Objectifs de la feuille

- La vue générique de base
- Les vues génériques d'affichage
- Les vues génériques de connexion et d'inscription
- La vue générique de déconnexion
- Premier formulaire DJANGO

Les vues génériques

Les vues ListProduct(), ListItems() et ListAttributes() sont très courtes et que dire des vues about() et contact(). Très rapidement vous vous rendrez compte que vos vues se ressemblent toutes plus ou moins.

Ces vues représentent un cas classique 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. Ce cas est tellement classique que DJANGO propose un raccourci, appelé le système de vues génériques.

Les vues génériques ajoutent une couche d'abstraction des procédés courants au point où vous n'avez même plus besoin d'écrire du code Python pour écrire une application. Finalement on voudra lister des modèles, puis pouvoir en créer, modifier et enfin supprimer. Ainsi, lorsque nous construisons l'interface utilisateur de notre site, on peut dire que nous construisons une interface CRUD.

Le concept de DJANGO c'est de faciliter au maximum la conception d'un projet. Inutile de répéter du code, on se veut DRY et factoriser le code est aussi intéressant sur la performance du développeur que sur la maintenance.

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

1. Convertir les urls
2. Supprimer quelques anciennes vues désormais inutiles.
3. Introduire de nouvelles vues basées sur les vues génériques de DJANGO.

Pourquoi ces changements de code ?
En général, lorsque vous écrivez une application DJANGO, vous devez estimer si les vues génériques correspondent bien à vos besoins et, le cas échéant, vous les utiliserez dès le début, plutôt que de réarranger votre code à mi-chemin. Mais ce cours s'est concentré intentionnellement sur l'écriture des vues «à la dure» jusqu'à ce TD, pour mettre l'accent sur les concepts de base. Tout comme vous devez posséder des bases de maths avant de commencer à utiliser une calculatrice.

La vue générique de base

Commençons par la vue générique de base: TemplateView . Cette vue se contente de renvoyer un fichier type static, pas de modèles, pas de requêtes en base de données. Nous allons appliquer cette vue générique pour notre «home». Pour cela, il faut agrémenter le fichier monapp/urls.py d'une nouvelle url comme ci-dessous (vous mettrez l'ancienne url en commentaire) :

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

urlpatterns = [
        #path("home", views.home, name="home"),
        ...
        path("home", TemplateView.as_view(template_name="home.html")),
]
```

Et il ne faut pas oublier de créer le fichier home.html dans le répertoire des templates avec ce contenu:

```
<h1>Hello Django</h1>
```

Dans cet exemple, le template home.html sera utilisé comme nous l'avons indiqué. Nul besoin d'instancier quoi que ce soit, la classe TemplateView ne nécessite que l'appel de la méthode as_view . Comme vous le voyez, il ne se passe plus rien dans le fichier view.py

N'oubliez pas de renseigner tous les chemins vers vos templates dans le fichier de configuration de votre projet settings.py :

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

- Les attributs de la classe TemplateView

```
content_type = None
http_method_names = [u'get', u'post', u'put', u'patch', u'delete', u'head', u'options', u'trace']
response_class = <class 'django.template.response.TemplateResponse'>
template_name = None
```

- Les méthodes de la classe TemplateView

```
_allowed_methods(self)
as_view(cls, **initkwargs)
dispatch(self, request, *args, **kwargs)
get(self, request, *args, **kwargs)
get_context_data(self, **kwargs)
get_template_names(self)
http_method_not_allowed(self, request, *args, **kwargs)
options(self,          request, *args, **kwargs)
render_to_response(self, context, **response_kwargs)
```

Vous pouvez dériver TemplateView pour mettre votre vue-classe dans la fichier views.py

```
class HomeView(TemplateView):
    template_name = "monapp/home.html"

    def post(self, request, **kwargs):
        return render(request, self.template_name)
```

L' url dans le fichier monapp/urls.py devient : path("home", views.HomeView.as_view()),
Et voilà, notre fonction initiale def home(request) peut disparaître de notre fichier views.py

Nous aimerions bien nous servir d'un même template pour les vues about et contact. Cependant nous ne voulons pas dire «bonjour DJANGO !» dans ces deux vues. Cela est spécifique à la vue home. Il faut donc définir du contexte. La méthode get_context_data(self, **kwargs) de la classe TemplateView nous le permet :

