



Créez vos applications web avec Django

Par Mathieu Xhonneux (MathX)
et Maxime Lorient (Ssx`z)



www.siteduzero.com

Sommaire

Sommaire	2
Lire aussi	3
Créez vos applications web avec Django	5
Partie 1 : Présentation de Django	6
Créez vos applications web avec Django	6
Qu'est-ce qu'un framework ?	6
Quels sont les avantages d'un framework ?	6
Quels sont les désavantages d'un framework ?	6
Qu'est-ce que Django ?	6
Pourquoi ce succès ?	7
Une communauté à votre service	7
Téléchargement et installation	7
Linux et Mac OS	8
Windows	8
Vérification de l'installation	9
Le fonctionnement de Django	9
Un peu de théorie : l'architecture MVC	10
La spécificité de Django : le modèle MVT	10
Projets et applications	11
Gestion d'un projet	12
Créons notre premier projet	13
Configurez votre projet	15
Créons notre première application	16
Les bases de données et Django	17
Une base de données, c'est quoi ?	18
Le langage SQL et les gestionnaires de base de données	18
La magie des ORM	19
Le principe des clés étrangères	20
Partie 2 : Premiers pas	22
Votre première page grâce aux vues	22
Hello World !	22
La gestion des vues	22
Routage d'URL : comment j'accède à ma vue ?	23
Organiser proprement vos URLs	25
Comment procède-t-on ?	25
Passer des arguments à vos vues	27
Des réponses spéciales	28
Simuler une page non trouvée	28
Rediriger l'utilisateur	29
Les templates	31
Lier template et vue	32
Affichons nos variable à l'utilisateur	34
Affichage d'une variable	34
Les filtres	34
Manipulons nos données avec les tags	35
Les conditions : {% if %}	35
Les boucles : {% for %}	36
Le tag {% block %}	38
Les liens vers les vues : {% url %}	39
Les commentaires : {% comment %}	40
Ajoutons des fichiers statiques	40
Les modèles	41
Créer un modèle	42
Jouons avec des données	43
Les liaisons entre modèles	47
Les modèles dans les vues	52
Afficher les articles du blog	52
Afficher un article précis	54
L'administration	57
Mise en place de l'administration	57
Les modules django.contrib	57
Accédons à cette administration !	57
Première prise en main	59
Administrons nos propres modèles	61
Personnalisons l'administration	62
Modifier l'aspect des listes	62
Modifier le formulaire d'édition	66
Retour sur notre problème de slug	69
Les formulaires	70
Créer un formulaire	71
Utiliser un formulaire dans une vue	72
Créons nos propres règles de validation	74
Des formulaires à partir de modèles	77

La gestion des fichiers	81
Enregistrer une image	81
Afficher une image	82
Encore plus loin	84
TP : un raccourcisseur d'URLs	85
Cahier de charges	86
Correction	87
Partie 3 : Techniques avancées	92
Les vues génériques	92
Premiers pas avec des pages statiques	92
Lister et afficher des données	93
Une liste d'objets en quelques lignes avec ListView	94
Afficher un article via DetailView	97
Agir sur les données	98
CreateView	98
UpdateView	99
DeleteView	101
Techniques avancées dans les modèles	103
Les requêtes complexes avec Q	104
L'agrégation	106
L'héritage de modèles	109
Les modèles parents abstraits	109
Les modèles parents classiques	109
Les modèles proxy	111
Simplifions nos templates : filtres, tags et contexte	111
Préparation du terrain : architecture des filtres et tags	112
Personnaliser l'affichage de données avec nos propres filtres	113
Un premier exemple de filtre sans argument	113
Un filtre avec arguments	115
Les contextes de templates	117
Un exemple maladroît : afficher la date sur toutes nos pages	118
Factorisons encore et toujours	118
Des structures plus complexes : les custom tags	120
Première étape : la fonction de compilation	121
Passage de variable dans notre tag	124
Quelques points à ne pas négliger	126
Les signaux et middlewares	127
Notifiez avec les signaux	128
Contrôlez tout avec les middlewares	130
Partie 4 : Des outils supplémentaires	133
Les utilisateurs	134
Commençons par la base	134
L'utilisateur	134
Les mots de passe	135
Étendre le modèle User	136
Passons aux vues	137
La connexion	137
La déconnexion	139
En général	139
Les vues génériques	140
Se connecter	141
Se déconnecter	141
Se déconnecter puis se connecter	141
Changer le mot de passe	141
Confirmation du changement de mot de passe	142
Demande de réinitialisation du mot de passe	142
Confirmation demande de réinitialisation du mot de passe	142
Réinitialiser le mot de passe	143
Confirmation de la réinitialisation du mot de passe	143
Les permissions et groupes	143
Les permissions	143
Les groupes	145
Les messages	145
Les bases	146
Dans les détails	147
La mise en cache	148
Cachez-vous !	149
Dans des fichiers	149
Dans la mémoire	149
Dans la base de données	150
En utilisant memcache	150
Pour le développement	151
Quand les données jouent à cache-cache	151
Cache par vue	151
Dans les templates	152
La mise en cache bas-niveau	152
Partie 5 : Annexes	154
Déployer votre application en production	155
Le déploiement	155
Gardez un oeil sur le projet	156

Activer l'envoi de mails	157
Quelques options utiles... ..	157
Hébergeurs supportant Django	158
Mémento des filtres	159
Liste des filtres	160
add	160
capfirst	160
center	160
cut	160
date	160
default	161
default_if_none	161
dictsort	161
dictsortreversed	162
divisibleby	162
escape	162
escapejs	162
filesizeformat	163
first	163
fix_ampersands	163
floatformat	163
force_escape	164
get_digit	164
iriencode	164
join	165
last	165
length	165
length_is	165
linebreaks	165
linebreaksbr	166
linenumbers	166
ljust	166
lower	166
make_list	167
phone2numeric	167
pluralize	167
pprint	167
random	167
removetags	167
rjust	168
safe	168
safeseq	168
slice	168
slugify	168
stringformat	168
striptags	169
time	169
timesince	169
timeuntil	170
title	170
truncatechars	171
truncatewords	171
truncatewords_html	171
unordered_list	171
upper	172
urlencode	172
urlize	172
urlizetrunc	173
wordcount	173
wordwrap	173
yesno	173



Créez vos applications web avec Django

Par



Maxime Lorant (Ssx'z) et



Mathieu Xhonneux (MathX)

Mise à jour : 08/01/2013

Difficulté : Intermédiaire



3 218 visites depuis 7 jours, classé 49/799

Django

« Le framework web pour les perfectionnistes sous pression »

En quelques années, les sites web n'ont cessé d'évoluer. Ils requièrent désormais un développement long et acharné, sans oublier le fait que ceux-ci peuvent parfois devenir très complexes et se mesurer en milliers de lignes de code. Aujourd'hui, la simple page web ne suffit plus, et que ce soit en entreprise ou en amateur, les attentes sont de plus en plus lourdes.

C'est de ce constat qu'est né Django : proposer un **développement plus efficace, plus rapide** d'une application dynamique web, tout en conservant la qualité ! Ce cours vous apprendra vous aussi à construire des sites web complexes et élégants, et en un temps record.



Ce tutoriel nécessite des connaissances préalables dans les domaines suivants :



- **Python** : bonne maîtrise des bases, de la programmation orienté objet et des expressions régulières ;
- **HTML/CSS** : maîtrise de toute la partie HTML (nous ne parlerons pas de CSS) ;

Si vous ne remplissez pas ces prérequis, nous ne pouvons que vous conseiller de les étudier avant d'entamer ce tutoriel.

Ce cours porte sur la **version 1.4 de Django**, et n'assure nullement que toutes les méthodes de ce cours marcheront forcément sur des versions antérieures ou postérieures à la 1.4

Partie 1 : Présentation de Django

Cette partie est avant tout introductive et théorique. Elle a pour but d'expliquer ce qu'est Django, son fonctionnement, la gestion d'un projet, etc...

Créez vos applications web avec Django

Django est un framework web écrit en Python. Avant de foncer sur son utilisation, nous allons tout d'abord voir dans ce chapitre ce qu'est un framework en général, ce qu'est Django et comment l'installer.

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 d'un programme ou d'un site web. Étant souvent identiques (un espace membre, ça reste un espace membre !), un développeur peut les réutiliser simplement et se concentrer sur l'essentiel de son projet.

Il s'agit donc d'une pile de bibliothèques coordonnées, qui permettent à un développeur d'éviter de réécrire plusieurs fois une application, et donc de réinventer constamment la roue. Inutile de dire que le gain en énergie et en temps est conséquent !

Quels sont les avantages d'un framework ?

Un framework instaure aussi en quelque sorte sa « ligne de conduite ». Tous les développeurs Django codent de façon assez homogène (leur code a le même fonctionnement, les mêmes principes), et tout développeur un tant soi peu expérimenté avec un framework qui rejoint un projet utilisant ce même framework sera donc directement familiarisé avec le code et pourra se mettre plus rapidement au travail.

Le fait que chaque framework possède une structure commune pour tous ses projets amène une conséquence toute aussi intéressante : en utilisant un framework, votre code sera le plus souvent déjà organisé, propre et facilement réutilisable par autrui.

Voilà d'ailleurs un grand challenge des frameworks : bien que ceux-ci doivent instaurer une structure commune, ils doivent aussi être souples et modulables, afin que ceux-ci puissent être utilisés pour une grande variété de projets, du plus banal au plus exotique. Autrement, leur intérêt en serait grandement limité !

Quels sont les désavantages d'un framework ?

Honnêtement, il n'existe pas vraiment de désavantages à utiliser un framework. Il faut bien évidemment prendre du temps à apprendre à manier un, mais ce temps d'apprentissage est largement récupéré par la suite vu la vitesse de développement qui peut parfois être décuplée par l'utilisation d'un framework. On pourrait éventuellement dire aussi que certains framework sont parfois un peu trop lourds, mais il incombe à son utilisateur de choisir le bon framework, adapté à ses besoins.

Qu'est-ce que Django ?

Django est donc un framework Python *destiné au web*. Ce n'est pas le seul dans sa catégorie, on peut compter d'autres frameworks Python du même genre comme web2py, Turbogears, CherryPy ou Zope. Django a cependant le mérite d'être le plus exhaustif et d'automatiser un bon nombre de choses.

Ce dernier est né dans une agence de presse en 2003. Ils devaient en effet développer des sites web complets et ceci dans des laps de temps très courts, d'où l'idée du framework.

En 2005, l'agence de presse [Lawrence Journal-World](#) décide de publier Django au grand public, le jugeant assez mature pour être réutilisé n'importe où.

Trois ans plus tard, la fondation Django Software est créée par les fondateurs du framework afin de pouvoir maintenir celui-ci et la communauté très active qui l'entoure.

Aujourd'hui, Django est devenu très populaire et est utilisé par des sociétés du monde entier, telles qu'[Instagram](#), [Pinterest](#) et même la [NASA](#) !





Pourquoi ce succès ?

Si Django est devenu très populaire, c'est notamment grâce à sa philosophie qui a su séduire de nombreux développeurs et chefs de projets. En effet, le framework prône le principe du « **Don't repeat yourself** », c'est à dire en français, « Ne vous répétez pas », et permet le développement rapide de meilleures et plus performantes applications web, tout en gardant le code élégant.

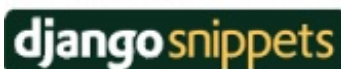
Django a pu appliquer sa philosophie de plusieurs manières. Par exemple, l'administration d'un site sera automatiquement générée, et celle-ci est très facilement adaptable. L'interaction avec une base de données se fait via un ensemble d'outils spécialisés et très pratique. Il est donc inutile de perdre son temps à écrire directement des requêtes SQL. De plus, d'autres bibliothèques complètes et bien pensées sont disponibles, comme un espace membre, ou une bibliothèque permettant la traduction de votre application web en plusieurs langages.

Une communauté à votre service

Évidemment, les avantages des frameworks en général s'adaptent aussi à Django. Ce dernier est soutenu par une communauté active et expérimentée, qui publie régulièrement de nouvelles versions du framework avec de nouvelles fonctionnalités, des corrections de bugs, etc.

Dernier point, mais pas le moins important, la communauté autour de Django a rédigé au fil des années une documentation très complète sur docs.djangoproject.com. Bien que celle-ci soit en anglais, elle reste **très accessible** pour des francophones. Nous ne pouvons que vous conseiller de la lire en parallèle de ce tutoriel si vous voulez approfondir un certain sujet ou si certaines zones d'ombres persistent.

Enfin, pour gagner encore plus de temps, les utilisateurs de Django ont généralement l'esprit *open source* et fournissent une liste de *snippets*, des portions de codes réutilisables par n'importe qui. Un site est dédié à ces snippets : [site officiel](#). Si vous devez vous attaquer à une grosse application ou à une portion de code particulièrement difficile, n'hésitez pas à aller chercher dans les snippets, vous trouverez souvent votre bonheur là-dedans !



Téléchargement et installation

Maintenant que nous avons vu l'intérêt de Django, il est temps de **l'installer sur votre machine**. Tout d'abord, assurez-vous que vous disposez bien d'une version de Python de la branche 2.x et supérieur ou égale à la version 2.5. Autrement dit, les seules versions de Python compatibles avec Django 1.4 sont les versions 2.5, 2.6 et 2.7.



La support de Python 3.x est en cours de développement. Cependant, l'utilisation en production n'est pas conseillé avant quelques versions encore. Le support de Python 2.5 sera d'ailleurs abandonné à partir de Django 1.5. Nous vous conseillons dès maintenant d'utiliser Python 2.7, qui est bien plus stable et à jour.



Il est également plus prudent de **supprimer toutes les anciennes installations de Django**, si vous en avez déjà. Il peut y avoir des conflits entre les versions, notamment lors de la gestion des projets. Il est essentiel de n'avoir que Django **1.4** sur votre machine, à part si vous avez déjà des applications en production sur des versions antérieures.



La version 1.4.1 de Django est actuellement disponible. C'est une version mineure qui contient que des mises à jour de sécurité (il est cependant important de l'installer !). Ce cours a été écrit pour toute la branche 1.4 et non seulement 1.4.0



Linux et Mac OS

Sous Linux et Mac OS, l'installation de Django peut s'effectuer de deux manières différentes, soit en utilisant le gestionnaire de paquets de votre distribution (ou MacPorts pour Mac OS), soit en installant Django manuellement, via une archive officielle. Nous ne couvrirons pas la première solution, celle-ci dépendant beaucoup trop de votre distribution. En revanche, si vous choisissez celle-ci, **faites attention à la version** de Django disponible dans les dépôts. Il se peut que ce ne soit pas toujours la dernière version qui soit disponible, et que celle-ci ne soit donc pas à jour, et incompatible avec ce cours.

Si vous ne passez pas par les dépôts, le plus simple reste de télécharger une archive à cette adresse :

<https://www.djangoproject.com/download/> Il suffit ensuite de l'extraire et de l'installer, en effectuant les commandes suivantes dans une console :

Code : Console

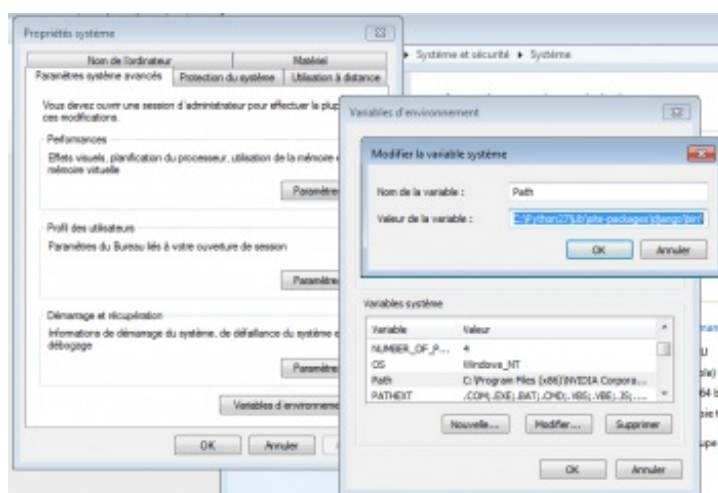
```
tar xzvf Django-1.4.tar.gz
cd Django-1.4
sudo python setup.py install
```

Windows

Contrairement aux environnement UNIX, l'installation de Django sous Windows requiert quelques manipulations supplémentaires.

Téléchargez l'archive de Django (disponible ici : <https://www.djangoproject.com/download/>) et extrayez là. Avant de continuer, nous allons devoir modifier quelques variables d'environnement, afin de permettre l'installation du framework. Pour cela (sous Windows 7) :

1. Rendez-vous dans les informations générales du système (via le raccourci Windows + Pause),
2. Cliquez sur **Paramètres système avancés**, dans le menu de gauche.
3. Une fois la fenêtre ouverte, cliquez sur **Variables d'environnement**
4. Cherchez la variable système (deuxième liste) « **Path** » et ajoutez ceci en fin de ligne (faites attention à votre version de Python) :
`;C:\Python27\;C:\Python27\Lib\site-packages\django\bin\`. Respectez bien le point virgule, permettant de séparer le répertoire de ceux déjà présents



Edition du Path sous Windows

Validez, puis quittez. On peut désormais installer Django via la console Windows (Windows + R puis la commande `cmd`) :

Code : Console

```
cd C:\Mon\repertoire\django1.4
python setup.py install
```


Les fichiers sont ensuite copiés dans votre dossier d'installation Python (ici C:\Python27).

Vérification de l'installation

Dès que vous avez terminé l'installation de Django, lancez une console Python (via la commande `python`) et tapez les deux commandes suivantes :

Code : Python

```
>>> import django
>>> print django.get_version()
1.4.2
```

Si vous obtenez également 1.4.2 comme réponse également, félicitations, **vous avez correctement installé Django !**



Dans la suite de ce cours, nous utiliserons **SQLite**, qui est simple et déjà inclus dans les librairies de base de Python. Si vous souhaitez utiliser un autre système de gestion de base de données, n'oubliez pas d'installer les outils nécessaires.

On vient de voir ce qu'était Django, et maintenant on est prêt à travailler ! Enfin, pas tout à fait...

Avant de pouvoir réaliser nos premières applications, il va falloir voir comment le framework est structuré, et avec quelle logique.

Le fonctionnement de Django

Abordons enfin le vif du sujet ! Dans ce chapitre, nous allons voir comment un projet Django fonctionne et comment les éléments d'une application classique s'articulent autour du modèle MVT, que nous introduirons également.



Ce chapitre est essentiellement théorique, mais fondamental pour comprendre correctement comment Django opère.

Un peu de théorie : l'architecture MVC

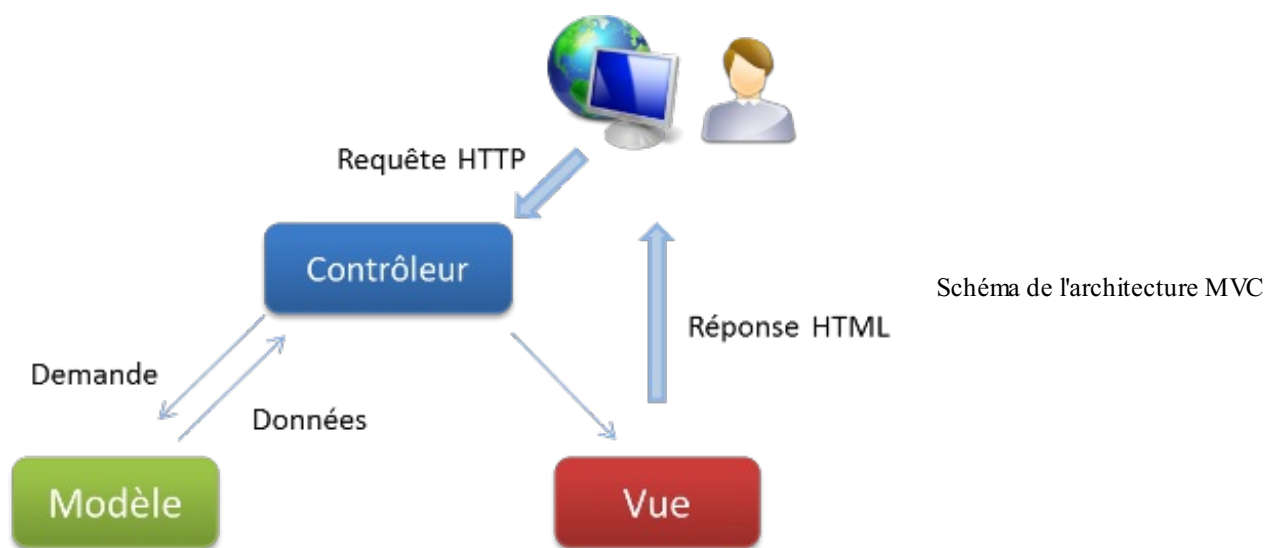
Lorsque l'on parle de frameworks qui fournissent une interface graphique à l'utilisateur (soit une page web, comme ici avec Django, soit l'interface d'une application graphique classique, comme celle de votre traitement de texte par exemple), on parle souvent de l'**architecture MVC**. Il s'agit d'un modèle distinguant plusieurs rôles précis d'une application qui doivent être accomplis.

Comme son nom l'indique, l'architecture (ou « patron ») Modèle-Vue-Contrôleur est composée 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. Il permet d'accéder à l'information, de la modifier, d'en ajouter une nouvelle, de vérifier que celle-ci correspond bien aux critères (on parle d'intégrité de l'information), de la mettre à jour, etc. Il s'agit d'une interface supplémentaire entre votre code et la base de données, mais qui simplifie grandement les choses.

Ensuite, la **vue**, est, comme son nom l'indique, la **visualisation de l'information**. C'est la seule chose que l'utilisateur peut voir. Non seulement elle sert à présenter une donnée, mais elle permet aussi de **recueillir une éventuelle action** de l'utilisateur (un clic sur un lien, ou la soumission d'un formulaire par exemple). Typiquement, une vue correspond à une page web par exemple.

Finalement, le **contrôleur** **prend en charge tous les événements de l'utilisateur** (accès à une page, soumission d'un formulaire, ...). Il se charge, en fonction de la requête de l'utilisateur, de récupérer 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. Lors de l'appel d'une page, c'est le contrôleur qui est chargé en premier, afin savoir ce qu'il est nécessaire d'afficher.



La spécificité de Django : 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, etc). Ainsi, on parle plutôt de framework utilisant l'architecture MVT : *Modèle-Vue-Template*.

Cette architecture reprend les définitions de modèle et de vues que nous avons vues, et en introduit une nouvelle : le *template*. Un template est un **fichier HTML**, aussi appelé "gabarit". Il sera récupéré par la vue et envoyé au visiteur, cependant, avant d'être envoyé, le template sera analysé et exécuté par le framework, comme s'il s'agissait d'un fichier avec du code.

Django fournit un *moteur de templates* très utile qui permet, dans le code HTML, d'afficher des variables, utiliser des structures conditionnelles (if / else) ou encore des boucles (for), etc.

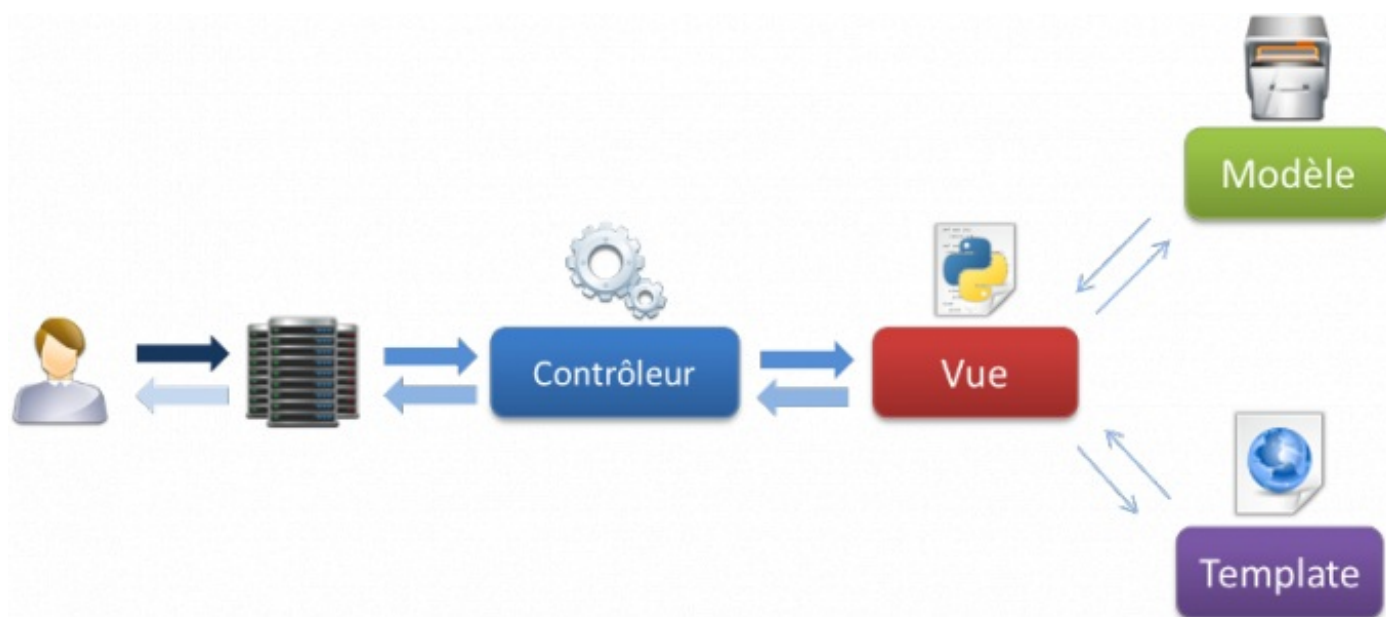


Schéma d'exécution d'une requête

Concrètement, lorsque l'internaute appelle une page de votre site réalisé avec Django (par exemple <http://www.crepes-bretonnes.com/blog/>), le framework 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 (la page HTML correspondant au lien "blog") au navigateur de l'internaute.

On distingue les 4 parties distinctes 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...), les modèles;
- L'affichage de ces données et de toutes autres informations au format HTML, les templates;
- Enfin le lien entre les 2 derniers points : la vue qui récupère les données et génère le template selon celles-ci.

On en revient donc au **modèle MVT**. Le développeur se doit de fournir le modèle, la vue et le template. Une fois ceci fait, il suffit juste d'assigner la vue à une URL précise, et la page est accessible.

Si le template est un fichier HTML classique, un **modèle** en revanche sera écrit sous la forme d'une **classe** où chaque attribut de la classe correspondra à un champ dans la base de données. Django se chargera ensuite de créer la table correspondante dans la base de données, et de faire la liaison entre la base de données et les objets de votre classe. Non seulement, il n'y a plus besoin d'écrire de requêtes SQL, mais en plus, le framework propose la représentation de chaque entrée de la table sous forme d'une instance de la classe qui a été écrite. Il suffit donc d'accéder et de modifier les attributs de la classe pour accéder et modifier les éléments dans la table, ce qui est très pratique !

Enfin, **une vue est une simple fonction**, qui prend comme paramètres des informations sur la requête (s'il s'agit d'une requête GET ou POST par exemple), et les paramètres qui ont été données dans l'URL. Par exemple, si l'identifiant ou nom d'un article du blog a été donné dans l'URL `crepes-bretonnes.com/blog/faire-des-bonnes-crepes`, la vue récupérera "*faire-de-bonnes-crepes*" comme titre et cherchera dans la base de données l'article correspondant à afficher. Suite à quoi la vue générera le template avec le bon article et le renverra à l'utilisateur.

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 tâche 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.

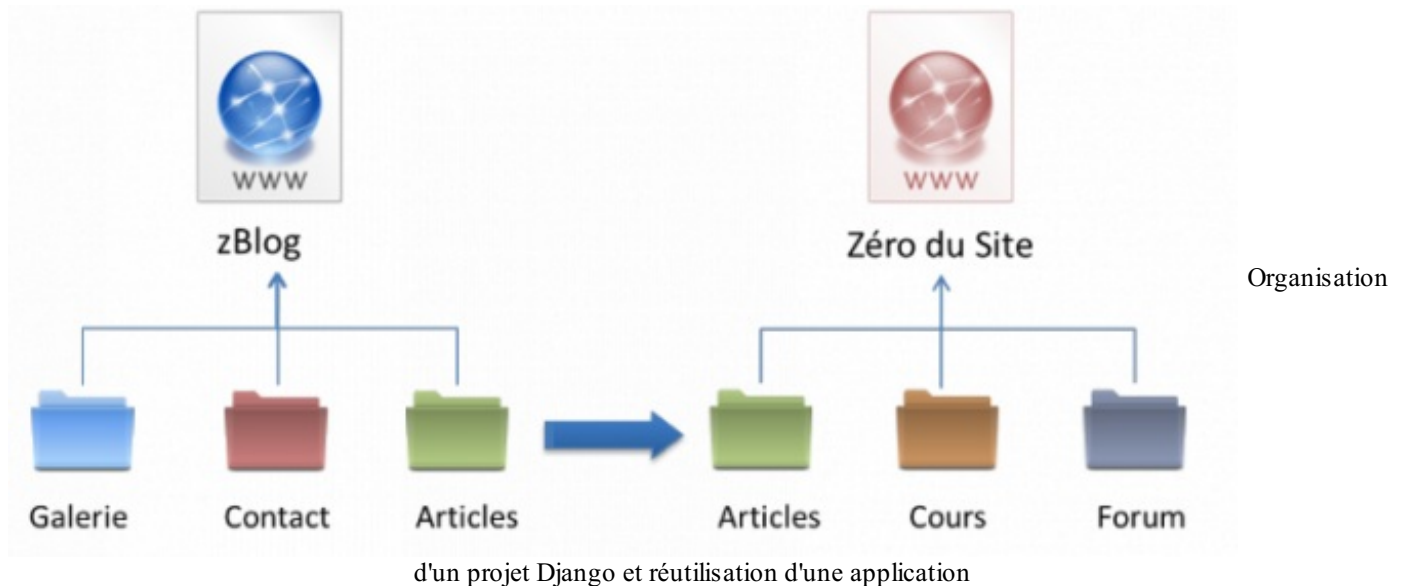
Lors de la conception de votre site, vous allez devoir penser aux applications que vous souhaitez développer. Voici quelques exemples d'applications :

- Module d'actualités
- Forum
- Contact
- Galerie photo

- Système de dons

Ce principe de séparation du projet en plusieurs applications possède deux avantages principaux :

- Le **code** est **beaucoup plus structuré**. Les modèles et templates d'une application ne seront que rarement ou jamais utilisés dans une autre, on garde donc une séparation nette entre les différents applications, ce qui évite de s'emmêler les pinceaux !
- Une application correctement conçue **pourra être réutilisée** dans d'autres projets très simplement, par un simple copié/collé.



Ici, le développement du système d'articles sera fait une fois uniquement. Pour le second site, une légère retouche des templates suffira. Ce système permet de voir le site web comme des boîtes que l'on agence ensemble, accélérant considérablement le développement pour les projets qui suivent.

La théorie est terminée, on peut passer à quelque chose de plus pratique : la gestion d'un projet Django et ses applications.

Gestion d'un projet

Django propose un outil en ligne de commande très utile qui permet énormément de choses :

- création de projets et applications;
- création des tables dans la base de données selon les modèles de l'application;
- lancement du serveur web de développement;
- etc.

Nous verrons dans ce chapitre comment utiliser cet outil, la structure d'un projet Django classique, comment créer ses projets et applications, et leur configuration.

Créons notre premier projet

L'outil fourni avec Django se nomme **django-admin.py**. Comme dit plus haut, celui-ci n'est accessible qu'en ligne de commande. Pour ce faire, munissez-vous d'une console MS-DOS sous Windows, ou d'un terminal sous Linux et Mac OS X.



Attention ! La console système n'est pas l'interpréteur Python ! Dans la console système, vous pouvez exécuter des commandes systèmes comme l'ajout de dossier, de fichier, tandis que dans l'interpréteur Python, vous écrivez du code Python.

Sous Windows, allez dans le menu démarrer, puis Exécuter et tapez dans l'invite de commande `cmd`. Une console s'ouvre, déplacez-vous dans le dossier dans lequel vous souhaitez créer votre projet grâce à la commande `cd`, suivi d'un chemin. Exemple :

Code : Console

```
cd C:\Mes Documents\Utilisateur\
```

Sous Mac OS X et Linux, lancez tout simplement l'application Terminal (elle peut parfois également être nommée Console sous Linux), et déplacez-vous dans le dossier dans lequel vous souhaitez créer votre projet, également à l'aide de la commande `cd`. Exemple :

Code : Console

```
cd /home/user/
```

Tout au long du tutoriel, nous utiliserons un blog sur les bonnes crêpes bretonnes comme exemple. Ainsi, appelons notre projet "crepes_bretonnes" (seuls les caractères alphanumériques et underscores sont autorisés pour le nom du projet) et créons-le grâce à la commande suivante :

Code : Console

```
django-admin.py startproject crepes_bretonnes
```

Un nouveau dossier nommé "crepes_bretonnes" est apparu et possède la structure suivante. Il s'agit de votre projet.

Code : Autre

```
crepes_bretonnes/  
  manage.py  
  crepes_bretonnes/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

Dans le dossier `crepes_bretonnes` principal, on retrouve deux éléments : un fichier `manage.py` et un autre sous-dossier nommé également `crepes_bretonnes`.

Créez dans le dossier principal un dossier nommé “`templates`”, lequel contiendra vos templates HTML. Nous y reviendrons plus tard.

Le sous-dossier contient 4 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 éloquentes : `settings.py` contiendra la configuration de votre projet, tandis que `urls.py` rassemblera toutes les URLs de votre site web et la liste des fonctions à appeler pour chaque URL. Nous reviendrons à ces deux fichiers plus tard.

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

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

Code : Console

```
$ python manage.py runserver
Validating models...

0 errors found
Django version 1.4, using settings 'crepes_bretonnes.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Cette console va vous donner des informations telles que les logs de type Apache (page accédée et par qui) et les exceptions Python lancées en cas d'erreur lors du développement. Par défaut, l'accès au site de développement se fait via l'adresse <http://localhost:8000>. Vous devriez obtenir quelque chose comme ceci dans votre navigateur :

It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Here's what to do next:

- If you plan to use a database, edit the `DATABASE_*` settings in `wikishot/settings.py`.
- Start your first app by running `python wikishot/manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Si ce n’est pas le cas, assurez-vous d’avoir bien respecté toutes les étapes susmentionnées !

Au passage, `manage.py` propose bien d’autres sous-commands, autres que `runserver`. Une petite liste est fournie avec la sous-commande `help` :

Code : Console

```
python manage.py help
```

Nous reviendrons à toutes ces commandes plus tard, c’était juste pour souligner le fait que cet outil est très puissant et

développeur Django y a recours quasi en permanence, d'où l'intérêt de savoir le manier correctement.

Configurez votre projet

Avant de commencer à écrire des applications Django, configurons notre projet. Ouvrez le fichier `settings.py` dont nous avons parlé tout à l'heure. Il s'agit d'un simple fichier Python avec une liste de variables que vous pouvez modifier à votre guise. Voici la liste des plus importantes :

Code : Python

```
DEBUG = True
TEMPLATE_DEBUG = DEBUG
```

Ces deux variables permettent d'indiquer si votre site web est en mode « DEBUG » ou pas. Le mode *debug* affiche des informations pour déboguer vos applications en cas d'erreur. Ces informations affichées peuvent contenir des données sensibles de votre fichier de configuration. Ne mettez donc jamais `DEBUG = True` en production !

Le tuple `ADMINS` contient quelques informations à propos des gestionnaires du site (nom et adresse mail). L'adresse mail servira notamment à envoyer les erreurs de votre application en production rencontrées par les visiteurs de votre site.

Code : Python

```
ADMINS = (
    ('Maxime Lorant', 'maxime@crepes-bretonnes.com'),
    ('Mathieu Xhonneux', 'mathieu@crepes-bretonnes.com'),
)
```

La configuration des bases de données se fait dans le dictionnaire `DATABASES`. Nous conseillons pour le développement local l'utilisation d'une base de donnée SQLite. L'avantage de SQLite comme gestionnaire de BDD 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 comme MySQL, Python et Django se chargent de tout. Nous couvrirons la configuration de Django avec MySQL et PostgreSQL en production dans une annexe.

Voici la configuration nécessaire pour l'utilisation de SQLite :

Code : Python

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': 'database.sql',
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

Modifiez le fuseau horaire et la langue de l'administration :

Code : Python

```
TIME_ZONE = 'Europe/Paris'
LANGUAGE_CODE = 'fr-FR'
```

`TEMPLATE_DIRS` est un simple tuple contenant les listes des dossiers vers les templates. Nous avons créé un dossier "templates" à la racine de notre projet tout à l'heure, incluons-le donc ici :

Code : Python

```
TEMPLATE_DIRS = (  
    "/home/crepes/crepes_bretonnes/templates/"  
)
```

Enfin, pour des raisons pratiques par la suite, nous allons rajouter une option qui permet de compléter automatiquement les URL par un slash à la fin, si il n'est pas déjà présent. Vous comprendrez l'utilité de cette partie lorsque l'on parlera de routage d'URL plus loin 😊

Code : Python

```
APPEND_SLASH = True # Ajoute un slash en fin d'URL
```

Voilà ! Les variables les plus importantes ont été expliquées ci-dessus. Pour que ce ne soit pas indigeste, nous n'avons pas tout expliqué, il en reste en effet beaucoup d'autres. Nous reviendrons sur certains paramètres plus tard. En attendant, si une variable vous intrigue, n'hésitez pas à lire le commentaire (bien qu'en anglais) à coté de la déclaration et à vous référer à la documentation en ligne.

Créons notre première application

Comme nous l'avons expliqué dans la partie précédente, un projet se compose de plusieurs applications, chacune ayant un but bien précis (module de news, forums, ...)

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` avec `startapp`, à l'intérieur de votre projet. Pour notre site sur les crêpes bretonnes, créons un blog pour publier nos nouvelles recettes :

Code : Console

```
python manage.py startapp blog
```

Comme pour `startproject`, `startapp` crée un dossier avec plusieurs fichiers à l'intérieur. La structure de notre projet ressemble à ceci :

Code : Autre

```
crepes_bretonnes/  
    manage.py  
    crepes_bretonnes/  
        __init__.py  
        settings.py  
        urls.py  
        wsgi.py  
    blog/  
        __init__.py  
        models.py  
        tests.py  
        views.py
```

Les noms des fichiers sont relativement évidents :

- `models.py` contiendra vos modèles ;
- `tests.py` permet la création de tests unitaires (nous y reviendrons plus tard) ;
- `views.py` contiendra toutes les vues de votre application ;



A partir de maintenant, on omet de parler des fichiers `__init__.py`, qui sont là que pour indiquer que notre dossier est un module Python. C'est une spécificité de Python qui ne concerne pas directement Django.

Dernière petite chose, il faut rajouter 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, rajoutez une chaîne de caractères avec le nom de votre application. Au passage, décommentez l'application `django.contrib.admin`, il s'agit de l'application qui génère automatiquement l'administration et dont nous nous occuperons plus tard.

Votre variable devrait ressembler à quelque chose comme ceci :

Code : Python

```
INSTALLED_APPS = (  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.sites',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'django.contrib.admin',  
    'blog',  
)
```

Les préliminaires sont terminés ! Nous avons introduit Django dans cette partie, on peut désormais passer au développement d'une application.

Les bases de données et Django

Pour que vous puissiez enregistrer les données de vos visiteurs, l'utilisation d'une base de données s'impose. Nous allons dans ce chapitre expliquer le fonctionnement d'une BDD, le principe des requêtes SQL et l'interface que Django propose entre les vues et les données enregistrées. À la fin de ce chapitre, vous aurez assez de connaissances théoriques pour comprendre par la suite le fonctionnement des modèles.

Une base de données, c'est quoi ?

Imaginez que vous souhaitiez classer sur papier la liste des films que vous possédez à la maison. Un film a plusieurs **caractéristiques** : le titre, le résumé, le réalisateur, les acteurs principaux, le genre, l'année de sortie, appréciation, etc. Il est important que votre méthode de classement permette de différencier très proprement ces caractéristiques. De même, vous devez être sûr que les caractéristiques que vous écrivez sont correctes et homogènes. Si vous écrivez, la date de sortie une fois en utilisant des chiffres, et puis en utilisant des lettres, vous perdez en lisibilité et risquez de compliquer les choses.

Il existe plusieurs méthodes de classement pour trier nos films, mais la plus simple et la plus efficace (et à laquelle vous avez sûrement dû penser) est tout simplement **un tableau**. Pour classer nos films, les colonnes du tableau renseigneraient les différentes caractéristiques qu'un film a, tandis que les lignes représenteront toutes les caractéristiques d'un seul film. Par exemple :

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

Le classement par tableau est très pratique et simple à comprendre. Les bases de données se basent donc sur cette méthode de tri pour enregistrer et classer les informations que vous lui spécifierez.

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



En anglais, on traduit « tableau » 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.

Nous avons évoqué un autre point important de ces bases de données, avec l'exemple de la date de sortie. Il faut en effet que toutes les données dans une colonne soient homogènes. Autrement dit, elles doivent appartenir à un **même type de données** : un entier, une chaîne de caractères, un texte, un booléen, une date, ... Si vous enregistrez un texte dans la colonne "Note", votre code vous renverra une erreur. Dès lors, chaque fois que vous irez chercher des données dans une table, vous serez sûr quant au type des variables que vous obtiendrez.

Le langage SQL et les gestionnaires de base de données

Il existe plusieurs programmes qui s'occupent de gérer des bases de données. On les appelle, tout naturellement, des gestionnaires de bases de données (ou *SGBD*). Ces derniers s'occupent de tout : création de nouvelles tables, rajout 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, elles 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).

Les gestionnaires de bases de données SQL sont les plus populaires et les plus utilisées 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 un simple exemple de requête SQL qui renvoie toutes les entrées de la table "films" dont le réalisateur doit être "Quentin Tarantino" et sont triées par date de sortie :

Code : SQL

```
SELECT titre, annee_sortie, note FROM films WHERE
realisateur="Quentin Tarantino" ORDER BY annee_sortie
```

On a déjà vu plus simple, mais voilà comment communiquent un serveur SQL et un client. Il existe bien d'autres commandes (une pour chaque type de requête : sélection, mise à jour, suppression, ...) et chaque commande possède ses paramètres.

Heureusement, tous les **SGBD SQL parlent à peu près le même SQL**, c'est à dire qu'une requête utilisée avec un gestionnaire fonctionnerait également avec un autre. Néanmoins, ce point est assez théorique, car même si les requêtes assez basiques marchent à peu près partout, les requêtes plus pointues et avancées commencent à diverger selon le SGBD, et si un jour vous devez changer de gestionnaire, nul doute que vous devrez réécrire certaines requêtes. Django a une solution pour ce genre de situations, nous verrons cela plus loin.

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

- MySQL : probablement le plus connu et le plus utilisé à travers le monde (gratuit)
- PostgreSQL : moins connu que MySQL, mais a quelques fonctionnalités de plus que ce dernier (gratuit)
- Oracle Database : généralement utilisé dans de grosses entreprises, une version gratuite existe, mais 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)

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 librairie Python (nommée `sqlite3`) 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 MySQL.

La magie des ORM

Apprendre le langage SQL et devoir écrire ses propres requêtes est quelque chose d'assez difficile et contraignant lorsqu'on débute. Cela prend beaucoup de temps et 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 surfait 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.

Sans rentrer dans les détails (nous verrons cela après), voici un modèle simple qui reviendra par la suite :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField(null=True)
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose_name="Date de parution")
```

A partir de ce modèle, Django va créer une table nommée "*blog_article*" (blog étant le nom de l'application dans laquelle le modèle est ajouté) dont les champs seront titre, auteur, contenu et date. Chaque champ a **son propre type** (tel que défini dans le modèle), et ses propres paramètres. Tout cela se fait, encore une fois, sans écrire la moindre requête SQL.

La manipulation de données est tout aussi simple bien évidemment :

Code : Python

```
Article(titre="Bonjour", auteur="Maxime", contenu="Salut").save()
```

... créera une nouvelle entrée dans la base de données. Notez la relation qui se crée : chaque instance du modèle Article qui se crée correspond à une entrée dans la table SQL. **Toute manipulation des données dans la base se fait depuis des objets Python**,

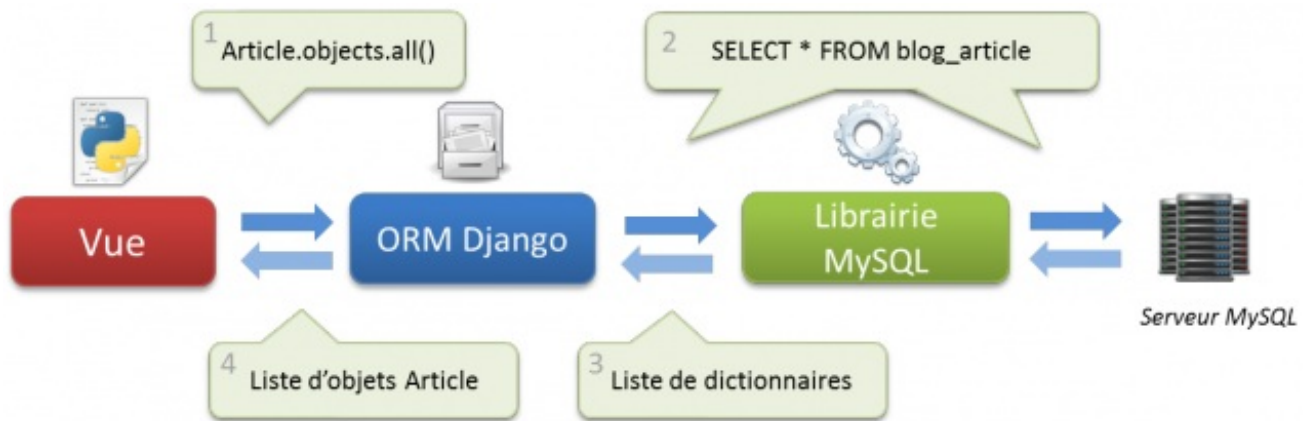
ce qui est bien plus intuitif et simple.

De la même façon, il est possible d'obtenir toutes les entrées de la table :

Code : Python

```
Article.objects.all()
```

... renverra des instances d'Article, une pour chaque entrée dans la table, encore une fois.



Pour conclure, l'ORM est un **système très flexible** de Django qui s'insère parfaitement bien dans l'architecture MVT que nous avons décrite précédemment.

Le principe des clés étrangères

Pour terminer ce chapitre, nous allons aborder une dernière notion théorique relative aux bases de données SQL, il s'agit des clés étrangères (ou *Foreign Key* en anglais).

Reprenons notre exemple de tout à l'heure : nous avons une table qui recense plusieurs films. Imaginons maintenant que nous souhaitons rajouter des données supplémentaires, qui ne concernent pas les films, mais ses réalisateurs. Nous voudrions par exemple rajouter le pays d'origine et la date de naissance des réalisateurs. Étant donné que certains réalisateurs reviennent plusieurs fois, il serait redondant de rajouter les caractéristiques des réalisateurs dans les caractéristiques des films. La bonne solution ? Créer une nouvelle table qui recensera les réalisateurs et **rajouter un lien** entre les films et les réalisateurs.

Lorsque Django crée une nouvelle table depuis un modèle, il va ajouter un autre champ qui n'est pas dans les attributs de la classe. Il s'agit d'un champ tout simple nommé **id** (pour « identifiant », synonyme ici de « clé »), qui contiendra un certain nombre unique à l'entrée, et qui est croissant au fil des entrées. Ainsi, le premier réalisateur ajouté aura l'identifiant 1, le deuxième l'identifiant 2, etc etc.

Voici donc à quoi ressemblerait notre table des réalisateurs :

ID	Nom	Pays d'origine	Date de naissance
1	Quentin Tarantino	USA	1963
2	David Fincher	USA	1962
3	Monty Python	Grande Bretagne	1969

Jusqu'ici, rien de spécial à part la nouvelle colonne « ID » introduite précédemment. En revanche, dans la table recensant les films, ça bouge un peu plus :

Titre	Réalisateur	Année de sortie	Note (sur 20)
Pulp Fiction	1	1994	20
Inglorious Basterds	1	2009	18
Holy Grail	3	1975	19
Fight Club	2	1999	20

Life of Brian	3	1979	17
---------------	---	------	----

Désormais, les noms des réalisateurs sont remplacés par des nombres. Ceux-ci correspondent aux **identifiants** de la table des réalisateurs. Si on veut obtenir le réalisateur du film « Fight Club », il faut aller regarder dans la table réalisateurs et sélectionner l'entrée ayant l'identifiant n° 2. On peut dès lors regarder le nom du réalisateur, on obtient bien à nouveau « David Fincher », et les données supplémentaires (date de naissance, pays d'origine), sont également accessibles.

Cette méthode de clé étrangère (car la clé vient d'une autre table) permet de créer simplement des liens entre des entrées dans différents tableaux. L'ORM de Django gère parfaitement cette méthode. Vous n'aurez probablement jamais besoin de l'identifiant pour gérer des liaisons, Django s'en occupera et renverra directement l'objet de l'entrée associée.

Partie 2 : Premiers pas

Dans cette partie, les bases du framework vous seront expliquées pas à pas. A la fin de celle-ci, vous serez capable de réaliser par vous-même un site basique avec Django !

Votre première page grâce aux vues

Dans cette partie, nous allons créer notre première page web avec Django. Nous verrons comment créer une vue dans une application, et rendre celle-ci accessible depuis une URL.

Hello World !

Commençons enfin notre blog sur les bonnes crêpes bretonnes ! Rappelez-vous au chapitre précédent, nous avons créé une application "blog" dans notre projet.

Comme vu dans la théorie, **chaque URL est associée à une vue**. Avec Django, une vue est représentée par une fonction définie dans le fichier `views.py`. Cette fonction va généralement récupérer des données dans les modèles (ce que l'on verra plus tard) et appeler le bon template pour générer le rendu HTML adéquat. Par exemple, on pourrait donner la liste des 10 derniers articles de notre blog au moteur de templates, qui se chargera de les insérer dans une page HTML finale, qui sera renvoyée à l'utilisateur.

Pour débiter, nous allons faire quelque chose de relativement simple : une page qui affichera "Bienvenue sur mon blog !".

La gestion des vues

Chaque application possède **son propre fichier `views.py`**, regroupant l'ensemble des fonctions que nous avons introduites ci-dessus. Comme tout bon blog, le nôtre possèdera plusieurs vues qui rempliront diverses tâches :

- L'index, l'accueil du blog
- La vue d'un article particulier
- Liste des articles par mois
- L'ajout de commentaires
- ...

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

Code : Python

```
# Create your views here.
```

Pour éviter tout problème par la suite, indiquons à l'interpréteur Python que le fichier sera en UTF-8, afin de prendre en charge les accents. En effet, Django gère totalement l'UTF-8 et il serait bien dommage de ne pas l'utiliser.

Insérez comme première ligne de code du fichier ceci :

Code : Python

```
# -*- coding: utf-8 -*-
```

Désormais, nous pouvons créer une fonction qui remplira le rôle de la vue. Bien que nous n'ayons vu pour le moment ni les modèles, ni les templates, il est tout de même possible d'écrire une vue, mais celle-ci restera basique. En effet, il est possible d'écrire du code HTML directement dans la vue, et de le renvoyer au client :

Code : Python

```
# -*- coding: utf-8 -*-  
from django.http import HttpResponse  
  
def home(request):  
    text = """<h1>Bienvenue sur mon blog !</h1>
```



```
<p>Les crêpes bretonnes ça tue des mouettes en plein vol !</p>""
return HttpResponse(text)
```

Ce code se divise en trois étapes :

- On importe la classe `HttpResponse` du module `django.http`. Cette classe permet de retourner une réponse (texte brut, JSON 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` a été déclarée, avec comme argument une instance de `HttpRequest`. Nous avons nommé ici (et c'est presque partout le cas) 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 déclare une chaîne de caractères nommée `text` et crée une nouvelle instance de `HttpResponse` à partir de cette chaîne, 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 on le fait ici. Passez toujours par des templates, ce que nous introduirons au chapitre suivant. Il s'agit de respecter l'architecture du framework dont nous avons parlé dans la partie précédente afin de bénéficier de ses avantages (la structuration du code notamment). Nous n'avons utilisé cette méthode que dans un **but pédagogique** et afin de montrer les choses une par une.

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` de votre projet (ici `crepes_bretonnes/urls.py`). Par défaut, ce fichier contient une aide basique :

Code : Python

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
# from django.contrib import admin
# admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    # url(r'^$', 'crepes.views.home', name='home'),
    # url(r'^crepes/', include('crepes.foo.urls')),

    # Uncomment the admin/doc line below to enable admin
    # documentation:
    # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),

    # Uncomment the next line to enable the admin:
    # url(r'^admin/', include(admin.site.urls)),
)
```

Quand un utilisateur appelle une page de votre site, la requête est directement prise en charge par le contrôleur de Django qui va chercher à **quelle vue correspond cette URL**. En fonction de l'ordre de définition dans le fichier ci-dessus, la première vue qui correspond à l'URL demandée sera appelée, et elle retournera donc la réponse HTML au contrôleur (qui lui la retournera à l'utilisateur). Si aucune URL ne correspond à un schéma que vous avez défini, alors Django renverra une page d'erreur 404. En résumé, le schéma d'exécution est le suivant :

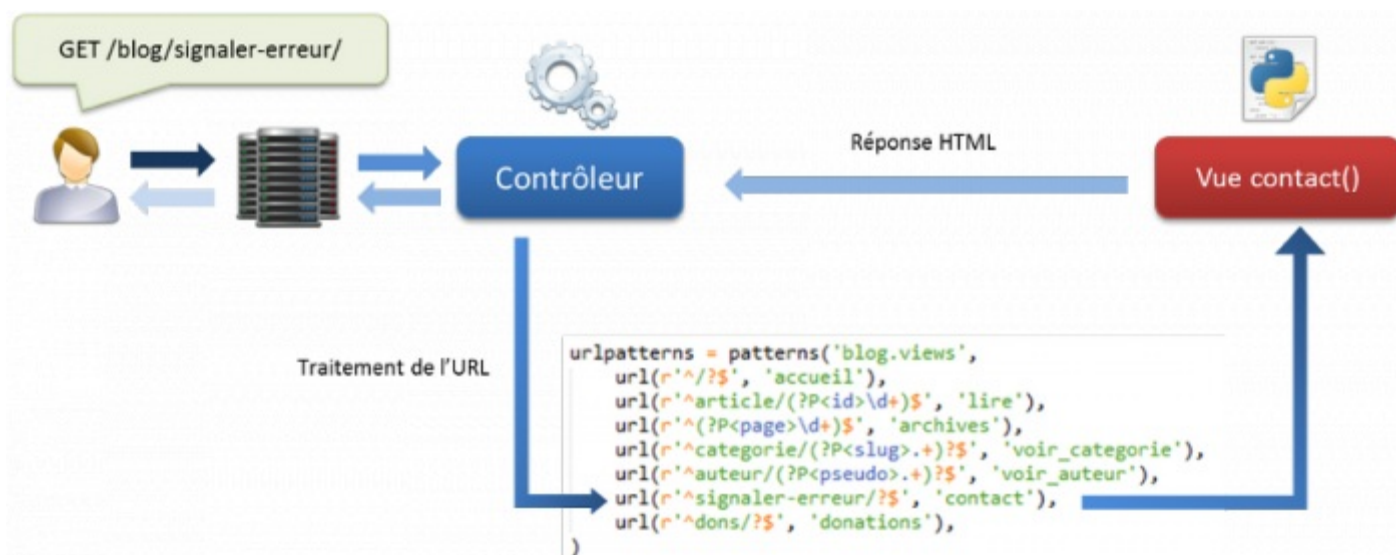


Schéma d'exécution d'une requête. Nous allons travailler pour le moment sans templates et sans modèles

Occupons-nous uniquement du tuple `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 pattern 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, ... ;
- Le chemin python vers la vue correspondante.

