**Développez votre site web avec le framework Django**

**Qu’est-ce qu’un framework ?**

Un framework est un ensemble d'outils qui simplifie le travail d'un développeur. Traduit littéralement de l'anglais, un framework est un « cadre de travail ».

Il apporte les bases communes à la majorité des programmes ou des sites web. Celles-ci étant souvent identiques (le fonctionnement d'un espace membres est commun à une très grande majorité de sites web de nos jours), un développeur peut les réutiliser simplement et se concentrer sur les particularités de son projet.

Il s'agit donc d'un ensemble de bibliothèques coordonnées, qui permettent à un développeur d'éviter de réécrire plusieurs fois une même fonctionnalité, et ainsi d'éviter de réinventer constamment la roue.

Qu’est-ce que Django ?

Django est donc un framework Python *destiné au web*

Django est créé en 2003 dans une agence de presse, [Lawrence Journal-World](http://www2.ljworld.com/), qui devait développer des sites web complets dans des laps de temps très courts (d'où l'idée du framework)

**Installation sous Linux**

pip install django

**Installation sous Mac**

sudo easy\_install pip

**Installer un environnement virtuel**

Sous Mac taper :

python3 -m venv NomDeLenvironnementVirtuel ex : python3 -m venv nikah

Cette ligne signifie je souhaite utiliser le module environnement virtuel que j’intitule nikah

cd nikah

Il faut maintenant l’activer en tapant : source bin/activate

Tu peux mettre un . à la place de source.

Pour quitter l’environnement on tape : deactivate.

Pour installer django on tape : pip install django

Pour upgrader pip on tape : pip install –U pip

Pour accéder au shell python tu tapes juste python , pour y sortir et revenir sur l’environnement virtuel tape : quit() ou exit() ou CTRL + D

Si tu veux connaître la version de ton django tapes :

python

import django

django.get\_version()

Création d’un projet

django-admin startproject nikah

Puis aller dans le projet via cd nikah

Lancement du serveur

python manage.py runserver

CTRL + D pour y sortir

Par défaut, l'accès au site de développement se fait via l'adressehttp://localhost:8000

Sans environnement virtuel et en ayant python 2.7 d’installé ainsi que python3 :

sudo python3 -m pip install django 🡪 Permet d’installer Django en utilisant python 3 et non le 2.7

**Le fonctionnement de Django**

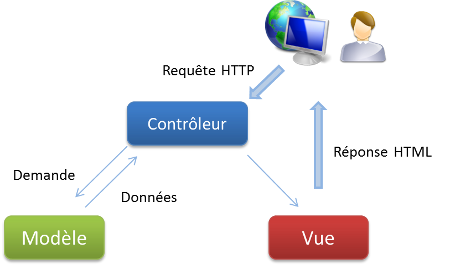
L’architecture MVC

 Il s'agit d'un modèle distinguant plusieurs rôles précis d'une application, qui doivent être accomplis. Comme son nom l'indique, l'architecture (ou « patron ») **M**odèle-**V**ue-**C**ontrôleur est composé de trois entités distinctes, chacune ayant son propre rôle à remplir.

Tout d'abord,**le modèle***représente une information* enregistrée quelque part, le plus souvent dans une base de données.

Ensuite **la vue** qui est, comme son nom l'indique, la *visualisation de l'information*. C'est la seule chose que l'utilisateur peut voir.

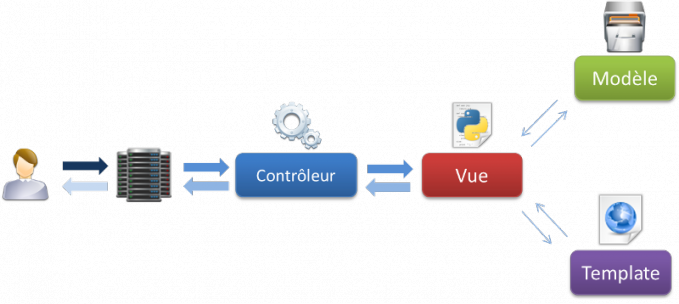
Dans le modèle MVC, **le contrôleur**reçoit une requête de l'utilisateur, il récupère ensuite les données voulues dans les modèles. Après un éventuel traitement sur ces données, il transmet ces données à la vue, afin qu'elle s'occupe de les afficher.