```
class AboutView(TemplateView):
    template_name = "monapp/home.html"

    def get_context_data(self, **kwargs):
        context = super(AboutView, self).get_context_data(**kwargs)
        context['titreh1'] = "About us..."
        return context

    def post(self, request, **kwargs):
        return render(request, self.template_name)
```

Modifiez le fichier monapp/urls.py en conséquence.

Et le contenu du fichier home.html change quelque peu :

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

{% block contenu %}
<h1> {{ titreh1 }} </h1>
{ endblock %}
```

Pour que la vue home continue de fonctionner, il faut aussi apporter une petite modification à la HomeView, en rajoutant la fonction ci dessous, et le tour est joué :

```
def get_context_data(self, **kwargs):
        context = super(HomeView, self).get_context_data(**kwargs)
        context['titreh1'] = "Hello DJANGO"
        return context
```

C'est à vous de jouer !! En gardant le même template home.html, modifiez votre vue contact et la vue dont l'url est home/<param> ; Vous aurez besoin de ce bout de code self.kwargs.get('param'). A vous de mettre cette instruction au bon endroit.

Les vues génériques d'affichage

L'une des vues génériques d'affichage les plus utilisées est ListView. Cette vue a vocation à travailler sur une liste d'objets. C'est le même principe que précédemment. La fonction ListProducts( ) dans le fichier views.py va être remplacée par la classe ProductView qui va hériter de ListView. Il lui faut un modèle, un template et du contexte :

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

Et il ne faut oublier le nouveau path de l'url : path("product/list",views.ProductListView.as_view()),

Vous pouvez bien évidemment choisir l'emplacement et le nom de votre template, ainsi que le nom de l'objet passé dans le contexte.

Il est possible de modifier la queryset qui liste les produits :

```
class ProductListView(ListView):
    model = Product
    template_name = "monapp/list_products.html"
    context_object_name = "products"
    queryset = Product.objects.filter(id=2)
```

ou en passant par la méthode get_queryset :

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

    def get_queryset(self ) :
        return Product.objects.order_by("price_ttc")
```

Si vous voulez passer d'autres données, vous pouvez passer par la méthode get_context_data () dans la classe ProductView:

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

Il vous suffit de récupérer votre variable {{titremenu}} dans votre template !

DetailView est identique à ListView sauf qu'il s'agit que d'un seul item ; le path de l'url est donc :

```
path("product/<pk>",views.ProductDetailView.as_view()),
```

La classe ProductDetailView hérite de DetailView :

```
class ProductDetailView(DetailView):
    model = Product
    template_name = "monapp/detail_product.html"
    context_object_name = "product"

    def get_context_data(self, **kwargs):
        context = super(ProductDetailView, self).get_context_data(**kwargs)
        context['titremenu'] = "Détail produit"
        return context
```

Nous définissons alors un nouveau template detail_product.html :

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

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

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

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

<ul>
    <li>Code : {{ product.code }}</li>
    <li>Prix HT : {{ product.price_ht }}</li>
    <li>Prix TTC: {{ product.price_ttc }}</li>
    <li>Date de création : {{ product.date_creation }}</li>
</ul>
{% endblock %}
```

Et le tour est joué !

C'est bien que nous puissions accéder à la page du produit ipod en visitant http://127.0.0.1:8000/monapp/product/1/, mais attendons-nous vraiment de nos utilisateurs qu'ils sachent que «1» est l'identifiant du produit ipod et qu'ils le tapent dans la barre d'adresse ? Bien sûr que non. De nombreux utilisateurs ne regardent même pas la barre d'adresse lorsqu'ils naviguent sur un site. Le fait que certains navigateurs modernes comme Safari cachent désormais le chemin d'accès à l'URL dans la barre d'adresse en est la preuve !

