

🔑 Objectifs de la feuille

- Les vues DJANGO
- Comprendre les templates
- Logique de programmation dans les templates
- Héritage de templates
- Mettre du style dans ses templates

Les vues : à quoi ça sert et comment ça marche ?

Petit focus sur les vues car nous avons jusqu'ici créer 3 ou 4 vues basiques sans vraiment creuser le sujet. Les vues sont la deuxième notion du concept MVC (Modèle Vue Contrôleur). Les vues créent un contexte, intègre un **template** (ou gabarit) puis retournent un objet de type **HttpResponse** . Tout d'abord votre serveur web reçoit une requête avec comme contenu: une adresse, un entête HTTP , divers données fournit par GET, POST, etc. Ensuite le serveur web demande à DJANGO ce qu'il doit faire avec cette requête. L'adresse est comparée dans une sorte de table de routage (**urls.py**) puis DJANGO exécute la fonction associée au pattern de l'adresse . Cette fonction est donc une vue, puisqu'elle reçoit un contexte et renvoie un objet **HttpResponse** . Reprenons les exemples de la semaine dernière :

- nous avions dans **monapp/urls.py** :

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

urlpatterns = [
    path("home", views.home, name="home"),
    path("contact", views.contact, name="contact"),
    path("about", views.about, name="about"),
    path("home/<param>", views.accueil, name='accueil'),
]
```

- nous avions dans **monapp/views.py** :

```
def home(request):
    return HttpResponse("<h1>Hello Django!</h1>")
def about(request):
    return HttpResponse("<h1>About us...</h1>")
def contact(request):
    return HttpResponse("<h1>Contact us!</h1>")
def accueil(request,param):
    return HttpResponse("<h1>Hello " + param + " ! You're connected</h1>")
```

Première chose que nous avons déjà remarqué, c'est que le vues dans notre fichier **views.py** ne sont en réalité que de simples fonctions qui récupèrent un objet **request** et qui retourne un objet **HttpReponse**.

En anglais **request** signifie "demande / sollicitation", ce sont donc des informations envoyées par un client et celui-ci attend une réponse en fonction des informations qu'il a envoyé. Qui a-t-il dans un objet **request** ? Pour cela, rien de plus simple, ajouter la fonction **dir** dans votre vue:

```
def home(request):
    print(dir(request))
    return HttpResponse("<h1>Hello Django!</h1>")
```

Le résultat est long...Vous voulez encore plus long ? Pour voir les données associées à l'objet **request**:

```
def home(request):
    print(request.__dict__)
    return HttpResponse("<h1>Hello Django!</h1>")
```


Je ne peux pas copier coller le résultat (il y en a trop). Je vous invite donc à effectuer cette manipulation sur votre machine. Mais vous verrez qu'il existe des données sur tout comme l' adresse IP du client , le navigateur utilisé, les données de session , le port utilisé, le langage , le csrftoken , les données GET et POST, etc.

Bon passons aux choses sérieuses. Comment lire les données GET et POST envoyées par le client ?

```
def home(request):
    string = request.GET['name']
    return HttpResponse("Bonjour %s!" % string)
```

Rendez vous à l'url [http://127.0.0.1:8000/monapp/home?name="cricri"](http://127.0.0.1:8000/monapp/home?name=)


Une réponse où aucune erreur n'est à signaler possède le code 200 , vous pouvez le vérifier sur votre navigateur en lisant l'entête de la réponse:

Nom	État
 home?name=%22cricri%22	200

Il est possible de retourner d'autres types de réponse, comme la fameuse 404 indiquant que le document est introuvable :