Le modèle MVT

L'architecture utilisée par Django diffère légèrement de l'architecture MVC classique. En effet, la « magie » de Django réside dans le fait qu'il *gère lui-même la partie contrôleur*(gestion des requêtes du client, des droits sur les actions…). Ainsi, nous parlons plutôt de framework utilisant l'architecture **MVT** : **M**odèle-**V**ue-**T**emplate.

Le contrôleur reçoit la requête, il se charge, via les règles de routage URL définies, d'exécuter la vue correspondante. Cette dernière récupère les données des modèles et génère un rendu HTML à partir du template et de ces données. Une fois la page générée, l'appel fait chemin arrière, et le serveur renvoie le résultat au navigateur de l'internaute.



On distingue les quatre parties qu'un développeur doit gérer :

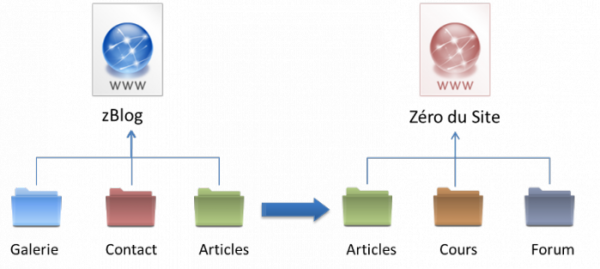
* Le routage des requêtes, en fonction de l'URL ;
* La représentation des données dans l'application, avec leur gestion (ajout, édition, suppression…), c'est-à-dire les modèles ;
* L'affichage de ces données et de toute autre information au format HTML, c'est-à-dire les templates ;
* Enfin le lien entre les deux derniers points : la vue qui récupère les données et génère le template selon celles-ci.

Projets et applications

En plus de l'architecture MVT, Django introduit le développement d'un site sous forme de projet. Chaque site web conçu avec Django est considéré comme un projet, composé de plusieurs applications. Une application consiste en un dossier contenant plusieurs fichiers de code, chacun étant relatif à une composante du modèle MVT que nous avons vu. En effet, chaque bloc du site web est isolé dans un dossier avec ses vues, ses modèles et ses schémas d'URL.

Lors de la conception de votre site, vous allez devoir penser aux applications que vous souhaitez développer. Voici quelques exemples d'applications :

* Un module d'actualités ;
* Un forum ;
* Un système de contact ;
* Une galerie de photos ;
* Un système de donations.
* Une application correctement conçue pourra être réutilisée dans d'autres projets par un simple copier/coller (ou un import), comme le montre la figure suivante.



Gestion d’un projet

Configuration du projet

Dans le fichier settings.py ne jamais mettre DEBUG = True quand le site est en production.

Nous conseillons pour le développement local de garder cette configuration. L'avantage de SQLite comme gestionnaire de base de données pour le développement est simple : il ne s'agit que d'un simple fichier. Il n'y a donc pas besoin d'installer un service à part ; Python et Django se chargent de tout. Si vous n'avez aucune idée de ce qu'est réellement une base de données SQL, n'ayez aucune crainte, le prochain chapitre vous expliquera en détail en quoi elles consistent et comment elles fonctionnent.

Si vous souhaitez utilisez une base de données MySQL par exemple, plus de champs sont nécessaires, voici une configuration d'exemple :

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.mysql', # Backends disponibles : 'postgresql', 'mysql', 'sqlite3' et 'oracle'.

'NAME': 'crepes\_bretonnes', # Nom de la base de données

'USER': '<nom d\'utilisateur>',

'PASSWORD': '<mot de passe MySQL>',

'HOST': '127.0.0.1', # Utile si votre base de données est sur une autre machine

'PORT': '3306', # ... et si elle utilise un autre port que celui par défaut

}

}

Le sous-dossier contient quatre fichiers Python, à savoir settings.py, urls.py, wsgi.py et \_\_init\_\_.py. Ne touchez surtout pas à ces deux derniers fichiers, ils n'ont pas pour but d'être modifiés ! Les deux autres fichiers ont des noms plutôt éloquents : settings.py contiendra la configuration de votre projet, tandis que urls.py rassemblera toutes les URL de votre site web et la liste des fonctions à appeler pour chaque URL. Nous reviendrons sur ces deux fichiers plus tard.