La plupart des utilisateurs s'attendent implicitement à être guidés sur le site par des éléments de navigation intuitifs, comme les hyperliens. Mettons à jour le modèle de la vue en liste, afin que chaque nom de produit devienne un lien cliquable qui nous amène à la vue détaillée de ce groupe.

```
<a href="/product/{{prdct.id}}">{{ prdct.name }}</a>
```

Mais vous ne devriez pas faire ça ! C'est un anti-pattern.
Cela va générer un lien fonctionnel : n'hésitez pas à tester ! Mais c'est un anti-pattern. Pourquoi ? Parce que c'est du code répété : nous construisons déjà ce chemin d'une manière similaire dans urls.py :

```
path("product/<pk>",views.ProductDetailView.as_view()),
```

Si nous voulions changer le chemin d'accès à la vue détaillée, par exemple, en «tous_les_produits/1/ », nous devrions le modifier dans urls.py *et* dans chaque gabarit où nous répétons le lien. Comment éviter la répétition du code et rendre nos liens plus faciles à maintenir ? Commençons par donner au modèle d'URL un argument name:

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

J'ai utilisé « product-detail » avec un tiret pour le nom de la vue ici, mais le name peut être ce que vous voulez. Le modèle d'URL est maintenant notre «source unique de vérité» pour ce lien. Pour générer ce lien dans notre gabarit, nous allons utiliser une balise de gabarits :

```
<a href="{% url 'product-detail' prdct.id %}">{{ prdct.name }}</a>
```

Nous passons deux arguments à la balise url :

- product-detail
- prdct.id.

C'est comme passer des arguments à une fonction Python, mais dans les balises de gabarits, et nous n'avons pas besoin de saisir les parenthèses. Maintenant, si nous rafraîchissons notre liste de produits, nous pouvons voir que chaque nom de produit est maintenant un lien, et nous pouvons cliquer sur la vue détaillée du produit.

Faisons maintenant la même chose dans l'autre sens : la vue détaillée du groupe doit comporter un lien vers la liste des groupes. Nous modifions monapp/urls.py en rajoutant le name :

```
path("product/list",views.ProductListView.as_view(),name="product-list"),
```

Et le template detail-product.html reçoit une ligne supplémentaire :

```
<a href="{% url 'product-list' %}">Retour à la liste des produits</a>
```

Puisque nous travaillons avec les liens, profitons-en pour ajouter une barre de navigation à notre site, ainsi qu'un lien vers la liste des produits. Nous l'ajouterons dans le gabarit de base, en tant que premier élément de la balise <body>, car la barre de navigation doit apparaître en haut de chaque page :

```
<nav class="navbar navbar-expand-md navbar-dark bg-dark mb-4">
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav ml-auto">
            <li class="nav-item">
                <a class="nav-link" href="{% url 'product-list' %}">Produits</a>
            </li>
        </ul>
    </div>
</nav >
```

C'est à vous !!
Maintenant, c'est à votre tour de créer des vues en liste et détaillées pour le modèle ProductItem et ProductAttribute et d'ajouter des liens entre les deux. Ajoutez également un lien vers la vue en liste dans notre barre de navigation pour chacun d'eux.

Les vues génériques de connexion et déconnexion

Il existe plusieurs manières d'utiliser le login/logout de base de DJANGO. Nous verrons dans cette partie pas forcément la manière la plus facile à mettre en œuvre, mais la plus souple. Nous allons créer nos vues login et logout dans le fichier monapp/views.py. Nous allons hériter des vues génériques fournies par DJANGO, LoginView et LogoutView ! Commençons par la vue générique de connexion :

```
class ConnectView(LoginView):

	template_name = 'monapp/login.html'

	def post(self, request, **kwargs):
		username = request.POST.get('username', False)
		password = request.POST.get('password', False)
		user = authenticate(username=username, password=password)
		if user is not None and user.is_active:
			login(request, user)
			return render(request, 'monapp/hello.html',{'titreh1':"hello "+username+", you're connected"})
		else:
			return render(request, 'monapp/register.html')
```

Comme vous pouvez le constater, nous avons besoin d'un template monapp/login.html, le voici :

