TP 3 : Technologies Web II [côté serveur]

Les vues (templates)

Objectifs du TP

Le but principal de ce TP est de découvrir la mise en place des vues (templates) dans Django permettant d'embellir et automatiser l'interface client.

Consigne

Lisez bien l'énoncé, il contient beaucoup d'informations et chaque mot est utile!
N'hésitez pas à chercher des informations sur les fonctions sur Internet, un bon informaticien doit savoir chercher des informations en ligne. Vous retrouverez notamment la documentation Django ici : `https://docs.djangoproject.com/en/4.1/`

Attention aux "copier-coller" du PDF vers votre code, des différences d'espace et d'indentation peuvent avoir lieu.
Il est donc préférable d'écrire vous même les commandes et le code demandés (c'est relativement court).

Introduction

Jusqu'à présent, nous avons vu deux éléments du patron Modèle-Vue-Contrôleur (MVC), les contrôleurs (appelées vues dans Django) et les modèles. Il nous reste donc à voir les vues (au sens MVC). Elles sont appelées **templates** dans Django.

Une vue est un « type » de page Web dans votre application Django qui sert généralement à une fonction précise et possède un gabarit spécifique. Par exemple, dans une application de blog, vous pouvez avoir les vues représentant : la page d'accueil du blog, la page de détail d'un poste, la page d'archive, etc.

Les vues permettent de définir l'interface utilisateur qui sera visible par les visiteurs du site Web (à l'inverse de l'interface administrateur vue au TP précédent).

Ce que contiennent les vues

Pas de logique!

Dans l'approche MVC, l'idée est que les vues (templates) sont un moyen de présenter des informations à l'utilisateur, et de recueillir ses actions. Il n'est donc pas question d'introduire de la logique métier dans les vues (templates).

Les vues doivent donc contenir :

— Essentiellement du code de présentation (c'est à dire du code HTML)
— Du code simple permettant d'exploiter les variables placées dans le contexte par le contrôleur (view) : itération sur des listes ou dictionnaires, conditions simples, etc.

Les vues ne doivent **pas** contenir :

— De la logique métier (du code permettant de réaliser des traitements compliquées liés au cœur de l'application)
— Des requêtes SQL
— Des accès directs aux paramètres de requête GET, POST, etc.

1 Le langage de templates de Django

Pour nous aider à afficher les données de manière plus structurée, Django fournit un système de "templating" intégré au framework. Ce moteur de "templating" permet de manipuler des variables au sein d'un contenu textuel. Pour cela un contrôleur retourne un **Template** à laquelle est associé un **Context** contenant les informations spécifiques à ajouter.

Pour se familiariser avec le langage de templates de Django, nous allons mettre en place un contrôleur particulier. Pour cela modifiez le fichier **views.py** d'une de vos applications (par exemple **polls**) en ajoutant une nouvelle fonction :

```
from django.http import HttpResponse

def test_template(request):
  return HttpResponse("Test du langage de template")
#...
```

Pour pouvoir appeler le comportement correspondant à l'aide d'une URL, modifiez le fichier **urls.py** :

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

urlpatterns = [
  path('', views.welcome, name='welcome-view'),
  path('hello/<firstname>', views.hello, name='hello-view'),
  path('test', views.test_template, name='test'),
]
```

Pour vérifier que tout est ok, démarrez le serveur et rendez-vous à l'url : `http://127.0.0.1:8000/polls/test`. Le texte "Test du langage de template" doit s'afficher.

1.1 Affichage d'une variable

Les variables sont représentées par des doubles accolades : `{{ ma_variable }}`. Nous allons créer un **Template** avec ce motif, puis nous allons envoyer une variable via un **Context** pour le retourner dans une **HttpResponse**. Modifiez la fonction **test** comme suit et observer le résultat (il faudra au préalable importer les modules **Template** et **Context**) :

```
from django.template import Template,Context

#...