Ensuite, à la racine du projet, le fichier manage.py est en quelque sorte un raccourci local de la commande django-admin qui prend en charge la configuration de votre projet. Vous pouvez désormais oublier la commande django-admin, elle ne sert en réalité qu'à créer des projets, tout le reste se fait viamanage.py. Bien évidemment, n'éditez pas ce fichier non plus.

Votre projet étant créé, pour vous assurer que tout a été correctement effectué jusqu'à maintenant, vous pouvez lancer le serveur de développement via la commande python manage.py runserver:

**Créons notre première application**

Comme nous l'avons expliqué précédemment, un projet se compose de plusieurs applications, chacune ayant un but bien précis (système d'actualités, galerie photos…). Pour créer une application dans un projet, le fonctionnement est similaire à la création d'un projet : il suffit d'utiliser la commande manage.py startapp , à l'intérieur de votre projet. Pour notre site sur les crêpes bretonnes, créons un blog pour publier nos nouvelles recettes, il faut d’abord sortir du serveur avec CTRL + C et taper :

python manage.py startapp blog

Cette commande av créer un dossier blog avec plusieurs fichiers :

* admin.py  va permettre de définir ce que souhaitez afficher et modifier comme modèles depuis l'administration de l'application générée automatiquement ;
* apps.py  permet de configurer certaines propriétés de l'application, notamment son nom ;
* models.py contiendra vos modèles ;
* tests.py  permet la création de tests unitaires (un chapitre y est consacré dans la quatrième partie de ce cours) ;
* views.py  contiendra toutes les vues de votre application.

Dernière petite chose, il faut ajouter cette application au projet. Pour que Django considère le sous-dossier blog  comme une application, il faut donc l'ajouter dans la configuration.

Retournez dans settings.py, et cherchez la variable INSTALLED\_APPS

Tout en conservant les autres applications installées par défaut, ajoutez une chaîne de caractères avec le nom de votre application, exemple :

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'blog',

]

**Les bases de données et Django**

Une base de données peut contenir plusieurs tableaux, chacun servant à enregistrer un certain type d'élément. Par exemple, dans votre base, vous pourriez avoir un tableau qui recensera vos utilisateurs, un autre pour les articles, encore un autre pour les commentaires, etc.

| **Titre** | **Réalisateur** | **Année de sortie** | **Note (sur 20)** |
| --- | --- | --- | --- |
| *Pulp Fiction* | Quentin Tarantino | 1994 | 20 |
| *Inglorious Basterds* | Quentin Tarantino | 2009 | 18 |
| *Holy Grail* | Monty Python | 1975 | 19 |
| *Fight Club* | David Fincher | 1999 | 20 |
| *Life of Brian* | Monty Python | 1979 | 17 |

En anglais, « tableau » est traduit par « *table* ». Cependant, beaucoup de ressources francophones utilisent pourtant le mot anglais « *table* » pour désigner un tableau, à cause de la prépondérance de l'anglais dans l'informatique. À partir de maintenant, nous utiliserons également le mot « table » pour désigner un tableau dans une base de données.

**Le langage SQL et les gestionnaires de base de données**

Il existe plusieurs programmes qui s'occupent de gérer des bases de données. Nous les appelons, tout naturellement, des gestionnaires de bases de données (ou « SGBD » pour « systèmes de gestion de bases de données »). Ces derniers s'occupent de tout : création de nouvelles tables, ajout de nouvelles entrées dans une table, mise à jour des données, renvoi des entrées déjà enregistrées, etc. Il y a énormément de SGBD, chacun avec des caractéristiques particulières. Néanmoins, ils se divisent en deux grandes catégories : les bases de données SQL et les bases de données non-SQL. Nous allons nous intéresser à la première catégorie (celle que Django utilise majoritairement).

Les gestionnaires de bases de données SQL sont les plus populaires et les plus utilisés pour le moment. Ceux-ci reprennent l'utilisation du classement par tableau tel que nous l'avons vu. L'acronyme « SQL » signifie « *Structured Query Language* », ou en français « langage de requêtes structurées ». En effet, lorsque vous souhaitez demander au SGBD toutes les entrées d'une table, vous devez communiquer avec le serveur (le programme qui sert les données) dans un langage qu'il comprend. Ainsi, si pour commander un café vous devez parler en français, pour demander les données au gestionnaire vous devez parler en SQL.

