus de validation a maintenant ajouté des messages d'erreur dans les champs concernés.
            - Nous autorisons l'exécution de cette vue pour continuer à l'étape suivante ci-dessous.

Nous avons maintenant un «formulaire» qui est soit

- nouveau et vide,
- rempli avec des valeurs précédemment soumises et des messages d'erreur pertinents.

Dans les deux cas, nous passons « form » au modèle pour qu'il puisse être affiché dans le navigateur. Voici à quoi cela ressemble en Python, dans notre vue ProductCreate. Notez que nous traitons d'abord le cas POST dans l'instruction if et ensuite le cas GET dans l'instruction else :

```
def ProductCreate(request):

    if request.method == 'POST':
        form = ProductForm(request.POST)
        if form.is_valid():
            product = form.save()
            return redirect('product-detail', product.id)
    else:
        form = ProductForm()

    return render(request, "monapp/new_product.html", {'form': form})
```

Notez que la méthode form.save() ne fait pas qu'enregistrer le nouvel objet dans la base de données : elle renvoie également cet objet, ce qui signifie que nous pouvons l'utiliser à l'étape suivante, où nous redirigeons immédiatement vers le produit nouvellement créé.

Maintenant nous pouvons remplir le formulaire et nous sommes alors redirigés vers la page détaillée du produit que nous venons de créer. Notre nouvelle page fonctionne, mais elle ne sert à rien si nos utilisateurs ne peuvent pas la trouver. Nous devons créer un lien vers elle quelque part. Et si on établissait un lien à partir de la vue en liste de Product, dans le template list_product.html:

```
<a href="{% url 'product-add' %}">Créer un nouveau produit</a>
```

Les ModelForms sont un outil puissant pour créer des modèles rapidement et facilement.C'est formidable que DJANGO puisse générer un formulaire pour nous, mais que faire si le formulaire par défaut ne répond pas tout à fait à nos besoins ?

Vous pouvez créer un formulaire à partir de zéro, comme nous l'avons fait avec le ContactUsForm et l'utiliser. Mais il est également possible de personnaliser un ModelForm.

Disons que nous voulons que les utilisateurs puissent créer un nouveau produit, mais nous ne voulons pas qu'ils puissent modifier les champs price_ttc et status, nous voulons que seuls les administrateurs puissent modifier ces champs (ce qu'ils peuvent faire dans le site d'administration). Nous pouvons les exclure du formulaire, comme ceci :

```
class ProductForm(forms.ModelForm):
        class Meta:
                model = Product
                #fields = '__all__'
                exclude = ('price_ttc', 'status')
```

Et alors ces champs n'apparaîtront pas dans le formulaire. Le formulaire fonctionnera toujours et nous pourrons créer un nouvel objet Product sans aucune erreur, même si nous avons laissé ces champs de côté, parce que status a une valeur par défaut égale à 0 et price_ttc autorise les valeurs NULL avec null=True.

Si vous voulez exclure certains des autres champs (non nullables) du formulaire, vous devrez d'abord leur donner une valeur par défaut, ou les rendre nullables, dans monapp/models.py. Vous devrez également effectuer et exécuter une migration pour mettre à jour le schéma de la base de données, car les valeurs par défaut et les valeurs NULL sont configurées dans la base de données elle-même.

Validation coté client et coté serveur

Pour les deux formulaires que nous avons créés jusqu'à présent, nous avons ajouté novalidate à leur HTML : Cela a désactivé la validation côté client dans votre navigateur. Réactivons-la en enlevant novalidate du gabarit new_product.html. Essayez maintenant de soumettre ce formulaire vide.

La validation côté client offre une expérience utilisateur améliorée. Il valide les données du formulaire avant qu'elles ne soient envoyées au serveur. S'il y a des données non valides, le formulaire affiche un message d'erreur et ne soumet pas les données tant qu'elles ne sont pas valides. Cela évite à l'utilisateur de faire un «aller-retour» vers le serveur pour s'apercevoir qu'il a soumis des données non valides.

Alors pourquoi avons-nous désactivé la validation côté client au début ?… Si le formulaire ne se soumet pas tant que toutes les données ne sont pas valides, nous ne pouvons pas voir ce qui se passe lorsque nous envoyons des données invalides au serveur, et il n'est donc pas possible de démontrer que la validation côté serveur fonctionne.

Mais avons-nous vraiment besoin de la validation côté serveur si nous avons la validation côté client ? Oui, absolument. Et c'est une règle importante pour apprendre le développement web : «Ne jamais faire confiance au client.»