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

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

{% block contenu %}
<div class="align-center col-8 offset-2">
	<p class="lead mb-3 ml-0">Se connecter</p>
	<hr/>
	<form method="POST" action="{% url 'login' %}" autocomplete="off">
	{% csrf_token %}
		<div class="form-group mb-3">
			<label>Identifiant</label>
			<input class="form-control" type="text" name="username" placeholder="Utilisateur">
			<label>Mot de Passe</label>
			<input class="form-control" type="password" name="password" placeholder="Mot de passe">
		</div>
		<button type="submit" class="btn btn-success mt-2">Se connecter</button>
	</form>
</div>
{% endblock %}
```

Quelques explications s'imposent : lorsque nous arrivons sur la vue, le template associé nous propose un formulaire où l'utilisateur devra saisir son identifiant et son password. A la validation du formulaire (POST), la méthode post() de la classe ConnectView récupère les éléments du formulaire (identifiant et password). Si l'utilisateur est connu en BDD à l'aide de la méthode authenticate(), l'utilisateur est connecté avec la méthode login() et est redirigé sur la page de bienvenue. Sinon, il doit s'inscrire sur la page register.html dont voici le template :

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

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

{% block contenu %}
<div class="align-center col-8 offset-2">
        <p class="lead mb-3 ml-0">S'inscrire</p>
        <hr/>
        <form method="POST" action="{% url 'register' %}" autocomplete='off'>
        {% csrf_token %}
                <div class="form-group mb-3">
                        <label>Identifiant</label>
                        <input class="form-control" type="text" name="username" placeholder="Utilisateur">
                        <label>Mail</label>
                        <input class="form-control" type="mail" name="mail" placeholder="Email">
                        <label>Mot de Passe</label>
                        <input class="form-control" type="password" name="password" placeholder="Mot de passe">
                </div>
                <button type="submit" class="btn btn-success mt-2">S'inscrire</button>
        </form>
</div>
{% endblock %}
```

Même principe, un formulaire, trois champs à saisir et un clic sur le bouton « S'inscrire » qui nous envoie, via la méthode POST du formulaire et l'url portée par son action, vers la vue RegisterView :

```
class RegisterView(TemplateView):
    template_name = 'monapp/register.html'

    def post(self, request, **kwargs):
        username = request.POST.get('username', False)
        mail = request.POST.get('mail', False)
        password = request.POST.get('password', False)
        user = User.objects.create_user(username, mail, password)
        user.save()
        if user is not None and user.is_active:
            return render(request, 'monapp/login.html')
        else:
            return render(request, 'monapp/register.html')
```

La méthode post() de la classe RegisterView récupère les éléments du formulaire (identifiant, mail et password). L'utilisateur est créé en BDD à l'aide de la méthode User.objects.create_user(). Si la création se passe bien, l'utilisateur est redirigé vers la page login.html pour se connecter ou bien il reste sur place... Pour que tout cela se passe à merveille, vous aurez besoin des nouveaux éléments suivants dans le fichier monapp/urls.py

```
path('login/', views.ConnectView.as_view(), name='login'),
path('register/', views.RegisterView.as_view(), name='register')
path('logout/', views.DisconnectView.as_view(), name='logout'),
```

Il nous reste la vue DisconnectView qui hérite de TemplateView à implémenter dans views.py :

```
class DisconnectView(TemplateView):
    template_name = 'monapp/logout.html'

    def get(self, request, **kwargs):
        logout(request)
        return render(request, self.template_name)
```

Et son template associé tout simple logout.html :

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

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

{% block contenu %}
<div class="align-center col-8 offset-2">
        <p class="lead mb-3 ml-0">Se déconnecter</p>
        <hr/>
        <form method="POST" action="{% url 'home' %}" autocomplete="off">
                {% csrf_token %}
                <div class="form-group mb-3">
                        <label>Vous avez été déconnecté.</label>
                </div>
        </form>
</div>
{% endblock %}
```

Enfin, mettons à jour notre menu dans le fichier base.html avec ces éléments de navigation conditionnés au fait que l'utilisateur est connecté ou non. A vous de mettre ces lignes au bon endroit :