```
from django.http import HttpResponse, HttpResponseNotFound

def home(request):
    return HttpResponseNotFound("Erreur fichier introuvable")
```

Nom	État
 home?name=%22cricri%22	404

Il existe d'autres réponses HTTP:

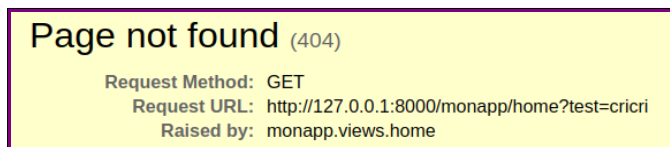
HttpResponse	→ 200
HttpResponseRedirect	→ 302
HttpResponsePermanentRedirect	→ 301
HttpResponseNotModified	→ 304
HttpResponseBadRequest	→ 400
HttpResponseNotFound	→ 404
HttpResponseForbidden	→ 403
HttpResponseNotAllowed	→ 405
HttpResponseGone	→ 410
HttpResponseServerError	→ 500

Si vous voulez relever une erreur 404 standard, vous pouvez lever une exception:

```
from django.http import HttpResponse, Http404

def home(request):
    if request.GET and request.GET["test"]:
        raise Http404
    return HttpResponse("Bonjour Monde!")
```

L'erreur suivante devrait apparaître si vous ajouter une variable GET test dans votre url tel que :
[http://127.0.0.1:8000/monapp/home?test="cricri"](http://127.0.0.1:8000/monapp/home?test='cricri')



Comme nous l'avons vu la semaine dernière, l'url passe par une sorte de table de routage. L'url peut contenir un paramètre voire un id ;

```
path("home/<param>",views.accueil,name='accueil'),
```

Vous savez comment récupérer ce paramètre, peut importe son nom (param, id, pk) du moment qu'il est renseigné dans la vue.

```
def accueil(request,param):
    return HttpResponse("<h1>Hello " + param + " ! You're connected</h1>")
```

Si vous travaillez en Ajax, il vous sera peut être utile de savoir retourner un JSON :

```
def ma_vue(request):
    return JsonResponse({'foo': 'bar'})
```

Comprendre les Templates

Nous allons commencer ce chapitre par la démonstration d'un anti-pattern. En programmation, un design pattern (patron de conception) est un style ou une technique qui représente la meilleure pratique et conduit à de bons résultats. À l'inverse, un anti-pattern représente une mauvaise pratique, quelque chose que nous devons éviter.

Pour les besoins de la démonstration, la semaine dernière, nous avons écrit dans `monapp/views.py` notre vue `ListProducts`

```
def ListProducts(request):
    prdcts = Product.objects.all()
    return HttpResponse(f"""
        <p>Mes produits sont :<p>
        <ul>
            <li>{prdcts[0].name}</li>
            <li>{prdcts[1].name}</li>
            <li>{prdcts[2].name}</li>
        </ul>
    """)
```

Nous pourrions pousser le vice jusqu'à ajouter les éléments structurels d'une page web, tels que `<head>`, `<body>`, `<title>` et la balise `<html>`. Ce code fonctionne et fonctionnerait... mais c'est un anti-pattern ! Depuis le début... Pourquoi, quel est le problème avec cette approche ? Notre vue commence déjà à avoir l'air un peu chargée et la quantité de HTML ici ne fera que s'accroître au fur et à mesure de la construction de notre application. Le problème est que notre vue a maintenant deux responsabilités :

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

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

Alors, comment transformer cet anti-pattern en design pattern, et utiliser les meilleures pratiques ? Nous allons adhérer au principe de la responsabilité unique en déplaçant la responsabilité de la présentation hors de la vue et en la plaçant à sa place légitime : un gabarit ou **template**.

Commencez par créer un nouveau fichier à l'adresse : `monapp/templates/monapp/list_products.html`. Notez la structure de répertoire que nous utilisons ici. Nous mettons toujours un sous-répertoire dans le répertoire des gabarits (**templates**) qui porte le même nom que l'application (**monapp**).

Vous pouvez aussi créer les templates à l'adresse `monprojet/templates/monapp/` sous réserve de préciser dans le fichier `settings.py` le répertoire `'DIRS': [BASE_DIR / 'templates']`, dans la variable `TEMPLATES`

Remplissons le fichier `list_products.html` avec notre HTML :

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    <h1>Hello mes produits !</h1>
    <p>Mes produits sont :</p>
    <!-- TODO : liste des produits -->
  </body>
</html>
```