Le problème est que, bien que nous ayons effectivement construit le client, n'importe quel utilisateur peut modifier le HTML en frontal, supprimer la validation et envoyer ce qu'il veut au serveur !

En fait, vous pouvez le faire dès maintenant. Ouvrez l'inspecteur de votre navigateur. Vous pouvez facilement augmenter l'attribut maxlength du champ name à 10000 ou même 100000 !

Si nous n'avions pas de validation côté serveur, un nom de 100 000 caractères pourrait être enregistrée directement dans la base de données. Cela ne ferait probablement pas planter le site et serait plus ennuyeux qu'autre chose. Mais vous pouvez imaginer qu'un pirate informatique pourrait essayer de trouver d'autres moyens d'exploiter cette absence de validation côté serveur.

La sécurité des applications web est un sujet important que tout développeur devrait connaître, mais nous ne pouvons pas tout couvrir dans ce TP. Cela dit, DJANGO intègre de nombreuses bonnes pratiques en matière de sécurité, ce qui vous permet de prendre un bon départ rien qu'en utilisant le framework.

Mais pour l'instant, souvenez-vous :

- La validation côté serveur est ce sur quoi vous vous appuyez pour vous assurer que vous n'acceptez que des données valides dans votre base de données.
- La validation côté client est fournie afin d'améliorer l'expérience de l'utilisateur et, bien qu'utile, elle ne doit pas être utilisée de manière exclusive !

La vue générique de création DJANGO

Est-ce qu'il nous arrive de créer manuellement un formulaire pour un modèle, ou est-ce que nous utilisons toujours un ModelForm ? Vous devriez très rarement avoir à créer manuellement un formulaire pour un modèle à partir de zéro, bien que rien ne vous empêche de le faire. Il faudrait alors que le modèle et le formulaire soient synchronisés l'un avec l'autre. L'utilisation d'un ModelForm dans le premier cas est logique, car vous gardez les choses DRY et il y a moins de code à maintenir.

Vous pouvez passer par la vue générique CreateView que fournit le framework DJANGO. Les changements s'opèrent dans le fichier views.py en transformant notre vue en une classe qui hérite de CreateView :

```
class ProductCreateView(CreateView):
        model = Product
        form_class=ProductForm
        template_name = "monapp/new_product.html"

        def form_valid(self, form: BaseModelForm) -> HttpResponse:
                product = form.save()
                return redirect('product-detail', product.id)
```

Et n'oublions pas la modification dans monapp/urls.py :

```
#path("product/add/",views.ProductCreate, name="product-add"),
path("product/add/",views.ProductCreateView.as_view(), name="product-add"),
```

La vue générique de mise à jour DJANGO

Maintenant qu'on a créé les objets du modèle, on doit être en mesure de les mettre à jour. Il est temps de choisir un motif d'URL pour notre prochaine vue, qui servira à mettre à jour un produit existant. Jusqu'à présent, nous avons :

- product/list pour la liste des produits ;
- product/1/ pour le produit 1, «product/2/» pour le produit 2, etc. ;
- product/add/ pour créer un produit.

Le problème, c'est qu'il faut savoir quel produit changer. Mettons aussi cette information dans l'URL. Ce sera donc «product/1/change/» pour mettre à jour le produit 1, «product/2/change/» pour mettre à jour le produit 2 et ainsi de suite. Là encore, vous pouvez utiliser le verbe «update» ou tout autre mot de votre choix, à la place de «change». Pourquoi pas «product/change/1/», avec l'identifiant à la fin de l'URL ? Ça marcherait aussi ! Lorsque nous choisissons nos chemins, nous pouvons choisir la structure parent-enfant qui nous semble la plus logique. J'aime imaginer que la page de modification est un «enfant» de la page détaillée du produit. Ainsi, si la page détaillée se trouve à l'adresse «product/1/», je place la page des modifications pour ce produit à l'adresse «product/1/change/». Il faut modifier le fichier urls.py en conséquence :

```
path("product/<pk>/update/",views.ProductUpdateView.as_view(), name="product-update"),
```

Vous devriez maintenant connaître la marche à suivre, il faut une nouvelle vue et un gabarit de base pour la mise à jour d'un produit. Plutôt que d'écrire une fonction pour la vue, je crée une nouvelle classe héritant de la vie générique UpdateView fournie par DJANGO.

```
class ProductUpdateView(UpdateView):
        model = Product
        form_class=ProductForm
        template_name = "monapp/update_product.html"

        def form_valid(self, form: BaseModelForm) -> HttpResponse:
                product = form.save()
                return redirect('product-detail', product.id)
```