```
{% if user.is_authenticated %}
        <li class="nav-item">
        <a class="nav-link" href="{% url 'logout' %}">Déconnexion</a>
        </li>
{%else%}
        <li class="nav-item">
        <a class="nav-link" href="{% url 'login' %}">Connexion</a>
        </li>
        <li class="nav-item">
        <a class="nav-link" href="{% url 'register' %}">Inscription</a>
        </li>
{%endif%}
```

Au passage, vous aurez mis en œuvre deux formulaires. Les formulaires permettent d'envoyer des données du front-end au back-end. Nos formulaires sont un peu perdus dans la nature au beau milieu de nos templates. C'est là que DJANGO intervient !

Premier formulaire DJANGO

Nous allons apprendre par la suite comment créer un formulaire dans les règles de l'art DJANGO et comment les utiliser dans DJANGO. Les formulaires DJANGO vont nous présenter quelques nouveaux concepts. Nous allons donc apprendre à créer un formulaire de « Contact » qui permet aux utilisateurs d'envoyer un message aux administrateurs de l'application.

Nous commençons par définir le formulaire comme une classe. Créez le fichier « monapp/forms.py » et ajoutez ce code :

```
from django import forms

class ContactUsForm(forms.Form):
    name = forms.CharField(required=False)
    email = forms.EmailField()
    message = forms.CharField(max_length=1000)
```

Nous avons défini trois champs de formulaire dans notre ContactUsForm. Les champs de formulaire sont similaires aux champs de modèle : nous avons différents types de champs pour différents types de données. Nous pouvons également préciser quand les champs doivent être facultatifs avec required=False. Ici, nous permettons à l'utilisateur de rester anonyme en rendant le champ name facultatif. Et nous pouvons définir max_length, tout comme nous le faisons dans un modèle.

Ensuite, utilisons notre ContactUsForm dans la vue ContactView :

```
def ContactView(request):
    form = ContactUsForm()
    titreh1 = "Contact us !"
    return render(request, "monapp/hello.html",{'titreh1':titreh1, 'form':form})
```

Et maintenant nous éditons notre gabarit «contact.html » peut recevoir ce nouveau contexte :

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

{% block contenu %}
    <h1> {{ titreh1 }} </h1>
    <p>Nous sommes là pour vous aider.</p>

    <form action="" method="post" novalidate>
        {% csrf_token %}
        {{ form }}
        <button type="submit" class="btn btn-success mt-2">Envoyer</button>
    </form>

{% endblock %}
```

Dans notre gabarit, nous avons ajouté une balise <form> avec un <input> de type="submit" à l'intérieur. C'est la norme pour tout formulaire HTML que vous pouvez créer. Mais maintenant les choses deviennent intéressantes. Au lieu de taper manuellement les balises <input> pour chacun des champs de notre formulaire, nous tapons simplement {{ form }}. Nous ajoutons également {% csrf_token % }, une protection facile à mettre en place contre les attaques de type CROSS SITE. Ces types d'attaques consiste à injecter du code malicieux à travers le bouton d'un formulaire.

Jetons un coup d'œil à cela dans le navigateur à l'adresse http://127.0.0.1:8000/monapp/contact pour comprendre ce qui se passe ici, et examinons le HTML généré dans les outils de développement de notre navigateur. Nous pouvons voir que DJANGO a automatiquement généré un <label> et un <input> pour chacun de nos champs de formulaire : name, email et message. C'est génial : cela signifie que chaque fois que nous voulons ajouter un nouveau champ à ce formulaire, nous l'ajoutons simplement à la classe ContactUsForm, et DJANGO s'occupera du HTML pour nous. Une petite modification s'impose dans le gabarit : les champs seraient plus beaux s'ils étaient alignés :

{{ form.as_p }}.

Cela encadre chaque paire balise de champ dans une balise <p>
Regardez à nouveau la balise <form> que nous avons utilisée :

- novalidate désactive la validation de formulaire de votre navigateur. Il s'agit d'une fonctionnalité utile de la plupart des navigateurs, et nous la réactiverons plus tard, mais nous devons d'abord vérifier que notre formulaire fonctionne correctement sans elle.
- La valeur de method est post : ce qui signifie que les données seront envoyées comme une requête HTTP POST. C'est un peu différent des requêtes GET que nous avons utilisées jusqu'à présent, car en plus d'une «méthode» et d'un «chemin», elle comprend également un «corps» de requête qui contient les données du formulaire.
- L'attribut action désigne l'URL où nous allons envoyer les données du formulaire. Si vous donnez à cet attribut la valeur d'une chaîne vide, il renverra à l'URL de la page où nous nous trouvons déjà, c'est-à-dire "http://127.0.0.1:8000/monapp/contact". Cela signifie que nous allons gérer les données du formulaire dans notre vue contact.