Voici quelques gestionnaires SQL bien connus (dont vous avez sûrement déjà dû voir le nom quelque part) :

* MySQL (et son dérivé, MariaDB) : gratuit, probablement le plus connu et le plus utilisé à travers le monde ;
* PostgreSQL : gratuit, moins connu que MySQL, mais possède quelques fonctionnalités de plus que ce dernier ;
* Oracle Database : généralement utilisé dans de grandes entreprises, une version gratuite existe, mais est très limitée ;
* Microsoft SQL Server : payant, développé par Microsoft ;
* SQLite : très léger, gratuit, et très simple à installer (en réalité, il n'y a rien à installer avec Django).

Lors de la configuration de votre projet Django dans le chapitre précédent, nous vous avons conseillé d'utiliser SQLite. Pourquoi ? Car contrairement aux autres SGBD qui ont besoin d'un serveur lancé en permanence pour traiter les données, une base de données SQLite consiste en un simple fichier. C'est la bibliothèque Python (nomméesqlite3) qui se chargera de modifier et renvoyer les données de la base. C'est très utile en développement, car il n'y a rien à installer, mais en production mieux vaut utiliser un SGBD plus performant comme PostgreSQL ou MySQL.

**La magie des ORM**

Apprendre le langage SQL et écrire ses propres requêtes est quelque chose d'assez difficile et contraignant lorsque nous débutons. Cela prend beaucoup de temps et est assez rébarbatif. Heureusement, Django propose un système pour bénéficier des avantages d'une base de données SQL sans devoir écrire ne serait-ce qu'une seule requête SQL !

Ce type de système s'appelle ORM pour « *object-relationnal mapping* ». Derrière ce nom un peu barbare se cache un fonctionnement simple et très utile. Lorsque vous créez un modèle dans votre application Django, le framework va automatiquement créer une table adaptée dans la base de données qui permettra d'enregistrer les données relatives au modèle.

Pour conclure, l'ORM est un système très flexible, utilisé dans la plupart des frameworks web aujourd'hui. Django l'insère parfaitement bien dans l'architecture MVT que nous avons décrite précédemment. L'avantage majeur est bien évidemment une meilleure compréhension du code pour les non habitués à SQL. Cela nous donne également la possibilité de migrer d'un système de base de données à un autre sans toucher à notre code.

**Votre première page grâce aux vues**

Pour rappel, comme vu dans la théorie, *chaque vue se doit d'être associée au minimum à une URL*. Avec Django, une vue est représentée par une fonction définie dans le fichier views.py, prenant en paramètre une requête HTTP et renvoyant une réponse http.

#### La gestion des vues

Chaque application possède son propre fichier views.py

Commençons à travailler dans blog/views.py. Par défaut, Django a généré gentiment ce fichier :

from django.shortcus import render

Nous allons créer une fonction dans ce fichier qui remplira le rôle de la vue. Ca sera du code HTML directement dans la vue puis renvoyé au client. Nous ignorons temporairement la méthode render déjà importée, et utilisons HttpResponse  qui permet de construire une réponse HTTP :

from django.http import HttpResponse

from django.shortcuts import render

def home(request):

""" Exemple de page non valide au niveau HTML pour que l'exemple soit concis """

return HttpResponse("""

<h1>Bienvenue sur mon blog !</h1>

<p>Les crêpes bretonnes ça tue des mouettes en plein vol !</p>

""")

# Create your views here.

Ce code se divise en trois parties :

* Nous importons la classe HttpResponse du module django.http. Cette classe permet de retourner une réponse (texte brut ou HTML comme ici) depuis une chaîne de caractères HttpResponse est spécifique à Django et permet d'encapsuler votre réponse dans un objet plus générique, que le framework peut traiter plus aisément.
* Une fonction home, avec comme argument une instance de HttpRequest. Nous avons nommé ici (et c'est partout le cas par convention) sobrement cet argument request. Celui-ci contient des informations sur la méthode de la requête (GET, POST), les données des formulaires, la session du client, etc. Nous y reviendrons plus tard.
* Finalement, la fonction retourne une instance de HttpResponse à partir d'une chaîne de caractères contenant du HTML, que la fonction renvoie ensuite au framework.

Toutes les fonctions prendront comme premier argument un objet du type HttpRequest. Toutes les vues doivent forcément retourner une instance de HttpResponse, *sans quoi Django générera une erreur*.