def test_template(request):
  template = Template('Bonjour {{ name }} !')
  name = 'Maxime'
  context = Context({'name': name})
  return HttpResponse(template.render(context))
#...
```

1.2 Filtres

Le moteur de template de Django accepte un certain nombre de filtres lors de l'utilisation de variables. Ces filtres permettent d'appliquer des traitements de mise en forme aux variables, est sont invoqués à l'aide d'un pipe : `{{ ma_variable | filtre }}`. Par exemple le filtre **capfirst** permet de retourner le nom de variable avec la première lettre en majuscule. Modifier la fonction **test** comme suit et observez le résultat :

```
from django.http import HttpResponse
from django.template import Template,Context

def test_template(request):
```

```
  template = Template('Bonjour {{ name | capfirst }} !')
  name = 'maxime'
  context = Context({'name': name})
  return HttpResponse(template.render(context))
```

Plus d'information sur les filtres existants ici : `https://docs.djangoproject.com/en/4.1/ref/templates/language/#filters`.

1.3 Itération sur une liste

Le langage de template de Django fournit une boucle for simple et proche de celle proposée en Python pour parcourir les éléments d'une liste (ou d'un dictionnaire). Par exemple, pour parcourir et afficher les élément d'une liste **persons** valant **["Maxime", "Malika", "Nour"]** :

```
def test_template(request):
  template = Template('<ul>{% for person in persons %} <li>{{person}}</li> {% endfor %
                                        } </ul>')
  persons = ['Maxime', 'Malika', 'Nour']
  context = Context({'persons': persons})
  return HttpResponse(template.render(context))
```

1.4 Condition

Une structure conditionnelle simple est également proposée par le langage de template de Django. Par exemple, imaginons que nous souhaitons nous assurer qu'une liste contient bien des valeurs avant de l'afficher :

```
def test_template(request):
  template = Template('{% if persons %} <ul>{% for person in persons %} <li>{{person}}
                                        </li> {% endfor %} </ul> {% else %} La
                                        liste est vide. {% endif %}')
  persons = ['Maxime', 'Malika', 'Nour']
  context = Context({'persons': persons})
  return HttpResponse(template.render(context))
```

Observez le résultat avec une liste vide et une liste non-vide.

2 Revenons à notre application de sondage

Maintenant que nous avons vu le langage de templates dans Django, explorons comment cela peut être utilisé pour nos applications Web. En particulier, reprenons notre application de sondage (**polls**). Pour cette application, il sera intéressant d'avoir les quatre vues (templates) suivantes :

- La page de sommaire des questions qui affiche les questions.
- La page de détail d'une question qui affiche le texte d'une question, sans les résultats mais avec un formulaire pour voter.
- La page des résultats d'une question qui affiche les résultats d'une question particulière.
- Action de vote qui gère le vote pour un choix particulier dans une question précise.

2.1 Vues supplémentaires pour afficher les informations nécessaires

Pour comprendre l'intérêt des templates, nous allons tout d'abord ajouter des contrôleurs (vues en Django) supplémentaires pour afficher des informations liées aux éléments du sondage, sans passer par des templates. De manière similaire à ce qui a été fait pour le jeu du nombre mystère, nous pouvons ajouter de nouvelles vues (contrôleurs) dans le fichier **polls/views.py** :

```
def detail(request, question_id):
    return HttpResponse("Voici la question %s." % question_id)

def results(request, question_id):
    response = "Voici les resultats de la question %s."
    return HttpResponse(response % question_id)

def vote(request, question_id):
    return HttpResponse("Vous votez a la question question %s." % question_id)
```

Pour observer le comportement de ces vues, il est nécessaire de les lier avec leurs URL dans le fichier **polls/urls.py** :

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

urlpatterns = [
  path('', views.welcome, name='welcome-view'),
  path('hello/<firstname>', views.hello, name='hello-view'),
  path('test', views.test_template, name='test'),
  path('<int:question_id>/', views.detail, name='detail'),
  path('<int:question_id>/results/', views.results, name='results'),
  path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Essayez les urls suivantes et observez le résultat :

— `http://127.0.0.1:8000/polls/2/`
— `http://127.0.0.1:8000/polls/2/results/`
— `http://127.0.0.1:8000/polls/2/vote/`

2.2 Amélioration des vues

Dans Django, les contrôleurs (vues) sont responsables de retourner un objet de type **HttpResponse** (ou bien lever une exception, comme nous verrons plus tard). C'est à nous de définir le contenu des **HttpResponse** qui définiront le contenu de la page demandée par le client. Par exemple, il peut être intéressant d'utiliser l'API de base de données pour y récupérer des informations (comme nous l'avons fait pour le jeu du nombre mystère).

1. Modifiez la fonction **welcome** du fichier **views.py** (ou réécrivez-la si vous l'avez supprimée) comme suit :

