esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

Django
Views

UP Web

AU: 2021/2022

United Nations Educational, Scientific and Cultural Organization
UNESCO Chair "Project-based learning" ESPRIT School of engineering, Tunisia
EUR-ACE® Délivrée par la Commission des Titres d'Ingénieur
CONFÉRENCE DES GRANDES ÉCOLES
cdio CONCEIVE DESIGN IMPLEMENT OPERATE

La couche « View »

- Une « **Vue** » est une fonction spécifique qui accepte une requête et renvoie une réponse. Cette réponse peut être sous la forme d'une redirection, une erreur 404, un contenu HTML présenté selon un « **template** ».

- Par convention, le code relatif aux vues est placé dans un fichier nommé **views.py**.

Simple HttpResponse

- Ci-après un exemple des vues, les méthodes correspondantes sont implémentées dans views.py.

Importer la classe **HttpResponse**

Chaque fonction de vue renvoie un objet **HttpResponse**

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("You're looking to the index page.")

def list_books (request):
    return HttpResponse("You're looking to the list books page")

def publisher_details(request, publisher_id):
    response = "You're looking at the details of publisher %s."
    return HttpResponse(response % publisher_id)
```

Implémenter un traitement

- Il est possible de permettre aux vues de rendre des données concrètes. Par exemple, pour rendre la liste des livres:

```
def list_books (request):
      books_list = Book.objects.all()
      output = ', '.join([b.title for b in books_list])
      return HttpResponse(output)
```

Renvoi d'erreurs

- Il est possible de signaler une erreur. Django fournit des sous-classes de **HttpResponse**.
- Par exemple:

```
def book_detail (request, book_id):
        try:
                book = Book.objects.get(pk=book_id)
        except Book.DoesNotExist:
                return HttpResponseNotFound('<h1>Page not
        found</h1>')
        return HttpResponse(book)
```

- Il est aussi possible de transmettre le code de statut HTTP

```
return HttpResponse(status = <statut HTTP>)
return HttpResponse(status = 404)
```

Renvoi d'erreurs

- Il est possible de signaler une erreur en utilisant l'exception Http404.
- Django intercepte l'exception et renvoie une page d'erreur.
- Par exemple:

```
def book_detail(request, book_id):
        try:
                book = Book.objects.get(pk=book_id)
        except Book.DoesNotExist:
                raise Http404("Book does not exist")
        return HttpResponse(book)
```

Présenter les données

- Pour renvoyer un gabarit (template), Django fournit une fonction « **render** ».
- Il faut donc:
  - Charger les données
  - Remplir le context
  - Renvoyer un objet HttpResponse

```
from django.shortcuts import render

def list_books(request):
    books_list = Book.objects.all()
    context = {
        'books_list': books_list,
    }
    return render(request, 'app/list_books.html',
          context)
```

Présenter les données

- La fonction « **render** » prend comme paramètre:
  - L'objet requête
  - Le nom du gabarit
  - Le contexte comme dictionnaire (facultatif)

- Par exemple

```
return render(request, 'app/list_books.html',  context)
```

Présenter les données

- Pour spécifier à Django la manière avec laquelle on veut présenter nos données, il faut utiliser le système de gabarits « templates ».

- Pour ce faire:
  - Créer un répertoire nommé « templates » dans le dossier de l'application (par exemple: app/templates/app/).
  - Créer les sous templates sous forme de fichiers HTML (par exemple: list_books.html).
  - Définir le contenu du HTML,

Présenter les données

- Par exemple:

```
<h1>Books List</h1>
{% if books_list %}
    <ul>
    {% for b in books_list %}
        <li>{{ b.title }}</li>
    {% endfor %}
    </ul>
{% else %} {# No Books #}
    <p> Books unavailable.</p>
{% endif %}
```

Gabarits

- Un gabarit contient deux parties; une statique avec le résultat HTML et une dynamique contenant le résultat du contexte.
- Le moteur de template par défaut de Django est **DTL** mais il prend aussi en charge l'alternative **Jinja2**.
- Les templates sont configurés dans le fichier **settings.py**

```
TEMPLATES = [
 {
   'BACKEND': 'django.template.backends.django.DjangoTemplates',
   'DIRS': [],
   'APP_DIRS': True,
   'OPTIONS': {  },
 },
]
```


https://docs.djangoproject.com/fr/3.1/ref/templates/language/

Gabarits

- Les variables sont entourées par {{ et }}.
- Une variable est traduite en une valeur selon le contexte.

```
from django.shortcuts import render

def book_detail (request, book_id):
  book = Book.objects.get(pk=book_id)
  context = {
        'book_title': book.title,
  }
  return render(request,'app/book_detail.html', context)
```

```
<h1> The book title is {{ book_title }} </h1>
```

Gabarits

- Les filtres permettent la modification de l'affichage des variables.
- Un filtre s'écrit sous la forme: **{{ variable | filtre }}**.
- Les filtres peuvent s'enchaîner:

  {{ variable | filtre1 | filtre2 }}.
- Il existe des filtres qui acceptent des paramètres.

```
<p> The book title is {{ book_title|capfirst }} </p>
<p> The publication date is {{ pub_date|date:"Y-m-d" }}
</p>
```

Gabarits

- Les balises « tags » sont entourées par **{%** et **%}**.
- Certaines balises nécessitent une balise fermante:

  {% endtag %}.
- Les balises produisent du texte, contrôlent la logique, effectuent des boucles, chargent des informations externes...