Par la suite, *ne renvoyez jamais du code HTML directement depuis la vue*comme nous le faisons ici. Passez toujours par des templates

**Routage d’URL : Comment j’accède à ma vue ?**

Nous avons désormais une vue opérationnelle, il n'y a plus qu'à l'appeler depuis une URL. Mais comment ? En effet, nous n'avons pas encore défini vers quelle URL pointait cette fonction. Pour ce faire, il faut modifier le fichier urls.py du projet de base donc dans nikah/url.py

La liste urlpatterns, qui permet de définir les associations entre URL et vues. Une association de routage basique se définit par un sous-tuple composé des éléments suivants :

* Le schéma de l'URL : une URL peut être composée d'arguments qui permettent par la suite de retrouver des informations dans les modèles par exemple. Exemple : un titre d'article, le numéro d'un commentaire, etc. ;
* Le chemin Python vers la vue correspondante.

Par exemple, en reprenant la vue définie tout à l'heure, si nous souhaitons que celle-ci soit accessible depuis l'URL http://www.nikah.com/accueil, il suffit de rajouter cette règle dans votre url patterns:

from django.urls import path

from blog import views

urlpatterns = [

path('admin/', admin.site.urls),

path('accueil', views.home),

]

Le premier argument représente le fragment d'URL auquel le visiteur accédera et le second argument est la vue que l'on a créé dans le fichier blog/views.py On y accède donc via le module views  importé juste au-dessus.

Grâce à cette règle, dès qu’un client demandera la page http://www.nikah.com/accueil, le contrôleur lui enverra la vue blog.views.home

Enregistrez les modifications, lancez le serveur de développement Django et laissez-le tourner (pour rappel :python manage.py runserver ), et rendez-vous sur http://localhost:8000/accueil

Généralement, la page d'accueil est situé à la racine du site, à savoir 127.0.0.1:8000 dans notre cas. Il est bien sûr possible d'avoir ce comportement, en mettant une chaine vide à la place de  'accueil'  dans notre configuration :  path('', views.home).

**Organiser proprement ses urls**

il est généralement bien vu de créer dans chaque application un fichier également nommé urls.py  et d'inclure ce dernier par la suite dans le fichier urls.py  global du projet.

#### Comment procède-t-on ?

Tout d'abord, il faut créer un fichier urls.py  dans le dossier de votre application, ici blog. Ensuite, il suffit d'y réécrire l'URL que nous avons déjà écrite précédemment (ne pas oublier l'importation des modules nécessaires !) :

from django.urls import path

from . import views

urlpatterns = [

path('', views.home),

]

Maintenant, retournons à crepes\_bretonnes/urls.py. Nous pouvons y enlever la règle réécrite dans blog/urls.py  et la remplacer par une nouvelle qui utilise la fonction include  de django.conf.urls

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('blog/', include('blog.urls')),

]

Maintenant cette vue sera accessible à http://localhost:8000/blog/

**Passer des arguments à une vue**

Pour passer des arguments dans une URL, il faut capturer ces arguments directement depuis l'écriture de nos URL. Par exemple, si nous souhaitons sur notre blog pouvoir accéder à un certain article via l'adresse/blog/article/\*\*  où \*\*  sera l'identifiant de l'article (un nombre unique), il faut fournir le routage suivant dans blog/urls.py

urlpatterns = [

path('', views.home),

path('article/<id\_article>', views.view\_article),

]

Lorsque l'URL /blog/article/42 est demandée, Django regarde le routage et exécute la fonction view\_article en passant en paramètre 42. Autrement dit, Django appelle la vue de cette manière : view\_article(request, 42).

Exemple de fonction view\_article

ll est possible de capturer plusieurs paramètres, dans une même URL et forcer son type :

urlpatterns = [

path('accueil', views.home),

path('article/<id\_article>', views.view\_article),

path('articles/<str:tag>', views.list\_articles\_by\_tag),

path('articles/<int:year>/<int:month>', views.list\_articles),

]

Ici, on peut imaginer que l'on a deux autres vues. La première qui prend en paramètre du texte, permettant de filtrer les articles à afficher selon leurs mots-clés et une autre qui affiche les articles par mois.