```
#...
from .models import Question

#...
def welcome(request):
  latest_question_list = Question.objects.order_by('-publication_date')[:3]
  output = ', '.join([q.question_text for q in latest_question_list])
  return HttpResponse(output)
#...
```

Cela affiches les 3 dernières questions (s'il y en a au moins 3), séparées par des virgules et classées par ordre de publication. Vérifiez l'affichage en vous rendant à l'url : `http://127.0.0.1:8000/polls/`

2. De la même manière, modifiez la vue **detail** pour qu'elle affiche en plus de l'id, l'intitulé de la question : "Voici la question 2 intitulée Participerez-vous à la nuit de l'Info cette année 2022?"

Le problème avec ces deux exemples est que nous avons codé l'allure du résultat final en dur dans la vue (contrôleur). Si on souhaite modifier le style, il faut alors le modifier dans le code Python, ce qui n'est pas très souhaitable. Nous allons donc utiliser le système de templates comme vu précédemment mais légèrement différemment pour bien faire la séparation entre style de la page et code Python.

2.3 Création de templates (ou gabarits)

Pour faciliter l'organisation du projet, commencez par créer un dossier nommé **templates** dans le répertoire de votre application de sondages. Par convention, c'est ici que Django ira chercher les gabarits. Une bonne pratique consiste à créer dans **templates** un sous répertoire du nom de l'application, ici **polls**. Dans **templates/polls/**, créer une première template nommée **index.html**.

Pourquoi créer un sous-répertoire du nom de l'application? Et pourquoi pas juste le dossier **templates**? Django va chercher les templates dans toutes les applications installées du projet (donc dans les dossiers **templates** de chaque application. Il récupère le premier template qui correspond à la demande. Ainsi, si plusieurs applications utilisent les mêmes noms de templates (par exemple **index.html**, il peut donc y avoir confusion. C'est pourquoi des sous-répertoires sont utilisés.

1. Dans le fichier **index.html**, définissez le template comme suit :

```
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a>
                                                </li>
    {% endfor %}
    </ul>
{% else %}
    <p>Pas de question disponible.</p>
{% endif %}
```

Pour prendre en considération ce template, il faut l'indiquer dans la vue (contrôleur) correspondante du fichier **views.py**. la méthode **loader** est utilisée pour charger le template :

```
from django.http import HttpResponse
#...
from .models import Question
from django.template import loader

#...
def welcome(request):
  latest_question_list = Question.objects.order_by('-publication_date')[:3]
  template = loader.get_template('polls/index.html')
  context = {
        'latest_question_list': latest_question_list,
    }
  return HttpResponse(template.render(context, request))
#...
```

La différence majeure, avec ce que nous avons vu plus haut en introduction sur le langage des templates, est que le template est chargé depuis un fichier extérieur. Cela est beaucoup plus simple pour écrire les templates et pour séparer le style du code Python.

Rendez-vous à l'url suivante suivante pour observer le résultat : `http://127.0.0.1:8000/polls/`. Beaucoup mieux n'est-ce pas?

2. Créez un nouveau template **detail.html** qui sera dédié à la vue correspondante (**detail()**) pour afficher une question et les choix possibles comme ci-dessous. A vous de jouer!

   N'hésitez pas à ajouter des données (questions et choix) dans votre base de données (à l'aide de l'interface d'administration ou du shell).

   En vous rendant à l'url `http://127.0.0.1:8000/polls/2/`, l'objectif est d'obtenir quelque chose comme ceci :

2.4 Le raccourci render()

Comme nous avons vu auparavant, la mise en place de vues (templates) consiste à :

1. Charger un template
2. Remplir un contexte
3. Retourner un objet HttpResponse avec le résultat du template interprété dans le contexte

Django fournit un raccourci pour ces opérations : **render()** importé de `django.shortcuts`. Voici un exemple pour la vue **welcome()** :

