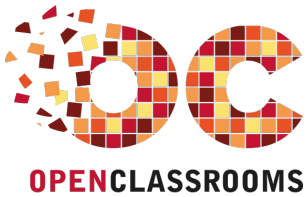
Développez votre site web avec le framework Django

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



www.openclassrooms.com

Dernière mise à jour le 2/04/2013

Sommaire

Développez votre site web avec le framework Django

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

Mise à jour : 02/04/2013
Difficulté : Intermédiaire Durée d'étude : 15 jours

Django

« Le framework web pour les perfectionnistes sous pression »

En quelques années, les sites web n'ont cessé d'évoluer. Ils requièrent désormais des développements longs et acharnés, 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 dans un cadre professionnel ou personnel, les attentes sont de plus en plus lourdes.

C'est de ce constat qu'est né Django : proposer *un développement plus efficace et plus rapide* d'une application dynamique web, tout en conservant la qualité ! Ce cours vous apprendra 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 orientée objet et des expressions régulières ;
- HTML/CSS : maîtrise de toute la partie HTML (nous ne parlerons pas de CSS).

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

Ce cours porte sur **la version 1.5 de Django**, et n'assure nullement que toutes les méthodes présentées fonctionneront forcément sur des versions antérieures ou postérieures.

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

Si vous lisez ceci, c'est que vous avez décidé de vous lancer dans l'apprentissage de **Django**. Avant de commencer, des présentations s'imposent : Django est un **framework web** écrit en Python, qui se veut complet tout en facilitant la création d'applications web riches.

Avant de commencer à écrire du code, nous allons tout d'abord voir dans ce chapitre ce qu'est un framework en général, et plus particulièrement ce qu'est Django. Dans un second temps, nous verrons comment l'installer sur votre machine, pour pouvoir commencer à travailler ! Est-il utile de vous rappeler encore ici qu'*il est nécessaire d'avoir les bases en Python* pour pouvoir commencer ce cours ?

Qu'est-ce qu'un framework ?

Un framework est un ensemble d'outils qui simplifie le travail d'un développeur. Traduit littéralement de l'anglais, un framework est un « cadre de travail ». Il apporte les bases communes à la majorité des programmes ou des sites web. Celles-ci étant souvent identiques (le fonctionnement d'un espace membres est commun à une très grande majorité de sites web de nos jours), un développeur peut les réutiliser simplement et se concentrer sur les particularités de son projet.

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

Quels sont les avantages d'un framework ?

Un framework instaure en quelque sorte sa « ligne de conduite ». Tous les développeurs Django codent de façon assez homogène (leurs codes ont le même fonctionnement, les mêmes principes). De ce fait, lorsqu'un développeur rejoint un projet utilisant un framework qu'il connaît déjà, il comprendra très vite ce projet et pourra se mettre rapidement au travail.

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

Voici d'ailleurs un grand défi des frameworks : bien que ceux-ci doivent instaurer une structure commune, ils doivent aussi être souples et modulables, afin de pouvoir être utilisés pour une grande variété de projets, du plus banal au plus exotique. Autrement, leur intérêt 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 en manier un, mais ce temps d'apprentissage est largement récupéré par la suite, vu la vitesse de développement qui peut parfois être décuplée. Nous pourrions éventuellement dire que certains frameworks 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, nous pouvons compter d'autres frameworks Python du même genre comme web2py, TurboGears, CherryPy ou Zope. Il a cependant le mérite d'être le plus exhaustif, d'automatiser un bon nombre de choses et de disposer d'une très grande communauté.

Le logo de Django

Django est né en 2003 dans une agence de presse qui devait développer des sites web complets 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 !

Logos d'Instagram, de la NASA et de Pinterest

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 conservant un 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 pratiques. Il est donc inutile de perdre son temps à écrire directement des requêtes destinées à la base de données, car Django le fait automatiquement. De plus, d'autres bibliothèques complètes et bien pensées sont disponibles, comme un espace membres, ou une bibliothèque permettant la traduction de votre application web en plusieurs langues.

Une communauté à votre service

Évidemment, Django dispose des avantages de tous les frameworks en général. Il 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.

Encore un point, et non des moindres, 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 cours si vous voulez approfondir un certain sujet ou si certaines zones d'ombre 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 code réutilisables par n'importe qui. Un site est dédié à ces *snippets*. 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 y trouverez souvent votre bonheur !

Téléchargement et installation

Maintenant que nous avons vu les avantages qu'apporte Django, il est temps de passer à son installation. Tout d'abord, assurez-vous que vous disposez bien d'une **version de Python supérieure ou égale à la 2.6.5 pour la branche 2.6.x ou à la 2.7.3 pour la branche 2.7.x** et supérieure. Pour plus d'informations à ce sujet, vous pouvez vous reporter au cours sur le Python du Site du Zéro.

Django 1.5 est également compatible avec Python 3, mais de façon *expérimentale* : quelques modules, comme la connexion avec MySQL sont indisponibles, faute de bibliothèque compatible Python 3... Nous vous recommandons donc d'attendre Django 1.6 avant de sauter vers Python 3 pour vos applications web.
Par ailleurs, le support de Python 2.5 est abandonné depuis Django 1.5. Nous vous conseillons dès maintenant d'utiliser Python 2.7.3, 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 1.5 sur votre machine, à part si vous avez déjà des applications en production sur des versions antérieures. Dans ce cas, il est conseillé soit de porter toutes vos applications pour Django 1.5, soit d'exécuter vos deux projets avec deux versions de Django bien indépendantes.

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. Si toutefois vous choisissez cette solution, 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, 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. Il suffit ensuite de l'extraire et de l'installer, en effectuant les commandes suivantes dans une console :

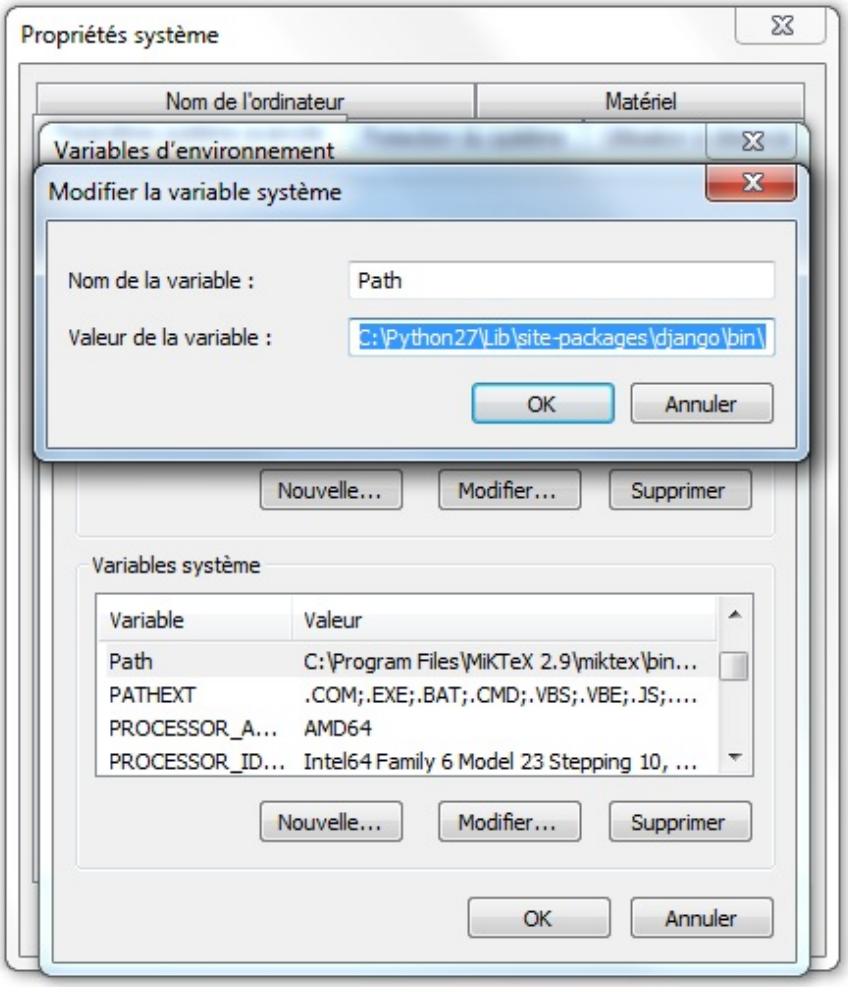
Code : Console

```
tar xzvf Django-1.5.tar.gz
cd Django-1.5
sudo python setup.py install
```

Windows

Contrairement aux environnements UNIX, l'installation de Django sous Windows requiert quelques manipulations supplémentaires. Téléchargez l'archive de Django et extrayez-la. 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, comme indiqué à la figure suivante.



Édition du Path sous Windows 7

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

Code : Console

```
cd C:\Users\<nom_d'utilisateur>\Downloads\django1.5  # A adapter à votre répertoire
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 nouvelle console Windows, puis lancez l'interpréteur Python (via la commande `python`) et tapez les deux lignes suivantes :

Code : Python

```
>>> import django
>>> print django.get_version()
1.5 # <- Résultat attendu
```

Si vous obtenez également 1.5 comme réponse, félicitations, vous avez correctement installé Django !

Il se peut que vous obteniez un numéro de version légèrement différent (du type 1.5.1). En réalité, Django est régulièrement mis à jour de façon mineure, afin de résoudre des failles de sécurité ou des bugs. Tenez-vous au courant de ces mises à jour, et appliquez-les dès que possible.

Dans la suite de ce cours, nous utiliserons SQLite, qui est simple et déjà inclus dans les bibliothèques 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 (dépendances, packages, etc.).

En résumé

- Un framework (cadre de travail en français) est un ensemble d'outils qui simplifie le travail d'un développeur.
- Un framework est destiné à des développeurs, et non à des novices. Un framework nécessite un temps d'apprentissage avant de pouvoir être pleinement utilisé.
- Django est un framework web pour le langage Python très populaire, très utilisé par les entreprises dans le monde : Mozilla, Instagram ou encore la NASA l'ont adopté !
- Ce cours traite de la version 1.5, sortie en février 2013. Nous ne garantissons pas que les exemples donnés soient compatibles avec des versions antérieures et postérieures.

Le fonctionnement de Django

Attaquons-nous au vif du sujet ! Dans ce chapitre, théorique mais fondamental, nous allons voir comment sont construits la plupart des frameworks grâce au modèle MVC, nous aborderons ensuite les spécificités du fonctionnement de Django et comment les éléments d'une application classique Django s'articulent autour du modèle MVT, que nous introduirons également. En dernier lieu, nous expliquerons le système de projets et d'applications, propre à Django, qui permet une séparation nette, propre et précise du code.

Au terme de ce chapitre, vous aurez une vue globale sur le fonctionnement de Django, ce qui vous sera grandement utile lorsque vous commencerez à créer vos premières applications.

Un peu de théorie : l'architecture MVC

Lorsque nous parlons 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), nous parlons 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 ») **M**odèle-**V**ue-**C**ontrôleur est composé de trois entités distinctes, chacune ayant son propre rôle à remplir.

Tout d'abord, le modèle *représente une information* enregistrée quelque part, le plus souvent dans une base de données. 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, comme nous le verrons par la suite.

Ensuite la vue qui est, comme son nom l'indique, la *visualisation de l'information*. C'est la seule chose que l'utilisateur peut voir. 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, un exemple de vue est une page web, ni plus, ni moins.

Finalement, le contrôleur *prend en charge tous les événements de l'utilisateur* (accès à une page, soumission d'un formulaire, etc.). 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 de savoir ce qu'il est nécessaire d'afficher.

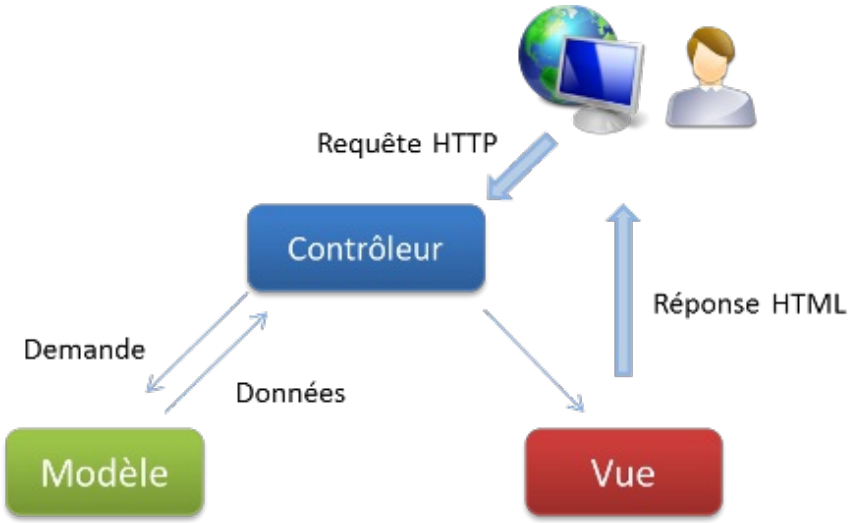


Schéma de l'architecture MVC

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…). Ainsi, nous parlons plutôt de framework utilisant l'architecture **MVT** : **M**odèle-**V**ue-**T**emplate.

Cette architecture reprend les définitions de modèle et de vue que nous avons vues, et en introduit une nouvelle : le **template** (voir figure suivante). Un template est un fichier HTML, aussi appelé en français « gabarit ». Il sera récupéré par la vue et envoyé au visiteur ; cependant, avant d'être envoyé, il 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, d'utiliser des structures conditionnelles (`if/else`) ou encore des boucles (`for`), etc.

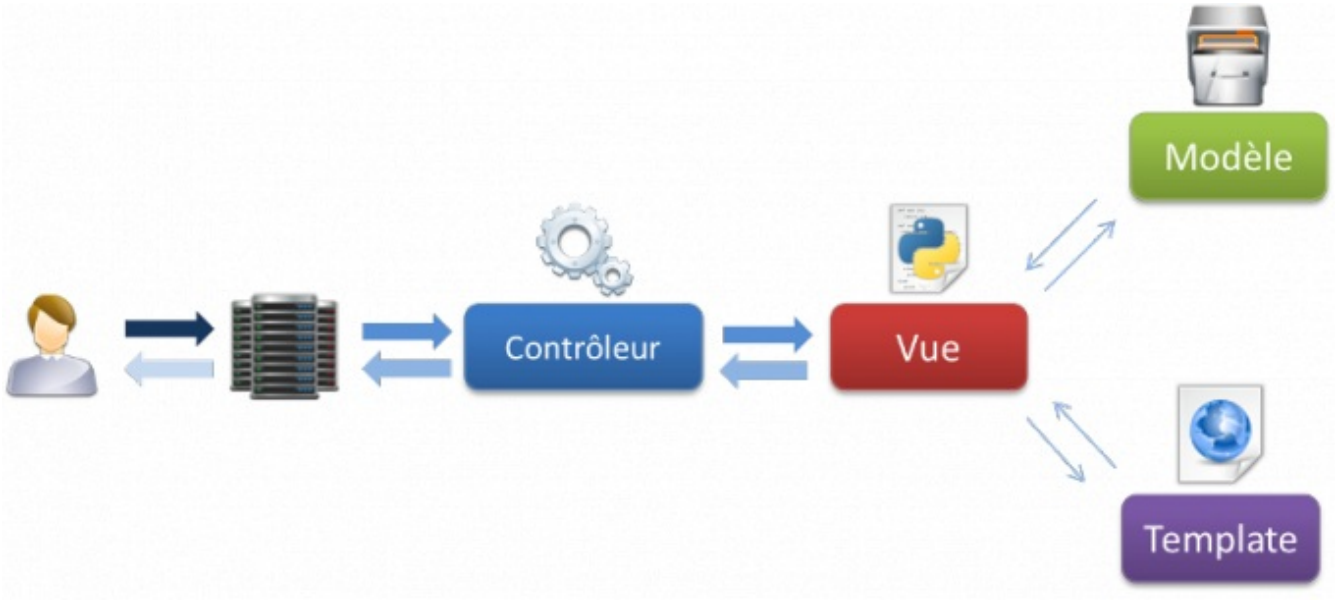


Schéma d'exécution d'une requête

Concrètement, lorsque l'internaute appelle une page de votre site réalisé avec Django, 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 au navigateur de l'internaute.

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

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

On en revient donc au modèle **MVT**. Le développeur se doit de fournir le modèle, la vue et le template. Une fois cela 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 celle-ci 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 pour interagir avec la base de données, 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 aux attributs de la classe pour accéder aux éléments dans la table et pouvoir les modifier, 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 donnés dans l'URL. Par exemple, si l'identifiant ou le nom d'un article du blog a été donné dans l'URL `crepes-bretonnes.com/blog/faire-de-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

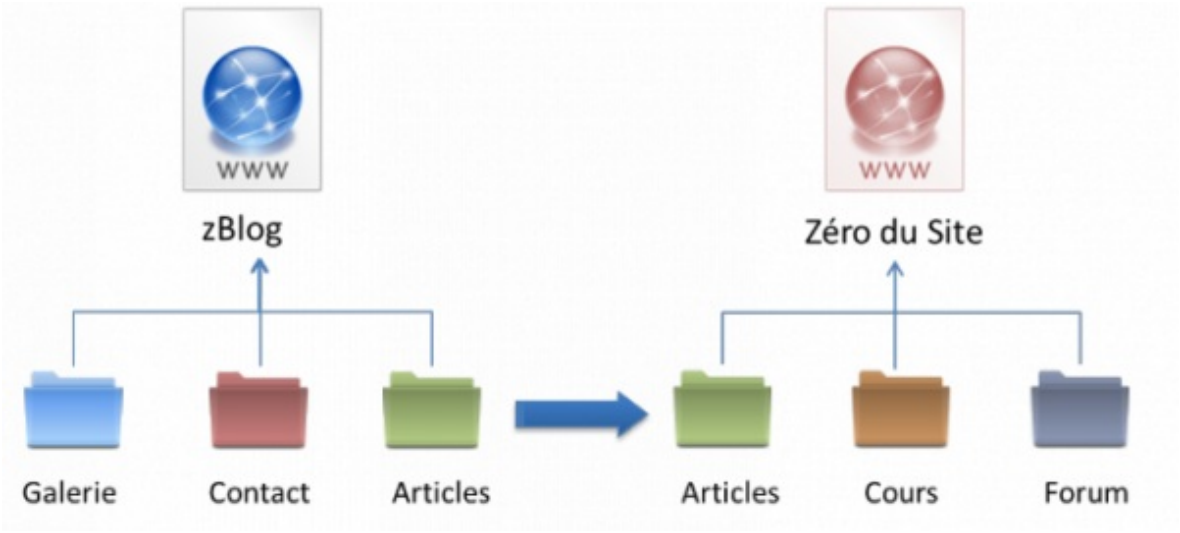
En plus de l'architecture MVT, Django introduit le développement d'un site sous forme de projet. Chaque site web conçu avec Django est considéré comme un projet, composé de plusieurs applications. Une application consiste en un dossier contenant plusieurs fichiers de code, chacun étant relatif à une 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 :

- Un module d'actualités ;
- Un forum ;
- Un système de contact ;
- Une galerie de photos ;
- Un 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, nous gardons donc une séparation nette entre les différentes 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 copier/coller, comme le montre la figure suivante.



Organisation d'un projet Django et réutilisation d'une application

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 nous agençons ensemble, accélérant considérablement le développement pour les projets qui suivent.

En résumé

- Django respecte l'architecture MVT, directement inspirée du très populaire modèle MVC ;
- Django gère de façon autonome la réception des requêtes et l'envoi des réponses au client (partie contrôleur) ;
- Un projet est divisé en plusieurs applications, ayant chacune un ensemble de vues, de modèles et de schémas d'URL ;
- Si elles sont bien conçues, ces applications sont réutilisables dans d'autres projets, puisque chaque application est indépendante.

Gestion d'un projet

Django propose un outil en ligne de commandes 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 de gestion fourni avec Django se nomme `django-admin.py` et il n'est accessible qu'en ligne de commandes. 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 système 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 > 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`, suivie 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/mathx/Projets/
```

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 :

Code : Autre

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

Il s'agit de votre projet.

Dans le dossier principal `crepes_bretonnes`, nous retrouvons 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.

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

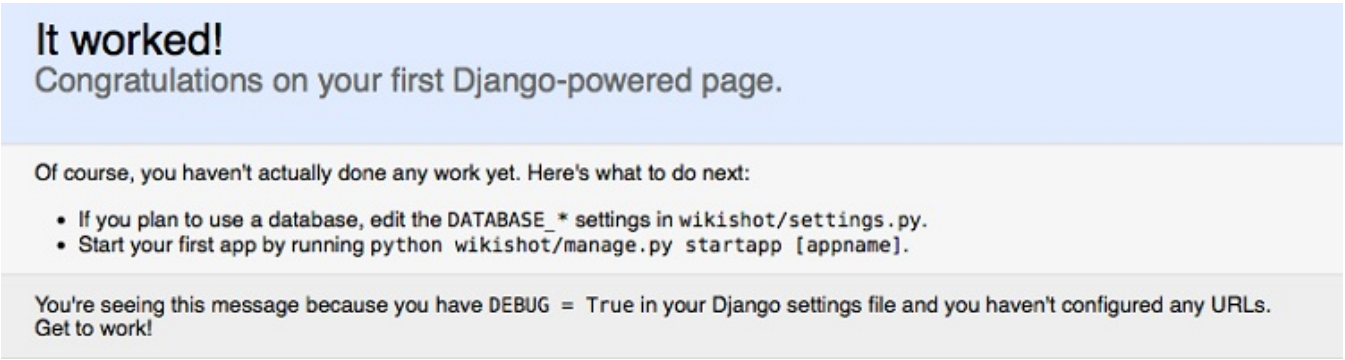
Ensuite, le fichier `manage.py` est en quelque sorte un raccourci local de la commande `django-admin.py` qui prend en charge la configuration de votre projet. Vous pouvez désormais oublier la commande `django-admin.py`, elle ne sert en réalité qu'à créer des projets, tout le reste se fait via `manage.py`.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

Votre projet étant créé, pour vous assurer que tout a été correctement effectué jusqu'à 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
March 04, 2013 - 20:31:54
Django version 1.5, using settings 'crepes_bretonnes.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Cette console vous donnera des informations, des logs (quelle page a été accédée et par qui) et les exceptions de 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 la figure suivante dans votre navigateur :



Votre première page Django

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

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

Code : Console

```
python manage.py help
```

Toutes ces commandes sont expliquées dans une annexe, donc nous vous invitons à la survoler de temps en temps, au fur et à mesure que vous avancez dans ce cours, et nous reviendrons sur certaines d'entre elles dans certains chapitres. Il s'agit là d'un outil très puissant qu'il ne faut surtout pas sous-estimer. Le développeur Django y a recours quasiment en permanence, d'où l'intérêt de savoir le manier correctement.

Configurez votre projet

Avant de commencer à écrire des applications Django, configurons le projet que nous venons de créer. 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 les 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 de débogage 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`, qui est par défaut vide, est censé contenir quelques informations à propos des gestionnaires du site (nom et adresse e-mail). L'adresse e-mail servira notamment à envoyer les erreurs rencontrées par les visiteurs de votre site en production. En voici un exemple :

Code : Python

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

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

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/"
)
```

Finalement, pour des raisons pratiques qui seront explicitées par la suite, ajoutons une option qui permet de compléter automatiquement les URL par un slash (« / ») à la fin de celles-ci, si celui-ci n'est pas déjà présent. Vous en comprendrez l'utilité lorsque nous aborderons le routage d'URL :

Code : Python

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

Voilà ! Les variables les plus importantes ont été expliquées. Pour que ce ne soit pas indigeste, nous n'avons pas tout traité, 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 côté de la déclaration et à vous référer à la documentation en ligne.

Créons notre première application

Comme nous l'avons expliqué précédemment, un projet se compose de plusieurs applications, chacune ayant un but bien précis (système d'actualités, galerie photos…). Pour créer une application dans un projet, le fonctionnement est similaire à la création d'un projet : il suffit d'utiliser la commande `manage.py` 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 (un chapitre y est consacré dans la quatrième partie de ce cours) ;
- `views.py` contiendra toutes les vues de votre application.

À partir de maintenant, nous ne parlerons plus des fichiers `__init__.py`, qui ne 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 ajouter cette application au projet. Pour que Django considère le sous-dossier `blog` comme une application, il faut donc l'ajouter dans la configuration.

Retournez dans `settings.py`, et cherchez la variable `INSTALLED_APPS`. Tout en conservant les autres applications installées, ajoutez 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',
)
```

En résumé

- L'administration de projet s'effectue via la commande `python manage.py`. Tout particulièrement, la création d'un projet se fait via la commande `django-admin.py startproject mon_projet`.
- À la création du projet, Django déploie un ensemble de fichiers, facilitant à la fois la structuration du projet et sa configuration.
- Pour tester notre projet, il est possible de lancer un serveur de test, via la commande `python manage.py runserver`, dans le dossier de notre projet. Ce serveur de test ne doit pas être utilisé en production.
- Il est nécessaire de modifier le `settings.py`, afin de configurer le projet selon nos besoins. Ce fichier ne doit pas être partagé avec les autres membres ou la production, puisqu'il contient des données dépendant de votre installation, comme la connexion à la base de données.

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 base de données, 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, une appréciation, etc. Il est important que votre méthode de classement permette de différencier très proprement ces caractéristiques. De même, vous devez être sûrs que les caractéristiques que vous écrivez sont correctes et homogènes. Si vous écrivez la date de sortie une fois en utilisant des chiffres, puis une autre fois 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 renseignent les différentes caractéristiques qu'un film peut avoir, tandis que les lignes représentent toutes les caractéristiques d'un même 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 s'appuient sur cette méthode de tri pour enregistrer et classer les informations que vous spécifierez.

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

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

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

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

Voici un simple exemple de requête SQL qui renvoie toutes les entrées de la table `films` dont le réalisateur doit être Quentin Tarantino et qui 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 fonctionnera 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 par la suite.

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

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

Lors de la configuration de votre projet Django dans le chapitre précédent, nous vous avons conseillé d'utiliser SQLite. Pourquoi ? Car contrairement aux autres SGBD qui ont besoin d'un serveur lancé en permanence pour traiter les données, une base de données SQLite consiste en un simple fichier. C'est la bibliothèque Python (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 écrire ses propres requêtes est quelque chose d'assez difficile et contraignant lorsque nous débutons. Cela prend beaucoup de temps et est assez rébarbatif. Heureusement, Django propose un système pour bénéficier des avantages d'une base de données SQL sans devoir écrire ne serait-ce qu'une seule requête SQL !

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

Sans entrer 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")
```

À partir de ce modèle, Django va créer une table `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. Le code suivant…

Code : Python

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

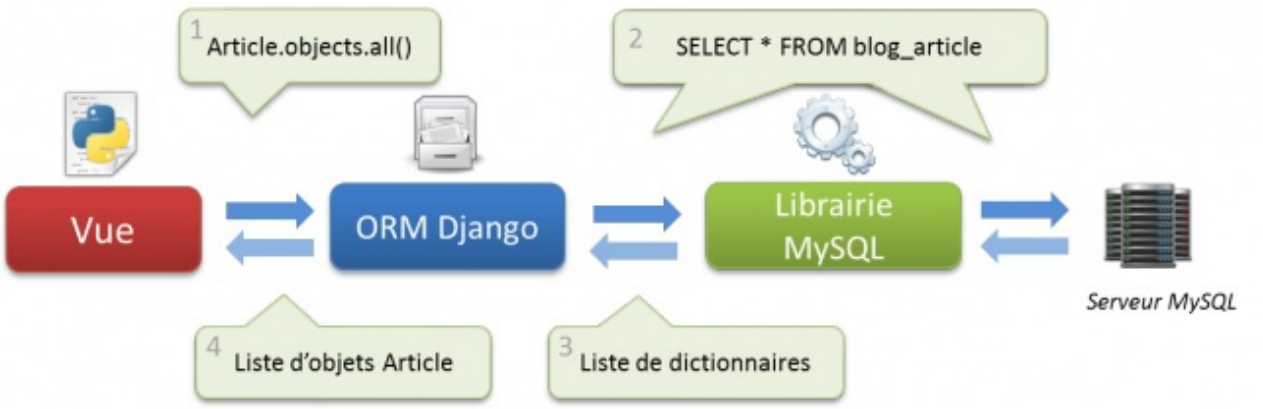
… créera une nouvelle entrée dans la base de données. Notez la relation qui se crée : chaque instance du modèle `Article` qui se crée correspond à une entrée dans la table SQL. *Toute manipulation des données* dans la base *se fait depuis des objets Python*, ce qui est bien plus intuitif et simple.

De la même façon, il est possible d'obtenir toutes les entrées de la table. Ainsi le code suivant…

Code : Python

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

… renverra des instances d'`Article`, une pour chaque entrée dans la table, comme le schématise la figure suivante :



Fonctionnement de l'ORM de Django

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 Keys* en anglais).

Reprenons notre exemple de tout à l'heure : nous avons une table qui recense plusieurs films. Imaginons maintenant que nous souhaitions ajouter des données supplémentaires, qui ne concernent pas les films mais les réalisateurs. Nous voudrions par exemple ajouter le pays d'origine et la date de naissance des réalisateurs. Étant donné que certains réalisateurs reviennent plusieurs fois, il serait redondant d'ajouter les caractéristiques des réalisateurs dans les caractéristiques des films. La bonne solution ? Créer une nouvelle table qui recensera les réalisateurs et *ajouter 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 va croissant au fil des entrées. Ainsi, le premier réalisateur ajouté aura l'identifiant 1, le deuxième l'identifiant 2, 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, une colonne a été modifiée :

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 nous souhaitons obtenir le réalisateur du film *Fight Club*, il faut aller regarder dans la table `réalisateurs` et sélectionner l'entrée ayant l'identifiant 2. Nous pouvons dès lors regarder le nom du réalisateur : nous obtenons bien à nouveau David Fincher, et les données supplémentaires (date de naissance et 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.

En résumé

- Une base de données permet de stocker vos données de façon organisée et de les récupérer en envoyant des requêtes à votre système de gestion de base de données ;
- De manière générale, nous communiquons la plupart du temps avec les bases de données via le langage SQL ;
- Il existe plusieurs systèmes de gestion de bases de données, ayant chacun ses particularités ;
- Pour faire face à ces différences, Django intègre une couche d'abstraction, afin de communiquer de façon uniforme et plus intuitive avec tous les systèmes : il s'agit de l'ORM que nous avons présenté brièvement ;
- Une ligne dans une table peut être liée à une autre ligne d'une autre table via le principe de clés étrangères : nous gardons l'identifiant de la ligne de la seconde table dans une colonne de la ligne de la première table.

Partie 2 : Premiers pas

Les bases du framework vous seront expliquées pas à pas dans cette partie. À la fin de celle-ci, vous serez capables de réaliser par vous-mêmes un site basique avec Django !

Votre première page grâce aux vues

Dans ce chapitre, nous allons créer notre première page web avec Django. Pour ce faire, nous verrons comment créer une vue dans une application et la rendre accessible depuis une URL. Une fois cela fait, nous verrons comment organiser proprement nos URL afin de rendre le code plus propre et structuré. Nous aborderons ensuite deux cas spécifiques des URL, à savoir la gestion de paramètres et de variables dans celles-ci, et les redirections, messages d'erreur, etc.

Cette partie est fondamentale pour aborder la suite et comprendre le fonctionnement du framework en général. Autrement dit, nous ne pouvons que vous conseiller de bien vous accrocher tout du long !

Hello World !

Commençons enfin notre blog sur les bonnes crêpes bretonnes. En effet, au chapitre précédent, nous avons créé une application « blog » dans notre projet, il est désormais temps de se mettre au travail !

Pour rappel, comme vu dans la théorie, *chaque vue se doit d'être associée au minimum à une URL*. Avec Django, une vue est représentée par une fonction définie dans le fichier `views.py`. Cette fonction va généralement récupérer des données dans les modèles (ce que nous verrons plus tard) et appeler le bon template pour générer le rendu HTML adéquat. Par exemple, nous pourrions 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 débuter, nous allons réaliser 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 précédemment. Comme tout bon blog, le nôtre possèdera plusieurs vues qui rempliront diverses tâches, comme l'affichage d'un article par exemple.

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

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 ceci comme première ligne de code du fichier :

Code : Python

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

Cela vaut pour tous les fichiers que nous utiliserons à l'avenir. Spécifiez toujours un encodage UTF-8 au début de ceux-ci !

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 parties :

- Nous importons 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 nous le faisons 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 précédent, 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. Le schéma d'exécution est celui de la figure suivante.

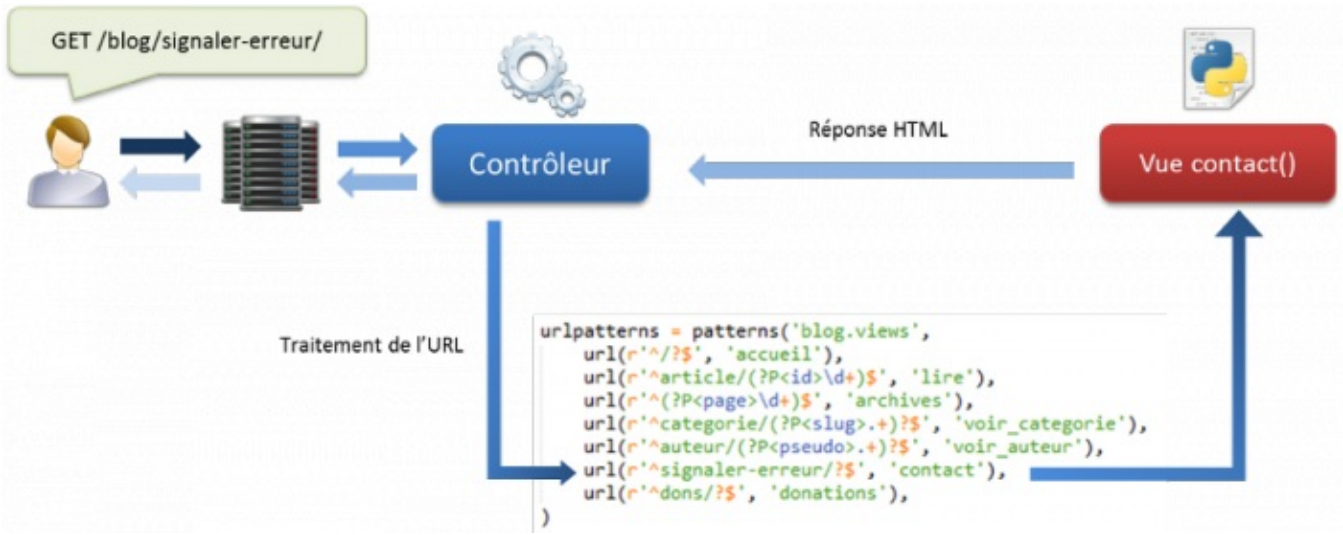


Schéma d'exécution d'une requête (nous travaillons pour le moment sans templates et sans modèles)

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, etc. ;
- Le chemin Python vers la vue correspondante.

Par exemple, en reprenant la vue définie tout à l'heure, si nous souhaitons que celle-ci soit accessible depuis l'URL `http://www.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'^$'` comme URL équivaut à spécifier la racine du site web. Autrement dit, si nous avions utilisé cette URL à la place de `r'^accueil/$'`, 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 expressions régulières, nous vous conseillons de faire une pause et d'aller voir le chapitre « Les expressions régulières » du cours « Apprenez à programmer en Python ».

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.com/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 la figure suivante.

L'affichage de votre première vue

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

Organiser proprement vos URL

Dans la partie précédente, nous avions 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 URL que nous avons faites : si nous avons plusieurs applications, toutes les URL 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 URL d'une application en l'incluant dans un projet, et une application complexe peut avoir des dizaines d'URL, 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 de vues déjà écrites. 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 précédemment (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`. Nous pouvons 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 d'ajouter 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 URL incluses. Autrement dit, nous avions 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 URL importées. Cependant, rien ne vous empêche de laisser cette chaîne de caractères vide (`/accueil` restera `/accueil`), mais il s'agit d'une bonne solution pour structurer vos URL.

Nous avons scindé nos URL dans un fichier `urls.py` pour chaque application. Cependant, nous allons bientôt ajouter d'autres URL plus complexes dans notre `blog/urls.py`. Toutes ces URL 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 jusqu'à 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 ; cela 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 URL à des vues et comment les organiser. Cependant, un besoin va bientôt se faire sentir : pouvoir passer des paramètres dans nos adresses directement. Si vous observez les adresses du site Instagram (qui est basé sur Django pour rappel), le lien vers une photo est construit ainsi : `http://instagr.am/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 expressions régulières. Par exemple, si nous souhaitons sur notre blog pouvoir accéder à un certain article via l'adresse `/blog/article/**` où `**` sera l'identifiant de l'article (un nombre unique), il 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
)
```

Lorsque l'URL `/blog/article/42` est demandée, Django regarde le routage et exécute la fonction `view_article`, en passant en paramètre 42. Autrement dit, Django appelle la vue de cette manière : `view_article(request, 42)`. 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, lorsque nous souhaitons 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 : `/blog/articles/2012/06`.