##### Un dernier mot sur les paramètres de vue

Quand nous avons écrit nos URL, nous avons pris soin de nommer les paramètres à chaque fois. Cela permet à Django de faire la correspondance entre ce qu'il a décortiqué dans l'URL envoyé par l'utilisateur et ce qui sera envoyé à votre vue.

Ainsi, l'ordre des paramètres importe peu dans votre vue (hormis le premier, qui est toujours  request). Ainsi, en reprenant notre configuration plus haut, la vue suivante est tout à fait valable :

def list\_articles(request, month, year):

""" Liste des articles d'un mois précis. """

return HttpResponse(

"Vous avez demandé les articles de {0} {1}.".format(month, year)

)

Dans cet exemple, mois et année (month  et year) ne sont pas dans le même ordre entre le urls.py  et le views.py, mais Django s'en occupe et règle l'ordre des arguments en fonction des noms qui ont été donnés dans leurls.py.

Il faut juste s'assurer que les noms de variables donnés dans le fichier urls.py  coïncident avec les noms donnés dans la déclaration de la vue, sans quoi Python retournera une erreur.

Il est également possible d'avoir plusieurs URL qui pointe vers la même vue, via des arguments optionnels.

# urls.py

urlpatterns = [

path('articles/<int:year>/', views.list\_articles),

path('articles/<int:year>/<int:month>', views.list\_articles),

]

# views.py

def list\_articles(request, year, month=1):

return HttpResponse('Articles de %s/%s' % (year, month))

Dans cet exemple, les adresses  /articles/2020/5  et  /articles/2020/  sont valides, cette dernière aura implicitement "1" pour le mois. Par ailleurs,  /articles/2020/  et  /articles/2020/1  renverront le même résultat.

**Des réponses spéciales**

**Simuler une page non trouvée**

Parfois, une URL correspond bien à un pattern mais ne peut tout de même pas être considérée comme une page existante. Par exemple, lorsque vous souhaitez afficher un article avec un identifiant introuvable, il est impossible de renvoyer une page, même si Django a correctement identifié l'URL et utilisé la bonne vue. Dans ce cas-là, nous pouvons le faire savoir à l'utilisateur via une page d'erreur 404, qui correspond au code d'erreur indiquant qu'une page n'a pas été trouvée. Pour ce faire, il faut utiliser une exception du framework :Http404. Cette exception, du module django.http, arrête le traitement de la vue, et renvoie l'utilisateur vers une page d'erreur.

Voici un rapide exemple d'une vue compatible avec une des règles de routage que nous avons décrites dans le sous-chapitre précédent :

from django.http import HttpResponse, Http404

def view\_article(request, id\_article):

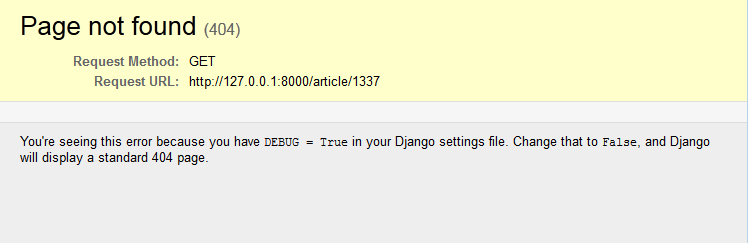
# Si l'ID est supérieur à 100, nous considérons que l'article n'existe pas

if id\_article > 100:

raise Http404

return HttpResponse('<h1>Mon article ici</h1>')

Si à l'appel de la page l'argument id\_article  est supérieur à 100, la page retournée sera une erreur 404 de Django, visible à la figure suivante. Il est bien entendu possible de personnaliser par la suite cette vue, avec un template, afin d'avoir une page d'erreur qui soit en accord avec le design de votre site, mais cela ne fonctionne uniquement qu'avec DEBUG = False  dans le settings.py(en production donc). Si vous êtes en mode de développement, vous aurez toujours une erreur similaire à la figure suivante :



#### Rediriger l'utilisateur

Le second cas que nous allons aborder concerne les redirections. Il arrive que vous souhaitiez rediriger votre utilisateur vers une autre page lorsqu'une action vient de se dérouler, ou en cas d'erreur rencontrée. Par exemple, lorsqu'un utilisateur se connecte, il est souvent redirigé soit vers l'accueil, soit vers sa page d'origine. Une redirection est réalisable avec Django via la méthode redirect  qui renvoie un objet HttpResponseRedirect  (classe héritant de HttpResponse), qui redirigera l'utilisateur vers une autre URL. La méthode redirect  peut prendre en paramètres plusieurs types d'arguments, dont notamment une URL brute (chaîne de caractères) ou le nom d'une vue.