Par exemple, en reprenant la vue définie tout à l'heure, si l'on souhaite que celle-ci soit accessible depuis l'URL `http://www.crepes-bretonnes.com/accueil`, il suffit de rajouter cette règle dans votre `urlpatterns` :

Code : Python

```

urlpatterns = patterns('',
    url(r'^accueil/$', 'blog.views.home'),
)

```



Mettre `r'^$'` comme URL équivaut à spécifier la racine du site web. Autrement dit, si nous avons utilisé cette URL à la place de `r'^accueil/$'`, la vue serait accessible depuis `http://www.crepes-bretonnes.com/`.



Qu'est-ce que c'est tous ces caractères bizarres dans l'URL ?

Il s'agit d'expressions régulières (ou *regex*) qui permettent de créer des URL plus souples. Il est généralement conseillé de maîtriser au moins les bases des *regex* pour pouvoir écrire des URL correctes. Dans ce cas-ci :

- `^` indique le début de la chaîne (autrement dit, il ne peut rien y avoir avant */accueil*) ;
- `?` indique que le caractère précédent peut être absent ;
- `$` est le contraire de `^`, il indique la fin de la chaîne.

Bien évidemment, toute expression régulière compatible avec le module `re` de Python sera compatible ici aussi.



Si vous n'êtes pas assez à l'aise avec les *regex* (expressions régulières), nous vous conseillons de faire une pause et d'aller voir le chapitre "Les expressions régulières" du tutoriel Python officiel.

Concernant le lien vers la vue, il s'agit du même type de lien utilisé lors d'une importation de module. Ici :

- **blog** indique le module qui forme l'application « blog » ;
- **views** indique le fichier concerné du module ;
- **home** est la fonction du fichier `views.py`.

Grâce à cette règle, Django saura que lorsqu'un client demande la page `http://www.crepes-bretonnes.fr/accueil`, il devra appeler 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/`. Vous devriez obtenir quelque chose comme ceci :



Si c'est le cas, félicitations, vous venez de créer votre première vue !

Organiser proprement vos URLs

Dans la partie précédente, nous avons parlé de deux avantages importants de Django : la réutilisation d'applications et la structuration du code. Sauf qu'évidemment, un problème se pose avec l'utilisation des URLs que nous avons faite : si nous avons plusieurs applications, toutes les URLs de celles-ci iraient dans `urls.py` du projet, ce qui compliquerait nettement la réutilisation d'une application et ne structure en rien votre code.

En effet, il faudrait sans cesse recopier toutes les URLs d'une application en l'incluant dans un projet, et une application complexes peut avoir des dizaines d'URLs, ce qui ne facilite pas la tâche du développeur. Sans parler de la problématique qui survient lorsqu'il faut retrouver la bonne vue parmi la centaine déjà écrite. C'est pour cela qu'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` 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 ci-dessus (ne pas oublier l'importation des modules nécessaires !) :

Code : Python

```
from django.conf.urls import patterns, url

urlpatterns = patterns('',
    url(r'^accueil/$', 'blog.views.home'),
)
```

Et c'est déjà tout pour `blog/urls.py`.

Maintenant, retournons à `crepes bretonnes/urls.py`. On peut y enlever la règle réécrite dans `blog/urls.py`. Il ne

devrait donc plus rester grand chose.

L'importation des règles de `blogs/urls.py` est tout aussi simple, il suffit d'utiliser la fonction `include` de `django.conf.urls` et de rajouter ce sous-tuple à `urlpatterns` :

Code : Python

```
url(r'^blog/', include('blog.urls'))
```



En quoi consiste l'URL `^blog/` ici ?

Cette URL (en réalité portion d'URL), va précéder toutes les URLs incluses. Autrement dit, nous avons une URL `/accueil` qui envoyait vers la vue `blog.views.home`, désormais celle-ci sera accessible depuis `/blog/accueil`. Et cela vaut pour toutes les futures URLs importées.

Cependant, rien ne vous empêche de laisser cette string vide (`/accueil` restera `/accueil`), mais il s'agit d'une bonne solution pour structurer vos URLs.

Nous avons scindé nos URLs dans un fichier `urls.py` pour chaque application. Cependant, nous allons bientôt rajouter d'autres URLs plus complexes dans notre `blog/urls.py`. Toutes ces URLs seront routées vers des vues de `blog.views`. Au final, la variable `urlpatterns` de notre `blog/urls.py` risque de devenir longue :

Code : Python

```
urlpatterns = patterns('',
    url(r'^accueil/$', 'blog.views.home'),
    url(r'^truc/$', 'blog.views.truc'),
    url(r'^chose/$', 'blog.views.chose'),
    url(r'^machin/$', 'blog.views.machin'),
    url(r'^foo/$', 'blog.views.foo'),
    url(r'^bar/$', 'blog.views.bar'),
)
```

Maintenant, imaginez que votre application `blog` change de nom, vous allez devoir réécrire tous les chemins vers vos vues ! Pour éviter de devoir modifier toutes les règles une à une, il est possible de spécifier un module par défaut qui contient toutes les vues. Pour ce faire, il faut utiliser le premier élément de notre tuple qui est resté une chaîne de caractères vide jusque maintenant :

Code : Python

```
urlpatterns = patterns('blog.views',
    url(r'^accueil/$', 'home'),
    url(r'^truc/$', 'truc'),
    url(r'^chose/$', 'chose'),
    url(r'^machin/$', 'machin'),
    url(r'^foo/$', 'foo'),
    url(r'^bar/$', 'bar'),
)
```

Tout est beaucoup plus simple et facilement éditable. Le module par défaut ici est `blog.views` car toutes les vues viennent de ce fichier-là, ce qui est désormais possible car nous avons scindé notre `urls.py` principal en plusieurs `urls.py` propres à chaque application.

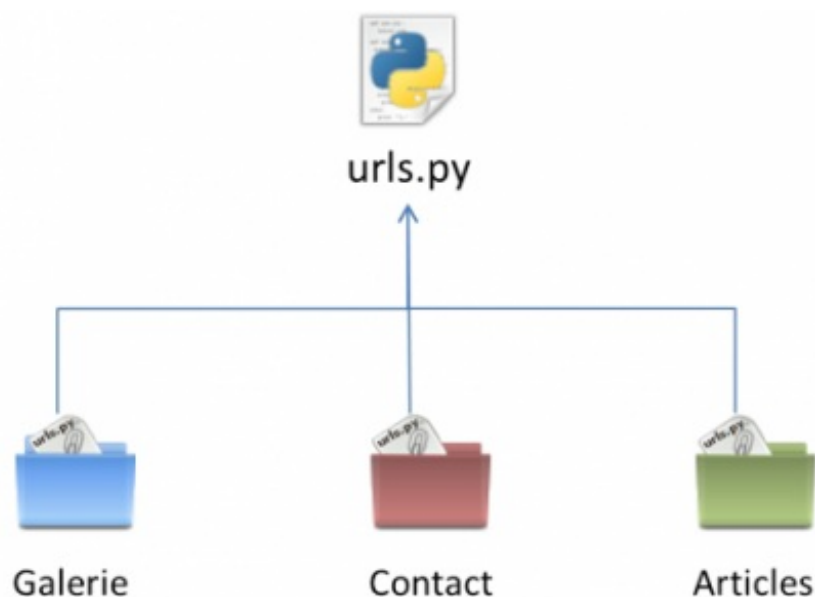
Finalement, notre `blog/urls.py` ressemblera à ceci :

Code : Python

```
from django.conf.urls import patterns, url

urlpatterns = patterns('blog.views',
    url(r'^accueil/$', 'home'),
)
```

Ne négligez pas cette solution, utilisez-la, dès maintenant ! Il s'agit d'une excellente méthode pour structurer votre code, parmi tant d'autres que Django offre. Pensez aux éventuels développeurs qui pourraient maintenir votre projet après vous et qui n'ont pas envie de se retrouver avec une structure proche de l'anarchie.



Passer des arguments à vos vues

Nous avons vu comment lier des URLs à des vues et comment les organiser. Cependant, un besoin va bientôt se faire ressentir : **pouvoir passer des paramètres dans nos adresses** directement. Si vous observez les adresses d'*Instagram* (qui est basé sur Django pour rappel), le lien vers une photo est construit ainsi : `http://instagram/p/*****` où `*****` est une suite de caractères alphanumériques. Cette suite représente en réalité l'identifiant de la photo sur le site et permet à la vue de récupérer les informations en relation avec cette photo.

Pour passer des arguments dans une URL, il suffit de capturer ces arguments directement depuis les *regex*. Par exemple, si l'on souhaite via notre blog pouvoir accéder à un certain article via l'adresse : `/article/**` où `**` sera l'identifiant de l'article (un nombre unique), il suffit de fournir le routage suivant dans votre `urls.py` :

Code : Python

```
urlpatterns = patterns('blog.views',
    url(r'^accueil/$', 'home'), # Accueil du blog
    url(r'^article/(\d+)/$', 'view_article'), # Vue d'un article
    url(r'^articles/(\d{4})/(\d{2})/$', 'list_articles'), # Vue des
    articles d'un mois précis
)
```

Quand l'URL `/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)`. Voici un exemple d'implémentation :

Code : Python

```
def view_article(request, id_article):
    """ Vue qui affiche un article selon son identifiant (ou ID,
    ici un numéro). Son ID est le second paramètre de la fonction (pour
    rappel,
    le premier paramètre est TOUJOURS la requête de l'utilisateur) """
    text = "Vous avez demandé l'article n°{0} !".format(id_article)
    return HttpResponse(text)
```

Il faut cependant faire attention à l'ordre des paramètres dans l'URL afin qu'il corresponde à l'ordre des paramètres de la fonction. En effet, lorsqu'on souhaite obtenir la liste des articles d'un mois précis, selon la troisième règle que nous avons écrite, il faudrait accéder à l'URL suivante pour le mois de Juin 2012 : `/articles/2012/06`.

Cependant si on souhaite changer l'ordre des paramètres de l'URL pour afficher le mois, et ensuite l'année, celle-ci deviendrait `/articles/06/2012`. Il faudra donc modifier l'ordre des paramètres dans la déclaration de la fonction en conséquence.

Pour éviter cette lourdeur et un bon nombre d'erreurs, il est possible d'associer une variable de l'URL à un paramètre de la vue. Voici la démarche :

Code : Python

```
urlpatterns = patterns('blog.views',
    url(r'^home/$', 'home'), # Accueil du blog
    url(r'^article/(?P<id_article>\d+)/$', 'view_article'), # Vue
    d'un article
    url(r'^articles/(?P<year>\d{4})/(?P<month>\d{2})/$',
    'list_articles'), # Vue des articles d'un mois précis
)
```

et la vue correspondante :

Code : Python

```
def list_articles(request, month, year):
    """ Liste des articles d'un mois précis. """
    text = "Vous avez demandé les articles de {0}
    {1}.".format(month, year)
    return HttpResponse(text)
```

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 régle l'ordre des arguments en fonction des noms qui ont été donnés dans le `urls.py`. En réalité, le framework va exécuter la fonction de cette manière :

Code : Python

```
list_articles(request, year=2012, month=6)
```

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.

Pour terminer, sachez qu'il est toujours possible de passer des paramètres GET. Par exemple : `http://www.crepes-bretonnes.fr/article/1337?ref=twitter`. Django tentera de trouver le pattern correspondant en ne prenant en compte que ce qui est avant les paramètres GET, c'est à dire `/article/1337/`. Les paramètres passés par la méthode GET sont bien évidemment récupérables, ce que nous verrons plus tard.

Des réponses spéciales

Jusqu'ici, nous avons vu comment renvoyer une page HTML standard. Cependant, il se peut que l'on souhaite renvoyer autre chose que du HTML : une erreur 404 (page introuvable), une redirection vers une autre page, etc.

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, on peut 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.

Rapide exemple d'une vue compatible avec une des règles de routage que nous avons décrite dans le sous-chapitre précédent :

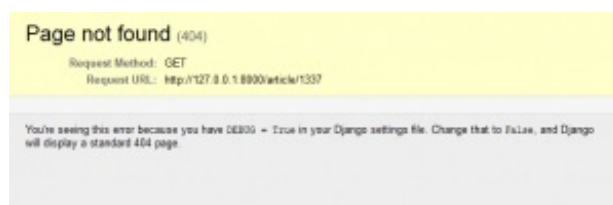
Code : Python

```
from django.http import HttpResponseRedirect, Http404

def view_article(request, id_article):
    if int(id_article) > 100: #Si l'ID est supérieur à 100, on considère que l'article n'existe pas
        raise Http404

    return HttpResponseRedirect('<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 (voir ci-contre). 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 l'image ci-contre.



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 paramètre 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 du Zéro, il faudrait procéder ainsi :

Code : Python

```
from django.shortcuts import redirect

def to_sdz(request):
    #On pourrait ici réaliser quelques opérations.

    return redirect("http://www.siteduzero.com") #On le redirige vers le Site du Zéro
```

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 indépendant la configuration des URLs et des vues. On doit donc passer en argument le nom de la vue vers laquelle on veut rediriger l'utilisateur, avec éventuellement des arguments destinés à celle-ci.

Code : Python

```
from django.http import HttpResponseRedirect, Http404
from django.shortcuts import redirect

def view_article(request, id_article):
    if int(id_article) > 100:
```



```
raise Http404

return redirect(view2)

def view2(request):
    return HttpResponse("Vous avez été redirigé.")
```

Ici, si l'utilisateur accède à l'URL `/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 rajoutant 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 l'on souhaitait rediriger un visiteur vers la vue `view_article` définie précédemment un ID d'article spécifique, il suffirait simplement d'utiliser la méthode `redirect` ainsi :

Code : Python

```
return redirect('blog.views.view_article', id_article=42)
```



Pourquoi est-ce qu'on utilise une chaîne de caractères pour désigner la vue maintenant, au lieu de la fonction elle-même ?

Il est possible d'indiquer une vue de trois manières différentes :

- En passant directement la fonction Python, comme on l'a vu au début ;
- En donnant le *chemin* vers la fonction, dans une chaîne de caractères (ce qui évite de l'importer si elle se situe dans un autre fichier) ;
- En indiquant le nom de la vue tel qu'indiqué dans un `urls.py` (voir l'exemple ci-dessous).

En réalité, la fonction `redirect` va construire l'URL vers la vue selon le routage indiqué dans `urls.py`. Ici, il va générer l'URL `/article/42` tout seul et rediriger l'utilisateur vers cette URL. Ainsi, si par la suite vous souhaitez modifier vos URLs, vous n'aurez qu'à le faire dans les fichiers `urls.py`, tout le reste se mettra à jour automatiquement. Il s'agit d'une fonctionnalité vraiment pratique, il ne faut donc **jamais** écrire d'URLs en dur, 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 dans `urls.py`. Par exemple :

Code : Python

```
url(r'^article/(?P<id_article>\d+)/$', 'view_article',
    name="afficher_article"),
```

Notez le paramètre `name="aficher_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, on peut faire beaucoup plus court et procéder comme ceci :

Code : Python

```
return redirect('afficher_article', id_article=42)
```

Pour terminer, sachez qu'il existe également une fonction qui permet de simplement générer l'URL et s'utilise de la même façon que `redirect`, il s'agit de `reverse` (`from django.core.urlresolvers import 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 utilisé dans les templates peu après pour générer des liens HTML vers les autres pages du site.

Les templates

Nous avons vu comment créer une vue et renvoyer du code HTML à l'utilisateur. Cependant, la méthode que nous avons utilisée n'est pas très pratique, le code HTML était en effet intégré à la vue elle-même ! Le code Python et le code HTML deviennent plus difficile à éditer et à maintenir pour plusieurs raisons :

- Les indentations HTML et Python se confondent ;
- La coloration syntaxique de votre éditeur favori ne fonctionnera généralement pas pour le code HTML, celui-ci n'étant qu'une simple chaîne de caractères ;
- Si vous avez un designer dans votre projet, celui-ci risque de casser votre code Python en voulant éditer le code HTML ;
- etc.

C'est à cause de ces raisons que 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) qu'on appelle des tags. Le moteur transforme les tags qu'il rencontre dans le fichier par le rendu HTML correspondant. Grâce à ceux-ci, il est possible d'effectuer plusieurs actions algorithmiques : afficher une variable, réaliser des conditions ou des boucles, faire des opérations sur des chaînes de caractères, etc.

Lier template et vue

Avant d'aborder le cœur même du fonctionnement des templates, retournons brièvement vers les vues. Dans la première partie, nous avons vu que nos vues étaient liées à des templates (et des modèles), comme montré sur le schéma ci-dessous :

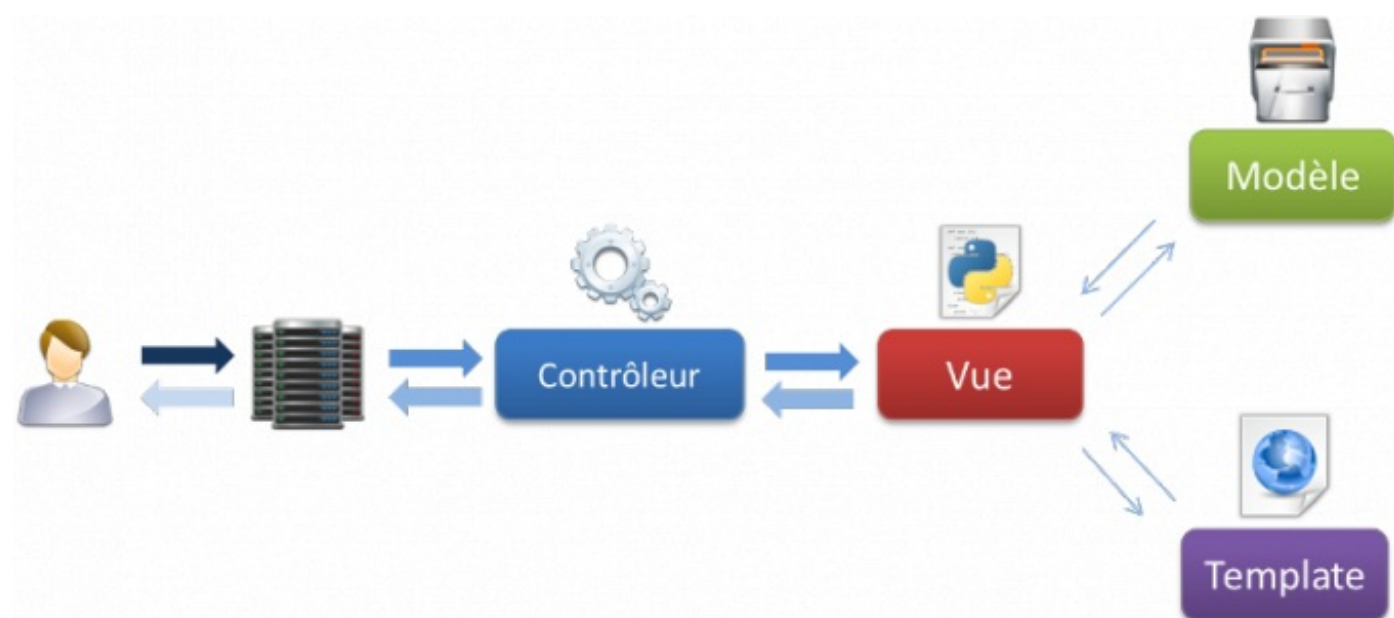


Schéma d'exécution d'une requête, vue dans la première partie

C'est la vue qui se charge de transmettre l'information de la requête au template, puis 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 dans ce sens.



La fonction `render` est en réalité une méthode de `django.shortcuts` qui nous simplifie la vie : elle génère un objet `HttpResponse` après avoir traité notre template. Pour les puristes qui veulent savoir comment ça fonctionne en interne, n'hésitez pas à aller fouiller dans la [documentation officielle](#).

On va commencer par un exemple, avec cette vue qui renvoie juste la date actuelle à notre vue, et son fichier `urls.py` associé :

Code : Python

```
from datetime import datetime
from django.shortcuts import render

def home(request):
```

```
return render(request, 'blog/home.html', {'current_date':
datetime.now()})
```

Code : Python - blog/urls.py

```
from django.conf.urls import patterns, url

urlpatterns = patterns('blog.views',
    url(r'^$', 'home'),
)
```

Cette fonction prend en argument 3 paramètres :

1. La requête initiale, qui a permis de construire la réponse (request dans notre cas)
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

Ici, notre template sera *home.html*, dans le sous-dossier *blog*, et on aura accès à une seule variable : *current_date* qui aura comme valeur la date renvoyée par la fonction *datetime.now()*.

Créons le template correspondant dans le dossier *templates/blog/*, ici nommé *home.html* :

Code : Jinja - templates/blog/home.html

```
<h1>Bienvenue sur mon blog</h1>
<p>La date actuelle est : {{ current_date }}</p>
```

Nous retrouvons *current_date*, comme passé dans *render()* ! Si vous accédez à cette page (après lui avoir assigné une URL), le `{{ current_date }}` est remplacé par la date actuelle 😊

Deuxième exemple : une vue qui additionne deux nombres donnés dans l'URL :

Code : Python - blog/views.py

```
def addition(request, number1, number2):
    total = int(number1) + int(number2)

    # retourne number1, number2 et total
    return render(request, 'blog/addition.html', locals())
```

Et le template associé :

Code : Jinja - templates/blog/addition.html

```
<h1>Ma super calculatrice</h1>
<p>{{ number1 }} + {{ number2 }}, ça fait <strong>{{ total }}</strong> !<br />
On peut également calculer la somme dans le template : {{
number1|add:number2 }}.</p>

{% if number1 > number2 %}
Le premier nombre est plus grand.
{% elif number1 < number2 %}
Le second nombre est plus grand
```

```
{% else %}  
Les nombres sont égaux  
{% endif %}
```



On expliquera un peu plus loin le fonctionnement des structures présentées ici, ne vous inquiétez pas !

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 appelé. Les clés seront les noms de variables (par exemple 'total'), et les valeurs du dictionnaire seront tout simplement... les valeurs des variables de la fonction ! Ainsi, si `number1` valait 42, la valeur 'number1' du dictionnaire vaudrait elle aussi 42.

Cet exemple introduit également les conditions dans les templates... On va voir d'ailleurs maintenant les différentes structures disponibles.

Affichons nos variable à l'utilisateur

Affichage d'une variable

Comme on vous le rabâche depuis le depuis de ce cours, la vue transmet au template les données à destination de l'utilisateur. Ces données correspondent à des variables classiques de votre vue. On peut les afficher dans le template grâce à l'expression `{{ }}` qui prend à l'intérieur des accolades un argument (on pourrait assimiler cette expression à une fonction), le nom de la variable à afficher. Le nom des variables est également limité aux caractères alphanumériques et aux underscores.

Code : Jinja

```
Bonjour {{ pseudo }}, nous sommes le {{ date }}.
```

Ici, on considère que la vue a transmis deux variables au template : `pseudo` et `date`. Ceux-ci seront affichés par le moteur de template. Si `pseudo` vaut "Zozor" et `date` "28 décembre", le moteur de template affichera "Bonjour Zozor, nous sommes le 28 décembre".

Si jamais la variable n'est pas une chaîne de caractères, le moteur de templates utilisera la méthode `__str__` de l'objet pour afficher. Par exemple, les listes seront affichés sous la forme `['element 1', 'element 2', ...]`, comme si vous demandiez son affichage dans une console Python.

Il est possible d'accéder aux attributs d'une variable via le point (`.`). Plus tard, nos articles de blog seront représentés par des objets, avec des attributs `titre`, `contenu`, etc. Pour y accéder, la syntaxe est la suivante :

Code : Jinja

```
{# On suppose que notre vue a retourné un objet Article nommé  
article #}  
<h2>{{ article.titre }}</h2>  
<p><em>Article publié par {{ article.auteur }}</em></p>  
  
{{ article.contenu }}
```



Si jamais une variable n'existe pas, ou n'a pas été envoyé au template, alors la valeur qui est affichée par celle définie par `TEMPLATE_STRING_IF_INVALID`, qui est une chaîne vide par défaut.

Les filtres

Lors de l'affichage des données, il est fréquent de devoir gérer plusieurs cas. Les filtres permettent de



modifier l'affichage en fonction d'une variable, sans passer par la vue. Prenons un exemple concret : sur la page d'accueil du Site du Zéro, le texte des dernières news est tronqué, seul le début est affiché. Pour faire ceci avec Django, on peut utiliser un filtre qui limite l'affichage aux 80 premiers mots de notre article :

Code : Jinja

```
{{
texte|truncatewords:80
}}
```



Bloc de news du SdZ

Ici, le filtre `truncatewords` (qui prend comme paramètre un nombre, séparé par un double-point) est appliqué à la variable `texte`. A l'affichage, cette dernière sera tronquée et l'utilisateur ne verra que les 80 premiers mots de celle-ci.

Ces filtres ont pour but d'effectuer des opérations de façon claire, afin d'alléger les vues, et ne marchent que lorsqu'une variable est affichée (avec la structure `{{ }}` donc). Il est par exemple possible d'accorder correctement les phrases de votre site avec le filtre `pluralize` :

Code : Jinja

```
Vous avez {{ nb_messages }} message{{ nb_messages|pluralize }}.
```

Dans ce cas, un "s" sera ajouté si le nombre de messages est supérieur à 1. Il est possible de passer des arguments au filtre afin de coller au mieux à notre chère langue française :

Code : Jinja

```
Il y a {{ nb_chevaux }} chev{{ nb_chevaux|pluralize:"al,aux" }} dans
l'écurie.
```

Ici, on aura "cheval" si `nb_chevaux` est égal à 1 et "chevaux" pour le reste.

Et un dernier pour la route : imaginons que vous souhaitez afficher le pseudo du membre connecté, ou le cas échéant "visiteur". Il est possible de le faire en quelques caractères, sans avoir recours à une condition !

Code : Jinja

```
Bienvenue {{ pseudo|default:"visiteur" }}
```

En bref, il existe des dizaines de filtres par défaut : `safe`, `length`, ... Pour vous aider, un mémo vous est proposé en annexes de ce tutoriel.

Manipulons nos données avec les tags

Le second type d'opération que l'on peut implémenter dans un template sont **les tags**. C'est grâce à ceux-ci que conditions, boucles, etc, sont disponibles.

Les conditions : `{% if %}`

Code : Jinja

```
Bonjour
{% if sexe == "Femme" %}
Madame
```

```
{% else %}  
Monsieur  
{% endif %} !
```

Ici, en fonction du contenu de la variable `sexe`, l'utilisateur ne verra pas le même texte à l'écran. Si l'utilisateur est une femme et que la variable `sexe` contient "Femme", elle verra "Bonjour Madame !". Sinon, on suppose que c'est un homme et on affiche "Bonjour Monsieur !".

Ce template est similaire au code HTML généré par la vue suivante :

Code : Python

```
def home(request):  
    sexe = "Femme"  
    html = "Bonjour "  
    if sexe == "Femme":  
        html += "Madame"  
    else:  
        html += "Monsieur"  
    html += " !"  
    return HttpResponse(html)
```

La séparation entre vue et template simplifie grandement les choses et a le mérite d'être **beaucoup plus lisible** que d'écrire directement le code HTML dans la vue !

Il est également possible d'utiliser les structures `if`, `elif`, `else` de la même façon :

Code : Jinja

```
{% if age > 25 %}  
Bienvenue Monsieur, veuillez passer un excellent moment dans nos  
locaux  
{% elif age > 16 %}  
Vas-y, tu peux passer.  
{% else %}  
Tu ne peux pas rentrer petit, tu es trop jeune !  
{% endif %}
```

Les boucles : {% for %}

Tout comme les conditions, le moteur de templates de Django permet l'utilisation de la boucle `for`, similaire à celle de Python. Admettons que l'on possède dans notre vue un tableau de couleurs définis en Python par

Code : Python

```
couleurs = ['rouge', 'orange', 'jaune', 'vert', 'bleu', 'indigo',  
            'violet']
```

On peut décider d'afficher cette liste dans notre template grâce à la syntaxe `{% for %}` suivante :

Code : Jinja

```
Les couleurs de l'arc-en-ciel sont :  
<ul>  
{% for couleur in couleurs %}
```



```
<li>{{ couleur }}</li>
{% endfor %}
</ul>
```

Avec ce template, le moteur va itérer la liste (ou n'importe quelle autre variable itérable), remplacer la variable couleur par l'élément actuel de l'itération et générer le code compris entre {% for %} et {% endfor %} pour chaque élément de la liste. On obtient la page HTML suivante :

Code : HTML

```
Les couleurs de l'arc-en-ciel sont :
<ul>
  <li>rouge</li>
  <li>orange</li>
  <li>j jaune</li>
  <li>vert</li>
  <li>bleu</li>
  <li>indigo</li>
  <li>violet</li>
</ul>
```

Il est aussi possible de parcourir un dictionnaire, en passant par la directive {% for cle, valeur in dictionnaire.items %} :

Code : Python

```
couleurs = {'FF0000': 'rouge',
            'ED7F10': 'orange',
            'FFFF00': 'jaune',
            '00FF00': 'vert',
            '0000FF': 'bleu',
            '4B0082': 'indigo',
            '660099': 'violet'}
```

Code : Jinja

```
Les couleurs de l'arc-en-ciel sont :
<ul>
  {% for code, nom in couleurs.items %}
  <li style="color:#{{ code }}">{{ nom }}</li>
  {% endfor %}
</ul>

Résultat :
<ul>
  <li style="color:#ED7F10">orange</li>
  <li style="color:#4B0082">indigo</li>
  <li style="color:#0000FF">bleu</li>
  <li style="color:#FFFF00">j jaune</li>
  <li style="color:#660099">violet</li>
  <li style="color:#FF0000">rouge</li>
  <li style="color:#00FF00">vert</li>
</ul>
```

Vous pouvez aussi réaliser n'importe quelle opération classique avec la variable générée par la boucle for (ici couleur): condition, utiliser une autre boucle, l'afficher, etc.



Rappelez-vous que la manipulation de données doit être fait au maximum dans les vues. Ces tags doivent juste servir à l'affichage.

Enfin, il existe une 3ème directive qui peut être associée au `{% for %}` qui est `{% empty %}`. Elle permet d'afficher un message par défaut si la liste que l'on souhaite parcourir est vide. Par exemple :

Code : Jinja

```
<h3>Commentaires de l'article</h3>
{% for commentaire in commentaires %}
<p>{{ commentaire }}</p>
{% empty %}
<p class="empty">Pas de commentaires pour le moment.</p>
{% endfor %}
```

Ici, si il y a au moins un élément dans `commentaires`, alors on affichera une suite de paragraphes, contenant chacun un élément de la liste. Sinon si la liste est vide, on retourne le paragraphe "Pas de commentaires pour le moment"

Le tag `{% block %}`

Sur la quasi-totalité des sites web, **une page est toujours composé de la même façon** : un haut de page, un menu et un pied de page. Si vous copiez-collez le code de vos menus dans tous vos templates et qu'un jour vous souhaitez modifier un élément de votre menu, il vous faudrait modifier tous vos templates !

Heureusement, le tag `{% block %}` va nous sauver de cette situation. En effet, il est possible de déclarer des blocs, qui seront définis dans un autre template. Dès lors, on peut créer un fichier, que l'on appelle usuellement `base.html`, qui va définir la structure globale de la page, autrement dit son *squelette*. Par exemple:

Code : Jinja

```
<!DOCTYPE html>
<html lang="fr">
<head>
<link rel="stylesheet" href="/media/css/style.css" />
<title>{% block title %}Mon blog sur les crêpes bretonnes{% endblock %}</title>
</head>

<body>

<header>Crêpes bretonnes</header>
<nav>
{% block nav %}
<ul>
<li><a href="/">Accueil</a></li>
<li><a href="/blog/">Blog</a></li>

<li><a href="/contact/">Contact</a></li>
</ul>
{% endblock %}
</nav>

<section id="content">
{% block content %}{% endblock %}
</section>

<footer>&copy; Crêpes bretonnes</footer>
</body>
</html>
```

Ce template est composé de plusieurs éléments `{% block %}` :

Dans la balise `<title>:{% block title %}Mon blog sur les crêpes bretonnes{% endblock %}`

Dans la balise `<nav>`, qui définit un menu

Dans le corps de la page, qui recevra le contenu

Tous ces blocs **pourront être redéfinis** ou inclus tels quels **dans un autre template**. Voyons d'ailleurs comment redéfinir et inclure ces blocs. Ces blocs ayant été écrits dans le fichier `"base.html"`, on peut appeler ce fichier dans chacun de nos templates de notre blog. Pour ce faire, on utilise le tag `{% extends %}` (pour ce qui ont déjà fait de la programmation objet, ça doit vous dire quelque chose ! Cette méthode peut aussi être assimilée à `include` en PHP) On parle alors d'**héritage de templates**. On prend la base que l'on surcharge, afin d'obtenir un résultat dérivé :

Code : Jinja

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

{% block title %}Ma page d'accueil{% endblock %}

{% block content %}
<h2>Bienvenue !</h2>
<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec
rhoncus massa non tortor.
Vestibulum diam diam, posuere in viverra in, ullamcorper et libero.
Donec eget libero quis risus congue imperdiet ac id lectus.
Nam euismod cursus arcu, et consequat libero ullamcorper sit amet.
Sociosqu ad litora torquent per conubia nostra, per inceptos
himenaeos. Integer
sit amet diam. Vivamus imperdiet felis a enim tincidunt
interdum.</p>
{% endblock %}
```

Dans cet exemple, nous avons redéfini deux blocs, `title` et `content`. Le tag `extends` va reprendre le template donné en argument, ici `base.html` et remplacer les blocs de ce dernier par les blocs de même nom redéfinis dans le template appelé par la vue. Ainsi, `title` et `content` seront repris du template fils, mais `nav` sera celui-ci défini par `base.html` et sera donc inchangé. Pour résumer, on a alors :



Comment fonctionne le tag `{% block %}`

Les liens vers les vues : `{% url %}`

Nous avons vu dans le chapitre précédent les fonctions `redirect` et `reverse`, qui respectivement, redirige l'utilisateur et génère le lien vers une vue, selon certains paramètres. Une variante sous la forme de tag de la fonction `reverse` existe, il s'agit de `{% url %}`. Le fonctionnement de ce tag est très similaire à la fonction dont il est dérivé :

Code : Jinja

```
<a href="{% url blog.views.view_article 42 %}">Lien vers mon super article N° 42</a>
```

générera le code HTML suivant :

Code : HTML

```
<a href="/article/42">Lien vers mon super article N° 42</a>
```

On indique le chemin vers la vue ou son nom comme premier paramètre, ceux qui suivent seront ceux de la vue (à condition de respecter le nombre et l'ordre des paramètres selon la déclaration de la vue bien entendu).

On aurait tout à fait pu utiliser une variable comme paramètre pour la vue :

Code : Jinja

```
<a href="{% url blog.views.view_article ID_article %}">Lien vers mon super article N° 42</a>
```

Les commentaires : {% comment %}

Enfin, il existe également un tag qui permet de définir des commentaires dans les templates. Ces commentaires sont **différents des commentaires HTML** : ils n'apparaîtront pas dans la page HTML. Cela permet par exemple de cacher temporairement une ligne, ou tout simplement documenter votre template, afin de pouvoir mieux s'y retrouver par la suite.

Il y a deux syntaxes pour les commentaires : la première permet de faire un commentaire sur une ligne uniquement :

```
{# Mon commentaire #}
```

Code : Jinja

```
<p>Ma page HTML</p>
<!-- Ce commentaire HTML sera visible dans le code source. -->
{# Ce commentaire Django ne sera pas visible dans le code source.
#}
```

Si vous souhaitez faire un commentaire sur plusieurs lignes, il vous faudra utiliser le tag {% comment %}

Code : Jinja

```
{% comment %}
Ceci est une page d'exemple. Elle est composée de 3 tableaux
- Tableau des ventes
- locations
- retours en garantie
{% endcomment %}
```

Ajoutons des fichiers statiques

Pour le moment, nous n'avons utilisé que du HTML dans nos templates. Cependant, un site web est composé aujourd'hui de nombreuses ressources : CSS, JavaScript, images, etc. On va voir comment les intégrer dans nos templates.

Tout d'abord, créons un dossier à la racine du projet dans lequel vous enregistrerez vos fichiers. Nous l'appellerons "assets". Il faut ensuite renseigner ce dossier et lui assigner une URL dans votre settings.py. Voilà les deux variables qu'il faudra modifier, ici selon notre exemple :

Code : Python

```
STATIC_URL = '/assets/'

STATICFILES_DIRS = (
    "/home/crepes/crepes_bretonnes/assets/",
)
```

La première variable indique l'URL du dossier depuis lequel vos fichiers seront accessibles. La deuxième renseigne le chemin vers ces fichiers sur votre disque dur.

Par la suite, si vous mettez une image, nommée par exemple *salut.jpg*, dans votre dossier "assets" (toujours selon notre exemple), vous pourrez l'inclure par la suite depuis votre template de la façon suivante :

Code : Jinja

```
{% load static %}

```

Vous avez besoin de faire `{% load static %}` une fois au début de votre template, et Django s'occupe tout seul de fournir l'URL vers votre ressource. Il est déconseillé d'écrire en dur le lien complet vers les fichiers statiques, utilisez toujours `{% static %}`. En effet, si en production vous décidez que vos fichiers seront servis depuis l'URL `assets.crepes-bretonnes.fr`, vous devrez modifier toutes vos URLs si celles-ci sont écrites en dur ! En revanche, si celles-ci utilisent `{% static %}`, vous n'aurez qu'à éditer cette variable dans votre configuration, ce qui est tout de même bien plus pratique.

En réalité, Django ne doit pas s'occuper de servir ces fichiers, c'est à votre serveur web qu'incombe cette tâche. Cependant, en développement, étant donné que nous utilisons le serveur intégré fourni par défaut par Django, il est tout de même possible de s'arranger pour que le framework serve ces fichiers.

Pour ce faire, il faut rajouter un routage spécifique à votre `urls.py` principal :

Code : Python

```
from django.conf.urls import patterns, include, url
from django.contrib.staticfiles.urls import staticfiles_urlpatterns

urlpatterns = patterns('',
    # Ici vos règles classiques, comme on l'a vu au chapitre
    # précédent
)

urlpatterns += staticfiles_urlpatterns()
```



Cette fonction ne marche qu'avec `DEBUG=True` et ne doit donc être utilisée qu'en développement !

Cette fonction va se baser sur les variables de votre fichier `settings.py` (URL et location des fichiers) pour générer une règle de routage correcte et adaptée. Pour le déploiement des fichiers statiques en production, on le verra dans notre annexe à ce sujet.

Les modèles

Nous avons vu comment créer des vues et des templates. Cependant, sans les modèles, ces derniers sont presque inutiles, car votre site n'aurait rien de dynamique. Autant créer des pages HTML statiques !

Dans ce chapitre, nous verrons les modèles, qui, comme expliqués dans la première partie, sont des interfaces permettant d'accéder et mettre à jour des données dans une base de données plus simplement.

Créer un modèle

Un modèle s'écrit sous la forme d'une classe et représente une table dans la base de données, dont ses attributs correspondent aux champs de la table.

Ceux-ci se rédigent dans le fichier `models.py` de chaque application. Il est important d'organiser correctement vos modèles pour que chaque modèle ait sa place dans son application, et ne pas mélanger tous les modèles dans le même `models.py`. Pensez à la réutilisation et à la structure du code !

Tout modèle Django se doit d'hériter de la classe mère `Model` incluse dans `django.db.models` (il ne sera pas pris en compte par le framework sinon). Par défaut, le fichier `models.py` généré automatiquement importe le module `models` de `django.db`. Voici un simple exemple de modèle représentant un article de blog :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField(null=True)
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose_name="Date de parution")

    def __unicode__(self):
        """
        Cette méthode que nous définirons dans tous les modèles
        nous permettra de reconnaître facilement les différents objets que
        nous
        traiterons plus tard et dans l'administration
        """
        return u"%s" % self.titre
```

Pour que Django puisse créer une table dans la base de données, il faut lui préciser le type des champs qu'il doit créer. Pour ce faire, le framework propose une liste de champs qu'il sera ensuite capable de retranscrire en langage SQL. Ces derniers sont également situés dans le module `models`.

Dans l'exemple ci-dessus, nous avons créés 4 attributs avec trois types de champs différents. Un `CharField` (littéralement, un champ de caractères), a été assigné à `titre` et `auteur`. Ce champ permet d'enregistrer une chaîne de caractères, dont la longueur maximale a été spécifiée via paramètre `max_length`. Dans le premier cas, la chaîne de caractères pourra être composée de 100 caractères maximum.

Le deuxième type de champ, `TextField`, permet lui aussi d'enregistrer des caractères, un peu comme `CharField`. En réalité, Django va utiliser un autre type de champ qui ne fixe pas de taille maximale à la chaîne de caractères, ce qui est très pratique pour enregistrer de longs textes.

Finalement, le champ `DateTimeField` prend comme valeur un objet `DateTime` du module `datetime` de la librairie standard. Il est donc possible d'enregistrer autre chose que du texte !

Insistons ici sur le fait que les champs du modèle peuvent prendre plusieurs arguments. Certains sont spécifiques au champ, d'autres non. Par exemple, le champ `DateTimeField` possède un argument facultatif : `auto_now_add`. S'il est mis à `True`, lors de la création d'une nouvelle entrée, Django mettra automatiquement à jour la valeur avec la date et heure de la création de l'objet.

Un autre argument du même genre existe, `auto_now`, il permet à peu près la même chose, mais fera en sorte que la date soit mise à jour à chaque modification de l'entrée.

L'argument `verbose_name` en revanche est un argument commun à tous les champs de Django. Il peut être passé à un `DateTimeField`, `CharField`, `TextField`, etc. Il sera notamment utilisé dans l'administration générée automatiquement pour donner une précision quant au nom du champ. Ici, nous avons insisté sur le fait que la date correspond bien à la date de parution de l'article.

Le paramètre `null`, lorsque mis à `True`, indique à Django que ce champ peut être laissé vide et qu'il est donc optionnel.

Il existe beaucoup d'autres champs disponibles, ceux-ci sont repris dans un mémo dans les annexes et sur la documentation de Django. N'hésitez pas à les consulter en cas de doute ou question !

Pour que Django crée la table associée au modèle, il suffit de lancer la commande `syncdb` via `manage.py` :

Code : Console

```
python manage.py syncdb
```

Étant donné que c'est la première fois que vous lancez la commande, Django va créer d'autres tables plus générales (utilisateurs, groupes, sessions, etc). À un moment, Django vous proposera de créer un compte administrateur. Répondez par "yes" et complétez les champs qu'il proposera par la suite. Nous reviendrons sur tout cela plus tard.

Table	Rows	Charset	Overhead
auth_group	0	latin1	8 MB
auth_group_permissions	0	latin1	8 MB
auth_permission	27	latin1	8 MB
auth_user	0	latin1	8 MB
auth_user_groups	0	latin1	8 MB
auth_user_user_permissions	0	latin1	8 MB
blog_article	0	latin1	8 MB
blog_categorie	0	latin1	8 MB
django_admin_log	0	latin1	8 MB
django_content_type	9	latin1	8 MB
django_session	0	latin1	8 MB
django_site	1	latin1	8 MB

Aperçu des tables créées dans un outil de gestion de base de données

La table associée au modèle `Article` étant créée, nous pouvons commencer à jouer avec !

Jouons avec des données

Django propose un interpréteur interactif Python synchronisé avec votre configuration du framework. Il est possible via celui-ci de manipuler nos modèles comme si on était dans une vue. Pour ce faire, il suffit d'utiliser une autre commande de l'utilitaire `manage.py` :

Code : Console

```
$ python manage.py shell
Python 2.7.3 (default, Apr 24 2012, 00:00:54)
[GCC 4.7.0 20120414 (prerelease)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

Commençons par importer le modèle que nous avons justement créé :

Code : Python

```
>>> from blog.models import Article
```

Pour rajouter une entrée dans la base de données, il suffit de créer une nouvelle instance de la classe `Article` et de la sauvegarder. Chaque instance d'un modèle correspond donc à une entrée dans la base de données.

On peut spécifier la valeur des attributs directement pendant l'instanciation de classe, ou les assigner par la suite :

Code : Python

```
>>> article = Article(titre="Bonjour", auteur="Maxime")
>>> article.contenu = "Les crêpes bretonnes sont trop bonnes !"
```



Pourquoi n'a-t-on pas mis de valeur à l'attribut `date` ?

`date` est un `DateTimeField` dont le paramètre `auto_now_add` a été mis à `True`. Dès lors, Django se charge tout seul de le mettre à jour avec la bonne date et heure lors de la création. Cependant, il est tout de même obligatoire de remplir tous les champs pour chaque entrée sauf cas comme celui-ci, sans quoi Django retournera une erreur !

On peut bien évidemment accéder aux attributs de l'objet comme pour n'importe quel autre objet Python :

Code : Python

```
>>> print article.auteur
Maxime
```

Pour sauvegarder l'entrée dans la base de données (les modifications ne sont pas enregistrées en temps réel), il suffit d'appeler la méthode `save`, de la classe mère `Model` dont hérite chaque modèle :

Code : Python

```
>>> article.save()
```

L'entrée a été créée et enregistrée !