Et le petit template qui va bien dans le fichier update_product.html :

```
{% extends 'monapp/base.html' %}
{% block contenu %}
<h1> Modification d'un produit </h1>
<form action="" method="post" >
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="btn btn-success mt-2">Modifier</button>
</form>
{% endblock %}
```

Ce qui est bien avec les formulaires dans DJANGO, c'est qu'il n'est pas nécessaire d'en créer un nouveau spécifiquement pour l'opération de mise à jour : nous pouvons réutiliser le formulaire que nous utilisons déjà pour l'opération de création. Après tout, nous ne faisons que mettre à jour les mêmes champs que ceux avec lesquels nous avons travaillé précédemment.

Si vous optez pour l'option d'utiliser une fonction pour la définition de la vue, voilà à quoi cela doit ressembler :

```
def ProductUpdate(request, id):

    prdct = Product.objects.get(id=id)

    if request.method == 'POST':
        form = ProductForm(request.POST, instance=prdct)

        if form.is_valid():
            # mettre à jour le produit existant dans la base de données
            form.save()
            # rediriger vers la page détaillée du produit que nous venons de mettre à jour
            return redirect('product-detail', prdct.id))
    else:
        form = ProductForm(instance=prdct)

    return render(request,'monapp/product-update.html', {'form': form})
```

La différence notoire est que lorsque nous chargeons la vue pour la première fois, plutôt que de générer un formulaire vide, nous voulons préremplir le formulaire avec les valeurs existantes de l'objet que nous modifions.
Nous faisons cela en passant l'objet au mot-clé `instance`, par exemple `ProductForm(instance=prdct)`
Nous devons modifier la vue pour mettre à jour notre objet avec les valeurs que nous renvoyons en POST au serveur. Comme pour nos traitements précédents des formulaires POST, nous devons gérer les scénarios GET et POST dans la vue. Après avoir mis à jour le produit avec les valeurs soumises, nous redirigeons vers la page détaillée du produit.

Il faut enfin créer un lien vers la page de mise à jour. Il existe deux endroits à partir desquels nous pourrions créer un lien vers une page de «mise à jour» du produit. Mettons en œuvre les deux possibilités :

- nous pouvons établir un lien à partir de la vue en liste produit
- nous pouvons établir un lien à partir de la vue détaillée du produit

Dans les deux cas, c'est le même code : c'est à vous de la mettre au bon endroit :

```
<a href="{% url 'product-update' product.id %}">modifier</a>
```

La vue générique de suppression DJANGO

L'opération de suppression est une opération à laquelle il faut faire particulièrement attention. Dès que nous supprimons un objet de la base de données, il disparaît définitivement, il n'y a pas de fonction d'annulation. Nous devons donc être sûrs que c'est ce que l'utilisateur avait vraiment l'intention de faire et que ce n'est pas un accident !

Pour cette raison, les interfaces CRUD exigent souvent deux étapes pour supprimer un objet. Par exemple, dans l'administration de Django, la première étape est une sorte de bouton de suppression. En cliquant sur le bouton, au lieu d'effectuer immédiatement l'opération de suppression, vous êtes redirigé vers une page de confirmation, donnant à l'utilisateur la possibilité de changer d'avis.

Ce n'est qu'après avoir cliqué sur «Yes, I'm sure» (Oui, je suis sûr) que l'opération de suppression sera effectuée dans la base de données. Implémentons quelque chose de similaire dans notre interface CRUD.

Nous allons avoir besoin d'une autre vue pour cette page de confirmation, ce qui signifie que nous avons également besoin d'un nouveau modèle d'URL Suivons le modèle que nous avons établi au chapitre précédent, la page de mise à jour du produit 1 se situait au chemin : «product/1/change/». Donc la page de confirmation pour la suppression du groupe 1 sera : «product/1/delete/».

Allez-y et créez le modèle d'URL, la vue et le gabarit de base pour notre page de confirmation de suppression. Le modèle a juste besoin d'un en-tête pour que nous puissions tester qu'il se charge correctement. Etablissons également un lien entre la page détaillée du produit et la page de confirmation de la suppression. Nous avons déjà procédé de la sorte pour la page de mise à jour dans le chapitre précédent, alors suivez la même méthode et reportez-vous à ce chapitre si vous avez besoin d'un rappel. Je vous laisse le choix de l'implémentation entre une fonction ou une classe pour la vue, mais sachez que DJANGO fournit une classe DeleteView…

Lorsqu'un utilisateur clique sur la page de suppression, la première chose à faire est de confirmer quel objet nous sommes sur le point de supprimer. Nous pouvons le faire en affichant l'un des champs de l'objet. Affichons donc le nom du produit à l'utilisateur, afin qu'il puisse vérifier qu'il s'agit bien de celui qu'il voulait supprimer.