```
#...
from .models import Question
#...
from django.shortcuts import render
#...
def welcome(request):
  latest_question_list = Question.objects.order_by('-publication_date')[:3]
  context = {'latest_question_list': latest_question_list}
  return render(request, 'polls/index.html', context)
#...
```

1. Utilisez ce raccourci et modifiez la vue **detail()** de la même manière.

2.5 Héritage de templates

Les templates que nous avons créés jusqu'à présent permettent de retourner des bouts de code HTML. Cependant, comme vous l'avez probablement vu dans le cours d'HTML, il est fortement conseillé d'utiliser des documents HTML complets contenant :

— La balise `<!DOCTYPE html>` pour indiquer qu'il s'agit d'un code HTML.
— Les balises `<html></html>` englobant tout le code HTML.
— Les balises `<head></head>` englobant le header de la page HTML.
— La balise `<meta>` dans le header permettant d'indiquer des métadonnées comme par exemple l'encodage utilisé : `charset="utf-8"`.
— Les balises `<title></title>` pour définir le titre de la page HTML.
— Les balises `<body></body>` englobant le corp de la page HTML.

Lors de la création de nos pages Web, il est donc préférable d'ajouter tous ces éléments.

Quoi? Nous allons devoir ajouter toutes ces informations dans chacun de nos templates?

Heureusement, non (même si pour l'instant, nous n'avons que 2 templates). Django permet l'héritage de templates. En plus de rendre les pages HTML plus facilement conformes, cela permet aussi de les uniformiser. En effet, certaines informations seront toujours les mêmes pour tous les templates. Il est ainsi possible de définir des parties identiques pour chaque template et des partie dynamiques qui seront différentes selon les templates. Les parties identiques (statiques) sont stockées dans un fichier HTML séparé. Les parties dynamiques sont les templates déjà créés.

Voyons un exemples de parties identiques pour nos pages HTML. Créez un fichier nommé **baseLayout.html** avec le contenu suivant :

```
<!DOCTYPE html>
<html lang="fr" >
<head>
   <meta charset="utf-8" >
   <title>
      {% block title %} {% endblock %}
   </title>
</head>
<body>
   <header>
      <h2>Application de sondages</h2>
   </header>
   <section>
      {% block content %} {% endblock %}
   </section>
   <footer>
      Copyright ENSISA IR 1A
   </footer>
</body>
</html>
```

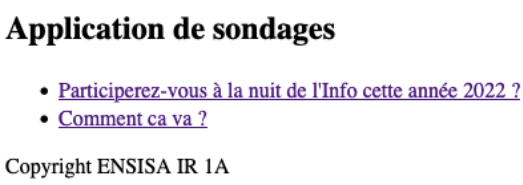
Ce fichier HTML représente la structure générale qu'auront toutes les pages HTML. Si vous regardez bien le code, on retrouve à deux endroits les éléments `{% block title %}` et `{% block content %}`. Ces éléments permettent d'ajouter du contenu dynamique à ces endroits par l'intermédiaire des templates spécifiques héritant de **baseLayout.html**.

Par exemple, pour faire en sorte que le template **index.html** hérite de **baseLayout.html**, il est nécessaire de l'indiquer dans le template ainsi que de définir le contenu des deux blocs `{% block title %}` et `{% block content %}`. Modifiez le contenu du fichier **index.html** comme suit :