Bien évidemment, il est toujours possible de modifier l'objet par la suite :

Code : Python

```
>>> article.titre = "Salut !"
>>> article.auteur = "Mathieu"
>>> article.save()
```

Il ne faut cependant pas oublier d'appeler la méthode `save` à chaque modification, sinon les changements ne seront pas sauvegardés.

Pour supprimer une entrée dans la base de données, rien de plus simple, il suffit d'appeler la méthode `delete` d'un objet :

Code : Python

```
>>> article.delete()
```

Nous avons vu comment créer, éditer et supprimer des entrées. Il serait pourtant également intéressant de pouvoir les obtenir par la suite, pour les afficher par exemple.

Pour ce faire, chaque modèle (la classe, et non l'instance, attention !), possède plusieurs méthodes dans la sous-classe `objects`. Par exemple, pour obtenir toutes les entrées enregistrées d'un modèle, il faut appeler la méthode `all()` :

Code : Python


```
>>> Article.objects.all()
[]
```

Bien évidemment, étant donné que nous avons supprimé l'article créé un peu plus tôt, l'ensemble renvoyé est vide, créons rapidement deux nouvelles entrées :

Code : Python

```
>>> Article(auteur="Mathieu", titre="Les crepes", contenu="Les crepes
c'est cool").save()
>>> Article(auteur="Maxime", titre="La Bretagne", contenu="La
Bretagne c'est trop bien").save()
```

Ceci étant fait, réutilisons la méthode `all` :

Code : Python

```
>>> Article.objects.all()
[<Article: Les crepes>, <Article: La Bretagne>]
```

L'ensemble renvoyé par la fonction n'est pas une vraie liste, mais un `QuerySet`. Il s'agit d'un conteneur itérable qui propose d'autres méthodes sur lesquelles nous nous attarderons par la suite. Nous avons donc deux éléments, chacun correspondant à un des articles que nous avons créé.

On peut donc par exemple afficher les différents titres de nos articles :

Code : Python

```
>>> for article in Article.objects.all():
...     print article.titre

Les crepes
La Bretagne
```

Maintenant, imaginons que je souhaite prendre tous les articles d'un seul auteur uniquement. La méthode `filter` a été conçue dans ce but. Elle prend en paramètre une valeur d'un ou plusieurs attributs et va passer en revue toutes les entrées de la table et ne sélectionner que les instances qui ont également la valeur de l'attribut correspondant. Par exemple :

Code : Python

```
>>> for article in Article.objects.filter(auteur="Maxime"):
...     print article.titre, "par", article.auteur

La Bretagne par Maxime
```

Efficace ! L'autre article n'a pas été repris dans le `QuerySet` car son auteur n'était pas Maxime mais Mathieu.

Une méthode similaire à `filter` existe, mais fait le contraire : `exclude`. Comme son nom l'indique, elle exclut les entrées dont la valeur des attributs passés en argument coïncide :

Code : Python

```
>>> for article in Article.objects.exclude(auteur="Maxime") :  
...     print article.titre, "par", article.auteur  
  
Les crepes par Mathieu
```

Il est même possible d'aller plus loin, en filtrant par exemple les articles dont leur titre doit contenir certains caractères (et non pas être strictement égal à une chaîne entière). Si l'on souhaite prendre tous les articles dont le titre comporte le mot "crêpe", il faut procéder ainsi :

Code : Python

```
>>> Article.objects.filter(titre__contains="crepe")  
[<Article: Les crepes>]
```

Ces méthodes de recherches spéciales sont construites en prenant le champ concerné (ici titre), auquel on rajoute deux underscore (__), suivi finalement de la méthode souhaitée. Ici, il s'agit donc de `titre__contains`, qui veut dire littéralement, prend tous les éléments dont le titre contient (traduction française de `contains`) le mot passé en argument.

D'autres méthodes du genre existent, dont notamment la possibilité de prendre des valeurs du champ (strictement) inférieures ou (strictement) supérieures à l'argument passé, grâce à la méthode `lt` (*less than*, plus petit que) et `gt` (*greater than*, plus grand que) :

Code : Python

```
>>> from datetime import datetime  
>>> Article.objects.filter(date__lt=datetime.now())  
[<Article: Les crepes>, <Article: La Bretagne>]
```

Les deux articles ont été sélectionnés car ils remplissent tous deux la condition (leur date de parution est inférieure au moment actuel). Si nous avions utilisé `gt` au lieu de `lt`, la requête aurait renvoyé un `QuerySet` vide, car aucun article n'a été publié après le moment actuel.

De même, il existe `lte` et `gte` qui opèrent de la même façon, la différence réside juste dans le fait que ceux-ci prendront tout élément inférieur/supérieur ou égal (`lte` : *less than or equal*, plus petit ou égal, idem pour `gte`).

Sur la page d'accueil de notre blog, on souhaitera afficher les articles par date de parution, du plus récent au plus ancien. Pour ce faire, on utilise la méthode `order_by`. Cette dernière prend comme argument une liste de chaînes de caractères qui correspondent aux attributs du modèle :

Code : Python

```
>>> Article.objects.order_by('date')  
[<Article: Les crepes>, <Article: La Bretagne>]
```

Le tri se fait par ordre ascendant (ici du plus ancien au plus récent, nous avons enregistré l'article sur les crêpes avant celui sur la Bretagne). Pour spécifier un ordre descendant, il suffit de précéder le nom de l'attribut par le caractère '-' :

Code : Python

```
>>> Article.objects.order_by('-date')  
[<Article: La Bretagne>, <Article: Les crepes>]
```

Il est possible de passer plusieurs noms d'attributs à `order_by`. La priorité de chaque attribut dans le tri est déterminé par sa position dans la liste d'arguments. Ainsi, si l'on trie les articles par noms et que deux d'entre-eux possèdent le même nom, Django

les départagera selon le deuxième attribut, et ainsi de suite tant que des attributs comparés seront identiques.

Accessoirement, on peut inverser les éléments d'un QuerySet en utilisant la méthode `reverse()`.

Finalement, dernière caractéristique importante des méthodes de QuerySet, elles sont cumulables, ce qui garantit une grande souplesse dans vos requêtes :

Code : Python

```
>>>
Article.objects.filter(date__lt=datetime.now()).order_by('date', 'name').reverse()
[<Article: La Bretagne>, <Article: Les crepes>]
```

Pour terminer ce (long) sous-chapitre, nous allons introduire des méthodes qui, contrairement aux précédentes, retournent un seul objet et non un QuerySet.

Premièrement, `get`, comme son nom l'indique, permet d'obtenir une et une seule entrée d'un modèle. Il prend les mêmes arguments de la même manière que `filter` ou `exclude`. S'il ne retrouve aucun élément correspondant aux conditions, ou plus d'un seul, il retourne une erreur :

Code : Python

```
>>> Article.objects.get(titre="Je n'existe pas")

...
DoesNotExist: Article matching query does not exist.

>>> Article.objects.get(auteur="Mathieu").titre
u'Les crepes'

>>> Article.objects.get(titre__contains="L")

...
MultipleObjectsReturned: get() returned more than one Article -- it
returned 2! Lookup parameters were {'titre__contains': 'L'}
```

Dans le même style, il existe une méthode permettant de créer une entrée si aucune autre existe avec les conditions spécifiées. Il s'agit de `get_or_create`. Cette dernière va renvoyer un tuple contenant l'objet désiré et un booléen qui indique si une nouvelle entrée a été créée ou non :

Code : Python

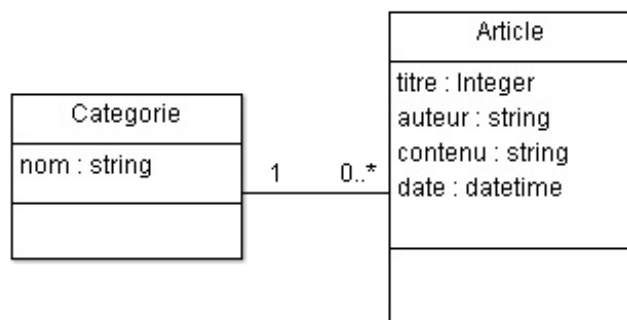
```
Article.objects.get_or_create(auteur="Mathieu")
>>> (<Article: Les crepes>, False)

Article.objects.get_or_create(auteur="Zozor", titre="Hi han")
>>> (<Article: Hi han>, True)
```

Les liaisons entre modèles

Il est souvent pratique de lier deux modèles entre eux, pour relier un article à une catégorie par exemple. Django propose tout un système permettant de simplifier grandement les différents types de liaison. Nous traiterons ce sujet dans ce sous-chapitre.

Reprenons notre exemple des catégories et des articles. Lorsque vous allez concevoir votre base de données, ce que l'on fait souvent en passant par de l'[UML](#), vous allez souvent faire des liens entre les classes (qui représente nos tables SQL dans notre site) :



Ici, un article peut être lié à une et une seule catégorie, et une catégorie peut être attributée à une infinité d'articles

Pour traduire cette relation, nous allons d'abord devoir créer un autre modèle représentant les catégories. Ce dernier est relativement simple :

Code : Python

```

class Categorie(models.Model):
    nom = models.CharField(max_length=30)

    def __unicode__(self):
        return self.nom
  
```

Maintenant, on va créer la liaison depuis notre modèle `Article`, qu'il va falloir modifier en lui rajoutant un nouveau champ :

Code : Python

```

class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField(null=True)
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose name="Date de parution")
    categorie = models.ForeignKey(Categorie)

    def __unicode__(self):
        return self.titre
  
```

Assurez-vous que `Categorie` soit bien déclaré avant `Article` !

Nous avons donc rajouté un champ `ForeignKey`. En français, ce dernier est traduit par « clé étrangère ». Il va enregistrer une clé, un identifiant propre à chaque catégorie enregistrée (il s'agit la plupart du temps d'un nombre), qui permettra donc de retrouver la catégorie associée.

Nous avons modifié notre classe `Article` et allons rencontrer un des rares défauts de Django. En effet, si l'on lançait maintenant `manage.py syncdb`, il créerait bien la table correspondant au modèle `Categorie`, mais ne rajouterait pas le champ `ForeignKey` dans la table `Article` pour autant.

Pour résoudre ce problème, deux méthodes s'offrent à vous :

- Vous créez manuellement le champ via l'interface SQLite et en langage SQL si vous en êtes capables, c'est la solution propre, mais plus difficile ;
- Vous supprimez la base de données (qui n'est qu'un simple fichier) ou la table (via SQLite), et vous en recréez un en relançant `manage.py syncdb`. Les données seront perdues mais les structures des tables seront à jour. Étant donné que nous n'avons pas de vraies données pour le moment, privilégiez cette solution.

La base de donnée étant prête, ouvrez à nouveau un shell via `manage.py shell`. Importons les modèles et créons une nouvelle catégorie :

Code : Python

```
>>> from blog.models import Categorie, Article

>>> cat = Categorie(nom="Crêpes")
>>> cat.save()

>>> art = Article()
>>> art.titre="Les nouvelles crêpes"
>>> art.auteur="Maxime"
>>> art.contenu="On a fait de nouvelles crêpes avec du trop bon rhum"
>>> art.categorie = cat
>>> art.save()
```

Pour accéder aux attributs et méthodes de la catégorie associée à l'article, rien de plus simple :

Code : Python

```
>>> print art.categorie.nom
Crêpes
```

Dans cet exemple, si un article ne peut avoir qu'une seule catégorie, une catégorie en revanche, peut avoir plusieurs articles. Pour réaliser l'opération en sens inverse (accéder aux articles d'une catégorie depuis cette dernière), une sous-classe s'est créée toute seule avec la `ForeignKey` :

Code : Python

```
cat.article_set.all()
[<Article: Les nouvelles crêpes>]
```

Le nom que prendra une relation en sens inverse est composé du nom du modèle source (qui a la `ForeignKey` comme attribut), d'un seul underscore (`_`) et finalement du mot `set` qui signifie en anglais "ensemble". On accède ici donc à l'ensemble des articles d'une catégorie.

Cette relation opère exactement comme n'importe quelle sous-classe objects d'un modèle, et renvoie ici tous les articles de la catégorie. On peut utiliser les méthodes que nous avons vues précédemment : `all`, `filter`, `exclude`, `order_by`, ...

Point important : il est possible d'accéder aux attributs du modèle lié par une clé étrangère depuis un `filter`, `exclude`, `order_by`, ... On pourrait ici par exemple filtrer tous les articles dont le titre de la catégorie possède un certain mot :

Code : Python

```
Article.objects.filter(categorie__nom__contains="crêpes")
```

Accéder à un élément d'une clé étrangère se fait en rajoutant deux underscore `__`, comme avec les méthodes de recherche spécifiques, suivi du nom de l'attribut recherché. Comme montré dans l'exemple, on peut encore rajouter une méthode spéciale de recherche sans aucun problème !

Un autre type de liaison existe, très similaire au principe des clés étrangères : le `OneToOneField`. Ce dernier permet de lier un modèle à un autre tout aussi facilement, et garantit qu'une fois la liaison effectuée, plus aucun autre objet ne pourra plus être associé à ceux déjà associés. La relation devient unique. Si nous avions utilisé notre exemple avec un `OneToOneField`, chaque catégorie ne pourrait avoir qu'un seul article associé, et de même pour chaque article.

Un autre bref exemple :

Code : Python

```

class Moteur(models.Model):
    nom = models.CharField(max_length=25)

    def __unicode__(self):
        return self.nom

class Voiture(models.Model):
    nom = models.CharField(max_length=25)
    moteur = models.OneToOneField(Moteur)

    def __unicode__(self):
        return self.nom

```

Nous avons deux objets, un moteur nommé "Vroum", une voiture nommée "Crêpes-mobile" et cette dernière est liée au moteur, on peut accéder du moteur à la voiture ainsi :

Code : Python

```

>>> moteur = Moteur.objects.create(nom="Vroum") #create crée
directement l'objet et l'enregistre
>>> voiture = Voiture.objects.create(nom="Crêpes-mobile",
moteur=moteur)

>>> moteur.voiture
<Voiture: Crêpes-mobile>
>>> voiture.moteur
<Moteur: Vroum>

```

Ici, le `OneToOneField` a créé une relation en sens inverse qui ne va plus renvoyer un `QuerySet` mais directement l'élément concerné (ce qui est logique, celui-ci étant unique). Cette relation inverse prendra simplement le nom du modèle, et n'est donc plus suivi par `_set`.

Sachez qu'il est possible de changer le nom de la variable créée par la relation inverse (précédemment `article_set` et `moteur`). Pour ce faire, on utilise l'argument `related_name` du `ForeignKey` ou `OneToOneField` et on lui passe une chaîne de caractères désignant le nouveau nom de la variable (à condition que cette chaîne représente bien un nom de variable valide !). Cette solution est notamment utilisée en cas de conflit entre noms de variables. Accessoirement, il est même possible de désactiver la relation inverse en donnant `related_name='+'`.

Finalement, dernier type de liaison, le plus complexe : le `ManyToManyField` (traduit littéralement, plusieurs-à-plusieurs). Reprenons un autre exemple simple : nous construisons un comparateur de prix pour les ingrédients nécessaires à la réalisation de crêpes. Plusieurs vendeurs proposent plusieurs produits, parfois identiques, à des prix différents.

Il nous faudra trois modèles :

Code : Python

```

class Produit(models.Model):
    nom = models.CharField(max_length=30)

    def __unicode__(self):
        return self.nom

class Vendeur(models.Model):
    nom = models.CharField(max_length=30)
    produits = models.ManyToManyField(Produit, through='Offre')

    def __unicode__(self):
        return self.nom

```

```
class Offre(models.Model):
    prix = models.IntegerField()
    produit = models.ForeignKey(Produit)
    vendeur = models.ForeignKey(Vendeur)

    def __unicode__(self):
        return "{0} vendu par {1}".format(self.produit,
self.vendeur)
```

Explications ! Les modèles `Produit` et `Vendeur` sont classiques, à l'exception du fait que nous avons utilisé un `ManyToManyField` dans `Vendeur`, au lieu d'une `ForeignKey` ou `OneToOneField` comme précédemment. La nouveauté, en revanche, est bien le troisième modèle : `Offre`. C'est celui-ci qui fait le lien entre `Produit` et `Vendeur` et permet de rajouter des informations supplémentaires sur la liaison (ici le prix, caractérisé par un `IntegerField` qui enregistre un nombre).

Un `ManyToManyField` va toujours créer une table intermédiaire qui enregistrera les clés étrangères des différents objets des modèles associés. On peut soit laisser Django s'en occuper tout seul, soit la créer nous-mêmes pour y rajouter des attributs supplémentaires (pour rappel, ici on rajoute le prix). Dans ce deuxième cas, il faut spécifier le modèle faisant la liaison via l'argument `through` du `ManyToManyField` et ne surtout pas oublier d'ajouter des `ForeignKey` vers les deux modèles qui seront liés.

Créez les tables via `syncdb` et lancez un shell. Enregistrons un vendeur et deux produits :

Code : Python

```
>>> vendeur = Vendeur.objects.create(nom="Carouf")
>>> p1 = Produit.objects.create(nom="Lait")
>>> p2 = Produit.objects.create(nom="Farine")
```

Désormais, la gestion du `ManyToMany` se fait de deux manières différentes. Soit on a spécifié manuellement la table intermédiaire, soit on a laissé Django le faire.

Étant donné que nous avons opté pour la première méthode, tout ce qu'il reste à faire, c'est créer un nouvel objet `Offre` qui reprend le vendeur, le produit et le prix proposé pour celui-ci :

Code : Python

```
>>> o1 = Offre.objects.create(vendeur=vendeur, produit=p1, prix=10)
>>> o2 = Offre.objects.create(vendeur=vendeur, produit=p2, prix=42)
```

Si nous avons laissé Django générer automatiquement la table, il aurait fallu procéder ainsi :

Code : Python

```
vendeur.produits.add(p1,p2)
```

Pour supprimer une liaison entre deux objets, deux méthodes se présentent encore. Avec une table intermédiaire spécifiée manuellement, il suffit de supprimer l'objet faisant la liaison (supprimer un objet `Offre` ici), autrement on utilise une autre méthode du `ManyToManyField` :

Code : Python

```
vendeur.produits.remove(p1) #On a supprimé p1, il ne reste plus que
p2 qui est lié au vendeur
```

Ensuite, pour accéder aux objets associés du modèle source (possédant la déclaration du `ManyToManyField`, ici `Vendeur`), au modèle destinataire (ici `Produit`), rien de plus simple, on obtient à nouveau un `QuerySet` :

Code : Python

```
>>> vendeur.produits.all()
[<Produit: Lait>, <Produit: Farine>]
```

Encore une fois, toutes les méthodes des `QuerySet` (`filter`, `exclude`, `order_by`, `reverse`, ...) sont également accessibles.

Comme pour les `ForeignKey`, une relation inverse s'est créée :

Code : Python

```
>>> p1.vendeur_set.all()
[<Vendeur: Carouf>]
```

Pour rappel, il est également possible avec des `ManyToMany` de modifier le nom de la variable faisant la relation inverse via l'argument `related_name`.

Accessoirement si l'on souhaite accéder aux valeurs du modèle intermédiaire (ici `Offre`), il faut procéder de manière classique :

```
>>> Offre.objects.get(vendeur=vendeur, produit=p1).prix 10
```

Finalement, pour supprimer toutes les liaisons d'un `ManyToManyField`, que la table intermédiaire soit générée automatiquement ou manuellement, on peut appeler la méthode `clear` :

Code : Python

```
>>> vendeur.produits.clear()
>>> vendeur.produits.all()
[]
```

Et tout a disparu !

Les modèles dans les vues

Nous avons vu comment utiliser les modèles d'une manière plutôt théorique et dans la console. Ce sous-chapitre est là pour introduire les modèles dans un autre milieu plus utile : les vues.

Afficher les articles du blog

Pour afficher les articles de notre blog, il suffit de reprendre une de nos requêtes précédentes, et l'incorporer dans une vue. Dans notre template, on ajoutera un lien vers notre article pour pouvoir le lire en entier. Le problème qui se pose ici, et que nous n'avons pas soulevé avant est le choix d'un identifiant. En effet, comment passer dans l'URL une information facile à transcrire pour désigner un article particulier ?

En réalité, nos modèles contiennent plus d'attributs et de champs SQL que nous en déclarons. On peut le remarquer depuis la commande `python manage.py sql blog`, qui renvoie la structure SQL des tables créées :

Code : SQL

```
BEGIN;
CREATE TABLE "blog_categorie" (
```



```

        "id" integer NOT NULL PRIMARY KEY,
        "nom" varchar(30) NOT NULL
    )
;
CREATE TABLE "blog_article" (
    "id" integer NOT NULL PRIMARY KEY,
    "titre" varchar(100) NOT NULL,
    "auteur" varchar(42) NOT NULL,
    "contenu" text NOT NULL,
    "date" datetime NOT NULL,
    "categorie_id" integer NOT NULL REFERENCES "blog_categorie"
("id")
)
;
COMMIT;

```

Chaque table contient les attributs définis dans le modèle, mais également un champ *"id"* qui est un nombre auto-incrémenté (le premier article aura l'ID 1, le deuxième l'ID 2, etc), et donc unique ! C'est ce champ qui sera utilisé pour désigner un article particulier.

Passons à quelque chose de plus concret, voici un exemple d'application :

Code : Python - blog/views.py

```

from django.http import Http404
from django.shortcuts import render
from blog.models import Article

def accueil(request):
    """ Afficher tous les articles de notre blog """
    articles = Article.objects.all() # On sélectionne tous nos
articles
    return render(request, 'blog/accueil.html',
{'derniers_articles':articles})

def lire(request, id):
    """ Affiche un article complet """
    pass # Voir ci-dessous.

```

Code : Python - Extrait de blog/urls.py

```

urlpatterns = patterns('blog.views',
    url(r'^$', 'accueil'),
    url(r'^article/(?P<id>\d+)\$', 'lire'),
)

```

Code : Jinja - Template accueil.html

```

<h1>Bienvenue sur le blog des crêpes bretonnes !</h1>

{% for article in derniers_articles %}
<div class="article">
<h3>{{ article.titre }}</h3>
<p>{{ article.contenu|truncatewords_html:80 }}</p>
<p><a href="{% url blog.views.lire article.id %}">Lire la suite</a>
</div>
{% empty %}
<p>Aucun article.</p>
{% endfor %}

```

Nous récupérerons tous les articles via la méthode `objects.all()` et on renvoie la liste au template. Dans le template, rien de fondamentalement nouveau non plus : on affiche un à un les articles. Le seul point nouveau est celui que nous avons cité plus haut : on fait un lien vers l'article complet, en jouant avec le champ `id` de la table SQL.

Afficher un article précis

L'affichage d'un article précis est plus délicat : il faut vérifier que l'article demandé existe, et le cas contraire, renvoyer une erreur 404.

Notons déjà qu'il n'y a pas besoin de vérifier si l'ID précisé est bel et bien un nombre, cela est déjà spécifié dans `l'urls.py`.

Une vue possible est la suivante :

Code : Python

```
def lire(request, id):
    try:
        article = Article.objects.get(id=id)
    except Article.DoesNotExist:
        raise Http404

    return render(request, 'blog/lire.html', {'article':article})
```

C'est assez verbeux, or les développeurs Django sont très friands de raccourcis. Un raccourci particulièrement utile ici `get_object_or_404`, permettant de récupérer un objet selon certaines conditions, ou renvoyer la page d'erreur 404 si aucun objet n'a été trouvé

Le même raccourci existe pour obtenir une liste d'objets : `get_list_or_404`.

Code : Python

```
# Il faut rajouter l'import get_object_or_404, attention !
from django.shortcuts import render, get_object_or_404

def lire(request, id):
    article = get_object_or_404(Article, id=id)
    return render(request, 'blog/lire.html', {'article':article})
```

Voici le template `lire.html` associé à la vue :

Code : Jinja

```
<h1>{{ article.titre }} <span class="small">dans {{
article.categorie.nom }}</span></h1>
<p class="infos">Rédigé par {{ article.auteur }}, le {{
article.date|date:"DATE_FORMAT" }}</p>
<div class="contenu">{{ article.contenu|linebreaks }}</div>
```

Recette du vendredi : la crêpe à la bière ! dans Recettes

Rédigé par Sax'z, le 13 juillet 2012

Préparation : 1h ; Cuisson : 2 min

Ingrédients (pour 20 crêpes environ) :

- 500 g de farine
- sel
- 6 œufs
- 2 cuillères à soupe d'huile
- 2 cuillères à soupe de rhum
- 25 cl de bière
- 50 cl de lait

Préparation :

Mettez la farine dans un saladier. Faites-y un puits et ajoutez l'huile, le rhum et les œufs.

Mélangez, puis ajoutez petit à petit le lait, puis la bière.

Laissez reposer 1 heure avant de faire cuire les crêpes. Les déguster avec du sucre en poudre ou de la cassonade.

La vue de notre article, à l'adresse <http://127.0.0.1:8000/blog/article/2> !

Des URL plus esthétiques

Comme vous pouvez le voir, nos URL contiennent pour le moment un ID afin de savoir quel article on souhaite afficher. C'est pratique, mais cela a l'inconvénient de ne pas être très parlant pour l'utilisateur... Pour remédier à cela, on voit de plus en plus fleurir sur le web des adresses contenant le titre de l'article réécrit. Par exemple, le Site du Zéro emploie cette technique à plusieurs endroits, comme ici avec l'adresse de ce cours : <http://www.siteduzero.com/tutoriel-3-700664-creez-vos-applications-web-avec-django.html>. Nous pouvons identifier dedans la chaîne "creez-vos-applications-web-avec-django" qui nous permet de savoir ce de quoi parle le lien, sans même avoir cliqué dessus. Cette chaîne est couramment appelée un **slug**. Et pour définir ce terme barbare, rien de mieux que Wikipédia :

Citation : Wikipédia - Slug (journalisme)

Un slug est en journalisme un label court donné à un article publié, ou en cours d'écriture. Il permet d'identifier l'article tout au long de sa production et dans les archives. Il peut contenir des informations sur l'état de l'article, afin de les catégoriser.

Nous allons intégrer la même chose à notre système de blog. Pour ça, il existe un type de champ un peu spécial dans les modèles : le `SlugField`. Il permet de stocker une chaîne de caractères, d'une certaine taille maximale.

Ainsi, notre modèle devient le suivant :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    slug = models.SlugField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField(null=True)
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose_name="Date de parution")

    def __unicode__(self):
    return self.titre
```

Maintenant, on peut aisément rajouter notre slug dans l'URL, en l'ajoutant en plus de l'ID lors de la construction d'une URL. Nous pouvons par exemple faire `/blog/article/1-titre-de-l-article`. La mise en œuvre est rapide également à mettre en place :

Code : Python - `blog/urls.py`

```
urlpatterns = patterns('blog.views',
    url(r'^$', 'accueil'),
    url(r'^article/(?P<id>\d+)-(P<slug>.+)$', 'lire'),
)
```

Code : Python - blog/views.py

```
def lire(request, id, slug):  
    # notre fonction identique à la précédente
```

Code : Jinja - accueil.html

```
<p><a href="{% url blog.views.lire article.id article.slug %}">Lire  
la suite</a>
```

Ici, on a le paramètre `slug` que l'on pourrait éviter de récupérer (en évitant sa capture parenthésée dans la regex du `urls.py`), mais l'utilisation du tag `{% url %}` aurait été plus compliquée...



Il existe également des sites qui n'utilisent qu'un slug dans les adresses. Dans ce cas, il faut faire attention à avoir des slugs unique dans votre base, ce qui n'est pas forcément le cas avec notre modèle ! Si vous créez un article "Bonne année" en 2012 puis un autre avec le même titre l'année suivante, ils auront le même slug... Il existe cependant des snippets qui contournent ce soucis.

L'inconvénient ici, c'est qu'il faut renseigner pour le moment le slug à la main à la création d'un article... Nous allons voir au chapitre suivant qu'il est possible d'automatiser son remplissage ! 😊

Les modèles est un principe important à comprendre et maîtriser. N'hésitez pas à relire ce chapitre plusieurs fois pour vous assurer de tout comprendre, vous en aurez besoin par la suite !

L'administration

Sur un bon nombre de sites, l'interface d'administration est un élément capital à ne pas négliger lors du développement. C'est cette partie qui permet en effet de pouvoir gérer les divers informations disponibles : les articles d'un blog, les comptes utilisateurs, etc.

Un des gros points fort de Django est qu'il génère de façon automatique l'administration en fonction de vos modèles. Celle-ci est personnalisable à souhait en quelques lignes et est très puissante.

Nous verrons dans ce chapitre comment déployer l'administration et la personnaliser.

Mise en place de l'administration

Les modules `django.contrib`

L'administration Django est **optionnelle** : il est tout à fait possible de développer un site sans l'utiliser. Pour cette raison, il est placé dans le module `django.contrib`, contenant un ensemble d'extensions fournies par Django, réutilisables dans n'importe quel projet. Ces modules sont bien pensés et vous permettent d'éviter de réinventer la roue à chaque fois.

Nous allons étudier ici le module `django.contrib.admin` qui génère l'administration, il en existe toutefois des dizaines, dont certains que l'on recroisera plus tard : `django.contrib.messages` (gestion de messages destinés aux visiteurs), `django.contrib.auth` (système d'authentification et de gestion des utilisateurs), ...

Accédons à cette administration !

Import des modules

Étant optionnel, il est nécessaire d'ajouter quelques lignes dans notre fichier de configuration pour pouvoir profiter de ce module. Ouvrons donc notre fichier `settings.py`. Comme vu précédemment, la variable `INSTALLED_APPS` permet à Django de savoir quels sont les modules à charger au démarrage du serveur. Si `django.contrib.admin` n'apparaît pas, ajoutez-le (l'ordre dans la liste n'a pas d'importance).

L'administration nécessite toutefois quelques dépendances pour fonctionner, également fournies dans `django.contrib`. Ces dépendances sont `django.contrib.auth`, `django.contrib.contenttypes` et `django.contrib.sessions`, qui sont normalement inclus de base dans le `INSTALLED_APPS`.

Au final, vous devriez avoir une variable `INSTALLED_APPS` qui ressemble à ceci :

Code : Python

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog', #Nous avons ajouté celui-ci lors de l'ajout de
l'application
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)
```

Enfin, le module `admin` nécessite aussi l'import de `middlewares`, fourni également par défaut dans Django et normalement inclus par défaut (ouf!) :

- `django.middleware.common.CommonMiddleware`,
- `django.contrib.sessions.middleware.SessionMiddleware`,
- `django.contrib.auth.middleware.AuthenticationMiddleware`

Sauvegardez le fichier `settings.py`, on en a fini avec. Désormais, lors du lancement du serveur, le module contenant l'administration sera importé.

Mise à jour de la base de données



Si jamais le module `django.contrib.auth` était déjà importé (ce qui est le cas avec la configuration livré par défaut avec Django 1.4), alors vous pouvez sauter cette étape, puisque vous avez déjà eu à faire cette étape dans le chapitre sur les modèles.

Pour fonctionner, il faut créer de nouvelles tables dans la base de données qui serviront à logger les actions des administrateurs, définir les droits de chacun, etc. Pour ce faire, c'est comme avec les modèles : `python manage.py syncdb`. A la première exécution, la commande shell va vous demander de renseigner des informations pour créer un compte **superutilisateur**, qui sera le seul compte à pouvoir accéder à l'administration pour commencer. Cette opération commence notamment par la suivante suivante.

Code : Console

```
You just installed Django's auth system, which means you don't have any superusers
```

Si vous sautez cette étape, il sera toujours possible de (re)créer ce compte via la commande `python manage.py createsuperuser`.

Intégration à notre projet : définissons-lui une adresse

Enfin, tout comme nos vues, il est nécessaire de dire au serveur "Hey, quand j'appelle cette URL, redirige moi vers l'admin". En effet, pour l'instant on a bien importé le module, mais on ne sait pas comment y accéder.

Comme pour les vues, il suffit d'aller faire un tour dans le fichier d'URLs. Ouvrez le fichier `urls.py` de votre projet (à l'intérieur du sous-dossier ayant le même nom que le projet) et non d'une application : l'administration est commune à toutes les applications d'un projet. On peut aller encore plus loin : l'administration peut être considéré comme **une application elle même !** Par défaut, Django a déjà indiqué plusieurs lignes pour l'administration, mais elle sont commentées. Au final, après avoir décommenté ces lignes, vous devez avoir un fichier `urls.py` ressemblant à :

Code : Python

```
from django.conf.urls import patterns, include, url

# Uncomment the next two lines to enable the admin:
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    # Examples:
    url(r'^accueil/$', 'blog.views.home'),
    [...]
    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

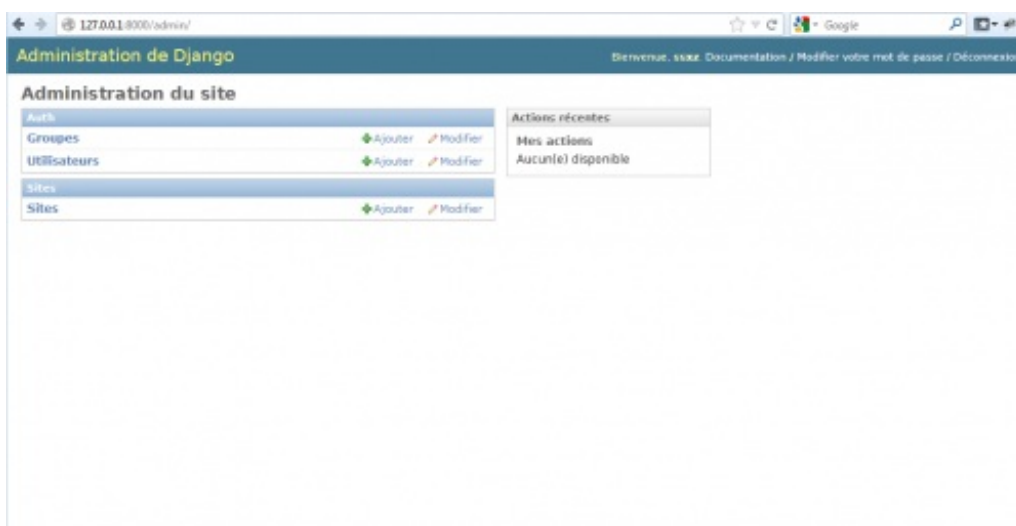
On voit que par défaut, l'administration sera disponible à l'adresse <http://localhost:8000/admin/>. Une fois les lignes décommentées, lancez le serveur Django (ou relancez-le s'il est déjà lancé). Vous pouvez dès lors accéder à l'administration depuis l'URL définie, il suffira juste de vous connecter avec le nom d'utilisateur et le mot de passe que vous avez spécifié lors du `syncdb` ou `createsuperuser`.



Écran de connexion de l'administration !

Première prise en main

Une fois que vous avez saisi vos identifiants super-utilisateur, vous devez arriver sur la page suivante :



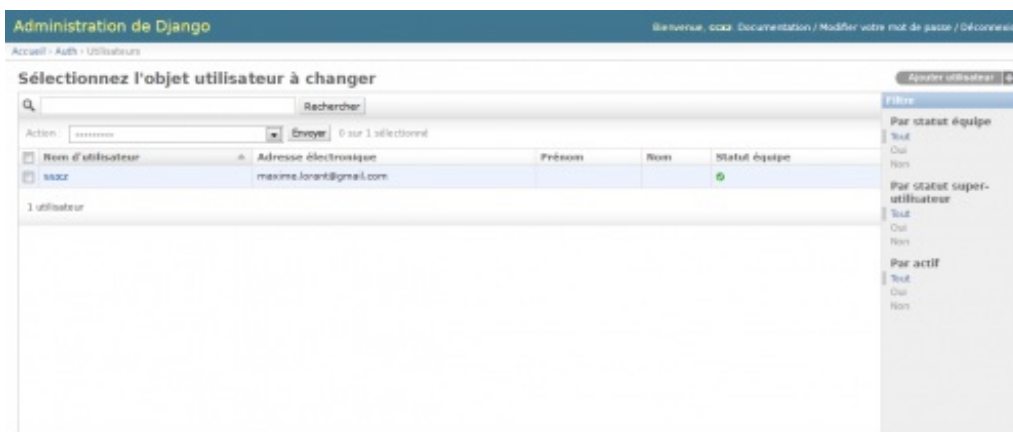
Accueil de l'administration

C'est encore un peu vide, mais vous inquiétez pas, on va bientôt pouvoir manipuler nos modèles Article et Catégorie, que l'on a rédigé dans le chapitre précédent.

Tout d'abord, faisons un petit tour des fonctionnalités disponibles. Sur cette page, vous avez la **liste des modèles que vous pouvez gérer**. Ces modèles sont au nombre de 3 : Groupes, Utilisateurs et Sites. Ce sont les modèles par défaut.

Chaque modèle possède ensuite une interface qui permet de réaliser les 4 opérations de base "**CRUD**" : Create, Read, Update, Delete (littéralement Créer, Lire, Mettre à jour, Supprimer)

Pour ce faire, allons dans l'administration des comptes sur notre site, en cliquant sur "*Utilisateurs*". Pour le moment, vous n'avez logiquement qu'un compte dans la liste, le votre :



Liste des utilisateurs

C'est à partir d'ici que l'on peut constater la puissance de cette administration : sans avoir écrit une seule ligne de code, il est possible de **manipuler la liste des utilisateurs dans tous les sens** : la filtrer selon certains paramètres, la trier avec certains champs, effectuer des actions sur certaines lignes, etc.

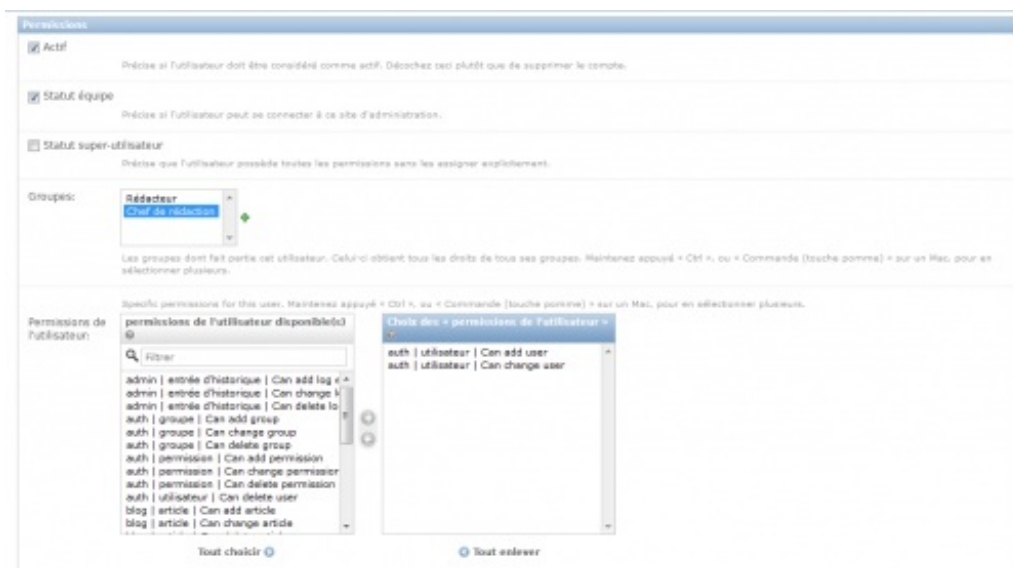
Pour essayer tout ça, on va d'abord créer un second compte utilisateur. Il suffit de cliquer sur le bouton "*Ajouter utilisateur*", disponible en haut à droite. Le premier formulaire vous demande de renseigner le nom d'utilisateur et le mot de passe. On peut déjà remarquer que les formulaires peuvent gérer des contraintes, et l'affichage d'erreurs :

Formulaire de création de comptes, après la validateur avec erreur. On remarque la mise en évidence des champs fautifs.

Une fois validé, vous accédez directement à formulaire plus complet, permettant de renseigner plus d'informations sur l'utilisateur qui vient d'être créé : informations personnelles mais aussi ses droits sur le site.

Django fournit de base une **gestion précise des droits**, par groupe et par utilisateur, offrant souplesse et rapidité dans l'attribution des droits. Ainsi, ici on peut voir que l'on peut assigner un ou plusieurs groupes à l'utilisateur, et des **permissions spécifiques**. D'ailleurs, vous pouvez créer un groupe sans quitter cette fenêtre en cliquant sur le "+" vert à côté des choix (qui est vide chez vous pour le moment) !

Deux champs importants sont également "*Statut équipe*" et "*Statut super-utilisateur*" : le premier permet de définir si l'utilisateur peut accéder au panel d'administration, et le second de donner "les pleins pouvoirs" à l'utilisateur.



Exemple d'édérations des permissions, ici j'ai crée deux groupes avant d'éditer l'utilisateur

Une fois que vous avez fini de gérer l'utilisateur, vous êtes redirigé vers la liste de tout à l'heure, avec une ligne en plus. Désormais, vous pouvez tester le tri, et les filtres qui sont disponible à la droite du tableau ! On va même voir plus tard comment définir les champs à afficher, quels filtres, etc.

Enfin, pour finir ce rapide tour, vous avez peut-être remarqué la présence d'un bouton "Historique" en haut de la fiche de chaque fiche utilisateur ou groupe. Celui-ci est très pratique, puisqu'il vous permet de suivre les modifications apportées, et donc voir rapidement l'évolution de l'objet sur le site. En effet, **chaque action effectuée via l'administration est inscrite dans un journal des actions**. De même, sur l'index vous avez la liste de vos dernières actions, vous permettant de voir ce que vous avez fait récemment, et pouvoir accéder rapidement aux liens, en cas d'erreur par exemple.

Administration de Django		
Accueil > Auth > Utilisateurs > MathX > Historique		
Historique des changements : MathX		
Date/heure	Utilisateur	Action
11 juillet 2012 19:04:28	ssxz (Maxime Lorant)	
11 juillet 2012 19:17:15	ssxz (Maxime Lorant)	Modifié password, is_staff, groupe et user_permissions.
11 juillet 2012 19:37:25	ssxz (Maxime Lorant)	Modifié password, first_name et last_name.

Historique des modifications d'un objet utilisateur

Administrons nos propres modèles

Pour le moment, on a vu comment manipuler les données des objets de base de Django, ceux concernant les utilisateurs. Il serait bien maintenant de **faire de même avec nos propres modèles**. Comme on l'a déjà dit avant, l'administration est auto-généré : vous n'aurez pas à écrire beaucoup de lignes pour obtenir le même résultat qu'au dessus. En réalité, 4 lignes suffisent : créer un fichier `admin.py` dans le répertoire `blog/` et insérez ces lignes :

Code : Python

```
from django.contrib import admin
from blog.models import Categorie, Article

admin.site.register(Categorie)
admin.site.register(Article)
```

Ici, on indique à Django de **prendre en compte les modèles Article et Categorie** dans l'administration. Rafraichissez la page (relancez le serveur Django si nécessaire) et vous devez voir apparaitre une nouvelle section, pour notre blog :



La 2eme section nous permet enfin de gérer notre blog !

Les fonctionnalités sont les mêmes que celles que l'on a pour les utilisateurs : on peut éditer des articles, des catégories, les supprimer, voir l'historique, etc. Vous pouvez désormais créer vos articles depuis cette interface et voir le résultat depuis les éventuels templates que vous avez fait.

Comme vous pouvez le voir, l'administration **prend en compte notre clé étrangère** sur la catégorie, et maintenant que l'on a vu vous pouvez faire de même avec l'auteur !

Comment cela fonctionne t-il ?

Vu comme ça, ça paraît magique : 4 lignes dans un fichier, et tout fonctionne. En réalité, le processus derrière utilisé par Django est tout aussi simple : au lancement du serveur, le framework charge le fichier `urls.py` et tombe sur la ligne `admin.autodiscover()`. Cette méthode va aller chercher dans chaque application installée (celles qui sont listées dans `INSTALLED_APPS`) un fichier `admin.py`, et si il existe exécuter son contenu.

Ainsi, si l'on souhaite activer l'administration pour toutes nos applications, il suffit de créer un fichier `admin.py` à chaque fois, et appeler la méthode `register()` de `admin.site` sur chacun de nos modèles.

On peut alors deviner que le module `django.contrib.auth` contient son propre fichier `admin.py`, qui génère l'administration des utilisateurs et des groupes.

De même le module Site, que l'on a ignoré depuis le début fonctionne de la même façon. Ce module sert à pouvoir faire plusieurs sites, avec le même code. Il est rarement utilisé, et si vous souhaitez le désactiver, il vous suffit de commenter la ligne surlignée ci-dessous, dans votre `settings.py` :

Code : Python

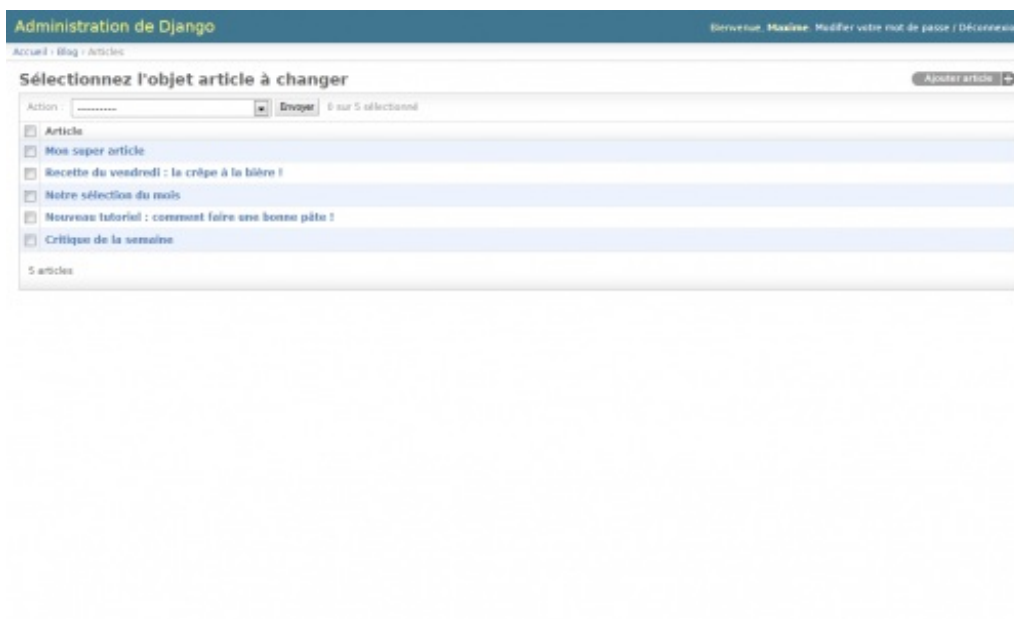
```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    'blog',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
)
```

Personnalisons l'administration

Avant tout, créez quelques articles depuis l'administration, si ce n'est déjà fait. Cela va vous permettre de tester tout au long de ce chapitre les différents exemples qui seront donnés

Modifier l'aspect des listes

Dans un premier temps, nous allons tout d'abord voir comment améliorer la liste. En effet pour le moment, les listes que l'on a avec nos modèles sont un assez vides :



Notre liste d'articles, avec uniquement le titre comme colonne

Le tableau ne contient qu'une colonne, qui est le titre de notre article. Cette colonne n'est pas un hasard : c'est en réalité le résultat de la méthode `__unicode__` que l'on a écrit dans notre modèle !

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField()
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
        verbose_name="Date de parution")
    categorie = models.ForeignKey(Categorie)

    def __unicode__(self):
        return self.titre
```

C'est utile, mais on aimerait bien pouvoir gérer plus facilement nos articles : les trier selon certains champs, filtrer par catégorie, etc. Pour cela, on va devoir créer une nouvelle classe, qui va nous permettre de spécifier tout ça. Tout cela se fait encore dans le fichier `admin.py`. Pour rappel, on a pour le moment ceci :

Code : Python

```
from django.contrib import admin
from blog.models import Categorie, Article

admin.site.register(Categorie)
admin.site.register(Article)
```

On va donc créer une nouvelle classe pour chaque modèle. Notre classe héritera de `admin.ModelAdmin` et aura principalement 5 attributs :

Nom de l'attribut	Utilité
<code>list_display</code>	Liste des champs du modèle à afficher dans le tableau
<code>list_filter</code>	Liste des champs selon lesquels on pourra filtrer les données
<code>date_hierarchy</code>	Permet de filtrer par date de façon intuitive

ordering	Tri par défaut du tableau
search_fields	Configuration du champ de recherche

Nous pouvons alors rédiger notre première classe adapté au modèle Article :

Code : Python

```
class ArticleAdmin(admin.ModelAdmin):
    list_display = ('titre', 'auteur', 'date')
    list_filter = ('auteur', 'categorie',)
    date_hierarchy = 'date'
    ordering = ('date',)
    search_fields = ('titre', 'contenu')
```

Ces attributs définissent les règles suivantes :

- Le tableau affiche les champs "titre", "auteur" et "date". Notez que les en-têtes sont nommés selon leurs attributs verbose_name respectifs.
- Il est possible de filtrer selon les différents auteurs et la catégorie des articles (menu de droite)
- L'ordre par défaut est la date de parution, dans l'ordre croissant (du plus ancien au plus récent)
- Il est possible de chercher les articles contenant un mot, soit dans leur titre, soit dans leur contenu
- Enfin, il est possible de voir les articles publiés sur une certaine période (1ère ligne au dessus du tableau)

Maintenant, il faut spécifier à Django de prendre en compte ces données pour le modèle Article. Pour ce faire, modifions la ligne `admin.site.register(Article)`, en ajoutant un second paramètre :

Code : Python

```
admin.site.register(Article, ArticleAdmin)
```

Avec ce deuxième argument, Django prendra en compte les règles qui ont été spécifiées dans la classe ArticleAdmin.

Code : Python

```
from django.contrib import admin
from blog.models import Categorie, Article

class ArticleAdmin(admin.ModelAdmin):
    list_display = ('titre', 'auteur', 'date')
    list_filter = ('auteur', 'categorie',)
    date_hierarchy = 'date'
    ordering = ('date',)
    search_fields = ('titre', 'contenu')

admin.site.register(Categorie)
admin.site.register(Article, ArticleAdmin)
```

La même liste, bien plus complète, et plus pratique !

Et voilà, on peut voir les différents changements que l'on souhaite. Vous pouvez bien sûr, modifier à votre sauce : ajouter le champ "Catégorie" dans le tableau, changer le tri...