Le code HTML n'est-il pas tellement plus beau dans un fichier `.html` que dans un fichier `.py` ? Vous bénéficiez d'une coloration syntaxique appropriée et d'une indentation automatique ! Mettons maintenant à jour notre vue `monapp/views.py`, de sorte qu'au lieu de définir son propre HTML, elle génère notre modèle à la place :

```
def ListProducts(request):
    prdcts = Product.objects.all()
    return render(request, 'monapp/list_products.html')
```

Tout d'abord, nous importons la fonction `render`. Cet élément est probablement déjà présent puisqu'il est inclus dans le code de base, mais ajoutez-le si nécessaire. Dans la déclaration de retour, nous n'appelons plus le constructeur `HttpResponse`. Au lieu de cela, nous appelons la fonction `render` avec 2 arguments :

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

Sous le capot, la fonction `render` crée un objet `HttpResponse` avec le HTML de notre modèle et le renvoie. Notre vue renvoie donc toujours une `HttpResponse` (ce qu'elle doit faire, pour être une vue). Jetez un coup d'œil à notre page dans le navigateur à l'url <http://127.0.0.1:8000/monapp/product/list>

C'est un bon début, mais les noms de nos produits n'apparaissent plus sur la page. Nous avons besoin d'un moyen d'injecter nos données de modèle dans notre gabarit. Il faut donc passer un objet contextuel au gabarit contenant une liste d'objets. Revenons à notre vue et ajoutons un troisième argument à notre appel de la méthode `render`. Cet argument doit être un dictionnaire python.

```
def ListProducts(request):
    prdcts = Product.objects.all()
    return render(request, 'monapp/list_products.html', {'premier_produit': prdcts[0]})
```

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

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    <h1>Hello mes produits !</h1>
    <p>Mes produits sont :</p>
    <!-- TODO : liste des produits -->
    <ul>
      <li>{{ premier_produit.name }}</li>
    </ul>
  </body>
</html>
```

Jetez un coup d'œil à la page dans notre navigateur pour voir cela en action. C'est quoi ces doubles accolades ? Ce n'est pas du HTML ! Bien vu ! Dans un code HTML valide, les gabarits de DJANGO peuvent inclure cette syntaxe avec des accolades, également connue sous le nom de langage de gabarits DJANGO. Chaque fois que vous voyez des doubles accolades contenant un nom de variable, la valeur de cette variable sera insérée. Elles sont appelées variables de gabarits. Nous pourrions continuer à ajouter chaque produit individuellement au dictionnaire contextuel, mais gagnons du temps et passons-les tous en une seule fois dans `monapp/views.py`:

```
def ListProducts(request):
    prdcts = Product.objects.all()
    return render(request, 'monapp/list_products.html',{'prdcts': prdcts})
```

Puis, dans notre template :

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    <h1>Hello mes produits !</h1>
    <p>Mes produits sont :</p>
    <!-- TODO : liste des produits -->
    <ul>
      <li>{{ prdcts.0.name }}</li>
      <li>{{ prdcts.1.name }}</li>
      <li>{{ prdcts.2.name }}</li>
    </ul>
  </body>
</html>
```

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

En résumé, les gabarits sont un moyen pour définir le contenu d'une page qui ne change pas. À l'intérieur de ces gabarits, nous insérons des variables de gabarits, qui servent d'espaces réservés pour le contenu qui change. Lorsque nous générons un gabarit dans une vue, nous passons un dictionnaire de contexte au gabarit et les variables de contexte sont injectées dans leurs espaces respectifs. En gardant la vue libre de tout code de présentation (HTML), nous pouvons limiter la responsabilité de la vue à une seule chose : la logique pour récupérer les données correctes de la base de données, et les injecter dans la page. Maintenant que vous avez compris les principes de base des gabarits, examinons certaines de leurs fonctionnalités plus utiles.