Maintenant que nous savons que les données de notre formulaire arriveront au serveur sous forme de requête POST, voyons comment gérer les requêtes POST dans notre vue. Pour avoir un aperçu plus clair du fonctionnement des données POST dans une vue, nous allons utiliser un peu de journalisation dans le terminal, avec des instructions print :

```
def ContactView(request):
        form = ContactUsForm()
        titreh1 = "Contact us !"
        print('La méthode de requête est : ', request.method)
        print('Les données POST sont : ', request.POST)
        return render(request, "monapp/contact.html",{'titreh1':titreh1, 'form':form})
```

Tout d'abord, appelons cette vue comme une requête GET. Le moyen le plus sûr de le faire est de cliquer dans la barre d'adresse du navigateur et d'appuyer sur Entrée. Maintenant, regardez dans le terminal :

```
La méthode de requête est :  GET
Les données POST sont :  <QueryDict: {}>
```

Nous pouvons voir que pendant une requête GET, request.POST est un QueryDict vide (qui est un type spécial de dict Python). Maintenant, demandons la même vue en tant que requête POST. Pour ce faire, nous saisissons des données dans les champs du formulaire, puis nous cliquons sur «Envoyer». Cette fois, nous pouvons voir que notre QueryDict contient les données de notre formulaire ! (Y compris le mystérieux jeton CSRF).

```
La méthode de requête est :  POST
Les données POST sont :
<QueryDict:{
'csrfmiddlewaretoken':'gG1HEKfwmqlh9nTE8mbdg5X9VtcjzbmdWCXOgzDw71qVGLWyaN2DSnq8WjmNWWml'],
'name': ['titof'], '
email': ['titof@gmail.com'],
'message': ['bonjour']
}>
```

Notez qu'après que nous avons soumis les données et que la vue contact s'est rechargée, nous voyons à nouveau un formulaire vide, parce qu'actuellement notre vue crée simplement un nouveau formulaire vide à chaque fois qu'elle s'exécute (form = ContactUsForm()).

Ensuite, nous devons d'une manière ou d'une autre gérer les deux scénarios de demande dans notre vue :

- S'il s'agit d'une requête GET, nous devons afficher un formulaire vide à l'utilisateur.
- S'il s'agit d'une demande POST, nous devons examiner les données et voir si elles sont valides.

Nous avons donc besoin d'une instruction if:

```
def ContactView(request):
    titreh1 = "Contact us !"

    if request.method=='POST':
        form = ContactUsForm(request.POST)
    else:
        form = ContactUsForm()

    return render(request, "monapp/contact.html",{'titreh1':titreh1, 'form':form})
```

Dans les deux branches de l'instruction if, nous créons un formulaire qui est transmis au modèle, mais dans le cas d'une requête POST, nous remplissons également le formulaire avec les données POST. Maintenant, lorsque nous soumettons un formulaire rempli, les données sont toujours visibles dans le

navigateur lorsque la page est rechargée. Mais ce n'est pas tout : si nous soumettons des données non valides, le formulaire affichera des messages d'erreur.

Ce que nous voyons ici est une validation côté serveur : notre formulaire DJANGO a validé les champs par rapport à nos règles, a généré des messages d'erreur en cas de problème, puis les a retournés dans le modèle comme faisant partie du formulaire. L'utilisateur peut alors modifier les valeurs et soumettre à nouveau le formulaire, et DJANGO vérifiera à nouveau le formulaire. Ce cycle peut se répéter autant de fois que nécessaire jusqu'à ce que tous les champs du formulaire soient valides. A ce stade, nous sommes enfin prêts à effectuer l'action que nous voulions faire en premier lieu : envoyer un e-mail !

Nous avons maintenant un peu plus de logique à mettre en œuvre dans notre vue :

- Si c'est une requête POST...
  - Si les données du formulaire sont valides, envoyer un e-mail ;
  - Si les données du formulaire ne sont pas valides, afficher à nouveau le formulaire avec des messages d'erreur (comme nous le faisons déjà).