Pour finir, on va voir comment créer des colonnes plus complexes. Il peut arriver que vous ayez envie d'**afficher une colonne après un certain traitement**. Par exemple, afficher les 40 premiers caractères de notre article. Pour ce faire, on va devoir créer une méthode dans notre `ModelAdmin`, qui va se charger de renvoyer ce que l'on souhaite, et la lier à notre `list_display`.

Créons tout d'abord notre méthode. Celles de notre `ModelAdmin` auront toujours la même structure :

Code : Python

```
def apercu_contenu(self, article):
    """
    Retourne les 40 premiers caractères du contenu de l'article. Si il
    y a plus de 40 caractères, on rajoute des points de suspension
    """
    text = article.contenu[0:40]
    if len(article.contenu) > 40:
        return '%s...' % text
    else:
        return text
```

La méthode prend **en argument l'instance de l'article**, et nous permet d'accéder à tous ces attributs. Ensuite, il suffit de faire ce que l'on souhaite, puis renvoyer une chaîne de caractères. On a plus qu'à intégrer ça dans notre `ModelAdmin`.



Et comment on le met dans notre `list_display` ?

Il faut traiter la fonction comme un champ. Il suffit donc de rajouter '`apercu_contenu`' à la liste, Django s'occupe du reste. Pour ce qui est de l'en-tête, il faudra par contre ajouter une ligne supplémentaire pour spécifier le titre de la colonne :

Code : Python

```
# -*- coding:utf8 -*-
from django.contrib import admin
from blog.models import Categorie, Article

class ArticleAdmin(admin.ModelAdmin):
    list_display = ('titre', 'categorie', 'auteur', 'date',
'apercu_contenu')
    list_filter = ('auteur', 'categorie',)
    date_hierarchy = 'date'
    ordering = ('date',)
    search_fields = ('titre', 'contenu')

    def apercu_contenu(self, article):
        """
        Retourne les 40 premiers caractères du contenu de l'article. Si il
        y a plus de 40 caractères, on rajoute des points de suspension
        """
```

```

text = article.contenu[0:40]
if len(article.contenu) > 40:
    return '%s...' % text
else:
    return text

# En-tête de notre colonne
aperçu_contenu.short_description = 'Aperçu du contenu'

admin.site.register(Categorie)
admin.site.register(Article, ArticleAdmin)

```

Et on obtient notre nouvelle colonne avec les premiers mots de chaque article :

Titre	Categorie	Auteur	Date de parution	Aperçu du contenu
<input type="checkbox"/> Mon super article	Autres	Box'z	13 juillet 2012 20:28:25	
<input type="checkbox"/> Recette du vendredi : la crêpe à la bière !	Recettes	Box'z	13 juillet 2012 20:34:11	Préparation : 1h Cuisson : 2 min 30...
<input type="checkbox"/> Notre sélection du mois	Recettes	MathX	13 juillet 2012 20:37:55	Début de mon article
<input type="checkbox"/> Nouveau tutoriel : comment faire une bonne pâte !	Autres	La crêpière	13 juillet 2012 20:38:59	A rédiger plus tard
<input type="checkbox"/> Critique de la semaine	Autres	MathX	13 juillet 2012 20:40:43	Lorem ipsum, non ?

Modifier le formulaire d'édition

On va désormais s'occuper du formulaire d'édition. Pour le moment on a un formulaire affichant tous les champs, hormis la date de publication (à cause du paramètre `auto_now_add=True`) :

Administration de Django

Accueil | Blog | Articles | Recette du vendredi : la crêpe à la bière | Modifier

Modification de l'article

Titre: Recette du vendredi : la crêpe à la bière

Auteur: Box'z

Contenu: Préparation : 1h | Cuisson : 2 min
 Ingrédients (pour 20 crêpes environ) :
 - 500 g de farine
 - 6 œufs
 - 50 cl de lait
 - 2 cuillères à soupe d'huile
 - 2 cuillères à soupe de rhum
 - 25 cl de bière (Bois)
 - sel

Categorie: Recettes

Supprimer Enregistrer et ajouter un nouveau Enregistrer et continuer les modifications Enregistrer

Le formulaire d'édition d'un article par défaut

L'ordre d'apparition des champs dépend actuellement de l'ordre de déclaration dans notre modèle. Nous allons ici séparer le contenu des autres champs.

Tout d'abord, modifions l'ordre via un nouvel attribut dans notre `ModelAdmin:fields`. Cet attribut prend une liste de champs, qui seront affichés dans l'ordre souhaité. Cela va nous permettre de cacher des champs (inutile dans le cas présent) et, bien évidemment, de changer leur ordre :

Code : Python

```
fields = ('titre', 'auteur', 'categorie', 'contenu')
```

On observe peu de changements, à part le champ *Catégorie* qui est désormais au dessus de contenu.

The screenshot shows the Django Admin interface for editing an article. The form is titled 'Modification de l'article'. It contains the following fields: 'Titre' (with the value 'Recette du vendredi : la crêpe à la bière'), 'Auteur' (with the value 'Bix'z'), 'Catégorie' (with a dropdown menu showing 'Recettes'), and 'Contenu' (a large text area containing a recipe for crêpes). The 'Catégorie' field is now positioned above the 'Contenu' field. At the bottom of the form, there are buttons for 'Supprimer', 'Enregistrer et ajouter un nouveau', 'Enregistrer et continuer les modifications', and 'Enregistrer'.

Notre formulaire, avec une nouvelle organisation des champs

Pour le moment, notre formulaire est dans un **unique fieldset** (ensemble de champs). Conséquence : tous les champs sont les uns à la suite, sans distinction. On peut **hiérarchiser** cela en utilisant un attribut plus complexe que `fields`.

Comme exemple, nous allons mettre les champs titre, auteur et catégorie dans un fieldset et contenu dans un autre.

Code : Python

```
fieldsets = (
    # Fieldset 1 : Meta-info (titre, auteur...)
    ('Général', {
        'classes': ['collapse'],
        'fields': ('titre', 'auteur', 'categorie')
    }),
    # Fieldset 2 : Contenu de l'article
    ('Contenu de l'article', {
        'description': 'Le formulaire accepte les balises HTML. Utilisez-les à bon escient !',
        'fields': ('contenu', )
    })
)
```

Voyons pas à pas la construction de ce tuple :

1. On a 2 élément dans le tuple fieldset, qui correspond à nos **2 fieldsets distincts**
2. Chaque élément contient est un tuple contenant **exactement 2 informations** : son nom, et les informations sur son

contenu, sous forme de dictionnaire.

3. Ce dictionnaire contient trois type de données
 - a. *fields*: liste des champs à afficher dans le fieldset
 - b. *description* : une description qui sera affichée en haut du fieldset, avant le 1er champ
 - c. *classes* : des classes CSS supplémentaires à appliquer sur le fieldset (par défaut il en existe 3 : wide, extrapretty et collapse)



Si vous mettez en place un fieldset, il faut retirer l'attribut field. C'est soit l'un, soit l'autre !

Ici, nous avons donc séparé les champs en 2 fieldsets et affiché quelques informations supplémentaires pour aider à la saisie. Au final, nous avons le fichier admin.py suivant :

Code : Python

```
# -*- coding:utf8 -*-
from django.contrib import admin
from blog.models import Categorie, Article

class ArticleAdmin(admin.ModelAdmin):

    # Configuration de la liste d'articles
    list_display = ('titre', 'categorie', 'auteur', 'date')
    list_filter = ('auteur', 'categorie', )
    date_hierarchy = 'date'
    ordering = ('date', )
    search_fields = ('titre', 'contenu')

    # Configuration du formulaire d'édition
    fieldsets = (
        # Fieldset 1 : Meta-info (titre, auteur...)
        ('Général', {
            'classes': ['collapse'],
            'fields': ('titre', 'auteur', 'categorie')
        }),
        # Fieldset 2 : Contenu de l'article
        ('Contenu de l\'article', {
            'description': 'Le formulaire accepte les balises HTML. Utilisez-les à bon escient !',
            'fields': ('contenu', )
        }),
    )

    # Colonnes personnalisées
    def apercu_contenu(self, article):
        """
        Retourne les 40 premiers caractères du contenu de l'article. Si il y a plus de 40 caractères, on rajoute des points de suspension """
        text = article.contenu[0:40]
        if len(article.contenu) > 40:
            return '%s...' % text
        else:
            return text

    apercu_contenu.short_description = 'Aperçu du contenu'

admin.site.register(Categorie)
admin.site.register(Article, ArticleAdmin)
```

... qui rend la page suivante :

Notre formulaire, mieux présenté qu'avant



Si ni le champ `fields`, ni le champ `fieldset` sont présents, Django affichera par défaut tous les champs qui ne sont pas des `AutoField`, et qui ont l'attribut `editable` à `True` (qui est le cas par défaut de nombreux champs). Comme on l'a vu, l'ordre des champs sera alors celui du modèle.

Retour sur notre problème de slug

Souvenez-vous, au chapitre précédent nous avons parlé des slugs, ces chaînes de caractères qui permettent d'identifier un article dans notre URL. Dans notre zone d'administration actuellement, ce champ est ignoré... Nous souhaitons toutefois le remplir, mais en plus que cela se fasse automatiquement !

Tout d'abord, nous allons ajouter notre champ `slug` dans notre `fieldset` :

Code : Python



```
fieldsets = (
    # Fieldset 1 : Meta-info (titre, auteur...)
    ('Général', {
        'classes': ['collapse', ],
        'fields': ('titre', 'slug', 'auteur', 'categorie')
    }),
    # Fieldset 2 : Contenu de l'article
    ('Contenu de l\'article', {
        'description': 'Le formulaire accepte les balises HTML. Utilisez-les à bon escient !',
        'fields': ('contenu', )
    }),
)
```

Ainsi, on a notre champ que l'on peut éditer à la main. Mais nous pouvons aller plus loin, en rajoutant une option qui remplit instantanément ce champ en Javascript. Pour ce faire, il existe un attribut aux classes `ModelAdmin` qui est `prepopulated_fields`. Ce champ a pour principal usage de remplir les champs de type `SlugField` en fonction d'un ou plusieurs autres champs.

Code : Python

```
prepopulated_fields = {'slug': ('titre', ), }
```

Ici, notre champ slug est rempli automatiquement en fonction du champ titre. Il est possible bien entendu de concaténer plusieurs chaînes, si vous voulez par exemple faire apparaître l'auteur.

Général (Masquer)	
Titre:	<input type="text" value="Les légendaires crêpes basques !"/>
Slug:	<input type="text" value="les-legendaires-crepes-basques"/>
Auteur:	<input type="text" value="Ssx`z"/>
Categorie:	<input type="text" value="Découvertes"/>  

Exemple d'utilisation du `prepopulated_fields`

Les formulaires

Si vous avez déjà fait du web auparavant, vous avez forcément dû concevoir des formulaires. Entre le code HTML à réaliser, la validation des données entrées par l'utilisateur et la mise à jour de celles-ci dans la base de données, réaliser un formulaire était un travail fastidieux. Heureusement, Django est là pour vous simplifier la tâche !

Créer un formulaire

La déclaration d'un formulaire est très similaire à la déclaration d'un modèle. Il s'agit également d'une classe héritant d'une classe mère fournie par Django. Les attributs eux aussi correspondent aux champs du formulaire.

Si les modèles ont leurs fichiers `models.py`, les formulaires n'ont malheureusement pas la chance d'avoir un endroit qui leur est défini. Cependant, toujours dans une optique de code structuré, nous nous invitons à créer dans chaque application (bien que pour le moment nous n'en ayons qu'une) un fichier `forms.py` dans lequel nous créerons nos formulaires.

Un formulaire hérite donc de la classe mère `Form` du module `django.forms`. Tous les champs sont bien évidemment également dans ce module et reprennent la plupart du temps les mêmes noms que ceux des modèles. Bref exemple d'un formulaire de contact :

Code : Python

```
from django import forms

class ContactForm(forms.Form):
    sujet = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    envoyeur = forms.EmailField(label="Votre adresse mail")
    renvoi = forms.BooleanField(help_text="Cochez si vous souhaitez  
obtenir une copie du mail envoyé.", required=False)
```

Toujours très similaire aux formulaires, un champ peut avoir des arguments qui lui sont propres (ici `max_length` pour `sujet`), ou avoir des arguments génériques à tous les champs (ici `label`, `help_text`, `widget` et `required`).

Un `CharField` enregistre toujours du texte. Notons une différence avec le `CharField` des modèles : l'argument `max_length` devient optionnel. L'attribut `message` qui est censé recueillir de grands et longs textes n'en a pas.



Ne devrait-on pas utiliser un `TextField` comme dans les modèles pour `message` dans ce cas ?

Django ne propose pas de champ `TextField` similaire aux modèles. Tant que l'on recueille du texte, il faut utiliser un `CharField`. En revanche, il semble logique que les boîtes de saisie pour le sujet et pour le message ne doivent pas avoir la même taille ! Le message est souvent beaucoup plus long. Pour ce faire, Django propose en quelque sorte de « maquiller » les champs du formulaire grâce aux widgets. Ces derniers transforment le code HTML pour le rendre plus adapté à la situation actuelle. Nous avons utilisé ici le widget `forms.Textarea` pour le champ `message`. Celui-ci fera en sorte d'agrandir considérablement la boîte de saisie pour le champ et la rendre plus confortable pour le visiteur.

Il existe bien d'autres widgets (tous également dans `django.forms`) : `PasswordInput` (pour cacher le mot de passe), `DateInput` (pour rentrer une date), `CheckboxInput` (pour avoir une case à cocher), etc. N'hésitez pas à consulter la documentation Django pour avoir une liste exhaustive !

Il est très important de comprendre la logique des formulaires. Lorsqu'on choisit un champ, on le fait selon le type de données qu'on doit recueillir (du texte, un nombre, un email, une date, ...). C'est le champ qui s'assurera que ce qu'a rentré l'utilisateur est valide. En revanche, tout ce qui concerne l'apparence du formulaire concerne les widgets.

Généralement, il n'est pas utile de spécifier des widgets pour tous les champs. Par exemple, le `BooleanField`, qui recueille un booléen, utilisera par défaut le widget `CheckboxInput` et l'utilisateur verra donc une boîte à cocher. Néanmoins, si cela ne vous convient pas pour une quelconque raison, vous pouvez toujours changer.

Revenons rapidement à notre formulaire : l'attribut `email` contient un `EmailField`. Ce dernier s'assurera que l'utilisateur a bien envoyé une adresse mail correcte et le `BooleanField` de `renvoi` affichera une boîte à cocher, comme nous l'avons expliqué ci-dessus.

Ces deux derniers champs possèdent des arguments génériques : `label`, `help_text` et `required`. `label` permet de changer le nom de

la boîte de saisie qui est généralement définie selon le nom de la variable. `help_text` permet de rajouter un petit texte d'aide concernant le champ. Celui-ci apparaîtra généralement à droite ou en bas du champ. Finalement, `required` permet d'indiquer si le champ doit obligatoirement être rempli ou non. Il s'agit d'une petite exception lorsque cet argument est utilisé avec `BooleanField` car si la boîte n'est pas cochée, Django considère que le champ est invalide car laissé "vide", ce qui oblige l'utilisateur à cocher la boîte, et ce n'est pas ce que nous souhaitons ici.

Utiliser un formulaire dans une vue

Nous avons vu comment créer un formulaire. Passons à la partie la plus intéressante : utiliser celui-ci dans une vue.

Avant tout, il faut savoir qu'il existe deux types principaux de requêtes HTTP (le protocole, "langage", que nous utilisons pour communiquer sur le web). Le type de requête le plus souvent utilisé est le type "GET". Il demande une page et le serveur web le lui renvoie, aussi simplement que ça. Le deuxième type, qui nous intéresse le plus ici, est "POST". Celui-ci en revanche, va également demander une page du serveur, mais va aussi envoyer des données à celui-ci, généralement depuis un formulaire. Donc, pour savoir si l'utilisateur a complété un formulaire ou non, on se fie à la requête HTTP qui nous est transmise :

- GET : pas de formulaire envoyé
- POST : formulaire complété et envoyé

L'attribut `method` de l'objet `request` passé à la vue indique le type de requête (il peut être mis à "GET" ou "POST"). Les données envoyées par l'utilisateur via une requête POST sont accessibles sous forme d'un dictionnaire depuis `request.POST`. C'est ce dictionnaire que nous passerons comme argument lors de l'instanciation du formulaire pour vérifier si les données sont valides ou non.

Une vue qui utilise un formulaire suit la plupart du temps une certaine procédure. Cette procédure, bien que non-officielle, est reprise par la majorité des développeurs Django, probablement à cause de son efficacité.

La voici :

Code : Python

```
from blog.forms import ContactForm

def contact(request):
    if request.method == 'POST': # S'il s'agit d'une requête POST
        form = ContactForm(request.POST) # On reprend les données

        if form.is_valid(): # On vérifie que les données envoyées
            sont valides

            # Ici on peut traiter les données du formulaire
            sujet = form.cleaned_data['sujet']
            message = form.cleaned_data['message']
            envoyeur = form.cleaned_data['envoyeur']
            renvoi = form.cleaned_data['renvoi']

            destinataires = ['webmaster@crepes-bretonnes.com']
            if renvoi:
                destinataires.append(envoyeur)

            # On envoie le mail grâce à une fonction fourni par
            Django
            from django.core.mail import send_mail
            send_mail(sujet, message, envoyeur, destinataires)

            return HttpResponseRedirect('/merci-contact/') # On
            redirige l'utilisateur vers une page de confirmation
        else: # Si c'est pas du POST, c'est probablement une requête
            GET
            form = ContactForm() # On crée un formulaire vide

            return render(request, 'contact.html', {'form': form,}) # On
            envoie le template avec le formulaire qu'on a construit plus haut
```

Si le formulaire est valide, un nouvel attribut de l'objet `form` est apparu qui nous permettra d'accéder aux données : `cleaned_data`.

Ce dernier va renvoyer un dictionnaire contenant comme clés, les noms de vos différents champs (les mêmes noms qui ont été renseignés dans la déclaration de la classe), et comme valeurs les données validées de chaque champ. Par exemple, on pourrait accéder au sujet du message ainsi :

Code : Python

```
print form.cleaned_data["sujet"]
"Le super sujet qui a été envoyé"
```

Coté utilisateur, ça se passe en trois étapes :

1. Le visiteur arrive sur la page, complète le formulaire et l'envoie
2. Si le formulaire est faux, on retourne la même page tant que celui-ci n'est pas correct
3. Si le formulaire est correct, on le redirige vers une autre page

Il est important de remarquer que si le formulaire est faux, il n'est pas remis à zéro ! Un formulaire vide est créé lorsque la requête est de type GET. Par la suite, elles seront toujours de type POST. Dès lors, si les données sont fausses, on retourne encore une fois le template avec le formulaire invalide. Celui-ci contient encore les données fausses et des messages d'erreur pour aider l'utilisateur à le corriger.

Si nous avons fait la vue, il ne reste plus qu'à faire le template. Ce dernier est très simple à faire car Django va automatiquement générer le code HTML des champs du formulaire. Il faut juste spécifier une balise form et un bouton. Exemple :

Code : Jinja

```
<form action="{% url blog.views.contact %}" method="post">{%
csrf_token %}
{{ form.as_p }}
<input type="submit" value="Submit" />
</form>
```

Chaque formulaire (valide ou non) possède plusieurs méthodes qui permettent de générer le code HTML des champs du formulaire de plusieurs manières. Ici, il va le générer sous la forme d'un paragraphe (as_p, p pour la balise <p>), mais il pourrait tout aussi bien le générer sous la forme de tableau grâce à la méthode as_table ou ou sous la forme de liste grâce à as_ul. Utilisez ce qui vous pensez être le plus adapté.

D'ailleurs, ces méthodes ne créent pas seulement le code HTML des champs, mais rajoutent aussi les messages d'erreurs lorsqu'un champ n'est pas correct !

Dans le cas actuel, le code suivant sera généré (avec un formulaire vide) :

Code : HTML

```
<p><label for="id_sujet">Sujet:</label> <input id="id_sujet"
type="text" name="sujet" maxlength="100" /></p>
<p><label for="id_message">Message:</label> <textarea
id="id_message" rows="10" cols="40" name="message"></textarea></p>
<p><label for="id_envoyeur">Votre adresse mail:</label> <input
type="text" name="envoyeur" id="id_envoyeur" /></p>
<p><label for="id_renvoi">Renvoi:</label> <input type="checkbox"
name="renvoi" id="id_renvoi" /> <span class="helptext">Cochez si
vous souhaitez obtenir une copie du mail envoyé.</span></p>
```

Et voici l'image du rendu (bien entendu libre à vous de l'améliorer avec un peu de CSS) :

Sujet:

Message:

Votre adresse mail: Renvoi: ☐ Cochez si vous souhaitez obtenir une copie du mail envoyé.

C'est quoi ce tag : {% csrf_token %} ?

Ce tag est une fonctionnalité très pratique de Django. Il empêche les attaques de type CSRF (Cross-site request forgery). Imaginons qu'un de vos visiteurs obtienne l'URL qui permet de supprimer tous les articles de votre blog. Cependant, seul un administrateur peut effectuer cette action. Votre visiteur peut alors tenter de vous rediriger vers cette URL à votre insu, ce qui supprimerait tous vos articles ! Pour éviter ce genre d'attaques, Django va sécuriser le formulaire en y ajoutant un code unique et caché qu'il gardera de côté. Lorsque l'utilisateur renverra le formulaire, il va également envoyer le code avec. Django pourra alors vérifier si le code envoyé est bel et bien le code qu'il a généré et mis de côté. Si c'est le cas, le framework sait que l'administrateur a vu le formulaire et qu'il est sûr de ce qu'il fait !

Créons nos propres règles de validation

Imaginons que nous, administrateurs du blog sur les crêpes bretonnes, recevons souvent des messages impolis des fanatiques de la pizza italienne depuis le formulaire de contact. Chacun ses goûts, mais nous avons d'autres chats à fouetter !

Pour éviter de recevoir ces messages, nous avons eu l'idée d'intégrer un filtre dans notre formulaire pour que celui-ci soit invalide si le message contient le mot "pizza". Heureusement pour nous, il est facile d'ajouter de nouvelles règles de validation sur un champ. Il y a deux méthodes : soit le filtre ne s'applique qu'à un seul champ et ne dépend pas des autres, soit le filtre dépend des données des autres champs.

Pour la première méthode (la plus simple), il faut ajouter une méthode à la classe dont le nom doit obligatoirement commencer par `clean_`, puis être suivi par le nom de la variable du champ. Par exemple, si nous souhaitons filtrer le champ "message", il faut rajouter une méthode semblable à celle-ci :

Code : Python

```
def clean_message(self):
    message = self.cleaned_data['message']
    if "pizza" in message:
        raise forms.ValidationError("On ne veut pas entendre parler
de pizza !")

    return message #Ne pas oublier de renvoyer le contenu du champ
traité
```

On récupère le contenu du message comme depuis une vue, en utilisant l'attribut `cleaned_data` qui retourne toujours un dictionnaire. Dès lors, on vérifie si le message contient bien le mot pizza, c'est si c'est le cas, on retourne une exception avec une erreur (il est important d'utiliser l'exception `forms.ValidationError` !). Django se servira du contenu de l'erreur passé en argument pour indiquer quel champ n'a pas été validé et pourquoi.

Le rendu HTML nous donne ceci avec des données invalides après traitement du formulaire :

Sujet:

- On ne veut pas entendre parler de pizza !

Les pizzas italiennes c'est trop bien !
Les crêpes bretonnes c'est trop nul !

Message:

Votre adresse mail:

Renvoi: ☐ Cochez si vous souhaitez obtenir une copie du mail envoyé.

Maintenant, imaginons que nos fanatiques de la pizza italienne se soient adoucis et nous avons décidé d'être moins sévères, nous ne rejeterons que les messages qui possèdent le mot pizza dans le message et dans le sujet (juste parler de pizzas dans le message serait accepté). Étant donné que la validation dépend de plusieurs champs en même temps, nous devons écraser la méthode `clean` hérité de la classe mère `Form`. Les choses se compliquent un petit peu :

Code : Python

```
def clean(self):
    cleaned_data = super(ContactForm, self).clean()
    sujet = cleaned_data.get('sujet')
    message = cleaned_data.get('message')

    if sujet and message: #Est-ce que sujet et message sont valides
        ?
        if "pizza" in sujet and "pizza" in message:
            raise forms.ValidationError("Vous parlez de pizzas dans
le sujet ET le message ? Non mais ho !")

        return cleaned_data #On n'oublie pas de renvoyer les données si
tout est OK
```

La première ligne de la méthode permet d'appeler la méthode `clean` héritée de `Form`. En effet, si nous avons un formulaire d'inscription qui prend l'adresse mail de l'utilisateur, avant de vérifier si celle-ci a déjà été utilisée, il faut laisser Django vérifier si l'adresse mail est valide ou non. Appeler la méthode mère permet au framework de vérifier tous les champs comme d'habitude pour s'assurer que ceux-ci soient corrects, suite à quoi nous pouvons traiter ces données en sachant qu'elles ont déjà passé la validation basique.

La méthode mère `clean` va également renvoyer un dictionnaire avec toutes les données valides. Dans notre dernier exemple, si l'adresse mail spécifiée était incorrecte, elle ne sera pas reprise dans le dictionnaire renvoyé. Pour savoir si les valeurs que nous souhaitons filtrer sont valides, on utilise la méthode `get` du dictionnaire qui renvoie la valeur d'une clé si elle existe, et renvoie `None` sinon. Par la suite, nous vérifions que les valeurs des variables ne sont pas à `None` (`if sujet and message`) et nous les traitons comme d'habitude.

Voilà comment ce que donne le formulaire lorsqu'il ne passe pas la validation que nous avons écrite :

- Vous parlez de pizzas dans le sujet ET le message ? Non mais ho !

Sujet:

Les pizzas c'est trop cool !

Message:

Votre adresse mail:

Renvoi: ☐ Cochez si vous souhaitez obtenir une copie du mail envoyé.

Il faut cependant remarquer une chose : le message d'erreur est tout en haut et n'est plus lié aux champs qui n'ont pas passé la vérification. Si sujet et message étaient les derniers champs du formulaire, le message d'erreur serait tout de même tout en haut. Pour éviter cela, il est possible d'assigner une erreur à un champ précis :

Code : Python

```
def clean(self):
    cleaned_data = super(ContactForm, self).clean()
    sujet = cleaned_data.get('sujet')
    message = cleaned_data.get('message')

    if sujet and message: #Est-ce que sujet et message sont valides
        ?
        if "pizza" in sujet and "pizza" in message:
            msg = u"Vous parlez déjà de pizzas dans le sujet, n'en
parlez plus dans le message !"
            self._errors["message"] = self.error_class([msg])

            del cleaned_data["message"]

    return cleaned_data
```

Le début est identique, en revanche, si les deux champs contiennent le mot pizza, on ne renvoie plus une exception mais on définit une liste d'erreurs à un dictionnaire (self._errors) avec comme clé le nom du champ concerné. Cette liste doit obligatoirement être le résultat d'une fonction de la classe mère Form (self.error_class) et celle-ci doit recevoir une liste de chaînes de caractères qui contiennent les différents messages d'erreurs.

Une fois l'erreur indiquée, il ne faut pas oublier de supprimer la valeur du champ du dictionnaire car celle-ci n'est pas valide. Rappelez-vous, un champ manquant dans le dictionnaire cleaned_data correspond à un champ invalide !

Et voilà le résultat :

Sujet:

- Vous parlez déjà de pizzas dans le sujet, n'en parlez plus dans le message !

Pizza pizza !

Message:

Votre adresse mail:

Renvoi: ☐ Cochez si vous souhaitez obtenir une copie du mail envoyé.

Des formulaires à partir de modèles

Dernière fonctionnalité que nous verrons à propos des dictionnaires : les ModelForm. Il s'agit de formulaires générés automatiquement à partir d'un modèle, ce qui évite la plupart du temps à devoir écrire un formulaire pour chaque modèle créé. C'est un gain de temps non négligeable ! Ils reprennent la plupart des caractéristiques des formulaires classiques et s'utilisent comme eux.

Nous avons dans le chapitre sur les modèles créé une classe Article. La voici (à la dernière modification) :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField(null=True)
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose_name="Date de parution")
    categorie = models.ForeignKey(Categorie)

    def __unicode__(self):
        return self.titre
```

Pour faire un formulaire à partir de ce modèle, c'est très simple :

Code : Python

```
from django import forms
from models import Article

class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
```

Et c'est tout ! Notons que nous héritons maintenant de forms.ModelForm et non plus de forms.Form. Il y a également une sous-classe Meta (comme pour les modèles), qui permet de spécifier des informations supplémentaires. Dans l'exemple, nous avons juste indiqué sur quelle classe le ModelForm devait se baser (à savoir le modèle Article, bien entendu).

Le rendu HTML du formulaire est plutôt éloquent :

Titre:

Auteur:

Contenu:

Categorie:

Tout sur les crêpes ▼

Tout sur les crêpes

L'histoire des crêpes en Bretagne

En plus de convertir les champs modèles vers des champs formulaires adéquats, Django va même chercher toutes les catégories enregistrées dans la base de données et les propose comme choix pour la ForeignKey !

Le framework va aussi utiliser certains paramètres des champs du modèle pour les champs du formulaire. Par exemple, l'argument `verbose_name` du modèle sera utilisé comme l'argument `label` des formulaires, `help_text` reste `help_text` et `blank` devient `required` (`blank` est un argument des champs des modèles qui permet d'indiquer à l'administration et aux `ModelForm` si un champ peut être laissé vide ou non, il est par défaut à `False`).

Une fonctionnalité très pratique des `ModelForm` réside dans le fait qu'il n'y a pas besoin d'extraire les données une à une pour créer ou mettre à jour un modèle. En effet, il fournit directement une méthode `save` qui va mettre à jour la base de données tout seul. Petit exemple dans le shell :

Code : Python

```
>>> from blog.models import Article, Categorie
>>> from blog.forms import ArticleForm
>>> donnees = {
...     'titre': "Les crêpes c'est trop bon",
...     'auteur': "Maxime",
...     'contenu': "Vous saviez que les crêpes bretonnes c'est trop bon ? La pêche c'est nul à côté.",
...     'categorie': Categorie.objects.all()[0].id #On prend l'identifiant de la première catégorie qui vient
... }
>>> form = ArticleForm(donnees)
>>> Article.objects.all()
[]
>>> form.save()
<Article: Les crêpes c'est trop bon>
>>> Article.objects.all()
[<Article: Les crêpes c'est trop bon>]
```

Note : tout objet d'un modèle sauvegardé possède un attribut `id`, c'est un identifiant propre à chaque entrée. Avec les `ForeignKey`, c'est lui qu'on utilise généralement comme clé étrangère.

Pratique n'est-ce pas ? Nous avons ici simulé avec un dictionnaire le contenu d'un éventuel `request.POST` et l'avons passé au constructeur d'`ArticleForm`. Depuis la méthode `save`, le `ModelForm` va directement créer une entrée dans la base de données et retourne l'objet créé.

De la même façon, il est possible de mettre à jour une entrée très simplement. En donnant un objet du modèle sur lequel le `ModelForm` est basé, il peut directement remplir les champs du formulaire et mettre l'entrée à jour selon les modifications de l'utilisateur.

Pour ce faire, dans une vue, il suffit d'appeler le formulaire ainsi :

Code : Python

```
form = ArticleForm(instance=article) #article est bien entendu un objet d'Article quelconque dans la BDD
```

Et Django se charge du reste :

Titre:

Auteur:

Contenu:

Vous saviez que les crêpes bretonnes c'est trop bon ? La pêche c'est nul à côté.

Categorie:

Une fois les modifications du formulaire envoyées depuis une requête POST, il suffit de reconstruire un `ArticleForm` à partir de l'article et de la requête et d'enregistrer les changements si le formulaire est valide :

Code : Python

```
form = ArticleForm(request.POST, instance=article)
if form.is_valid():
    form.save()
```

L'entrée est désormais à jour.

Si vous souhaitez que certains champs ne soient pas éditables par vos utilisateurs, il est possible de sélectionner ou exclure certains, toujours grâce à la sous-classe `Meta` :

Code : Python

```
class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ('auteur', 'categorie',) #exclure les champs nommés "auteur" et "categorie"
```

En ayant exclus ces deux champs, cela revient à sélectionner uniquement les champs titre et contenu, comme ceci :

Code : Python

```
class ArticleForm(forms.ModelForm):  
    class Meta:  
        model = Article  
        fields = ('titre', 'contenu',)
```

Petite précision : l'attribut fields permet également de déterminer l'ordre des champs. Le premier du tuple arriverait en première position dans le formulaire, le deuxième en deuxième position, etc.

Résultat :

Titre:

Contenu:

Cependant, lors de la création d'une nouvelle entrée, si certains champs obligatoires du modèle (ceux qui n'ont pas `null=True` comme argument) ont été exclus, il ne faut pas oublier de les rajouter par la suite. Il ne faut donc pas appeler la méthode `save` telle quelle sur un `ModelForm` avec des champs exclus, sinon Django lèvera une exception. Un paramètre spécial de la méthode `save` a été prévu pour cette situation :

Code : Python

```
>>> from blog.models import Article, Categorie  
>>> from blog.forms import ArticleForm  
>>> donnees = {  
...     'titre': "Un super titre d'article !",  
...     'contenu': "Un super contenu ! (ou pas)"  
... }  
>>> form = ArticleForm(donnees) #Pas besoin de spécifier les autres  
champs, ils ont été exclus  
>>> article = form.save(commit=False) #Ne sauvegarde pas directement  
l'article dans la base de données  
>>> article.categorie = Categorie.objects.all()[0] #On rajoute les  
attributs manquants  
>>> article.auteur = "Mathieu"  
>>> article.save()
```

La chose importante dont il faut se souvenir ici est donc `form.save(commit=False)` qui permet de ne pas sauvegarder directement l'article dans la BDD, mais renvoie un objet avec les données du formulaire sur lequel on peut continuer à travailler dessus.

La gestion des fichiers

Autre point essentiel du web actuel : il est souvent utile d'envoyer des fichiers sur un site web afin que celui-ci puisse les réutiliser par la suite (avatar d'un membre, album photos, chanson, ...). Nous couvrirons dans cette partie la gestion des fichiers coté serveur et les méthodes que Django propose pour y arriver.

Enregistrer une image



Préambule : avant de commencer à jouer avec des images, il est nécessaire d'installer la Python Imaging Library (PIL), Django se sert en effet de cette dernière pour faire ses traitements sur les images. Vous pouvez télécharger la bibliothèque à [cette adresse](#).

Pour introduire la gestion des images, prenons un exemple simple : considérons un répertoire de contacts dans lequel les contacts ont 3 caractéristiques : leur nom, leur adresse et une photo. Pour ce faire, créons un nouveau modèle (placez-le dans l'application de votre choix, nous réutiliserons personnellement ici l'application « blog ») :

Code : Python

```
class Contact(models.Model):
    nom = models.CharField(max_length=255)
    adresse = models.TextField()
    photo = models.ImageField(upload_to="photos/")

    def __unicode__(self):
        return self.nom
```

La nouveauté ici est bien entendu ImageField. Il s'agit d'un champ Django comme les autres, si ce n'est qu'il contiendra une image (au lieu d'une chaîne de caractère, date, nombre, ...).

ImageField prend un argument obligatoire : `upload_to`. Ce paramètre permet de désigner l'endroit où seront enregistrées sur le disque dur les images assignées à l'attribut photo pour toutes les instances du modèle. Nous n'avons pas indiqué d'adresse absolue ici car en réalité, le répertoire indiqué depuis le paramètre sera en fait ajouté au chemin absolu fourni par la variable `MEDIA_ROOT` dans votre `settings.py`. Il est impératif de configurer correctement cette variable avant de commencer à jouer avec des fichiers.

Afin d'avoir une vue permettant de créer un nouveau contact, il faudra pour ce faire créer un formulaire adapté. Créons un formulaire similaire au modèle (un `ModelForm` est tout à fait possible aussi), tout ce qu'il y a de plus simple :

Code : Python

```
class NouveauContactForm(forms.Form):
    nom = forms.CharField()
    adresse = forms.CharField(widget=forms.Textarea)
    photo = forms.ImageField()
```

Le champ `ImageField` vérifie que le fichier envoyé est bien une image valide, sans quoi le formulaire sera considéré comme invalide. Et le tour est joué !

Revenons-en donc à la vue. Elle est également similaire à un traitement de formulaire classique, si ce n'est un petit détail :

Code : Python

```
def nouveau_contact(request):
    sauvegarde = False

    if request.method == "POST":
        form = NouveauContactForm(request.POST, request.FILES)
        if form.is_valid():
            contact = Contact()
```

```

        contact.nom = form.cleaned_data["nom"]
        contact.adresse = form.cleaned_data["adresse"]
        contact.photo = form.cleaned_data["photo"]
        contact.save()

        sauvegarde = True
    else:
        form = NouveauContactForm()

    return render(request, 'contact.html', locals())

```

Faites bien attention à la ligne surlignée : un deuxième argument a été rajoutée, il s'agit de `request.FILES`. En effet, `request.POST` ne contient que des données textuelles, tous les fichiers sélectionnés sont envoyés depuis une autre méthode, et sont finalement recueillis par Django dans le dictionnaire `request.FILES`. Si vous ne passez pas cette variable au constructeur, celui-ci considérera que le champ `photo` est vide et n'a donc pas été complété par l'utilisateur, le formulaire sera donc invalide.

Le champ `ImageField` renvoie une variable du type `UploadedFile`, qui est une classe définie par Django. Cette dernière hérite de la classe `django.core.files.File`. Sachez que si vous souhaitez créer une entrée en utilisant une photo sur votre disque dur (autrement dit, vous ne disposez pas d'une variable `UploadedFile` renvoyée par le formulaire), vous devez créer un objet `File` (prenant un fichier ouvert classiquement) et le passer à votre modèle. Exemple depuis la console :

Code : Python

```

>>> from blog.models import Contact
>>> from django.core.files import File
>>> c = Contact(nom="Jean Dupont", adresse="Rue Neuve 34, Paris")
>>> photo = File(open('/chemin/vers/photo/dupont.jpg', 'r'))
>>> c.photo = photo
>>> c.save()

```

Pour terminer, le template est également habituel, toujours à une exception près :

Code : Jinja

```

<h1>Ajouter un nouveau contact</h1>

{% if sauvegarde %}
<p>Ce contact a bien été enregistré.</p>
{% endif %}

<p>
<form method="post" enctype="multipart/form-data" action=".">
{% csrf_token %}
{{ form.as_p }}
<input type="submit"/>
</form>
</p>

```

Faites bien attention au nouvel attribut de la balise `form` : `enctype="multipart/form-data"`. En effet, sans ce dernier, le navigateur n'envoiera pas les fichiers au serveur web. Oublier cet attribut et le dictionnaire `request.FILES` décrit précédemment sont des erreurs courantes qui peuvent vous faire perdre bêtement beaucoup de temps, ayez le réflexe d'y penser !

Sachez que Django n'acceptera pas n'importe quel fichier. En effet, il s'assurera bien que le fichier envoyé est bien une image, sans quoi il retournera une erreur.

Vous pouvez essayer le formulaire : vous constaterez qu'un nouveau fichier a été créé dans le dossier renseigné dans la variable `MEDIA_ROOT`. Le nom du fichier créé sera en fait le même que celui sur votre disque dur (autrement dit, si vous avez envoyé un fichier nommé "mon_papa.jpg", le fichier côté serveur gardera le même nom). Il est possible de modifier ce comportement, nous y reviendrons plus tard.

Afficher une image

Maintenant que nous possédons une image enregistrée coté serveur, il ne reste plus qu'à l'afficher chez le client. Cependant, un petit problème se pose : par défaut, Django ne s'occupe pas du service de fichiers médias (images, musiques, vidéos, ...), et généralement, il est conseillé de laisser un autre serveur s'en occuper (voir l'annexe sur le déploiement du projet en production). Néanmoins, pour la phase de développement, il est tout de même possible de laisser le serveur de développement s'en charger. Pour ce faire, il vous faut compléter la variable `MEDIA_URL` dans `settings.py` et rajouter cette directive dans votre `urls.py` global :

Code : Python

```
from django.conf.urls.static import static
from django.conf import settings

urlpatterns += static(settings.MEDIA_URL,
                      document_root=settings.MEDIA_ROOT)
```

Ceci étant fait, tous les fichiers consignés dans le répertoire configuré depuis `MEDIA_ROOT` (dans lequel Django déplace les fichiers enregistrés), seront accessibles depuis l'adresse telle qu'indiquée depuis `MEDIA_URL` (un exemple de `MEDIA_URL` serait simplement `"/media/"` ou `"media.monsite.fr/"` en production).

Ceci étant fait, l'affichage d'une image est trivial. Si nous reprenons la liste des contacts enregistrés dans une vue simple :

Code : Python

```
def voir_contacts(request):
    contacts = Contact.objects.all()
    return render(request,
                  'voir_contacts.html', {'contacts': contacts})
```

Coté template :

Code : Jinja

```
<h1>Liste des contacts</h1>

{% for contact in contacts %}
<h2>{{ contact.nom }}</h2>
Adresse : {{ contact.adresse|linebreaks }}<br/>

{% endfor %}
```

Avant de s'attarder aux spécificités de l'affichage de l'image, une petite explication concernant le tag `linebreaks`. Par défaut, Django ne convertit pas les retours à la ligne d'une chaîne de caractères (comme l'adresse ici) en un `
` automatiquement, et ceci pour des raisons de sécurité. Pour autoriser l'ajout de retours à la ligne en HTML, il faut utiliser ce tag, comme dans le code ci-dessus, sans quoi toute la chaîne sera sur la même ligne.

Revenons donc à l'adresse de l'image. Vous aurez déjà plus que probablement reconnu la variable `MEDIA_URL` de `settings.py` qui fait son retour. Elle est accessible depuis le template grâce à un processeur de contexte inclus par défaut.

`contact.photo` renvoie simplement l'adresse relative vers le dossier et nom du fichier associé. Afin de construire une adresse complète, il est impératif d'associer ces deux parties d'adresses, simplement en les juxtaposant.

Si `MEDIA_URL` vaut `"media.monsite.fr/"` et `contact.photo` vaut `"photos/mon_papa.jpg"`, l'adresse absolue concaténée sera donc `"media.monsite.fr/photos/mon_papa.jpg"`.

Le résultat est plutôt simple :

Liste des contacts

Chuck Norris

Adresse : Chuck Norris n'a pas d'adresse, le monde est sa maison.



Il est important de ne jamais renseigner en dur le lien vers l'endroit où est situé le dossier contenant les fichiers. Passer par `MEDIA_URL` est une méthode bien plus propre.

Avant de généraliser pour tous les types de fichiers, sachez qu'un `ImageField` non nul possède deux attributs supplémentaires : `width` et `height`. Ces deux attributs renseignent respectivement la largeur et la hauteur en pixels de l'image.

Encore plus loin

Heureusement, la gestion des fichiers ne s'arrête pas aux images. N'importe quel type de fichier peut être enregistré. La différence avec les images est plutôt maigre.

Au lieu d'utiliser `ImageField` dans les formulaires et modèles, il suffit tout simplement d'utiliser `FileField`. Que ce soit dans les formulaires ou les modèles, le champ s'assurera bien que ce qui lui est passé est bien un fichier, mais cela ne devra plus être nécessairement une image valide.

`FileField` retournera toujours un objet de `django.core.files.File`. Cette classe possède notamment les attributs suivants (l'exemple ici est réalisé avec un `ImageField`, mais les attributs sont également valides avec un `FileField` bien évidemment) :

Code : Python

```
>>> from blog.models import Contact
>>> c = Contact.objects.get(nom="Chuck Norris")
>>> c.photo.name
u'photos/chuck_norris.jpg' #Chemin relatif vers le fichier à partir
de MEDIA_ROOT
>>> c.photo.path
u'/home/mathx/crepes-bretonnes/media/photos/chuck_norris.jpg'
#Chemin absolu
>>> c.photo.url
'http://media.crepes-bretonnes.fr/photos/chuck_norris.jpg' #URL
telle que construite à partir de MEDIA_URL
>>> c.photo.size
45300 #Taille du fichier en bytes
```

De plus, un objet `File` possède également des attributs `read` et `write`, comme un fichier (ouvert à partir d'`open()`) classique.

Dernière petite précision concernant le nom des fichiers côté serveur. Nous avons mentionné plus haut qu'il est en effet possible de les renommer à sa guise, et de ne pas devoir garder le nom que l'utilisateur avait sur son disque dur.

La méthode est plutôt simple : au lieu de passer une chaîne de caractères à comme paramètre `upload_to` dans le modèle, il faut lui passer une fonction qui retournera le nouveau nom du fichier. Cette fonction prend 2 arguments : l'instance du modèle où le `FileField` est défini, et le nom d'origine du fichier.

Un exemple de fonction serait donc simplement :

Code : Python

```
def renommage(instance, nom):  
    return instance.id+'.'+nom.split('.')[ -1] #On se base sur l'ID  
de l'entrée et on garde l'extension du fichier (en supposant ici  
que le fichier possède bien une extension)
```

Un exemple de modèle utilisant cette fonction serait donc simplement :

Code : Python

```
class Document(models.Model):  
    nom = models.CharField(max_length=100)  
    doc = models.FileField(upload_to=renommage,  
        verbose_name="Document")
```

Désormais, vous devriez être en mesure de gérer correctement toute application nécessitant des fichiers !

TP : un raccourcisseur d'URLs

Dans ce chapitre, nous allons mettre en pratique tout ce que vous avez appris jusqu'ici. Il s'agit d'un excellent exercice qui permet d'apprendre à lier les différents éléments du framework que nous avons étudié (URLs, modèles, vues, formulaires, administration et templates).

Cahier de charges

Pour ce travail pratique, nous allons réaliser un raccourcisseur d'URLs. Ce type de service est notamment utilisé sur les sites de micro-blogging (comme Twitter) ou les messageries instantanées, où utiliser une très longue URL est difficile car le nombre de caractères est limité.

Typiquement, si vous avez une longue URL, un raccourcisseur créera une autre URL, beaucoup plus courte, que vous pourrez distribuer. Lorsque quelqu'un clique sur le lien raccourci, il sera directement redirigé vers l'URL plus longue. Par exemple, *tib.ly/abcde* redirigerait vers *www.mon-super-site.com/qui-a-une/URL-super-longue*. Le raccourcisseur va générer un code (ici *abcde*) qui sera propre à l'URL plus longue. Un autre code redirigera le visiteur vers une autre URL.

Vous allez devoir créer une nouvelle application qu'on nommera *mini_url*. Cette application ne contiendra qu'un modèle appelé MiniURL, c'est lui qui enregistrera les raccourcis. Il comportera les champs suivants :

- La plus longue URL : URLField
- Le code qui permet d'identifier le raccourci
- La date de création du raccourci
- Le pseudo du créateur du raccourci (optionnel)
- Le nombre d'accès au raccourci (une redirection = un accès)

Nous avons indiqué le type du champ pour l'URL, ne l'ayant jamais vu dans le cours auparavant. Les autres sont classiques et vus, nous supposons donc que vous choisirez le bon type.

Les deux premiers champs (URL et code) devront avoir le paramètre `unique=True`. Ce paramètre garantit que deux entrées ne partageront jamais le même code ou la même URL, ce qui est primordial ici.

Finalement, le nombre d'accès sera par défaut mis à 0 grâce au paramètre `default=0`.

Vous devrez également créer un formulaire, plus spécialement un ModelForm basé sur le modèle MiniURL. Celui-ci ne contiendra que les champs URL et pseudo, le reste sera soit initialisé selon les valeurs par défaut, ou alors généré par la suite (le code notamment).

Nous vous fournissons la fonction qui permet de générer le code :

Code : Python

```
def generer(N):
    caracteres = string.letters + string.digits
    aleatoire = [random.choice(caracteres) for _ in xrange(N)]

    return ''.join(aleatoire)
```



En théorie, il faudrait vérifier que le code n'est pas déjà utilisé ou alors faire une méthode nous assurant aucun doublon. Dans un souci de simplicité et pédagogique, nous allons sauter cette étape. Attention cependant à **bien vérifier que vos données sont bien uniques** lorsque vous souhaitez générer des identifiants !

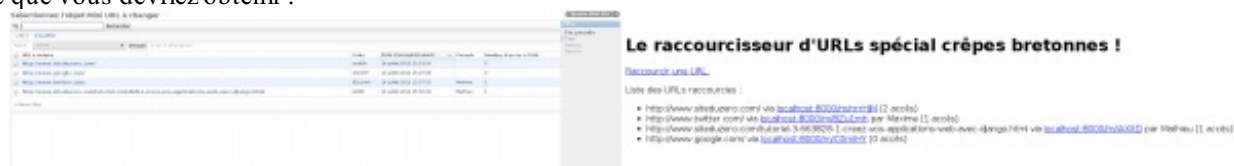
Vous aurez ensuite 3 vues :

- Une vue affichant toutes les redirections créées et leurs informations, trié par ordre descendant, de la redirection avec le plus d'accès au moins d'accès
- Une vue avec le formulaire pour créer une redirection
- Une vue qui prend comme paramètre dans l'URL le code et redirige l'utilisateur vers la plus longue URL

Partant de ces trois fonctions, il ne faudra que 2 templates (la redirection n'en ayant pas besoin), et 3 routages d'URLs bien entendu.

L'administration devra être activée, et le modèle accessible depuis celle-ci. Il devra être possible de rechercher des redirections depuis la longue URL via une barre de recherche, tous les champs devront être affichés dans une catégorie et le tri par défaut sera fait selon la date de création du raccourci.

Voici ce que vous devriez obtenir :



Le raccourcisseur d'URLs spécial crêpes bretonnes !

[Raccourcir une URL](#)

Liste des URL's raccourcies :

- <http://www.siteduzero.com/> via [siteduzero](#) [2 accès]
- <http://www.better.com/> via [better](#) [1 accès]
- <http://www.siteduzero.com/forums/3-063029-1-crepez-vois-application-kras-vois-jeanpierre.html> via [siteduzero](#) [1 accès]
- <http://www.google.com/> via [google](#) [12 accès]

Raccourcir une URL

URL à réduire:

Pseudo:

Si vous coincez sur quelque chose, n'hésitez pas à aller relire les explications dans le chapitre concerné, tout y a été expliqué.

Correction

Normalement, ça ne devait pas avoir posé de problème !

Il fallait donc créer une nouvelle application et l'inclure dans votre settings.py :

Code : Console

```
python manage.py startapp mini_url
```

Votre models.py devrait ressembler à ceci :

Code : Python - models.py