Cependant, si nous souhaitons changer l'ordre des paramètres de l'URL pour afficher le mois, et ensuite l'année, celle-ci deviendrait `/blog/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 règle 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.com/blog/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 `/blog/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 nous souhaitions 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, nous pouvons le faire savoir à l'utilisateur via une page d'erreur 404, qui correspond au code d'erreur indiquant qu'une page n'a pas été trouvée. Pour ce faire, il faut utiliser une exception du framework : `Http404`. Cette exception, du module `django.http`, arrête le traitement de la vue, et renvoie l'utilisateur vers une page d'erreur.

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

Code : Python

```
from django.http import HttpResponse, Http404

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

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

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

Erreur 404, page introuvable

Rediriger l'utilisateur

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

Si par exemple vous voulez que votre vue, après une certaine opération, redirige vos visiteurs vers le Site du Zéro, il faudrait procéder ainsi :

Code : Python

```
from django.shortcuts import redirect

def list_articles(request, year, month):
    # Il veut des articles ?
    return redirect("http://www.siteduzero.com") # Nous le
redirigeons vers le Site du Zéro
```

N'oubliez pas qu'une URL valide pour accéder à cette vue serait `/blog/articles/2005/05`.

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

Code : Python

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

def view_article(request, id_article):
    if int(id_article) > 100:
        raise Http404

    return redirect(view_redirection)

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

Code : Python - Extrait de blog/urls.py

```
url(r'^redirection/$', 'view_redirection'),
```

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

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

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

Code : Python

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

Pourquoi est-ce que nous utilisons 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 :

1. En passant directement la fonction Python, comme nous l'avons vu au début ;
2. 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) ;
3. En indiquant le nom de la vue tel qu'indiqué dans un `urls.py` (voir l'exemple suivant).

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 `/blog/article/42` tout seul et rediriger l'utilisateur vers cette URL. Ainsi, si par la suite vous souhaitez modifier vos URL, 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'URL 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="afficher_article"` qui permet d'indiquer le nom de la vue. Avec ce routage, en plus de pouvoir passer directement la fonction ou le chemin vers celle-ci en argument, nous pouvons faire beaucoup plus court et procéder comme ceci :

Code : Python

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

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

En résumé

- Le minimum requis pour obtenir une page web avec Django est une vue, associée à une URL.
- Une vue est une fonction placée dans le fichier `views.py` d'une application. Cette fonction doit toujours renvoyer un

objet `HttpResponse`.

- Pour être accessible, une vue doit être liée à une ou plusieurs URL dans les fichiers `urls.py` du projet.
- Les URL sont désignées par des expressions régulières, permettant la gestion d'arguments qui peuvent être passés à la vue pour rendre l'affichage différent selon l'URL visitée.
- Il est conseillé de diviser le `urls.py` du projet en plusieurs fichiers, en créant un fichier `urls.py` par application.
- Il existe des réponses plus spéciales permettant d'envoyer au navigateur du client les codes d'erreur 404 (page non trouvée) et 403 (accès refusé), ou encore d'effectuer des redirections.

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 difficiles 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.) que nous appelons 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 le montre la figure suivante.

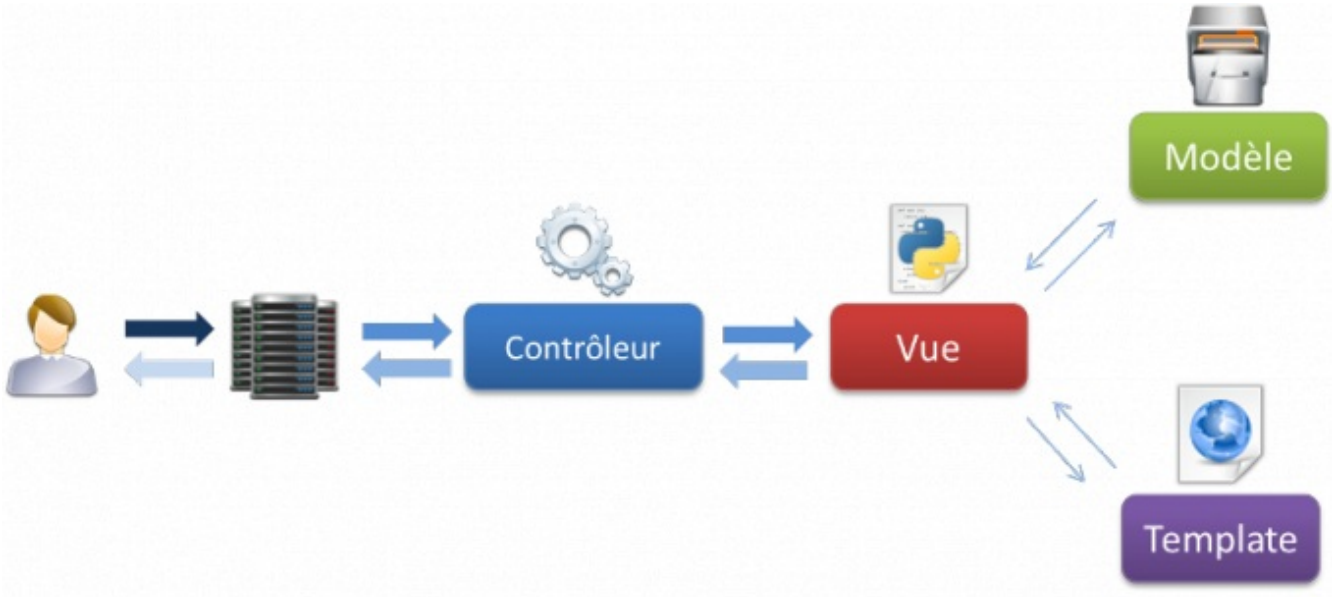


Schéma d'exécution d'une requête

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

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

Nous commençons par un exemple avec une vue qui renvoie juste la date actuelle à l'utilisateur, et son fichier `urls.py` associé :

Code : Python - blog/views.py

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

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

Code : Python - Extrait de blog/urls.py

```
    url(r'^$', 'tpl'),
```

Cette fonction prend en argument trois 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 `tpl.html`, dans le sous-dossier `blog`, et nous aurons 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 `tpl.html` :

Code : Jinja - templates/blog/tpl.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 bel et bien remplacé par la date actuelle !

Deuxième exemple : une vue, et son template associé, qui additionne deux nombres donnés dans l'URL.

Code : Python - blog/views.py

```
def addition(request, nombre1, nombre2):
    total = int(nombre1) + int(nombre2)

    # retourne nombre1, nombre2 et la somme des deux
    return render(request, 'blog/addition.html', locals())
```

Code : Python - Extrait de blog/urls.py

```
    url(r'^addition/(?P<nombre1>\d+)/(?P<nombre2>\d+)/$', 'addition'),
```

Code : Jinja - templates/blog/addition.html

```
    <h1>Ma super calculatrice</h1>
    <p>{{ nombre1 }} + {{ nombre2 }}, ça fait <strong>{{ total
    }}</strong> !<br />
    Nous pouvons également calculer la somme dans le template : {{
    nombre1|add:nombre2 }}.
```

Nous expliquerons bientôt les structures présentes dans ce template, 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 appelée. 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 `nombre1` valait 42, la valeur `nombre1` du dictionnaire vaudra elle aussi 42.

Affichons nos variables à l'utilisateur

Affichage d'une variable

Comme nous l'avons déjà expliqué, la vue transmet au template les données destinées à l'utilisateur. Ces données correspondent à des variables classiques de la vue. Nous pouvons 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, nous considérons 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 templates 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 l'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'un objet comme en Python, en les juxtaposant avec un point. Plus tard, nos articles de blog seront représentés par des objets, avec des attributs `titre`, `contenu`, etc. Pour y accéder, la syntaxe sera la suivante :

Code : Jinja

```
{# Nous supposons que notre vue a fourni un objet nommé article
contenant les attributs titre, auteur et contenu #}
<h2>{{ article.titre }}</h2>
<p><em>Article publié par {{ article.auteur }}</em></p>

<p>{{ article.contenu }}</p>
```

Si jamais une variable n'existe pas, ou n'a pas été envoyée au template, la valeur qui sera affichée à sa place est celle définie par `TEMPLATE_STRING_IF_INVALID` dans votre `settings.py`, 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 des sites d'actualités, le texte des dernières nouvelles est généralement tronqué, seul le début est affiché. Pour réaliser la même chose avec Django, nous pouvons utiliser un filtre qui limite l'affichage aux 80 premiers mots de notre article :

Code : Jinja

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

Ici, le filtre `truncatewords` (qui prend comme paramètre un nombre, séparé par un deux-points) est appliqué à la variable `texte`. À 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 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, nous aurons « cheval » si `nb_chevaux` est égal à 1 et « chevaux » pour le reste.

Et un dernier pour la route : imaginons que vous souhaitiez 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`, etc. Tous les filtres sont répertoriés et expliqués dans la documentation officielle de Django, n'hésitez pas à y jeter un coup d'œil pour découvrir d'éventuels filtres qui pourraient vous être utiles.

Manipulons nos données avec les tags

Abordons maintenant le second type d'opération implémentable dans un template : les tags. C'est grâce à ceux-ci que les conditions, boucles, etc. sont disponibles.

Les conditions : `{% if %}`

Tout comme en Python, il est possible d'exécuter des conditions dans votre template selon la valeur des variables passées au template :

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

Code : Python

```
def tpl(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 permet *une plus grande lisibilité* que lorsque le code

HTML est écrit directement 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, passez 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 nous possédions dans notre vue un tableau de couleurs définies en Python :

Code : Python

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

Nous décidons dès lors 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 (cela fonctionne avec n'importe quel autre type 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. Comme résultat, nous obtenons le code HTML suivant :

Code : HTML

```
Les couleurs de l'arc-en-ciel sont :
<ul>
    <li>rouge</li>
    <li>orange</li>
    <li>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">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`) : une condition, utiliser une autre boucle, l'afficher, etc.

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

Enfin, il existe une troisième directive qui peut être associée au `{% for %}`, il s'agit de `{% empty %}`. Elle permet d'afficher un message par défaut si la liste parcourue 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, s'il y a au moins un élément dans `commentaires`, alors une suite de paragraphes sera affichée, contenant chacun un élément de la liste. Sinon, le paragraphe « Pas de commentaires pour le moment. » sera renvoyé à l'utilisateur.

Le tag {% block %}

Sur la quasi-totalité des sites web, *une page est toujours composée 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 faudra modifier tous vos templates ! Heureusement, le tag `{% block %}` nous permet d'éviter cette épineuse situation. En effet, il est possible de déclarer des blocs, qui seront définis dans un autre template, et réutilisables dans le template actuel. Dès lors, nous pouvons créer un fichier, appelé 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>© 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. Ayant été écrits dans le fichier `base.html`, nous appelons ce fichier dans chacun des templates de notre blog. Pour ce faire, nous utilisons le tag `{% extends %}` (pour ceux qui ont déjà fait de la programmation objet, cela doit vous dire quelque chose ; cette méthode peut aussi être assimilée à `include` en PHP). Nous parlons alors d'*héritage de templates*. Nous prenons la base que nous surchargeons, 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 défini deux blocs, `title` et `content`. Le tag `extends` va aller chercher dans le template donné en argument, ici `base.html`, et remplacer les blocs vides de ce dernier par les blocs de même nom définis dans le

template appelé par la vue. Ainsi, `title` et `content` seront repris du template fils, mais `nav` sera le bloc `nav` défini dans `base.html`. En résumé, regardez la structure représentée dans l'image suivante :

Fonctionnement du 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="/blog/article/42">Lien vers mon super article n° 42</a>
```

Ce code nous indique le chemin vers la vue ou son nom comme premier paramètre, entre guillemets. Les arguments 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).

Nous aurions tout à fait pu utiliser une variable comme paramètre, que ce soit pour le nom de la vue ou les arguments :

Code : Jinja

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

Les commentaires : {% comment %}

Finalement, il existe 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 de documenter votre template, afin de pouvoir mieux s'y retrouver par la suite.

Il existe 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. Nous allons donc 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 depuis votre template de la façon suivante :

Code : Jinja

```
{% load static %}
<img src="{% static 'salut.jpg' %}" alt="Mon image" />
```

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.com`, vous devrez modifier toutes vos URL si elles sont écrites en dur ! En revanche, si elles 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 ajouter 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 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 emplacement des fichiers) pour générer une règle de routage correcte et adaptée. Pour le déploiement des fichiers statiques en production, référez-vous à l'annexe consacrée à ce sujet.

En résumé

- En pratique, et pour respecter l'architecture dictée par le framework Django, toute vue doit retourner un objet `HttpResponse` construit via un template.
- Pour respecter cette règle, il existe des fonctions nous facilitant le travail, comme `render`, présentée tout au long de ce chapitre. Elle permet de construire la réponse HTML en fonction d'un fichier template et de variables.
- Les templates permettent également de faire plusieurs traitements, comme afficher une variable, la transformer, faire des conditions... Attention cependant, ces traitements ont pour unique but d'afficher les données, pas de les modifier.
- Il est possible de factoriser des blocs HTML (comme le début et la fin d'une page) via l'utilisation des tags `{% block %}` et `{% extends %}`.
- Afin de faciliter le développement, Django possède un tag `{% url %}` permettant la construction d'URL en lui fournissant la vue à appeler et ses éventuels paramètres.
- L'ajout de fichiers statiques dans notre template (images, CSS, JavaScript) peut se faire via l'utilisation du tag `{% static %}`.

Les modèles

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

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

Créer un modèle

Un modèle s'écrit sous la forme d'une classe et représente une table dans la base de données, dont les 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 chacun 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` (sinon il ne sera pas pris en compte par le framework). 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

```
#-*- coding: utf-8 -*-
from django.db import models

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 précédent, nous avons créé quatre 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 le 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 bibliothèque 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 l'heure de la création de l'objet. Un autre argument du même genre existe, `auto_now`, qui 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.

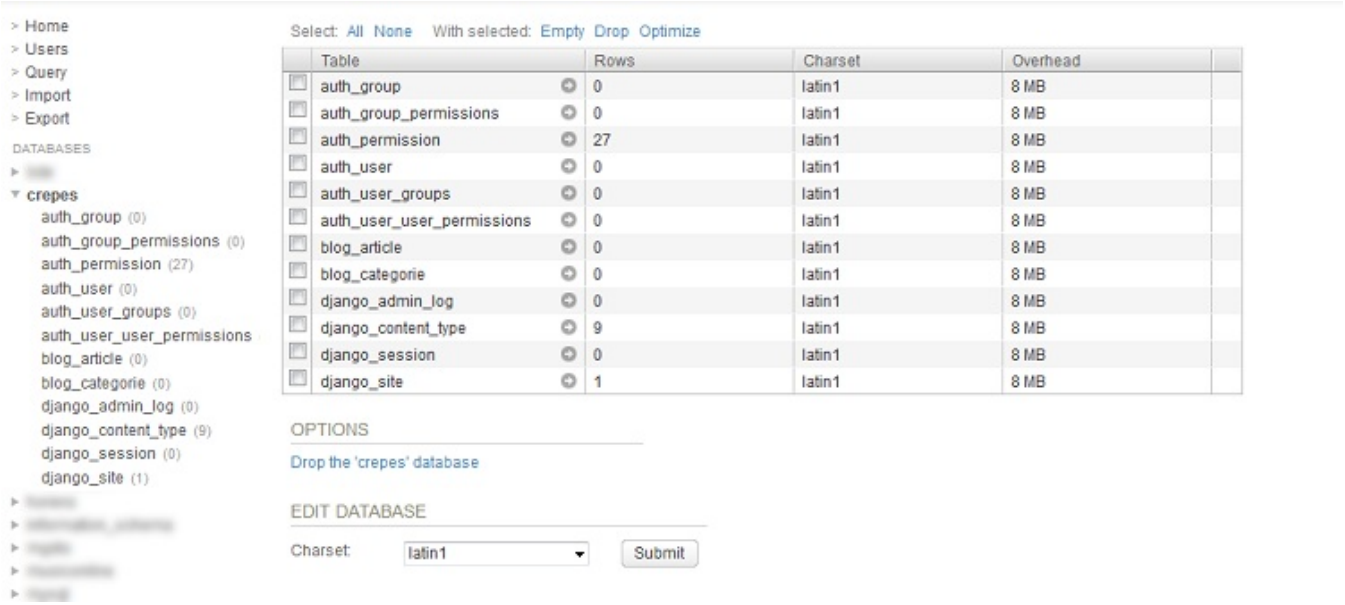
Il existe beaucoup d'autres champs disponibles, ceux-ci sont repris dans la documentation de Django. N'hésitez pas à la consulter en cas de doute ou question !

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.), comme à la figure suivante. 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.



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 nous étions 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 ajouter 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. Nous pouvons spécifier la valeur des attributs directement pendant l'instanciation de classe, ou l'assigner par la suite :

Code : Python

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

Nous vous conseillons de toujours utiliser des chaînes de caractères unicode pour toutes vos chaînes contenant des accents, comme le fait la dernière ligne. Cela vous évitera bien des mauvaises surprises. Pour rappel, une chaîne unicode est toujours précédée par `u`. Exemple : `u"Eh, toi, là !"`.

Pourquoi n'avons-nous pas mis de valeur à l'attribut `date` du modèle ?

`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-là, sans quoi Django retournera une erreur !

Nous pouvons 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=u"Les crêpes", contenu=u"Les
crêpes c'est cool").save()
>>> Article(auteur="Maxime", titre="La Bretagne", contenu="La
Bretagne c'est trop bien").save()
```

Cela étant fait, réutilisons la méthode `all` :

Code : Python

```
>>> Article.objects.all()
[<Article: Les crêpes>, <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 créés.

Nous pouvons donc par exemple afficher les différents titres de nos articles :

Code : Python

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

Les crêpes
La Bretagne
```

Maintenant, imaginons que vous souhaitiez sélectionner 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 arguments coïncide :

Code : Python

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

Les crêpes par Mathieu
```

Sachez que vous pouvez également filtrer ou exclure des entrées à partir de plusieurs champs : `Article.objects.filter(titre="Coucou", auteur="Mathieu")` renverra un `QuerySet` vide, car il n'existe aucun article de Mathieu intitulé « Coucou ».

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

Code : Python

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

Ces méthodes de recherche spéciales sont construites en prenant le champ concerné (ici `titre`), auquel nous ajoutons deux underscores « __ », suivis finalement de la méthode souhaitée. Ici, il s'agit donc de `titre__contains`, qui veut dire littéralement « prends tous les éléments dont le titre contient le mot passé en argument ».

D'autres méthodes du genre existent, 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 crêpes>, <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, nous souhaiterons organiser les articles par date de parution, du plus récent au plus ancien. Pour ce faire, il faut utiliser 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 crêpes>, <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 crêpes>]
```

Il est possible de passer plusieurs noms d'attributs à `order_by`. La priorité de chaque attribut dans le tri est déterminée par sa position dans la liste d'arguments. Ainsi, si nous trions les articles par nom et que deux d'entre eux ont le même nom, Django les départagera selon le deuxième attribut, et ainsi de suite tant que des attributs comparés seront identiques.

Accessoirement, nous pouvons 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','titre').reverse(
[<Article: La Bretagne>, <Article: Les crêpes>]
```

Pour terminer cette (longue) section, 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 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. Lookup
parameters were {'titre': "Je n'existe pas"}
>>> print Article.objects.get(auteur="Mathieu").titre
Les crêpes
>>> 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 n'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 :

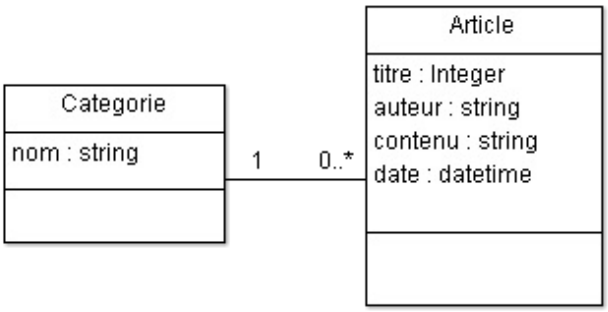
Code : Python

```
Article.objects.get_or_create(auteur="Mathieu")
>>> (<Article: Les crêpes>, 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 concevrez votre base de données, vous allez souvent faire des liens entre les classes (qui représentent nos tables SQL dans notre site), comme à la figure suivante.



Ici, un article peut être lié à une et une seule catégorie, et une catégorie peut être attribuée 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, créons la liaison depuis notre modèle `Article`, qu'il va falloir modifier en lui ajoutant 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
```

Nous avons donc ajouté un champ `ForeignKey`. En français, ce terme 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 nous lancions maintenant `manage.py syncdb`, il créerait bien la table correspondant au modèle `Categorie`, mais n'ajouterait 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, cette solution conserve vos données, mais c'est la plus difficile à implémenter ;
- Vous supprimez le fichier SQLite dans lequel la base de données est enregistrée (dans la partie précédente, nous avons indiqué `database.sql` comme nom de fichier) et utilisez ensuite la commande `manage.py syncdb`. Les données dans vos tables 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 données 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=u"Crêpes")
>>> cat.save()

>>> art = Article()
>>> art.titre=u"Les nouvelles crêpes"
>>> art.auteur="Maxime"
>>> art.contenu=u"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 peut en revanche 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 ». Nous accédons donc ici 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. Nous pouvons 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`… Nous pourrions 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=u"crêpes")
[<Article: Les nouvelles crêpes>]
```

Accéder à un élément d'une clé étrangère se fait en ajoutant deux underscores « __ », comme avec les méthodes de recherche spécifiques, suivis du nom de l'attribut recherché. Comme montré dans l'exemple, nous pouvons encore ajouter 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 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
```

N'oubliez pas de faire un `manage.py syncdb` !

Nous avons deux objets, un moteur nommé « Vroum » et une voiture nommée « Crêpes-mobile » qui est liée au moteur. Nous pouvons accéder du moteur à la voiture ainsi, depuis `manage.py shell` :

Code : Python

```
>>> from blog.models import Moteur, Voiture
>>> moteur = Moteur.objects.create(nom="Vroum") # create crée
directement l'objet et l'enregistre
>>> voiture = Voiture.objects.create(nom=u"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, qui 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, il faut utiliser l'argument `related_name` du `ForeignKey` ou `OneToOneField` et lui passer 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 de `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 d'ajouter 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. Nous pouvons soit laisser Django s'en occuper tout seul, soit la créer nous-mêmes pour y ajouter des attributs supplémentaires (pour rappel, ici nous ajoutons 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

```
>>> from blog.models import Vendeur, Produit, Offre
>>> 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 nous spécifions manuellement la table intermédiaire, soit nous laissons 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 son prix :

Code : Python

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

Si nous avions 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 nous utilisons une autre méthode du `ManyToManyField` :

Code : Python

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

Ensuite, pour accéder aux objets du modèle source (possédant la déclaration du `ManyToManyField`, ici `Vendeur`) associés au modèle destinataire (ici `Produit`), rien de plus simple, nous obtenons 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 nous souhaitons accéder aux valeurs du modèle intermédiaire (ici `Offre`), il faut procéder de manière classique :

Code : Python

```
>>> 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, nous pouvons 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 dans la console, et d'une manière plutôt théorique. Nous allons ici 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, nous ajouterons 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. Nous pouvons 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;
```

Note : nous n'avons sélectionné ici que les modèles `Categorie` et `Article`.

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() # Nous sélectionnons tous nos
articles
    return render(request, 'blog/accueil.html',
{'derniers_articles':articles})

def lire(request, id):
    """ Afficher un article complet """
    pass # Le code de cette fonction est donné un peu plus loin.
```

Code : Python - Extrait de blog/urls.py

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

Code : Jinja - templates/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>
{% empty %}
<p>Aucun article.</p>
{% endfor %}
```

Nous récupérons tous les articles via la méthode `objects.all()` et nous renvoyons la liste au template. Dans le template, il n'y a rien de fondamentalement nouveau non plus : nous affichons un à un les articles. Le seul point nouveau est celui que nous avons cité précédemment : nous faisons un lien vers l'article complet, en jouant avec le champ `id` de la table SQL. Si vous avez correctement suivi le sous-chapitre sur les manipulations d'entrées et tapé nos commandes, vous devriez avoir un article enregistré.

Afficher un article précis

L'affichage d'un article précis est plus délicat : il faut vérifier que l'article demandé existe, et renvoyer une erreur 404 si ce n'est pas le cas. 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 `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 est `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 ajouter 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>
```

Ce qui nous donne la figure suivante.

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

Rédigé par Ssx`z, le 13 juillet 2012

Préparation : 1h ; Cuisson : 2 min

Ingrédients (pour 20 crêpes environ) :
- 500 g de farine
- sel
- 6 oeufs
- 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 oeufs.

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.

À cette adresse, la vue de notre article (l'ID à la fin est variable, attention) : 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 permettant de déterminer quel article il faut afficher. C'est relativement pratique, mais cela a l'inconvénient de ne pas être très parlant pour l'utilisateur. Pour remédier à cela, nous voyons 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 avec l'adresse de ce cours : `http://www.siteduzero.com/informatique/tutoriels/creez-vos-applications-web-avec-django-1`. Nous pouvons y identifier la chaîne « creez-vos-applications-web-avec-django » qui nous permet de savoir 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 cela, 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
```

N'oubliez pas de mettre à jour la structure de votre table, comme nous l'avons déjà expliqué précédemment, et de créer une nouvelle entrée à partir de `manage.py shell` !

Désormais, nous pouvons aisément ajouter notre slug dans l'URL, en plus de l'ID lors de la construction d'une URL. Nous pouvons par exemple utiliser des URL comme celle-ci : `/blog/article/1-titre-de-l-article`. La mise en œuvre est également rapide à mettre en place :

Code : Python - Extrait de blog/urls.py

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

Code : Python - Extrait de blog/views.py

```
from django.shortcuts import render, get_object_or_404

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

Code : Jinja - templates/blog/accueil.html

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

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 uniques 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 souci.

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

En résumé

- Un modèle représente une table dans la base de données et ses attributs correspondent aux champs de la table.
- Tout modèle Django hérite de la classe mère `Model` incluse dans `django.db.models`.
- Chaque attribut du modèle est typé et décrit le contenu du champ, en fonction de la classe utilisée : `CharField`, `DateTimeField`, `IntegerField`...
- Les requêtes à la base de données sur le modèle `Article` peuvent être effectuées via des appels de méthodes sur `Article.objects`, tels que `all()`, `filter(nom="Un nom")` ou encore `order_by('date')`.
- L'enregistrement et la mise à jour d'articles dans la base de données se fait par la manipulation d'objets de la classe `Article`, et via l'appel à la méthode `save()`.
- Deux modèles peuvent être liés ensemble par le principe des clés étrangères. La relation dépend cependant des contraintes de multiplicité qu'il faut respecter : `OneToOneField`, `ManyToManyField`.
- Il est possible d'afficher les attributs d'un objet dans un template de la même façon qu'en Python via des appels du type `article.nom`. Il est également possible d'itérer une liste d'objets, pour afficher une liste d'articles par exemple.

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 gérer les diverses informations disponibles : les articles d'un blog, les comptes utilisateurs, etc. Un des gros points forts de Django est que celui-ci 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, elle est placée 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 existe toutefois bien d'autres modules, dont certains que nous aborderons par la suite : `django.contrib.messages` (gestion de messages destinés aux visiteurs), `django.contrib.auth` (système d'authentification et de gestion des utilisateurs), etc.

Accédons à cette administration !