```
{% extends "polls/baseLayout.html" %}
{% block title %}Liste des questions{% endblock %}
{% block content %}
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>Pas de question disponible.</p>
{% endif %}
{% endblock %}
```

Par défaut, lors d'un héritage de templates, Django va chercher la template de base dans le dossier **templates/** et pas **templates/polls/**. Il est possible de modifier cela dans les configuration mais le plus simple est d'indiquer le chemin restant vers le fichier de template dans la commande **extends** du template ("polls/baseLayout.html").

Redémarrez le serveur et rendez-vous à l'URL `http://127.0.0.1:8000/polls/` pour observer le résultat suivant :



1. Utilisez l'héritage pour également modifier le template **detail.html**.

2.5.1 Template générique

Pour information, il n'est pas rare qu'un template puisse être réutilisé non pas par d'autres vue d'une application mais pour plusieurs applications du même projet. Pour cela, la pratique consiste à mettre ce template dans un dossier **templates** à la racine du projet. Cependant, par défaut, Django ne cherche pas les templates dans ce dossier à la racine du projet. Pour l'autoriser à le faire, il faut modifier le fichier de configuration **settings.py**, en particulier la clé `DIRS` du dictionnaire `TEMPLATE`, comme ceci :

```
#...
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        #'DIRS': [],
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        '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',
            ],
        },
    },
]
#...
```

Cela pourrait vous être utile pour la suite.

2.6 Les erreurs 404

Comme dit plus haut, les vues (contrôleurs) sont responsables de retourner une réponse HTTP ou une exception. Une des exceptions les plus connues (que vous avez probablement déjà expérimentée) est l'erreur 404 indiquant que l'information demandé n'a pas été trouvée.

Dans notre exemple, cela peut par exemple être le cas pour la vue **detail()** appelée avec un id de question qui n'existe pas. Il n'est alors pas possible d'afficher la question demandée. Pour gérer cela en Django, on passe par un **try except** de Python. On teste si l'id est présent, si oui on affiche la vue correspondante, si non on lève une exception 404. Pour mettre en place ce procédé, modifiez la vue **detail** comme ceci :

```
#...
from .models import Question
#...
from django.http import Http404
#...
def detail(request, question_id):
  try:
    q = Question.objects.get(id=question_id)
  except Question.DoesNotExist:
    raise Http404("Question does not exist")
  return render(request, 'polls/detail.html', {'question': q})
#...
```

Rendez-vous aux URL `http://127.0.0.1:8000/polls/2/` et `http://127.0.0.1:8000/polls/42/` et observez les deux résultats différents.

Sachez pour information qu'il est possible de personnaliser les pages d'erreur. Plus d'informations ici : `https://docs.djangoproject.com/fr/4.1/topics/http/views/#customizing-error-views`.

2.6.1 Le raccourci get_object_or_404()

Comme pour **render()**, Django dispose également d'un raccourci pour l'erreur 404. En effet, l'opération **get()** effectuée précédemment dans le **try** et le **except** si l'opération **get()** n'aboutit pas, est très

courante pour gérer les éventuelles erreurs 404. Pour faciliter le travail des développeurs, Django a mis en place un raccourci **get_object_or_404()**. Ainsi, la vue **detail()** devient avec ce nouveau raccourci :

```
#...
from .models import Question
#...
from django.shortcuts import render, get_object_or_404
#...
def detail(request, question_id):
  question = get_object_or_404(Question, id=question_id)
  return render(request, 'polls/detail.html', {'question': question})
#...
```

Testez à nouveau les deux URL pour vérifiez que votre code fonctionne correctement.

2.7 Suppression des URL codés en dur dans les gabarits

Reprenez à nouveau le template **index.html** et regardez en particulier cette ligne :

```
<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>
```

Que remarquez-vous ?

Par exemple pour la question 2, le lien de la question est `/polls/2/`. C'est exactement le schéma d'URL pour appeler la vue **detail()** et afficher la question. Cependant, le problème ici est que le nom de l'application (poll) est utilisé "en dur" dans le template pour définir l'URL. Si l'URL doit être changée, cela peut vite devenir fastidieux pour les applications avec beaucoup de templates. Il faudrait alors modifier le contenu du template pour mettre le bon chemin d'URL. C'est loi d'être l'idéal.

Pour améliorer cela, nous pouvons remplacer le début de l'URL dans le template et utiliser la balise de template `{% url %}`, suivie du nom du motif de l'URL en question, comme ceci :