Si par exemple vous voulez que votre vue, après une certaine opération, redirige vos visiteurs vers le site officiel de Django, il faudrait procéder ainsi :

from django.shortcuts import redirect

def list\_articles(request, year, month):

# Il veut des articles ? Soyons fourbe et redirigeons-le vers djangoproject.com

return redirect("https://www.djangoproject.com")

N'oubliez pas qu'une URL valide pour accéder à cette vue serait par exemple ainsi /blog/articles/2014/09

Cependant, si vous souhaitez rediriger votre visiteur vers une autre page de votre site web, il est plus intéressant de privilégier l'autre méthode, qui permet de garder indépendante la configuration des URL et des vues. Nous devons donc passer en argument le nom de la vue vers laquelle nous voulons rediriger l'utilisateur, avec éventuellement des arguments destinés à celle-ci.

from django.http import HttpResponse, Http404

from django.shortcuts import redirect

def view\_article(request, id\_article):

if id\_article > 100:

raise Http404

return redirect(view\_redirection)

def view\_redirection(request):

return HttpResponse("Vous avez été redirigé.")

path('redirection', views.view\_redirection),

Ici, si l'utilisateur accède à l'URL/blog/article/101, il aura toujours une page 404. Par contre, s'il choisit un ID inférieur à 100, alors il sera redirigé vers la seconde vue, qui affiche un simple message.

Il est également possible de préciser si la redirection est temporaire ou définitive en ajoutant le paramètre permanent=True. L'utilisateur ne verra aucune différence, mais ce sont des détails que les moteurs de recherche prennent en compte lors du référencement de votre site web.

Si nous souhaitions rediriger un visiteur vers la vue view\_article  définie précédemment par un ID d'article spécifique, il suffirait simplement d'utiliser la méthode redirect  ainsi :

return redirect(view\_article, id\_article=42)

Finalement, il est également possible d'indiquer une vue à redirect à l'aide d'une nouvelle manière : en indiquant le nom de la vue tel que renseigné dans urls.py

En réalité, la fonction redirect  agit en deux temps. Elle va tout d'abord construire l'URL vers la vue selon le routage indiqué dansurls.py. Ici, elle va donc générer l'URL/blog/article/42et ensuite rediriger l'utilisateur vers cette URL. Ainsi, si par la suite vous souhaitez modifier vos URL dansurls.py, toutes les redirections se mettront à jour automatiquement. Il s'agit d'une fonctionnalité extrêmement pratique, il ne faut donc jamais écrire d'URL en dur dans vos vues, sauf quand cette méthode est inutilisable (vers des sites tiers par exemple).

Sachez qu'au lieu d'écrire à chaque fois tout le chemin d'une vue ou de l'importer, il est possible de lui assigner un nom plus court et plus facile à utiliser dansurls.py. Par exemple :

path('article/<int:id\_article>$', views.view\_article, name='afficher\_article'),

Notez le paramètre name='afficher\_article'  qui permet d'indiquer le nom de la vue. Avec ce routage, en plus de pouvoir passer directement la fonction ou le chemin vers celle-ci en argument, nous pouvons faire beaucoup plus court et procéder comme ceci :

return redirect('afficher\_article', id\_article=42)

Pour terminer, sachez qu'il existe également une fonction qui permet de générer simplement l'URL et s'utilise de la même façon que redirect; il s'agit de reverse(django.urls.reverse). Cette fonction ne retournera pas un objet HttpResponseRedirect, mais simplement une chaîne de caractères contenant l'URL vers la vue selon les éventuels arguments donnés. Une variante de cette fonction sera utilisée dans les templates peu après pour générer des liens HTML vers les autres pages du site.