Import des modules

Ce module étant optionnel, il est nécessaire d'ajouter quelques lignes dans notre fichier de configuration pour pouvoir en profiter. 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`
- `django.contrib.sessions`

qui sont normalement incluses de base dans `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 avions 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',
)
```

Finalement, le module `admin` nécessite aussi l'import de *middlewares*, normalement inclus par défaut également :

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

Sauvegardez le fichier `settings.py`. 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 inclus dans votre `INSTALLED_APPS` lors de votre premier `syncdb` (ce qui est le cas avec la configuration livrée par défaut), vous pouvez sauter cette étape.

Pour fonctionner, il faut créer de nouvelles tables dans la base de données, qui serviront à enregistrer les actions des administrateurs, définir les droits de chacun, etc. Pour ce faire, il faut procéder comme avec les modèles et utiliser la commande suivante : `python manage.py syncdb`. À la première exécution, cette commande vous demandera de renseigner des informations pour créer un compte **super-utilisateur**, qui sera au début le seul compte à pouvoir accéder à l'administration. Cette opération commence notamment par la directive suivante :

Code : Console

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

Répondez `yes` et insérez les informations utilisateur que Django vous demande.

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 pour nos vues, il est nécessaire de dire au serveur « Quand j'appelle cette URL, redirige-moi vers l'administration. » En effet, pour l'instant nous avons bel et bien importé le module, mais nous ne pouvons pas encore y accéder.

Comme pour les vues, cela se fait à partir d'un `urls.py`. Ouvrez le fichier `crepes_bretonnes/urls.py`.
Par défaut, Django a déjà indiqué plusieurs lignes pour l'administration, mais celles-ci sont commentées. Au final, après avoir décommenté ces lignes, votre fichier `urls.py` devrait ressembler à ceci :

Code : Python

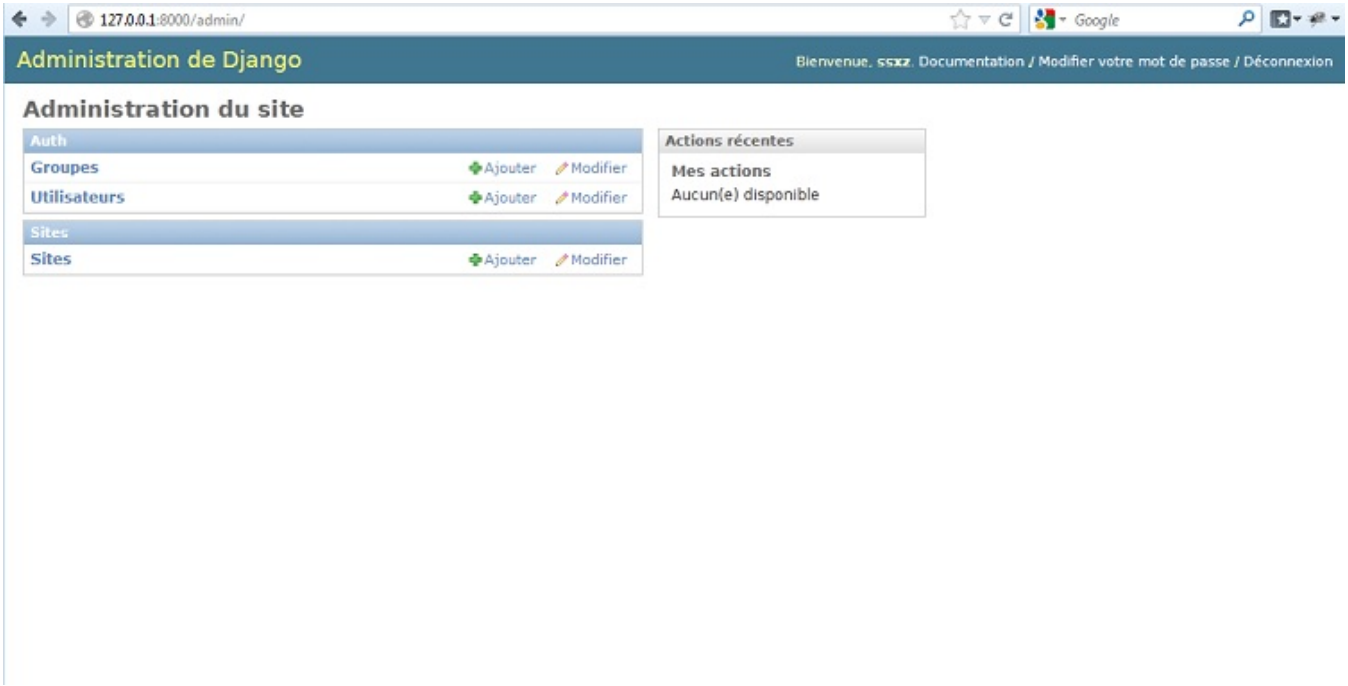
```
from django.conf.urls import patterns, include, url

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

urlpatterns = patterns('',
    # D'autres éventuelles directives.
    # Uncomment the next line to enable the admin:
    url(r'^admin/', include(admin.site.urls)),
)
```

Nous voyons 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 (voir la figure suivante), il suffira juste de vous connecter avec le nom d'utilisateur et le mot de passe que vous avez spécifiés lors du `syncdb` ou `createsuperuser`.

L'écran de connexion de l'administration

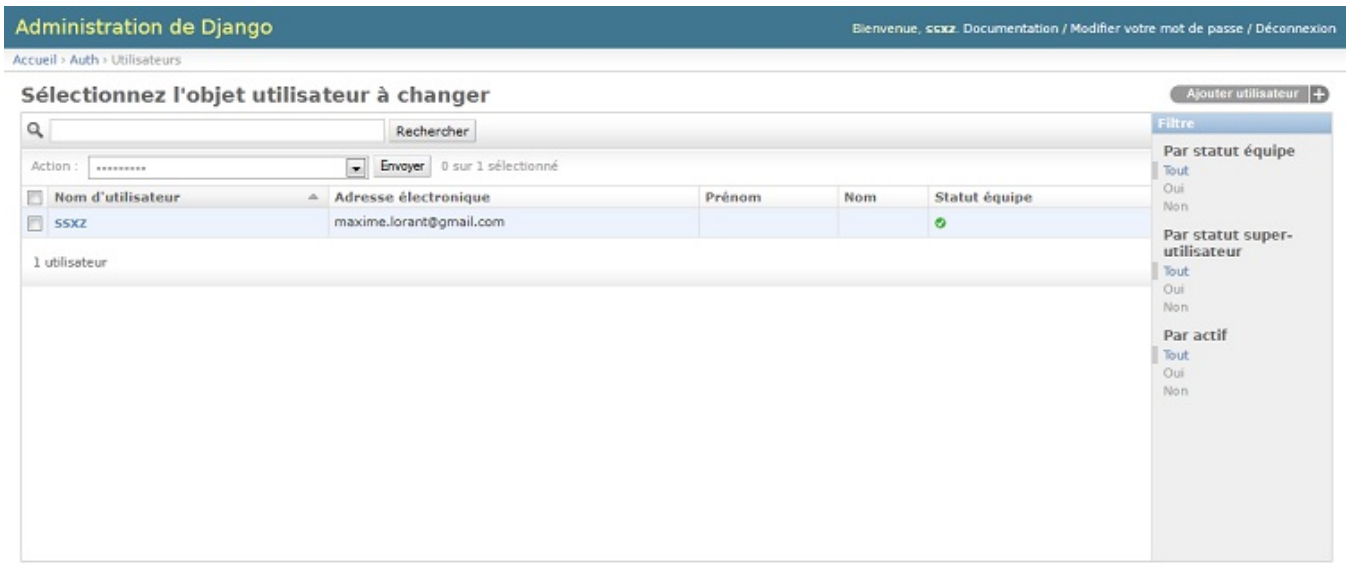
Première prise en main

Une fois que vous avez saisi vos identifiants de super-utilisateur, vous devez arriver sur une page semblable à la figure suivante.



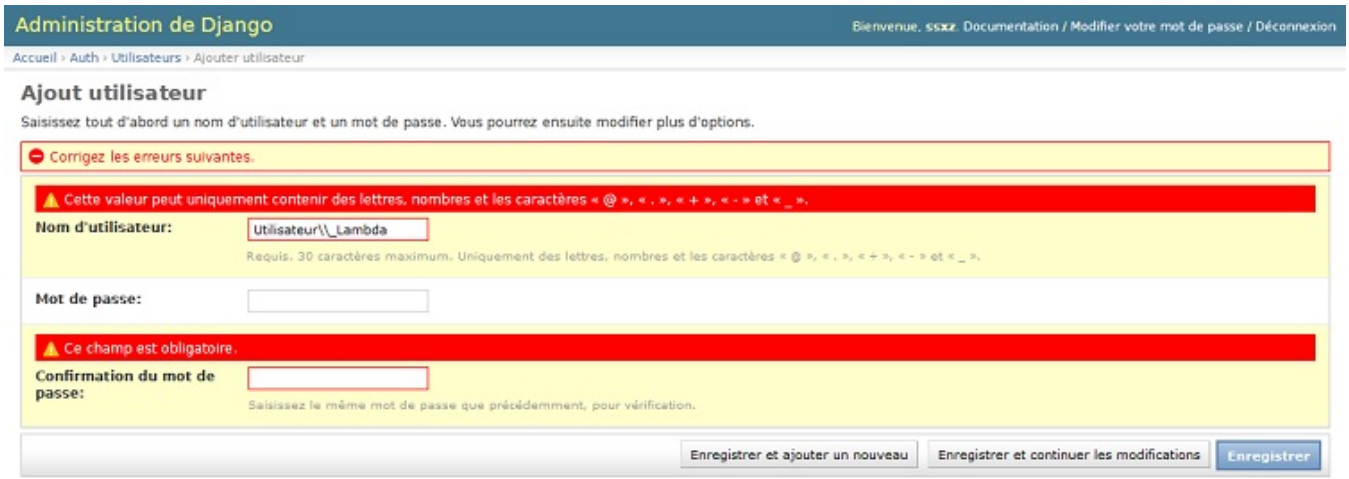
Accueil de l'administration

C'est encore un peu vide, mais ne vous inquiétez pas, nous allons bientôt pouvoir manipuler nos modèles `Article` et `Catégorie`, rédigés 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 vôtre, ainsi que vous pouvez le voir sur la figure suivante.



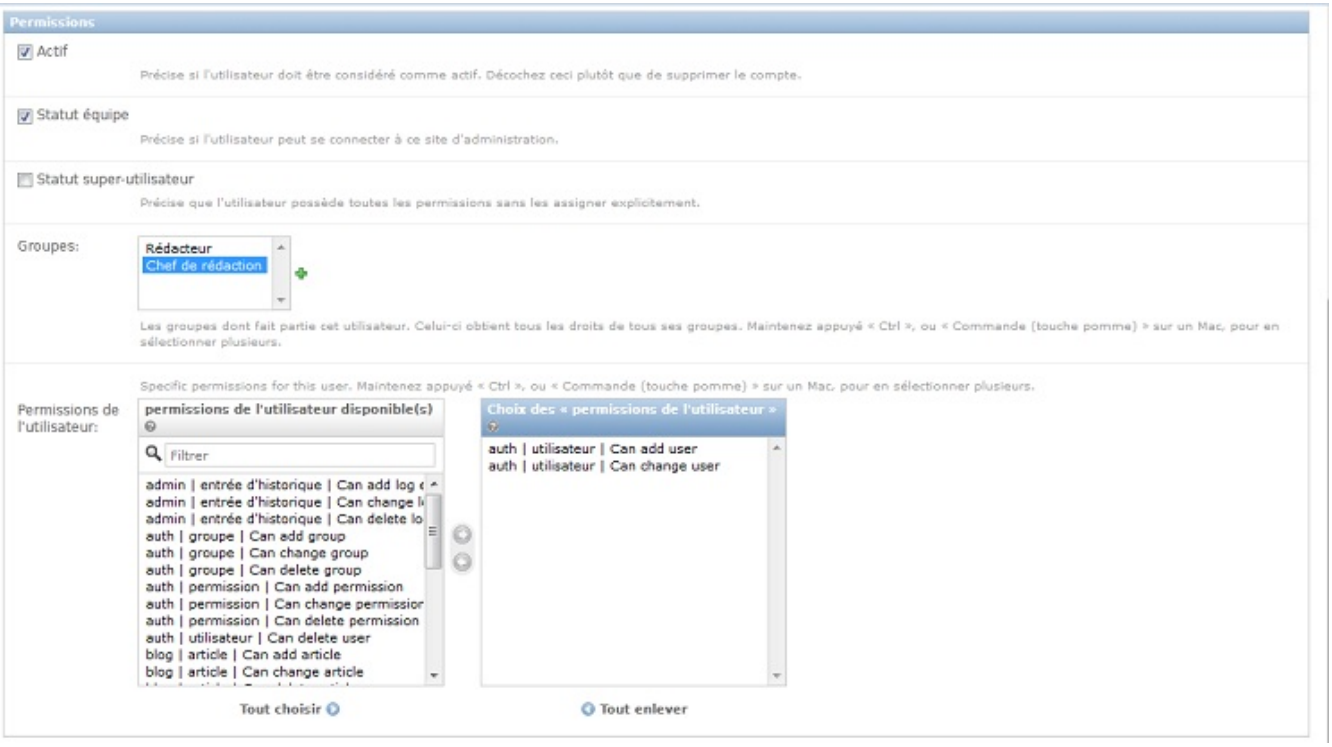
Liste des utilisateurs

C'est à partir d'ici que nous pouvons 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 ces opérations, nous allons d'abord créer un deuxième 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. Nous pouvons déjà remarquer sur la figure suivante que les formulaires peuvent gérer des contraintes, et l'affichage d'erreurs.



Formulaire de création de comptes, après le validateur avec erreur

Une fois cela validé, vous accédez directement à un formulaire plus complet, permettant de renseigner plus d'informations sur l'utilisateur qui vient d'être créé : ses 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 nous pouvons voir qu'il est possible d'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) !
Également, deux champs importants sont `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 (voir la figure suivante).



Exemple d'édition des permissions, ici j'ai créé deux groupes avant d'éditer l'utilisateur

Une fois que vous avez fini de gérer l'utilisateur, vous êtes redirigés vers la liste de tout à l'heure, avec une ligne en plus. Désormais, vous pouvez tester le tri, et les filtres qui sont disponibles à la droite du tableau ! Nous verrons d'ailleurs plus tard comment définir les champs à afficher, quels filtres utiliser, etc.

En définitive, pour finir ce rapide tour des fonctionnalités, vous avez peut-être remarqué la présence d'un bouton `Historique` en haut de chaque fiche utilisateur ou groupe. Ce bouton est très pratique, puisqu'il vous permet de suivre les modifications apportées, et donc de 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 d'accéder rapidement aux liens, en cas d'erreur par exemple (voir la figure suivante).

Historique des modifications d'un objet utilisateur

Administrons nos propres modèles

Pour le moment, nous avons vu comment manipuler les données des objets de base de Django, ceux concernant les utilisateurs. Il serait pratique désormais de *faire de même avec nos propres modèles*. Comme dit précédemment, l'administration est auto-générée : vous n'aurez pas à écrire beaucoup de lignes pour obtenir le même résultat que ci-avant. En réalité, quatre lignes suffisent : créez 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, nous indiquons à Django de *prendre en compte les modèles* `Article` et `Catégorie` dans l'administration. Rafraîchissez la page (relancez le serveur Django si nécessaire) et vous devez voir apparaître une nouvelle section, pour notre blog, semblable à la figure suivante.

La deuxième section nous permet enfin de gérer notre blog !

Les fonctionnalités sont les mêmes que celles pour les utilisateurs : nous pouvons éditer des articles, des catégories, les supprimer, consulter l'historique, etc. Vous pouvez désormais créer vos articles depuis cette interface et voir le résultat depuis les vues que nous avons créées précédemment.
Comme vous pouvez le voir, l'administration *prend en compte la clé étrangère* de la catégorie.

Comment cela fonctionne-t-il ?

Au lancement du serveur, le framework charge le fichier `urls.py` et tombe sur la ligne `admin.autodiscover()`. Cette méthode ira chercher dans chaque application installée (celles qui sont listées dans `INSTALLED_APPS`) un fichier `admin.py`, et si celui-ci existe exécutera son contenu.
Ainsi, si nous souhaitons activer l'administration pour toutes nos applications, il suffit de créer un fichier `admin.py` dans chacune, et d'appeler la méthode `register()` de `admin.site` sur chacun de nos modèles.
Nous pouvons 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 nous avons 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 5 du code ci-dessous, dans votre `settings.py` :

Code : Python

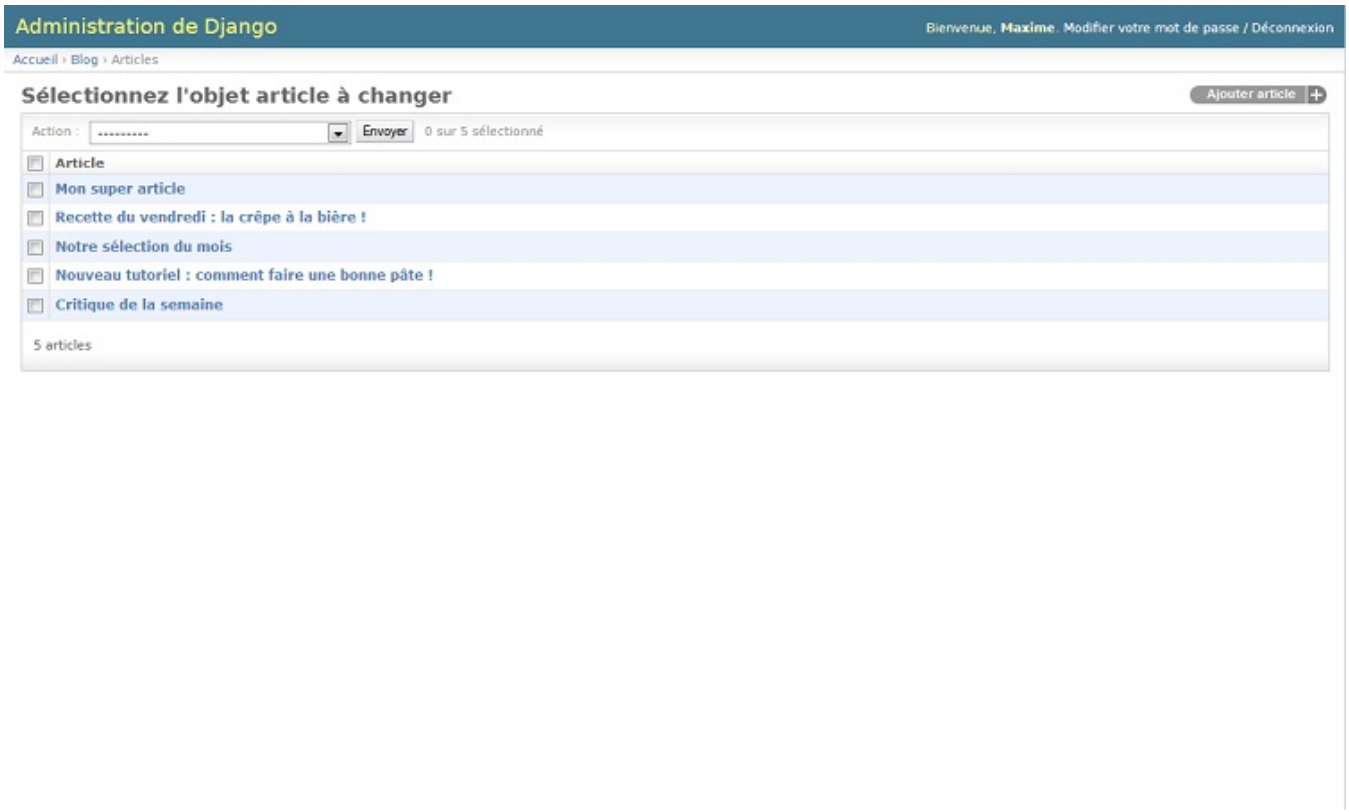
```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    'blog',
    # 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 vous permettra 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 voir comment améliorer la liste. En effet, pour le moment, nos listes sont assez vides, comme vous pouvez le constater sur la figure suivante.



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

Le tableau ne contient qu'une colonne contenant le titre de notre article. Cette colonne n'est pas due au hasard : c'est en réalité le résultat de la méthode `__unicode__` que nous avons définie dans notre modèle.

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    slug = models.SlugField(max_length=100)
    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
```

Ce résultat par défaut est assez utile, mais nous aimerions pouvoir gérer plus facilement nos articles : les trier selon certains champs, filtrer par catégorie, etc. Pour ce faire, nous devons créer une nouvelle classe dans notre fichier `admin.py`, contenant actuellement ceci :

Code : Python

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

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

Nous allons donc créer une nouvelle classe pour chaque modèle. Notre classe héritera de `admin.ModelAdmin` et aura principalement 5 attributs, listés dans le tableau suivant :

Nom de l'attribut	Utilité
`list_display`	Liste des champs du modèle à afficher dans le tableau
`list_filter`	Liste des champs à partir desquels nous pourrons filtrer les entrées
`date_hierarchy`	Permet de filtrer par date de façon intuitive
`ordering`	Tri par défaut du tableau
`search_fields`	Configuration du champ de recherche

Nous pouvons dès lors rédiger notre première classe adaptée 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 leur attribut `verbose_name` respectif.
- 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 (première ligne au-dessus du tableau).

Désormais, 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 deuxième paramètre :

Code : Python

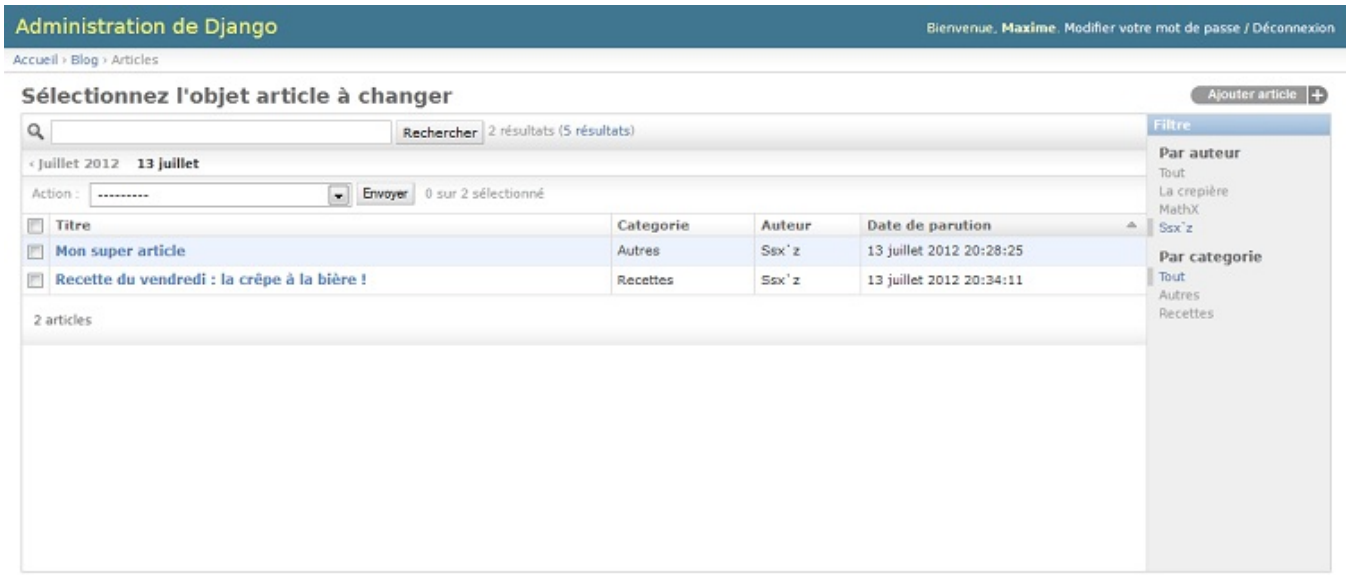
```
admin.site.register(Article, ArticleAdmin)
```

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)
```

Vous pouvez maintenant observer le résultat sur la figure suivante :



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

Les différents changements opérés sont désormais visibles. Vous pouvez bien sûr modifier selon vos besoins : ajouter le champ `Catégorie` dans le tableau, changer le tri…

Pour terminer, nous allons 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, nous allons devoir créer une méthode dans notre `ModelAdmin`, qui va se charger de renvoyer ce que nous souhaitons, 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. S'il
y a plus de 40 caractères, il faut ajouter 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 ses attributs. Ensuite, il suffit d'exécuter quelques opérations, puis de renvoyer une chaîne de caractères. Il faut ensuite intégrer cela dans notre `ModelAdmin`.

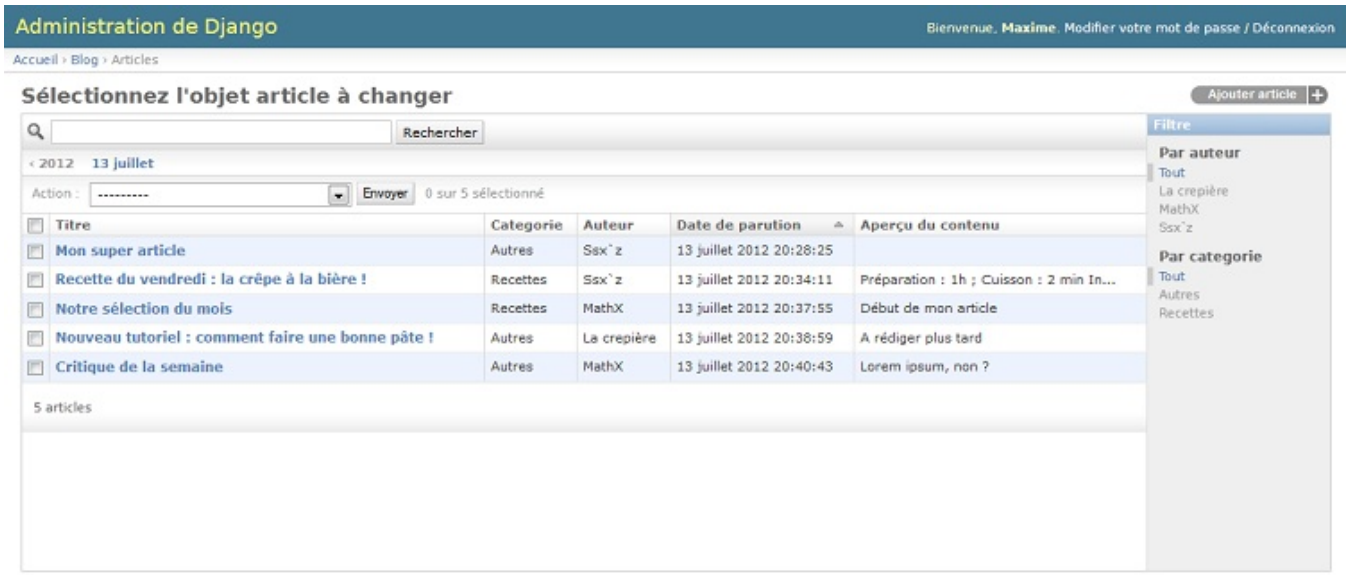
Et comment l'ajoute-t-on à notre `list_display` ?

Il faut traiter la fonction comme un champ. Il suffit donc d'ajouter `'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:utf-8 -*-
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. S'il
y a plus de 40 caractères, il faut ajouter 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
    apercu_contenu.short_description = u'Aperçu du contenu'

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

Nous obtenons notre nouvelle colonne avec les premiers mots de chaque article (voir la figure suivante).



La liste des articles est accessible depuis l'administration

Modifier le formulaire d'édition

Nous allons désormais nous occuper du formulaire d'édition. Pour le moment, comme vous pouvez le voir sur la figure suivante, nous avons un formulaire affichant tous les champs, hormis la date de publication (à cause du paramètre `auto_now_add=True` dans le modèle).

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 nous permettra de cacher des champs (inutile dans le cas présent) et, bien évidemment, de changer leur ordre :

Code : Python

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

Nous observons peu de changements, à part le champ `Catégorie` qui est désormais au-dessus de `Contenu` (voir la figure suivante).

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 des autres, sans distinction. Nous pouvons *hiérarchiser* cela en utilisant un attribut plus complexe que `fields`.
À titre d'exemple, nous allons mettre les champs `titre`, `auteur` et `categorie` dans un fieldset et `contenu` dans un autre.

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':u'Le formulaire accepte les balises HTML.
Utilisez-les à bon escient !',
           'fields': ('contenu', )
        }),
    )
```

Voyons pas à pas la construction de ce tuple :

1. Nos deux éléments dans le tuple `fieldset`, qui correspondent à nos *deux fieldsets distincts*.
2. Chaque élément contient un tuple contenant *exactement deux informations* : son nom, et les informations sur son contenu, sous forme de dictionnaire.
3. Ce dictionnaire contient trois types 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 premier champ ;
   c. `classes` : des classes CSS supplémentaires à appliquer sur le fieldset (par défaut il en existe trois : `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 !

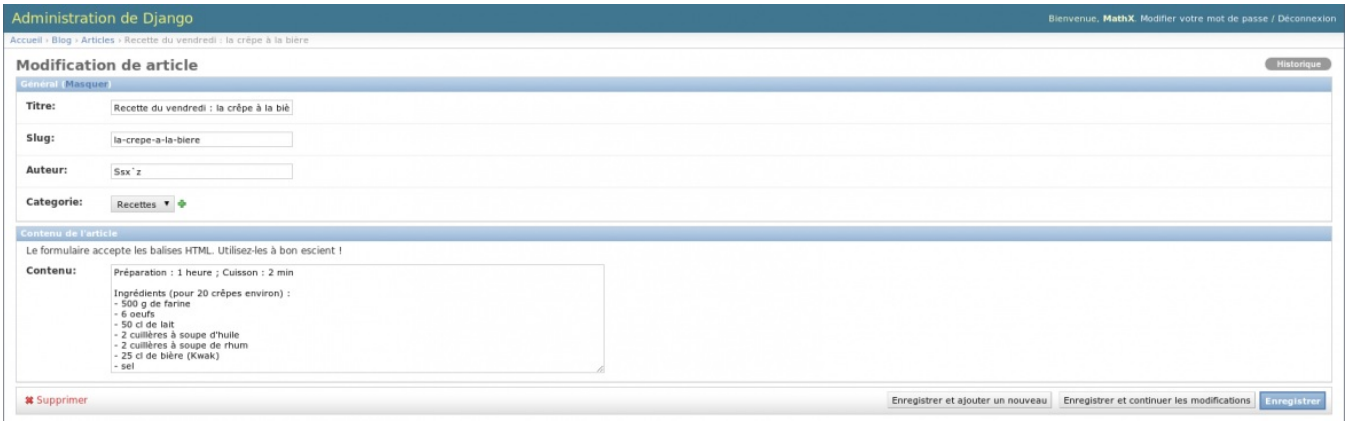
Ici, nous avons donc séparé les champs en deux fieldsets et affiché quelques informations supplémentaires pour aider à la saisie. Au final, nous avons le fichier `admin.py` suivant :

Code : Python

```
# -*- coding:utf-8 -*-
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', 'slug', 'auteur', 'categorie')
         }),
         # Fieldset 2 : contenu de l'article
         ('Contenu de l\'article', {
            'description': u'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. S'il
y a plus de 40 caractères, il faut rajouter 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 donne la figure suivante.



Notre formulaire, mieux présenté qu'avant

Si ni le champ `fields`, ni le champ `fieldset` ne sont présents, Django affichera par défaut tous les champs qui ne sont pas des `AutoField`, et qui ont l'attribut `editable` à `True` (ce qui est le cas par défaut de nombreux champs). Comme nous l'avons 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, ce champ est actuellement ignoré… Nous souhaitons toutefois le remplir, mais en plus que cela se fasse automatiquement !

Nous avons notre champ `slug` que nous pouvons désormais éditer à la main. Mais nous pouvons aller encore plus loin, en ajoutant une option qui remplit instantanément ce champ grâce à un script JavaScript. Pour ce faire, il existe un attribut aux classes `ModelAdmin` nommé `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 (voir la figure suivante).

Exemple d'utilisation de prepopulated_fields

En résumé

- L'administration est un outil optionnel : il est possible de ne pas l'utiliser. Une fois activée, de très nombreuses options sont automatisées, sans qu'il y ait besoin d'ajouter une seule ligne de code !
- Ce module requiert l'usage de l'authentification, et la création d'un super-utilisateur afin d'en restreindre l'accès aux personnes de confiance.
- De base, l'administration permet la gestion complète des utilisateurs, de groupes et des droits de chacun, de façon très fine.
- L'administration d'un modèle créé dans une de nos applications est possible en l'enregistrant dans le module d'administration, via `admin.site.register(MonModele)` dans le fichier `admin.py` de l'application.
- Il est également possible de personnaliser cette interface pour chaque module, en précisant ce qu'il faut afficher dans les tableaux de listes, ce qui peut être édité, etc.