```
<li><a href="{% url 'detail' question.id %}">{{ question.question_text }}</a></li>
```

Mais comment le template sait que cette variable `'detail'` correspond au motif d'URL que l'on souhaite ?

Parce que nous avons utilisé le paramètre `name='detail'` dans le fichier d'URLconf **polls/urls.py** :

```
#...
path('<int:question_id>/', views.detail, name='detail'),
#...
```

Grâce à cela, le motif d'URL est nommé `'detail'` et peut être retrouvé par Django lorsque la variable `{% url %}` est utilisée dans le template.

1. Faites les modifications et testez pour vous assurer que cela ne change pas le comportement souhaité.

Imaginez maintenant que vous souhaitez modifier l'URL pour afficher le détail d'une question en `polls/questiondetail/2`. Vous aurez alors uniquement à le modifier à un seul endroit, le fichier **polls/url.py** (sans avoir à toucher au template) :

```
#...
path('questiondetail/<int:question_id>/', views.detail, name='detail'),
#...
```

2. Faites la modification et vérifiez à l'URL `http://127.0.0.1:8000/polls/` que le clique sur la question renvoie bien à l'URL `polls/questiondetail/2`.

2.8 Espaces de noms et noms d'URL

Comme nous venons de le voir, en enlevant l'URL "en dur" du template, il est maintenant facile de modifier les URL sans modifier le template.

Mais attendez, en enlevant la dépendance à l'application polls dans le template, comment Django sait que c'est la vue **detail** de l'application polls qu'il faut appeler?

En effet, il est tout à fait possible qu'une autre application possède également le même nom de vue. Pour spécifier l'application (et donc la vue correspondante) dans le template, on va utiliser la balise `{% url %}`. Il est d'abord nécessaire de modifier légèrement le fichier **urls.py** en indiquant un nom à l'application :

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

app_name = 'polls'
urlpatterns = [
  path('', views.welcome, name='welcome-view'),
  path('hello/<firstname>', views.hello, name='hello-view'),
  path('test', views.test_template, name='test'),
  path('questiondetail/<int:question_id>/', views.detail, name='detail'),
  path('<int:question_id>/results/', views.results, name='results'),
  path('<int:question_id>/vote/', views.vote, name='vote'),
]
```

Ensuite, nous pouvons modifier le template et plus particulièrement la ligne de lien qui nous intéresse comme ceci :

```
<li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
```

De cette manière, nous spécifions que la vue **detail** qui doit être appelée est celle de l'application **polls**.

1. Faites la modification et vérifiez que cela fonctionne correctement.

2.9 Les ressources statiques

En plus du code HTML généré par le serveur, les applications Web doivent généralement servir des fichiers supplémentaires tels que des images, du JavaScript ou du CSS, utiles pour produire une page Web complète. Django appelle ces fichiers des « fichiers statiques ».

Ainsi, lors du déploiement de votre projet, Django collecte l'ensemble des fichiers statiques de toutes les applications pour les mettre à un seul endroit qui peut facilement être configuré pour servir les fichiers en production.

Django utilise le même principe que pour les templates pour aller chercher les informations statiques.

1. Ainsi il est nécessaire de créer un dossier **static** dans le répertoire de votre application polls.
2. Créez ensuite un nouveau répertoire **polls** (du nom de l'application) à l'intérieur du dossier **static**

On crée un sous-dossier du nom de l'application pour la même raison que pour les templates.

2.9.1 Ajout d'un CSS

Pour ajouter du code CSS, il faut d'abord créer un fichier **style.css** à l'emplacement que vous venez de créer (`polls/static/polls/`). A l'intérieur, écrivez le code suivant :

```
li a {
    color: green;
}
```

Ensuite, modifiez le début de votre template **base_layout.html** comme suit :

```
<!DOCTYPE html>
{% load static %}
<html lang="fr" >
<head>
   <meta charset="utf-8" >
   <title>
      {% block title %}{% endblock %}
   </title>
   {% block styles %}
   <link rel="stylesheet" href="{% static 'polls/style.css' %}">
   {% endblock %}
</head>
<body>
   <header>
      <h2>Application de sondages</h2>
   </header>
   <section>
      {% block content %}{% endblock %}
   </section>
   <footer>
      Copyright ENSISA IR 1A
   </footer>
</body>
</html>
```

Regardons les modifications apportées :

— Les balises `{% load static %}` indiquent à Django qu'il va falloir charger les informations statiques. Django génère alors l'URL absolue des fichiers statiques trouvés.