Afin de supprimer réellement l'objet, nous allons une fois de plus utiliser un formulaire et une requête HTTP POST. Mais ce formulaire est différent du ModelForm que nous avons utilisé jusqu'à présent : il ne va pas envoyer de valeurs pour les champs du modèle, parce que nous n'avons besoin de changer aucun de ces champs, nous voulons juste supprimer l'objet dans sa totalité. Le formulaire est si simple que nous pouvons le coder directement dans le template product_delete.html sans créer de nouvelle classe de formulaire :

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

{% block contenu %}
<h1> Suppression d'un produit </h1>
<form action="" method="post" >
        {% csrf_token %}
        <p>Etes-vous sûr de vouloir supprimer le produit : {{ object.name }} ?</p>
        <button type="submit" class="btn btn-success mt-2">Confirmer</button>
</form>

{% endblock %}
```

Le formulaire ne comporte qu'une seule entrée : le bouton d'envoi (deux si vous comptez le jeton CSRF). Nous n'avons besoin de rien de plus.

Mais comment le serveur saura-t-il quel groupe supprimer ? Ne devrions-nous pas au moins envoyer l'id du produit ? Nous envoyons déjà l'identifiant du groupe dans l'URL ! Si nous sommes à l'url «/product/1/change/» et que nous soumettons le formulaire, la requête POST renvoi à la même URL. L'identifiant est transmis à la vue et c'est ainsi que la vue sait quel produit supprimer.

Ensuite dans la vue, nous utilisons le modèle que nous avons déjà utilisé précédemment. Nous redirigeons ensuite l'utilisateur vers la liste des produit, où il verra que le produit qu'il a supprimé n'apparaît plus dans la liste. Remarquez que la validation du formulaire n'est pas nécessaire ici, car il n'y a pas de données à valider (pas de méthode form_valid)

```
class ProductDeleteView(DeleteView):
        model = Product
        template_name = "monapp/delete_product.html"
        success_url = reverse_lazy('product-list')
```

Avec cette vue, notre url est donc :

```
path("product/<pk>/delete/",views.ProductDeleteView.as_view(), name="product-delete"),
```

Si vous avez opté pour une fonction de vue, votre code doit ressembler à ceci :

```
def band_delete(request, id):

    prdct = Product.objects.get(id=id)  # nécessaire pour GET et pour POST

    if request.method == 'POST':
        # supprimer le produit de la base de données
        prdct.delete()

        # rediriger vers la liste des produit
        return redirect('product-list'))

    # pas besoin de « else » ici. Si c'est une demande GET, continuez simplement

    return render(request, 'monapp/prodcut-delete.html', {'object': prdct})
```

Comme dans les deux fonctions de vues précédentes (si vous aviez choisi cette option), nous traitons les scénarios GET et POST. Pas de validation de formulaire car il n'y a pas de données à valider. Pour supprimer l'objet, nous vérifions juste si c'était une requête POST, et ensuite nous appelons `prdct.delete()`. La redirection est la même.

Bien. C'est à vous ! Vous avez du pain sur la planche. Vous allez faire le même travail pour les `ProductItem` et les `ProductAttribute`. A savoir :

- Ajouter une page Créer (modèle d'URL, vue et gabarit) pour chacun des deux éléments;
- Ajouter un formulaire de type ModelForm et l'utiliser dans la nouvelle page pour créer les objets ;
- Lier la liste des objets à la page de création d'un objet ;
- Activer la validation côté client pour tous vos formulaires.2
- Ajouter une page de «Mise à jour» (modèle d'URL, vue et gabarit) pour chacun des deux éléments
- Utiliser le Form existant dans cette page pour mettre à jour les objets ;
- Lier les pages de liste des objets et de détails d'un objet vers la page de mise à jour.
- Ajouter une page de suppression (modèle d'URL, vue et gabarit) pour chacun des deux éléments
- Inclure un formulaire de suppression dans le gabarit
- Rediriger l'utilisateur vers la liste des objets après l'opération de suppression ;
- Lier la page détaillée des objets à la page de suppression.

Voilà. Vive le CRUD !