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 vous 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. Voici un bref exemple de formulaire de contact :

Code : Python - blog/forms.py

```
#-*- coding: utf-8 -*-
from django import forms

class ContactForm(forms.Form):
    sujet = forms.CharField(max_length=100)
    message = forms.CharField(widget=forms.Textarea)
    envoyeur = forms.EmailField(label=u"Votre adresse mail")
    renvoi = forms.BooleanField(help_text=u"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 est, lui, identique.

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 nous recueillons 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 entrer 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. Lorsque nous choisissons un champ, nous le faisons selon le type de données qu'il faut recueillir (du texte, un nombre, une adresse e-mail, une date…). C'est le champ qui s'assurera que ce qu'a entré l'utilisateur est valide. En revanche, tout ce qui se rapporte à l'apparence du champ 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 bel et bien envoyé une adresse e-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

modifier le nom de la boîte de saisie qui est généralement défini selon le nom de la variable. `help_text` permet d'ajouter 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 ». Cela oblige l'utilisateur à cocher la boîte, et ce n'est pas ce que nous souhaitons ici.

La documentation officielle liste tous les champs et leurs options. N'hésitez pas à y jeter un coup d'œil si vous ne trouvez pas le champ qu'il vous faut.

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 (HTTP est le protocole, le « 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 la lui renvoie, aussi simplement que cela. Le deuxième type, qui nous intéresse le plus ici, est `POST`. Celui-ci va également demander une page du serveur, mais va en revanche aussi envoyer des données à celui-ci, généralement depuis un formulaire. Donc, pour savoir si l'utilisateur a complété un formulaire ou non, nous nous fions 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 en raison de son efficacité.

La voici :

Code : Python - Extrait de blog/views.py

```
from blog.forms import ContactForm

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

        if form.is_valid(): # Nous vérifions que les données
envoyées sont valides

            # Ici nous pouvons 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']

            # Nous pourrions ici envoyer l'e-mail grâce aux données
que nous venons de récupérer

            envoi = True

    else: # Si ce n'est pas du POST, c'est probablement une requête
GET
        form = ContactForm()  # Nous créons un formulaire vide

    return render(request, 'blog/contact.html', locals())
```

Code : Python - Extrait de blog/urls.py

```
url(r'^contact/$', 'contact'),
```

Si le formulaire est valide, un nouvel attribut de l'objet `form` est apparu, il 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, nous pourrions accéder au sujet du message ainsi :

Code : Python

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

Côté utilisateur, cela se passe en trois étapes :

1. Le visiteur arrive sur la page, complète le formulaire et l'envoie.
2. Si le formulaire est faux, nous retournons la même page tant que celui-ci n'est pas correct.
3. Si le formulaire est correct, nous le redirigeons 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, nous retournons 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 - templates/blog/contact.html

```
{% if envoi %}Votre message a bien été envoyé !{% endif %}

<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 sous la forme de liste grâce à `as_ul`. Utilisez ce que vous pensez être le plus adapté.
D'ailleurs, ces méthodes ne créent pas seulement le code HTML des champs, mais ajoutent aussi les messages d'erreur 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 à la figure suivante l'image du rendu (bien entendu, libre à vous de l'améliorer avec un peu de CSS).

Rendu du formulaire

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. Heureusement, 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 renvoyer 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, recevions 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 `ContactForm` du formulaire 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 ajouter une méthode semblable à celle-ci :

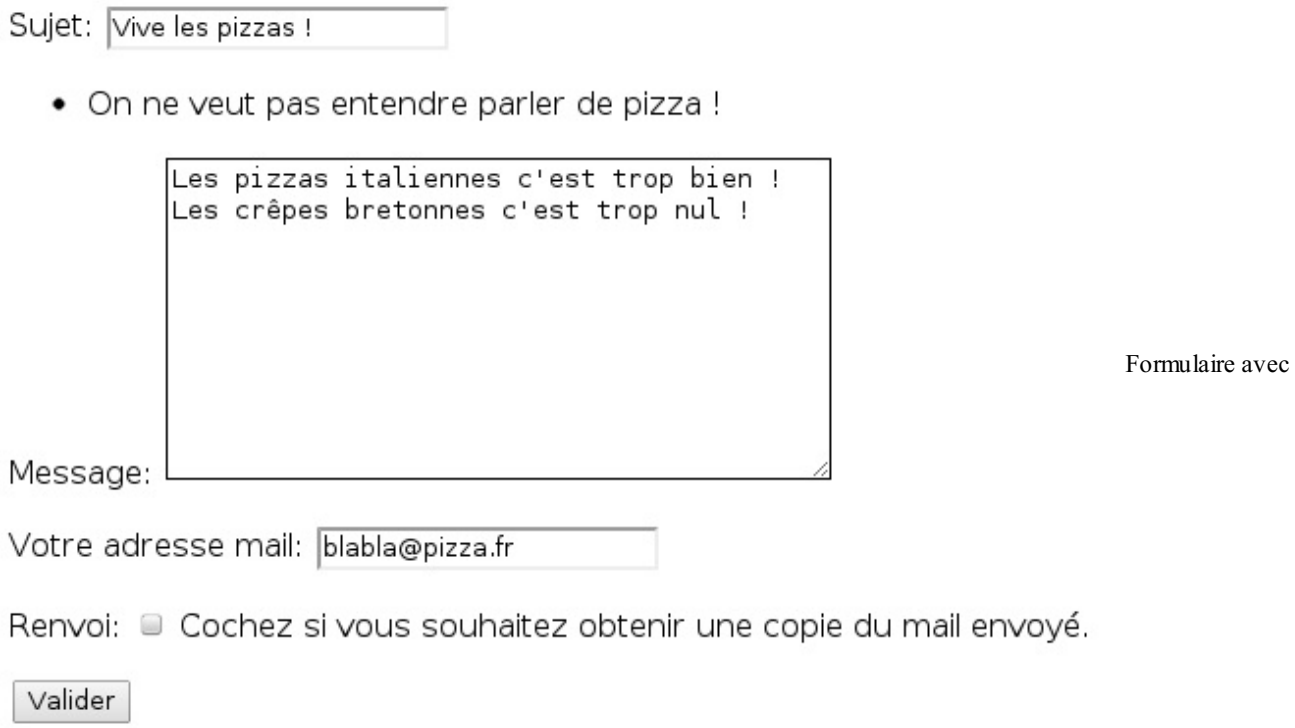
Code : Python

```
    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
```

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

Le rendu HTML nous donne le résultat que vous pouvez observer sur la figure suivante, avec des données invalides après

traitement du formulaire.



Formulaire avec données invalides

Maintenant, imaginons que nos fanatiques de la pizza italienne se soient adoucis et que nous ayons décidé d'être moins sévères, nous ne rejetterions 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` héritée de la classe mère `Form`. Les choses se compliquent un petit peu :

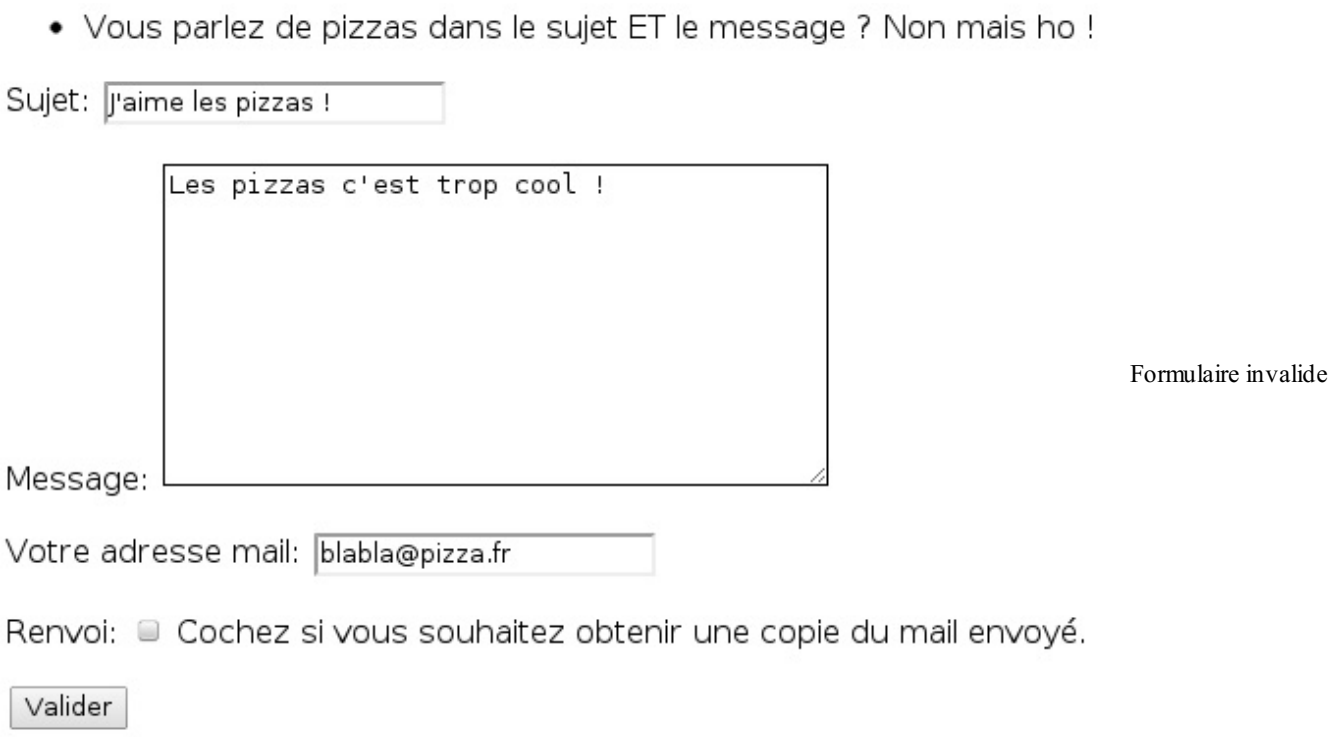
Code : Python

```
    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  # N'oublions 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 e-mail de l'utilisateur, avant de vérifier si celle-ci a déjà été utilisée, il faut laisser Django vérifier si l'adresse e-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 sont 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 e-mail spécifiée était incorrecte, elle ne sera pas reprise dans le dictionnaire renvoyé. Pour savoir si les valeurs que nous souhaitons filtrer sont valides, nous utilisons 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.

Voici à la figure suivante ce que donne le formulaire lorsqu'il ne passe pas la validation que nous avons écrite.



Formulaire invalide

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 :

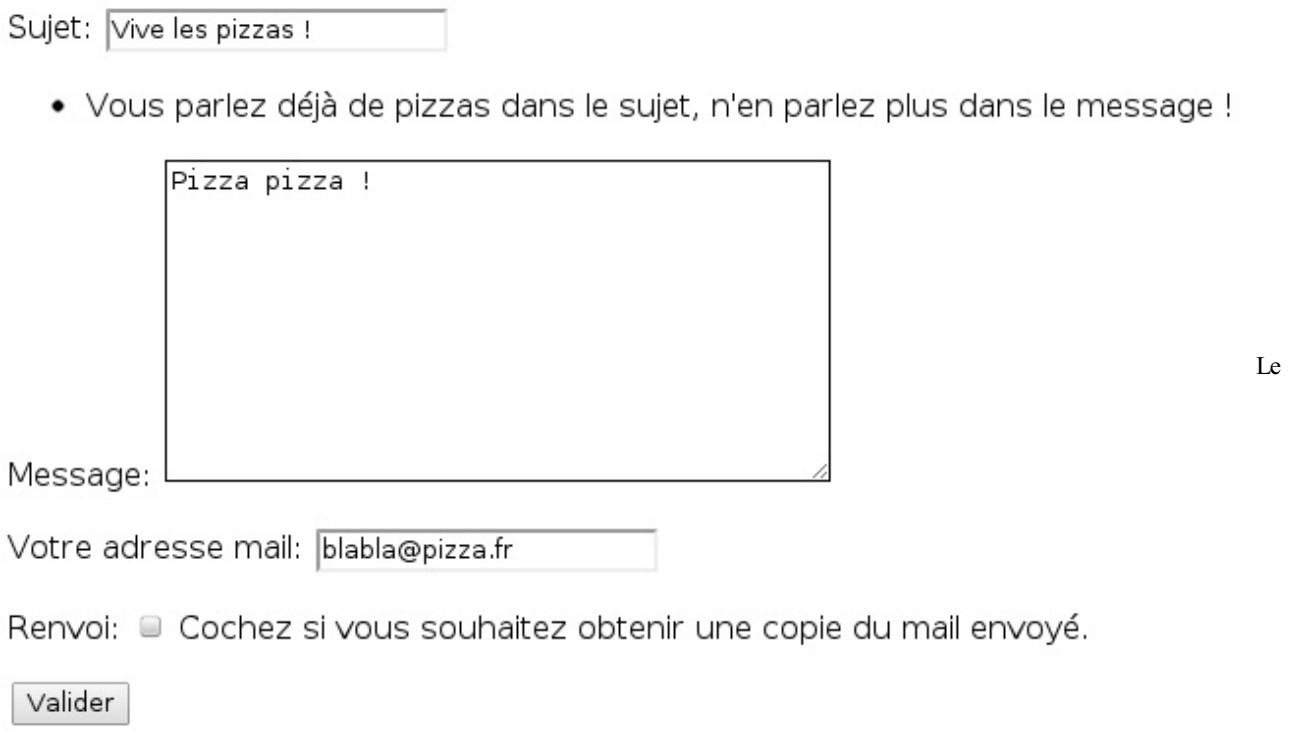
Code : Python

```
    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 », nous ne renvoyons plus une exception, mais nous définissons 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'erreur.

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 voici le résultat à la figure suivante.



Le message d'erreur est bien adapté

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 de 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.

Dans le chapitre sur les modèles, nous avons créé une classe `Article`. Pour rappel, la voici :

Code : Python

```
class Article(models.Model):
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    slug = models.SlugField(max_length=100)
    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
```

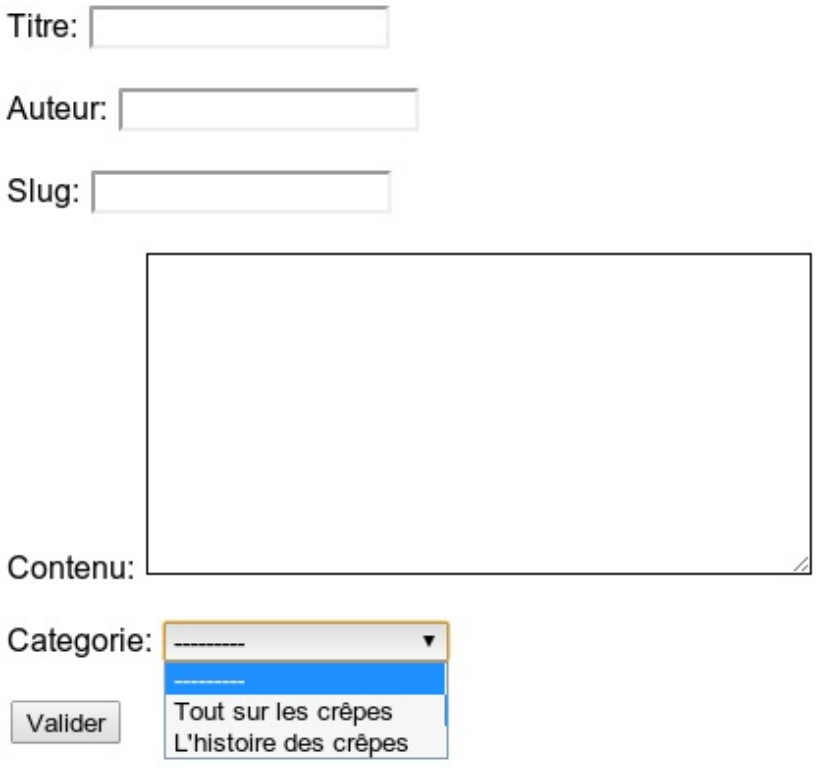
Pour faire un formulaire à partir de ce modèle, c'est très simple :

Code : Python - Extrait de blog/forms.py

```
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. Observez la figure suivante :



Le choix de la catégorie apparaît dans le formulaire

En plus de convertir les champs de modèle vers des champs de formulaire 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` est 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 toute seul. Petit exemple dans le `shell`:

Code : Python

```
>>> from blog.models import Article, Categorie
>>> from blog.forms import ArticleForm
>>> donnees = {
... 'titre':u"Les crêpes c'est trop bon",
... 'slug':"les-crepes-cest-trop-bon",
... 'auteur':"Maxime",
... 'contenu':u"Vous saviez que les crêpes bretonnes c'est trop bon
? La pêche c'est nul à côté.",
... 'categorie':Categorie.objects.all()[0].id  # Nous prenons
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>]
```

Tout objet d'un modèle sauvegardé possède un attribut `id`, c'est un identifiant propre à chaque entrée. Avec les `ForeignKey`, c'est lui que nous utilisons généralement comme clé étrangère.

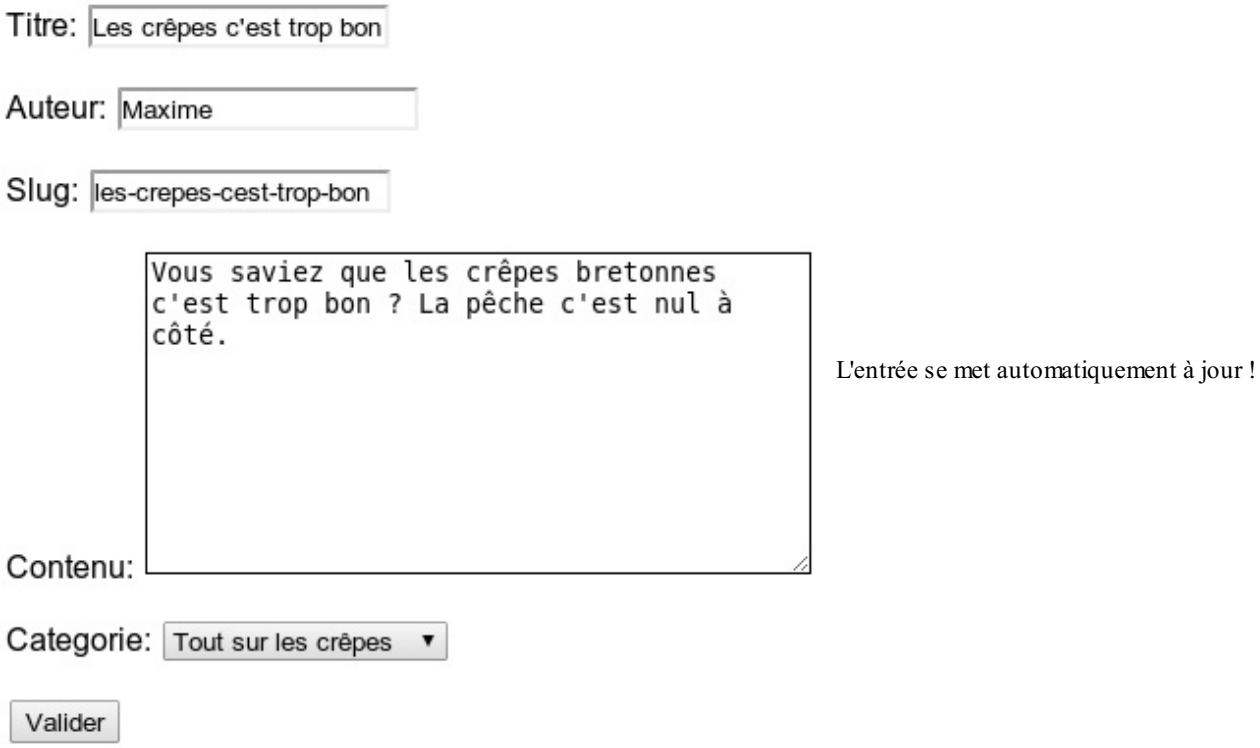
Pratique, n'est-ce pas ? Nous avons ici simulé avec un dictionnaire le contenu d'un éventuel `request.POST` et l'avons passé au constructeur d'`ArticleForm`. Depuis la méthode `save`, le `ModelForm` va directement créer une entrée dans la base de données et retourner l'objet créé.

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 base de données
```

Django se charge du reste, comme vous pouvez le voir sur la figure suivante !



L'entrée se met automatiquement à jour !

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 d'en sélectionner ou d'en exclure certains, toujours grâce à la sous-classe `Meta`:

Code : Python

```
class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ('auteur','categorie','slug')  # Exclura les
champs nommés « auteur », « categorie » et « slug »
```

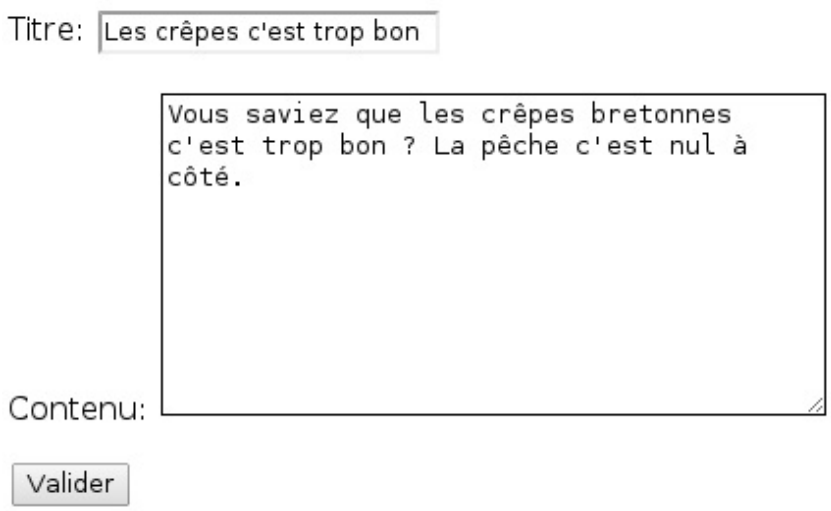
En ayant exclu ces trois champs, cela revient à sélectionner uniquement les champs `titre` et `contenu`, comme ceci :

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.

Observez le résultat à la figure suivante.



Seuls les champs « titre » et « contenu » sont éditables

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]  # Nous ajoutons
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 base de données, mais renvoie un objet avec les données du formulaire sur lequel nous pouvons continuer à travailler.

En résumé

- Un formulaire est décrit par une classe, héritant de `django.forms.Form`, où chaque attribut est un champ du formulaire défini par le type des données attendues.
- Chaque classe de `django.forms` permet d'affiner les données attendues : taille maximale du contenu du champ, champ obligatoire ou optionnel, valeur par défaut…
- Il est possible de récupérer un objet `Form` après la validation du formulaire et de vérifier si les données envoyées sont valides, via `form.is_valid()`.
- La validation est personnalisable, grâce à la réécriture des méthodes `clean_NOM_DU_CHAMP()` et `clean()`.
- Pour moins de redondances, la création de formulaires à partir de modèles existant se fait en héritant de la classe `ModelForm`, à partir de laquelle nous pouvons modifier les champs éditables et leurs comportements.

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 côté serveur et les méthodes proposées par Django.

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 trois caractéristiques : leur nom, leur adresse et une photo. Pour ce faire, créons un nouveau modèle (placez-le dans l'application de votre choix, personnellement nous réutiliserons 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 caractères, une date, un 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 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 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, à un petit détail près :

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 5 : un deuxième argument a été ajouté, 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'enverra 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 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 côté 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 média (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 ajouter 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)
```

Cela 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).

Cela 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})
```

Côté template :

Code : Jinja

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

{% for contact in contacts %}
<h2>{{ contact.nom }}</h2>
Adresse : {{ contact.adresse|linebreaks }}<br/>
<img src="{{ MEDIA_URL }}{{ contact.photo }}"/>
{% 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 chaine de caractères (comme l'adresse ici) en un `<br/>` automatiquement, et cela 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 le nom du fichier associé. Afin de construire une adresse complète, il est impératif d'associer ces deux parties d'adresse, 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, comme vous pouvez le constater sur la figure suivante.

L'adresse de Chuck Norris !

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 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.com/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 possible de les renommer à notre guise, et de ne pas garder le nom que l'utilisateur avait sur son disque dur.

La méthode est plutôt simple : au lieu de passer une chaîne de caractères comme paramètre `upload_to` dans le modèle, il faut lui passer une fonction qui retournera le nouveau nom du fichier. Cette fonction prend deux 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]  # Nous nous basons sur
l'ID de l'entrée et nous gardons 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 !

En résumé

- L'installation de la bibliothèque PIL (*Python Imaging Library*) est nécessaire pour gérer les images dans Django. Cette bibliothèque permet de faire des traitements sur les images (vérification et redimensionnement notamment).
- Le stockage d'une image dans un objet en base se fait via un champ `models.ImageField`. Le stockage d'un fichier quelconque est similaire, avec `models.FileField`.
- Les fichiers uploadés seront stockés dans le répertoire fourni par `MEDIA_ROOT` dans votre `settings.py`.

TP : un raccourcisseur d'URL

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 étudiés (URL, modèles, vues, formulaires, administration et templates).

Cahier des charges

Pour ce travail pratique, nous allons réaliser un raccourcisseur d'URL. Ce type de service est notamment utilisé sur les sites de microblogging (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 cliquera 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 que nous nommerons `mini_url`. Cette application ne contiendra qu'un modèle appelé `MiniURL`, c'est lui qui enregistrera les raccourcis. Il comportera les champs suivants :

- L'URL longue : `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, car vous ne l'avez pas vu dans le cours auparavant. Les autres sont classiques et ont été 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`. Il ne contiendra que les champs URL et pseudo, le reste sera soit initialisé selon les valeurs par défaut, soit 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 l'absence de doublon. Dans un *souci de simplicité* et de pédagogie, nous allons sauter cette étape.

Vous aurez ensuite trois vues :

- Une vue affichant toutes les redirections créées et leurs informations, triées par ordre descendant, de la redirection avec le plus d'accès vers celle en ayant le moins ;
- 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 l'URL longue.

Partant de ces trois fonctions, il ne faudra que 2 templates (la redirection n'en ayant pas besoin), et 3 routages d'URL 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 aux figures suivantes ce que vous devriez obtenir.

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, cela ne devrait pas avoir posé de problèmes !

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

```
#-*- coding: utf-8 -*-
from django.db import models
import random
import string


class MiniURL(models.Model):
    url = models.URLField(verbose_name=u"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=u"Nombre
    d'accès à l'URL")

        def __unicode__(self):
            return u"[{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 URL"
```

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. Nous avons 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 ne 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 nous avons modifié le nom qui sera utilisé dans les `ModelForm`, l'administration (`verbose_name`) et sa forme plurielle (`verbose_name_plural`).

Après la création de nouveaux modèles, il fallait les ajouter dans la base de données via la commande `python manage.py syncdb` (n'oubliez pas d'ajouter l'application dans votre `settings.py` !) :

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

```python
#-*- 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

```python
#-*- 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.
)
```

La directive permettant d'importer le `mini_url/urls.py` dans votre `urls.py` principal :

Code : Python

```python
url(r'^m/', include('mini_url.urls')),
```

Nous avons nommé les URL ici pour des raisons pratiques que nous verrons plus tard dans ce cours.

Et pour finir, le fichier `views.py` :

Code : Python - mini_url/views.py

```python
#-*- 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 de 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 cliquera 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 avions 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'URL).

Code : HTML - liste.html

```
<h1>Le raccourcisseur d'URL spécial crêpes bretonnes !</h1>

<p><a href="{% url 'url_nouveau' %}">Raccourcir une URL.</a></p>

<p>Liste des URL 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 }} accès)</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.
Ce TP conclut la partie 2 du tutoriel. Nous avons couvert les bases du framework. Vous devez normalement être capables de réaliser des applications basiques. Dans les prochaines parties, nous allons approfondir ces bases et étudier les autres bibliothèques que Django propose.

Nous vous conseillons de copier/coller le code que nous avons fourni en solution dans votre projet. Nous serons amenés à l'utiliser dans les chapitres suivants.

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 anonyme une redirection ;
- Etc.

Partie 3 : Techniques avancées

Nous avons vu de manière globale chaque composante du framework dans la partie précédente. Vous êtes désormais capables de réaliser de petites applications fonctionnelles, mais qui pourraient se révéler limitées à certains endroits. Afin de repousser ces limites, nous allons voir dans cette nouvelle partie des méthodes avancées du framework, permettant de réduire nos efforts lors de la conception et d'ouvrir de nouvelles possibilités.

Les vues génériques

Sur la plupart des sites web, il existe certains types de pages où créer une vue comme nous l'avons fait précédemment est lourd et presque inutile : pour une page statique sans informations dynamiques par exemple, ou encore de simple listes d'objets sans traitements particuliers.
Django est conçu pour n'avoir à écrire que *le minimum* (philosophie DRY), le framework inclut donc un système de vues génériques, qui évite au développeur de devoir écrire des fonctions simples et identiques, nous permettant de gagner du temps et des lignes de code.

Ce chapitre est assez long et dense en informations. 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

Les vues génériques sont en quelque sorte des vues très modulaires, prêtes à être utilisées directement, incluses par défaut dans le framework et cela sans devoir écrire la vue elle-même.
Pour illustrer le fonctionnement global des vues génériques, prenons cet exemple de vue classique, qui ne s'occupe que d'afficher un template à l'utilisateur, sans utiliser de variables :

Code : Python - Exemple de vue mal conçue

```
#-*- coding: utf-8 -*-
from django.shortcuts import render

def faq(request):
    return render(request, 'blog/faq.html', {})
```

Code : Python - ... avec sa définition d'URL associée

```
url('faq', 'blog.views.faq', name='faq'),
```

Une première caractéristique des vues génériques est que ce ne sont pas des fonctions, comme la vue que nous venons de présenter, 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 existe *deux méthodes principales d'utilisation* pour les vues génériques :

1. Soit nous créons une classe, héritant d'un type de vue générique dont nous surchargerons les attributs ;
2. Soit nous appelons directement la classe générique, en passant en arguments les différentes informations à utiliser.

Toutes les classes de vues génériques sont situées dans `django.views.generic`. Un premier type de vue générique est `TemplateView`. 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 - Dans un fichier 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()),    # Nous demandons la vue
correspondant à la classe FAQView créée précédemment
)
```

C'est tout ! Lorsqu'un visiteur 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` retourne une vue (en réalité, il s'agit d'une fonction classique) qui se basera sur ses attributs pour 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 ce soit pratique, il n'y a jusqu'ici rien de très puissant.

Abordons maintenant 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, etc.), et lorsque vous cliquez sur un élément, vous êtes redirigés 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`, nous pouvons 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('',
    # Nous allons réécrire l'URL de l'accueil
    url(r'^$', ListView.as_view(model=Article,)),

    # Et nous avons 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`, et 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 : nous ne souhaitons généralement pas tout afficher sur une même page, mais par exemple filtrer les articles affichés. 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 URL 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`. Référez-vous au chapitre sur ce sujet.

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> -
{% endif %}
Page {{ page_obj.number }} sur {{ page_obj.paginator.num_pages }}
{% if page_obj.has_next %}
- <a href="?page={{ page_obj.next_page_number }}">Suivante</a>
{% 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 première catégorie créée seront affichés. Vous pouvez bien entendu effectuer des requêtes identiques à celle des vues, avec du tri, plusieurs conditions, etc. Il est également possible de *passer des arguments* pour rendre la sélection *un peu plus dynamique* en ajoutant 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 quatre arguments, à quoi correspond le deuxième ? le quatrième ?).

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):
        # Nous récupérons le contexte depuis la super-classe
        context = super(ListeArticles,
self).get_context_data(**kwargs)
        # Nous ajoutons 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 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 nous affichons un objet, c'est d'avoir la possibilité 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):
    # Nous récupérons l'objet, via la super-classe
    article = super(LireArticle, self).get_object()

    article.nb_vues += 1  # Imaginons que nous ayons un attribut « Nombre de vues »
    article.save()

    return article  # Et nous retournons 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 fait qu'*afficher 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 quatre types d'actions applicables sur une donnée :

- `Create` (créer) ;
- `Read` (lire, que nous avons déjà traité juste au-dessus) ;
- `Update` (mettre à jour) ;
- `Delete` (supprimer).

Nous montrerons comment réaliser ces opérations dans l'ordre indiqué. En effet, 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 : des commentaires, des posts de forum, etc., ou encore de 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 nous travaillons, et `template_name` permet de spécifier le chemin vers le template (par défaut, le chemin est `<app>/<model>_create_form.html`, avec le nom du modèle tout en minuscules).
La nouveauté ici réside dans les deux 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 avions écrite précédemment étant donné qu'elle est suffisante pour l'exemple.

Le dernier argument permet quant à lui de *spécifier vers où rediriger l'utilisateur 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 URL 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 cela 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 nous l'avions fait avec l'ancienne méthode.

En premier lieu, 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, mais nous avons écrit sensiblement moins.

UpdateView

Après la création, attaquons-nous à la *mise à jour des données*. Imaginons que nous souhaitions 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 ?

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 et 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 auto-complétés 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 remplacer la 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 URL distinctes. Il suffit de se dire : « Quand nous validons le formulaire, nous soumettons 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 premier objet, `/url/edition/2` pour le deuxième, etc.