— Les balises `{% block styles %}{% endblock %}` sont une bonne pratique qui suivent l'idée de l'héritage de templates pour permettre à d'autres templates d'ajouter d'éventuelles feuilles de styles supplémentaires dans ce bloc.

— `{% static 'polls/style.css' %}` permet dans l'attribut href de la balise link permet de combiner l'URL des informations statiques connue par Django avec le chemin restant vers la feuille de style souhaitée.

1. Faites les modifications et observez les changements à l'URL : `http://127.0.0.1:8000/polls/`.

2.9.2 Ajout d'une image

Les images sont aussi considérées comme des ressources statiques. Pour ajouter une image dans une page HTML, on peut modifier la template **index.html** en ajoutant une balise <img> comme ceci :

```
{% extends "polls/baseLayout.html" %}
{% load static %}
{% block title %}Liste des questions{% endblock %}
{% block content %}
{% if latest_question_list %}
    <ul>
    {% for question in latest_question_list %}
        <li><a href="{% url 'polls:detail' question.id %}">{{ question.question_text }}</a></li>
        <!--<li><a href="/polls/{{ question.id }}/">{{ question.question_text }}</a></li>-->
    {% endfor %}
    </ul>
{% else %}
    <p>Pas de question disponible.</p>
{% endif %}
<img src="{% static 'polls/django-python.png' %}" alt="My image">
{% endblock %}
```

C'est la même idée que pour le fichier CSS, la balise `{% static 'polls/django-python.png' %}"` permet de combiner les chemins pour aller chercher l'image correspondante.

1. Ajouter une image sur votre page d'index en suivant cette procédure

2.9.3 Ajout d'un script javascript

Les fichiers javascript sont également des ressources statiques. Leur inclusion dans les pages HTML dans Django suit la même procédure. Créez un fichier javascript **myscript.js** avec le contenu suivant :

```
alert('hello world!');
```

Ensuite modifiez le template **base_layout.html** en ajoutant ces lignes dans le header :

```
{% block js %}
<script type="text/javascript" src="{% static 'polls/myscript.js' %}"></script>
{% endblock %}
```

1. Testez votre code en relançant votre serveur et en vous rendant sur une page, vous devriez voir apparaître la fenêtre de dialogue javascript.

De manière générale (comme dans votre projet), n'hésitez pas à utiliser du CSS et du javascript plus complet pour obtenir des pages Web plus jolies. Pour information, le framework Bootstrap est notamment très puissant pour faire des choses jolies de manière (relativement) simple.

3 Et pour les autres vues de l'application de sondage ?

Dans notre application de sondage, nous avons défini 4 vues principales (**welcome**, **detail**, **results** et **vote**). Pour le moment nous nous sommes concentrés sur les deux premières. Nous verrons comment gérer la vue **vote** dans le prochain TP. Il nous reste alors la vue **results** à faire :

1. Reprenez ce que nous avons vu dans ce TP pour l'appliquer sur la vue **results** qui affiche les résultats pour une question (nombre de votes par choix).
2. Les résultats d'une question pourront être accessibles par un lien proposé depuis la page qui affiche le détail de la question (Ex : "Voir les résultats")
3. N'hésitez pas à modifier la base de données avec plus de contenu et de votes pour que l'affichage soit plus intéressant (via l'interface administrateur ou le shell)

4 Et pour l'application de gestion de projet ?

Reprenez tout ce que nous avons vu sur les vues (templates) pur l'appliquer à l'application de gestion de projets. Vous êtes libre d'organiser comme vous le souhaitez votre application.