```
#!/usr/bin/env python
#-*- coding: utf-8 -*-
from django.db import models
import random
import string

class MiniURL(models.Model):
    url = models.URLField(verbose_name="URL à réduire", unique=True)
    code = models.CharField(max_length=6, unique=True)
    date = models.DateTimeField(auto_now_add=True,
        verbose_name="Date d'enregistrement")
    pseudo = models.CharField(max_length=255, blank=True, null=True)
    nb_acces = models.IntegerField(default=0, verbose_name="Nombre
        d'accès à l'URL")

    def __unicode__(self):
        return "[{0}] {1}".format(self.code, self.url)

    def save(self, *args, **kwargs):
        if self.pk is None:
            self.generer(6)

        super(MiniURL, self).save(*args, **kwargs)

    def generer(self, N):
        caracteres = string.letters + string.digits
        aleatoire = [random.choice(caracteres) for _ in xrange(N)]
```

```

        self.code = ''.join(aleatoire)

class Meta:
    verbose_name = "Mini URL"
    verbose_name_plural = "Minis URLs"

```

Il y a plusieurs commentaires à faire dessus. Tout d'abord, nous avons surchargé la méthode `save()`, afin de générer automatiquement le code de notre URL. J'ai pris le soin d'intégrer la méthode `generer()` au sein du modèle, mais il est aussi possible de la déclarer à l'extérieur et de faire `self.code = generer(6)`. Il ne faut surtout pas oublier la ligne qui appelle le `save()` parent, sinon lorsque vous validerez votre formulaire il se passera tout simplement rien !

La classe `Meta` ici est similaire à la classe `Meta` d'un `ModelForm`, elle permet d'indiquer des métadonnées concernant le modèle. Ici on a modifié le nom qui sera utilisé dans les `ModelForm` et l'administration (`verbose_name`) et sa forme plurielle (`verbose_name_plural`).

Après la création de nouveaux modèles, il fallait les rajouter dans la base de données via la commande

```
python manage.py syncdb :
```

Code : Console

```

$ python manage.py syncdb
Creating tables ...
Creating table mini_url_miniurl
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)

```

Le `forms.py` est tout à fait classique :

Code : Python - mini_url/forms.py

```

#-*- coding: utf-8 -*-
from django import forms
from models import MiniURL

class MiniURLForm(forms.ModelForm):
    class Meta:
        model = MiniURL
        fields = ('url', 'pseudo')

```

de même pour `admin.py` :

Code : Python - mini_url/admin.py

```

#-*- coding: utf-8 -*-
from django.contrib import admin
from models import MiniURL

class MiniURLAdmin(admin.ModelAdmin):
    list_display = ('url', 'code', 'date', 'pseudo', 'nb_acces')
    list_filter = ('pseudo',)
    date_hierarchy = 'date'
    ordering = ('date',)
    search_fields = ('url',)

admin.site.register(MiniURL, MiniURLAdmin)

```

Voici `mini_url/urls.py`. N'oubliez pas de l'importer dans votre `urls.py` principal. Rien de spécial non plus :

Code : Python - `mini_url/urls.py`

```
# -*- coding: utf-8 -*-
from django.conf.urls import patterns, url

urlpatterns = patterns('mini_url.views',
    url(r'^$', 'liste', name='url_liste'), # Une string vide
    indique la racine
    url(r'^nouveau/$', 'nouveau', name='url_nouveau'),
    url(r'^(?P<code>\w{6})/$', 'redirection',
    name='url_redirection'), # (?P<code>\w{6}) capturera 6 caractères
    alphanumériques.
)
```



Nous avons nommé les URLs ici pour des raisons pratiques, que l'on verra plus tard dans ce cours.

Et enfin pour finir, le fichier `views.py` :

Code : Python - `mini_url/views.py`

```
# -*- coding: utf-8 -*-
from django.shortcuts import redirect, get_object_or_404, render
from models import MiniURL
from forms import MiniURLForm

def liste(request):
    """Affichage des redirections"""
    minis = MiniURL.objects.order_by('-nb_acces')

    return render(request, 'mini_url/liste.html', locals())

def nouveau(request):
    """Ajout d'une redirection"""
    if request.method == "POST":
        form = MiniURLForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect(liste)
    else:
        form = MiniURLForm()

    return render(request, 'mini_url/nouveau.html', {'form': form})

def redirection(request, code):
    """Redirection vers l'URL enregistrée"""
    mini = get_object_or_404(MiniURL, code=code)
    mini.nb_acces += 1
    mini.save()

    return redirect(mini.url, permanent=True)
```

Notez qu'à cause à l'argument `permanent=True`, le serveur renvoie le code HTTP 301 (redirection permanente).



Certains navigateurs mettent en cache une redirection permanente. Ainsi, la prochaine fois que le visiteur clique sur votre lien, le navigateur se souviendra de la redirection et vous redirigera sans même appeler votre page. Le nombre



d'accès ne sera alors pas incrémenté.

Pour terminer, les deux templates, liste.html et nouveau.html. Remarquez `{{ request.get_host }}` qui donne le nom de domaine et le port utilisé. En production, par défaut il s'agit de localhost:8000. Néanmoins, si nous avons un autre domaine comme bit.ly, c'est ce domaine qui serait utilisé (il serait d'ailleurs beaucoup plus court et pratique comme raccourcisseur d'URLs).

Code : HTML - liste.html

```
<h1>Le raccourcisseur d'URLs spécial cr&ecirc;pes bretonnes !</h1>

<p><a href="{% url url_nouveau %}">Raccourcir une URL.</a></p>

<p>Liste des URLs raccourcies :</p>
<ul>
    {% for mini in minis %}
    <li> {{ mini.url }} via <a href="http://{{ request.get_host }}{%
url url_redirection mini.code %}">{{ request.get_host }}{% url
url_redirection mini.code %}</a>
    {% if mini.pseudo %}par {{ mini.pseudo }}{% endif %} ({{
mini.nb_acces }} acc&egrave;s)</li>
    {% empty %}
    <li>Il n'y en a pas actuellement.</li>
    {% endfor %}
</ul>
```

Code : HTML - nouveau.html

```
<h1>Raccourcir une URL</h1>

<form method="post" action="{% url url_nouveau %}">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit"/>
</form>
```

À part la sous-classe Meta du modèle et `request.get_host`, tout le reste a été couvert dans les chapitres précédents. Si quelque chose vous semble étrange, n'hésitez pas à aller relire le chapitre concerné (encore une fois).

Télécharger l'application

Archive zip contenant l'application et les templates.



miniurl.zip - 3,5ko

Ce TP conclut la partie 2 du tutoriel. Nous avons couvert les bases du framework. Vous devez normalement être capable de réaliser des applications basiques. Dans les prochains parties, nous allons approfondir ces bases et étudier les autres bibliothèques que Django propose.

En attendant, voici quelques idées d'améliorations pour ce TP :

- Intégrer un style CSS et des images depuis des fichiers statiques via le tag `{% block %}` ;
- Donner davantage de statistiques sur les redirections ;
- Proposer la possibilité de rendre une redirection anonyme ;
- etc.

Partie 3 : Techniques avancées

Nous avons vu de manière globale chaque composante du framework dans la partie précédente. Vous êtes désormais capable de réaliser de petites applications fonctionnelles, mais qui ne seront pas forcément le plus optimisé possible. Pour résoudre ce point, nous allons voir dans cette nouvelle partie des méthodes avancées, permettant de réduire nos efforts lors de la conception. En effet, Django nous fournit plusieurs outils permettant d'accélérer le développement de nos applications, en écrivant moins de lignes de code !

Les vues génériques

Dans la première partie, nous avons vu le fonctionnement classique des vues qui, associées à une URL (ou un schéma d'URL), se contentaient de récupérer et traiter des informations afin de retourner du HTML, via les templates.

Cependant sur la plupart des sites web, il existe certains types de pages où créer une vue est lourd et presque inutile : par exemple, pour une page statique (comme une page "Mentions légales" ou "F.A.Q."), la vue a de fortes chances de ressembler à ceci :

Code : Python - Exemple de vue mal conçue

```
def faq(request):  
    return render(request, 'pages/faq.html', {})
```

Créer une vue juste pour afficher un template risque d'être lourd et redondant si vous avez plusieurs pages statiques à afficher !

De plus, parfois vos pages se contenteront uniquement de lister des objets (d'un modèle le plus souvent), sans filtre ou traitement particulier, ce qui donnera des vues très similaires à celle présentée ci-dessus, et d'autant plus redondantes. Or, étant donné que Django est conçu pour n'avoir à écrire que **le minimum** (philosophie [DRY](#)), le framework inclut un système de vues génériques, qui évite au développeur de devoir écrire des fonctions simples et identiques, nous permettant de sauver du temps et des lignes de code.

Ces vues génériques sont en quelque sorte des vues très modulaires prêtes à être utilisées directement et incluses par défaut dans Django. Pour reprendre l'affichage d'une page statique, il suffit d'indiquer dans le routage d'URLs la vue générique chargée de renvoyer une page statique en lui indiquant sur quel template se baser, et le tour est joué !



Ce chapitre est assez long et dense en information. Nous vous conseillons de lire en plusieurs fois : nous allons faire plusieurs types distincts de vues, qui ont chacune une utilité différente. Il est tout à fait possible de poursuivre ce cours sans connaître tous ces types.

Premiers pas avec des pages statiques

Pour illustrer le fonctionnement global des vues génériques, reprenons l'exemple de l'introduction, à savoir celui des pages statiques, qui en soi, ne se charge que du rendu d'un template.

Première caractéristique des vues génériques : c'est ne sont pas des fonctions, comme dans les vues classiques, mais des classes. L'amalgame **1 vue = 1 fonction** en est du coup quelque peu désuet. Ces classes doivent également être renseignées dans vos *views.py*.

Il y a **deux méthodes principales d'utilisation** pour les vues génériques :

1. Soit on crée une classe, héritant d'un *type* de vue générique dans laquelle on surchargera les attributs.
2. Soit on appelle directement la classe générique, en passant en argument les attributs que l'on souhaite surcharger !

Toutes les classes de vues génériques sont situées dans `django.views.generic`. Concentrons-nous pour le moment sur `TemplateView`, qui est un type de vue générique. Typiquement, `TemplateView` permet, comme son nom l'indique, de créer une vue qui s'occupera du rendu d'un template.

Comme dit précédemment, créons une classe héritant de `TemplateView`, et surchargeons ses attributs :

Code : Python - blog/views.py

```
from django.views.generic import TemplateView
```



```
class FAQView(TemplateView):
    template_name = "blog/faq.html" # chemin vers le template à
    afficher
```

Dès lors, il suffit de router notre URL vers une méthode héritée de la classe `TemplateView`, ici `as_view` :

Code : Python - `blog/urls.py`

```
from django.conf.urls import patterns, url, include
from blog.views import FAQView # N'oubliez pas d'importer la classe
mère

urlpatterns = patterns('',
    (r'^faq/$', FAQView.as_view()), # On demande la vue
    correspondant à la classe FAQView que l'on a créée
)
```

C'est déjà tout ! Lorsqu'on accède à `/blog/faq/`, le contenu du fichier `templates/blog/faq.html` sera affiché.



Que se passe-t-il concrètement ?

La méthode `as_view` de `FAQView` va retourner une vue (en réalité, il s'agit d'une fonction classique) qui se basera sur ses attributs pour en déterminer son fonctionnement. Étant donné que nous avons indiqué un template à utiliser depuis l'attribut `template_name`, la classe l'utilisera pour générer une vue adaptée.

Nous avons indiqué précédemment qu'il y avait deux méthodes pour utiliser les vues génériques. Le principe de la seconde est de directement instancier `TemplateView` dans le fichier `urls.py`, en lui passant en argument notre `template_name` :

Code : Python - `blog/urls.py`

```
from django.conf.urls import patterns, url, include
from django.views.generic import TemplateView # L'import a changé,
attention !

urlpatterns = patterns('',
    url(r'^faq/',
    TemplateView.as_view(template_name='blog/faq.html')),
)
```



Et dans notre `views.py` ?

Vous pouvez alors retirer `FAQView`, la classe ne sert plus à rien. Pour les `TemplateView`, la première méthode présente peu d'intérêt, cependant nous verrons par la suite qu'hériter d'une classe sera plus facile que tout définir dans `urls.py`.

Lister et afficher des données

Jusqu'ici, nous avons vu comment **afficher des pages statiques** avec des vues génériques. Bien que pratique, il n'y a jusqu'ici rien de très puissant.

Abordons désormais quelque chose de plus intéressant. Un schéma utilisé presque partout sur le web est le suivant : vous avez une liste d'objets (des articles, des images, ...), et lorsque vous cliquez sur un élément, vous êtes redirigé vers une page présentant plus en détail ce même élément.

Nous avons déjà réalisé quelque chose de semblable dans le [chapitre sur les modèles](#) avec notre liste d'articles et l'affichage individuel d'articles.

Nous allons repartir de la même idée, mais cette fois-ci avec des vues génériques. Pour ce faire, nous utiliserons deux nouvelles classes : `ListView` et `DetailView`. Nous réutiliserons les deux modèles `Article` et `Categorie`, qui ne changeront pas.

Une liste d'objets en quelques lignes avec ListView

Commençons par une simple liste de nos articles, sans pagination. À l'instar de `TemplateView`, on peut utiliser `ListView` directement en lui passant en paramètre le modèle à traiter :

Code : Python - `blog/urls.py`

```
from django.conf.urls import patterns, url, include
from django.views.generic import ListView
from blog.models import Article

urlpatterns = patterns('',
    # On va réécrire l'URL de l'accueil
    url(r'^$', ListView.as_view(model=Article,)),

    # Et on a toujours nos autres pages...
    url(r'^article/(?P<id>\d+)$', 'blog.views.lire'),
    url(r'^(?P<page>\d+)$', 'blog.views.archives'),
    url(r'^categorie/(?P<slug>.+)$', 'blog.views.voir_categorie'),
)
```

Avec cette méthode, Django impose quelques conventions :

- Le template devra s'appeler `<app>/<model>_list.html`. Dans notre cas, le template serait nommé `blog/article_list.html`
- L'unique variable retournée par la vue générique et utilisable dans le template est appelée `object_list`, qui contiendra ici tous nos articles.

Il est possible de redéfinir ces valeurs en passant des arguments supplémentaires à notre `ListView` :

Code : Python - `blog/urls.py`

```
urlpatterns = patterns('',
    url(r'^$', ListView.as_view(model=Article,
                                context_object_name="derniers_articles",
                                template_name="blog/accueil.html")),
    ...
)
```

Par souci d'économie, nous souhaitons réutiliser le template `blog/accueil.html` qui utilisait comme nom de variable `derniers_articles`, à la place d'`object_list`, celui par défaut de Django.

Vous pouvez dès lors supprimer la fonction `accueil` dans `views.py`, et vous obtiendrez le même résultat qu'avant (ou presque, si vous avez plus de 5 articles) ! L'ordre d'affichage des articles est celui défini dans le modèle, via l'attribut `ordering` de la sous-classe `Meta`, qui se base par défaut sur la clé primaire de chaque entrée.

Il est possible d'aller plus loin : on ne souhaite généralement pas tout afficher sur une même page, on veut filtrer les articles affichés, etc. Il existe donc plusieurs attributs et méthodes de `ListView` qui étendent les possibilités de la vue.

Par souci de lisibilité, nous vous conseillons plutôt de renseigner les classes dans `views.py`, comme vu précédemment. Tout d'abord, changeons notre `urls.py`, pour appeler notre nouvelle classe :

Code : Python - `urls.py`

```
from blog.views import ListeArticles

urlpatterns = patterns('',
    url(r'^$', ListeArticles.as_view(), name="blog_categorie"), #
    # Via la fonction as_view, comme vu tout à l'heure
    ...
)
```



Pourquoi avoir nommé l'URL avec l'argument `name` ?

Pour profiter au maximum des possibilités de Django et donc écrire les URL via la **fonction `reverse`**, et son tag associé dans les templates. L'utilisation du tag `url` se fera dès lors ainsi : `{% url blog_categorie categorie.id %}`. Cette fonctionnalité ne dépend pas des vues génériques, mais est inhérente au fonctionnement des URLs en général. Vous pouvez donc également associer le paramètre `name` à une vue normale.

Ensuite, créons notre classe qui reprendra les mêmes attributs que notre `ListView` de tout à l'heure :

Code : Python - `blog/views.py`

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
```

Désormais, nous souhaitons paginer nos résultats, afin de n'afficher que 5 articles par page, par exemple. Il existe un attribut adapté :

Code : Python - `blog/views.py`

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
    paginate_by = 5
```

De cette façon, la page actuelle est définie via l'argument `page`, passé dans l'URL (`/?page=2` par exemple). Il suffit dès lors d'adapter le template pour faire apparaître la pagination. Sachez que vous pouvez définir le style de votre pagination dans un template séparé, et l'inclure à tous les endroits nécessaires, via `{% include pagination.html %}` par exemple.



Le fonctionnement par défaut de la pagination est celui de la classe `Paginator`, présent dans `django.core.paginator`

Code : Jinja - `blog/accueil.html`

```
<h1>Bienvenue sur le blog des crêpes bretonnes !</h1>

{% for article in derniers_articles %}
<div class="article">
<h3>{{ article.titre }}</h3>
<p>{{ article.contenu|truncatewords_html:80 }}</p>
<p><a href="{% url blog.views.lire article.id %}">Lire la suite</a>
</div>
{% endfor %}

{# Mise en forme de la pagination ici #}
{% if is_paginated %}
<div class="pagination">
{% if page_obj.has_previous %}
<a href="?page={{ page_obj.previous_page_number }}">Précédente</a>
&mdash;
{% endif %}
Page {{ page_obj.number }} sur {{ page_obj.paginator.num_pages }}
{% if page_obj.has_next %}
&mdash; <a href="?page={{ page_obj.next_page_number }}">Suivante</a>
{% endif %}
</div>
{% endif %}
```

```
{% endif %}
</div>
{% endif %}
```

Allons plus loin ! Nous pouvons également **surcharger la sélection des objets** à récupérer, et ainsi soumettre nos propres filtres :

Code : Python - views.py

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
    paginate_by = 5
    queryset = Article.objects.filter(categorie__id=1)
```

Ici, seuls les articles de la 1^{ère} catégorie créée seront affichés. Vous pouvez bien sûr effectuer des requêtes identiques à celle des vues, avec du tri, plusieurs conditions, etc. Il est aussi possible de **passer des arguments** également pour rendre la **sélection un peu plus dynamique en passant l'ID** souhaité dans l'URL.

Code : Python - blog/urls.py

```
from django.conf.urls import patterns, url
from blog.views import ListeArticles

urlpatterns = patterns('blog.views',
    url(r'^categorie/(\w+)$', ListeArticles.as_view()),
    url(r'^article/(?P<id>\d+)$', 'lire'),
    url(r'^(?P<page>\d+)$', 'archives')
)
```

Dans la vue, nous sommes obligés de surcharger `get_queryset`, qui renvoie la liste d'objets à afficher. En effet, il est impossible d'accéder aux paramètres lors de l'assignation d'attributs comme nous le faisons depuis le début.

Code : Python - views.py

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
    paginate_by = 10

    def get_queryset(self):
        return Article.objects.filter(categorie__id=self.args[0])
```

Tâchez tout de même de vous poser des limites, le désavantage ici est le suivant : la lecture de votre `urls.py` devient plus difficile, tout comme votre vue (imaginez si vous avez 4 arguments, à quoi correspond le 2^{ème}, le 4^{ème} ?).

Enfin, il est possible d'ajouter des éléments au contexte, c'est à dire les variables qui sont renvoyées au template. Par exemple, renvoyer l'ensemble des catégories, afin de faire une liste de liens vers celles-ci. Pour ce faire, nous allons **ajouter au tableau context une clé** `categories` qui contiendra notre liste :

Code : Python - blog/views.py

```
def get_context_data(self, **kwargs):
    # On récupère le contexte depuis la super classe
```

```

        context = super(ListeArticles,
self).get_context_data(**kwargs)
        # On ajoute la liste des catégories, sans filtre particulier
        context['categories'] = Categories.objects.all()
        return context

```

Il est facile d'afficher la liste des catégories dans notre template :

Code : Jinja - Extrait de blog/accueil.html

```

<h3>Catégories disponibles</h3>
<ul>
{% for categorie in categories %}
<li><a href="{% url blog_categorie categorie.id %}">{{ categorie.nom
}}</a></li>
{% endfor %}
</ul>

```

Afficher un article via DetailView

Malgré tout cela, nous ne pouvons qu'afficher que des listes, et non pas un objet précis. Heureusement, la plupart des principes vus précédemment avec les classes héritant de `ListView` sont applicables avec celles qui héritent de `DetailView`.

Le but de `DetailView` est de **renvoyer un seul objet** d'un modèle, et non une liste. Pour cela, il va falloir passer un paramètre bien précis dans notre URL : `pk`, qui représentera **la clé primaire de l'objet à récupérer** :

Code : Python - blog/urls.py

```

from blog.views import ListeArticles, LireArticle

urlpatterns = patterns('blog.views',
    url(r'^categorie/(\w+)$', ListeArticles.as_view()),
    url(r'^article/(?P<pk>\d+)$', LireArticle.as_view(),
        name='blog_lire'),
)

```

Maintenant que nous avons notre URL, avec la clé primaire en paramètre, il nous faut écrire la classe qui va récupérer l'objet voulu et le renvoyer à un template précis :

Code : Python - blog/views.py

```

class LireArticle(DetailView):
    context_object_name = "article"
    model = Article
    template_name = "blog/lire.html"

```

... et encore une fois l'ancienne vue devient inutile. Souvenez-vous que notre fonction `lire()` gérait le cas où l'ID de l'article n'existait pas, il en est de même ici. Comme tout à l'heure, vu que nous avons nommé notre objet `article`, il n'y a **aucune modification à faire dans le template** :

Code : Jinja

```

<h1>{{ article.titre }} <span class="small">dans {{
article.categorie.nom }}</span></h1>
<p class="infos">Rédigé par {{ article.auteur }}, le {{
article.date|date:"DATE_FORMAT" }}</p>
<div class="contenu">{{ article.contenu|linebreaks }}</div>

```

Comme pour les `ListView`, il est possible de personnaliser la sélection avec `get_queryset`, afin de ne sélectionner l'article que s'il est public par exemple. Une autre spécificité utile lorsque l'on affiche un objet, c'est d'avoir le **besoin de modifier un de ses attributs**, par exemple son nombre de vues ou sa date de dernier accès. Pour faire cette opération, il est possible de surcharger la méthode `get_object`, qui renvoie l'objet à afficher :

Code : Python - `blog/views.py`

```
class LireArticle(DetailView):
    context_object_name = "article"
    model = Article
    template_name = "blog/lire.html"

    def get_object(self):
        # On récupère l'objet, via la super classe
        article = super(LireArticle, self).get_object()

        article.nb_vues += 1 # Imaginons qu'on ait un attribut "Nombre
de vues"
        article.save()

        return article # Et on retourne l'objet à afficher
```

Enfin, sachez que la variable `request`, qui contient les informations sur la requête et l'utilisateur est également disponible dans les vues génériques. C'est un **attribut de la classe**, que vous pouvez donc appeler dans n'importe quelle méthode via `self.request`.

Agir sur les données

Jusqu'ici, nous n'avons qu'affiché des données, statiques ou en provenance de modèles. Nous allons maintenant nous occuper de la gestion de données. Pour cette partie, nous reprendrons comme exemple notre application de raccourcissement d'URL, que nous avons développée lors du chapitre précédent.

Pour rappel, dans le schéma **CRUD**, il y a 4 types d'actions applicables sur une donnée :

- Create (*Créer*)
- Read (*Lire*, que l'on a déjà traité juste au-dessus)
- Update (*Mettre à jour*)
- Delete (*Supprimer*)

Nous montrerons comment réaliser ces dérivées dans l'ordre indiqué. Chacune possède une vue générique associée !

CreateView

Commençons par la **création d'objets**, souvent utile sur le web de nos jours : un site un tant soit peu communautaire permet à n'importe qui de fournir du contenu : commentaires, forums, etc, ou encore poster un lien pour le *minifier*. Pour simplifier notre formulaire d'ajout de liens, nous allons surcharger la classe `CreateView` :

Code : Python - `blog/views.py`

```
from django.views.generic import CreateView
from django.core.urlresolvers import reverse_lazy

class URLCreate(CreateView):
    model = MiniURL
    template_name = 'mini_url/nouveau.html'
    form_class = MiniURLForm
    success_url = reverse_lazy(liste)
```

Comme tout à l'heure, l'attribut `model` permet de spécifier avec quel modèle on travaille et `template_name` pour spécifier le chemin vers le template (par défaut, le chemin est `<app>/<model>_create_form.html`, avec le nom du modèle tout en minuscule).

La nouveauté ici se trouve sur les 2 attributs suivants. Le premier, `form_class` permet de spécifier quel `ModelForm` utiliser pour **définir les champs disponibles à l'édition**, et tout ce qui est propriété du formulaire. Ici, nous allons réutiliser la classe que nous avons écrite précédemment étant donné qu'elle est suffisante pour l'exemple.

Le dernier argument quant à lui permet de **spécifier vers où rediriger quand le formulaire est validé et enregistré**. Nous avons utilisé ici `reverse_lazy`, qui permet d'utiliser la méthode `reverse()`, même si la configuration des URLs n'a pas encore eu lieu (ce qui est le cas ici, puisque les vues sont analysées avant les `urls.py`)



Comment est-ce que ça fonctionne ?

Le comportement de cette classe est similaire à notre ancienne vue `nouveau()` : s'il n'y a pas eu de requêtes de type `POST`, elle affiche le formulaire, selon les propriétés de `form_class`, et dans le template fourni.

Une fois validé et si, et seulement si, le formulaire est considéré comme correct (`if form.is_valid()` dans notre ancienne vue), alors la méthode `save()` est appelée sur l'objet généré par le formulaire, puis redirige l'utilisateur vers l'URL `success_url`.

Notre template est déjà prêt pour cette vue, puisque l'objet `Form` renvoyé par cette vue générique est nommé `form`, comme on l'avait fait avec l'ancienne méthode.

Avant tout, il faut rapidement éditer `urls.py` :

Code : Python - `urls.py`

```
#-*- coding: utf-8 -*-
from django.conf.urls import patterns, url
from views import URLCreate

urlpatterns = patterns('mini_url.views',
    url(r'^$', 'liste', name='url_liste'), # Une string vide
    indique la racine
    url(r'^nouveau/$', URLCreate.as_view(), name='url_nouveau'),
    url(r'^(?P<code>\w{6})/$', 'redirection',
    name='url_redirection'), # (?P<code>\w{6}) capturera 6 caractères
    alphanumériques.
)
```

Une fois cette ligne modifiée, nous pouvons retenter la génération d'une URL raccourcie. Si vous vous rendez sur `/url/nouveau`, vous remarquerez que le comportement de la page n'a pas changé. En réalité, notre nouvelle vue en fait autant que l'ancienne, en ayant écrit sensiblement moins.

UpdateView

Après la création, attaquons-nous à la **mise à jour des données**. Imaginons que l'on souhaite pouvoir changer l'URL ou le pseudo entré, il nous faut une nouvelle vue, qui va nous permettre de fournir de nouveau ces informations. Cette fois, nous allons hériter de la classe `UpdateView` qui se présente comme `CreateView` :

Code : Python - `blog/views.py`

```
from django.views.generic import CreateView, UpdateView
from django.core.urlresolvers import reverse_lazy

class URLUpdate(UpdateView):
    model = MiniURL
    template_name = 'mini_url/nouveau.html'
    form_class = MiniURLForm
    success_url = reverse_lazy(liste)
```



Les deux classes sont quasi identiques non ?

En effet, nous n'avons même pas pris le soin de changer le nom du template ! En fait, les attributs des classes `CreateView` et `UpdateView` sont les mêmes, et leur fonctionnement est très proche. En effet, entre la création d'un objet, ou sa mise à jour, la page n'a pas réellement besoin d'être modifiée. Tout au plus, en cas de mise à jour, les champs sont autocomplétés avec les données de l'objet.

Par défaut, le nom du template attribué à une vue générique de type `UpdateView` est `<app>/<model>_update_form.html`, afin de pouvoir le différencier de la création.

Pour rendre notre template totalement fonctionnel, il faut juste changer une ligne :

Code : Jinja

```
<form method="post" action="{% url url_nouveau %}">
```

par

Code : Jinja

```
<form method="post" action="">
```

En effet, nous utiliserons cette page pour deux types d'actions, ayant deux URLs distincte. Il suffit de se dire « *quand on valide le formulaire, on soumet la requête à la même adresse que la page actuelle* ».

Il ne reste plus qu'à modifier notre `urls.py`. Comme pour `DetailView`, il faut récupérer la clé primaire, appelée `pk`. Pas de changement profond, voici la ligne :

Code : Python

```
url(r'^edition/(?P<pk>\d)/$', URLUpdate.as_view(),
    name='url_update'), # Pensez à importer URLUpdate en début de
                        # fichier
```

Désormais, vous pouvez accéder à l'édition d'un objet `MiniURL`. Pour y accéder, cela se fait depuis les adresses suivantes : `/url/edition/1` pour le 1^{er} objet, `/url/edition/2` pour le deuxième, etc.



Exemple de formulaire de mise à jour, reprenant le même template que l'ajout

Le résultat est satisfaisant. Bien évidemment, **la vue est très minimaliste** : n'importe qui peut éditer tous les liens, pas de message de confirmation, etc. Par contre, il y a une gestion des objets qui n'existe pas en renvoyant une 404, des formulaires incorrects, etc. Tout ceci est améliorable.

Améliorons nos URLs avec la méthode `get_object()`

Pour le moment, nous utilisons l'identifiant numérique, nommé `pk`, qui est la clé primaire dans l'URL. Ce n'est pas forcément le meilleur choix (pour le référencement par exemple). On pourrait prendre le code présent dans l'URL réduite.



Ce que l'on actuellement et ce que l'on souhaite avoir.

Pas de souci d'unicité, nous savons que chaque entrée possède un code unique. Surchargeons donc la méthode `get_object`, qui s'occupe de récupérer l'objet à mettre à jour.

Code : Python - `blog/views.py`

```
class URLUpdate(UpdateView):
    model = MiniURL
    template_name = 'mini_url/nouveau.html'
    form_class = MiniURLForm
    success_url = reverse_lazy(liste)

    def get_object(self, queryset=None):
        code = self.kwargs.get('code', None)
        return get_object_or_404(MiniURL, code=code)
```

Nous utilisons encore une fois la fonction `get_object_or_404`, qui nous permet de renvoyer une page d'erreur si jamais le code demandé n'existe pas. Le code de l'adresse est accessible depuis le dictionnaire `self.kwargs`, qui contient les arguments nommés dans l'URL (précédemment, les arguments de `ListView` n'étaient pas nommés). Il faut donc changer un peu `urls.py` également, pour accepter l'argument `code`, qui prend des lettres et des chiffres :

Code : Python

```
url(r'^edition/(?P<code>\w{6})/$', URLUpdate.as_view(),
    name='url_update'), # Le code est composé de 6 chiffres/lettres
```

Effectuer une action lorsque le formulaire avec validé avec `form_valid()`

De la même façon, il est possible de changer le comportement lorsque le formulaire est validé, en redéfinissant la méthode `form_valid`. Cette méthode est appelée dès qu'un formulaire est soumis et est considéré comme validé. Par défaut, il s'occupe d'enregistrer les modifications et de rediriger l'utilisateur, mais vous pouvez très bien en changer son comportement :

Code : Python

```
def form_valid(self, form):
    self.object = form.save()
    messages.success(self.request, "Votre profil a été mis à
jour avec succès.") # Envoi d'un message à l'utilisateur
    return HttpResponseRedirect(self.get_success_url())
```

Ici, nous précisons à l'utilisateur, au moyen d'une méthode particulière, que l'édition s'est bien déroulée. Grâce à ce genre de méthodes, vous pouvez affiner le fonctionnement de votre vue, tout en conservant la puissance de la généricité.

DeleteView

Pour terminer, attaquons-nous à la suppression d'un objet. Cette vue prend, comme pour l'`UpdateView`, un objet et demande la confirmation de suppression. Si l'utilisateur confirme, alors la suppression est effectuée, puis on redirige l'utilisateur. Les attributs

de la vue sont donc globalement identiques à ceux utilisés précédemment :

Code : Python - mini_url/views.py

```
class URLDelete(DeleteView):
    model = MiniURL
    context_object_name = "mini_url"
    template_name = 'mini_url/supprimer.html'
    success_url = reverse_lazy(liste)

    def get_object(self, queryset=None):
        code = self.kwargs.get('code', None)
        return get_object_or_404(MiniURL, code=code)
```

Toujours pareil, la vue est associée à notre modèle, un template, et une URL à cibler en cas de réussite. Nous avons encore une fois la sélection de notre objet via le code assigné en base plutôt que la clé primaire. Cette fois-ci, nous devons créer notre template `supprimer.html`, qui demandera juste à l'utilisateur s'il est sûr de vouloir supprimer, et le cas échéant, le redirige vers la liste.

Code : Jinja - supprimer.html

```
<h1>Êtes-vous sûr de vouloir supprimer cette URL ?</h1>

<p>{{ mini_url.code }} -> {{ mini_url.url }} (créée le {{
mini_url.date|date:"DATE_FORMAT" }})</p>

<form method="post" action="">
    {% csrf_token %} <!-- On prend bien soin d'ajouter le csrf_token -->
    <input type="submit" value="Oui, supprime moi ça" /> - <a href="{%
url url_liste %}">Pas trop chaud en fait</a>
</form>
```

Encore une fois, notre ligne en plus dans le fichier `urls.py` ressemble beaucoup à celle de `URLUpdate` :

Code : Python - mini_url/urls.py

```
url(r'^supprimer/(?P<code>\w{6})/$', URLDelete.as_view(),
    name='url_delete'), # Ne pas oubliez l'import de URLDelete !
```

Afin de faciliter le tout, 2 liens ont été ajoutés dans la liste, définie dans le template `liste.html`, afin de pouvoir mettre à jour ou supprimer une URL rapidement :

Code : Jinja - liste.html

```
<h1>Le raccourcisseur d'URLs spécial cr&ecirc;pes bretonnes !</h1>

<p><a href="{% url url_nouveau %}">Raccourcir une URL.</a></p>

<p>Liste des URLs raccourcies :</p>
<ul>
    {% for mini in minis %}
    <li> <a href="{% url url_update mini.code %}">Mettre à jour</a> - <a
href="{% url url_delete mini.code %}">Supprimer</a>
    | {{ mini.url }} via <a href="http://{{ request.get_host }}{% url
url_redirection mini.code %}">{{ request.get_host }}{% url
url_redirection mini.code %}</a>
    {% if mini.pseudo %}par {{ mini.pseudo }}{% endif %} ({{
mini.nb_acces }} acc&egrave;s)</li>
    {% empty %}
```

```
<li>Il n'y en a pas actuellement.</li>
{% endfor %}
</ul>
```

Même refrain : on enregistre, et nous pouvons tester grâce au lien ajouté :



Notre vue, après avoir cliqué sur un des liens "Supprimer" qui apparaît dans la liste de liens

Ce chapitre touche enfin à sa fin. Néanmoins, nous n'avons même pas pu ... vous présenter toutes les spécificités des vues génériques ! Il existe en effet une multitude de classes de vues générique, mais aussi d'attributs et méthodes non abordés ici. Voici un diagramme UML des classes du module *django.views.generic* qui montre bel et bien l'étendue du sujet :

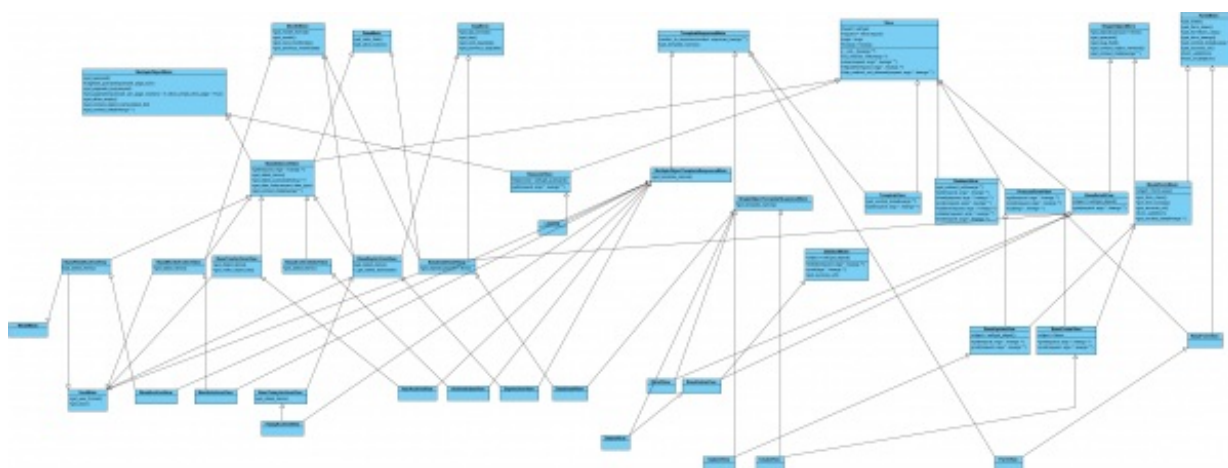


Diagramme de

classe UML de la partie Generic views

Nous avons essayé de vous présenter les plus communes, celles qui vous seront probablement le plus utile, mais il est clairement impossible de tout présenter sans être indigeste, vu la taille de ce diagramme. Par exemple, nous avons décidé de ne pas couvrir toutes les classes qui permettent de faire des pages de tri par date ou d'archives. Cependant, si vous souhaitez en savoir plus, ces deux liens vous seront plus qu'utiles :

- [Documentation officielle sur les vues génériques](#)
- [Documentation non officielle mais très complète, listant les attributs et méthodes de chaque classe](#)

Techniques avancées dans les modèles

Dans la partie précédente, nous avons vu comment créer, lier des modèles et faire des requêtes sur ceux-ci. Cependant, les modèles ne se résument pas qu'à des opérations basiques, Django propose en effet des techniques plus avancées qui peuvent se révéler très utiles dans certaines situations. C'est ces techniques que nous aborderons dans ce chapitre.

Les requêtes complexes avec Q

Django propose un outil très puissant et utile, nommé Q, pour créer des requêtes complexes sur des modèles. Il se peut que vous vous soyez demandés lors de l'introduction aux requêtes comment formuler des requêtes avec la clause « OU » (OR en anglais, par exemple, la catégorie de l'article que je recherche doit être "Crêpes" ou "Bretagne"), c'est ici qu'intervient juste l'objet Q, mais pas seulement. Il permet aussi de créer des requêtes de manière plus dynamique.

Avant tout, prenons un modèle simple pour illustrer nos exemples :

Code : Python

```
class Eleve(models.Model):
    nom = models.CharField(max_length=31)
    moyenne = models.IntegerField(default=10)

    def __unicode__(self):
        return u"Élève {0} ({1}/20 de moyenne)".format(self.nom,
self.moyenne)
```

Rajoutons quelques élèves dans le shell interactif :

Code : Python

```
>>> from test.models import Eleve
>>> Eleve(nom="Mathieu",moyenne=18).save()
>>> Eleve(nom="Maxime",moyenne=7).save() # Le vilain petit canard !
>>> Eleve(nom="Thibault",moyenne=10).save()
>>> Eleve(nom="Sofiane",moyenne=10).save()
```

Pour créer une requête dynamique, rien de plus simple, on peut formuler une condition avec un objet Q ainsi :

Code : Python

```
>>> from django.db.models import Q
>>> Q(nom="Maxime")
<django.db.models.query_utils.Q object at 0x222f650> #On voit bien
qu'on possède ici un objet de la classe Q
>>> Eleve.objects.filter(Q(nom="Maxime"))
[<Eleve: Élève Maxime (7/20 de moyenne)>]
>>> Eleve.objects.filter(nom="Maxime")
[<Eleve: Élève Maxime (7/20 de moyenne)>]
```

En réalité, les deux dernières requêtes sont équivalentes.



Quel intérêt d'utiliser Q dans ce cas ?

Comme dit plus haut, on peut en construire une clause "OU" :

Code : Python

```
Eleve.objects.filter(Q(moyenne__gt=16) | Q(moyenne__lt=8)) #On prend  
les moyennes strictement au-dessus de 16 ou en dessous de 8  
[<Eleve: Élève Mathieu (18/20 de moyenne)>, <Eleve: Élève Maxime  
(7/20 de moyenne)>]
```

L'opérateur `|` est généralement connu comme l'opérateur de l'opération de disjonction ("OU") dans l'algèbre de Boole, il est repris ici par Django pour désigner cette fois l'opérateur "OR" du langage SQL.

Sachez qu'il est également possible d'utiliser l'opérateur `&` pour signifier "ET" :

Code : Python

```
>>> Eleve.objects.filter(Q(moyenne=10) & Q(nom="Sofiane"))  
[<Eleve: Élève Sofiane (10/20 de moyenne)>]
```

Néanmoins, cet opérateur n'est pas indispensable, car il suffit de séparer les objets `Q` avec une virgule, le résultat est identique :

Code : Python

```
>>> Eleve.objects.filter(Q(moyenne=10), Q(nom="Sofiane"))  
[<Eleve: Élève Sofiane (10/20 de moyenne)>]
```

Il est aussi possible de "négationner" une condition. Autrement dit, demander la condition inverse ("NOT" en SQL). Cela se fait en précédant un objet `Q` dans une requête avec le caractère `~`.

Code : Python

```
>>> Eleve.objects.filter(Q(moyenne=10), ~Q(nom="Sofiane"))  
[<Eleve: Élève Thibault (10/20 de moyenne)>]
```

Pour aller plus loin, construisons quelques requêtes dynamiquement !

Tout d'abord, il faut savoir qu'un objet `Q` peut se construire de la façon suivante :

`Q(('moyenne', 10))` et est identique à `Q(moyenne=10)`

Quel intérêt ? Imaginons que nous devons obtenir tous les objets suivants s'ils remplissent une des conditions dans la liste suivante :

Code : Python

```
conditions = [ ('moyenne', 15), ('nom', 'Thibault'), ('moyenne', 18) ]
```

Nous pouvons construire plusieurs objets `Q` de la manière suivante :

Code : Python

```
objets_q = [Q(x) for x in conditions]
```

et les incorporer dans une requête ainsi (avec une clause "OU"):

Code : Python

```
import operator
Eleve.objects.filter(reduce(operator.or_, objets_q))
[<Eleve: Élève Mathieu (18/20 de moyenne)>, <Eleve: Élève Thibault
(15/20 de moyenne)>]
```



Qu'est-ce que reduce et operator.or_ ?

reduce est une fonction par défaut de Python qui permet d'appliquer une fonction à plusieurs valeurs successivement. Petit exemple pour comprendre plus facilement :

reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) va calculer (((1+2)+3)+4)+5) donc 15. La même chose sera faite ici, mais avec l'opérateur "OU" qui est accessible depuis operator.or_. En réalité, Python va donc faire :

Code : Python

```
Eleve.objects.filter(objets_q[0] | objets_q[1] | objets_q[2])
```

C'est une méthode très puissante et très pratique !

L'agrégation

Il est souvent utile de vouloir extraire une information spécifique à travers plusieurs entrées d'un seul et même modèle. Si nous reprenons nos élèves de la sous-partie précédente, leur professeur aura plus que probablement un jour besoin de calculer la moyenne globale des élèves. Pour ce faire, Django fournit plusieurs outils qui permettent de tels calculs très simplement. On parle de méthode d'agrégation.

En effet, si nous voulons obtenir la moyenne des moyennes de nos élèves (pour rappel, Mathieu (moyenne de 18), Maxime (7), Thibault (10) et Sofiane(10)), on peut procéder à partir de la méthode aggregate :

Code : Python

```
from django.db.models import Avg
>>> Eleve.objects.aggregate(Avg('moyenne'))
{'moyenne__avg': 11.25}
```

En effet, $(18+7+10+10)/4 = 11.25$!

Cette méthode prend à chaque fois une fonction spécifique fournie par Django, comme Avg (pour Average, signifiant « moyenne » en anglais) et s'applique sur un champ du modèle. Cette fonction va ensuite parcourir toutes les entrées du modèle et effectuer les calculs propre à celle-ci.

Notons que la valeur retournée par la méthode est un dictionnaire, avec à chaque fois une clé générée automatiquement à partir du nom de la colonne utilisée et de la fonction appliquée (nous avons utilisé la fonction Avg dans la colonne 'moyenne', Django renvoie donc 'moyenne__avg'), avec la valeur calculée correspondante (ici 11.25 donc).

Il existe d'autres fonctions comme Avg, dont notamment :

- Max : prend la plus grande valeur
- Min : prend la plus petite valeur
- Count : compte le nombre d'entrées

Il est même possible d'utiliser plusieurs de ces fonctions en même temps :

Code : Python

```
>>> Eleve.objects.aggregate(Avg('moyenne'),
Min('moyenne'),Max('moyenne'))
{'moyenne__max': 18, 'moyenne__avg': 11.25, 'moyenne__min': 7}
```

Si vous souhaitez préciser une clé spécifique, il suffit de la faire précéder de la fonction :

Code : Python

```
>>> Eleve.objects.aggregate(Moyenne=Avg('moyenne'),
Minimum=Min('moyenne'),Maximum=Max('moyenne'))
{'Minimum': 7, 'Moyenne': 11.25, 'Maximum': 18}
```

Bien évidemment, il est également possible d'appliquer une agrégation sur un QuerySet obtenu par la méthode filter par exemple :

Code : Python

```
>>>
Eleve.objects.filter(nom__startswith="Ma").aggregate(Avg('moyenne'),
Count('moyenne'))
{'moyenne__count': 2, 'moyenne__avg': 12.5}
```

Étant donné qu'il n'y a que Mathieu et Maxime comme prénoms qui commencent par "Ma", uniquement ceux-ci seront sélectionnés, comme l'indique moyenne_count.

En réalité, la fonction Count est assez inutile ici, d'autant plus qu'une méthode pour obtenir le nombre d'entrées dans un QuerySet existe déjà :

Code : Python

```
>>> Eleve.objects.filter(nom__startswith="Ma").count()
2
```

Cependant, cette fonction peut se révéler bien plus intéressante lorsque nous l'utilisons avec des liaisons entre modèles. Pour ce faire, rajoutons un autre modèle :

Code : Python

```
class Cours(models.Model):
    nom = models.CharField(max_length=31)
    eleves = models.ManyToManyField(Eleve)

    def __unicode__(self):
        return self.nom
```

Créons 2 cours :

Code : Python

```
>>> c1 = Cours(nom="Maths")
```

```
>>> c1.save()
>>> c1.eleves.add(*Eleve.objects.all())
>>> c2 = Cours(nom="Anglais")
>>> c2.save()
>>> c2.eleves.add(*Eleve.objects.filter(nom__startswith="Ma"))
```

Il est tout à fait possible d'utiliser les agrégations depuis des liaisons comme une ForeignKey, ou comme ici, avec un ManyToManyField :

Code : Python

```
>>> Cours.objects.aggregate(Max("eleves__moyenne"))
{'eleves__moyenne__max': 18}
```

Nous avons été chercher la meilleure moyenne dans les élèves de tous les cours enregistrés.

Il est également possible de compter le nombre d'affiliations à des cours :

Code : Python

```
>>> Cours.objects.aggregate(Count("eleves"))
{'eleves__count': 6}
```

En effet, on a 6 "élèves", à savoir 4+2 car Django ne fait vérifie pas si un élève est déjà dans un autre cours ou non.

Pour terminer, abordons une dernière fonctionnalité utile. Il est possible de rajouter des attributs à un objet selon les objets auxquels il est lié. On parle d'annotation. Exemple :

Code : Python

```
>>>
Cours.objects.annotate(Avg("eleves__moyenne"))[0].eleves__moyenne__avg
11.25
```

Un nouvel attribut a été créé. Au lieu de retourner les valeurs dans un dictionnaire, elles sont désormais directement ajoutées à l'objet lui-même. Il est bien évidemment possible de redéfinir le nom de l'attribut comme vu précédemment :

Code : Python

```
>>>
Cours.objects.annotate(Moyenne=Avg("eleves__moyenne"))[1].Moyenne
12.5
```

Et pour terminer en beauté, il est même possible d'utiliser l'attribut créé dans des méthodes du QuerySet comme filter, exclude ou order_by ! Par exemple :

Code : Python

```
>>>
Cours.objects.annotate(Moyenne=Avg("eleves__moyenne")).filter(Moyenne__gte=12)
[<Cours: Anglais>]
```


En définitive, l'agrégation et l'annotation sont des outils réellement puissants qu'il ne faut pas hésiter à utiliser si l'occasion se présente !

L'héritage de modèles

Les modèles étant des classes, ils possèdent les mêmes propriétés que n'importe quelle classe, y compris l'héritage de classes. Néanmoins, Django propose trois méthodes principales pour gérer l'héritage de modèles qui interagiront différemment avec la base de données. Nous aborderons ici une à une.

Les modèles parents abstraits

Les modèles parents abstraits sont utiles lorsque vous souhaitez utiliser plusieurs méthodes et attributs dans différents modèles, sans devoir les réécrire à chaque fois. Tout modèle héritant d'un modèle abstrait récupère automatiquement toutes les caractéristiques de la classe dont elle hérite. La grande particularité d'un modèle abstrait réside dans le fait que Django ne l'utilisera pas comme représentation pour créer une table dans la base de données. En revanche, tous les modèles qui hériteront de ce parent abstrait auront bel et bien une table qui leur sera dédiée.

Afin de rendre un modèle abstrait, il suffit de lui assigner l'attribut `abstract=True` dans sa sous-classe `Meta`. Django se charge entièrement du reste.

Pour illustrer cette méthode, prenons un exemple simple :

Code : Python

```
class Document(models.Model):
    titre = models.CharField(max_length=255)
    date_ajout = models.DateTimeField(auto_now_add=True,
    verbose_name="Date d'ajout du document")
    auteur = models.CharField(max_length=255, null=True, blank=True)

    class Meta:
        abstract = True

class Article(Document):
    contenu = models.TextField()

class Image(Document):
    image = models.ImageField(upload_to="images")
```

Ici, deux tables seront créées dans la base de données : `Article` et `Image`. Le modèle `Document` ne sera pas utilisé comme table, étant donné que celui-ci est abstrait. En revanche, les tables `Article` et `Image` auront bien les champs de `Document` (donc par exemple, la table `Article` aura les champs `titre`, `date_ajout`, `auteur` et `contenu`).

Bien entendu, il est impossible de faire des requêtes sur un modèle abstrait, celui-ci n'ayant aucune table dans la base de données pour enregistrer des données. Vous ne pouvez interagir avec les champs du modèle abstrait que depuis les modèles qui en héritent.

Les modèles parents classiques

Contrairement aux modèles abstraits, il est possible d'hériter de modèles tout à fait normaux. Si un modèle hérite d'un autre modèle non-abstrait, il n'y aura aucune différence pour ce dernier, il sera manipulable comme n'importe quel modèle. Django créera une table pour le modèle parent et le modèle enfant.

Prenons un exemple simple :

Code : Python