Vous pouvez le constater sur la figure suivante : le résultat est satisfaisant. Bien évidemment, la vue est *très minimaliste* : n'importe qui peut éditer tous les liens, il n'y a pas de message de confirmation, etc. Par contre, il y a une gestion des objets qui n'existe pas en renvoyant une page d'erreur 404, des formulaires incorrects, etc. Tout cela est améliorable.

Exemple de formulaire de mise à jour, reprenant le même template que l'ajout

Améliorons nos URL 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). Nous pourrions prendre le code présent dans l'URL réduite (voir la figure suivante).

Ce que nous avons actuellement et ce que nous souhaitons 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 est 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 considéré comme validé. Par défaut, il s'occupe *d'enregistrer les modifications et de rediriger l'utilisateur*, mais vous pouvez très bien 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. Comme pour `UpdateView`, cette vue prend un objet et demande la confirmation de suppression. Si l'utilisateur confirme, alors la suppression est effectuée, puis l'utilisateur est redirigé. 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 redirigera 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 %} <!-- Nous prenons 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 oublier l'import de URLDelete !
```

Afin de faciliter le tout, deux 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'URL spécial crêpes bretonnes !</h1>

<p><a href="{% url "url_nouveau" %}">Raccourcir une URL.</a></p>

<p>Liste des URL 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 }} accès)</li>
        {% empty %}
        <li>Il n'y en a pas actuellement.</li>
        {% endfor %}
        </ul>
```

Même refrain : nous enregistrons, et nous pouvons tester grâce au lien ajouté (voir la figure suivante).

Notre vue, après avoir cliqué sur un des liens « Supprimer » qui apparaissent dans la liste

Ce chapitre touche 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 génériques, 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 :

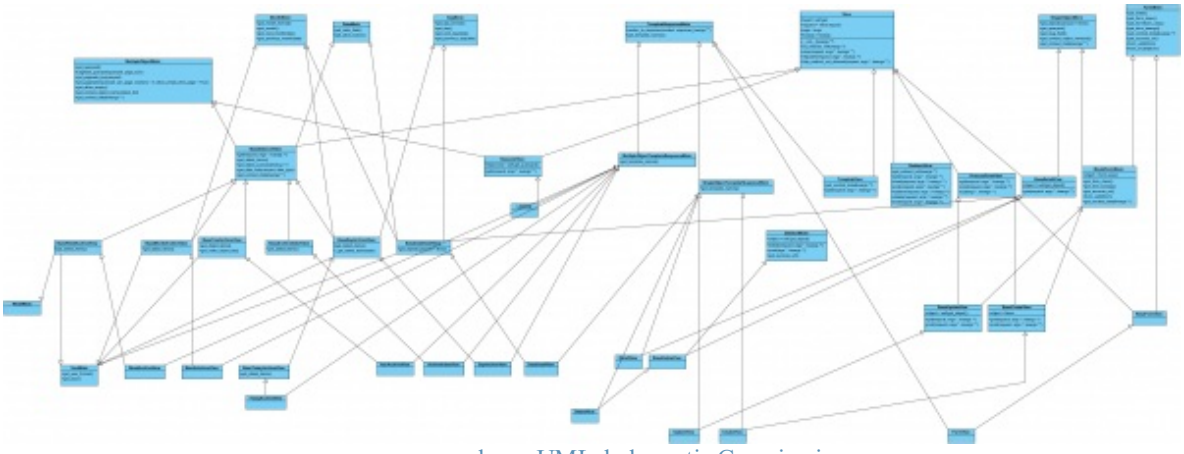


Diagramme de classe UML de la partie Generic views

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.

En résumé

- Django fournit un ensemble de classes permettant d'éviter de réécrire plusieurs fois le même type de vue (affichage d'un template statique, liste d'objets, création d'objets…) ;
- Les vues génériques peuvent être déclarées directement au sein de `urls.py` (cas le plus pratique pour les `TemplateView`) ou dans `views.py` ;
- Chaque vue générique dispose d'un ensemble d'attributs permettant de définir ce que doit faire la vue : modèle concerné, template à afficher, gestion de la pagination, filtres… ;
- Il est possible d'automatiser les formulaires d'ajout, de mise à jour et de suppression d'objets via des vues génériques ;
- Le module `django.views.generic` regorge de classes (plusieurs dizaines en tout), n'hésitez pas à regarder si l'une d'entre elles fait ce que vous souhaitez avant de vous lancer.

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. Ce sont 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 demandé 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 »). Eh bien, c'est ici qu'intervient l'objet `Q` ! 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)
```

Ajoutons quelques élèves dans la console interactive (`manage.py shell`) :

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, nous pouvons 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>  # Nous voyons
bien que nous possédons 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, il est possible de construire une clause « OU » à partir de `Q` :

Code : Python

```
Eleve.objects.filter(Q(moyenne__gt=16) | Q(moyenne__lt=8)) # Nous
prenons 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 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 prendre la négation d'une condition. Autrement dit, demander la condition inverse (« NOT » en SQL). Cela se fait en faisant précéder un objet `Q` dans une requête par 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))`, ce qui est identique à `Q(moyenne=10)`.

Quel intérêt ? Imaginons que nous devions obtenir les objets qui 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)>]
```

Que sont `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 d'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. Il s'agit de la 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)), nous pouvons 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 propres 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`, également issues de `django.db.models`, 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, ajoutons 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 deux 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 parmi 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, nous avons 6 « élèves », à savoir 4+2, car Django ne vérifie pas si un élève est déjà dans un autre cours ou non.

Pour terminer, abordons une dernière fonctionnalité utile. Il est possible d'ajouter des attributs à un objet selon les objets auxquels il est lié. Nous parlons 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 d'être retournées dans un dictionnaire, les valeurs 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 les 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.

Lorsque nous sauvegardons 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 correspondant à `Lieu`. Les valeurs des deux attributs `nom` et `adresse` seront enregistrées 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, à la différence que c'est 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=u"La crêperie bretonne",adresse="42 Rue de la
crêpe 35000 Rennes", menu=u"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 #Ça 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, etc., nous pouvons ajouter 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 # Nous spécifions qu'il s'agit d'un proxy
        ordering = ["nom"] # Nous changeons 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
```

L'application ContentType

Il est possible qu'un jour vous soyez amenés à devoir jouer avec des modèles. Non pas avec des instances de modèles, mais des modèles mêmes.

En effet, jusqu'ici, nous avons pu lier des entrées à d'autres entrées (d'un autre type de modèle éventuellement) avec des liaisons du type `ForeignKey`, `OneToOneField` ou `ManyToManyField`. Néanmoins, Django propose un autre système qui peut se révéler parfois utile : la liaison d'une entrée de modèle à un autre modèle en lui-même. Si l'intérêt d'une telle relation n'est pas évident à première vue, nous pouvons pourtant vous assurer qu'il existe, et nous vous l'expliquerons par la suite. Ce genre de relation est possible grâce à ce que nous appelons des `ContentTypes`.

Avant tout, il faut s'assurer que l'application `ContentTypes` de Django est bien installée. Elle l'est par défaut, néanmoins, si vous avez supprimé certaines entrées de votre variable `INSTALLED_APPS` dans le fichier `settings.py`, il est toujours utile de vérifier si le tuple contient bien l'entrée `'django.contrib.contenttypes'`. Si ce n'est pas le cas, ajoutez-la.

Un `ContentType` est en réalité un modèle assez spécial. Ce modèle permet de représenter un autre modèle installé. Par exemple, nous avons déclaré plus haut le modèle `Eleve`. Voici sa représentation depuis un `ContentType` :

Code : Python

```
>>> from blog.models import Eleve
>>> from django.contrib.contenttypes.models import ContentType
>>> ct = ContentType.objects.get(app_label="blog", model="eleve")
>>> ct
<ContentType: eleve>
```

Désormais, notre variable `ct` représente le modèle `Eleve`. Que pouvons-nous en faire ? Le modèle `ContentType` possède deux méthodes :

- `model_class` : renvoie la classe du modèle représenté ;
- `get_object_for_this_type` : il s'agit d'un raccourci vers la méthode `objects.get` du modèle. Cela évite de faire `ct.model_class().objects.get(attr=arg)`.

Illustration :

Code : Python

```
>>> ct.model_class()
<class 'blog.models.Eleve'>
>>> ct.get_object_for_this_type(nom="Maxime")
<Eleve: Élève Maxime (7/20 de moyenne)>
```

Maintenant que le fonctionnement des `ContentType` vous semble plus familier, passons à leur utilité. À quoi servent-ils ? Imaginons que vous deviez implémenter un système de commentaires, mais que ces commentaires ne se restreignent pas à un seul modèle. En effet, vous souhaitez que vos utilisateurs puissent commenter vos articles, vos images, vos vidéos, ou même des commentaires eux-mêmes.

Une première solution serait de faire hériter tous vos modèles d'un modèle parent nommée `Document` (un peu comme démontré ci-dessus) et de créer un modèle `Commentaire` avec une `ForeignKey` vers `Document`. Cependant, vous avez déjà écrit vos modèles `Article`, `Image`, `Video`, et ceux-ci n'héritent pas d'une classe commune comme `Document`. Vous n'avez pas envie de devoir réécrire tous vos modèles, votre code, adapter votre base de données, etc. La solution magique ? Les relations génériques des `ContentTypes`.

Une relation générique d'un modèle est une relation permettant de lier une entrée d'un modèle défini à une autre entrée de n'importe quel modèle. Autrement dit, si notre modèle `Commentaire` possède une relation générique, nous pourrions le lier à un modèle `Image`, `Video`, `Commentaire`... ou n'importe quel autre modèle installé, sans la moindre restriction.

Voici une ébauche de ce modèle `Commentaire` avec une relation générique :

Code : Python

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes import generic

class Commentaire(models.Model):
    auteur = models.CharField(max_length=255)
    contenu = models.TextField()
    content_type = models.ForeignKey(ContentType)
    object_id = models.PositiveIntegerField()
    content_object = generic.GenericForeignKey('content_type',
'object_id')

    def __unicode__(self):
        return "Commentaire de {0} sur {1}".format(self.auteur,
self.content_object)
```

La relation générique est ici l'attribut `content_object`, avec le champ `GenericForeignKey`. Cependant, vous avez sûrement remarqué l'existence de deux autres champs : `content_type` et `object_id`. À quoi servent-ils ?

Auparavant, avec une `ForeignKey` classique, tout ce qu'il fallait indiquer était, dans la déclaration du modèle, la classe à laquelle la clé serait liée, et lors de la déclaration de l'instance, l'entrée de l'autre modèle à laquelle la clé serait liée, cette dernière contenant alors l'ID de l'entrée associée.

Le fonctionnement est similaire ici, à une exception près : le modèle associé n'est pas défini lors de la déclaration de la classe (vu que ce modèle peut être n'importe lequel). Dès lors, il nous faut un champ supplémentaire pour représenter ce modèle, et ceci se fait grâce à un `ContentType` (que nous avons expliqué plus haut). Si nous avons le modèle, il ne nous manque plus que l'ID de l'entrée pour pouvoir récupérer la bonne entrée. C'est ici qu'intervient le champ `object_id`, qui est juste un entier positif,

contenant donc l'ID de l'entrée. Dès lors, le champ `content_object`, la relation générique, est en fait une sorte de produit des deux autres champs. Il va aller chercher le type de modèle associé dans `content_type`, et l'ID de l'entrée associée dans `object_id`. Finalement, la relation générique va aller dans la table liée au modèle obtenu, chercher l'entrée ayant l'ID obtenu, et renvoyer la bonne entrée. C'est pour cela qu'il faut indiquer à la relation générique lors de son initialisation les deux attributs depuis lesquels il va pouvoir aller chercher le modèle et l'ID.

Maintenant que la théorie est explicitée, prenons un petit exemple pratique :

Code : Python

```
>>> from blog.models import Commentaire, Eleve
>>> e = Eleve.objects.get(nom="Sofiane")
>>> c = Commentaire.objects.create(auteur="Le
professeur",contenu="Sofiane ne travaille pas assez.",
content_object=e)
>>> c.content_object
<Eleve: Élève Sofiane (10/20 de moyenne)>
>>> c.object_id
4
>>> c.content_type.model_class()
<class 'blog.models.Eleve'>
```

Lors de la création d'un commentaire, il n'y a pas besoin de remplir les champs `object_id` et `content_type`. Ceux-ci seront automatiquement déduits de la variable donnée à `content_object`.
Bien évidemment, vous pouvez adresser n'importe quelle entrée de n'importe quel modèle à l'attribut `content_object`, cela marchera !

Avant de terminer, sachez qu'il est également possible d'ajouter une relation générique « en sens inverse ». Contrairement à une `ForeignKey` classique, aucune relation inverse n'est créée. Si vous souhaitez tout de même en créer une sur un modèle bien précis, il suffit d'ajouter un champ nommé `GenericRelation`.

Si nous reprenons le modèle `Eleve`, cette fois modifié :

Code : Python

```
from django.contrib.contenttypes import generic

class Eleve(models.Model):
    nom = models.CharField(max_length=31)
    moyenne = models.IntegerField(default=10)
    commentaires = models.GenericRelation(Commentaire)

    def __unicode__(self):
        return u"Élève {0} ({1}/20 de moyenne)".format(self.nom,
self.moyenne)
```

Dès lors, le champ `commentaires` contient tous les commentaires adressés à l'élève :

Code : Python

```
>>> e.commentaires.all()
[u"Commentaire de Le professeur sur Élève Sofiane (10/20 de
moyenne)"]
```

Sachez que si vous avez utilisé des noms différents que `content_type` et `object_id` pour construire votre `GenericForeignKey`, vous devez également le spécifier lors de la création de la `GenericRelation` :

Code : Python

```
commentaires = models.GenericRelation(Commentaire,
```

```
    content_type_field="le_champ_du_content_type",
                                        object_id_field="le
    champ_de_l_id")
```

En résumé

- La classe `Q` permet d'effectuer des requêtes complexes avec les opérateurs « OU », « ET » et « NOT ».
- `Avg`, `Max` et `Min` permettent d'obtenir respectivement la moyenne, le maximum et le minimum d'une certaine colonne dans une table. Elles peuvent être combinées avec `Count` pour déterminer le nombre de lignes retournées.
- L'héritage de modèles permet de factoriser des modèles ayant des liens entre eux. Il existe plusieurs types d'héritage : abstrait, classique et les modèles proxy.
- L'application `ContentType` permet de décrire un modèle et de faire des relations génériques avec vos autres modèles (pensez à l'exemple des commentaires !).

Simplifions nos templates : filtres, tags et contextes

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 combler 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 que nous appelons le contexte du template). Nous aborderons ces différentes possibilités dans ce chapitre.

Préparation du terrain : architecture des filtres et tags

Pour construire nos propres filtres et tags, Django impose que ces derniers soient placés dans une application, tout comme les vues ou les modèles. À partir d'ici, nous retrouvons 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), dans ce cas vous pouvez directement le(s) placer au sein de l'application concernée ; nous préférons cette méthode ;
- 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 de 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 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 `settings.py`, 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 entrer dans le vif du sujet, à savoir la création de filtres et de tags !

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 deux filtres : l'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 appelées 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 bord lors du « chaînage » de filtres par exemple.

Un premier exemple de filtre sans argument

Attaquons la réalisation de notre premier filtre. Pour commencer, prenons comme exemple le modèle « Citation » de Wikipédia : nous allons encadrer la chaîne fournie par des guillemets français doubles.
Ainsi, si dans notre template nous avons `{{ "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, encadré de guillemets français
doubles et d'espaces insécables
"""
    return "« %s »" % 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 ;
- Soit en appelant la méthode `register.filter('citation', citation)`.

Notons qu'avec ces deux 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:utf-8 -*-
from django import template

register = template.Library()

@register.filter
def citation(texte):
    """
Affiche le texte passé en paramètre, encadré de guillemets français
doubles et d'espaces insécables
"""
    return "« %s »" % texte


@register.filter(name='citation_nom_different')
def citation2(texte):
    """ […] """
    return "« %s »" % texte

def citation3(texte):
    """ […] """
    return "« %s »" % texte

register.filter('citation3', citation3)
```

Par commodité, nous n'utiliserons plus que les première et deuxième méthodes dans ce cours. La dernière est pour autant tout à fait valide, 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`, introduit récemment, 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 à la figure suivante.

Un jour, une certaine personne m'a dit : « Bonjour le monde ! ». Le résultat incorrect de notre filtre

Mais pourquoi les espaces insécables sont-elles échappées ?

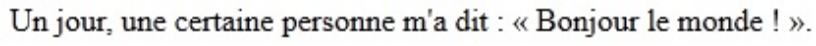
Par défaut, Django échappe automatiquement tous les caractères spéciaux des chaînes de caractères affichées dans un template, ainsi que le résultat des filtres. Nous allons donc devoir préciser au framework que le résultat de notre filtre est contrôlé et sécurisé, et qu'il n'est pas nécessaire de l'échapper. 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 passé en paramètre, encadré de guillemets français
```

```
    doubles et d'espaces insécables
    """
        return "« %s »" % texte
```

De cette façon, tout le HTML renvoyé par le filtre est correctement interprété et nous obtenons le résultat voulu (voir la figure suivante).



Le résultat correct de notre filtre

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 `{{ "<strong>Bonjour</strong> le monde !"|citation }}`, alors le mot « Bonjour » sera en gras. En soi, ce n'est pas un problème si vous êtes sûrs de la provenance de la chaîne de caractères. Il se pourrait en revanche que, parfois, vous deviez 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 éviter cela, nous allons échapper les caractères spéciaux de notre argument de base. Cela peut être fait via la fonction `espace` du module `django.utils.html`. Au final, voici ce que nous obtenons :

Code : Python

```
#-*- coding:utf-8 -*-
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, encadré de guillemets français
doubles et d'espaces insécables.
"""
    return "« %s »" % escape(texte)
```

Finalement, notre chaîne est encadrée 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 certain 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 est bien un nombre et qu'il y a des caractères à tronquer. Voici un début de fonction :

Code : Python

```
def smart_truncate(texte, nb_caracteres):

    # Nous vérifions 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 originale sinon

    # Si la chaîne est plus petite que le nombre de caractères
maximum voulus,
    # nous renvoyons directement la chaîne telle quelle.
    if len(texte) <= nb_caracteres:
            return texte

    # […]
```

La suite de la fonction est tout aussi classique : nous coupons notre chaîne au nombre de caractères maximum voulu, et nous retirons 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
souhaité,
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..."
"""

    # Nous vérifions 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 originale sinon

    # Si la chaîne est plus petite que le nombre de caractères
maximum voulus,
    # nous renvoyons directement la chaîne telle quelle.
    if len(texte) <= nb_caracteres:
            return texte

    # Sinon, nous coupons au maximum, tout en gardant le caractère
suivant
    # pour savoir si nous avons coupé à la fin d'un mot ou en plein
milieu
    texte = texte[:nb_caracteres + 1]

    # Nous vérifions d'abord que le dernier caractère n'est pas une
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 les 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 20e caractère, si aucun argument n'est passé. Dans ce cas, la méthode est classique : nous pouvons 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 contextes de templates). Le but des *template context processor* est de préremplir le contexte de la requête et ainsi de disposer de données dans tous les templates de notre projet. Le contexte est l'ensemble des variables disponibles dans votre template. Prenons l'exemple suivant :

Code : Python

```
return render(request, 'blog/archives.html', {'news': news, 'date':
date_actuelle})
```

Ici, nous indiquons au template les variables `news` et `date_actuelle` qui seront incorporées au contexte, avec les noms `news` et `date`. Cependant, par défaut notre contexte ne contiendra 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, un cache, etc.
Imaginons que vous souhaitiez 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, à partir duquel tous les autres templates seront étendus. Néanmoins, cette méthode est lourde et répétitive, c'est ici que les processeurs de contextes 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 appelée à chaque page, et qui se chargera d'incorporer la date dans les données disponibles 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 passé 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. À 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",
)
```

Nous voyons 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'avions précisé dans le premier 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.shortcut` 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.

Faites donc 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 comparés 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, nous devons *préciser comment l'écrire et ce qu'il renvoie*.

Pour mieux comprendre, regardons comment marche un template avec Django.

À 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.
Le schéma de la figure suivante récapitule tout cela.

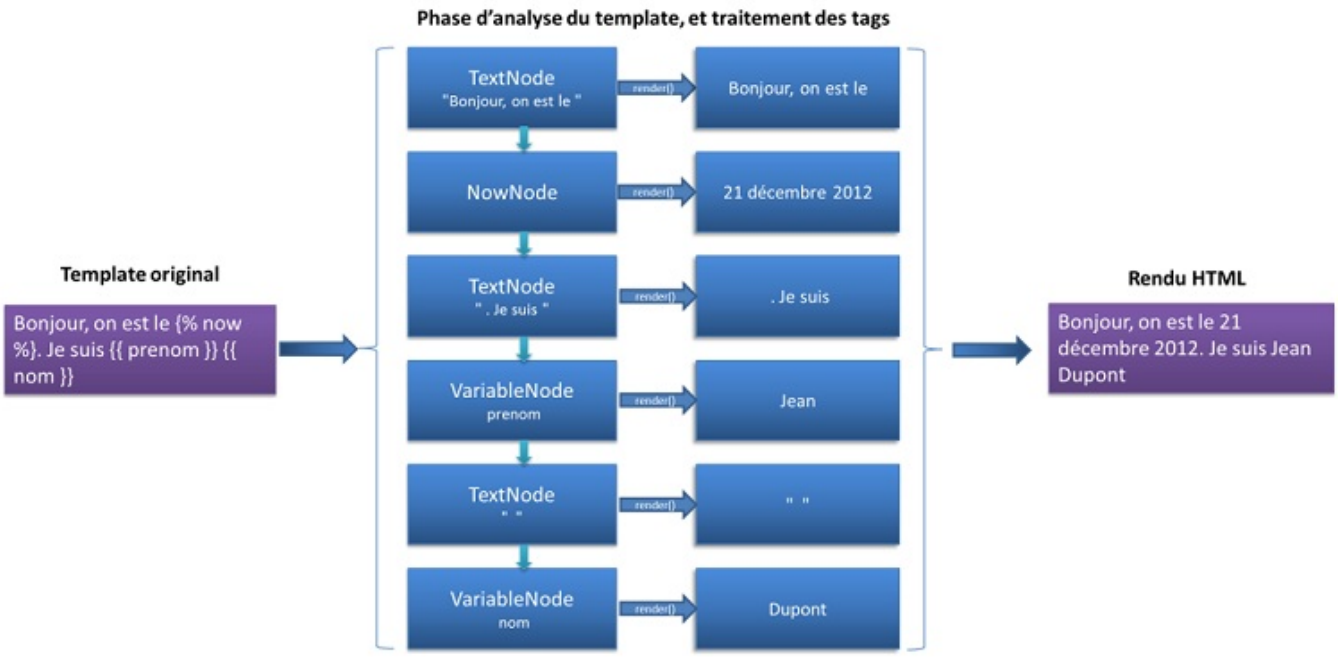


Schéma d'exécution

Lorsque nous créons un nouveau tag, la fonction appelée à la compilation doit renvoyer un objet dont la 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

À 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 : afficher un nombre aléatoire compris entre deux arguments. Cette opération est notamment impossible avec un filtre, ou du moins pas proprement.

Notre tag pourra être utilisé de la façon suivante : `{% random 0 42 %}` et renverra donc un nombre entier compris 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 prenne deux arguments précis* : `parser`, qui est l'objet en charge de parser le template actuel (que nous n'utiliserons pas ici), et `token`, qui contient les informations sur le tag actuel, comme les paramètres passés. `token` contient de plus quelques méthodes sympathiques 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 arguments """
    #Séparation des paramètres contenus 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 = u'Le tag %s doit prendre exactement deux
arguments.' % token.split_contents()[0]
            raise template.TemplateSyntaxError(msg)
```

```
    #Nous vérifions ensuite que nos deux paramètres sont bien des
entiers
    try:
        begin, end = int(begin), int(end)
    except ValueError:
        msg = u'Les arguments du tag %s sont obligatoirement des
entiers.' % nom_tag
        raise template.TemplateSyntaxError(msg)

    #Nous vérifions si le premier est inférieur au second
    if begin > end:
        msg = u'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'arguments incorrect, types des arguments invalides, etc.), alors le template ne se construira pas et *une erreur HTTP 500 sera renvoyée* au client, avec comme message d'erreur ce qui est précisé dans la variable `msg`, si jamais vous êtes en mode « debug » (voir la figure suivante).

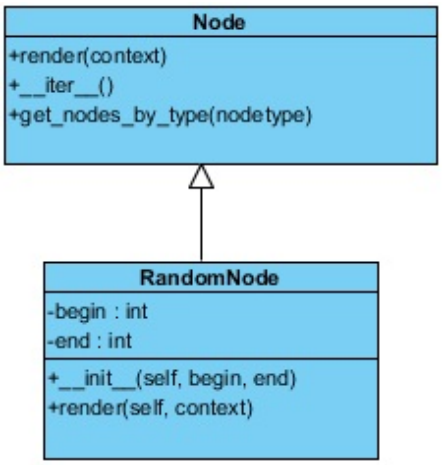


Diagramme UML de notre classe RandomNode

Il ne nous reste plus qu'à écrire la classe `RandomNode`, qui est renvoyée par la méthode ci-dessus. Vu son appel, il semble évident que sa méthode `__init__` prend trois 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 caractères, qui remplacera notre tag dans notre rendu HTML. Cette méthode prend en paramètre le contexte du template, auquel nous pouvons accéder et que nous pouvons éditer.

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))
```

Comme pour la fonction de structuration, le code en lui-même n'est pas complexe. Nous nous contentons ici de nous souvenir des arguments, et une fois que la fonction `render` est appelée, nous générons un nombre aléatoire. Il ne faut cependant pas oublier de le *transposer en chaîne de caractères*, puisque Django fait après une simple concaténation des nœuds !

Il ne nous reste plus qu'à enregistrer notre tag désormais ! Comme pour les filtres, il existe plusieurs méthodes :

- `@register.tag()` au début de notre fonction de compilation ;

- `@register.tag(name='nom_du_tag')` si jamais nous prenons 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:utf-8 -*-
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 arguments """
    #Séparation des paramètres contenus dans l'objet token
    try:
            nom_tag, begin, end = token.split_contents()
    except ValueError:
            msg = u'Le tag %s doit prendre exactement deux
arguments.' % token.split_contents()[0]
            raise template.TemplateSyntaxError(msg)

    #Nous vérifions que nos deux paramètres sont bien des entiers
    try:
            begin, end = int(begin), int(end)
    except ValueError:
            msg = u'Les arguments du tag %s sont obligatoirement des
entiers.' % nom_tag
            raise template.TemplateSyntaxError(msg)

    #Nous vérifions si le premier est bien inférieur au second
    if begin > end:
            msg = u'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 compris 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 :

Erreur 500 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 pourrons faire qu'au rendu, et non plus à la compilation du tag. Nous allons continuer sur notre tag `{% random %}`, en lui passant en paramètres deux 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 décaler 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 arguments """
    #Séparation des paramètres contenus dans l'objet token
    try:
            nom_tag, begin, end = token.split_contents()
```

```
        except ValueError:
                msg = u'Le tag random doit prendre exactement deux
arguments.'
                raise template.TemplateSyntaxError(msg)

        return RandomNode(begin, end)
```

Désormais, notre méthode `render` dans la classe `RandomNode` sera 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 *récupérer le contenu d'une variable à partir de son nom* dans le contexte. Si jamais nous lui donnons une constante, nous obtiendrons cette même constante en retour, ce qui nous permet de rester compatibles 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 = u'L\'argument "%s" n\'existe pas, ou n\'est pas
un entier.' % not_exist
               raise template.TemplateSyntaxError(msg)

            #Nous vérifions si le premier entier est bien inférieur
au second
            if self.begin > self.end:
               msg = u'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()`, nous vérifions les arguments passés. Via la classe `template`, nous récupérons le contenu de la variable ou les constantes 1 et 10 si jamais nous avons `{% random 1 10 %}`. Nous renvoyons une exception de base de Django, `VariableDoesNotExist`, si la variable n'existe pas.
- En cas d'erreur, nous renvoyons les mêmes messages d'erreur qu'avant, comme si nous étions à la compilation.
- Enfin nous vérifions toujours à la fin que la première borne est bien inférieure à la seconde, et nous retournons 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 %} où a n'existe pas
```

Les *simple tags*

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 ne se basant que sur nos deux 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(u'Les arguments doivent
nécessairement être 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 nous pouvons faire un tag en moins de lignes, et plus simplement ?

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`.

Finalement, sachez que nous n'avons pas présenté ici tous les types de tags possibles, cela serait beaucoup trop lourd et indigeste. Voici des cas spécifiques :

- Les tags composés, par exemple `{% if %} {% endif %}` ;
- Les tags incluant d'autres templates, et possédant leur propre contexte ;
- Et enfin, les tags agissant sur le contexte plutôt que de renvoyer une valeur.

Ces types de tags assez spécifiques sont en revanche décrits dans la documentation officielle, n'hésitez pas à y jeter un coup d'œil si vous en avez besoin.

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`, telle que nous l'avons vue avec les filtres.

De plus, les développeurs de Django recommandent de rester vigilants lorsque nous souhaitons garder un état interne avec les tags. En effet, certains environnements fonctionnent de façon *multithreadée*, et donc un même nœud pourrait 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, consultez la la documentation officielle.

En résumé

- Django permet aux développeurs d'étendre les possibilités des templates en créant des filtres et des tags.
- Les filtres et tags créés sont organisés par modules. Pour utiliser un filtre ou un tag il faut charger son module via `{% load nom_module %}`.
- Les filtres sont de simples fonctions, prenant en entrée 1 ou 2 arguments et renvoyant *toujours* une chaîne de caractères.
- Le contexte des templates est l'ensemble des variables disponibles et utilisables dans un template. Ce contexte est rempli par toutes les fonctions citées dans `TEMPLATE_CONTEXT_PROCESSORS`, puis par la vue appelée et enfin par les éventuels tags du template.
- Les tags permettent des traitements plus complexes sur les données à afficher. Les tags peuvent avoir une « mémoire », plusieurs arguments, former des blocs…

Les signaux et middlewares

Django délimite proprement et nettement ses différentes composantes. Il est impossible de se charger du routage des URL depuis un template, et il est impossible de 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 deux 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 ayons plusieurs fichiers sur le disque dur liés à une instance d'un modèle. Lorsque l'instance est supprimée, nous souhaitons que les fichiers associés soient également supprimés. 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 ce serait trop lourd ou que cela ne dépend simplement pas de vous). Les signaux sont la solution parfaite à ce problème.

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. 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 quel que 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 trois :

- `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 de tenter de la sauvegarder) ;
- `using` : l'alias de la base de données utilisée (si vous utilisez plusieurs bases 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 dans le dictionnaire.

Où faut-il mettre l'enregistrement des signaux ?

Nous pouvons 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, nous choisissons un `models.py` (étant donné que certains signaux agissent à partir d'actions sur des modèles, c'est plutôt un 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, voici une liste non exhaustive en contenant les principaux, avec les arguments transmis avec la notification :

Nom	Description	Arguments
`django.db.models.signals.pre_save`	Envoyé avant qu'une instance de modèle ne soit enregistrée.	• `sender` : le modèle concerné • `instance` : l'instance du modèle concernée • `using` : l'alias de la BDD utilisée • `raw` : un 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 a été enregistrée.	• `sender` : le modèle concerné • `instance` : l'instance du modèle concernée • `using` : l'alias de la BDD utilisée • `raw` : un booléen, mis à `True` si l'instance sera enregistrée telle qu'elle est présentée depuis l'argument • `created` : un 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.	• `sender` : le modèle concerné • `instance` : l'instance du modèle concernée • `using` : l'alias de la BDD utilisée
`django.db.models.signals.post_delete`	Envoyé après qu'une instance de modèle a été supprimée.	• `sender` : le modèle concerné • `instance` : l'instance du modèle concernée • `using` : l'alias de la BDD utilisée
`django.core.signals.request_started`	Envoyé à chaque fois que Django reçoit une	• `sender` : la classe qui a envoyé la requête, par exemple

	nouvelle requête HTTP.	`django.core.handlers.wsgi.WsgiHandler`
`django.core.signals.request_finished`	Envoyé à chaque fois que Django termine de répondre à une requête HTTP.	• `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, et de lui dire quels arguments le signal peut transmettre :

Code : Python

```
import django.dispatch

crepe_finie = django.dispatch.Signal(providing_args=["adresse",
"prix"])
```

Ici, nous créons un nouveau signal nommé `crepe_finie`. Nous lui indiquons une liste contenant les noms 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 !

Nous pourrions dès lors enregistrer une fonction sur ce signal, comme vu précédemment :

Code : Python

```
crepe_finie.connect(faire_livraison)
```

Lorsque nous souhaitons 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):
  # Nous préparons la crêpe pour l'expédier à l'adresse transmise
  crepe_finie.send(sender=self, adresse=adresse, prix=self.prix)
```

À chaque fois que la méthode `preparer` d'une crêpe sera appelée, la fonction `faire_livraison` le sera également avec les arguments adéquats.
Notons ici qu'il est toujours obligatoire de préciser un argument `sender` lorsque nous utilisons 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 étant 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. Pour ce faire, il faut utiliser la méthode `disconnect` du signal, 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 se situe juste avant l'appel de la vue.