Nous avons besoin d'une instruction if imbriquée :

```
def ContactView(request):
    titreh1 = "Contact us !"
    if request.method=='POST':
        form = ContactUsForm(request.POST)
        if form.is_valid():
            send_mail(
            subject=f'Message from {form.cleaned_data["name"] or "anonyme"} via MonProjet
            Contact Us form',
            message=form.cleaned_data['message'],
            from_email=form.cleaned_data['email'],
            recipient_list=['admin@monprojet.com'],
            )
        else:
        form = ContactUsForm()
    return render(request, "monapp/contact.html",{'titreh1':titreh1, 'form':form})
```

Nous importons la fonction send_mail de DJANGO au début. Ensuite, nous insérons l'instruction if imbriquée, commençant par : if form.is_valid():
Si tous les champs de notre formulaire contiennent des données valides, alors form.is_valid() renvoie True et nous appelons send_mail pour envoyer notre e-mail.

form.cleaned_data est un dict contenant les données du formulaire après qu'elles ont subi le processus de validation. Lorsque nous sommes prêts à faire quelque chose avec les données de notre formulaire, nous pouvons accéder à chacun des champs via form.cleaned_data['name_of_field'], mais nous devons d'abord appeler form.is_valid().

L'envoi d'un véritable e-mail implique la configuration d'un serveur SMTP, que nous n'avons malheureusement pas le temps de couvrir ici ! Mais nous pouvons utiliser le serveur de messagerie fictif de DJANGO pour tester notre formulaire. Ceci affichera tous les e-mails envoyés par DJANGO dans le terminal. Ajoutez cette ligne au tout début du fichier monapp/settings.py :

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Maintenant nous pouvons soumettre notre formulaire et regarder le résultat apparaître dans le terminal :

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: Message from titof via MonProjet Contact Us form
From: titof@gmail.com
To: admin@monprojet.com
Date: Sat, 14 Sep 2024 13:23:50 -0000
Message-ID: <172632023027.66891.2783504844626258837@infoens-cd2>

bonjour
-------------------------------------------------------------------------------
[14/Sep/2024 15:23:50] "POST /monapp/contact HTTP/1.1" 200 16856
```

Nous avons presque terminé, mais il y a encore une chose à faire pour améliorer la convivialité de notre formulaire…

Une redirection est un type de réponse HTTP qui demande au navigateur de charger une nouvelle page. Les avantages de cette démarche sont doubles :

- Nous réduisons la probabilité d'un POST dupliqué ;
- Nous pouvons améliorer l'expérience de l'utilisateur en affichant une page de confirmation au lieu de simplement recharger la même page.

Pour implémenter la redirection, nous allons utiliser la fonction redirect, un raccourci pratique. Nous pouvons lui fournir un modèle d'URL avec des arguments ou directement un chemin d'URL.

Ajoutons cette déclaration d'importation `from django.shortcuts import redirect,`

et une autre ligne de code `return redirect('email_sent')` dans notre déclaration if telle que :

```
def ContactView(request):
    titreh1 = "Contact us !"

    if request.method=='POST':
        form = ContactUsForm(request.POST)
        if form.is_valid():
            send_mail(
            subject=f'Message from {form.cleaned_data["name"] or "anonyme"} via MonProjet
            Contact Us form',
            message=form.cleaned_data['message'],
            from_email=form.cleaned_data['email'],
            recipient_list=['admin@monprojet.com'],
            )
            return redirect('email-sent')
        else:
            form = ContactUsForm()

    return render(request, "monapp/contact.html",{'titreh1':titreh1, 'form':form})
```

Veillez à ajouter l'instruction de retour à l'intérieur du bloc if `form.is_valid()`, donc en retrait de l'instruction if et non au même niveau.

Lorsque le formulaire est valide et que le courriel a été envoyé, cette redirection guidera le navigateur de la page du formulaire vers une page de confirmation. C'est une page qui a un motif URL avec le nom email-sent. Bien entendu, cette page n'a pas encore été créée. Vous devriez maintenant être en mesure de créer vous-même un modèle d'URL, une vue et un gabarit pour cette page de confirmation, je vous laisse donc vous en charger !