**Les templates**

Tous les frameworks web actuels utilisent un moteur de templates. Les templates sont écrits dans un mini-langage de programmation propre à Django et qui possède des expressions et des structures de contrôle basiques (if/else, boucle for, etc.) que nous appelons des tags. Le moteur transforme les tags qu'il rencontre dans le fichier par le rendu HTML correspondant

**Lier template et vue**

C'est la vue qui se charge de transmettre l'information de la requête au template, puis de retourner le HTML généré au client. Dans le chapitre précédent, nous avons utilisé la méthode HttpResponse(text)  pour renvoyer le HTML au navigateur. Cette méthode prend comme paramètre une chaîne de caractères et la renvoie sous la forme d'une réponse HTTP. La question ici est la suivante : comment faire pour appeler notre template, et générer la réponse à partir de celui-ci ? La fonction render  a été conçue pour résoudre ce problème.

La fonction render  est en réalité une méthode de django.shortcut  qui nous simplifie la vie : elle génère un objet HttpResponse  après avoir traité notre template.

D’abord on ajoute cette vue dans le dossier views de notre application blog :

**from datetime import datetime**

**def date\_actuelle(request):**

**return render(request, 'blog/date.html', {'date': datetime.now()})**

**def addition(request, nombre1, nombre2):**

**total = nombre1 + nombre2**

**# Retourne nombre1, nombre2 et la somme des deux au tpl**

**return render(request, 'blog/addition.html', locals())**

Ensuite on ajoute ces urls dans le dossier urls.py de l’application blog :

**path('date', views.date\_actuelle),**

**path('addition/<int:nombre1>/<int:nombre2>/', views.addition)**

Comme vous pouvez le voir, la fonction renderprend en argument trois paramètres :

1. La requête HTTP initiale, que l'on appellerequest  par convention pour rappel ;
2. Le chemin vers le template adéquat dans un des dossiers de templates donnés dans settings.py;
3. Un dictionnaire reprenant les variables qui seront accessibles dans le template.

Avant d'écrire notre template, il faut d'abord se demander où est-ce que l'on va l'enregistrer. Par défaut, Django va chercher les templates aux endroits suivants :

1. Dans la liste des dossiers fournis du paramètre DIR  de la variable de configurationTEMPLATES  ;
2. S'il ne l'a pas trouvé, dans un sous-dossier "templates" de l'application.

Dans settings.py on met ainsi :

'DIRS': [

# Cette ligne ajoute le dossier templates/ à la racine du projet

os.path.join(BASE\_DIR, 'templates'),

],

Ensuite il faut créer un dossier templates  tout à la racine du projet.

Puis dans blog on créé un dossier templates dans lequel on va créer un dossier blog et deux fichiers date.html et addition.html

Dans date :

<h1>Bienvenue sur mon blog</h1>

<p>La date actuelle est : {{ date }}</p>

Dans addition :

<h1>Ma super calculatrice</h1>

<p>{{ nombre1 }} + {{ nombre2 }}, ça fait <strong>{{ total }}</strong> !<br />

Nous pouvons également calculer la somme dans le template : {{ nombre1|add:nombre2 }}.</p>

En allant sur http://127.0.0.1:8000/blog/addition/5/3 , vous obtiendrez le résultat de l'addition 5 + 3.

La seule différence dans la vue réside dans le deuxième argument donné à render. Au lieu de lui passer un dictionnaire directement, nous faisons appel à la fonction locals()qui va retourner un dictionnaire contenant toutes les variables locales de la fonction depuis laquelle locals()  a été appelée. Les clés seront les noms de variables (par exempletotal), et les valeurs du dictionnaire seront tout simplement… les valeurs des variables de la fonction ! Ainsi, si nombre1 valait 42, la valeur nombre1  du dictionnaire vaudra elle aussi 42.

dddd

**Affichons nos variables à l’utilisateur**

**Manipulons nos données avec les tags**

**Résumé pour la fiche :**

 1° Il faut d’abord créer l’application voulue car un projet est un ensemble d’applications :

python manage.py startapp galerie

2° Dans settings.py au niveau de INSTALLED\_APPS il faut ajouter tout à la fin le nom de notre application.

3°Créer un fichier views.py dans le dossier de fichiers de notre nouvelle application.

4° Créer les fonctions voulues dans ce fichier views.py sans oublier d’importer les modules nécessaires.

5° Créer un fichier urls.py dans le dossier de fichiers de la nouvelle application.

6° Ajouter la route dans le fichier urls.py sans oublier les modules nécessaires.

7° Ajouter une fonction avec un include dans le fichier urls.py de base sans oublier les modules nécessaires.