```
<h1>Books List</h1>
{% if books_list %}
    <ul>
    {% for b in books_list %}
        <li>{{ b.title }}</li>
    {% endfor %}
    </ul>
{% else %} {# No Books #}
    <p> Books unavailable.</p>
{% endif %}
```

Gabarits

- Les structures conditionnelle sont exprimées par

  {% if condition **%}** instrctions **{%** endif **%}**

```
{% if books_list and journals_list %}
      <p>Both books and journals are available. </p>
{% endif %}
```

- Pour une structure conditionnelle composite, on utilise

  if… elif…else…

```
{% if books_list %}
      Number of books:{{books_list|length }}
{% elif
      ...
{% else %}
      <p>No journals. </p>
{% endif %}
```

Gabarits

- La structure itérative est exprimée par **for**
- Il est parfois utile de récupérer des données d'un dictionnaire via le mot clé **items**.

```
{% for key, value in data.items %}
        {{ key }}: {{ value }}
{% endfor %}
```

- Par exemple:

```
{% for book in books_list %}
        <p>{{ book.title }}</p>
{% empty %}
        <p> There are no books! </p>
{% endfor %}
```

- « **empty** » est vrai si la liste est vide.

Gabarits

- On a recourt parfois à l'appel d'urls dans les templates. Exemple:

```
{% for b in books_list %}
      <li><a href="/books/{{ b.id}}">{{ b.title }}</a></li>
{% endfor %}
```

- Ceci n'est pas recommendé, on utilise plutôt le tag **{% url %}** afin de respecter le principe **DRY** et éviter de figer les URL dans les templates ➔ `{% url 'url_name' V1 V2 %}`
- Par exemple:

```
<a href="{% url 'book_details' b.id %}">
      {{ b.title }}
</a>
```

Gabarits

- On peut définir un template de base pour servir comme modèle.
- On crée un template « **base.html** », Exemple:

```
<head>
    <title> Welcome to - {% block title %} {% endblock %} </title>
</head>
<body>
    {% block header %} {% endblock header %}
    {% block content %} {% endblock %}
    {% block footer %}{% endblock %}
</body>
```

- Le template « **index.html** » hérite de « base.html »:

```
{% extends "base.html" %}
{% block title %} Home Page {% endblock %}
{% block content %} <h1> We sell books! </h1> {% endblock %}
{% block footer %} <h2> Well, that's it, the end! </h2> {% endblock %}
```

Gabarits

- On peut inclure un template via le tag « **include** », Exemple:

header.html

```
{{ greeting }} {{ username| default: "Visitor" }}
```

index.html

```
{% extends "base.html" %}
{% block title %} Home Page {% endblock %}
{% block header %}
  {% include "header.html" with greeting="Hello" username=user.username %}
{% endblock %}
{% block content %} <h1> We sell books! </h1> {% endblock %}
{% block footer %} <h2> Well, that's it, the end! </h2> {% endblock %}
```

Gabarits

- Les fichiers statiques (CSS, JS, images) sont gérés par Django via **django.contrib.staticfiles**
- Il faut:
  - Définir le chemin vers ces fichiers dans settings.py:

    ```
    STATIC_URL = '/static/'
    ```

  - Charger les éléments dans le gabarit via:

    ```
    {% load static %}
    ```

  - Faire appel à la ressource (`App/static/App/img/example.jpg`), par exemple:

```
<img src="{% static "App/img/example.jpg" %}" alt="My image"/>
```

URLs & Views

- Lorsqu'un utilisateur essaye de se connecter:
  1. Django identifie le module racine (**urls.py**) définie au niveau de **settings.py**: **ROOT_URLCONF**
  2. Django charge ce module et parcourt la variable **urlpatterns**. Il s'agit d'une séquence d'instance composée soit de:
     - django.urls.path()
     - Django.urls.re_path()
  3. Django s'arrête à la première correspondance avec l'URL demandée et importe la vue correspondante.
  4. Sinon en cas où il n'y a pas de correspondance ou en cas d'exception, Django appelle une vue d'erreur.

URLs & Views

- Pour déclarer une vue, il faut:
  - Déclarer un urlpattern dans urls.py de l'application
  - Définir la méthode correspondante dans views.py
- Exemple:

```
from django.urls import path
from django.views.generic import TemplateView
from .views import list_books, book_detail, MainView

urlpatterns = [
    #pre-defined class from django: we don't need to specify a view
    path('', TemplateView.as_view(template_name='index.html')),
    # function based views
    path('list_books', list_books),
    path('book_detail/<int:id_book>', book_detail),
    # class based views
    #path('main', MainView.as_view())
],
```

URLs & Views

- Django présente plusieurs convertisseurs de chemin par défaut:
  - str
  - int
  - slug
  - uuid
- Exemple:

```
path('book_detail/<int:id_book>', book_detail),
path('book_detail/<str:title_book>', book_detail),
```

- Il faut indiquer au projet d'inclure les URLs relatives à l'application.
- Exemple:

```
path('books/', include('BooksApp.urls')),
```

URLs & Views

- Pour décrire des URLS via les expressions régulières, il faut utiliser la fonction re-path.
  - (?P<nom>motif): correspond à la syntaxe des expressions régulières en Python. Ceci se base sur la syntaxe des « re » en Python
  - Chaque paramètre est envoyé à la vue en tant que chaîne de caractères, peu importe la correspondance qu'effectue l'expression régulière.

https://docs.djangoproject.com/fr/3.1/topics/http/urls/