En effet, le framework va à ce moment exécuter certains bouts de code appelés 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 cela aux différents moments que nous avons listés 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 formules 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 des 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)` : lorsque le code 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 que Django renvoie la réponse.
- `process_exception(self, request, exception)` : juste avant que Django renvoie une exception si une erreur s'est produite. `exception` est un objet de type **`Exception`**.

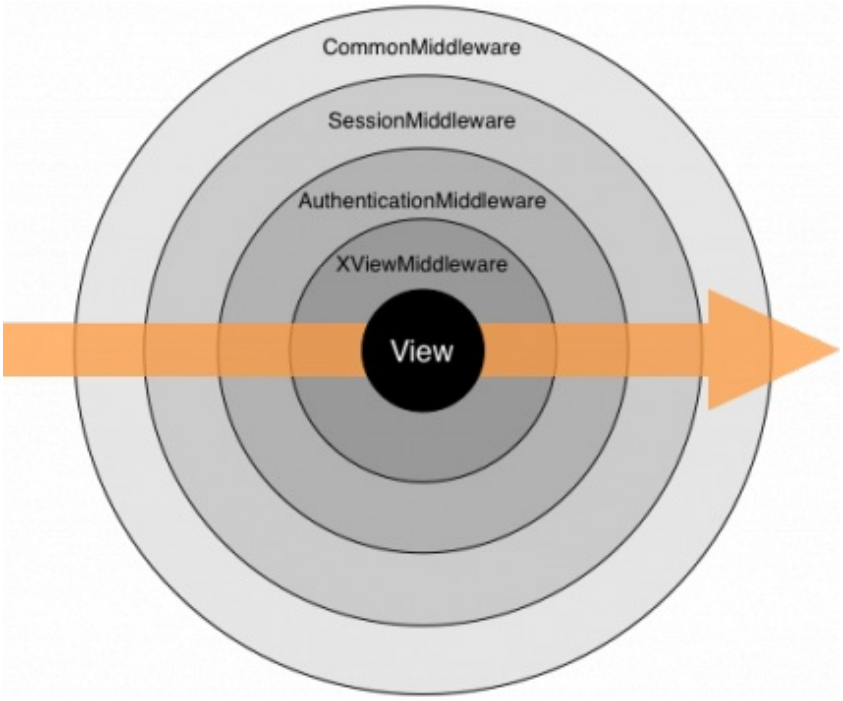
Toutes ces fonctions peuvent ne rien retourner (`None`), ou retourner un `HttpResponse`. En cas de retour vide, Django va continuer le processus normalement (appeler les autres middlewares, lancer la vue, renvoyer la réponse, etc.).

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.

Cela vaut pour toutes les méthodes susnommées, 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 middlewares 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, ainsi que vous pouvez le constater sur la figure suivante.



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 middlewares, 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 cela 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 #Nous importons 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)  # Le compteur
lié à la page est récupéré
            p.nb_visites += 1
            p.save()
        except Page.DoesNotExist:  # Si la page n'a pas encore été
consultée
            Page(url=request.path).save()  # Un nouveau compteur à 1
par défaut est créé

    def process_response(self, request, response):  # À chaque
réponse
        if response.status_code == 200:
            p = Page.objects.get(url=request.path)
            response.content += u"Cette page a été 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 s'assurant que le code HTTP de la réponse est bien 200 (ce code signifie que tout s'est bien déroulé) ; ensuite nous reprenons le compteur et nous modifions le contenu de la réponse (inclus dans `response.content`, la documentation vous donnera également tout ce qu'il faut savoir sur l'objet `HttpResponse`). Bien évidemment, si vous renvoyez du HTML au client, la phrase ajoutée ne sera pas intégrée correctement au document, néanmoins vous pouvez très bien coder quelque chose de plus sophistiqué qui permette 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 tout seule, sans devoir modifier toutes les vues une à une !

En résumé

- 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.
- Les signaux permettent d'effectuer des actions à chaque fois qu'un événement précis survient.
- Les middlewares sont des classes instanciées à chaque requête, exception, ou encore génération de template, dans l'ordre donné par `MIDDLEWARE_CLASSES`.
- Ils permettent d'effectuer une tâche précise à chaque appel.

Partie 4 : Des outils supplémentaires

Nous avons parcouru la base globale de Django au cours des trois premières parties. Nous allons désormais attaquer une partie plus diversifiée, où nous verrons un ensemble d'outils et méthodes fournis par Django afin d'accélérer le développement de votre site web.
Cette partie nécessite une bonne compréhension des chapitres précédents, n'hésitez pas à les retravailler avant de commencer cette partie !

Les utilisateurs

S'il y a bien une application très puissante et couramment utilisée que Django propose, c'est la gestion des utilisateurs. Le framework propose en effet une solution complète pour gérer le cas classique d'un accès membres, très flexible et indéniablement pratique. Nous expliquerons l'essentiel de cette application 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`.
Cela 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 e-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`, l'utilisateur est considéré comme désactivé et ne peut plus se connecter. 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` représente la date/l'heure à laquelle l'utilisateur s'est connecté la dernière fois ;
- `date_joined` : `datetime` représente la date/l'heure à laquelle l'utilisateur s'est inscrit ;
- `user_permissions` : une relation `ManyToMany` vers les permissions (introduites par la suite) ;
- `groups` : une relation `ManyToMany` vers les groupes (introduits par la suite).

Vous ne vous servirez pas nécessairement de tous ces attributs, mais ils peuvent se révéler pratiques si l'occasion de les utiliser se présente. La première question qui devrait vous venir à l'esprit est la suivante : « Est-il possible d'ajouter des attributs ? La liste est plutôt limitée. » N'ayez crainte, nous aborderons cela bientôt.

La façon la plus simple de créer un utilisateur est d'utiliser la fonction `create_user` fournie avec le modèle. Elle prend trois arguments : le nom de l'utilisateur, son adresse e-mail et son mot de passe (les trois 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 e-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 possède 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 (en clair) dans la base de données. Si un jour une personne non autorisée accède à votre base de données, elle aura 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, il faut donner le mot de passe à une fonction de hachage qui va le transformer en une autre chaîne de caractères, ce que nous appelons une « empreinte » ou un *hash*. Cette fonction est à sens unique : le même mot de passe donnera toujours la même empreinte, en revanche, nous ne pouvons 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 base de données, il ne pourra rien en faire. Chaque fois qu'un utilisateur voudra se connecter, il suffira d'appliquer la même fonction de hachage au mot de passe fourni lors de la connexion, et de 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 « m0nsup3rm0td3p4ss3 » 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 utilisée pour le mot de passe (ici `pbkdf2_sha256`, la fonction de hachage par défaut de Django 1.5) ;
- Sel : le sel est une chaîne de caractères insérée dans le mot de passe originel pour rendre son déchiffrage plus difficile (ici 10000). Django s'en charge tout seul, 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 quatre 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. Cette méthode ne sauvegarde pas l'entrée dans la base de données, il faut faire un `.save()` par la suite.
- `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")     # Nous changeons le mot de passe
>>> user.check_password("salut")    # Nous essayons un mot de passe
invalide
```

```
False
>>> user.check_password("coucou")   # Avec le bon mot de passe, ça
marche !
True
>>> user.set_unusable_password()    # Nous désactivons 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, ce qui pourrait se révéler embêtant si nous souhaitons par exemple adjoindre un avatar à chaque utilisateur.

Une solution répandue pour étendre le modèle `User` est d'utiliser un autre 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 ajoutant un `OneToOneField` vers ce dernier.

Imaginons que nous souhaitions 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 modèle 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 u"Profil de {0}".format(self.user.username)
```

Les différents attributs que nous avons listés sont bel et bien repris dans notre modèle `Profil`, dont notamment la liaison vers le modèle `User`.

N'oubliez pas que vous pouvez accéder à l'instance `Profil` associée à une instance `User` depuis cette dernière en utilisant la relation inverse créée automatiquement par le `OneToOneField`. Pour illustrer le fonctionnement de cette relation inverse, 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')  # Nous créons 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()  # Nous l'enregistrons
>>> profil
<Profil: Profil de Mathieu>
>>> user.profil
<Profil: Profil de Mathieu>
>>> user.profil.signature
u"Coucou ! C'est moi !"
```

Voilà ! Le modèle `User` est désormais correctement étendu avec les nouveaux attributs et méthodes de `Profil`.

Passons aux vues

Maintenant que nous avons assimilé les bases, il est temps de passer aux vues permettant à un utilisateur de se connecter, déconnecter, etc. Nous n'aborderons pas ici l'enregistrement de nouveaux utilisateurs. En effet, nous avons déjà montré 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 que 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 ajoutée 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 auparavant tout ce qu'elle doit faire :

1. Afficher le formulaire ;
2. Après la saisie par 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 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 de `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 le mot de passe correspondant, 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

```
from django.contrib.auth import authenticate, login

def connexion(request):
    error = False

    if request.method == "POST":
        form = ConnexionForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data["username"]  # Nous
récupérons le nom d'utilisateur
            password = form.cleaned_data["password"]  # … et le mot
de passe
            user = authenticate(username=username,
password=password)  #Nous vérifions si les données sont correctes
            if user:  # Si l'objet renvoyé n'est pas None
                login(request, user)  # nous connectons
l'utilisateur
            else: #sinon une erreur sera affichée
                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 grâce à la commande `manage.py createsuperuser`, soit 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 connectés !

La déconnexion

Heureusement, la déconnexion est beaucoup plus simple que la connexion. En effet, il suffit d'appeler la fonction `logout` de `django.contrib.auth`. Il n'y a même pas besoin de vérifier si le visiteur est connecté ou non (mais libre à vous de le faire si vous souhaitez ajouter 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 qu'un processeur de contexte se chargeait d'ajouter 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 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 souhaitions autoriser l'accès de certaines vues *uniquement* aux utilisateurs connectés. Nous pourrions vérifier si l'utilisateur est connecté, et si ce n'est pas le cas le rediriger vers une autre page, mais cela serait lourd et redondant. Pour éviter de se répéter, Django fournit un petit décorateur très pratique qui nous permet de nous assurer qu'uniquement des visiteurs authentifiés accèdent à la vue. Son nom est `login_required` et il se situe dans `django.contrib.auth.decorators`. En voici un exemple d'utilisation :

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 - Extrait de settings.py

```
LOGIN_URL = '/connexion/'
```

Il faut savoir que si l'utilisateur n'est pas connecté, non seulement il sera redirigé vers `'/connexion/'`, mais l'URL complète de la redirection sera `"/connexion/?next=/bonjour/"`. En effet, Django ajoute 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 utilisateurs 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 dédié ; nous ne ferons donc ici que les lister, avec leurs paramètres et modes 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 : aucun.

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 l'e-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 de l'e-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 e-mail valide depuis laquelle sera envoyé l'e-mail (par défaut, Django utilise `settings.DEFAULT_FROM_EMAIL`).

Contexte du template :

- `form` : le formulaire à afficher.

Contexte de l'e-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 e-mail du compte à récupérer. L'utilisateur recevra alors un e-mail (il est important de configurer l'envoi d'e-mails, référez-vous à l'annexe sur le déploiement en production pour davantage d'informations à ce sujet) avec un lien vers la vue de confirmation de réinitialisation du mot de passe.

Voici un exemple du template pour générer l'e-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 de 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 réinitalisation 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 : aucun.

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 les groupes

Le système utilisateurs 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 automatiquement trois permissions pour chaque modèle enregistré. Ces permissions sont notamment utilisées dans l'administration. Si nous reprenons par exemple le modèle `Article` de l'application `blog`, trois 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 renseignée 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 vos permissions, 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 *context processor* :

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 (ce 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 nous pouvons 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 cela fait, l'utilisateur « Mathieu » dispose de toutes les permissions attribuées au groupe « Les gens géniaux ».

Voilà ! Vous avez désormais vu de long en large le système utilisateurs que propose Django. Vous avez pu remarquer qu'il est tout aussi puissant que flexible. Inutile donc de réécrire tout un système utilisateurs lorsque le framework en propose déjà un plus que correct.

En résumé

- Django propose une base de modèles qui permettent de décrire les utilisateurs et groupes d'utilisateurs au sein du projet. Ces modèles possèdent l'ensemble des fonctions nécessaires pour une gestion détaillée des objets : `make_password`, `check_password`...
- Il est possible d'étendre le modèle d'utilisateur de base, pour ajouter ses propres champs.
- Le framework dispose également de vues génériques pour la création d'utilisateurs, la connexion, l'inscription, la déconnexion, le changement de mot de passe… En cas de besoin plus spécifique, il peut être nécessaire de les réécrire soi-même.
- Il est possible de restreindre l'accès à une vue aux personnes connectées via `@login_required` ou à un groupe encore plus précis via les permissions.

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 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. Elles le sont 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'`.

Cela étant fait, nous pouvons entrer dans le vif du sujet.

Django peut envoyer des notifications (aussi appelées messages) à tous les visiteurs, qu'ils soient connectés ou non. Il existe plusieurs niveaux de messages par défaut (nous verrons comment en ajouter par la suite) :

- `DEBUG` (debug) : message destiné pour la phase de développement uniquement. Ces messages ne seront affichés que si `DEBUG=True` dans votre `settings.py`.
- `INFO` (info) : message 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 entre parenthèses sont ce que nous appelons 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, u'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, u'%s requêtes SQL ont été exécutées.' %
compteur)
messages.info(request, u'Rebonjour !')
messages.success(request, u'Votre article a bien été mis à jour.')
messages.warning(request, u'Votre compte expire dans 3 jours.')
messages.error(request, u'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 de l'itérer, étant donné qu'elle peut contenir plusieurs notifications. À 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 une 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 messages 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

Au final, ajouter un niveau suffit juste à créer une nouvelle constante. Par exemple :

Code : Python

```
CRITICAL = 50

messages.add_message(request, CRITICAL, u'Une erreur critique est
survenue.')
```

Il est dès lors tout aussi possible d'ajouter des tags à un message :

Code : Python

```
messages.add_message(request, CRITICAL, u'Une erreur critique est
survenue.', extra_tags="fail")
```

Ici, le tag « fail » sera ajouté.

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`), soit en faisant cette requête dans la vue :

Code : Python

```
    messages.set_level(request, messages.DEBUG)
```

En utilisant cela et pour cette vue uniquement, tous les messages dont le niveau est égal ou supérieur à 10 (la valeur de `messages.DEBUG`) seront affichés.

En résumé

- Les messages permettent d'afficher des notifications à l'utilisateur, en cas de succès ou d'échec lors d'une action, ou si un événement particulier se produit.
- Il existe différents types de messages : ceux d'information, de succès, d'erreur, d'avertissement, et enfin de débogage.
- Il est possible de créer son propre type de message et de configurer son comportement.
- L'affichage de messages peut être limité : chaque type de message est caractérisé par une constante entière, et nous pouvons afficher les messages ayant un niveau supérieur ou égal à un certain seuil, via `messages.set_level`.

La mise en cache

Lorsqu'un site devient populaire, que son trafic augmente et que toutes les ressources du serveur sont utilisées, nous constatons 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 d'effectuer les calculs une fois, de 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, or il dispose de plusieurs systèmes de cache différents de base. Nous tâcherons de les introduire brièvement dans ce qui suit.

Chacun de ces systèmes a ses avantages et désavantages, et tous fonctionnent un peu 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 du serveur. 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 vers un fichier spécifique. 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 différente 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 ce système :

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 d'entrer 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. Cela 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 cela 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 Memcached

Memcached 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. Memcached va enregistrer les données dans la mémoire vive, comme le système vu précédemment qui utilisait la même technique, sauf qu'en comparaison de ce dernier Memcached est bien plus efficace et utilise moins de mémoire.

Memcached n'existe officiellement que sous Linux. Si vous êtes sous Debian ou dérivés, vous pouvez l'installer grâce à `apt-get install memcached`. Pour les autres distributions, référez-vous à la liste des paquets fournis par votre gestionnaire de paquets. Une fois que Memcached est installé, vous pouvez lancer le démon en utilisant la commande `memcached -d -m 512 -l 127.0.0.1 -p 11211`, où le paramètre `-d` permet le lancement du démon, `-m` indique la taille maximale de mémoire vive allouée au cache (en mégaoctets), et `-l` et `-p` donnent respectivement l'adresse IP et le port d'écoute du démon.

La configuration côté Django 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 Memcached est accessible. Nous avons adapté la valeur de la variable à la commande indiquée ci-dessus.

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 Memcached) 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, Memcached 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 Memcached, 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 est calculé, il 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 15 fois 60 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 restera 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 carrousel %}
/* mon carrousel */
{% endcache %}
```

Ici, nous enregistrons dans le cache notre carrousel. Celui-ci expirera dans 500 secondes et nous utilisons la clé `carrousel`.

Sachez que vous pouvez également enregistrer plusieurs copies en cache d'une même partie de template dépendant de plusieurs variables. Par exemple, si notre carrousel est différent pour chaque utilisateur, vous pouvez réutiliser une clé dynamique et différente pour chaque utilisateur. Ainsi, chaque utilisateur connecté aura dans le cache la copie du carrousel adaptée à son profil. Exemple :

Code : Jinja

```
{% load cache %}
{% cache 500 user.username %}
/* mon carrousel adapté à l'utilisateur actuel */
{% endcache %}
```

La mise en cache de 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 de l'objet `cache` du module `django.core.cache`. Cet objet `cache` se comporte un peu comme un dictionnaire. Nous pouvons 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 ou plus :

Code : Python

```
>>> cache.get('ma_cle', u'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 découvert 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.

En résumé

- Le cache permet de sauvegarder le résultat de calculs ou traitements relativement longs, afin de présenter le résultat sauvegardé pour les prochaines visites, plutôt que de recalculer la même donnée à chaque fois.
- Il existe plusieurs systèmes de mise en cache : par fichier, en base de données, dans la mémoire RAM.
- La mise en cache peut être définie au niveau de la vue, via `@cache_page`, dans le fichier `urls.py`, ou encore dans les templates avec le tag `{% cache %}`.
- Django fournit également un ensemble de fonctions permettant d'appliquer une mise en cache à tout ce que nous souhaitons, et d'en gérer précisément l'expiration de la validité.

La pagination

Un autre outil fréquemment utilisé sur les sites web de nos jours est la pagination. La pagination est le fait de *diviser une liste d'objets en plusieurs pages*, afin d'alléger la lecture d'une page (et son chargement). Nous allons voir dans ce chapitre comment réaliser une pagination avec les outils que Django propose.

Exerçons-nous en console

Django permet de *répartir des ensembles d'objets sur plusieurs pages* : des listes, des `QuerySet`, etc. En réalité, tous les objets ayant une méthode `count` ou `__len__` sont acceptés.

Notre premier exemple utilisera une simple liste et sera effectué dans l'interpréteur interactif. Ouvrez une console et tapez la commande `python manage.py shell` pour lancer l'interpréteur.

Django fournit une classe nommée `Paginator` qui effectue la pagination. Elle se situe dans le module `django.core.paginator`. Nous devons donc en premier lieu l'importer :

Code : Python

```
>>> from django.core.paginator import Paginator
```

Ensuite, créons une liste quelconque. Nous avons sélectionné une liste de grandes villes :

Code : Python

```
>>> from django.core.paginator import Paginator
>>> villes = ['Tokyo','Mexico','Seoul','New
York','Bombay','Karachi','Sao Paulo','Manille','Bangkok',
'New Delhi','Djakarta','Shanghai','Los Angeles','Kyoto','Le
Caire','Calcutta','Moscou','Istanbul',
'Buenos Aires','Dacca','Gauteng','Teheran','Pekin']
```

La classe `Paginator` est instanciée avec deux paramètres : la liste d'objets à répartir et le nombre maximum d'objets à afficher par page. Imaginons que nous souhaitions afficher 5 villes par page :

Code : Python

```
>>> p = Paginator(villes, 5)
```

Nous venons d'instancier un objet `Paginator`. Cet objet possède les attributs suivants :

Code : Python

```
>>> p.count      #Nombre d'objets au total, toutes pages confondues
23
>>> p.num_pages  #Nombre de pages nécessaires pour répartir toutes
les villes
5                #En effet, 4 pages avec 5 villes et 1 page avec 3
villes

>>> p.page_range  #La liste des pages disponibles
[1, 2, 3, 4, 5]
```

Nous pouvons obtenir les villes d'une page précise grâce la méthode `page()`. Cette méthode renvoie un objet `Page`, dont voici les méthodes principales :

Code : Python

```
>>> page1 = p.page(1) #Renvoie un objet Page pour notre première
page
>>> page1
<Page 1 of 5>
>>> page1.object_list
['Tokyo', 'Mexico', 'Seoul', 'New York', 'Bombay'] #Le contenu de
cette première page
>>> p.page(5).object_list #Même opération pour la cinquième page
['Gauteng', 'Teheran', 'Pekin']
>>> page1.has_next()      #Est-ce qu'il y a une page suivante ?
True #Oui
>>> page1.has_previous()  #Est-ce qu'il y a une page précédente ?
False #Non
```

Soyez vigilants, la numérotation des pages commence bien à 1, et non pas à 0 comme pour les listes par exemple. Remarquez les comportements suivants :

Code : Python

```
>>> p.page(0)
Traceback (most recent call last):
   […]
django.core.paginator.EmptyPage: That page number is less than 1
>>> p.page(6)
Traceback (most recent call last):
   […]
django.core.paginator.EmptyPage: That page contains no results
>>> p.page('abc')
Traceback (most recent call last):
  […]
django.core.paginator.PageNotAnInteger: That page number is not an
integer
```

Avant d'attaquer l'utilisation de la pagination dans nos vues et templates, étudions deux autres situations permettant de compléter au mieux notre système de pagination. Il y a deux arguments de `Paginator` que nous n'avons pas traités. En effet, le constructeur complet de `Paginator` accepte deux paramètres optionnels.

Tout d'abord, le paramètre `orphans` permet de préciser le nombre minimum d'éléments qu'il faut pour afficher une dernière page. Si le nombre d'éléments est inférieur au nombre requis, alors ces éléments sont déportés sur la page précédente (qui devient elle-même la dernière page), en plus des éléments qu'elle contient déjà. Prenons notre exemple précédent :

Code : Python

```
>>> p = Paginator(villes, 10, 5)
>>> p.num_pages
2
>>> p.page(1).object_list
['Tokyo', 'Mexico', 'Seoul', 'New York', 'Bombay', 'Karachi', 'Sao
Paulo', 'Manille', 'Bangkok', 'New Delhi']
>>> p.page(2).object_list
['Djakarta', 'Shanghai', 'Los Angeles', 'Kyoto', 'Le Caire',
'Calcutta', 'Moscou', 'Istanbul', 'Buenos Aires', 'Dacca',
'Gauteng', 'Teheran', 'Pekin']
```

Nous voyons que la dernière page théorique (la 3e) aurait du contenir 3 éléments (`Gauteng`, `Teheran` et `Pekin`), ce qui est inférieur à 5. Ces éléments sont donc affichés en page 2, qui devient la dernière, avec 13 éléments.

Le dernier attribut, `allow_empty_first_page`, permet de lancer une exception si la première page est vide. Autrement dit, une exception est levée s'il n'y a aucun élément à afficher. Un exemple est encore une fois plus parlant :

Code : Python

```
>>> pt = Paginator([], 42)
>>> pf = Paginator([], 42, 0, False)  #Nous initialisons deux
Paginator avec une liste vide
>>> pt.page(1)  #Comportement par défaut si la liste est vide
<Page 1 of 1>
>>> pt.page(1).object_list
[]
>>> pf.page(1)
Traceback (most recent call last):
 […]
django.core.paginator.EmptyPage: That page contains no results
```

Nous avons désormais globalement fait le tour, place à la pratique !

Utilisation concrète dans une vue

Nous avons vu comment utiliser la pagination de façon autonome, maintenant nous allons l'utiliser dans un cas concret. Nous reprenons notre vue simple (pas celle utilisant les vues génériques) du TP sur la minification d'URL :

Code : Python

```
def liste(request):
    """Affichage des redirections"""
    minis = MiniURL.objects.order_by('-nb_acces')

    return render(request, 'mini_url/liste.html', locals())
```

Nous allons tout d'abord ajouter un argument `page` à notre vue, afin de savoir quelle page l'utilisateur souhaite voir. Pour ce faire, il y a deux méthodes :

- Passer le paramètre `page` via un paramètre `GET` (`/url/?page=1`) ;
- Modifier la définition de l'URL et la vue pour prendre en compte un numéro de page (`/url/1` pour la première page).

Nous traiterons ici le second cas. Le premier cas se résume à un simple `request.GET.get('page')` dans la vue pour récupérer le numéro de page. Nous modifions donc légèrement notre vue pour le paramètre `page` :

Code : Python - mini_url/views.py

```
def liste(request, page=1):
   """Affichage des redirections"""
   minis = MiniURL.objects.order_by('-nb_acces')

   return render(request, 'mini_url/liste.html', locals())
```

Et notre fichier `urls.py` :

Code : Python - mini_url/urls.py

```
urlpatterns = patterns('mini_url.views',
    url(r'^$', 'liste', name='url_liste'),    # Pas d'argument page
précisé -> vaudra 1 par défaut
    url(r'^(?P<page>\d)$', 'liste', name='url_liste'),
    # …
```

Nous créons donc un objet `Paginator` à partir de cette liste, comme nous avons pu le faire au début de ce chapitre. Nous avons également vu que `Paginator` permettait de récupérer les objets d'une page précise : c'est ce que nous utiliserons désormais pour renvoyer au template la liste d'URL à afficher.

Code : Python

```
from django.core.paginator import Paginator, EmptyPage  # Ne pas
oublier l'importation

def liste(request, page=1):
    """Affichage des redirections"""
    minis_list = MiniURL.objects.order_by('-nb_acces')
    paginator = Paginator(minis_list, 5)  # 5 liens par page

    try:
            # La définition de nos URL autorise comme argument «
page » uniquement des entiers,
            # nous n'avons pas à nous soucier de l'erreur
PageNotAnInteger
            minis = paginator.page(page)
    except EmptyPage:
            # Nous vérifions toutefois que nous ne dépassons pas la
limite de page
            # Par convention, nous renvoyons la dernière page dans
ce cas
            minis = paginator.page(paginator.num_pages)

    return render(request, 'mini_url/liste.html', locals())
```

En ajoutant 5 lignes de code (sans prendre en compte les commentaires), nous disposons désormais d'une pagination robuste, gérant tous les cas limites. Si vous testez la vue actuellement, vous verrez que l'adresse http://127.0.0.1:8000/m/ (attention, l'URL dépend de la configuration que vous avez utilisée durant le TP) renvoie les 5 premières URL (ajoutez-en si vous en avez moins), http://127.0.0.1:8000/m/2 les 5 suivantes, etc.

Passons désormais au template. En effet, pour l'instant il est impossible de passer d'une page à l'autre sans jouer avec l'URL et il est impossible de savoir le nombre de pages qu'il y a. Pour renseigner toutes ces informations, nous allons utiliser les informations que nous avons vues précédemment :

Code : Jinja

```
<h1>Le raccourcisseur d'URL spécial crêpes bretonnes !</h1>

<p><a href="{% url 'url_nouveau' %}">Raccourcir une URL.</a></p>

<p>Liste des URL 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 }} accès)</li>
{% empty %}
<li>Il n'y en a pas actuellement.</li>
{% endfor %}
</ul>

<div class="pagination">
<span class="step-links">
{% if minis.has_previous %}
<a href="{% url 'url_liste' minis.previous_page_number
%}">Précédente</a> -
{% endif %}

<span class="current">
Page {{ minis.number }} sur {{ minis.paginator.num_pages }}
</span>

{% if minis.has_next %}
- <a href="{% url 'url_liste' minis.next_page_number
%}">Suivante</a>
{% endif %}
```

```
    </span>
    </div>
```

Nous utilisons bien ici les méthodes `has_next` et `has_previous` pour savoir s'il faut afficher les liens « Précédent » et « Suivant ». Nous profitons également de l'attribut `num_pages` de `Paginator` afin d'afficher le total de pages.

Un bon conseil que nous pouvons vous donner, et en même temps un bon exercice à faire, est de créer un template générique gérant la pagination et de l'appeler où vous en avez besoin, via `{% include "pagination.html" with liste=minis view="url_liste" %}`.

Vous pouvez maintenant adapter la pagination comme vous voulez en modifiant la ligne appelant `Paginator` !

Code : Python

```
paginator = Paginator(minis_list, 20, 5)  # 20 liens par page, avec
un minimum de 5 liens sur la dernière
```

Nous en avons fini avec la pagination. Ce module est l'exemple le plus frappant de ce que nous pouvons faire avec Django en seulement quelques lignes, tout en changeant très peu de code par rapport à la base de départ.

En résumé

- La classe `django.core.paginator.Paginator` permet de générer la pagination de plusieurs types de listes d'objets et s'instancie avec au minimum une liste et le nombre d'éléments à afficher par page.
- Les attributs et méthodes clés de `Paginator` à retenir sont `p.num_pages` et `p.page()`. La classe `Page` a notamment les méthodes `has_next()`, `has_previous()` et est itérable afin de récupérer les objets de la page courante.
- Il est possible de rendre la pagination plus pratique en prenant en compte l'argument `orphans` de `Paginator`.
- Pensez à uniformiser vos paginations en terme d'affichage au sein de votre site web, pour ne pas perturber vos visiteurs.

L'internationalisation

De nos jours, la plupart des sites web proposent plusieurs langues à leurs utilisateurs, et ciblent même la langue par défaut en fonction du visiteur. Ce concept apporte son lot de problèmes lors de la réalisation d'un site : que faut-il traduire ? Quelle méthode faut-il utiliser pour traduire facilement l'application, sans dupliquer le code ?

Nous allons voir dans ce chapitre comment traduire notre site en plusieurs langues, de façon optimale sans dupliquer nos templates et vues, via des méthodes fournies dans Django et l'outil **gettext** permettant de créer des fichiers de langue.

Sachez que par convention en informatique le mot « internationalisation » est souvent abrégé par « i18n » ; cela est dû au fait que 18 lettres séparent le « i » du « n » dans ce mot si long à écrire ! Nous utiliserons également cette abréviation tout au long de ce chapitre.

Qu'est-ce que le i18n et comment s'en servir ?

Avant de commencer, vous devez avoir installé gettext sur votre machine.

Sous Mac OS X

Vous devez télécharger le code source et le compiler, ou installer le paquet gettext à partir des MacPorts.

Sous Linux

Gettext est généralement installé par défaut. Si ce n'est pas le cas, cherchez un paquet nommé « gettext » adapté à votre distribution, ou procédez à l'installation manuelle, comme pour Mac OS X.

Sous Windows

Voici la méthode d'installation complète :

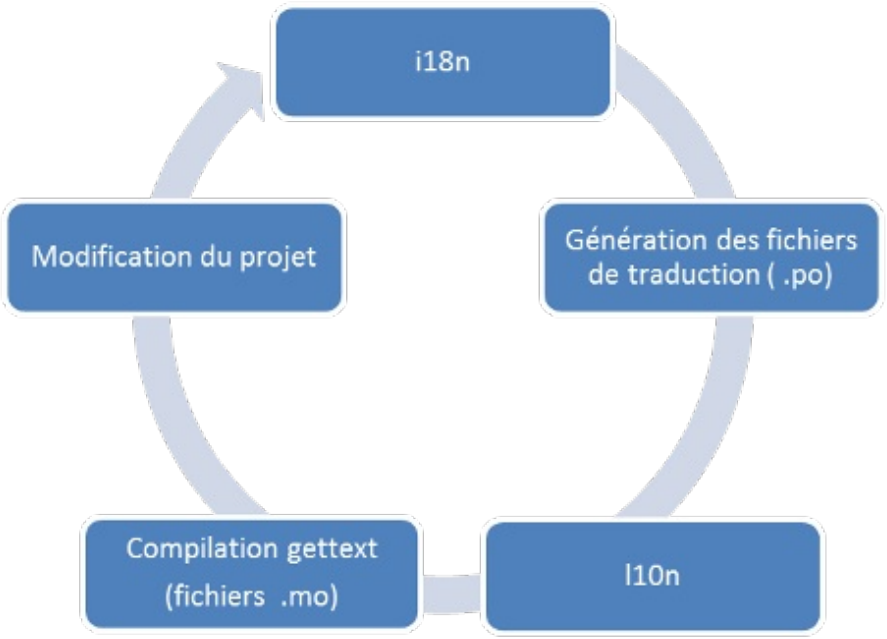
1. Téléchargez les archives suivantes depuis le serveur GNOME, ou un de ses miroirs (avec X supérieur ou égal à 0.15) :
   - `gettext-runtime-X.zip`
   - `gettext-tools-X.zip`
2. Extrayez le contenu des deux dossiers dans un répertoire `\bin` commun (par exemple `C:\Program Files\gettext\bin`).
3. Mettez à jour votre variable d'environnement `PATH` (voir le chapitre d'installation de Django pour plus d'informations) en ajoutant `;C:\Program Files\gettext\bin` à la fin de la valeur de la variable.

Vérifiez en ouvrant une console que la commande `xgettext --version` fonctionne sans erreur. En cas d'erreur, retéléchargez gettext via les liens précédents !

Pour commencer, nous devons nous attarder sur quelques définitions, afin d'avoir les idées au clair. Dans l'introduction, nous avons parlé d'*internationalisation*, dont le but est de traduire une application dans une autre langue. En réalité, ce travail se divise en deux parties :

1. L'internationalisation (i18n) à proprement parler, qui consiste à adapter la partie technique (le code même de l'application) afin de permettre la traduction littérale par la suite ;
2. La localisation (l10n), qui est la traduction (et parfois l'adaptation culturelle) de l'application.

La figure suivante schématise les différentes étapes.



Cycle de traduction d'un logiciel

Tout au long de ce chapitre, nous allons suivre le déroulement du cycle de la figure précédente, dont tous les détails vous seront expliqués en temps et en heure. Mais tout d'abord, il nous faut configurer un peu notre projet, via `settings.py`.

Par défaut, lors de la création du projet, Django prédéfinit trois variables concernant l'internationalisation et la localisation :

Code : Python