```
class Lieu(models.Model):
    nom = models.CharField(max_length=50)
    adresse = models.CharField(max_length=100)
```

```
def __unicode__(self):  
    return self.nom  
  
class Restaurant(Lieu):  
    menu = models.TextField()
```

À partir de ces deux modèles, Django créera bien deux tables, une pour Lieu, l'autre pour Restaurant. Il est important de noter que la table Restaurant ne contient pas les champs de Lieu (à savoir nom et adresse). En revanche, elle contient bien le champ menu et une clé étrangère vers Lieu que le framework ajoutera tout seul.

En effet, si Lieu est un modèle tout à fait classique, Restaurant agira un peu différemment.

Lorsqu'on sauvegarde une nouvelle instance de Restaurant dans la base de données, une nouvelle entrée sera créée dans la table correspondant au modèle Restaurant, mais également dans celle correspondante à Lieu. Les valeurs des deux attributs nom et adresse seront enregistrés dans une entrée de la table Lieu, et l'attribut menu sera enregistré dans une entrée de la table Restaurant. Cette dernière entrée contiendra donc également la clé étrangère vers l'entrée dans la table Lieu qui possède les données associées.

Pour résumer, l'héritage classique s'apparente à la liaison de deux classes avec une clé étrangère telle que nous en avons vu dans le chapitre introductif sur les modèles, excepté que Django qui se charge de réaliser lui-même cette liaison.

Sachez aussi que lorsque vous créez un objet Restaurant, vous créez aussi un objet Lieu tout à fait banal qui peut être obtenu comme n'importe quel objet Lieu créé précédemment. De plus, même si les attributs du modèle parent sont dans une autre table, le modèle fils a bien hérité de toutes ses méthodes et attributs :

Code : Python

```
>>> Restaurant(nom="La crêperie bretonne", adresse="42 Rue de la  
crêpe 35000 Rennes", menu="Des crêpes !").save()  
>>> Restaurant.objects.all()  
[<Restaurant: La crêperie bretonne>]  
>>> Lieu.objects.all()  
[<Lieu: La crêperie bretonne>]
```

Pour finir, tous les attributs de Lieu sont directement accessibles depuis un objet Restaurant :

Code : Python

```
>>> resto = Restaurant.objects.all()[0]  
>>> print resto.nom+", "+resto.menu  
La crêperie bretonne, Des crêpes !
```

En revanche, il n'est pas possible d'accéder aux attributs spécifiques de Restaurant depuis une instance de Lieu :

Code : Python

```
>>> lieu = Lieu.objects.all()[0]  
>>> print lieu.nom  
La crêperie bretonne  
>>> print lieu.menu #Ca ne marche pas  
Traceback (most recent call last):  
  File "<console>", line 1, in <module>  
AttributeError: 'Lieu' object has no attribute 'menu'
```

Pour accéder à l'instance de Restaurant associée à Lieu, Django crée tout seul une relation vers celle-ci qu'il nommera selon le

nom de la classe fille :

Code : Python

```
>>> print type(lieu.restaurant)
<class 'blog.models.Restaurant'>
>>> print lieu.restaurant.menu
Des crêpes !
```

Les modèles proxy

Dernière technique d'héritage avec Django, et probablement la plus complexe, il s'agit de modèles proxy (en français, des modèles "passerelles").

Le principe est trivial : un modèle proxy hérite de tous les attributs et méthodes du modèle parent, mais aucune table ne sera créée dans la base de données pour le modèle fils. En effet, le modèle fils sera en quelque sorte une passerelle vers le modèle parent (tout objet créé avec le modèle parent sera accessible depuis le modèle fils, et vice-versa).

Quel intérêt ? À première vue, il n'y en a pas, mais pour quelque raison de structure du code, d'organisation, d'etc, on peut rajouter des méthodes dans le modèle proxy, ou modifier des attributs de la sous-classe Meta sans que le modèle d'origine ne soit altéré, et continuer à utiliser les mêmes données.

Petit exemple de modèle proxy qui hérite du modèle Restaurant que nous avons défini tout à l'heure (notons qu'il est possible d'hériter d'un modèle qui hérite lui-même d'un autre !) :

Code : Python

```
class RestoProxy(Restaurant):
    class Meta:
        proxy = True #On spécifie qu'il s'agit d'un proxy
        ordering = ["nom"] #On change le tri par défaut, tous les
        QuerySet seront triés selon le nom de chaque objet

    def crepes(self):
        if u"crêpe" in self.menu: #Il y a des crêpes dans le menu
            return True
        return False
```

Depuis ce modèle, il est donc possible d'accéder aux données enregistrées du modèle parent, tout en bénéficiant des méthodes et attributs supplémentaires :

Code : Python

```
>>> from blog.models import RestoProxy
>>> print RestoProxy.objects.all()
[<RestoProxy: La crêperie bretonne>]
>>> resto = RestoProxy.objects.all()[0]
>>> print resto.adresse
42 Rue de la crêpe 35000 Rennes
>>> print resto.crepes()
True
```

Simplifions nos templates : filtres, tags et contexte

Comme nous l'avons vu rapidement dans le premier chapitre sur les templates, Django offre une panoplie de filtres et de tags. Cependant, il se peut que vous ayez un jour un besoin particulier impossible à réaliser avec les filtres et tags de base. Heureusement, Django permet également de créer nos propres filtres et tags, et même de générer des variables par défaut lors de la construction d'un template (ce qu'on appelle le contexte du template). Nous aborderons ces différents éléments dans ce chapitre.

Préparation du terrain : architecture des filtres et tags

Pour construire nos propres filtres et tags, Django requiert que ces derniers soient placés dans une application, tout comme les vues ou les modèles. A partir d'ici, on retrouve deux écoles dans la communauté de Django :

- Soit votre fonctionnalité est propre à une application (par exemple un filtre utilisé uniquement lors de l'affichage d'articles), alors vous pouvez directement le(s) placer au sein de l'application concernée ;
- soit vous créez une application à part, qui regroupe tous vos filtres et tags personnalisés.

Une fois ce choix fait, la procédure est identique : l'application choisie doit contenir un dossier nommé "**templatetags**" (attention au *s* final !), dans lequel il faut créer un fichier Python par groupe de filtres/tags (plus de détails sur l'organisation sur ces fichiers viendront plus tard).



Le dossier *templatetags* doit en réalité être un module Python classique afin que les fichiers qu'il contient puissent être importés. Il est donc impératif de créer un fichier `__init__.py` vide, sans quoi Django ne pourra rien en faire.

La nouvelle structure de l'application *blog* est donc la suivante :

Code : Autre

```
blog/  
    __init__.py  
    models.py  
    templatetags/  
        __init__.py    # À ne pas oublier  
        blog_extras.py  
    views.py
```

Une fois les fichiers créés, il est nécessaire de spécifier une instance de classe qui nous permettra d'enregistrer nos filtres et tags, de la même manière que dans nos fichiers *admin.py* avec `admin.site.register()`. Pour ce faire, il faut déclarer les deux lignes suivantes au début du fichier *blog_extras.py* :

Code : Python

```
from django import template  
  
register = template.Library()
```

L'application incluant les templatetags doit être incluse dans le fameux `INSTALLED_APPS` de notre configuration, si vous avez décidé d'ajouter vos tags et filtres personnalisés dans une application spécifique. Une fois les nouveaux tags et filtres codés, il sera possible de les intégrer dans n'importe quel template du projet via la ligne suivante :

Code : Jinja

```
{% load blog_extras %}
```



Le nom `blog_extras` vient du nom de fichier que nous avons renseigné plus haut, à savoir `blog_extras.py`.



Tous les dossiers *templatetags* de toutes les applications partagent le même espace de noms. Si vous utilisez des filtres et tags de plusieurs applications, veillez à ce que leur noms de fichiers soient différents, afin qu'il n'y ait pas de conflit.

Nous pouvons désormais passer au vif du sujet, à savoir la création de tags et filtres !

Personnaliser l'affichage de données avec nos propres filtres

Commençons par les filtres. En soi, un filtre est une fonction classique qui prend 1 ou 2 arguments :

- La variable à afficher, qui peut être n'importe quel objet en Python ;
- et de façon facultative, un paramètre.

Comme petit rappel au cas où vous auriez la mémoire courte, voici un filtre, un sans paramètre, le deuxième avec :

Code : Jinja

```
{{ texte|upper }} -> Filtre upper sur la variable "texte"
{{ texte|truncatewords:80 }} -> Filtre truncatewords, avec comme
argument "80" sur la variable "texte"
```

Les fonctions Python associées à ces filtres ne sont appelés qu'au sein du template. Pour cette raison, il faut éviter de lancer des exceptions, et toujours renvoyer un résultat. En cas d'erreur, il est plus prudent de renvoyer l'entrée de départ ou une chaîne vide, afin d'éviter des effets de bords lors du « chaînage » de filtres par exemple.

Un premier exemple de filtre sans argument

Attaquons la réalisation de notre 1er filtre ! Pour commencer, prenons comme exemple le modèle "Citation" de Wikipedia : nous allons entourer la chaine fournie par des guillemets français doubles.

Ainsi, si dans notre template on a `{{ "Bonjour le monde !" |citation }}`, le résultat dans notre page sera « Bonjour le monde ! ».

Pour ce faire, il faut ajouter une fonction nommée *citation* dans `blog_extras.py`. Cette fonction n'a pas d'argument particulier et son écriture est assez intuitive :

Code : Python

```
def citation(texte):
    """
    Affiche le texte passé en paramètre, entourée de guillemets
    français
    doubles et d'espaces insécables
    """
    return "«&nbsp;%s&nbsp;»" % texte
```

Une fois la fonction écrite, il faut préciser au framework d'attacher cette méthode au filtre qui a pour nom "citation". Encore une fois, il y a deux façons différentes de procéder :

- Soit en ajoutant la ligne `@register.filter` comme décorateur de la fonction. L'argument `name` peut être indiqué pour choisir le nom du filtre ;
- ou en appelant la méthode `register.filter('citation', citation)`.

Notons qu'avec ces 2 méthodes, le nom du filtre n'est donc pas directement lié au nom de la fonction, et cette dernière aurait pu s'appeler `filtre_citation` ou autre, cela n'aurait posé aucun souci tant qu'elle est correctement renseignée par la suite.

Ainsi, ces trois fonctions sont équivalentes :

Code : Python

```

#-*- coding:utf8 -*-
from django import template

register = template.Library()

@register.filter
def citation(texte):
    """
    Affiche le texte passée en paramètre, entourée de guillemets
    français
    doubles et d'espaces insécables
    """
    return "<&nbsp; %s &nbsp;>" % texte

@register.filter(name='citation_nom_différent')
def citation2(texte):
    """ [...] """
    return "<&nbsp; %s &nbsp;>" % texte

def citation3(texte):
    """ [...] """
    return "<&nbsp; %s &nbsp;>" % texte

register.filter('citation3', citation3)

```



Par commodité, nous n'utiliserons plus que la 1ère et 2ème méthode dans ce cours. La dernière est pour autant tout à fait valide et libre à vous de l'utiliser si vous préférez celle-ci.

Nous pouvons maintenant essayer le nouveau filtre dans un template. Il faut tout d'abord charger les filtres dans notre template, via le tag **load**, vu dans la sous-partie précédente, puis appeler notre filtre *citation* sur une chaîne de caractères quelconque :

Code : Jinja

```

{% load blog_extras %}
Un jour, une certaine personne m'a dit : {{ "Bonjour le monde
!"|citation }}

```

Et là... c'est le drame ! En effet, voici le résultat :

Un jour, une certaine personne m'a dit : « Bonjour le monde ! ».



Mais pourquoi les espaces insécables () sont-ils échappés ?

Par défaut, Django échappe automatiquement tous les caractères spéciaux des chaînes de caractères affichées dans un template, ainsi que le résultat des filtres. Nous allons donc devoir préciser au framework que le résultat de notre filtre est contrôlé et sécurisé, et qu'il est pas nécessaire de l'échapper. Pour cela, il est nécessaire de transformer un peu l'enregistrement de notre fonction avec `register()`. La méthode `filter()` peut prendre comme argument `is_safe`, qui permet de signaler au framework par la suite que notre chaîne est sûre :

Code : Python

```

@register.filter(is_safe=True)
def citation(texte):
    """

```

```

Affiche le texte passée en paramètre, entourée de guillemets
français
doubles et d'espaces insécables
"""
    return "<&nbsp;%s&nbsp;>" % texte

```

De cette façon, tout le HTML renvoyé par le filtre est correctement interprété et on obtient le résultat voulu :

Un jour, une certaine personne m'a dit : « Bonjour le monde ! ».

Cependant, un problème se pose avec cette méthode. En effet, si du HTML est présent dans la chaîne donnée en paramètre, il sera également interprété. Ainsi, si dans le template nous remplaçons l'exemple précédent par `{{ "Bonjour le monde !" | citation }}`, alors le mot *Bonjour* sera en gras. En soi, ce n'est pas un problème si vous êtes sûr de la provenance de la chaîne de caractères. Il se pourrait en revanche que, parfois, vous devez afficher des données entrées par vos utilisateurs, et à ce moment là, n'importe quel visiteur mal intentionné pourrait y placer du code HTML dangereux, ce qui conduirait à des failles de sécurité. Pour régler ça, nous allons échapper les caractères spéciaux de notre argument de base. Ceci peut être fait via la fonction `espace` du module `django.utils.html`. Au final, voici ce que nous obtenons :

Code : Python

```

#-*- coding:utf8 -*-
from django import template
from django.utils.html import escape

register = template.Library()

@register.filter(is_safe=True)
def citation(texte):
    """
    Affiche le texte passé en paramètre, entouré de guillemets français
    doubles et d'espaces insécables.
    """
    return "<&nbsp;%s&nbsp;>" % escape(texte)

```

Finalement, notre chaîne est entourée de guillemets et d'espaces insécables corrects, mais l'intérieur du message est tout de même échappé.

Un filtre avec arguments

Nous avons pour le moment traité uniquement le cas des filtres sans paramètre. Cependant, il peut arriver que l'affichage doive être **différent selon un paramètre spécifié**, et ce indépendamment de la variable de base.

Un exemple parmi tant d'autres est la troncature de texte, il existe même déjà un filtre pour couper une chaîne à une certaine position. Nous allons ici plutôt réaliser un filtre qui va couper une chaîne après un certains nombre de caractères mais sans couper en plein milieu d'un mot.

Comme nous l'avons précisé tout à l'heure, la forme d'un filtre avec un argument est la suivante :

Code : Jinja

```

{{ ma_chaine|smart_truncate:40 }}

```

Nous souhaitons ici appeler un nouveau filtre `smart_truncate` sur la variable `ma_chaine`, tout en lui passant en argument le nombre 40. La structure du filtre sera similaire à l'exemple précédent. Il faudra cependant bien vérifier que le paramètre soit bien un nombre et qu'il y ait des caractères à tronquer. Voici un début de fonction :

Code : Python

```
def smart_truncate(texte, nb_caracteres):

    # On vérifie tout d'abord que l'argument passé est bien un
    nombre
    try:
        nb_caracteres = int(nb_caracteres)
    except ValueError:
        return texte # Retour de la chaine original sinon

    # Si la chaine est plus petite que le nombre de caractères
    maximum voulu,
    # on renvoie directement la chaîne telle quelle.
    if len(texte) <= nb_caracteres:
        return texte

    # [...]
```

La suite de la fonction est tout aussi classique : on coupe notre chaîne au nombre de caractères maximum voulu, et on retire la dernière suite de lettres, si jamais cette chaîne est coupée en plein milieu d'un mot :

Code : Python

```
def smart_truncate(texte, nb_caracteres):
    """
    Coupe la chaîne de caractères jusqu'au nombre de caractères
    souhaités,
    sans couper la nouvelle chaîne au milieu d'un mot.
    Si la chaîne est plus petite, elle est renvoyée sans points de
    suspension.
    ---
    Exemple d'utilisation :
    {{ "Bonjour tout le monde, c'est Diego"|smart_truncate:18 }} renvoie
    "Bonjour tout le..."
    """

    # On vérifie tout d'abord que l'argument passé est bien un
    nombre
    try:
        nb_caracteres = int(nb_caracteres)
    except ValueError:
        return texte # Retour de la chaîne original sinon

    # Si la chaîne est plus petite que le nombre de caractères
    maximum voulus,
    # on renvoie directement la chaîne telle quelle.
    if len(texte) <= nb_caracteres:
        return texte

    # Sinon, on coupe au maximum, tout en gardant le caractère
    suivant
    # pour savoir si on a coupé à la fin d'un mot ou en plein
    milieu
    texte = texte[:nb_caracteres + 1]

    # On vérifie d'abord que le dernier caractère n'est pas un
    espace
    # autrement, il est inutile d'enlever le dernier mot !
    if texte[-1:] != ' ':
        mots = texte.split(' ')[:-1]
        texte = ' '.join(mots)
    else:
        texte = texte[0:-1]

    return texte + '...'
```


Il ne reste plus qu'à enregistrer notre filtre (via le décorateur `@register.filter` au dessus de la ligne `def smart_truncate`(texte, nb_caracteres) : par exemple) et vous pouvez dès à présent tester ce tout nouveau filtre :

Code : Jinja

```
<p>
{{ "Bonjour"|smart_truncate:14 }}<br />
{{ "Bonjour tout le monde"|smart_truncate:15 }}<br />
{{ "Bonjour tout le monde, c'est bientôt Noël"|smart_truncate:18
}}<br />
{{ "To be or not to be, that's the question"|smart_truncate:16 }}<br
/>
</p>
```

Ce qui affiche le paragraphe suivant :

Citation

Bonjour
Bonjour tout le...
Bonjour tout le...
To be or not to...

Pour finir, il est possible de mixer le cas filtre sans argument et filtre avec un argument. Dans notre cas de troncature, nous pouvons par exemple vouloir par défaut tronquer à partir du 20ème caractère, si aucun argument n'est passé. Dans ce cas, la méthode est classique : on peut indiquer qu'un argument est facultatif et lui **donner une valeur par défaut**. Il suffit de changer la déclaration de la fonction par :

Code : Python

```
def smart_truncate(texte, nb_caracteres=20):
```

Désormais, la syntaxe suivante est acceptée :

Code : Jinja

```
{{ "To be or not to be, that's the question"|smart_truncate }}<br />
```

et renvoie "To be or not to be,...".

Les contextes de templates

Avant d'attaquer les tags, nous allons aborder un autre point essentiel qui est la création de "**template context processor**" (ou en français, des processeurs de contexte de templates). Le but des Templates Context Processor est de pré-remplir le contexte de la requête et ainsi disposer de données dans tous les templates de notre projet. Le contexte est l'ensemble des variables disponible dans votre template. Prenons l'exemple suivant :

Code : Python

```
return render(request, 'blog/archives.html', {'news': news, 'date':  
date_actuelle})
```

Ici, on indique au template les variables `news` et `date_actuelle` qui seront incorporées au contexte, avec les noms `news` et `date`. Cependant, notre contexte ne contiendra par défaut pas que ces variables, il est même possible d'en ajouter davantage, si le besoin se fait sentir.

Pour mieux comprendre l'utilité des contextes, démarrons par un petit exemple.

Un exemple maladroit : afficher la date sur toutes nos pages

Il arrive que vous ayez besoin d'accéder à certaines variables depuis tous vos templates, et que ceux-ci soient enregistrés dans votre base de données, un fichier, cache, etc.

Imaginons que vous souhaitez afficher dans tous vos templates la date du jour. Une première idée serait de récupérer la date sur chacune des vues :

Code : Python

```
from django.shortcuts import render
from datetime import datetime

def accueil(request):
    date_actuelle = datetime.now()
    # [...] Récupération d'autres données (exemple : une liste de
    news)
    return render(request, 'accueil.html', locals())

def contact(request):
    date_actuelle = datetime.now()
    return render(request, 'contact.html', locals())
```

Une fois cela fait, il suffit après d'intégrer la date via `{{ date_actuelle }}` dans un template parent, dont tous les autres templates seront étendus. Néanmoins, cette méthode est lourde et répétitive, c'est ici que les processeurs de contexte entrent en jeu.



Sachez que l'exemple pris ici n'est pas réellement pertinent puisque Django permet déjà par défaut d'afficher la date avec le tag `{% now %}`. Néanmoins il s'agit d'un exemple simple et concret qui s'adapte bien à l'explication.

Factorisons encore et toujours

Pour résoudre ce problème, nous allons créer une fonction qui sera appelé à chaque page, et qui se chargera d'incorporer la date dans les données disponible de façon automatique.

Tout d'abord, créez un fichier Python, que nous appellerons `context_processors.py`, par convention, dans une de vos applications. Vu que cela concerne tout le projet, il est même conseillé de le créer dans le sous-dossier ayant le même nom que votre projet ("`crepes_bretonnes`" dans le cas de ce cours).

Dans ce fichier, nous allons coder une ou plusieurs fonctions, qui renverront des dictionnaires de données que le framework intégrera à tous nos templates.

Tout d'abord, écrivons notre fonction qui va récupérer la date actuelle. La fonction ne prend qu'un paramètre, qui est notre déjà très connu objet `request`.

En retour, la fonction renvoie un dictionnaire, contenant les valeurs à intégrer dans les templates, assez similaire au dictionnaire qu'on passe à la fonction `render` pour construire un template. Par exemple :

Code : Python

```
from datetime import datetime

def get_infos(request):
    date_actuelle = datetime.now()
    return {'date_actuelle': date_actuelle}
```

Sachez que Django **exécute d'abord la vue** et seulement après le contexte. Faites donc attention à prendre des noms de variables suffisamment explicites et qui ont peu de chances de se retrouver dans vos vues, et donc d'entrer en collision. Si jamais vous appelez une variable "date_actuelle", elle sera tout simplement écrasée par la fonction ci-dessus.

Il faut maintenant indiquer au framework d'exécuter cette fonction à chaque page. Pour cela, nous allons encore une fois nous plonger dans le fichier *settings.py* et y définir une nouvelle variable. A chaque page, Django exécute et récupère les dictionnaires de plusieurs fonctions, listées dans la variable `TEMPLATE_CONTEXT_PROCESSORS`. Par défaut, elle est égale au tuple suivant, qui n'est pas présent dans le fichier *settings.py* :

Code : Python

```
TEMPLATE_CONTEXT_PROCESSORS =
(
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
)
```

On peut voir que Django utilise lui-même quelques fonctions, afin de nous fournir quelques variables par défaut. Pour éviter de casser ce processus, il faut **recopier cette liste** et juste **ajouter à la fin** nos fonctions :

Code : Python

```
TEMPLATE_CONTEXT_PROCESSORS =
(
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.debug",
    "django.core.context_processors.i18n",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.tz",
    "django.contrib.messages.context_processors.messages",
    "crepes_bretonnes.context_processors.get_infos",
)
```

Nous pouvons désormais utiliser notre variable `date_actuelle` dans tous nos templates et afficher fièrement la date sur notre blog :

Code : Jinja

```
<p>Bonjour à tous, nous sommes le {{ date_actuelle }} et il fait
beau en Bretagne !</p>
```

Et peu importe le template où vous intégrez cette ligne, vous aurez forcément le résultat suivant (si vous n'avez pas de variable `date_actuelle` dans votre vue correspondante, bien sûr) :

Code : HTML

```
Bonjour à tous, nous sommes le 15 novembre 2012 23:58:16 et il fait
beau en Bretagne !
```

Petit point technique sur l'initialisation du contexte

Attention : dans ce cours nous avons toujours utilisé `render()` comme retour de nos vues (hormis quelques cas précis où

nous avons utilisé `HttpResponse`). Comme nous l'on avons précisé dans le 1er chapitre sur les templates, la fonction `render` est un "raccourci", effectuant plusieurs actions en interne, nous évitant la réécriture de plusieurs lignes de code. Cette méthode prend notamment en charge le fait de charger le contexte !

Cependant toutes les fonctions de `django.shortcuts` ne le font pas, comme par exemple `render_to_response()`, dont nous n'avons pas parlé et qui fonctionne de la façon suivante pour le cas des archives de notre blog :

Code : Python

```
from django.shortcuts import render_to_response
[...]  
return render_to_response('blog/archives.html', locals())
```

Si vous rechargez la page, vous remarquerez que la date actuelle a disparu, et que ceci apparaît *"Bonjour à tous, nous sommes le et il fait beau en Bretagne !"*. En effet, par défaut `render_to_response` ne prend pas en compte les fonctions contenues dans `TEMPLATE_CONTEXT_PROCESSOR...` Pour régler ce problème il faut à chaque fois ajouter un argument :

Code : Python

```
return render_to_response('blog/archives.html', locals(),  
context_instance=RequestContext(request))
```

... ce qui est plus lourd à écrire ! Cependant certains utilisateurs avancés peuvent préférer cette méthode afin de gérer de façon précise le contexte à utiliser.

En bref, faites attention à vos contextes si jamais vous vous écartez de la fonction `render`.

Des structures plus complexes : les custom tags

Nous avons vu précédemment que les filtres nous permettent de faire de légères opérations sur nos variables, afin de factoriser un traitement qui pourra être souvent répété dans notre template (par exemple la mise en forme d'une citation). Nous allons maintenant aborder les tags, qui sont légèrement plus complexes à mettre en œuvre, mais bien plus puissants.

Alors que les filtres peuvent être comparé à des fonctions, les tags doivent être décomposés en deux parties : la structuration du tag et son rendu. Pour définir de façon précise un tag, on doit **préciser comment l'écrire et ce qu'il renvoie**.

Pour mieux comprendre, regardons comment marche un template avec Django.

A la compilation du template, Django découpe votre fichier template en plusieurs nœuds de plusieurs types. Prenons le cas du template suivant :

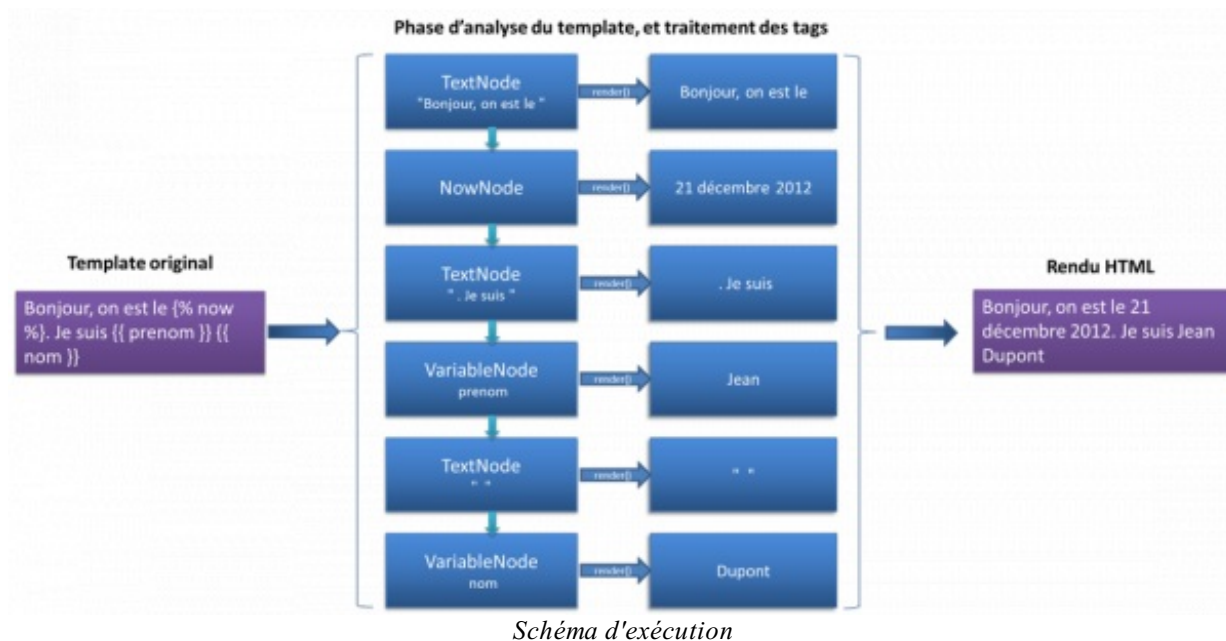
Code : Jinja

```
Bonjour, nous sommes le {% now %}. Je suis {{ prenom }} {{ nom|upper  
}}
```

Ici, les nœuds détectés lors de la lecture du template seront :

- `TextNode`: "Bonjour, nous sommes le "
- `Now node` (sans argument)
- `TextNode`: ". Je suis "
- `VariableNode` : `prenom`
- `TextNode` " "
- `VariableNode` : `nom` et un `FilterExpression` "upper"

Lors de l'exécution de la fonction `render` à la fin d'une vue, Django se charge d'appeler la méthode `render` de chaque nœud et concatène le tout.



Lorsque nous créons un nouveau tag, la fonction appelée à la compilation doit renvoyer un objet dont sa classe hérite de *Node*, avec sa propre méthode *render*.

C'est à partir de ce principe que nous obtenons les deux étapes de description d'un tag, à savoir :

- Décrire comment il peut être écrit pour être reconnu (fonction de compilation) ;
- décrire ce qu'il rend, via une classe contenant au moins une fonction *render* (fonction de rendu).

Première étape : la fonction de compilation

A chaque fois que le parseur de template rencontre un tag, il appelle la méthode correspondant au nom du tag enregistré comme pour nos filtres. La fonction se charge ici de vérifier si les paramètres fournis sont corrects ou de renvoyer une erreur si jamais le tag est mal utilisé. Nous allons nous baser sur un exemple assez simple pour commencer : affichons un nombre aléatoire compris entre 2 arguments. Cette opération est notamment impossible avec un filtre, du moins, pas proprement.

Notre tag pourra être utilisé de la façon suivante : `{% random 0 42 %}` et renverra donc un nombre entier situé entre 0 et 42. Il faudra faire attention à ce que les paramètres soient bien des entiers, et que le premier soit inférieur au second.

Contrairement au filtre, Django requiert que **notre méthode prennent 2 arguments précis** : *parser* qui est l'objet en charge de parser le template actuel (que l'on utilisera pas ici) et *token* qui contient les informations sur le tag actuel, comme les paramètres passés. *token* contient de plus quelques méthodes sympatiques qui vont nous simplifier le traitement des paramètres. Par exemple, la méthode *split_contents()* permet de séparer les arguments dans une liste. Il est extrêmement déconseillé d'utiliser la méthode classique *token.contents.split(' ')*, qui pourrait « casser » vos arguments si jamais il y a des chaînes de caractères avec des espaces.

Voici un bref exemple de fonction de compilation :

Code : Python

```
def random(parser, token):
    """ Tag générant un nombre aléatoire, entre les bornes données
    en argument """
    # Séparation des paramètres contenu dans l'objet token
    # Le premier élément du token est toujours le nom du tag en
    cours
    try:
        nom_tag, begin, end = token.split_contents()
    except ValueError:
        msg = 'Le tag %s doit prendre exactement deux arguments.'
        % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)

    # On vérifie que nos deux paramètres sont bien des entiers
```

```

try:
    begin, end = int(begin), int(end)
except ValueError:
    msg = 'Les arguments du tag %s sont obligatoirement des entiers.' % nom_tag
    raise template.TemplateSyntaxError(msg)

# On vérifie si le premier est inférieur au second
if begin > end:
    msg = 'L\'argument "begin" doit obligatoirement être inférieur à l\'argument "end" dans le tag %s.' % nom_tag
    raise template.TemplateSyntaxError(msg)

return RandomNode(begin, end)

```

Jusqu'ici, il n'y a qu'une suite de conditions afin de vérifier que les arguments sont bien ceux attendus. Si jamais un tag est mal formé (nombre d'argument incorrect, types des arguments invalides, etc) alors le template ne se construira pas et **et une erreur HTTP500 sera renvoyée** au client, avec comme message d'erreur ce qui est précisé dans la variable `msg`, si jamais vous êtes en mode *DEBUG*.

Il ne nous reste plus qu'à écrire la classe `RandomNode`, qui est renvoyé par la méthode ci-dessus. Vu son appel, il semble évident que sa méthode `__init__` prend 3 arguments : `self`, `begin` et `end`. Comme nous l'avons vu tout à l'heure, cette classe doit également définir une méthode `render(self, context)`, qui va renvoyer une chaîne de caractère, qui remplacera notre tag dans notre rendu HTML. Cette méthode prend en paramètre le contexte du template, auquel on peut accéder et éditer celui-ci.

Code : Python

```

from random import randint

class RandomNode(template.Node):
    def __init__(self, begin, end):
        self.begin = begin
        self.end = end

    def render(self, context):
        return str(randint(self.begin, self.end))

```

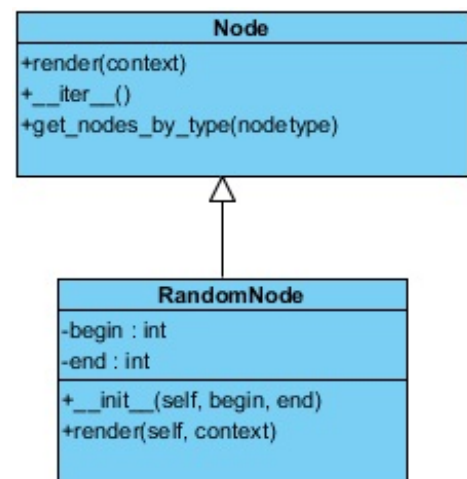


Diagramme UML de notre classe `RandomNode`

Comme pour la fonction de structuration, le code en lui-même n'est pas complexe. On se contente juste ici de se souvenir des arguments, et une fois que la fonction `render()` est appelé, on génère un nombre aléatoire. Il faut cependant pas oublier de le **transposer en chaîne de caractères**, puisque Django après fait une simple concaténation de chaque nœud !

Il ne nous reste plus qu'à enregistrer notre tag désormais ! Comme pour les filtres, il y a plusieurs méthodes :

- `@register.tag()` au début de notre fonction de compilation ;
- `@register.tag(name='nom_du_tag')` si jamais on souhaite prendre un nom différent ;
- `register.tag('nom_du_tag', random)` pour l'enregistrer après la déclaration de la fonction.

Ici, nous allons garder la première méthode, comme pour les filtres. Au final, notre tag complet ressemble à ceci :

Code : Python

```

#-*- coding:utf8 -*-
from django import template
from random import randint

```

```

register = template.Library()

@register.tag
def random(parser, token):
    """ Tag générant un nombre aléatoire, entre les bornes données
    en argument """
    # Séparation des paramètres contenu dans l'objet token
    try:
        nom_tag, begin, end = token.split_contents()
    except ValueError:
        msg = 'Le tag %s doit prendre exactement deux arguments.'
        % token.split_contents()[0]
        raise template.TemplateSyntaxError(msg)

    # On vérifie que nos deux paramètres sont bien des entiers
    try:
        begin, end = int(begin), int(end)
    except ValueError:
        msg = 'Les arguments du tag %s sont obligatoirement des
entiers.' % nom_tag
        raise template.TemplateSyntaxError(msg)

    # On vérifie si le premier est bien inférieur au second
    if begin > end:
        msg = 'L\'argument "begin" doit obligatoirement être
inférieur à l\'argument "end" dans le tag %s.' % nom_tag
        raise template.TemplateSyntaxError(msg)

    return RandomNode(begin, end)

class RandomNode(template.Node):
    def __init__(self, begin, end):
        self.begin = begin
        self.end = end

    def render(self, context):
        return str(randint(self.begin, self.end))

```



Si vous oubliez d'enregistrer votre tag et que vous tentez tout de même de l'utiliser, vous obtiendrez l'erreur suivante :
Invalid block tag: 'random'

Nous allons enfin pouvoir en profiter dans notre template ! En incorporant `{% random 1 20 %}`, vous allez afficher un nombre situé entre 1 et 20 à chaque appel de la page.

Vous pouvez d'ailleurs tester les cas incorrects cités dans la méthode de compilation. Par exemple, `{% random "a" 10 %}` affiche la page d'erreur 500 suivante :


```

TemplateSyntaxError at /blog/1
Les arguments du tag random sont obligatoirement des entiers.

Request Method: GET
Request URL: http://127.0.0.1:8000/blog/1
Django Version: 1.4
Exception Type: TemplateSyntaxError
Exception Value: Les arguments du tag random sont obligatoirement des entiers.
Exception Location: d:\Programation\crepes\blog\templatetags\blog_extras.py in random, line 70
Python Executable: c:\Python27\python.exe
Python Version: 2.7.2
Python Path: ['d:\Programation\crepes',
'c:\Python27\lib\site-packages\setuptools-0.6c11-py2.7.egg',
'c:\Python27\lib\site-packages\mechanize-0.2.5-py2.7.egg',
'c:\Python27\lib\site-packages\django_blog_sinnia-0.11.2-py2.7.egg',
'c:\Python27\lib\site-packages\pytz-2012d-py2.7.egg',
'c:\Python27\lib\site-packages\pprinting-1.5.6-py2.7.egg',
'c:\Python27\lib\site-packages\django_mailrcp-0.1.4-py2.7.egg',
'c:\Python27\lib\site-packages\django_tagging-0.3.1-py2.7.egg',
'c:\Python27\lib\site-packages\django_mptt-0.5.2-py2.7.egg',
'c:\Python27\lib\site-packages\beautifulsoup-3.2.1-py2.7.egg',
'c:\Python27\lib\site-packages\pprinting-1.5.6-py2.7.egg',
'c:\Windows\system32\python27.zip',
'c:\Python27\DLLs',
'c:\Python27\lib',
'c:\Python27\lib\site-packages\django_blog_sinnia',
'c:\Python27\lib\site-packages\pytz-2012d-py2.7.egg',
'c:\Python27\lib\site-packages\pprinting-1.5.6-py2.7.egg',
'c:\Python27\lib\site-packages\pprinting-1.5.6-py2.7.egg',
'c:\Python27\lib\site-packages\pprinting-1.5.6-py2.7.egg']
Server time: dim, 18 Nov 2012 00:01:50 +0100

```

Erreur 500

Error during template renderingIn template `S:\Programation\crepes\templates\blog\archives.html`, error at line 13

Les arguments du tag random sont obligatoirement des entiers.

lorsque le tag est mal utilisé

Passage de variable dans notre tag

Avec le tag que nous venons d'écrire, il n'est **possible que de passer des entiers en paramètres**. Il est cependant parfois pratique de pouvoir donner des variables en arguments, comme nous avons pu le faire avec `{% url %}` dans le premier TP.

Pour ce faire, il va falloir revoir un peu l'architecture de notre tag. Une variable est par définition indéterminée, il y a donc plusieurs tests que nous ne pourrions faire qu'au rendu, et non plus à la compilation du tag. Nous allons continuer sur notre tag `{% random %}`, en lui passant en paramètre 2 variables, qui seront définies dans notre vue comme ceci :

Code : Python

```

def ma_vue(request):
    begin = 1
    end = 42
    return render(request, 'template.html', locals())

```

Code : Jinja

```
{% random begin end %}
```

Nous allons devoir changer notre tag pour interpréter les variables et faire attention au cas où une des variables entrées n'existe pas dans notre contexte (qui est l'ensemble des variables passées au template depuis la vue)... Le problème, comme nous l'avons dit plus haut, c'est que ce genre d'informations n'est **disponible qu'au rendu**. Il va donc falloir bouger la plupart de nos tests au rendu. Cela pouvait paraître logique de tester nos entrées dès leur réception, mais cela devient tout simplement impossible.

Tout d'abord, supprimons les tests sur le type et la comparaison entre `begin` et `end` de la méthode de compilation, ce qui nous laisse uniquement :

Code : Python

```

@register.tag
def random(parser, token):
    """ Tag générant un nombre aléatoire, entre les bornes données
    en argument """
    # Séparation des paramètres contenu dans l'objet token
    try:
        nom_tag, begin, end = token.split_contents()
    except ValueError:

```



```

        msg = 'Le tag random doit prendre exactement deux
arguments.'
        raise template.TemplateSyntaxError(msg)

    return RandomNode(begin, end)

```

Maintenant, notre méthode `render()` dans la classe `RandomNode` va être un peu plus complexe. Nous allons devoir vérifier dedans si la variable passée en paramètre existe et si oui, vérifier s'il s'agit bien d'un entier. Pour ce faire, il existe dans le module `template` une classe `Variable` qui permet de **recupérer le contenu d'une variable à partir de son nom** dans le contexte. Si jamais on lui donne une constante, on obtiendra cette même constante en retour, ce qui nous permet de rester compatible avec notre ancien tag !

Code : Python

```

from django.template.base import VariableDoesNotExist

class RandomNode(template.Node):
    def __init__(self, begin, end):
        self.begin = begin
        self.end = end

    def render(self, context):
        not_exist = False

        try:
            begin = template.Variable(self.begin).resolve(context)
            self.begin = int(begin)
        except (VariableDoesNotExist, ValueError):
            not_exist = self.begin
        try:
            end = template.Variable(self.end).resolve(context)
            self.end = int(end)
        except (VariableDoesNotExist, ValueError):
            not_exist = self.end

        if not_exist:
            msg = 'L\'argument "%s" n\'existe pas, ou n\'est pas
un entier.' % not_exist
            raise template.TemplateSyntaxError(msg)

        # On vérifie si le premier entier est bien inférieur au
second
        if self.begin > self.end:
            msg = 'L\'argument "begin" doit obligatoirement être
inférieur à l\'argument "end" dans le tag random.'
            raise template.TemplateSyntaxError(msg)

        return str(randint(self.begin, self.end))

```

Quelques explications s'imposent.

- Notre méthode `__init__` n'a pas changé, elle ne fait que garder les paramètres passés dans des attributs de l'objet ;
- au début de `render()`, on vérifie les arguments passés. Via la classe `Template`, on récupère le contenu de la variable ou les constantes 1 et 10 si jamais on a `{% random 1 10 %}`. On renvoie une exception de base de Django, `VariableDoesNotExist`, si la variable n'existe pas ;
- en cas d'erreur, on renvoie les mêmes messages d'erreur qu'avant, comme si nous étions à la compilation ;
- enfin on vérifie toujours à la fin que la première borne est bien inférieure à la seconde, et on retourne notre nombre aléatoire.

Vous pouvez désormais tester votre tag dans n'importe quel sens :

Code : Jinja

```
{% random 0 42 %}
{% random a b %} avec a = 0 et b = 42
{% random a 42 %}
```

Mais aussi avec des cas qui ne marchent pas :

Code : Jinja

```
{% random a 42 %} avec a = "Bonjour"
{% random a 42 %} avec où a n'existe pas
```

<titel>Simple tags</titel>

Il ne nous reste plus qu'à voir comment coder des tags simples, qui prennent des arguments et dont la sortie ne dépend que de ces arguments. C'est le cas de notre tag `random` par exemple, qui renvoie un nombre en se basant que sur nos 2 paramètres. Il est alors possible de simplifier tout notre tag par :

Code : Python

```
@register.simple_tag(name='random') # L'argument name est toujours facultatif
def random(begin, end):
    try:
        return randint(int(begin), int(end))
    except ValueError:
        raise template.TemplateSyntaxError('Les arguments sont nécessairement des entiers')
```

Il est aussi possible d'accéder au contexte depuis ce genre de tags, en le précisant à son enregistrement :

Code : Python

```
@register.simple_tag(takes_context=True)
def random(context, begin, end):
    # ...
```



Pourquoi avoir fait toute cette partie si au final on peut faire un tag aussi en quelques lignes plus simples ?

D'une part, il n'est pas possible de tout faire avec des simple tags. Dès que vous avez besoin d'avoir un état interne par exemple (comme pour *cycle*), il est plus facile de passer via une classe (notre nœud) qui stockera cet état. De plus, les simple tags fonctionnent en réalité de la même façon que nos tags précédents : un objet `SimpleNode` est instancié et sa fonction `render` ne fait qu'appeler notre fonction `random()`.

Enfin, nous n'avons pas le temps ici de vous présenter tous les types de tags possibles. Il reste en effet d'autres types qui seront abordés au prochain chapitre :

- Les tags composés, par exemple `{% if %} {% endif %}` ;
- les tags incluant d'autres templates, et ayant leur propre contexte ;
- et enfin, les tags agissant sur le contexte plutôt que renvoyer une valeur.

Quelques points à ne pas négliger

Pour finir, il est important de savoir que les tags renvoient toujours du texte considéré comme sécurisé, c'est à dire que le HTML y est interprété. Il est donc important de penser à échapper le HTML quand il est nécessaire, via la fonction *escape*, tel que nous

l'avons vu avec les filtres.

De plus, les développeurs de Django recommandent de rester vigilants lorsqu'on souhaite garder un état interne avec les tags. En effet, certains environnements fonctionnent de façon multi-threadé, et donc un même nœud peut être exécuté à deux endroits différents, dans deux contextes différents dans un ordre indéterminé. Ainsi son état interne est partagé entre les deux contextes et le résultat peut être inattendu.

Dans ce cas, il est conseillé de garder un état interne dans le contexte, via le paramètre disponible dans la fonction *render*, afin de savoir où en était l'exécution pour ce lieu, et non pour l'ensemble du template.

Ce point est assez complexe, pour plus d'informations à ce sujet, consulter la [la documentation officielle](#).

Le chapitre s'arrête ici mais notre apprentissage des tags n'est pas fini ! Comme nous l'avons fait remarquer dans la dernière sous-partie, ils nous reste plusieurs cas de tags à voir. Cet approfondissement continue donc dans le chapitre suivant, suite à quoi vous pourrez créer n'importe quel template !

Les signaux et middlewares

Django délimite proprement et nettement ses différentes composantes. On ne peut pas se charger du routage des URLs depuis un template, et on ne peut pas créer des modèles dans les vues. Si cette structuration a bien évidemment des avantages (propreté du code, réutilisation, etc), sa lourdeur peut parfois empêcher de réaliser certaines actions.

En effet, comment effectuer une action précise à chaque fois qu'une entrée d'un modèle est supprimée, et ce depuis n'importe où dans le code ? Ou comment analyser toutes les requêtes d'un visiteur pour s'assurer que son adresse IP n'est pas bannie ? Pour ces situations un peu spéciales qui nécessitent de répéter la même action à plusieurs moments et endroits dans le code, Django intègre 2 mécanismes différents qui permettent de résoudre ce genre de problèmes : les signaux et les middlewares.

Notifiez avec les signaux

Premier mécanisme : les signaux. Un signal est une notification envoyée par une application à Django lorsqu'une action se déroule, et renvoyée par le framework à toutes les autres parties d'applications qui se sont enregistrées pour savoir quand ce type d'action se déroule, et comment.

Reprenons l'exemple de la suppression d'un modèle : imaginons que nous souhaitons supprimer plusieurs fichiers sur le disque dur associés à une entrée d'un modèle lorsque cette entrée est supprimée. Cependant, cette entrée peut être supprimée depuis n'importe où dans le code, et vous ne pouvez pas à chaque fois rajouter un appel vers une fonction qui se charge de la suppression des fichiers associés (parce que ça serait trop lourd ou que ça ne dépend simplement pas de vous). Les signaux sont la solution parfaite.

Pour résoudre ce problème, une fois que vous avez écrit la fonction de suppression des fichiers associés, vous n'avez qu'à indiquer à Django d'appeler cette fonction à chaque fois qu'une entrée de modèle est supprimée. En pratique, cela se fait ainsi :

Code : Python

```
from django.models.signals import post_delete

post_delete.connect(ma_fonction_de_suppression, sender=MonModele)
```

La méthode est plutôt simple : il suffit d'importer le signal et d'utiliser la méthode connect pour connecter une fonction à ce signal. Le signal ici importé est `post_delete`, et comme son nom l'indique, il est notifié à chaque fois qu'une instance a été supprimée. À chaque fois que Django recevra le signal, il le transmettra en appelant la fonction passée en argument (`ma_fonction_de_suppression` ici). Cette méthode peut prendre plusieurs paramètres, comme par exemple ici `sender`, qui permet de restreindre l'envoi de signaux à un seul modèle (`MonModele` donc), sans quoi la fonction sera appelée pour toute entrée supprimée, et quelque soit le modèle dont elle dépend.

Une fonction appelée par un signal prend souvent plusieurs arguments. Généralement, elle prend presque toujours un argument appelé `sender`. Son contenu dépend du type de signal en lui-même (par exemple, pour `post_delete`, la variable `sender` passée en argument sera toujours le modèle concerné, comme vu précédemment). Chaque type de signal possède ses propres arguments. `post_delete` en prend 3 :

- `sender` : le modèle concerné, comme vu précédemment ;
- `instance` : l'instance du modèle supprimée (celle-ci étant supprimée, il est très déconseillé de modifier ses données ou tenter de la sauvegarder) ;
- `using` : l'alias de la base de donnée utilisée (si vous utilisez plusieurs base de données, il s'agit d'un point particulier et inutile la plupart du temps).

Notre fonction `ma_fonction_de_suppression` pourrait donc s'écrire de la sorte :

Code : Python

```
def ma_fonction_de_suppression(sender, instance, **kwargs):
    #processus de suppression selon les données fournies par instance
```



Pourquoi spécifier un `**kwargs` ?

Vous ne pouvez jamais être certains qu'un signal renverra bien tous les arguments possibles, cela dépend du contexte. Dès lors, il est toujours important de spécifier un dictionnaire pour récupérer les valeurs supplémentaires, et si vous avez éventuellement besoin d'une de ces valeurs, il suffit de vérifier si la clé est bien présente ou non dans le dictionnaire.



Où doit-on mettre l'enregistrement des signaux ?

On peut mettre l'enregistrement n'importe où, tant que Django charge le fichier afin qu'il puisse faire la connexion directement. Le framework charge déjà par défaut certains fichiers comme les *models.py*, *urls.py*, etc. Le meilleur endroit serait donc un de ces fichiers. Généralement, on choisit un *models.py* (étant donné que certains signaux agissent à partir d'actions sur des modèles, c'est un plutôt bon choix !).

Petit détail, il est également possible d'enregistrer une fonction à un signal directement lors de sa déclaration avec un décorateur. En reprenant l'exemple ci-dessus :

Code : Python