Logique de programmation dans les templates

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

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

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    <h1>Hello mes produits !</h1>
    <p>
      Mes produits sont :
      {% for prdct in prdcts %}
        {{ prdct.name }},
      {% endfor %}
    </p>
  </body>
</html>
```

Voici quelques éléments à prendre en compte :

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

Jetez un coup d'œil à notre page dans le navigateur.

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

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    <p>Mes produits sont :</p>
    <ul>
      {% for prdct in prdcts %}
        <li>{{ prdct.name }}</li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Remarquez que les balises d'ouverture et de fermeture `` sont en dehors de la boucle : nous voulons qu'elles n'apparaissent qu'une seule fois. Les balises `` sont à l'intérieur de la boucle, car nous avons besoin qu'elles soient répétées : une fois pour chaque groupe de la liste. Nous revoilà avec notre liste, mais maintenant nous la générons avec une boucle.

Parfois, lorsque nous injectons une variable dans notre gabarit, nous voudrions appliquer un certain formatage. Les filtres de gabarits peuvent nous y aider. Imaginons que nous voulions afficher les noms de nos produits dans une casse différente.

Nous pourrions utiliser le filtre `lower` (minuscules) ou `upper` (majuscules), comme ceci :

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

Vous appliquez un filtre à une variable en utilisant la barre verticale (|). Les filtres ne se limitent pas à la mise en forme du texte. Voyons comment nous pourrions afficher le nombre de groupes dans notre liste, en appliquant le filtre `length` à notre liste de groupes :

```
<p>J'ai {{ prdcts | length }} produits préférés.</p>
```

Nous pouvons également utiliser une logique d'embranchements dans nos gabarits. Dans notre gabarit, sous notre liste de groupes, ajoutons le code suivant :

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

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

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

Je veux que vous créiez des gabarits pour la vue **about**, la vue **contact**, la vue **ListItems** qui liste les déclinaisons de produits et la vue **ListAttributes** qui listent les attributs de produits. Vos gabarits doivent être des pages HTML complètes, comprenant les balises **<html>**, **<head>**, **<title>** et **<body>**. Pour les vue listings, vous devez injecter vos objets dans le gabarit.

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

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

Héritage de templates

Avez-vous remarqué à la fin du dernier chapitre que nous avons maintenant du code répété dans nos fichiers de gabarits ? Cela deviendra ingérable à mesure que nous ajouterons des gabarits à notre site : si nous voulons modifier la balise `head`, nous devons la modifier dans chaque gabarit.

Voyons comment nous pouvons appliquer le principe DRY : Don't Repeat Yourself (Ne vous répétez pas), et éviter la répétition du code dans nos gabarits en utilisant un modèle de base.

Pour commencer, nous devons examiner nos gabarits existants et nous poser deux questions :

- Quelles sont les différences entre ces gabarits ?
- Qu'est-ce qui ne change pas ?
-

Si nous regardons tous les gabarits côte à côte, nous pouvons voir que les éléments communs aux trois sont :

- Les balises `<html>`
- Les balises `<head>`
- Les balises `<title>`
- Les balises `<body>`

Comme ces balises sont communes à toutes les pages, nous allons les définir à un seul endroit : notre gabarit de base. Plaçons ce code HTML pour l'ensemble du site dans un nouveau fichier gabarit à l'adresse «[monapp/templates/monapp/base.html](#)» :

```
<html>
  <head>
    <title>Mon app DJANGO</title>
  </head>
  <body>
    {% block contenu %}{% endblock %}
  </body>
</html>
```

Dans notre HTML, nous avons ajouté une balise de gabarit Django : la balise **block**. Et nous l'avons fermé avec une balise **endblock**. Nous avons également donné un nom à ce bloc : nous l'avons appelé **contenu**. Nous pouvons considérer la balise **block** comme un espace réservé, dans lequel nous pouvons injecter du contenu ; en ce sens, il est similaire aux variables de gabarits.

Ce que nous voulons faire ensuite, c'est injecter notre HTML spécifique à la page dans ce bloc. Revenons à nos gabarits de page pour voir comment on fait.

Ouvrons notre gabarit «`list_products.html`». Pour commencer, supprimons tout le HTML que nous définissons maintenant dans notre gabarit de base. On devrait aboutir à quelque chose comme ça :