```
# Language code for this installation. All choices can be found
here:
# http://www.i18nguy.com/unicode/language-identifiers.html
LANGUAGE_CODE = 'fr-fr'

# If you set this to False, Django will make some optimizations so
as not
# to load the internationalization machinery.
USE_I18N = True

# If you set this to False, Django will not format dates, numbers
and
# calendars according to the current locale
USE_L10N = True
```

`LANGUAGE_CODE` permet de définir la langue par défaut utilisée dans vos applications. La variable `USE_I18N` permet d'activer l'internationalisation, et donc de charger plusieurs éléments en interne permettant son fonctionnement. Si votre site n'utilise pas l'internationalisation, il est toujours bon d'indiquer cette variable à `False`, afin d'éviter le chargement de modules inutilisés. De la même manière, `USE_L10N` permet d'automatiquement formater certaines données en fonction de la langue de l'utilisateur : les dates ou encore la représentation des nombres.

Par exemple, la ligne `{% now "DATETIME_FORMAT" %}` renvoie des résultats différents selon la valeur de `USE_L10N` et de `LANGUAGE_CODE` :

LANGUAGE_CODE/USE_L10N	False	True
en-us	Jan. 30, 2013, 9:50 p.m.	Jan. 30, 2013, 9:50 p.m.
fr-fr	jan. 30, 2013, 9:50 après-midi	30 janvier 2013 21:50:46

Ensuite, nous devons préciser la liste des langues disponibles pour notre projet. Cette liste nous permet d'afficher un formulaire de choix, mais permet aussi à Django de limiter les choix possibles et donc d'éviter de chercher une traduction dans une langue

qui n'existe pas. Cette liste se présente comme ceci :

Code : Python

```
gettext = lambda x: x

LANGUAGES = (
    ('fr', gettext('French')),
    ('en', gettext('English')),
)
```

Au lieu d'importer la fonction `gettext`, introduite par la suite, nous avons créé une fausse fonction qui ne fait rien. En effet, il ne faut pas importer les fonctions de `django.utils.translation` dans notre fichier de configuration, car ce module dépend de notre fichier, ce qui créerait une boucle infinie dans les importations de modules !

Pour la suite de ce chapitre, il sera nécessaire d'avoir les réglages suivants dans votre `settings.py` :

Code : Python

```
#Nous avons par défaut écrit l'application en français
LANGUAGE_CODE = 'fr-fr'

#Nous souhaitons générer des fichiers contenant les traductions,
afin de permettre à l'utilisateur de choisir sa langue par la suite
USE_I18N = True

#Nous adaptons les formats d'écriture de certains champs à la
langue française
USE_L10N = True

gettext = lambda x: x

LANGUAGES = (
    ('fr', gettext('French')),
    ('en', gettext('English')),
)
```

Finalement, il nous faut encore ajouter deux lignes : un **middleware** et un **processeur de contexte**. Le premier permet de déterminer selon un certain ordre la langue courante de l'utilisateur. En effet, Django va tenter pour chaque visiteur de trouver la langue la plus adaptée en procédant par étape :

1. Dans un premier temps, il est possible de configurer les URL pour les préfixer avec la langue voulue. Si ce préfixe apparaît, alors la langue sera forcée.
2. Si aucun préfixe n'apparaît, le middleware vérifie si une langue est précisée dans la session de l'utilisateur.
3. En cas d'échec, le middleware vérifie dans les cookies du visiteur si un cookie nommé `django_language` (nom par défaut) existe.
4. En cas d'échec, il vérifie la requête HTTP et vérifie si l'en-tête `Accept-Language` est envoyé. Cet en-tête, envoyé par le navigateur du visiteur, spécifie les langues de prédilection, par ordre de priorité. Django essaie chaque langue une par une, selon celles disponibles dans notre projet.
5. Enfin, si aucune de ces méthodes ne fonctionne, alors Django se rabat sur le paramètre `LANGUAGE_CODE`.

Le second, le ***template context processor***, nous permettra d'utiliser les fonctions de traduction, via des tags, dans nos templates.

Pour activer correctement l'internationalisation, nous devons donc modifier la liste des middlewares et contextes à charger :

Code : Python

```
MIDDLEWARE_CLASSES = (
    […] # Liste des autres middlewares déjà chargés
    'django.middleware.locale.LocaleMiddleware',
)
TEMPLATE_CONTEXT_PROCESSORS = (
```

```
        […] # Liste des autres template context déjà chargés
        "django.core.context_processors.i18n",
    )
```

Nous en avons terminé avec les fichiers de configuration et pouvons dès lors passer à l'implémentation de l'internationalisation dans notre code !

Traduire les chaînes dans nos vues et modèles

Adaptons notre code pour l'internationalisation. Nous devons gérer deux cas distincts : les vues et modèles, dans lesquels nous pouvons parfois avoir des chaînes de caractères à internationaliser, et nos templates, qui contiennent également de nombreuses chaînes de caractères à traduire. Notez qu'ici nous parlerons toujours de traduire dans une autre langue sans pour autant préciser laquelle. En effet, nous nous occupons de préciser ici ce qui doit être traduit, sans spécifier pour autant les différentes traductions.

Commençons par les vues. Pour rendre traduisibles les chaînes de caractères qui y sont présentes, nous allons appliquer une fonction à chacune. Cette fonction se chargera ensuite de renvoyer la bonne traduction selon la langue de l'utilisateur. Si la langue n'est pas supportée ou si la chaîne n'a pas été traduite, la chaîne sera alors affichée dans la langue par défaut.

La bibliothèque dont provient cette fonction spéciale est bien connue dans le domaine de l'internationalisation, puisqu'il s'agit de **gettext**, une bibliothèque logicielle dédiée à l'internationalisation. Dans Django, celle-ci se décompose en plusieurs fonctions que nous allons explorer dans cette partie :

- `gettext` et `ugettext` ;
- `gettext_lazy` et `ugettext_lazy` ;
- `ngettext` et `ungettext` ;
- `ngettext_lazy` et `ungettext_lazy` ;
- Etc.

Avant d'explorer les fonctions une à une, sachez que nous allons tout d'abord diviser leur nombre par 2 : si vous regardez attentivement cette liste, vous remarquez que chacune existe avec et sans « u » comme première lettre. Les fonctions commençant par « u » signifient qu'elles supportent l'unicode, alors que les autres retournent des chaînes en ASCII. Nous partons du principe que nous utiliserons uniquement celles préfixées d'un « u », étant donné que la majorité de nos chaînes de caractères utilisent l'unicode.

Pour commencer, créons d'abord une nouvelle vue, qui renverra plusieurs chaînes de caractères au template :

Code : Python

```
def test_i18n(request):
    nb_chats = 1
    couleur = "blanc"
    chaine = u"Bonjour les zéros !"
    ip = u"Votre IP est %s" % request.META['REMOTE_ADDR']
    infos = u"… et selon mes informations, vous avez %s chats %s !"
% (nb_chats, couleur)

    return render(request, 'test_i18n.html', locals())
```

Et voici le fichier `test_i18n.html` :

Code : HTML

```
<p>
  Bonjour les zéros !<br />
  {{ chaine }}
</p>
<p>
  {{ ip }} {{ infos }}
</p>
```

Tout au long de ce chapitre, nous ne vous indiquerons plus les directives de routage de `urls.py`. Vous devriez en

effet être capables de faire cela vous-mêmes.

Ce fichier contient plusieurs chaînes écrites en français, qu'il faudra nécessairement traduire par la suite. Nous remarquons par ailleurs que certaines peuvent poser problème au niveau des formes du pluriel (dans le cas présent, il ne devrait pas y avoir de « s » à « chats », étant donné qu'il n'y en a qu'un seul !)

Tout d'abord, nous allons importer `ugettext`, afin de pouvoir utiliser la fonction dont nous avons parlé plus haut. Nous pourrions l'importer de la sorte :

Code : Python

```
from django.utils.translation import ugettext
```

Cependant, nous serons amenés à utiliser très souvent cette méthode, et tout le temps sur le même type d'objet (des chaînes de caractères statiques). Vous verrez alors apparaître de nombreux `ugettext(...)` dans votre application, ce qui serait très redondant. Une habitude, presque devenue une convention, est d'imposer cette fonction avec l'alias _ (*underscore*), afin de rendre votre code plus lisible :

Code : Python

```
from django.utils.translation import ugettext as _
```

Dès lors, il ne nous reste plus qu'à appliquer cette méthode à nos chaînes :

Code : Python

```
def test_i18n(request):
    nb_chats = 1
    couleur = "blanc"
    chaine = _(u"Bonjour les zéros !")
    ip = _(u"Votre IP est %s") % request.META['REMOTE_ADDR']
    infos = _(u"… et selon mes informations, vous avez %s chats %s
!") % (nb_chats, couleur)

    return render(request, 'test_i18n.html', locals())
```

En testant ce code, rien ne change à l'affichage de la page. En effet, la fonction `gettext` n'a aucun effet pour le moment, puisque nous utilisons la langue par défaut à l'affichage. Avant d'aller plus loin, il convient de préciser que ce code n'est en réalité pas totalement correct. Dans un premier temps, les pluriels ne s'accordent pas en fonction de la valeur de `nb_chats`, et cela même dans la langue par défaut. Pour corriger cela, il faut utiliser une autre fonction, cousine de `gettext`, qui est `ungettext`, dans le même module. Cette fonction permet de fournir une chaîne pour le cas singulier, une pour le cas pluriel et enfin la variable déterminant le cas à afficher. Pour le cas de nos amis les félins, cela donne :

Code : Python

```
from django.utils.translation import ugettext as _
from django.utils.translation import ungettext

def test_i18n(request):
    nb_chats = 2
    couleur = "blanc"
    chaine = _(u"Bonjour les zéros !")
    ip = _(u"Votre IP est %s") % request.META['REMOTE_ADDR']
    infos = ungettext(u"… et selon mes informations, vous avez %s
chat %s !",
                      u"… et selon mes informations, vous avez %s
chats %ss !",
                      nb_chats) % (nb_chats, couleur)

    return render(request, 'test_i18n.html', locals())
```

Vous pouvez déjà tester, en changeant la variable `nb_chats` de 2 à 1, les « s » après « chat » et « blanc » disparaissent.

Cependant, nous avons encore un autre problème. Lorsque nous imaginons une traduction en anglais de cette chaîne, une des solutions pour le cas singulier serait : « … and according to my informations, you have %s %s cat », afin d'avoir « … and according to my informations, you have 1 white cat » par exemple. Cependant, en français, l'adjectif se situe après le nom contrairement à l'anglais où il se situe avant. Cela nécessite d'inverser l'ordre de nos variables à l'affichage : dans le cas présent nous allons plutôt obtenir « you have white 1 cat » !

Pour ce faire, il est possible de nommer les variables au sein de la chaîne de caractères, ce que nous vous recommandons de faire tout le temps, même si vous n'avez qu'une variable dans votre chaîne. Une troisième version de notre vue est donc :

Code : Python

```
def test_i18n(request):
    nb_chats = 2
    couleur = "blanc"  # Nous supposons que tous les chats vont
avoir la même couleur
    chaine = _(u"Bonjour les zéros !")
    ip = _(u"Votre IP est %(ip)s") % {'ip':
request.META['REMOTE_ADDR']}
    infos = ungettext(u"… et selon mes informations, vous avez
%(nb)s chat %(color)s !",
                      u"… et selon mes informations, vous avez
%(nb)s chats %(color)ss !",
                      nb_chats) % {'nb': nb_chats, 'color': couleur}

    return render(request, 'test_i18n.html', locals())
```

De cette façon, il sera possible d'inverser l'ordre de `%(nb)s` et `%(color)s`, permettant la traduction la plus naturelle possible.

Pour finir la thématique des vues, nous allons aider encore un peu plus les traducteurs. Il se peut que certaines chaînes soient difficiles à traduire hors de leur contexte. Par exemple, imaginons que vous ayez la chaîne suivante :

Code : Python

```
quota = _("3 livres")
```

Ici, si nous ne connaissons pas du tout le contexte, le mot « livre » pourrait correspondre à l'objet que nous lisons, mais aussi à l'unité de poids dans le système de mesures anglo-saxon. Dans ce cas, il est bon de préciser au traducteur ce que signifie réellement la chaîne. Pour ce faire, nous allons commenter notre code, en préfixant le commentaire par `Translators` :

Code : Python

```
# Translators: This message informs the user about how many books
he can borrow
quota = _("3 livres")
```

Ainsi, vous signalez aux traducteurs la signification de votre chaîne, pour éviter toute ambiguïté. Dans le même genre, il se peut qu'apparaisse deux fois la même chaîne, mais ne signifiant pas la même chose, typiquement les homographes ! Pour résoudre ce problème, il est possible d'ajouter un marqueur de contexte, permettant de différencier les deux chaînes, et ainsi générer deux traductions distinctes. Cela nécessite l'import de la fonction `pgettext` :

Code : Python

```
from django.utils.translation import pgettext

sujet =  ("tu")
verbe = pgettext("verbe", "as")
```

```
valeur = pgettext("carte de jeu", "as")
couleur = _(u"de trèfle")
carte = _("%(suj)s %(ver)s : %(val)s %(col)s") % {"suj": sujet,
"ver": verbe, "val": valeur, "col": couleur}
```

Cas des modèles

Pour nos modèles, la technique est en réalité la même, sauf qu'il faut utiliser la méthode `ugettext_lazy` au lieu de `gettext` pour la même raison que celle évoquée avec les vues génériques et la fonction `reverse_lazy` : nous souhaitons que la traduction soit effectuée à l'exécution et non à la déclaration des classes, lors de la validation des modèles par le serveur. La traduction sera alors effectuée seulement quand la chaîne sera affichée. L'utilisation des fonctions `pgettext` et `ungettext` est également similaire à ce mode de fonctionnement, au détail près qu'il faut les suffixer par `_lazy`.

De la même façon, nous allons appeler la fonction `ugettext_lazy` (que nous renommons en `_`, pour plus de simplicité comme nous l'avons fait plus haut) et l'appliquer à toutes nos chaînes de caractères, par exemple ici avec le fichier `models.py` de notre application `mini_url` :

Code : Python

```
#-*- coding: utf-8 -*-
from django.db import models
import random
import string
from django.utils.translation import ugettext_lazy as _


class MiniURL(models.Model):
  url = models.URLField(verbose_name=_(u"URL à réduire"),
unique=True)
  code = models.CharField(max_length=6, unique=True)
  date = models.DateTimeField(auto_now_add=True,
verbose_name=_(u"Date d'enregistrement"))
  pseudo = models.CharField(max_length=255, blank=True, null=True)
  nb_acces = models.IntegerField(default=0, verbose_name=_(u"Nombre
d'accès à l'URL"))

  def __unicode__(self):
      return u"[{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 = _(u"Mini URL")
      verbose_name_plural = _(u"Minis URLs")
```

Traduire les chaînes dans nos templates

La traduction dans les templates repose également sur l'outil gettext. Cependant, dans nos templates, les tags permettant l'internationalisation ont des noms différents des fonctions disponibles dans `django.utils.translation`.

Commençons d'abord par importer les tags nécessaires. Comme pour nos propres *templatetags*, les tags d'internationalisation sont disponibles après avoir effectué un `{% load i18n %}`, au début de chacun de vos templates. Dès que vous souhaitez internationaliser un template, pensez donc à ajouter tout en haut cette ligne.

Nous allons maintenant voir les deux tags, assez similaires, permettant de faire de la traduction dans nos templates, à savoir `{% trans %}` et `{% blocktrans %}`.

Le tag {% trans %}

Ce tag permet de traduire à la fois une chaîne de caractères constante, mais aussi une variable :

Code : Jinja

```
<h2>{% trans "Contactez-nous !" %}</h2>
<p>{% trans ma_variable %}</p>
```

En interne, ces blocs appellent la fonction `ugettext()`, introduite précédemment. Vous pouvez donc imaginer que le fonctionnement est identique à ce que nous avons vu. Dans le cas d'une variable (`{% trans ma_variable %}` dans le code précédent), la traduction sera recherchée à l'exécution de la page, en fonction du contenu même de la variable. Comme pour les chaînes de caractères, si le contenu de cette variable n'est pas présent dans les traductions, alors son contenu sera affiché tel quel.

Il est également possible de générer la traduction d'une chaîne sans l'afficher. Cela est utile si vous utilisez plusieurs fois la même chaîne dans un même template :

Code : Jinja

```
{% trans "Contactez-nous" as titre %}
<title>{{ titre }} - {% trans "Blog sur les crêpes bretonnes"
%}</title>
<meta name="description" content="{{ titre }}, {% trans "sur notre
magnifique blog de crêpes" %}">
```

Cela permet aussi de clarifier le code, en réduisant le poids total du template. Enfin, le tag `{% trans %}` supporte également le marqueur de contexte, afin de différencier les homonymes :

Code : Jinja

```
{% trans "Est" context "Verbe être" %}
{% trans "Est" context "Cardinalité" %}
```

Ici, deux traductions différentes pourront être déclarées, pour différencier les deux cas, l'un qui concerne la cardinalité et l'autre qui concerne le fameux verbe auxiliaire. Comme vous pouvez le voir, ce marqueur de contexte est très utile dans le cas des homographes.

Le tag {% blocktrans %}

Contrairement au tag simple `{% trans %}`, les blocs `{% blocktrans %} {% endblocktrans %}` permettent d'exécuter des schémas plus complexes de traduction. Par exemple, il est possible d'incorporer des variables au sein d'un bloc :

Code : Jinja

```
{% blocktrans %}Vous avez {{ age }} ans.{% endblocktrans %}
```

Cependant, pour accéder à des expressions (attributs d'un objet, utilisation de filtres…), il faut déclarer ces expressions comme variables locales du bloc de traduction, avec le mot-clé `with` :

Code : Jinja

```
{% blocktrans with nom=user.nom prenom=user.prenom
nb_articles=panier|length %}
Vous êtes {{ prenom }} {{ nom }}. Vous avez {{ nb_articles }}
```

```
articles dans votre panier.
{% endblocktrans %}
```

À l'image de `ungettext`, `{% blocktrans %}` permet également la gestion des pluriels. Pour ce faire :

- Il faut préciser quel est le nombre qui différencie singulier et pluriel via la syntaxe `count nombre=ma_variable` (où `nombre` et `ma_variable` sont des variables) ;
- Déclarer deux nouveaux sous-blocs, séparés par le tag `{% plural %}` : un pour le cas du singulier et un pour le cas du pluriel. Il est également possible de déclarer une variable pour préciser le nombre d'articles précis, comme dans l'exemple ci-dessous avec `nb`.

Un exemple d'utilisation est probablement plus parlant :

Code : Jinja

```
{% blocktrans count nb=articles|length %}
Vous avez 1 article dans votre panier.
{% plural %}
Vous avez {{ nb }} articles dans votre panier.
{% endblocktrans %}
```

Il est bien entendu possible de combiner ce tag avec le mot-clé `with`, ce qui donne une structure encore plus complète :

Code : Jinja

```
{% blocktrans with total=commande.total count
nb=commande.articles|length %}
Vous avez 1 article dans votre panier. Prix total : {{ total }}
{% plural %}
Vous avez {{ nb }} articles dans votre panier. Prix total : {{ total
}}
{% endblocktrans %}
```

Enfin, comme pour le bloc `{% trans %}`, le bloc `{% blocktrans %}` supporte la gestion du contexte :

Code : Jinja

```
{% blocktrans with pseudo=user.username context "lien de parenté" %}
Les fils de {{ pseudo }}
{% endblocktrans %}
```

Aidez les traducteurs en laissant des notes !

Nous avons globalement fait le tour des fonctionnalités de traduction dans les templates ! En effet, tous les comportements sont gérés automatiquement par les deux tags que nous venons de voir, en fonction des arguments fournis. Il nous reste juste à voir comment guider les traducteurs dans leur tâche.

Comme pour la partie concernant les vues et les modèles, il est possible de laisser des notes aux traducteurs, via des commentaires. Pour ce faire, il faut également commencer par `Translators:`, dans un tag `{# #}` ou `{% comment %}`, comme ce que nous avons vu précédemment :

Code : Jinja

```
{# Translators: Phrase courte, dans un bouton pour la recherche #}
{% trans "Go !" %}

{% comment %}Translators: Phrase affichée dans le header du site {%
```

```
endcomment %}
{% blocktrans %}Notre phrase d'accroche géniale !{% endblocktrans %}
```

Sortez vos dictionnaires, place à la traduction !

Nous avons désormais terminé tout ce qui concerne le développeur, nous allons sortir du code et des templates quelques instants pour nous attaquer au travail du traducteur. En effet, nous avons déterminé quelles chaînes doivent être traduites ; maintenant, il ne reste plus qu'à les traduire.

Génération des fichiers `.po`

Afin d'attaquer la traduction, il va nous falloir un support où nous pourrons faire la correspondance entre les chaînes dans notre code et leur traduction. Pour cela, gettext utilise des fichiers spécifiques, contenant les traductions, dont l'extension est `.po` (pour *Portable Object File*). Voici un extrait d'un des fichiers que nous avons générés :

Code : Autre

```
#: .\blog\views.py:77
msgid "Bonjour les zéros !"
msgstr ""

#. Translators: Message shown in the logbox, if disconnected
#: .\blog\views.py:79
#, python-format
msgid "Votre IP est %s"
msgstr ""
```

Nous pouvons constater que chaque chaîne traduisible est encapsulée dans un `msgid`, suivi d'un `msgstr`, qui contiendra la chaîne traduite. Voici ce à quoi ressemblera le fichier avec les traductions :

Code : Autre

```
#: .\blog\views.py:77
msgid "Bonjour les zéros !"
msgstr "Hello newbies !"

#. Translators: Message shown in the logbox, if disconnected
#: .\blog\views.py:79
#, python-format
msgid "Votre IP est %s"
msgstr "Your IP is %s"
```

Pour permettre la traduction de l'application en plusieurs langues, chaque langue aura son propre dossier avec ses fichiers `.po`. Nous allons tout d'abord générer ces fichiers.

Avant de commencer, il va nous falloir créer un dossier `locale`, qui contiendra l'ensemble des traductions pour toutes nos langues. Ce dossier peut se situer soit au sein d'un projet, si nous souhaitons traduire l'ensemble des applications en un seul endroit, soit dans le dossier d'une application afin de lier chaque application à ses propres traductions. Nous allons dans notre cas mettre toutes les traductions de toutes les applications au sein d'un même dossier, à la racine du projet.

Les traductions faites au sein des dossiers d'applications sont prioritaires sur les traductions disponibles dans le dossier du projet.

Il faut ensuite indiquer à Django où se situent les fichiers de traduction pour qu'il puisse les utiliser. Pour ce faire, indiquez la ligne suivante dans votre `settings.py` et adaptez-la au chemin vers le dossier `locale` que vous venez de créer :

Code : Python

```
LOCALE_PATHS = (
    '/home/mathx/crepes_bretonnes/locale/',
```

```
    )
```

Si vous avez décidé de mettre les traductions directement dans vos applications, vous devrez indiquer le dossier des traductions de chaque application dans cette variable.

La création de ces fichiers `.po` est automatisé par Django : grâce à `manage.py`, il est possible de générer les fichiers, via un appel à gettext. Pour créer un dossier dans `locale` contenant les traductions en anglais, utilisez la commande suivante, à la racine de votre projet ou dans une application :

Code : Console

```
python manage.py makemessages -l en
```

Après quelques secondes, un dossier `locale/en/LC_MESSAGES`, contenant comme unique fichier `django.po` apparaît. Ce fichier contient des métadonnées ainsi que l'ensemble des traductions, comme nous avons pu le montrer précédemment.

Si vous ne possédez pas gettext, les fichiers seront très probablement vides ! Veuillez vous référer au début de ce chapitre si cela se produit.

Si jamais vous mettez à jour votre projet et modifiez les chaînes à traduire via le processus i18n, vous devrez mettre à jour vos fichiers `.po` en utilisant la commande :

Code : Console

```
python manage.py makemessages --all
```

Ainsi, les traductions déjà effectuées resteront intactes, et celles qui doivent être modifiées ou ajoutées seront insérées dans le fichier déjà existant.

Une fois ce fichier généré, le travail de traduction commence : chaque chaîne à traduire est représentée dans le fichier par une ligne `msgid`. Chaque `msgid` est éventuellement précédé d'informations afin d'aider le traducteur (fichier et ligne à laquelle se situe la chaîne), mais aussi les commentaires éventuels que vous avez laissés.

Les lignes qui suivent le `msgid` correspondent à la traduction dans la langue du dossier (ici l'anglais pour le code `en`). Dans le cas des traductions courtes, il y a juste un `msgstr` à renseigner. Autrement, deux cas se présentent.

Dans le cas de messages plus longs, `msgid` commence par une chaîne vide `""`, et continue à la ligne. Il est important de laisser pour gettext cette chaîne vide. Vous devez agir de la même façon pour la traduction : les chaînes seront automatiquement concaténées (attention aux espaces entre les mots après concaténation !). Ce cas se présente lorsque la longueur totale de la ligne dépasse les 80 caractères. Voici un exemple :

Code : Autre

```
#: .\templates\blog\accueil.html.py:4
msgid ""
"Ici, nous parlons de tout et de rien, mais surtout de rien. Mais alors, vous "
"allez me demandez quel est le but de ce paragraphe ? Je vais vous répondre "
"simplement : il n'en a aucun."
msgstr ""
"Here, we talk about everything and anything, but mostly nothing. So, you "
"will ask me what is the purpose of this paragraph... I will answer "
"simply : none, it's useless!"
```

Le second cas pouvant se présenter est celui des traductions avec la gestion du pluriel. Si vous regardez notre exemple avec nos fameux chats blancs, le rendu dans le fichier `.po` est le suivant :

Code : Autre

```
#: .\blog\views.py:80
#, python-format
msgid "… et selon mes informations, vous avez %(nb)s chat %(color)s !"
msgid_plural ""
"… et selon mes informations, vous avez %(nb)s chats %(color)ss !"
msgstr[0] ""
msgstr[1] ""
```

L'entrée `msgstr[0]` correspond alors au message traduit au singulier, et le second, au pluriel. Remarquez que, ici encore, `msgid_plural` est passé à la ligne, puisque la chaîne est légèrement trop longue (81 caractères sur les 80 maximum !).

Nous en avons déjà terminé avec les fichiers `.po`, vous savez désormais où répertorier vos traductions.

Génération des fichiers .mo

Une fois les fichiers `.po` complétés, il faut les compiler dans un format que gettext peut comprendre. En effet, afin d'optimiser ses temps de traitement, gettext souhaite obtenir les fichiers dans un nouveau format, binaire cette fois-ci, les fichiers `.mo` (pour *Machine Object File*). Pour ce faire, il suffit simplement d'entrer la commande suivante, au même endroit où vous avez effectué `makemessages` :

Code : Console

```
python manage.py compilemessages
```

Une fois cela effectué, un fichier `.mo` est censé apparaître à côté de chacun de vos fichiers `.po`. Django ira chercher automatiquement ces fichiers et en extraira le contenu, avec vos traductions !

Le changement de langue

Nous avons maintenant une application pouvant gérer plusieurs langues, mais l'utilisateur ne peut pas forcément choisir celle qu'il souhaite utiliser. Pour ce faire, il existe trois méthodes permettant de choisir la langue à afficher.

Une première méthode consiste à utiliser la fonction `django.utils.translation.activate`. Cette fonction s'utilise surtout en dehors des vues et des templates et est plutôt destinée à la gestion de la langue dans des crons par exemple. Elle permet d'assigner la langue à utiliser pour le thread actuel :

Code : Python

```
>>> from django.utils import translation
>>> from django.utils.translation import ugettext as _
>>> translation.activate('en')
>>> print _(u"Bonjour les zéros !")
Hello newbies!
```

Une deuxième méthode consisterait à modifier la variable `request.session['django_language']` dans une vue. Pour indiquer l'anglais comme langue à afficher, il suffit donc simplement d'indiquer `request.session['django_language'] = 'en'`. Néanmoins, cette méthode interfère dans le processus de détermination de la langue que Django effectue et il est donc déconseillé d'utiliser cette méthode.

Ce qui nous amène à la troisième méthode qui utilise une vue générique. Il vous faut tout d'abord ajouter cette directive de routage dans votre `urls.py` principal :

Code : Python

```
(r'^i18n/', include('django.conf.urls.i18n'))
```

Cette vue est censée recevoir des arguments provenant d'un formulaire. Pour ce faire, nous avons établi le template suivant :

Code : HTML

```
{% load i18n %}
<h1>Changer de langue</h1>

{% trans "Bonjour les zéros !" %}

<form action="/i18n/setlang/" method="post">
    {% csrf_token %}
    <input name="next" type="hidden" value="/une/url/" />

    <select name="language">
    {% for lang in LANGUAGES %}
        <option value="{{ lang.0 }}">{{ lang.1 }}</option>
    {% endfor %}
    </select>

    <input type="submit"/>
</form>
```

Rendez ce template accessible depuis une vue classique ou une vue générique et n'oubliez pas de modifier le paramètre `next` du formulaire qui indique vers quelle URL l'utilisateur doit être redirigé après avoir validé le formulaire. Si vous n'indiquez pas ce paramètre, l'utilisateur sera redirigé d'office vers `/`.

La liste des langues sera affichée dans le formulaire et le choix envoyé à Django après la soumission du formulaire. Vous verrez dès lors la phrase « Bonjour les zéros ! » traduite dans la langue que vous avez choisie.

En résumé

- Le processus de traduction se divise en deux parties :
  - L'internationalisation, où nous indiquons ce qui est à traduire ;
  - La localisation, où nous effectuons la traduction et l'adaptation à la culture.
- Ce processus se base essentiellement sur l'utilisation de l'outil gettext, permettant la génération de fichiers de traduction utilisables par des novices en développement.
- Django permet également d'adapter l'affichage des dates et des nombres à la langue, en même temps que leur traduction.
- Grâce aux sessions et aux middlewares, le framework peut deviner la langue de l'utilisateur automatiquement, en fonction de son navigateur ou de ses précédentes visites.

Les tests unitaires

Un test unitaire est une opération qui vérifie une certaine partie de votre code. Cela vous permet de vérifier certains comportements de fonctions : « est-ce que si nous appelons cette fonction avec ces arguments, nous obtenons bien ce résultat précis ? », ou de vérifier toute suite d'appels de vues de votre application en leur fournissant des données spécifiques. Ces tests pourront être exécutés automatiquement les uns à la suite des autres, chaque fois que vous le désirerez.

Dans ce chapitre, nous allons donc apprendre à tester des fonctions ou méthodes spécifiques et des vues entières.

Nos premiers tests

Pourquoi faire des tests unitaires ?

Ces tests ont plusieurs avantages. Tout d'abord, à chaque fois que vous modifierez ou ajouterez quelque chose à votre code, il vous suffira de lancer tous les tests afin de vous assurer que vous n'avez introduit aucune erreur lors de votre développement. Bien sûr, vous pourriez faire cela manuellement, mais lorsque que vous avez des dizaines de cas de figure à tester, la perte de temps devient considérable. Bien entendu, écrire des tests pour chaque cas prend un peu de temps, mais plus le développement s'intensifie, plus cela devient rentable. Au final, il y a un gain de temps et de productivité.

De plus, lorsqu'un bug s'introduit dans votre code, si vos tests sont bien conçus, ils vous indiqueront exactement où il se trouve, ce qui vous permettra de le résoudre encore plus rapidement.

Finalement, les tests sont considérés comme une preuve de sérieux. Si vous souhaitez que votre projet devienne un logiciel libre, sachez que de nombreux éventuels contributeurs risquent d'être rebutés si votre projet ne dispose pas de tests.

Cela étant dit, attaquons-nous au sujet.

Reprenons notre modèle `Article` introduit précédemment dans l'application « blog ». Nous y avons adjoint une méthode `est_recent` qui renvoie `True` si la date de parution de l'article est comprise dans les 30 derniers jours, sinon elle renvoie `False` :

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 est_recent(self):
        "Retourne True si l'article a été publié dans les 30
derniers jours"
        from datetime import datetime
        return (datetime.now()-self.date).days < 30

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

Une erreur relativement discrète s'y est glissée : que se passe-t-il si la date de parution de l'article se situe dans le futur ? L'article ne peut pas être considéré comme récent, car il n'est pas encore sorti ! Pour détecter une telle anomalie, il nous faut écrire un test.

Les tests sont répartis par application. Chaque application possède en effet par défaut un fichier nommé `tests.py` dans lequel vous devez insérer vos tests. N'oubliez pas d'inclure `#-*- coding: utf-8 -*-` au début du fichier si vous comptez utiliser des accents dans celui-ci !

Voici notre `tests.py`, incluant le test pour vérifier si un article du futur est récent ou non, comme nous l'avons expliqué ci-dessus :

Code : Python

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

```
        This file demonstrates writing tests using the unittest module.
        These will pass
        when you run "manage.py test".