```
from django.models.signals import post_delete
from django.dispatch import receiver

@receiver(post_delete, sender=MonModele)
def ma_fonction_de_suppression(sender, instance, **kwargs):
    #processus de suppression selon les données fournies par instance
```

Il existe bien entendu d'autres types de signaux, en voici une liste non exhaustive en contenant les principaux, avec les arguments qu'elle transmet avec la notification :

- `django.db.models.signals.pre_save` : envoyé avant qu'une instance de modèle ne soit enregistrée. Arguments : `sender` (le modèle concerné), `instance` (instance du modèle concernée), `using` (alias de la BDD utilisée), `raw` (booléen, mis à `True` si l'instance sera enregistrée telle qu'elle est présentée depuis l'argument)
- `django.db.models.signals.post_save` : envoyé après qu'une instance de modèle ne soit enregistrée. Arguments : `sender` (le modèle concerné), `instance` (instance du modèle concernée), `using` (alias de la BDD utilisée), `raw` (booléen, mis à `True` si l'instance sera enregistrée telle qu'elle est présentée depuis l'argument), `created` (booléen, mis à `True` si l'instance a été correctement enregistrée)
- `django.db.models.signals.pre_delete` : envoyé avant qu'une instance de modèle ne soit supprimée. Arguments : `sender` (le modèle concerné), `instance` (instance du modèle concernée), `using` (alias de la BDD utilisée)
- `django.db.models.signals.post_delete` : envoyé après qu'une instance de modèle ne soit supprimée. Arguments : `sender` (le modèle concerné), `instance` (instance du modèle concernée), `using` (alias de la BDD utilisée)
- `django.core.signals.request_started` : envoyé à chaque fois que Django reçoit une nouvelle requête HTTP. Arguments : `sender` (la classe qui a envoyé la requête, par exemple `django.core.handlers.wsgi.WsgiHandler`)
- `django.core.signals.request_finished` : envoyé à chaque fois que Django termine de répondre à une nouvelle requête HTTP. Arguments : `sender` (la classe qui a envoyé la requête, par exemple `django.core.handlers.wsgi.WsgiHandler`)

Il existe d'autres signaux inclus par défaut. Ils sont expliqués dans la documentation officielle :

<https://docs.djangoproject.com/en/1.4/ref/signals/>

Sachez que vous pouvez tester tous ces signaux simplement en créant une fonction affichant une ligne dans la console (avec `print`) et en liant cette fonction aux signaux désirés.

Heureusement, si vous vous sentez limités par la (maigre) liste de types de signaux fournis par Django, sachez que vous pouvez en créer vous-mêmes. Le processus est plutôt simple.

Chaque signal est en fait une instance de `django.dispatch.Signal`. Pour créer un nouveau signal, il suffit donc de créer une nouvelle instance, de et lui dire quels arguments le signal peut transmettre :

Code : Python

```
import django.dispatch

crepe_finie = django.dispatch.Signal(providing_args=["adresse",
"prix"])
```

Ici, on crée un nouveau signal nommé `crepe_finie`. On lui indique une liste contenant le nom d'éventuels arguments (les arguments de signaux n'étant jamais fixes, vous pouvez la modifier à tout moment) qu'il peut transmettre, et c'est tout !

On pourrait enregistrer une fonction sur ce signal comme vu précédemment :

Code : Python

```
crepe_finie.connect(faire_livraison)
```

Lorsqu'on souhaite lancer une notification à toutes les fonctions enregistrées au signal, il suffit simplement d'utiliser la méthode `send`, et ceci, depuis n'importe où. Nous l'avons fait depuis un modèle :

Code : Python

```
class Crepe(models.Model):
    nom_recette = models.CharField(max_length=255)
    prix = models.IntegerField()
    #d'autres attributs

    def preparer(self, adresse):
        #On prépare la crêpe pour l'expédier à l'adresse transmise
        crepe_finie.send(sender=self, adresse=adresse, prix=self.prix)
```

À chaque fois qu'on appellera la méthode `preparer` d'une crêpe, la fonction `faire_livraison` sera appelée avec les arguments adéquats.

Notons ici qu'il est toujours obligatoire de préciser un argument `sender` lorsqu'on utilise la méthode `send`. Libre à vous de choisir ce que vous souhaitez transmettre, mais il est censé représenter l'entité qui est à l'origine du signal. Nous avons ici choisi d'envoyer directement l'instance du modèle.

Aussi, la fonction `send` retourne une liste de paires de variables, chaque paire est un tuple de type `(receveur, retour)`, où le receveur est la fonction appelée, et le retour est la variable retournée par la fonction.

Par exemple, si nous n'avons que la fonction `faire_livraison` connectée au signal `crepe_finie`, et que celle-ci retourne `True` si la livraison s'est bien déroulée (considérons que c'est le cas maintenant), la liste renvoyée par `send` serait `[(faire_livraison, True)]`.

Pour terminer, il est également possible de déconnecter une fonction d'un signal. Il faut utiliser la méthode `disconnect` du signal, et cette dernière s'utilise comme `connect` :

Code : Python

```
crepe_finie.disconnect(faire_livraison)
```

`crepe_finie` n'appellera plus `faire_livraison` si une notification est envoyée. Sachez que si vous avez soumis un argument `sender` lors de la connexion, vous devez également le préciser lors de la déconnexion.

Contrôlez tout avec les middlewares

Deuxième mécanisme : les *middlewares*. Nous avons vu précédemment que lorsque Django recevait une requête HTTP, il analysait l'URL demandée et en fonction de celle-ci, choisissait la vue adaptée, et cette dernière se chargeait de renvoyer une

réponse au client (en utilisant éventuellement un template). Nous avons cependant omis une étape, qui se situe juste avant l'appel de la vue.

En effet, le framework va à ce moment exécuter certains bouts de code qu'on appelle des *middlewares*. Il s'agit en quelque sorte de fonctions qui seront exécutées à chaque requête. Il est possible d'appeler ces fonctions à différents moments du processus que nous verrons plus tard.

Typiquement, les middlewares se chargent de modifier certaines variables ou d'interrompre le processus de traitement de la requête, et ceci aux différents moments que nous avons listé ci-dessus.

Par défaut, Django inclut plusieurs middlewares intégrés au framework :

Code : Python

```
MIDDLEWARE_CLASSES = (  
    'django.middleware.common.CommonMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
)
```

Cette variable est en fait une variable reprise automatiquement dans la configuration si elle n'est pas déjà présente. Vous pouvez la réécrire en la définissant dans votre *settings.py*. Il est tout de même conseillé de garder les middlewares par défaut. Ils s'occupent de certaines tâches pratiques et permettent d'utiliser d'autres fonctionnalités du framework que nous verrons plus tard ou avons déjà vues (comme la sécurisation des formulaires contre les attaques CSRF, le système utilisateur, l'envoi de notifications aux visiteurs, etc).

La création d'un middleware est très simple et permet de réaliser de choses très puissantes. Un middleware est en réalité une simple classe qui peut posséder certaines méthodes. Chaque méthode sera appelée à un certain moment du processus de traitement de la requête. Voici les différentes méthodes implémentables, avec leurs arguments :

- `process_request(self, request)` : À l'arrivée d'une requête HTTP, avant de la router vers une vue précise. `request` est un objet `HttpRequest` (le même que celui passé à une vue).
- `process_view(self, request, view_func, view_args, view_kwargs)` : Juste avant d'appeler la vue. `view_func` est une référence vers la fonction prête à être appelée par le framework. `view_args` et `view_kwargs` sont les arguments prêts à être appelés avec la vue.
- `process_template_response(self, request, response)` : Lorsqu'on retourne un objet `TemplateResponse` d'une vue. `response` est un objet `HttpResponse` (celui retourné par la vue appelée).
- `process_response(self, request, response)` : Juste avant qu'on renvoie la réponse.
- `process_exception(self, request, exception)` : Juste avant qu'on renvoie une exception si une erreur s'est produite. `exception` est un objet de type `Exception`.

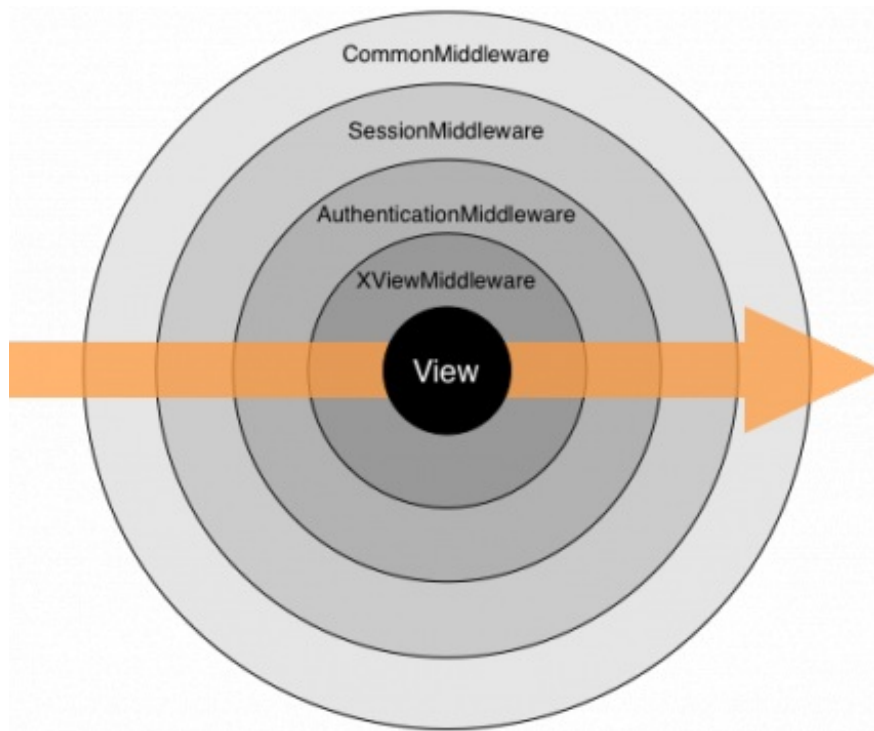
Toutes ces fonctions ne peuvent rien retourner (`None`), ou retourner un `HttpResponse`. En cas de retour vide, Django va continuer le processus normalement (appeler les autres middleware, lancer la vue, renvoyer la réponse, etc).

En revanche, si une valeur est renvoyée, cela doit être impérativement un objet `HttpResponse`. Le processus de traitement de la requête sera arrêté net (plus rien ne sera appelé, même pas un middleware), et l'objet `HttpResponse` retourné sera directement envoyé au client.

Ceci vaut pour toutes les méthodes suscitées, à l'exception de `process_response` qui doit obligatoirement renvoyer un objet `HttpResponse` ! Vous pouvez toujours renvoyer celui passé en argument si la réponse n'a pas besoin d'être modifiée.

Sachez également que vous pouvez altérer par exemple une requête, juste en modifiant des attributs de `request` dans `process_request`. L'instance modifiée de `HttpRequest` sera alors envoyée à la vue.

Dernier point avant de passer à la pratique : les middleware sont appelés dans l'ordre précisé dans le *setting.py*, de haut en bas, pour toutes les méthodes appelées avant l'appel de la vue (soit `process_request` et `process_view`). Après, les middlewares sont appelés dans le sens inverse, de bas en haut. Au final, les middlewares forment en quelque sorte des « couches » autour de la vue, comme un oignon :



Passons à la création de notre propre middleware. Comme exemple, nous avons choisi de coder un petit middleware simple mais pratique qui comptabilise le nombre de fois qu'une page est vue et affiche ce nombre à la fin de chaque page. Bien évidemment, vu le principe des middleware, il n'est nullement nécessaire d'aller modifier une vue pour arriver à nos fins, et cela marche pour toutes nos vues !

Pour ce faire, et pour des raisons de propreté et de structuration du code, le middleware sera placé dans une nouvelle application nommée « stats. »

Pour rappel, pour créer une application, rien de plus simple :

Code : Console

```
python manage.py startapp stats
```

Une fois ceci fait, la prochaine étape consiste à créer un nouveau modèle dans l'application qui permet de tenir compte du nombre de visites d'une page. Chaque entrée du modèle correspondra à une page.

Rien de spécial en définitive :

Code : Python

```
from django.db import models

class Page(models.Model):
    url = models.URLField()
    nb_visites = models.IntegerField(default=1)

    def __unicode__(self):
        return self.url
```

Il suffit dès lors d'ajouter l'application au *settings.py* et de lancer un *manage.py syncdb*. Voici notre middleware, que nous avons enregistré dans *stats/middleware.py* :

Code : Python


```

#-*- coding: utf-8 -*-
from models import Page #On importe le modèle défini précédemment

class StatsMiddleware(object):
    def process_view(self, request, view_func, view_args,
view_kwargs): #À chaque appel de vue
        try:
            p = Page.objects.get(url=request.path) #On récupère le
compteur lié à la page
            p.nb_visites += 1
            p.save()
        except Page.DoesNotExist: #Si la page n'a pas encore été
consultée
            Page(url=request.path).save() #on crée un nouveau
compteur à 1 par défaut

    def process_response(self, request, response): #À chaque réponse
        if response.status_code == 200:
            p = Page.objects.get(url=request.path)
            response.content += "Cette page a ?t? vue {0}
fois.".format(p.nb_visites)
        return response

```

N'oubliez pas de mettre à jour `MIDDLEWARE_CLASSES` dans votre `settings.py`. Chez nous, il ressemble finalement à ceci :

Code : Python

```

MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'stats.middleware.StatsMiddleware',
)

```

Le fonctionnement est plutôt simple. Avant chaque appel de vue, Django appelle la méthode `process_view` qui se chargera de déterminer si l'URL de la page a déjà été appelée ou non (l'URL est accessible à partir de l'attribut `request.path`, n'hésitez pas à consulter la documentation pour connaître toutes les méthodes et attributs de `HttpRequest`). Si la page a déjà été appelée, il incrémente le compteur de l'entrée. Sinon, il crée une nouvelle entrée.

Au retour, on vérifie tout d'abord si la requête s'est bien déroulée en vérifiant que le code HTTP de la réponse est bien 200 (ce code signifie que tout s'est bien déroulé), ensuite on reprend le compteur et on modifie le contenu de la réponse (inclut dans `response.content`, la documentation vous donnera également tout ce qu'il faut savoir sur `HttpResponse`). Bien évidemment, si vous renvoyez du HTML au client, la phrase rajoutée ne sera pas intégrée correctement au document, néanmoins vous pouvez très bien coder quelque chose de plus sophistiqué qui permet d'insérer la phrase à un endroit valide.

Au final, sur toutes vos pages, vous verrez la phrase avec le nombre de visites qui se rajoute toute seule, sans devoir modifier toutes les vues une à une !

Partie 4 : Des outils supplémentaires

Les utilisateurs

S'il y a bien une librairie très puissante et couramment utilisée que Django propose, il s'agit forcément des utilisateurs. Le framework propose en effet une solution complète pour gérer le cas classique d'un accès membre, très flexible et pourtant indéniablement pratique. Nous expliquerons l'essentiel de cette librairie dans ce chapitre.

Commençons par la base

Avant tout, il est nécessaire de s'assurer que vous avez bien ajouté l'application qui gère les utilisateurs, et ses dépendances. Elles sont ajoutées par défaut, néanmoins, vérifiez toujours que `'django.contrib.auth'` et `'django.contrib.contenttypes'` sont bien présents dans la variable `INSTALLED_APPS` de votre `settings.py`.

Ceci fait, nous pouvons commencer !

L'utilisateur

Tout le système utilisateur tourne autour du modèle `django.contrib.auth.models.User`. Celui-ci contient toutes les informations concernant vos utilisateurs, et quelques méthodes supplémentaires bien pratiques pour pouvoir les administrer. Voici les principaux attributs de `User` :

- `username` : nom d'utilisateur, 30 caractères maximum (lettres, chiffres et les caractères spéciaux `_`, `@`, `+`, `.` et `-`) ;
- `first_name` : prénom, optionnel, 30 caractères maximum ;
- `last_name` : nom de famille, optionnel, 30 caractères maximum ;
- `email` : adresse mail ;
- `password` : un *hash* du mot de passe. Django n'enregistre pas les mots de passe en clair dans la base de données, nous y reviendrons plus tard ;
- `is_staff` : booléen, permet d'indiquer si l'utilisateur a accès à l'administration de Django ;
- `is_active` : booléen, par défaut mis à `True`, si mis à `False`, au lieu de supprimer un utilisateur, il est conseillé de le désactiver afin de ne pas devoir supprimer d'éventuels modèles liés à l'utilisateur (avec une `ForeignKey` par exemple) ;
- `is_superuser` : booléen, si mis à `True`, l'utilisateur obtient toutes les permissions (nous y reviendrons plus tard également) ;
- `last_login` : `datetime` représentant la date/heure pendant laquelle l'utilisateur s'est connecté la dernière fois ;
- `date_joined` : `datetime` représentant la date/heure pendant laquelle l'utilisateur s'est inscrit.
- `user_permissions` : une relation `ManyToMany` vers les permissions (expliqué plus tard)
- `groups` : une relation `ManyToMany` vers les groupes (expliqué plus tard)

Vous ne vous servirez pas nécessairement de tous ces attributs, mais ils peuvent rester pratiques au cas où. La première question qui devrait vous sauter à l'esprit est la suivante : « est-il possible de rajouter des attributs ? La liste est plutôt limitée. ». La réponse est bien évidemment oui. Néanmoins, avant de s'attaquer à cette partie, jouons plutôt avec les bases !

La façon la plus simple de créer un utilisateur est d'utiliser la fonction `create_user` fournie avec le modèle. Elle prend 3 arguments : le nom de l'utilisateur, son adresse mail et son mot de passe (les 3 attributs obligatoires du modèle), et enregistre directement l'utilisateur dans la base de données :

Code : Python

```
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user('Maxime', 'maxime@crepes-
bretonnes.com', 'm0nsup3rm0td3p4ss3')
>>> user.id
2
```

Nous avons donc ici un nouvel utilisateur nommé « Maxime », avec l'adresse mail `maxime@crepes-bretonnes.com` et comme mot de passe « `m0nsup3rm0td3p4ss3` ». Son ID dans la base de données (preuve que l'entrée a bien été sauvegardée) est 2. Bien entendu, nous pouvons désormais modifier les autres champs :

Code : Python

```
>>> user.first_name, user.last_name = "Maxime", "Lorant"
>>> user.is_staff = True
>>> user.save()
```

Et les modifications sont enregistrées. Tous les champs sont éditables classiquement, sauf un : `password`, qui a ses propres fonctions.

Les mots de passe

En effet, les mots de passe sont quelque peu spéciaux. Il ne faut jamais enregistrer les mots de passe tels quels dans la base de données (en clair). Si un jour, quelqu'un accède à votre base de données, il aurait accès à tous les mots de passe de vos utilisateurs, ce qui serait plutôt embêtant du point de vue de la sécurité.

Pour éviter cela, on donne le mot de passe à une fonction de hachage qui va le transformer en une autre chaîne de caractères, ce qu'on appelle une empreinte ou un hash. Cette fonction est à sens unique : le même mot de passe donnera toujours la même empreinte, en revanche, on ne peut pas obtenir le mot de passe uniquement à partir de l'empreinte. En utilisant cette méthode, même si quelqu'un accède aux mots de passe enregistrés dans la BDD, il ne pourrait rien en faire. Chaque fois qu'un utilisateur voudra se connecter, il suffit d'appliquer la même fonction de hachage au mot de passe fourni lors de la connexion, et vérifier si celui-ci correspond bien à celui enregistré dans la base de données.

Quelle est la bonne nouvelle dans tout ça ? Django le fait automatiquement !

En effet, tout à l'heure nous avons renseigné le mot de passe « `m0n5up3rm0td3p4ss3` » pour notre utilisateur. Regardons ce qui a réellement été enregistré :

Code : Python

```
>>> user.password
'pbkdf2_sha256$10000$cRu9mKvGzMzW$DuQc3ZJ3cjT37g0TkiEYrfDRRj57LjuceDyapH/qjvQ='
```

Le résultat est plutôt inattendu. Tous les mots de passe sont enregistrés selon cette disposition : *algorithme\$sel\$empreinte*.

- **algorithme** : Le nom de l'algorithme, de la fonction de hachage utilisé pour le mot de passe (ici `pbkdf2_sha256`, la fonction de hachage par défaut de Django 1.4) ;
- **sel** : le sel est une chaîne de caractères insérée dans le mot de passe originel pour rendre son déchiffrement plus difficile (ici 10000). Django s'en charge, donc inutile de s'y attarder ;
- **empreinte** : l'empreinte finale, résultat de la combinaison du mot de passe originel et du sel par la fonction de hachage. Elle représente la majeure partie de `user.password`.

Maintenant que vous savez que le champ `password` ne doit pas s'utiliser comme un champ classique, comment l'utiliser ? Django fournit 3 méthodes au modèle `User` pour la gestion des mots de passe :

- `set_password(mot_de_passe)` : permet de modifier le mot de passe de l'utilisateur par celui donné en argument. Django va hacher ce dernier puis l'enregistrer dans la base de données, comme vu précédemment. Ne sauvegarde pas l'entrée dans la base de données, il faut faire un `.save()` après.
- `check_password(mot_de_passe)` : vérifie si le mot de passe donné en argument correspond bien à l'empreinte enregistrée dans la base de données. Retourne `True` si les deux mots de passe correspondent, sinon `False`.
- `set_unusable_password()` : permet d'indiquer que l'utilisateur n'a pas de mot de passe défini. Dans ce cas, `check_password` retournera toujours `False`.
- `has_usable_password()` : retourne `True` si le compte utilisateur a un mot de passe valide, `False` si `set_unusable_password` a été utilisé.

Petit exemple pratique désormais, en reprenant notre utilisateur de tout à l'heure :

Code : Python

```
>>> user = User.objects.get(username="Maxime")
```

```
>>> user.set_password("coucou") # On change le mot de passe
>>> user.check_password("salut") # On essaie un mot de passe
    invalide
    False
>>> user.check_password("coucou") # Avec le bon mot de passe, ça
    marche !
    True
>>> user.set_unusable_password() # On désactive le mot de passe
>>> user.check_password("coucou") # Comme prévu, le mot de passe
    précédent n'est plus bon
    False
```

Étendre le modèle User

Pour terminer ce sous-chapitre, abordons l'extension du modèle `User`. Nous avons vu plus tôt que les champs de `User` étaient assez limités, mais heureusement, il est possible d'en rajouter.

L'extension du modèle `User` se fait grâce à ce qu'on appelle des profils. Un profil est typiquement un modèle reprenant tous les champs que vous souhaitez ajouter à votre modèle utilisateur. Une fois ce modèle spécifié, il faudra le lier au modèle `User` en rajoutant un `OneToOneField` vers ce dernier.

Imaginons que nous souhaitons donner la possibilité à un utilisateur d'avoir un avatar, une signature pour ses messages, un lien vers son site web et de pouvoir s'inscrire à la newsletter de notre site. Notre profil ressemblerait à ceci, dans *blog/models.py* :

Code : Python

```
from django.contrib.auth.models import User
class Profil(models.Model):
    user = models.OneToOneField(User) # La liaison OneToOne vers
    le modèle User
    site_web = models.URLField(null=True, blank=True)
    avatar = models.ImageField(null=True, blank=True,
    upload_to="avatars/")
    signature = models.TextField(null=True, blank=True)
    inscrit_newsletter = models.BooleanField(default=False)

    def __unicode__(self):
        return "Profil de {}".format(self.user.username)
```

Les différents attributs que nous avons listé ci-dessus sont bel et bien repris dans notre modèle `Profil`, dont notamment la liaison vers le modèle `User`.

La prochaine étape est d'indiquer à l'application authentication d'utiliser notre classe `Profil`. Pour ce faire, il suffit de rajouter une petite variable dans notre `settings.py` :

Code : Python

```
AUTH_PROFILE_MODULE = "blog.Profil"
```

La variable représente l'application dans laquelle est située le profil (ici à savoir *blog*), et le nom du modèle concerné dans `models.py` (ici `Profil`, que nous avons détaillé ci-dessus).

Grâce à cette variable, la méthode `get_profile` d'`User` devient accessible. Elle permet de récupérer l'entrée du profil de l'utilisateur à partir de l'instance `User` en elle-même.

Pour illustrer le fonctionnement des profils, voici un petit exemple :

Code : Python

```
>>> from django.contrib.auth.models import User
>>> from blog.models import Profil
>>> user = User.objects.create_user('Mathieu', 'mathieu@crepes-
bretonnes.com', '_2b!84%sdb') # On crée un nouvel utilisateur
>>> profil = Profil(user=user, site_web="http://www.crepes-
bretonnes.com", signature="Coucou ! C'est moi !")
#Le profil qui va avec
>>> profil.save() # On l'enregistre
>>> profil
<Profil: Profil de Mathieu>
>>> user.get_profile()
<Profil: Profil de Mathieu>
>>> user.get_profile().signature
u"Coucou ! C'est moi !"
```

Voilà ! Le modèle User est désormais correctement étendu avec les nouveaux attributs et méthodes du profil.

Passons aux vues

Maintenant que nous avons assimilé les bases, il est temps de passer aux vues.

Nous n'aborderons pas ici l'enregistrement de nouveaux utilisateurs. En effet, nous avons montré plus haut la fonction à utiliser, et le reste est tout à fait classique : un formulaire, une vue pour récupérer et enregistrer les informations, un template, ...

La connexion

Nous avons désormais des utilisateurs, ils n'ont plus qu'à se connecter ! Pour ce faire, nous aurons besoin des éléments suivants :

- Un formulaire pour récupérer le nom d'utilisateur et le mot de passe ;
- Un template pour afficher ce formulaire ;
- Une vue pour récupérer les données, les vérifier, et connecter l'utilisateur.

Commençons par le formulaire. Il ne nous faut que deux choses : le nom d'utilisateur et le mot de passe. Autrement dit, le formulaire est très simple. Nous le plaçons dans `blog/forms.py` :

Code : Python

```
class ConnexionForm(forms.Form):
    username = forms.CharField(label="Nom d'utilisateur",
max_length=30)
    password = forms.CharField(label="Mot de passe",
widget=forms.PasswordInput)
```

Rien de compliqué, si ce n'est le widget utilisé : `forms.PasswordInput` permet d'avoir une boîte de saisie dont les caractères seront masqués, afin d'éviter que le mot de passe ne soit affiché en clair lors de sa saisie.

Passons au template :

Code : Jinja

```
<h1>Se connecter</h1>

{% if error %}
<p><strong>Utilisateur inexistant ou mauvais de mot de
passe.</strong></p>
{% endif %}

{% if user.is_authenticated %}
Vous êtes connecté, {{ user.username }} !
{% else %}
```

```
<form method="post" action=".">
{% csrf_token %}
{{ form.as_p }}
<input type="submit"/>
</form>
{% endif %}
```

La nouveauté ici est la variable `user`, qui contient l'instance `User` de l'utilisateur s'il est connecté, ou une instance de la classe `AnonymousUser`. La classe `AnonymousUser` est utilisée pour indiquer le visiteur n'est pas un utilisateur connecté. `User` et `AnonymousUser` partagent certaines méthodes comme `is_authenticated`, qui permet de définir si le visiteur est connecté ou non. Une instance `User` retournera toujours `True`, tandis qu'une instance `AnonymousUser` retournera toujours `False`.

La variable `user` dans les templates est rajoutée par un processeur de contexte inclus par défaut.

Notez l'affichage du message d'erreur si la combinaison utilisateur / mot de passe est incorrecte.

Pour terminer, passons à la partie intéressante : la vue. Récapitulons avant tout ce qu'elle doit faire :

1. Afficher le formulaire
2. Après la saisie de l'utilisateur, récupérer les données
3. Vérifier si les données entrées correspondent bien à un utilisateur
4. Si c'est le cas, le connecter et le rediriger vers une autre page
5. Sinon, afficher un message d'erreur

Vous savez d'ores et déjà comment réaliser les étapes 1, 2 et 5. Reste à savoir comment vérifier si les données sont correctes, et si c'est le cas, de connecter l'utilisateur. Pour cela, Django fournit deux fonctions, `authenticate` et `login`, toutes deux situées dans le module `django.contrib.auth`. Voici comment elles fonctionnent :

- `authenticate(username=nom, password=mdp)` : si la combinaison utilisateur / mot de passe est correcte, `authenticate` renvoie l'entrée du modèle `User` correspondante. Si ce n'est pas le cas, la fonction renvoie `None`.
- `login(request, user)` : permet de connecter l'utilisateur. La fonction prend l'objet `HttpRequest` passé à la vue par le framework, et l'instance d'`User` de l'utilisateur à connecter.



Attention ! Avant d'utiliser `login` avec un utilisateur, vous devez avant tout avoir utilisé `authenticate` avec le nom d'utilisateur et mot de passe correspondants, sans quoi `login` n'acceptera pas la connexion. Il s'agit d'une mesure de sécurité.

Désormais, nous avons tout ce qu'il nous faut pour coder notre vue. Voici notre exemple :

Code : Python

```
def connexion(request):
    error = False

    if request.method == "POST":
        form = ConnexionForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data["username"] # On récupère
nom d'utilisateur
            password = form.cleaned_data["password"] # ... et mot
de passe
            user = authenticate(username=username,
password=password) # On vérifie si les données sont correctes
            if user: #Si l'objet renvoyé n'est pas None
                login(request, user) #on connecte l'utilisateur
            else: #sinon on affiche une erreur
                error = True
        else:
            form = ConnexionForm()
```

```
return render(request, 'connexion.html', locals())
```

Et finalement la directive de routage dans *crepes_bretonnes/urls.py* :

Code : Python

```
url(r'^connexion/$', 'blog.views.connexion', name='connexion'),
```

Vous pouvez désormais essayer de vous connecter depuis l'adresse `/connexion/`. Vous devrez soit créer un compte manuellement dans la console si cela n'a pas été fait auparavant, ou renseigner le nom d'utilisateur et le mot de passe du compte super-utilisateur que vous avez créé lors de votre tout premier *syncdb*.

Si vous entrez une mauvaise combinaison, un message d'erreur sera affiché, sinon, vous serez connecté !

La déconnexion

Heureusement, la déconnexion est beaucoup plus simple que la connexion. En effet, il suffit de d'appeler la fonction `logout` de `django.contrib.auth`. Il n'y a même pas besoin de vérifier si le visiteur est bel et bien connecté ou non (mais libre à vous de le faire si vous souhaitez rajouter un message d'erreur si ce n'est pas le cas par exemple).

Code : Python

```
from django.contrib.auth import logout
from django.shortcuts import render
from django.core.urlresolvers import reverse

def deconnexion(request):
    logout(request)
    return redirect(reverse('connexion'))
```

avec comme routage :

Code : Python

```
url(r'^deconnexion/$', 'blog.views.deconnexion',
    name='deconnexion'),
```

C'est aussi simple que cela !

En général

Comme nos utilisateurs peuvent désormais se connecter et se déconnecter, il ne reste plus qu'à pouvoir interagir avec eux. Nous avons vu précédemment que un processeur de contexte se chargeait de rajouter une variable reprenant l'instance `User` de l'utilisateur dans les templates. Il en va de même pour les vues.

En effet, l'objet `HttpRequest` passé à la vue contient également un attribut « `user` » qui renvoie vers l'objet utilisateur du visiteur. Celui-ci peut, encore une fois, être une instance `User` s'il est connecté, ou `AnonymousUser` si ce n'est pas le cas. Exemple dans une vue :

Code : Python

```
from django.http import HttpResponse
```

```
def dire_bonjour(request):  
    if request.user.is_authenticated():  
        return HttpResponse("Salut, {0}  
!".format(request.user.username))  
    return HttpResponse("Salut, anonyme.")
```

Maintenant, imaginons que nous souhaitons restreindre l'accès de certaines vues aux utilisateurs connectés seulement. On pourrait vérifier si l'utilisateur est connecté, et si ce n'est pas le cas, le rediriger vers une autre page, mais cela serait lourd à force. Pour éviter de se répéter, Django fournit un petit décorateur très pratique qui nous permet de s'assurer qu'uniquement des visiteurs authentifiés accèdent à la vue. Son nom est `login_required` et il se situe dans `django.contrib.auth.decorators`.

Code : Python

```
from django.contrib.auth.decorators import login_required  
  
@login_required  
def ma_vue(request):  
    ...
```

Si l'utilisateur n'est pas connecté, il sera redirigé vers l'URL de la vue de connexion. Cette URL est normalement définie depuis la variable `LOGIN_URL` dans votre `settings.py`. Si ce n'est pas fait, la valeur par défaut est `'/accounts/login/'`. Comme nous avons utilisé l'URL `'/connexion/'` tout à l'heure, réindiquons-la ici :

Code : Python

```
LOGIN_URL = '/connexion/'
```

Il faut savoir que si l'utilisateur n'est pas connecté, non seulement il sera donc redirigé vers `'/connexion/'`, mais l'URL complète de la redirection sera `"/connexion/?next=/bonjour/"`. En effet, Django rajoute un paramètre GET, appelé « next », qui contient l'URL d'où provient la redirection. Si vous le souhaitez, vous pouvez récupérer ce paramètre dans la vue gérant la connexion, et ensuite rediriger l'utilisateur vers l'URL fournie. Néanmoins, ce n'est pas obligatoire.

Sachez que vous pouvez également préciser le nom de ce paramètre (au lieu de « next » par défaut), via l'argument `redirect_field_name` du décorateur :

Code : Python

```
from django.contrib.auth.decorators import login_required  
  
@login_required(redirect_field_name='rediriger_vers')  
def ma_vue(request):  
    ...
```

Vous pouvez également spécifier une autre URL de redirection pour la connexion (au lieu de prendre `LOGIN_URL` dans le `settings.py`) :

Code : Python

```
from django.contrib.auth.decorators import login_required  
  
@login_required(login_url='/utilisateurs/connexion/')  
def my_view(request):  
    ...
```


Les vues génériques

L'application `django.contrib.auth` contient certaines vues génériques très puissantes et pratiques qui permettent de réaliser les tâches communes d'un système utilisateur sans devoir écrire une seule vue : se connecter, se déconnecter, changer le mot de passe et récupérer un mot de passe perdu.



Pourquoi alors nous avoir expliqué comment gérer manuellement la (dé)connexion ?

Les vues génériques répondent à un besoin basique. Si vous avez besoin d'implémenter des spécificités lors de la connexion, il est important de savoir comment procéder manuellement.

Vous avez vu comment utiliser les vues génériques dans le chapitre sur celles-ci. Nous ne ferons donc ici que les lister, avec leurs paramètres et mode de fonctionnement.

Se connecter

Vue : `django.contrib.auth.views.login`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/login.html`)

Contexte du template :

- `form` : le formulaire à afficher
- `next` : l'URL vers laquelle l'utilisateur sera redirigé après la connexion

Affiche le formulaire et se charge de vérifier si les données saisies correspondent à un utilisateur. Si c'est le cas, la vue redirige l'utilisateur vers l'URL indiquée dans `settings.LOGIN_REDIRECT_URL` ou vers l'URL passée par le paramètre GET « `next` » s'il y en a un, sinon il affiche le formulaire. Le template doit pouvoir afficher le formulaire et un bouton pour l'envoyer.

Se déconnecter

Vue : `django.contrib.auth.views.logout`

Arguments optionnels (un seul à utiliser) :

- `next_page` : l'URL vers laquelle le visiteur sera redirigé après la déconnexion
- `template_name` : le template à afficher en cas de déconnexion (par défaut `registration/logged_out.html`)
- `redirect_field_name` : utilise pour la redirection l'URL du paramètre GET passé en argument

Contexte du template :

- `title` : chaîne de caractères contenant "Déconnecté"

Déconnecte l'utilisateur et le redirige.

Se déconnecter puis se connecter

Vue : `django.contrib.auth.views.logout_then_login`

Arguments optionnels :

- `login_url` : l'URL de la page de connexion à utiliser (par défaut utilise `settings.LOGIN_URL`)

Contexte du template : aucun.

Déconnecte l'utilisateur puis le redirige vers l'URL contenant la page de connexion.

Changer le mot de passe

Vue : `django.contrib.auth.views.password_change`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_change_form.html`)
- `post_change_redirect` : l'URL vers laquelle rediriger l'utilisateur après le changement du mot de passe
- `password_change_form` : pour spécifier un autre formulaire que celui utilisé par défaut

Contexte du template :

- `form` : le formulaire à afficher

Affiche un formulaire pour modifier le mot de passe de l'utilisateur, puis le redirige si le changement s'est correctement déroulé. Le template doit contenir ce formulaire et un bouton pour l'envoyer.

Confirmation du changement de mot de passe

Vue : `django.contrib.auth.views.password_change_done`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_change_done.html`)

Contexte du template : vide.

Vous pouvez vous servir de cette vue pour afficher un message de confirmation après le changement de mot de passe. Il suffit de faire pointer la redirection de `django.contrib.auth.views.password_change` sur cette vue.

Demande de réinitialisation du mot de passe

Vue : `django.contrib.auth.views.password_reset`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_reset_form.html`)
- `email_template_name` : le nom du template à utiliser pour générer le mail qui sera envoyé à l'utilisateur avec le lien pour réinitialiser le mot de passe (par défaut `registration/password_reset_email.html`)
- `subject_template_name` : le nom du template à utiliser pour générer le sujet du mail envoyé à l'utilisateur (par défaut `registration/password_reset_subject.txt`)
- `password_reset_form` : pour spécifier un autre formulaire à utiliser que celui par défaut
- `post_reset_direct` : l'URL vers laquelle rediriger le visiteur après la demande de réinitialisation
- `from_email` : une adresse mail valide depuis laquelle sera envoyée le mail (par défaut, Django utilise `settings.DEFAULT_FROM_EMAIL`)

Contexte du template :

- `form` : le formulaire à afficher

Contexte du mail et du sujet :

- `user` : l'utilisateur concerné par la réinitialisation du mot de passe
- `email` : un alias pour `user.email`
- `domain` : le domaine du site web à utiliser pour construire l'URL (utilise `request.get_host()` pour obtenir la variable)
- `protocol` : `http` ou `https`
- `uid` : l'ID de l'utilisateur encodé en base 36
- `token` : le token unique de la demande de réinitialisation du mot de passe

La vue affiche un formulaire permettant d'indiquer l'adresse mail du compte à récupérer. L'utilisateur recevra alors un mail (il est important de configurer l'envoi de mail !) avec un lien vers la vue de confirmation de réinitialisation du mot de passe.

Voici un exemple du template pour générer le mail :

Code : Jinja

```
Une demande de réinitialisation a été envoyée pour le compte {{
user.username }}. Veuillez suivre le lien ci-dessous :
{{ protocol }}://{{ domain }}{% url 'password_reset_confirm'
uidb36=uid token=token %}
```

Confirmation demande de réinitialisation du mot de passe

Vue : `django.contrib.auth.views.password_reset_done`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_reset_done.html`)

Contexte du template : vide.

Vous pouvez vous servir de cette vue pour afficher un message de confirmation après la demande de réinitialisation du mot de passe. Il suffit de faire pointer la redirection de `django.contrib.auth.views.password_reset` sur cette vue.

Réinitialiser le mot de passe

Vue : `django.contrib.auth.views.password_reset_confirm`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_reset_confirm.html`)
- `set_password_form` : pour spécifier un autre formulaire à utiliser que celui par défaut
- `post_reset_redirect` : l'URL vers laquelle sera redirigé l'utilisateur après la réinitialisation.

Contexte du template :

- `form` : le formulaire à afficher
- `validlink` : booléen, mis à `True` si l'URL actuelle représente bien une demande de réinitialisation valide

Cette vue affichera le formulaire pour introduire un nouveau mot de passe, et se chargera de la mise à jour de ce dernier.

Confirmation de la réinitialisation du mot de passe

Vue : `django.contrib.auth.views.password_reset_complete`

Arguments optionnels :

- `template_name` : le nom du template à utiliser (par défaut `registration/password_reset_complete.html`)

Contexte du template : vide.

Vous pouvez vous servir de cette vue pour afficher un message de confirmation après la réinitialisation du mot de passe. Il suffit de faire pointer la redirection de `django.contrib.auth.views.password_reset_confirm` sur cette vue.

Les permissions et groupes

Le système utilisateur de Django fournit un système de permissions simple. Ces permissions permettent de déterminer si un utilisateur a le droit d'effectuer une certaine action ou non.

Les permissions

Une permission a la forme suivante « `nom_application.nom_permission` ».

Django crée 3 permissions pour chaque modèle enregistré. Ces permissions sont notamment utilisées dans l'administration. Si nous reprenons par exemple le modèle « Article » dans l'application « blog », 3 permissions sont créées par Django :

- `blog.add_article` : la permission pour créer un article
- `blog.change_article` : la permission pour modifier un article
- `blog.delete_article` : la permission pour supprimer un article

Il est bien entendu possible de créer des permissions nous-mêmes. Chaque permission dépend d'un modèle et doit être renseigné dans sa sous-classe `Meta`. Petit exemple en reprenant notre modèle `Article` utilisé au début :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField()
    date = models.DateTimeField(auto_now_add=True, auto_now=False,
    verbose_name="Date de parution")
    categorie = models.ForeignKey(Categorie)
```

```
def __unicode__(self):
    return self.titre

class Meta:
    permissions = (
        ("commenter_article", "Commenter un article"),
        ("marquer_article", "Marquer un article comme lu"),
    )
```

Pour ajouter de nouvelles permissions, il y a juste besoin de créer un tuple contenant les paires de votre permission, avec à chaque fois le nom de la permission et sa description.

Il est ensuite possible d'assigner des permissions à un utilisateur dans l'administration (cela se fait depuis la fiche d'un utilisateur).

Par la suite, pour vérifier si un utilisateur possède ou non une permission, il suffit de faire :

`user.has_perm("blog.commenter_article")`. La fonction renvoie `True` ou `False`, selon si l'utilisateur dispose de la permission ou non. Cette fonction est également accessible depuis les templates, encore grâce à un contexte processeur :

Code : Jinja

```
{% if perms.blog.commenter_article %}
<p><a href="/commenter/">Commenter</a></p>
{% endif %}
```

Le lien ici ne sera affiché que si l'utilisateur dispose de la permission pour commenter.

De même que pour le décorateur `login_required`, il existe un décorateur permettant de s'assurer que l'utilisateur qui souhaite accéder à la vue dispose bien de la permission nécessaire. Il s'agit de `django.contrib.auth.decorators.permission_required`.

Code : Python

```
from django.contrib.auth.decorators import permission_required

@permission_required('blog.commenter_article')
def article_commenter(request, article):
    ...
```

Sachez qu'il est également possible de créer une permission dynamiquement. Pour cela, il faut importer le modèle `Permission`, situé dans `django.contrib.auth.models`.

Ce modèle possède les attributs suivants :

- `name` : le nom de la permission, 50 caractères maximum.
- `content_type` : un `content_type` pour désigner le modèle concerné.
- `codename` : le nom de code de la permission.

Donc, si nous souhaitons par exemple créer une permission « commenter un article » spécifique à chaque article, et ce à chaque fois que nous créons un nouvel article, voici comment procéder :

Code : Python

```
from django.contrib.auth.models import Permission
from blog.models import Article
from django.contrib.contenttypes.models import ContentType

... # Récupération des données
article.save()

content_type = ContentType.objects.get(app_label='blog', model='Article')
```

```
permission =  
Permission.objects.create(codename='commenter_article_{0}'.format(article.id),  
                           name='Commenter l'article  
"{0}"'.format(article.titre),  
                           content_type=content_type)
```

Une fois que la permission est créée, il est possible de l'assigner à un utilisateur précis de cette façon :

Code : Python

```
user.user_permissions.add(permission)
```

Pour rappel, `user_permissions` est une relation ManyToMany de l'utilisateur vers la table des permissions.

Les groupes

Imaginons que vous souhaitiez attribuer certaines permissions à tout un ensemble d'utilisateurs, mais sans devoir les assigner une à une à chaque utilisateur (ça serait beaucoup trop long et répétitif !). Devant cette situation épineuse, Django propose une solution très simple : les groupes.

Un groupe est simplement un regroupement d'utilisateurs auquel on peut assigner des permissions. Une fois qu'un groupe dispose d'une permission, tous ses utilisateurs en disposent automatiquement aussi. Il s'agit donc d'un modèle, `django.contrib.auth.models.Group`, qui dispose des champs suivants :

- `name` : le nom du groupe (80 caractères maximum) ;
- `permissions` : une relation ManyToMany vers les permissions, comme `user_permissions` pour les utilisateurs

Pour ajouter un utilisateur à un groupe, il faut utiliser la relation ManyToMany `groups` de `User` :

Code : Python

```
>>> from django.contrib.auth.models import User, Group  
>>> group = Group(name=u"Les gens géniaux")  
>>> group.save()  
>>> user = User.objects.get(username="Mathieu")  
>>> user.groups.add(group)
```

Une fois ceci fait, l'utilisateur « Mathieu » dispose de toutes les permissions attribuées au groupe « Les gens géniaux ».

Voilà ! Vous avez désormais vu en long et en large le système utilisateur que propose Django. Vous avez pu remarquer donc qu'il est tout aussi puissant que flexible. Inutile donc de réécrire tout un système utilisateur lorsque le framework en propose déjà un plus que correct.

Voilà ! Vous avez désormais vu en long et en large le système utilisateur que propose Django. Vous avez pu remarquer donc qu'il est tout aussi puissant que flexible. Inutile donc de réécrire tout un système utilisateur lorsque le framework en propose déjà un plus que correct.

Les messages

Il est souvent utile d'envoyer des notifications au visiteur, pour par exemple lui confirmer qu'une action s'est bien réalisée, ou qu'au contraire, lui indiquer une erreur. Django propose un petit système de notification simple et pratique qui répond parfaitement à ce besoin. Nous le présenterons dans ce chapitre.

Les bases

Avant tout, il faut s'assurer que l'application et ses dépendances sont bien installées. Elle l'est généralement par défaut, néanmoins il est toujours utile de vérifier. Autrement dit, dans votre settings.py, vous devez avoir :

- dans MIDDLEWARE_CLASSES, 'django.contrib.messages.middleware.MessageMiddleware'
- dans INSTALLED_APPS, 'django.contrib.messages'
- dans TEMPLATE_CONTEXT_PROCESSORS (si cette variable n'est pas dans votre settings.py, inutile de la rajouter, elle contient déjà l'entrée par défaut), 'django.contrib.messages.context_processors.messages'

Ceci étant fait, nous pouvons attaquer le sujet.

Django peut envoyer des notifications (aussi appelés messages) à tous les visiteurs, qu'ils soient connectés ou non. Il existe plusieurs niveaux de messages par défaut (nous verrons comment en rajouter par la suite) :

- DEBUG (debug) : messages destinés pour la phase de développement uniquement. Ces messages ne seront pas affichés en production
- INFO (info) : messages d'information pour l'utilisateur
- SUCCESS (success) : confirmation qu'une action s'est bien déroulée
- WARNING (warning) : une erreur n'a pas été rencontrée, mais pourrait être imminente
- ERROR (error) : une action ne s'est pas déroulée correctement ou une erreur quelconque est apparue

Les mots en parenthèses sont ce qu'on appelle les « tags » de chaque niveau. Ces tags sont notamment utilisés pour pouvoir définir un style CSS précis à chaque niveau, afin de pouvoir les différencier.

Voici la fonction à appeler pour envoyer un message depuis une vue :

Code : Python

```
from django.contrib import messages
messages.add_message(request, messages.INFO, 'Bonjour visiteur !')
```

Ici, nous avons envoyé un message, avec le niveau INFO, au visiteur, contenant « Bonjour visiteur ! ». Il est important de ne pas oublier le premier argument : request, l'objet HttpRequest donné à la vue. Sans cela, Django ne saura pas à quel visiteur envoyer le message.

Il existe également quelques raccourcis pour les niveaux par défaut :

Code : Python

```
messages.debug(request, '%s requêtes SQL ont été exécutées.' %
compteur)
messages.info(request, 'Rebonjour !')
messages.success(request, 'Votre article a bien été mis à jour.')
messages.warning(request, 'Votre compte expire dans 3 jours.')
messages.error(request, 'Cette image n\'existe plus.')
```

Maintenant que vous savez envoyer des messages, il ne reste plus qu'à les afficher. Pour ce faire, Django se charge de la majeure partie du travail. Tout ce qu'il reste à faire, c'est de choisir où afficher les notifications dans le template. Les variables contenant celles-ci sont automatiquement incluses grâce à un processeur de contexte.

Voici une ébauche de code de template pour afficher les messages au visiteur :

Code : Jinja

```
{% if messages %}
<ul class="messages">
{% for message in messages %}
<li{% if message.tags %} class="{{ message.tags }}" {% endif %}>{{
message }}</li>
{% endfor %}
</ul>
{% endif %}
```

La variable `messages` étant ajoutée automatiquement par le framework, il suffit d'itérer dessus comme elle peut contenir plusieurs notifications. Sur l'itération de chaque message, ce dernier sera supprimé tout seul, ce qui assure que l'utilisateur ne verra pas deux fois la même notification. Finalement, un message dispose de l'attribut `tags` pour d'éventuels styles CSS (il peut y avoir plusieurs tags, ils seront séparés par un espace), et il suffit ensuite d'afficher le contenu de la variable.

Ce code est de préférence à intégrer dans une base commune (appelée depuis `{% extends %}`) pour éviter de devoir le réécrire dans tous vos templates.

Dans les détails

Nous avons dit qu'il était possible de créer des niveaux de message personnalisés. Il faut savoir qu'en réalité, les niveaux de messages ne sont en fait que des entiers constants :

Code : Python

```
>>> from django.contrib import messages
>>> messages.INFO
20
```

Voici la relation entre niveau et entier par défaut :

- DEBUG : 10
- INFO : 20
- SUCCESS : 25
- WARNING : 30
- ERROR : 40

En réalité, rajouter un niveau suffit juste à créer une nouvelle constante. Par exemple :

Code : Python

```
CRITICAL = 50

messages.add_message(request, CRITICAL, 'Une erreur critique est
apparue.')
```

Il est dès lors tout aussi possible d'ajouter des tags à un message :

Code : Python

```
messages.add_message(request, CRITICAL, 'Une erreur critique est
apparue.', extra_tags="fail")
```

Ici, le tag « fail » sera rajouté.

Pour terminer, sachez que vous pouvez également limiter l'affichage des messages à un certain niveau (égal ou supérieur). Cela peut se faire de deux manières différentes. Soit depuis le `settings.py` en mettant `MESSAGE_LEVEL` au niveau minimum des

messages à afficher (par exemple 25 pour ne pas montrer les messages DEBUG et INFO), ou en faisant cette requête dans la vue :

Code : Python

```
messages.set_level(request, messages.DEBUG)
```

en utilisant ceci et pour cette vue uniquement, tous les messages dont le niveau est égal ou supérieur à 10 seront affichés.