```
<h1>Hello les produits !</h1>
<p>
    J'ai
    {% if prdcts|length < 5 %}
        peu de
    {% elif prdcts|length < 10 %}
        quelques
    {% else %}
        beaucoup de
    {% endif %}
    produits préférés.
</p>
<p>Mes produits préférés sont :</p>
<ul>
    {% for prdct in prdcts %}
        <li>{{ prdct.name| upper }}</li>
    {% endfor %}
</ul>
```

Maintenant notre gabarit de page ne contient que le contenu spécifique à cette page. Ensuite, ajoutons ces balises de gabarits supplémentaires en haut et en bas du gabarit

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

{% block contenu %}

    <h1>Hello Django !</h1>
    ...
</ul>

{% endblock %}
```

La balise de gabarits **extends** en haut indique à DJANGO que nous voulons que ce gabarit hérite de notre gabarit de base. Autour de notre contenu, nous avons une balise d'ouverture et de fermeture **block** et nous avons donné à la balise d'ouverture le même nom que dans notre gabarit de base : **contenu**. Comme vous l'avez peut-être deviné, DJANGO va prendre tout ce qui se trouve dans le bloc nommé **contenu** dans notre gabarit de page et l'injecter dans le bloc du même nom dans notre gabarit de base.

Voyons maintenant comment cela fonctionne dans notre navigateur. Votre page devrait apparaître exactement comme elle était avant. Nous n'avons pas modifié l'apparence, mais nous avons changé la façon dont les choses fonctionnent sous le capot. Cela rendra notre application plus facile à gérer au fur et à mesure de son développement.

Nous avons vu deux nouvelles balises de gabarits dans ce chapitre : **extends** et **block**, qui nécessitent qu'on leur passe un argument. C'est un peu comme lorsque nous passons un argument à une fonction Python, comme `print('Hello World!')`. Les arguments des balises de gabarits n'ont pas besoin d'être mis entre parenthèses, ils doivent simplement être séparés par des espaces.

Parfois, l'argument doit être une chaîne de caractères placée entre guillemets (par exemple : `{% extends "monapp/base.html" %}`), et parfois non (par exemple : `{% block content %}`), veuillez donc à utiliser la syntaxe appropriée pour la balise en question.

Bien entendu, maintenant que nous avons défini notre gabarit de base, vous pouvez l'utiliser pour toutes les pages existantes (et futures) de notre site !! C'est à vous de jouer !!! Je ne veux plus voir de HTML dans les vues...

Donnez du style à votre site grâce aux CSS

La balise `<head>` de notre site étant définie dans notre gabarit de base, c'est le bon moment pour ajouter une feuille de style. Les fichiers tels que les feuilles de style CSS sont appelés fichiers statiques dans DJANGO, car une fois l'application en cours d'exécution, ils ne changent pas. Nous plaçons les fichiers statiques à un endroit spécifique de notre application. Créez un dossier dans `monprojet/static` et à l'intérieur, créez un nouveau fichier appelé `styles.css` :

```
h1{ color: red }
```

Il ne s'agit pas de la feuille de style finale que nous utiliserons réellement dans notre application, c'est plutôt le « Hello World ! » des feuilles de style. Cela va donner à tous les titres h1 de nos pages une couleur rouge. C'est juste pour tester que notre feuille de style se charge correctement ! Ouvrons le fichier `base.html` et ajoutons une balise `<link>` , sous le titre, pour charger notre feuille de style :

```
<html>
  <head>
    <title>Mon app DJANGO</title>
    <link rel="stylesheet" href="{% static 'styles.css' %}" />
  </head>
  <body>
    {% block contenu %}{% endblock %}
  </body>
</html>
```

Que remarquez-vous à propos de l'attribut `href` de notre balise `<link>` ?