La mise en cache

Lorsqu'un site devient populaire, que son trafic augmente et que toutes les ressources du serveur sont utilisées, on constate généralement des ralentissements ou des plantages du site. Ces deux cas de figure sont généralement à éviter, et pour ce faire, il faut procéder à des optimisations. Une optimisation courante est la mise en cache, que nous aborderons dans ce chapitre.

Cachez-vous !

Avant d'aborder l'aspect technique, il faut comprendre l'utilité de la mise en cache. Si vous présentez dans une vue des données issues de calculs très longs et complexes (par exemple, une dizaine de requêtes SQL), l'idée est de n'effectuer les calculs une fois, sauvegarder le résultat et de présenter le résultat sauvegardé pour les prochaines visites, plutôt que de recalculer la même donnée à chaque fois.

En effet, sortir une donnée du cache étant bien plus rapide et plus simple que de la recalculer, la durée d'exécution de la page est largement réduite, ce qui laisse donc davantage de ressources pour d'autres requêtes. Finalement, le site sera moins enclin à planter ou être ralenti.

Une question subsiste : où faut-il sauvegarder les données ? Django enregistre les données dans un système de cache et le framework dispose de plusieurs systèmes de cache différents de base. Nous tâcherons de les introduire brièvement dans ce qui suit.

Chacun de ces systèmes ont leur avantages et désavantages, et tous fonctionnent à peu près différemment. Il s'agit donc de trouver le système de cache le plus adapté à vos besoins.

La configuration du système de cache se fait grâce à la variable `CACHES` de votre `settings.py`.

Dans des fichiers

Un système de cache simple est celui enregistrant les données dans des fichiers sur le disque dur. Pour chaque valeur enregistrée dans le cache, le système va créer un fichier et y enregistrer le contenu de la donnée sauvegardée. Voici comment le configurer :

Code : Python

```
CACHES = {
    'default': {
        'BACKEND':
        'django.core.cache.backends.filebased.FileBasedCache',
        'LOCATION': '/var/tmp/django_cache',
    }
}
```

La clé `'LOCATION'` doit pointer vers un dossier, et non pas un fichier spécifique. Sachez que si vous êtes sous Windows, voici ce à quoi doit ressembler la valeur de `'LOCATION'` : `'c:/mon/dossier'` (ne pas oublier le `c:/` ou autre identifiant de votre disque dur).

La clé `'BACKEND'` indique simplement le système de cache utilisé (et sera à chaque fois différent pour chaque système présenté par la suite).

Une fois ce système de cache configuré, Django va créer des fichiers dans le dossier concerné. Ces fichiers seront "sérialisés" en utilisant le module `pickle` pour y encoder les données à sauvegarder. Vous devez également vous assurer que votre serveur web a bien accès en écriture et en lecture sur le dossier que vous avez indiqué.

Dans la mémoire

Un autre système de cache simple est la mise en mémoire. Toutes vos données seront enregistrées dans la mémoire vive du serveur. Voici la configuration de votre serveur :

Code : Python

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.locmem.LocMemCache',
```

```
        'LOCATION': 'cache_crepes'
    }
}
```

Si vous utilisez cette technique, faites attention à la valeur de `'LOCATION'`. En effet, si plusieurs sites avec Django utilisant cette technique de cache tournent sur le même serveur, il est impératif que chacun d'entre eux dispose d'un nom de code différent pour `'LOCATION'`. Il s'agit en réalité juste d'un identifiant de l'instance du cache. Si plusieurs sites partagent le même identifiant, ils risquent de rentrer en conflit.

Dans la base de données

Pour utiliser la base de données comme système de cache, il faut avant tout créer une table dans celle-ci pour y accueillir les données. Ceci se fait grâce à une commande spéciale de `manage.py` :

Code : Console

```
python manage.py createcachetable [nom_table_cache]
```

où `nom_table_cache` est le nom de la table que vous allez créer (faites bien attention à utiliser un nom valide et pas déjà utilisé). Une fois ceci fait, tout ce qu'il reste à faire est de l'indiquer dans le `settings.py` :

Code : Python

```
CACHES = {
    'default': {
        'BACKEND': 'django.core.cache.backends.db.DatabaseCache',
        'LOCATION': 'nom_table_cache',
    }
}
```

Ce système peut se révéler pratique et rapide si vous avez dédié tout un serveur physique à votre base de données, néanmoins il faut disposer de telles ressources pour arriver à quelque chose de convenable.

En utilisant memcache

Memcache est un système de cache un peu à part. Celui-ci est en réalité indépendant de Django, et le framework ne s'en charge pas lui-même. Pour l'utiliser, il faut avant tout lancer un programme responsable lui-même du cache. Django ne fera qu'envoyer les données à mettre en cache et les récupérer par la suite, c'est au programme de sauvegarder et de gérer ces données.

Si cela peut sembler assez pénible à déployer, le système est en revanche très rapide et probablement le plus efficace de tous. Memcache va enregistrer les données dans la mémoire vive, comme le système vu précédemment qui utilisait la même technique, sauf qu'au contraire de ce dernier, Memcache est bien plus efficace et utilise moins de mémoire.

Nous ne couvrirons pas l'installation du programme dans ce tutoriel, mais uniquement sa configuration côté Django. La configuration est encore une fois relativement simple :

Code : Python

```
CACHES = {
    'default': {
        'BACKEND':
            'django.core.cache.backends.memcached.MemcachedCache',
        'LOCATION': '127.0.0.1:11211',
    }
}
```

La clé '**LOCATION**' indique la combinaison adresse IP/port depuis laquelle Memcache est accessible.

Pour le développement

Pour terminer, il existe un dernier système de cache. Celui-ci ne fait rien (il n'enregistre aucune donnée et n'en renvoie aucune). Il permet juste d'activer le système de cache, ce qui peut se révéler pratique si vous utilisez le cache en production, mais que vous n'en avez pas besoin en développement. Voici sa configuration :

Code : Python

```
CACHES = {  
    'default': {  
        'BACKEND': 'django.core.cache.backends.dummy.DummyCache',  
    }  
}
```



Au final, quel système choisir ?

Cela dépend de votre site et de vos attentes. En développement, vous pouvez utiliser le cache de développement ou le cache mémoire (le simple, pas celui-ci utilisant memcache) si vous en avez besoin. En production, si vous avez peu de données à mettre en cache, la solution la plus simple est probablement le système utilisant les fichiers.

En revanche, lorsque le cache devient fortement utilisé, Memcache est probablement la meilleure solution si vous pouvez l'installer, sinon utilisez le système utilisant la base de données, même s'il n'est pas aussi efficace que Memcache, il devrait tout de même apaiser votre serveur.

Quand les données jouent à cache-cache

Maintenant que notre cache est configuré, il ne reste plus qu'à l'utiliser. Il existe différentes techniques de mise en cache que nous expliquerons dans ce sous-chapitre.

Cache par vue

Une méthode de cache pratique est la mise en cache d'une vue. Avec cette technique, dès que le rendu d'une vue sera calculé, ce rendu sera directement enregistré dans le cache. Tant que celui-ci sera dans le cache, la vue ne sera plus appelée et la page sera directement cherchée dans le cache.

Cette mise en cache se fait grâce à un décorateur : `django.views.decorators.cache.cache_page`. Voici son utilisation :

Code : Python

```
from django.views.decorators.cache import cache_page  
  
@cache_page(60 * 15)  
def ma_vue(request):  
    ...
```

Le paramètre du décorateur correspond à la durée après laquelle le rendu dans le cache aura expiré. Cette durée est exprimée en secondes. Autrement dit, ici, après 60×15 secondes, donc 15 minutes, la donnée sera supprimée du cache et Django devra régénérer la page, puis remettre la nouvelle version dans le cache. Grâce à cet argument, vous êtes assurés que le cache reste à jour automatiquement.

Bien évidemment, chaque URL aura sa propre mise en cache. En effet, si `/article/42/` et `/article/1337/` pointent vers la même vue `lire_article`, où le nombre correspond à l'ID d'un article précis dans la base de données :

Code : Python

```
@cache_page
```

```
def lire_article(request, id):  
    article = Article.objects.get(id=id)  
    ...
```

il est normal que `/article/42/` et `/article/1337/` ne partagent pas le même résultat en cache (étant donné qu'ils n'affichent pas le même article).

Il est également possible de spécifier une mise en cache directement depuis les `URLconf`. Ainsi, la mise en cache de vues génériques est également possible :

Code : Python

```
from django.views.decorators.cache import cache_page  
  
urlpatterns = ('',  
              (r'^article/(\d{1,4})/$', cache_page(60 * 15)(lire_article)),  
              )
```

Ici, le décorateur `@cache_page` est tout de même appliqué à la vue. Faites bien attention à inclure la vue sous forme de référence, et non pas sous forme de chaîne de caractères.

Dans les templates

Il est également possible de mettre en cache certaines parties d'un template. Cela se fait grâce au tag `{% cache %}`. Ce tag doit au préalable être inclus grâce à la directive `{% load cache %}`. De plus, `cache` prend deux arguments au minimum : la durée d'expiration de la valeur (toujours en secondes), et le nom de cette valeur en cache (une sorte de clé que Django utilisera pour retrouver la bonne valeur dans le cache) :

Code : Jinja

```
{% load cache %}  
{% cache 500 carousel %}  
/* mon carousel */  
{% endcache %}
```

Ici, nous enregistrons dans le cache notre carousel. Celui-ci expirera dans 500 secondes et on utilise la clé "carousel".

Sachez également que vous pouvez également enregistrer plusieurs copies en cache d'une même partie de template dépendant de plusieurs variables. Par exemple, si notre carousel est différent pour chaque utilisateur, vous pouvez réutiliser une clé dynamique et différente pour chaque utilisateur. Ainsi, chaque utilisateur connecté aura dans le cache sa copie du carousel adapté à son profil. Exemple :

Code : Jinja

```
{% load cache %}  
{% cache 500 user.username %}  
/* mon carousel adapté à user */  
{% endcache %}
```

La mise en cache bas-niveau

Il arrive parfois qu'enregistrer toute une vue ou une partie de template soit une solution exagérée, et non adaptée. C'est là qu'interviennent plusieurs fonctions permettant de réaliser une mise en cache de variables bien précises. Presque tous les types de variables peuvent être mis en cache. Ces opérations sont réalisées grâce à plusieurs fonctions à l'objet `cache` du module

`django.core.cache`. Cet objet cache se comporte un peu comme un dictionnaire. On peut lui assigner des valeurs à travers des clés :

Code : Python

```
>>> from django.core.cache import cache
>>> cache.set('ma_cle', 'Coucou !', 30)
>>> cache.get('ma_cle')
'Coucou !'
```

Ici, la clé 'ma_cle' contenant la chaîne de caractères 'Coucou !' a été enregistrée pendant 30 secondes dans le cache (l'argument de la durée est optionnel, s'il n'est pas spécifié, la valeur par défaut de la configuration sera utilisée). Vous pouvez essayer, après ces 30 secondes, `get` renvoie `None` si la clé n'existe pas ou plus :

Code : Python

```
>>> cache.get('ma_cle')
None
```

Il est possible de spécifier une valeur par défaut si la clé n'existe pas/plus :

Code : Python

```
>>> cache.get('ma_cle', 'a expiré')
'a expiré'
```

Pour essayer d'ajouter une clé si elle n'est pas déjà présente, il faut utiliser la méthode `add`. Si cette clé est déjà présente, rien ne se passe :

Code : Python

```
>>> cache.set('cle', 'Salut')
>>> cache.add('cle', 'Coucou')
>>> cache.get('cle')
'Salut'
```

Pour ajouter et obtenir plusieurs clés à la fois, il existe deux fonctions adaptées, `set_many` et `get_many` :

Code : Python

```
>>> cache.set_many({'a': 1, 'b': 2, 'c': 3})
>>> cache.get_many(['a', 'b', 'c'])
{'a': 1, 'b': 2, 'c': 3}
```

Vous pouvez également supprimer une clé du cache :

Code : Python

```
>>> cache.delete('a')
```

ou plusieurs en même temps :

Code : Python

```
>>> cache.delete_many(['a', 'b', 'c'])
```

Pour vider tout le cache, voici la méthode `clear` :

Code : Python

```
>>> cache.clear()
```

Toutes les clés et leurs valeurs seront supprimées.

Pour terminer, il existe encore deux fonctions, `incr` et `decr`, qui permettent respectivement d'incrémenter et de décrémenter un nombre dans le cache :

Code : Python

```
>>> cache.set('num', 1)
>>> cache.incr('num')
2
>>> cache.incr('num', 10)
12
>>> cache.decr('num')
11
>>> cache.decr('num', 5)
6
```

Le deuxième paramètre permet de spécifier le nombre d'incrémentations ou de décrémentations à effectuer.

Voilà ! Vous avez désormais couvert les bases du système de cache de Django. Cependant, nous n'avons vraiment que couvert les bases, il reste plein d'options à explorer, la possibilité d'implémenter son propre système de cache, etc. Si vous vous sentez limités sur ce sujet ou que vous avez d'éventuelles questions non reprises dans ce chapitre, consultez la documentation :

<https://docs.djangoproject.com/en/1.4/topics/cache/>

Partie 5 : Annexes

Déployer votre application en production

Tout au long du tutoriel, nous avons utilisé le serveur de développement de Django. Cependant, ce serveur de développement n'est adapté que pour le développement, et pas du tout pour la mise en production dans une situation réelle.

Nous allons voir dans ce chapitre comment déployer un projet Django en production.

Le déploiement

Contrairement à ce que certains peuvent penser, **NON**, le serveur de développement ne peut pas être utilisé en production. En effet, celui-ci n'apporte pas les conditions de sécurité et de performances suffisant pour garantir un service stable. Le rôle d'un framework n'est pas de distribuer les pages web, c'est au serveur web qu'incombe ce travail.

Nous allons voir comment installer notre projet sur un **serveur Apache 2 avec le mod_wsgi** (cependant, tout autre serveur web avec le protocole WSGI peut faire l'affaire aussi). Le protocole WSGI (Web Server Gateway Interface) est une sorte de couche qui permet à un serveur web et une application web Python de communiquer ensemble. Par défaut, Django fournit un fichier `wsgi.py` qui s'occupera de cette liaison. Pour rappel :

Code : Autre

```
crepes_bretonnes/
  manage.py
  crepes_bretonnes/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Ce fichier n'a pas besoin d'être modifié. Il est normalement correctement généré selon les paramètres de votre projet.

Il faut savoir qu'un projet Django ne se déploie pas comme un projet PHP. En effet, si nous tentons d'héberger le projet sur un serveur Apache avec une configuration basique, voilà le résultat :

Index of /

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 Parent Directory		-	
 __init__.py	14-Jul-2012 10:22	0	
 blog/	14-Jul-2012 10:22	-	
 crepes_bretonnes/	14-Jul-2012 10:22	-	
 db	14-Jul-2012 10:22	47K	
 manage.py	14-Jul-2012 10:22	259	
 templates/	14-Jul-2012 10:22	-	
 urlshortener/	14-Jul-2012 10:22	-	

Ma liste de fichiers Python, qui est possible de télécharger seulement...

Non seulement votre code n'est pas exécuté, mais il est lisible par tous. Il nous faut donc spécifier à Apache d'utiliser le protocole WSGI pour que Django puisse exécuter le code et renvoyer du HTML.

Dans un premier temps il va falloir **installer le module WSGI**. Sous la plupart des distributions Linux, un paquet existe pour nous simplifier la vie. Par exemple, pour Debian :

Code : Console

```
# aptitude install libapache2-mod-wsgi
```

N'oubliez cependant pas, si vous n'avez pas Django ou Apache2 d'installé, de les installer également bien entendu. Ensuite, modifions le fichier `/etc/apache2/httpd.conf` pour indiquer où trouver notre application. Si ce fichier n'existe pas, créez-le. Voici la configuration à insérer :

Code : Autre

```
WSGIScriptAlias / /chemin/vers/crepes_bretonnes/crepes_bretonnes/wsgi.py
WSGIProxyPath /chemin/vers/crepes_bretonnes/
<Directory /chemin/vers/crepes_bretonnes/>
    <Files wsgi.py>
        Order deny,allow
        Allow from all
    </Files>
</Directory>
```

- La première ligne, *WSGIScriptAlias*, indique que toutes les URLs commençant par / (qui indique la racine du serveur) devront utiliser l'application définie par le second argument, qui est ici le chemin vers notre fichier *wsgi.py*.
- La seconde ligne, *WSGIProxyPath*, permet de rendre accessible via la commande import en Python, votre projet. Ainsi, le module *wsgi* pourra lancer notre projet Django.
- Enfin, la directive *<Directory ...>* permet de s'assurer que le serveur Apache peut accéder au fichier *wsgi.py* uniquement.

Sauvegardez ce fichier. Si vous souhaitez changer des informations sur le nom de domaine ou le port, il faudra passer par les *VirtualHosts* d'Apache (ce que nous ne couvrirons pas ici).

Nous allons pouvoir modifier les paramètres de notre projet (*settings.py*). Dans un premier temps, à la création de notre projet, nous avons défini quelques variables : base de données, chemin d'accès, etc. Il va falloir les adapter à notre serveur de production.

Voici les variables à modifier :

- Passer la variable `DEBUG` à `False` pour indiquer que le site est désormais en production. Ceci est très important, sans quoi les erreurs et des données sensibles seront affichées !
- Adaptez la connexion à la base de données en fonction de ce que vous souhaitez utiliser en production. N'oubliez pas d'installer les extensions nécessaires si vous souhaitez utiliser autre chose que *SQLite*.
- Adaptez le chemin vers le dossier `templates/` et les divers autres dossiers possibles

Sauvegardez et relancez Apache (`service apache2 reload`). Votre site doit normalement être accessible !



Si vous obtenez une erreur "Internal Server Error", pas de panique, c'est sûrement dû à une erreur dans votre configuration. Pour traquer l'erreur, faites un `tail -f /var/log/apache2/error.log`, et regarder l'exception lancée lors du chargement d'une page.

Gardez un oeil sur le projet

Une application n'est jamais parfaite, et des erreurs peuvent tout le temps faire surface, même après la mise en production malheureusement. Cependant, lorsqu'une erreur survient en production, un problème apparaît : comment être au courant de l'erreur rencontrée et dans quelles circonstances s'est-elle produite ? Une solution serait de vérifier régulièrement les journaux d'erreur de votre serveur web, mais si une erreur critique apparaît, vous seriez le dernier prévenu. Vous n'aurez pas non plus le contexte de l'erreur. Pour résoudre ce fâcheux problème, Django propose une solution simple : il vous enverra un mail à chaque erreur rencontrée avec le contexte de celle-ci !


```

De: root@localhost
Subject: [Django] INTERNAL IP: Internal Server Error: Accounts/login/
Pour: Maxime Lorant, Mathieu Xhonneux
Date: 18/01/2012 20:22
Autres actions:

Traceback (most recent call last):
  File "/usr/local/alwaysdata/python/django/1.4/django/core/handlers/base.py", line 133, in get_response
    response = callback(request, *callback_args, **callback_kwargs)
  File "/home/.../tools/ratelimit.py", line 22, in wrapper
    return self.view_wrapper(request, fn, *args, **kwargs)
  File "/home/.../tools/ratelimit.py", line 46, in view_wrapper
    return fn(request, *args, **kwargs)
  File "/home/.../home/views.py", line 88, in user_login
    if form.is_valid():
  File "/usr/local/alwaysdata/python/django/1.4/django/forms/forms.py", line 124, in is_valid
    return self.is_bound and not bool(self.errors)
  File "/usr/local/alwaysdata/python/django/1.4/django/forms/forms.py", line 115, in _get_errors
    self.full_clean()
  File "/usr/local/alwaysdata/python/django/1.4/django/forms/forms.py", line 271, in full_clean
    self._clean_form()
  File "/usr/local/alwaysdata/python/django/1.4/django/forms/forms.py", line 299, in _clean_form
    self.cleaned_data = self._clean()
  File "/home/.../home/forms.py", line 38, in _clean
    u = User.objects.get(email=email)
  File "/usr/local/alwaysdata/python/django/1.4/django/db/models/manager.py", line 131, in get
    return self.get_query_set().get(*args, **kwargs)

```

Un exemple de mail reçu sur un projet : une erreur toute bête qui a été corrigée quelques minutes après la réception de ce mail !

Ce mail contient plusieurs types d'informations : le traceback complet de l'erreur Python, les données HTTP de la requête et d'autres variables bien pratiques (informations sur la requête HTTP, état de la couche WSGI, etc.). Ces dernières ne sont pas affichées dans l'image (elles viennent après dans le mail).

Activer l'envoi de mails

Dans un premier temps, assurez-vous qu'un serveur de mail est installé sur votre machine, permettant d'envoyer des mails via le protocole SMTP.

Pour pouvoir recevoir ces alertes, assurez-vous d'avoir encore une fois votre variable `DEBUG` à `False`. Les mails ne sont envoyés que dans ce cas là. En effet, en production, les exceptions sont affichées directement dans le navigateur lorsque l'erreur est lancée.

Ensuite, assurez-vous également que la variable `ADMINS` de votre `settings.py` soit correcte et à jour. En effet, ce sont les administrateurs présents dans cette liste qui recevront les mails d'erreurs. Pour rappel, lors de la création de notre projet, nous avons mis ceci :

Code : Python

```

ADMINS = (
    ('Maxime Lorant', 'maxime@crepes-bretonnes.com'),
    ('Mathieu Xhonneux', 'mathieu@crepes-bretonnes.com'),
)

```

Ici, les mails d'erreurs sont envoyés aux deux personnes, en même temps. Si vous observez la capture ci-dessus, on reçoit tous les deux le message d'erreur sur le projet développé.



Par défaut, Django envoie les mails depuis l'adresse `root@localhost`. Cependant, certaines boîtes mail rejettent cette adresse, ou tout simplement vous souhaitez avoir quelque chose de plus propre. Dans ce cas, vous pouvez personnaliser l'adresse en ajoutant une variable dans votre `settings.py` : `SERVER_EMAIL='adresse@domain.com'`

Quelques options utiles...

Être averti des pages 404

Par défaut, les pages non trouvées ne sont pas signalées par mail. Si vous voulez toutefois les recevoir, ajoutez les lignes suivantes dans votre `settings.py` :

Code : Python

```

SEND_BROKEN_LINK_EMAILS = True

```

```
MANAGERS = ADMINS # A ajouter après ADMINS.
```

Assurez-vous par la même occasion que `CommonMiddleware` est dans votre `MIDDLEWARE_CLASSES` (ce qui est le cas par défaut). Si c'est le cas, Django va envoyer un mail à toutes les personnes dans `MANAGERS` (ici, les administrateurs en fait) lorsque le code d'erreur 404 sera déclenché par quelqu'un. Il est également possible de filtrer ces envois, via la configuration de `IGNORABLE_404_URLS`

Code : Python

```
import re
IGNORABLE_404_URLS = (
    re.compile(r'\.(php|cgi)$'),
    re.compile(r'^/phpmyadmin/'),
    re.compile(r'^/apple-touch-icon.*\.png$'),
    re.compile(r'^/favicon\.ico$'),
    re.compile(r'^/robots\.txt$'),
)
```

Ici, les fichiers `*.php`, le dossier `phpmyadmin/`, etc. ne seront pas concernés.

Filtrer les données sensibles

Enfin, il peut arriver qu'une erreur de votre code arrive lors de la saisie de **données sensibles** : saisie d'un mot de passe, d'un numéro de carte bleue... Pour des raisons de sécurité, il est nécessaire de cacher ces informations dans les mails d'erreurs ! Pour ce faire, on doit déclarer au dessus de chaque vue contenant des informations critiques quelles sont les variables à cacher :

Code : Python

```
from django.views.decorators.debug import sensitive_variables
from django.views.decorators.debug import sensitive_post_parameters

@sensitive_variables('user', 'password', 'carte')
def paiement(user):
    user = get_object_or_404(User, id=user)
    password = user.password
    carte = user.carte_credit

    raise Exception

@sensitive_post_parameters('password')
def connexion(request):
    raise Exception
```



Ne surtout pas laisser ces informations même si vous êtes le seul à avoir ces mails et que vous vous sentez confiant. L'accès au mot de passe en clair est très mal vu pour le bien des utilisateurs et on est jamais à l'abri d'une fuite (vol de compte mail, écoute de paquets...)

Hébergeurs supportant Django

Nous avons vu comment installer son projet Django sur un serveur dédié. Cependant tout le monde n'a pas la chance d'avoir un serveur à soi. Il existe cependant des alternatives. De plus en plus d'hébergeurs proposent désormais le support de langages et outils autres que le PHP : Java/J2EE, Ruby On Rails et bien sûr Django !

Voici la liste des hébergeurs notables :

Nom	Caractéristiques	Offre
Alwaysdata	Large panel, dont une offre gratuite. Le support est très réactif, leur site est même codé avec Django ! Plus de détails ici	Gratuite et

	Django : Plus de détails ici	Payante
WebFaction	Site international (serveurs à Amsterdam pour l'Europe), propose le support de Django sans frais supplémentaires. Les quotas sont très flexibles.	Payant
Djangoeurope	Comme son nom l'indique, Djangoeurope est spécialisé dans l'hébergement de projets Django. Il fournit donc une interface adaptée à la mise en production de votre projet	Payant
DjangoFoo Hosting	Support de plusieurs versions de Django, accès rapide à la gestion de projet, via le manage.py, redémarrage automatique de serveurs... Une vraie mine d'or d'après les utilisateurs !	Payant

Une ligne plus exhaustive est [disponible ici](#) (site officiel anglais). Comme vous pouvez le voir la majorité de ces hébergeurs sont cependant payants.

Mémento des filtres

Ce chapitre a pour unique but de faire une présentation exhaustive des filtres présent de base dans Django. Il est cependant possible de créer vos propres filtres par la suite, si le panel présenté ici ne vous suffit pas 😊

Liste des filtres add

Ajoute l'argument donné à la valeur de la variable. Ce filtre marche aussi bien sur des entiers que sur des chaînes ou des objets.

Code : Jinja

```
{{ 2|add:3 }} renvoie 5
{{ "Bonjour "|add:"tout le monde" }} renvoie "Bonjour tout le monde"
```

Si on a `list1 = [1, 2, 3]` et `list2 = [4, 5, 6]`
`{{ list1|add:list2 }}` renvoie `[1, 2, 3, 4, 5, 6]`

Si une chaîne peut être converti en entier, alors c'est l'addition qui est effectuée et non la concaténation de chaîne !

capfirst

Met en majuscule la première lettre de la chaîne

Code : Jinja

```
{{ "filtre"|capfirst }} retourne "Filtre"
{{ "bonjour le monde"|capfirst }} retourne "Bonjour le monde"
```

center

Ajoute des espaces avant et après la chaîne afin qu'elle soit centrée selon une taille donnée. Si le texte est plus grand que la taille donnée, alors rien n'est ajouté.

Code : Jinja

```
{{ "Texte"|center:20 }} renvoie " Texte "
```



Attention, le résultat n'est pas directement visible depuis une page web. Pensez à intégrer votre texte dans une balise `<pre>`

cut

Permet de supprimer toutes les occurrences d'une sous-chaîne dans une chaîne de caractères données. Ce filtre est sensible à la casse.

Code : Jinja

```
{{ "Mon tuto sur Django"|cut:" " }} renvoie "MontutosurDjango"
{{ "AhahAhah"|cut:"ah" }} renvoie "AhAh"
```

date

Permet de formater une date selon le format voulu. La syntaxe est similaire à celle utilisé par la fonction `date()` de PHP avec quelques nuances. La documentation complète sur cette syntaxe est [disponible sur le site de Django](#).

Des valeurs par défaut existe déjà, afin d'éviter de devoir recopier les mêmes chaînes à chaque fois. Ainsi, des constantes sont prédéfinies, qu'il est possible de redéfinir dans votre projet :

- `DATE_FORMAT` (Par défaut : 'N j, Y')
- `DATETIME_FORMAT` (Par défaut : 'N j, Y P')
- `SHORT_DATE_FORMAT` (Par défaut : 'm/d/Y')
- `SHORT_DATETIME_FORMAT` (Par défaut : 'm/d/Y P')

L'affichage de la date varie en fonction de la langue choisie dans le projet (notamment au niveau du nom des mois et des jours)

Code : Jinja

```
{{ current_date|date }} retourne "July 4, 2012" (configuration par défaut de Django)
{{ current_date|date:"d/m/Y H:i" }} retourne "04/07/2012 21:45"
```

default

Permet l'affichage d'une valeur par défaut, si jamais la variable donnée est égal vide ou égal à `False`.

Code : Jinja

```
Bonjour {{ pseudo|default:"visiteur" }} retourne "Bonjour visiteur"
si pseudo est nul ou vide
```

default_if_none

Son comportement est similaire à `default`, mais si et seulement si la variable est nul (`None`). Ainsi, une chaîne vide n'affichera rien.

Code : Jinja

```
Bonjour {{ ""|default_if_none:"visiteur" }} renvoie "Bonjour "
```

dictsort

Trie une liste de dictionnaires selon la clé donnée en argument. Par exemple, `{{ value|dictsort:"age" }}` sur cette liste :

Code : Python

```
[
    {'nom': 'Pierre', 'age': 23},
    {'nom': 'Mathieu', 'age': 17},
    {'nom': 'Maxime', 'age': 19}
]
```

renvoie la liste suivante :

Code : Python

```
[
    {'nom': 'Mathieu', 'age': 17},
    {'nom': 'Maxime', 'age': 19},
    {'nom': 'Pierre', 'age': 23}
]
```

dictsortreversed

Ce filtre a un comportement similaire à *dictsort*, sauf que le classement est décroissant. Avec l'exemple ci-dessus, le résultat serait

Code : Python

```
[
    {'nom': 'Pierre', 'age': 23},
    {'nom': 'Maxime', 'age': 19},
    {'nom': 'Mathieu', 'age': 17}
]
```

divisibleby

Renvoie `True` si la valeur est divisible par l'argument fourni.

Code : Jinja

```
{% if 21|divisibleby:3 %} 21 est divisible par 3 ! {% endif %}
{% if 22|divisibleby:3 %} 22 est divisible par 3 ! {% endif %}
```

...n'affiche que 21 est divisible par 3 !

escape

Convertit certains caractères en entités HTML. Les remplacements effectués sont les suivants :

Caractère	Entité
<	<
>	>
'	'
"	"
&	&

La conversion n'est effectuée qu'à l'**affichage**. Ainsi, la position du filtre n'importe pas lors de l'enchaînement de filtre : `escape` sera toujours le dernier filtre exécuté. Si vous souhaitez effectuer ce filtre avant un autre, le filtre *force_escape* sera plus adapté. Il est possible d'automatiser l'utilisation d'espace via la configuration de l'*auto-escaping*.

escapejs

Convertit les caractères utilisés dans les chaînes de caractères en Javascript. Ce filtre ne sécurise pas l'utilisation de la variable

dans du HTML, mais protège des erreurs de syntaxe possible lors de la génération de JSON par exemple.

Code : Jinja

```
{{ "Ma <b>super</b> variable"|escapejs }} retourne "Ma
\\u003Cb\\u003super\\u003C/b\\u003E variable.
```

filesizeformat

Formate la valeur donné de façon a obtenir une expression d'une taille de fichier facilement lisible.

Code : Jinja

```
{{ 1000|filesizeformat }}<br />
{{ 2048|filesizeformat }}<br />
{{ 123456789|filesizeformat }}<br />
{{ 2000000000000000|filesizeformat }}
```

Résultat :
1000 bytes
2.0 KB
117.7 MB
181.9 TB



Comme le montre l'exemple, la convention utilisée est 1ko = 1024 bytes

first

Retourne le premier élément d'une liste.

Code : Jinja

```
{{ notes|first }} avec notes = [4, 6, 9] renvoie 4
```

fix_ampersands

Remplace toutes les esperluettes (caractère &) par son équivalent en entité HTML, &.

Ainsi, {{ "Tic & Tac"|fix_ampersands }} deviendra "Tic & Tac". Cependant, cette opération ne sera pas faite sur les esperluettes collées au texte. Par exemple, si la valeur est "François" restera tel quel : le ç ne sera pas remplacé par &ccedil;. Cela peut d'ailleurs vous causer des soucis dans certains cas. V&B; sera par exemple conservé comme étant V&B; et non V&B; puisque &B; peut être considéré comme une entité HTML



Les esperluettes sont automatiquement échappé via la configuration par défaut, ce filtre ne devrait donc pas vous servir souvent

floatformat

floatformat peut être utilisé de trois façon distinctes :

1. Sans argument

Arrondi la chaine a une décimale après la virgule, hormis si le nombre est entier.

valeur	Template	Résultat
34.23234	{{ valeur floatformat }}	34.2
34.00000	{{ valeur floatformat }}	34
34.26000	{{ valeur floatformat }}	34.3

2. Avec un argument positif

Arrondi le nombre avec x chiffres après la virgule, où x est l'argument donné au filtre

valeur	Template	Résultat
34.23234	{{ valeur floatformat:3 }}	34.232
34.00000	{{ valeur floatformat:3 }}	34.000
34.26000	{{ valeur floatformat:3 }}	34.260

3. Avec un argument négatif

Arrondi le nombre avec x chiffres après la virgule, sauf si le nombre est entier.

valeur	Template	Résultat
34.23234	{{ valeur floatformat:-3 }}	34.232
34.00000	{{ valeur floatformat:-3 }}	34
34.26000	{{ valeur floatformat:-3 }}	34.260



Utilisez `floatformat` sans argument revient à la même chose qu'utiliser le filtre `avec` comme argument `-1`

force_escape

Applique le filtre `escape` sur la chaîne **immédiatement**. Ce filtre est utile dans certains cas lorsque vous voulez faire la conversion avant l'exécution d'un autre filtre, et avant l'affichage.

get_digit

Retourne la décimale souhaitée de la partie entière du nombre donné. Si un des arguments n'est pas valide (si la variable ou l'argument n'est pas un nombre ou si l'argument est inférieur à 1), alors la valeur retournée est la variable originale.

Code : Jinja

```
{{ 12345|get_digit:2 }} retourne 4
{{ 12345.6789|get_digit:2 }} retourne 4
```

iriencode

Convertit un identificateur de ressource internationalisé (IRI) en une chaîne de caractères pouvant être inclut dans un document HTML. Ce filtre est nécessaire si vous souhaitez utiliser des chaînes contenant des caractères non-ASCII dans une URL.

Code : Jinja

```
{{ "?arg=1&page=3"|iriencode }} renvoie "?arg=1&page=3"
```

join

Concatène tous les éléments d'une liste, en séparant chaque élément par la chaîne donnée en paramètre.

Avec `prenoms = ['Maxime', 'Mathieu', 'Pierre']`,

Code : Jinja

```
"{{ prenoms|join:", " }} sont amis" renvoie "Maxime, Mathieu, Pierre  
sont amis"
```

last

Retourne le dernier élément d'une liste.

Code : Jinja

```
{{ notes|last }} avec notes = [4, 6, 9] renvoie 9
```

length

Retourne la taille de la valeur. Cela peut être le nombre de lettres dans une chaîne de caractères mais aussi le nombre d'éléments d'une liste

Code : Jinja

```
{{ "abc"|length }} retourne 3  
{{ notes|length }} retourne 3 si notes = [4, 6, 9]
```

length_is

Renvoie vrai si la variable donnée à la taille qui est fourni en paramètre.

Code : Jinja

```
{{ "abc"|length_is:3 }} retourne True
```

linebreaks

Remplace les sauts à la ligne dans un texte par les balises HTML correspondantes. En cas de simple saut à la ligne, le `\n` est remplacé par un `
`. Si deux sauts à la ligne se suivent (provoquant une ligne blanche), alors un nouveau paragraphe est ouvert.

Soit `texte = "Je suis un texte\ntrès très très long.\n\nNouveau paragraphe"`, alors `{{ texte|linebreaks }}` retourne :

Code : HTML

```
<p>Je suis un texte<br />très très très long.</p>

<p>Nouveau paragraphe</p>
```

linebreaksbr

Comportement similaire à `linebreaks`. Cependant tous les `\n` sont remplacés par des `
`. Il n'y a pas d'idées de paragraphes avec ce filtre. En reprenant l'exemple précédent, `{{ texte|linebreaksbr }}` retourne :

Code : HTML

```
Je suis un texte<br />très très très long.<br /><br />Nouveau
paragraphe
```

linenumbers

Permet d'afficher le numéro de la ligne à chaque saut de ligne. Avec l'exemple précédent, `{{ texte|linenumbers|linebreaks }}` devient

Code : Autre

```
1. Je suis un texte
2. très très très long.
3.
4. Nouveau paragraphe
```



N'oubliez pas qu'il est possible de combiner les filtres, comme ici !

ljust

Aligne à gauche une valeur sur un certain nombre de caractères. Si la chaîne est plus longue que l'argument fourni, alors le filtre renvoie la valeur tel quel.

Code : Jinja

```
"{{ "Maxime"|ljust:10 }}" renvoie "Maxime "
"{{ "Maxime et Mathieu"|ljust:10 }}" renvoie "Maxime et Mathieu"
```

lower

Affiche tous la variable avec l'ensemble de ses caractères en minuscules

Code : Jinja

```
{{ "Bonjour TOUT le MonDe"|lower }} retourne "bonjour tout le monde"
```

make_list

Retourne la valeur sous forme de liste de caractères. Pour un nombre, l'argument est converti en caractères unicode puis mis sous forme de liste.

Code : Jinja

```
{{ "abc"|make_list }} renvoie [u'a', u'b', u'c']  
{{ 123|make_list }} renvoie [u'1', u'2', u'3']
```

phone2numeric

Convertit un numéro de téléphone, pouvant contenir des lettres, en son équivalent numérique. L'entrée n'est pas nécessairement un numéro de téléphone valide.

Code : Jinja

```
{{ "3615-LOLA"|phone2numeric }} renvoie 3615-5652
```

pluralize

Permet d'accorder un mot en fonction d'un nombre. Le filtre renvoie "s" si la variable est différent de 1.

Ily a {{ nb_chapitres }} chapitre {{ nb_chapitres|pluralize }} renverra "Il y a 1 chapitre", "Il y a 2 chapitres", etc.

Pour les mots dont le pluriel est particulier, il est possible de définir le suffixe du mot via un argument au filtre. Ainsi, {{ nb_chevaux|pluralize:"al", "aux" }} renvoie cheval si la variable vaut 1 et chevaux dans les autres cas.

pprint

Affichage identique au module [pprint](#). A n'utiliser que pour du débogage, afin de vous aider à détecter vos erreurs.

random

Retourne une valeur aléatoire de la liste.

Si `participants = ['Maxime', 'Mathieu', 'Pierre']`, alors {{ participants|random }} retournera un de ces 3 prénoms.

removetags

Supprime tous les éléments (x)HTML souhaité.

Code : Jinja

```
{{ "<strong>Gras</strong>, <em>Italique</em>,  
<h1>titre</h1>"|removetags:"em h1" }} retourne  
"<strong>Gras</strong>, Italique, titre"
```



Ce filtre est sensible à la casse. Ainsi, si l'argument de `removetags` avait été "EM h1", alors les balises `...` n'aurait pas été enlevé.

rjust

Tout comme *ljust*, ce filtre aligne à droite le texte, selon une certaine taille.

Code : Jinja

```
"{{ value|rjust:"10" }}" renvoie " Django".
```

safe

Permet de signaler qu'il n'est pas nécessaire d'échapper les balises HTML. Ainsi il est possible d'insérer du code HTML et de le faire afficher par le navigateur. Quand l'auto échappement est activé, ce filtre n'a aucun effet

safeseq

Applique le filtre safe à chaque élément d'une liste. Utile si vous manipulez une QuerySet par exemple.

Code : Jinja

```
{{ mes_articles|safeseq }}
```

slice

Retourne une sous-partie d'une liste. La syntaxe est similaire à celle de Python : [http://www.diveintopython.net/native_d \[...\]](http://www.diveintopython.net/native_d[...]er.list.slice)
[er.list.slice](#)

Code : Jinja

```
{{ notes|slice:":3" }}
```

Si notes est la liste [12, 10, 20, 18], alors le résultat sera [12, 10, 20]

slugify

Convertit une chaîne de caractères en enlevant tous les caractères non alphanumériques (excepté les underscores), en forçant toute la chaîne en minuscule et en remplaçant les espaces par des tirets.

Cette méthode est utile lorsque vous voulez produire des URLs composées de variables tel que le titre d'un article.

Code : Jinja

```
"{{ "Je suis un zéro"|slugify }}" renvoie "je-suis-un-zero"
```

stringformat

Converti un objet en chaîne de caractères, selon les règles de conversion Python :

<http://docs.python.org/library/stdtypes.html#string-formatting-operations>, avec une exception : le symbole % est ignoré.

Code : Jinja

```
{{ 3.00000|stringformat:".3f" }} renvoie 3.000<br />
{{ 3.00000|stringformat:"d" }} renvoie 3<br />
{{ 3.00000|stringformat:"s" }} renvoie 3.0 -- Représentation de
3.00000 via la méthode str()
```

striptags

Supprime tous les caractères (x)HTML présents.

Code : Jinja

```
{{ "Ma chaine <strong>très</strong> riche en <acronym
title=\"HyperText [...]\">HTML</acronym>"|striptags }} renvoie "Ma
chaine très riche en HTML"
```

time

Formate l'heure au format voulu. Une constante est défini par Django, dépendant de la langue utilisée dans le settings.py : TIME_FORMAT.

Par exemple, avec `date = datetime.now()` :

Code : Jinja

```
{{ date|time:"TIME_FORMAT" }} renvoie "8:14 p.m", avec LANGUAGE_CODE
= 'en-us'
{{ date|time:"TIME_FORMAT" }} renvoie "20:15:01", avec LANGUAGE_CODE
= 'fr-fr'
```

Il est possible de donner le format souhaité en argument, à la place de TIME_FORMAT :

Code : Jinja

```
{{ date|time:"H:i" }} renvoie "20:15"
```



Ce filtre ne s'occupe que de l'heure, et non des dates ! Pour cette raison, il est évident que le filtre n'accepte que des arguments sur l'heure d'une journée. Pour afficher une date complète, référez-vous au filtre date

timesince

Renvoie le temps écoulé entre 2 dates. Si aucun argument n'est donné, la comparaison se fera avec la date actuelle. La plus petite unité de comparaison sera la minute. Si la comparaison se fait avec une date future, alors le résultat retournée sera toujours "0 minutes"

Par exemple, voici notre vue :

Code : Python

```
def filter_timesince(request):
    new_year = datetime(2012, 1, 01)
    valentine_day = datetime(2012, 2, 14)
    last_hour = datetime(2012, 7, 9, 19, 18)
```

```
return render_to_response('blog/index.html', locals())
```

On peut alors manipuler `timesince` de plusieurs façon :

Code : Jinja

```
Le nouvel an, c'était il y a {{ new_year|timesince }}<br />
Le nouvel an et la St-Valentin sont séparés de {{
new_year|timesince:valentine_day }}<br />
J'ai commencé à travailler il y a {{ last_hour|timesince }}

... on obtient le résultat suivant (au 9 juillet 2012):

Le nouvel an, c'était il y a 6 mois, 1 semaine
Le nouvel an et la St-Valentin sont séparés de 1 mois, 2 semaines
J'ai commencé à travailler il y a 1 heure, 4 minutes
```



Le résultat ici est donné avec `LANGUAGE_CODE = 'fr-fr'`, vérifiez bien ce paramètre si vous obtenez "6 months, 1 week" par exemple

timeuntil

Filtre similaire à `timesince`, mais qui compare des dates futures. Il peut être utilisé pour savoir combien de temps il reste avant un événement :

Code : Python

```
def filter_timeuntil(request):
    new_year      = datetime(2013, 1, 1)
    conference    = datetime(2012, 7, 13, 10, 0)
    christmas     = datetime(2012, 12, 25)

    return render_to_response('blog/index.html', locals())
```

Code : Jinja

```
Le nouvel an, c'est dans {{ new_year|timeuntil }}<br />
Je serais en conférence dans {{ conference|timeuntil }}<br />
Entre Noël et le nouvel an, il y a {{ new_year|timeuntil:christmas
}}

on obtient (au 9 juillet 2012) :

Le nouvel an, c'est dans 5 mois, 3 semaines
Je serais en conférence dans 3 jours, 13 heures
Entre Noël et le nouvel an, il y a 1 semaine
```

title

Met en majuscule la première lettre de chaque mot de la chaîne de caractères.

Code : Jinja

```
{{ "bonjour tout le monde"|title }} devient "Bonjour Tout Le Monde"
```

truncatechars

Coupe une chaîne de caractères au nombre de lettres souhaitées. Après la troncature, des points de suspension sont ajoutés (...)

Code : Jinja

```
{{ "bonjour tout le monde"|truncatechars:8 }} devient "bonjour..."
```



Notez que le caractère ajouté est le caractère "Point de suspension", et non 3 points !

truncatewords

Similaire à *truncatechars*, cependant la troncature s'effectue au nombre de mots et non au caractère.

Code : Jinja

```
{{ "bonjour tout le monde"|truncatewords:2 }} devient "bonjour tout..."
```

truncatewords_html

Similaire à *truncatewords*, mais prend en compte la troncature des balises HTML. Si un tag est ouvert avant la troncature mais n'est pas fermé, alors la fermeture du tag a lieu immédiatement après la troncature.

Ce filtre est plus lent que *truncatewords*, il n'est à utiliser que si vous êtes certains de passer du HTML.

Code : Jinja

```
{{ "<strong>bonjour</strong> tout le monde"|truncatewords_html:2 }}  
devient <strong>bonjour</strong> tout ...  
{{ "<strong>bonjour tout le</strong> monde"|truncatewords_html:2 }}  
devient <strong>bonjour tout ...</strong>
```

unordered_list

Affiche une liste (pouvant contenir des sous-listes) sous formes de listes HTML, sans ouvrir, ni fermer, les balises ``. Si on a `pays = ['France', ['Bretagne', ['Morbihan', 'Ille et Vilaine', 'Finistere', 'Cotes d'Armor']], 'Belgique', ['Wallonie', 'Bruxelles', 'Flandre']]` alors on a :

Code : Jinja

```
{{ pays|unordered_list }} qui renvoie :  
<li>France  
<ul>  
<li>Bretagne  
<ul>  
<li>Morbihan</li>
```

```

<li>Ille et Vilaine</li>
<li>Finistere</li>
<li>Cotes d&#39;Armor</li>
</ul>
</li>
</ul>
</li>
<li>Belgique
<ul>
<li>Wallonie</li>
<li>Bruxelles</li>
<li>Flandre</li>
</ul>
</li>

```

upper

Met une chaîne de caractère tout en majuscules.

Code : Jinja

```
{{ "bonjour"|upper }} renvoie "BONJOUR"
```

urlencode

Échappe la chaîne de caractère, afin de pouvoir être utilisée dans une URL. Il est possible de donner en argument la liste des caractères à ne pas échapper.

Code : Jinja

```

{{ "cle_secrete:uHI6&cg_I="|urlencode }} devient
cle_secrete%3AuhI6%26cg_I%3D
{{ "cle_secrete:uHI6&cg_I="|urlencode:"=" }} devient
cle_secrete%3AuhI6%26cg_I=

```

urlize

Transforme une URL, contenu dans une chaîne de caractère, en un texte cliquable. Le filtre ne marche que si la valeur commence par http://, https://, ou www.

Ce filtre fonctionne toutefois pour les domaines seuls, si ils finissent par l'un des domaines suivant : .com, .edu, .gov, .int, .mil, .net ou .org.

Si la méthode est appliquée sur une URL comportant déjà le tag HTML `<a>`, le résultat peut être surprenant parfois...

Code : Jinja

```

{{ "Je ne suis pas une URL"|urlize }} renvoie "Je ne suis pas une
URL"
{{ "http://www.siteduzero.com"|urlize }} renvoie "<a
href='http://www.siteduzero.com'
rel='nofollow'>http://www.siteduzero.com</a>"
{{ "http://www.siteduzero.com/forums.html"|urlize }} renvoie "<a
href='http://www.siteduzero.com/forums.html'
rel='nofollow'>http://www.siteduzero.com/forums.html</a>"
{{ "siteduzero.com"|urlize }} renvoie "<a
href='http://siteduzero.com' rel='nofollow'>siteduzero.com</a>"
{{ "france.fr"|urlize }} renvoie "france.fr"

```


urlizetrunc

De la même façon qu'*urlize*, ce filtre convertit une url en chaîne cliquable, mais tronque l'URL affichée (pas celle contenu dans l'attribut href) à un certain nombre de caractères. Comme pour *urlize*, il est préférable d'appliquer ce filtre sur du texte uniquement.

Code : Jinja

```
{{ "http://www.siteduzero.com"|urlizetrunc:30 }} renvoie "<a  
href="http://www.siteduzero.com"  
rel="nofollow">http://www.siteduzero.com</a>"  
{{ "http://www.siteduzero.com/forums.html"|urlizetrunc:30 }}<br />  
renvoie "<a href="http://www.siteduzero.com/forums.html"  
rel="nofollow">http://www.siteduzero.com/f...</a>"
```

wordcount

Renvoie le nombre de mots présents dans la variable

Code : Jinja

```
{{ "Bonjour tout le monde"|wordcount }} retourne 4
```

wordwrap

Force le saut à la ligne à partir d'un certain nombre de caractères, sans couper un mot.

Code : Jinja

```
{{ "UnTresTresLongMot abcde fgh ijklmn"|wordwrap:5 }}
```

renvoie :

```
UnTresTresLongMot  
abcde fgh  
ijklmn
```

yesno

Permet d'afficher une valeur particulière en fonction de la variable à **True**, **False** et **None**. Par défaut, les valeurs associés sont "yes", "no" et "maybe" ("oui", "non" et "peut-être" si vous avez réglé Django en français). Il est possible de passer un ensemble particulier. Si aucun mapping n'est donné pour la valeur None, celle-ci est substituée par la valeur False automatiquement.

Un tableau pour mieux comprendre :

valeur	Filtre	Résultat
True	{{ valeur yesno }}	oui
True	{{ valeur yesno:"Bien sûr,NON !,Euh..." }}	Bien sûr
False	{{ valeur yesno }}	non
None	{{ valeur yesno }}	peut-être

None	{{ valeur yesno:"Correct,Faux!" }}	Faux!
------	------------------------------------	-------