Il s'agit en fait d'une autre balise de gabarits ! Cette balise `static` indique à DJANGO qu'il doit regarder dans le répertoire spécial `static` que nous avons créé précédemment et essayer de trouver le fichier `styles.css` que nous avons créé.

Maintenant, pour que la balise `static` fonctionne, nous devons d'abord la « charger » dans ce modèle. Nous le faisons en ajoutant une balise `load` au tout début du fichier, comme ceci :

```
{% load static %}
<html>
  <head>
    <title>Mon app DJANGO</title>
    <link rel="stylesheet" href="{% static 'styles.css' %}" />
  </head>
  <body>
    {% block contenu %}{% endblock %}
  </body>
</html>
```

Vous devriez voir vos titres en rouge. Cela signifie que tout fonctionne !

Si vous ne voyez pas de titre en rouge, essayez d'arrêter et de redémarrer le serveur de développement. Cela donnera à DJANGO une chance de trouver votre nouveau répertoire statique. Une fois que vous êtes sûr que votre fichier CSS se charge correctement, vous pouvez le modifier et commencer à donner à votre site le style que vous souhaitez. Mais nous pouvons mieux faire...

Installons dedans quelques librairies avec la commande **npm** ci-dessous qui installera dans le dossier **node_modules** les librairies demandées. Déplacez ensuite le répertoire **node_modules** dans **static**.

```
npm add bootstrap jquery-slim @popperjs/core
```

Vous allez pouvoir ainsi bénéficier de quelques fonctionnalités de Bootstrap notamment. Le fichier **base.html** devient par exemple

```
<!doctype html>
{% load static %}
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>{% block title %}{% endblock %}</title>

    {% block styles %}
    <link rel="stylesheet" href="{% static 'styles.css' %}" />
    <link rel="stylesheet" href="{% static 'node_modules/bootstrap/dist/css/bootstrap.min.css' %}">
    {% endblock %}
  </head>

  <body>
    {% block nav %}
    <nav class="navbar navbar-expand-md navbar-dark bg-primary mb-4">
      {% block menu %}{% endblock %}
    </nav>
    {% endblock %}

    <main role="main" class="container top-pushed">
      <div class="jumbotron">
        {% block contenu %}
        {% endblock %}
      </div>
    </main>

    <footer class="footer">
      <div class="container">
        <p>&copy; Copyright {% now "Y" %} by Titof Production Copyright (c)</p>
      </div>
    </footer>
  </body>
</html>
```

Notez les nouveaux blocs **title**, **styles**, **nav** et **menu** en plus de notre bloc **contenu** initial.

Notre fichier list_products.html change lui aussi un peu :

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

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

{% block menu %}
<h1>Liste des produits</h1>
{% endblock %}

{% block contenu %}
<table class="table">
  <thead>
    <th>Nom</th>
    <th>Code</th>
    <th>Prix HT</th>
    <th>Prix TTC</th>
    <th>Date Création</th>
  </thead>
  <tbody>
    {% for prdct in prdcts %}
    <tr>
      <td>{{ prdct.name }} </td>
      <td>{{ prdct.code }} </td>
      <td>{{ prdct.price_ht }} </td>
      <td>{{ prdct.price_ttc }} </td>
      <td>{{ prdct.date_creation }} </td>
    </tr>
    {% endfor %}
  </tbody>
</table>
<p>
J'ai
{% if prdcts|length < 5 %}
  peu de
{% elif prdcts|length < 10 %}
  quelques
{% else %}
  beaucoup de
{% endif %}
produits préférés.
</p>
{% endblock %}
```

Nous avons déjà une liste mieux présentée. Vous pouvez améliorer encore un peu la présentation en utilisant d'autres styles de bootstrap ...

Faites de même avec toutes vos listes (déclinaisons et attributs)