        Replace this with more appropriate tests for your application.
        """

        from django.test import TestCase

        from models import Article
        from datetime import datetime, timedelta


        class ArticleTests(TestCase):
            def test_est_recent_avec_futur_article(self):
                """
        vérifie si la méthode est_recent d'un Article ne
        renvoie pas True si l'Article a sa date de publication
        dans le futur.
        """

                futur_article = Article(date=datetime.now() +
        timedelta(days=20))
                #Il n'y a pas besoin de remplir tous les champs ou de
        sauvegarder l'entrée
                self.assertEqual(futur_article.est_recent(), False)
```

Les tests d'une même catégorie (vérifiant par exemple toutes les méthodes d'un même modèle) sont regroupés dans une même classe, héritant de `django.test.TestCase`. Nous avons ici notre classe `ArticleTests` censée regrouper tous les tests concernant le modèle `Article`.

Dans cette classe, chaque méthode dont le nom commence par `test_` représente un test. Nous avons ajouté une méthode `test_est_recent_avec_futur_article` qui vérifie si un article publié dans le futur est considéré comme récent ou non. Cette fonction crée un article censé être publié dans 20 jours et vérifie si sa méthode `est_recent` renvoie `True` ou `False` (pour rappel, elle devrait renvoyer `False`, mais renvoie pour le moment `True`).

La vérification même se fait grâce à une méthode de `TestCase` nommée `assertEqual`. Cette méthode prend deux paramètres et vérifie s'ils sont identiques. Si ce n'est pas le cas, le test est reporté à Django comme ayant échoué, sinon, rien ne se passe et le test est considéré comme ayant réussi.

Ici, la méthode `est_recent` doit bien renvoyer `False`. Comme nous avons introduit une erreur dans notre modèle, elle est censée renvoyer `True` pour le moment, et donc faire échouer le test.

Afin d'exécuter nos tests, il faut utiliser la commande `python manage.py test` qui lance tous les tests répertoriés :

Code : Console

```
python manage.py test
Creating test database for alias 'default'...
.....................................................
======================================================================
FAIL: test_est_recent_avec_futur_article (blog.tests.ArticleTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/mathx/crepes_bretonnes/blog/tests.py", line 31, in test_est_recent_av
    self.assertEqual(futur_article.est_recent(), False)
AssertionError: True != False

----------------------------------------------------------------------
Ran 419 tests in 10.256s

FAILED (failures=1, skipped=1)
Destroying test database for alias 'default'...
```

Comme prévu, le test que nous venons de créer a échoué.

419 tests ont été lancés. C'est normal, car les applications par défaut de Django (utilisateurs, sessions, etc.) possèdent elles aussi des tests qui ont également été exécutés. Si vous souhaitez juste exécuter les tests de l'application « blog », vous pouvez indiquer `python manage.py blog`, ou également `manage.py test blog.ArticleTests` pour lancer tous les tests de la classe `ArticleTests`, ou directement spécifier un seul test comme ceci : `python manage.py test blog.ArticleTests.est_est_recent_avec_futur_article`.

Modifions donc la méthode `est_recent` afin de corriger le bug :

Code : Python

```
    def est_recent(self):
            "Retourne True si l'article a été publié dans les 30
derniers jours"
            from datetime import datetime
            return (datetime.now()-self.date).days < 30 and self.date <
datetime.now()
```

Relançons le test. Comme prévu, celui-ci fonctionne désormais !

Code : Console

```
python manage.py test
Creating test database for alias 'default'...
.........................................
----------------------------------------------------------------------
Ran 419 tests in 10.165s

OK (skipped=1)
Destroying test database for alias 'default'...
```

Petite précision : vous pouvez préparer votre suite de tests en créant une méthode nommée `setUp` qui permet d'initialiser certaines variables à l'intérieur de votre classe, pour les utiliser dans vos tests par la suite :

Code : Python

```
class UnTest(TestCase):
    def setUp(self):
        self.une_variable = "Salut !"

    def test_verification(self):
        self.assertEqual(self.une_variable, "Salut !")
```

Testons des vues

Outre les tests sur des méthodes, fonctions, etc., qui sont spécifiques, il est également possible de tester des vues. Cela se fait grâce à un serveur web intégré au système de test qui sera lancé tout seul lors de la vérification des tests.

Pour tester quelques vues, nous allons utiliser l'application `mini_url` créée lors du TP de la deuxième partie. N'hésitez pas à reprendre le code que nous avons donné en solution si ce n'est pas déjà fait, nous nous baserons sur celui-ci afin de construire nos tests.

Voici le début de notre `mini_url/tests.py`, incluant notre premier test :

Code : Python

```
#-*- coding: utf-8 -*-
from django.test import TestCase
from django.core.urlresolvers import reverse
from models import MiniURL
from views import generer

def creer_url():
```

```
        mini = MiniURL(url="http://foo.bar",code=generer(6),
    pseudo=u"Foo foo !")
        mini.save()
        return mini

    class MiniURLTests(TestCase):
        def test_liste(self):
            """
    Vérifie si une URL sauvegardée est bien affichée
    """
            mini = creer_url()
            reponse = self.client.get(reverse('mini_url.views.liste'))
            self.assertEqual(reponse.status_code, 200)
            self.assertContains(reponse, mini.url)
            self.assertQuerysetEqual(reponse.context['minis'],
    [repr(mini)])
```

Nous avons créé une petite fonction nommée `creer_url` qui crée une redirection, l'enregistre et la retourne, afin de ne pas devoir dupliquer le code dans nos futurs tests.

Django créera à chaque séance de tests une nouvelle base de données vide et utilisera celle-ci, au lieu d'utiliser la base de données classique, afin d'éviter de compromettre cette dernière.

Intéressons-nous ensuite au test `test_liste` qui va s'assurer que lorsque nous créons une redirection dans la base de données celle-ci est bien affichée par la vue `views.liste`. Pour ce faire, nous créons tout d'abord une redirection grâce à la fonction `creer_url` et nous demandons ensuite au client intégré au système de test d'accéder à la vue `liste` grâce à la méthode `get` de `self.client`. Cette méthode prend une URL, c'est pourquoi nous utilisons la fonction `reverse` afin d'obtenir l'URL de la vue spécifiée.

`get` retourne un objet dont les principaux attributs sont `status_code`, un entier représentant le code HTTP de la réponse, `content`, une chaîne de caractères contenant le contenu de la réponse, et `context`, le dictionnaire de variables passé au template si un dictionnaire a été utilisé.

Nous pouvons donc vérifier si notre vue s'est bien exécutée en vérifiant le code HTTP de la réponse : `self.assertEqual(reponse.status_code, 200)`.

Pour rappel, 200 correspond à une requête correctement déroulée, 302 à une redirection, 404 à une page non trouvée et 500 à une erreur côté serveur.

Deuxième vérification : est-ce que l'URL qui vient d'être créée est bien affichée sur la page ? Cela se fait grâce à la méthode `assertContains` qui prend comme premier argument une réponse comme celle donnée par `get`, et en deuxième argument une chaîne de caractères. La méthode renvoie une erreur si la chaîne de caractères n'est pas contenue dans la page.

Dernière et troisième vérification : est-ce que le `QuerySet` `minis` contenant toutes les redirections dans notre vue (celui que nous avons passé à notre template et qui est accessible depuis `reponse.context`) est égal au `QuerySet` indiqué en deuxième paramètre ? En réalité, le deuxième argument n'est pas un `QuerySet`, mais est censé correspondre à la représentation du premier argument grâce à la fonction `repr`. Autrement dit, il faut que `repr(premier_argument) == deuxieme_argument`. Voici ce à quoi ressemble le deuxième argument dans notre exemple : `['<MiniURL: [ALSWM0] http://foo.bar>']`.

Dans ce test, nous n'avons demandé qu'une simple page. Mais comment faire si nous souhaitons par exemple soumettre un formulaire ? Une telle opération se fait grâce à la méthode `post` de `self.client`, dont voici un exemple à partir de la vue `nouveau` de notre application, qui permet d'ajouter une redirection :

Code : Python

```
    def test_nouveau_redirection(self):
            """
    Vérifie la redirection d'un ajout d'URL
    """
            data = {
                    'url':'http://www.siteduzero.com',
                    'pseudo':u'Un zéro',
            }
            reponse =
    self.client.post(reverse('mini_url.views.nouveau'), data)
            self.assertEqual(reponse.status_code, 302)  # Le retour doit
```

```
être une redirection
        self.assertRedirects(reponse,
reverse('mini_url.views.liste'))
```

La méthode `post` fonctionne comme `get`, si ce n'est qu'elle prend un deuxième argument, à savoir un dictionnaire contenant les informations du formulaire. De plus, nous avons utilisé ici une nouvelle méthode de vérification : `assertRedirects`, qui vérifie que la réponse est bien une redirection vers l'URL passée en paramètre. Autrement dit, si la requête s'effectue correctement, la vue `nouveau` doit rediriger l'utilisateur vers la vue `liste`.

Sachez que si vous gérez des redirections, vous pouvez forcer Django à suivre la redirection directement en indiquant `follow=True` à `get` ou `post`, ce qui fait que la réponse ne contiendra pas la redirection en elle-même, mais la page ciblée par la redirection, comme le montre l'exemple suivant.

Code : Python

```
def test_nouveau_ajout(self):
        """
Vérifie si après la redirection l'URL ajoutée est bien dans la
liste
"""
        data = {
                'url':'http://www.crepes-bretonnes.com',
                'pseudo':u'Amateur de crêpes',
        }
        reponse =
self.client.post(reverse('mini_url.views.nouveau'), data,
follow=True)
        self.assertEqual(reponse.status_code, 200)
        self.assertContains(reponse, data['url'])
```

Dernier cas de figure à aborder : imaginons que vous souhaitiez tester une vue pour laquelle il faut obligatoirement être connecté à partir d'un compte utilisateur, sachez que vous pouvez vous connecter et vous déconnecter de la façon suivante :

Code : Python

```
c = Client()
c.login(username='utilisateur', password='mot_de_passe')
reponse = c.get('/une/url/')
c.logout()  # La déconnexion n'est pas obligatoire
```

En résumé

- Les tests unitaires permettent de s'assurer que vous n'avez introduit aucune erreur lors de votre développement, et assurent la robustesse de votre application au fil du temps.
- Les tests sont présentés comme une suite de fonctions à exécuter, testant plusieurs assertions. En cas d'échec d'une seule assertion, il est nécessaire de vérifier son code (ou le code du test), qui renvoie un comportement anormal.
- Les tests sont réalisés sur une base de données différente de celles de développement ; il n'y a donc aucun souci de corruption de données lors de leur lancement.
- Il est possible de tester le bon fonctionnement des modèles, mais aussi des vues. Ainsi, nous pouvons vérifier si une vue déclenche bien une redirection, une erreur, ou si l'enregistrement d'un objet a bien lieu.

Ouverture vers de nouveaux horizons : django.contrib

Une des forces de Python est sa philosophie « *batteries included* » (littéralement « piles incluses ».) : la bibliothèque standard couvre un nombre d'utilisations très large, et permet d'intégrer énormément de fonctionnalités, sans qu'on doive les coder entièrement une à une. Django suit également cette philosophie, en incluant dans le framework un nombre de modules complémentaires très important.

Nous arrivons à la fin de ce cours, nous vous proposons donc de parcourir rapidement la liste de ces modules, afin de découvrir d'éventuels modules qui pourraient vous intéresser.

Vers l'infini et au-delà

L'ensemble des fonctionnalités supplémentaires de Django est situé dans le module `django.contrib`. Chaque sous-module est un module à part permettant d'intégrer une nouvelle fonctionnalité. Ces fonctionnalités sont le plus souvent *indépendantes*, malgré quelques exceptions.

L'unique caractéristique commune de toutes ces fonctionnalités est qu'elles ne sont pas essentielles au bon déroulement global de Django. Nous avons déjà présenté quelques-unes des fonctionnalités présentes dans `django.contrib`. En effet, l'administration ou encore le système utilisateurs font partie de `django.contrib`.

Nous avons adjoint ici une liste reprenant tous les modules compris dans `django.contrib`. Nous ne pouvons que vous conseiller de la regarder. Vous découvrirez peut-être de nouveaux modules qui pourraient vous être utiles, au lieu de devoir les réécrire vous-mêmes !

Nom du module	Description	Traité dans ce cours
`admin`	Système d'administration	Partie II, chapitre 4
`admindocs`	Auto-documentation de l'administration	Non traité
`auth`	Système utilisateurs	Partie IV, chapitre 1
`comments`	Application de gestion de commentaires. Très utile dans le cas d'un blog, d'un système de news…	Non traité
`contenttypes`	Permet d'obtenir la représentation de n'importe quel modèle	Partie III, chapitre 2
`flatpages`	Permet de gérer des pages HTML statiques au sein même de la base de données	Partie IV, chapitre 7
`formtools`	Ensemble d'outils pour compléter les formulaires (par exemple la prévisualisation avant envoi)	Non traité
`gis`	Bibliothèque complète qui apporte le support de SIG (systèmes d'information géographique), pour stocker des données géographiques et les exploiter. Voir http://www.geodjango.org pour plus d'informations	Non traité
`humanize`	Ensemble de filtres pour les templates, afin de rendre certaines données plus « naturelles », notamment pour les nombres et les dates	Partie IV, chapitre 7
`messages`	Gestion de notifications qui peuvent être affichées au visiteur	Partie IV, chapitre 2
`redirects`	Gestion des redirections au sein du projet via une base de données. Utile si vous changez le schéma de vos URL	Non traité
`sessions`	Gestion des sessions	Non traité
`sitemaps`	Génération de *sitemaps* XML	Non traité
`sites`	Permet la gestion de différents sites web avec la même base de données et la même installation de Django	Partie IV, chapitre 7
`staticfiles`	Gestion de fichiers statiques dans les templates	Partie II, chapitre 2
`syndication`	Génération de flux RSS et Atom	Non traité
`webdesign`	Intègre un outil de génération de *Lorem ipsum* via un unique tag : `{% lorem [nb] [methode] [random] %}`	La description dans ce tableau présente presque tout le module !

Nous allons clore cette partie par la présentation dans ce chapitre de deux fonctionnalités que nous avons jugées utiles et rapides à présenter : `flatpages` et `humanize`. Tous les autres modules sont très bien documentés dans la documentation officielle de Django.

Dynamisons nos pages statiques avec flatpages !

Sur la quasi-totalité des sites web, il existe des pages statiques dont le contenu doit parfois être modifié. Nous pouvons citer comme exemples les pages avec des informations de contact, de conditions générales d'utilisation, des foires aux questions, etc. Il peut être lourd de passer par des `TemplateView` pour ce genre de cas simple, puisque cela implique de devoir retourner dans le code ou le template à chaque modification du contenu.

Django propose le module `flatpage` pour contourner ce problème. Une `flatpage` est un objet caractérisé par une URL, un titre et un contenu. Tout cela sera inclus dans un template générique, ou bien dans un template que vous aurez adapté. Ces informations sont enregistrées dans la base de données et sont éditables à tout moment par l'administration.

Installation du module

Pour utiliser le module `flatpages`, il faut l'activer grâce à ces quelques étapes :

Dans votre fichier `settings.py` :

- Ajoutez les lignes `'django.contrib.sites'` et `'django.contrib.flatpages'` dans la liste `INSTALLED_APPS`, si elles ne sont pas déjà présentes.
- Vérifiez que votre `settings.py` contient bien une variable `SITE_ID`. Nous n'en avons encore jamais parlé dans ce cours, mais Django permet d'héberger plusieurs sites sur une même base de données via cette variable. Habituellement, cette valeur est mise à 1, mais si vous développez plusieurs sites, cette valeur peut changer !
- Lancez maintenant la commande `python manage.py syncdb`, pour créer les deux tables nécessaires.

Pour la suite, deux méthodes s'offrent à vous : vous pouvez soit spécifier clairement comment accéder à ces pages, soit activer un middleware, qui en cas d'erreur 404 (page non trouvée) vérifie si une `flatpage` correspond à l'URL saisie par l'utilisateur.

Le cas des URL explicites

Pour cette méthode, deux possibilités sont envisageables. Vous devrez placer le code suivant dans le fichier `urls.py` principal du projet.

Précisez un chemin précis vers les `flatpages` : dans cet exemple, toutes les URL de nos pages statiques commenceront par `/pages/` :

Code : Python

```
urlpatterns = patterns('',
        ('^pages/', include('django.contrib.flatpages.urls')),
    )
```

… ou créez un pattern qui attrape toutes les URL qui n'ont pas de vues associées et tentez de trouver une `flatpage` qui corresponde (ce qui est équivalent au middleware). Dans ce cas, le pattern doit être ajouté en toute fin du fichier `urls.py` !

Code : Python

```
urlpatterns += patterns('django.contrib.flatpages.views',
        (r'^(?P<url>.*)$', 'flatpage'),
    )
```

Utilisation du middleware `FlatpageFallbackMiddleware`

La seconde méthode laisse `FlatpageFallbackMiddleware` tout gérer. Une fois activé et dès qu'une erreur 404 est levée, le middleware vérifie dans la base de données si une page correspond à l'URL qui est demandée. Dans le cas où plusieurs sites sont activés, il vérifie également que la page demandée correspond bien au site web actuel. Si une page est trouvée, alors il l'affiche, sinon il laisse l'erreur 404 se poursuivre.

Lors de l'utilisation du middleware, le comportement des autres middlewares peut ne pas être pris en compte, notamment ceux

utilisant la méthode `process_view()` : le middleware est exécuté après la résolution de l'URL et après que l'erreur 404 s'est propagée, ce qui empêche ce type de middleware de s'exécuter. De plus, pour des raisons de fonctionnement, nous plaçons usuellement le middleware `FlatpageFallback` en dernier. Veillez donc à vérifier le comportement de vos middlewares avant d'utiliser cette méthode.

Enfin, vérifiez que vos middlewares laissent bien l'erreur 404 arriver jusqu'au middleware de `flatpage`. Si un autre middleware traite l'erreur et renvoie une exception, la réponse HTTP obtient le code 500 et notre nouveau middleware ne tentera même pas de chercher si une page existe.

L'installation du middleware semble plus simple au premier abord, mais nécessite de faire attention à de nombreux petits pièges. Si vous souhaitez l'utiliser, vous pouvez le faire en ajoutant `'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'` à votre `MIDDLEWARE_CLASSES`.

Quelle que soit la méthode choisie, la suite de ce cours ne change pas.

Gestion et affichage des pages

Maintenant que le module est installé et les tables créées, une nouvelle catégorie est apparue dans votre panneau d'administration : il s'agit du module `flatpages`, qui contient comme seul lien la gestion des pages statiques, comme le montre la figure suivante.

Le module « flatpages » dans l'administration de Django

Il n'est pas nécessaire de disserter plus longtemps sur cette partie : l'administration des pages statiques se fait comme tout autre objet, est assez intuitive et est plutôt bien documentée en cas de souci.

Pour la suite, nous avons créé une page de contact, avec comme URL `/contact/`, un titre et un peu de contenu. Vous pouvez vous aussi créer vos propres pages, pour prendre en main l'administration de ce module.

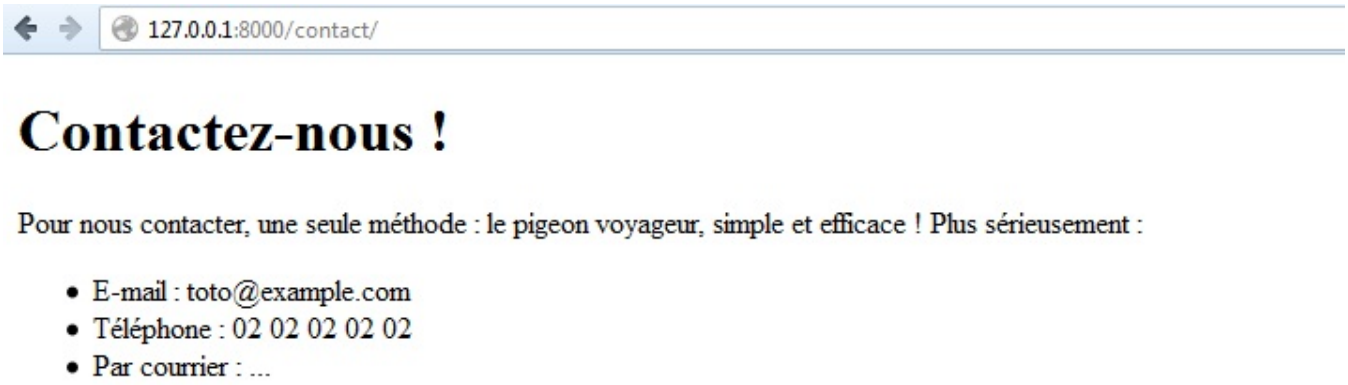
Il nous reste un seul point à traiter avant de pouvoir utiliser nos `flatpages` : *les templates*. Pour le moment, nous n'avons assigné aucun template à ces pages. Par défaut, chaque page sera traitée avec le template `flatpages/default.html` de votre projet. Nous allons donc tout d'abord le créer :

Code : HTML

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>{{ flatpage.title }}</title>
</head>
<body>
   <h1>{{ flatpage.title }}</h1>
   {{ flatpage.content }}
</body>
</html>
```

Une fois ce template enregistré, nous pouvons tester le module : vous pouvez à présent accéder à la page (voir figure suivante).



Exemple de page de contact utilisant « Flatpages »

Comme vous pouvez le voir, le champ `contenu` accepte également du HTML, pour mettre en forme votre page.

Enfin, plusieurs options avancées, que nous ne détaillerons pas ici, existent afin :

- D'autoriser les commentaires sur une page (via le module `django.contrib.comments`) ;
- De n'autoriser que les utilisateurs connectés à voir la page (via le module `django.contrib.auth`) ;
- D'utiliser un template différent de `flatpages/default.html`.

Lister les pages statiques disponibles

Pour terminer ce sous-chapitre, nous allons voir comment lister toutes les pages statiques disponibles via `flatpages`. Le module fournit plusieurs tags afin de récupérer ces listes. Vous devez donc tout d'abord charger la bibliothèque, via `{% load flatpages %}`. Après, vous pouvez lister l'ensemble des pages visibles par tous, comme ceci :

Code : Jinja

```
{% load flatpages %}
{% get_flatpages as flatpages %}
<ul>
{% for page in flatpages %}
<li><a href="{{ page.url }}">{{ page.title }}</a></li>
{% endfor %}
</ul>
```

Pour afficher celles également disponibles uniquement pour les personnes connectées, il faut spécifier l'utilisateur en cours, grâce au mot-clé `for` :

Code : Jinja

```
{% get_flatpages for user as flatpages %}
```

Si la variable `user` correspond à un utilisateur non connecté, ce tag aura le même effet que le premier, c'est-à-dire `{% get_flatpages as flatpages %}`. Enfin, il est possible de lister les `flatpages` commençant par une certaine URL uniquement, via un argument optionnel, avant les mots-clés **`for`** et **`as`** :

Code : Jinja

```
{% get_flatpages '/contact/' as contact_pages %}
{% get_flatpages prefixe_contact as contact_pages %}
{% get_flatpages '/contact/' for request.user as contact_pages %}
```

Rendons nos données plus lisibles avec humanize

Le module `django.contrib.humanize` fournit un ensemble de filtres permettant d'ajouter, selon les développeurs du framework, « une touche humaine aux données ». Nous allons voir les différents cas où les filtres du module `humanize` rendent la lecture d'une donnée plus agréable à l'utilisateur. L'avantage de ce module est qu'il prend mieux en charge la localisation : la transformation des données s'adapte à la langue de votre projet !

Avant de commencer, ajoutez la ligne `'django.contrib.humanize'` à votre variable `INSTALLED_APPS` dans le fichier `settings.py`. Pour intégrer les filtres présentés par la suite, il faut charger les `templatetags` du module, via la commande `{% load humanize %}`.

apnumber

Pour les nombres de 1 à 9, ce filtre va les traduire automatiquement en toutes lettres. Dans les autres cas (nombres supérieurs ou égaux à 10), ils seront affichés en chiffres. Cette convention suit le style de l'agence Associated Press.

Exemples (avec la langue du projet en français) :

Code : Jinja

```
{{ 1|apnumber }} renvoie "un" <br />
{{ "2"|apnumber }} renvoie "deux" <br />
{{ 10|apnumber }} renvoie 10.
```

Le filtre prend à la fois des entiers et des chaînes de caractères en paramètre.

intcomma

Ajoute des séparateurs de milliers, afin de simplifier la lecture. En réalité, le filtre prend en charge la localisation spécifiée dans le fichier `settings.py`, et le séparateur dépend donc de la langue courante : le séparateur sera la virgule si la langue courante est l'anglais par exemple.

Exemples (encore une fois en français puisque le séparateur est l'espace) :

Code : Jinja

```
{{ 300|intcomma }} renvoie 300 <br />
{{ "9000"|intcomma }} renvoie 9 000 <br />
{{ 90000|intcomma }} renvoie 90 000 <br />
{{ 9000000|intcomma }} renvoie 9 000 000 <br />
```

Le filtre prend à la fois des entiers et des chaînes de caractères en paramètre.

intword

Ce filtre permet de convertir les grands entiers en leur représentation textuelle, de préférence avec des nombres supérieurs à un million. Ce filtre respecte également la localisation pour le séparateur décimal.

Code : Jinja

```
{# Quelques cas classiques #}
{{ 1000000|intword }} renvoie 1,0 million. <br />
{{ "4525640"|intword }} renvoie 4,5 millions. <br />
{{ 20000000|intword }} renvoie 20,0 millions. <br />
{{ 999999999|intword }} renvoie 1000,0 millions. <br />
{{ 5500000000|intword }} renvoie 5,5 milliards. <br />

{# Et des cas plus extrêmes. Oui, il y a bien 101 zéros sur le
dernier exemple ;) #}
{{ 1000000000000000000|intword }} renvoie 1,0 quintillion. <br />
{{ 9000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000|intword }} renvoie
90,0 gogols.<br />


{# Ce filtre ne supporte pas les « petits nombres » #}
{{ 90000|intword }} renvoie 90000.
```

Le filtre prend à la fois des entiers et des chaînes de caractères en paramètre.

naturalday

Retourne « aujourd'hui », « hier » ou « demain » si la date est appropriée. Dans les autres cas, la date sera affichée selon le format fourni en paramètre.

Par exemple, si la date actuelle est le 4 mars 2130 :

Portion de notre vue :

Code : Python

```
date_avant_hier = datetime(2130, 3, 2)
date_hier = datetime(2130, 3, 3)
date_auj = datetime(2130, 3, 4)
date_demain = datetime(2130, 3, 5)
```

Template associé :

Code : Jinja

```
{{ date_avant_hier|naturalday:"DATE_FORMAT" }} renvoie "2 mars
2130"<br />
{{ date_avant_hier|naturalday:"d/m/Y" }} renvoie "02/03/2130"<br />
{{ date_hier|naturalday:"d/m/Y" }} renvoie "hier"<br />
{{ date_auj|naturalday:"d/m/Y" }} renvoie "aujourd'hui"<br />
{{ date_demain|naturalday:"d/m/Y" }} renvoie "demain"<br />
```

naturaltime

Retourne une chaîne de caractères précisant le nombre de secondes, minutes ou heures écoulées depuis la date (ou restantes dans le cas d'une date future) :

Exemple, en admettant que nous sommes le 4 mars 2130, à 14:20:00 :

Portion de notre vue :

Code : Python

```
date1 = datetime(2130, 3, 4, 14, 20, 0)
date2 = datetime(2130, 3, 4, 14, 19, 30)
date3 = datetime(2130, 3, 4, 13, 15, 25)
date4 = datetime(2130, 3, 4, 12, 20, 0)
date5 = datetime(2130, 3, 3, 13, 10, 0)
date6 = datetime(2130, 3, 5, 18, 20, 0)
```

Code : Jinja

```
{{ date1|naturaltime }} renvoie "maintenant"<br />
{{ date2|naturaltime }} renvoie "il y a 29 secondes"<br />
{{ date3|naturaltime }} renvoie "il y a une heure"<br />
{{ date4|naturaltime }} renvoie "il y a une heure"<br />
{{ date5|naturaltime }} renvoie "il y a 1 jour, 1 heure"<br />
{{ date6|naturaltime }} renvoie "dans 1 jour, 4 heures"<br />
```

ordinal

Convertit un entier en chaîne de caractères représentant une place dans un classement.

Exemple, encore une fois en français :

Code : Jinja

```
{{ 1|ordinal }} renvoie 1<sup>er</sup><br />
{{ "2"|ordinal }} renvoie 2<sup>e</sup><br />
{{ 98|ordinal }} renvoie 98<sup>e</sup><br />
```

Le filtre prend à la fois des entiers et des chaînes de caractères en paramètre.

Nous avons fini le tour du module `humanize` ! Celui-ci contient au total six filtres vous facilitant le travail pour certaines opérations esthétiques dans vos templates.

En résumé

- Django est un framework très puissant, il propose de nombreux modules complémentaires et optionnels pour simplifier le développement.
- Ce cours a traité de quelques-uns de ces modules, mais il est impossible de les présenter tous : la documentation présente de façon complète chacun d'entre eux.
- Il existe des centaines de modules non officiels permettant de compléter votre installation et d'intégrer de nouvelles fonctionnalités.
- Nous avons présenté ici `humanize`, qui rend vos données plus naturelles dans vos templates, et `flatpages` qui permet de gérer vos pages statiques via l'administration.

Partie 5 : Annexes

Déployer votre application en production

Tout au long du cours, 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 sur un serveur dédié Linux. Si vous ne disposez pas de serveur dédié, sachez qu'il existe certains hébergeurs qui proposent également l'installation et la gestion d'un projet Django, nous en avons listé quelques-uns à la fin de ce chapitre.

Le déploiement

Contrairement à ce que certains peuvent penser, 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 suffisantes 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) sur un serveur Linux dont vous devez avoir un accès root. Le protocole WSGI est une sorte de couche qui permet à un serveur web et une application web Python de communiquer ensemble.

Sachez que vous pouvez également déployer un projet Django sur certains hébergements mutualisés le supportant. Généralement, une documentation de l'hébergeur sera mise à votre disposition pour vous indiquer comment le déployer sur leur infrastructure. Si vous souhaitez déployer Django sur votre propre serveur, ce chapitre vous expliquera tout ce que vous devrez savoir.

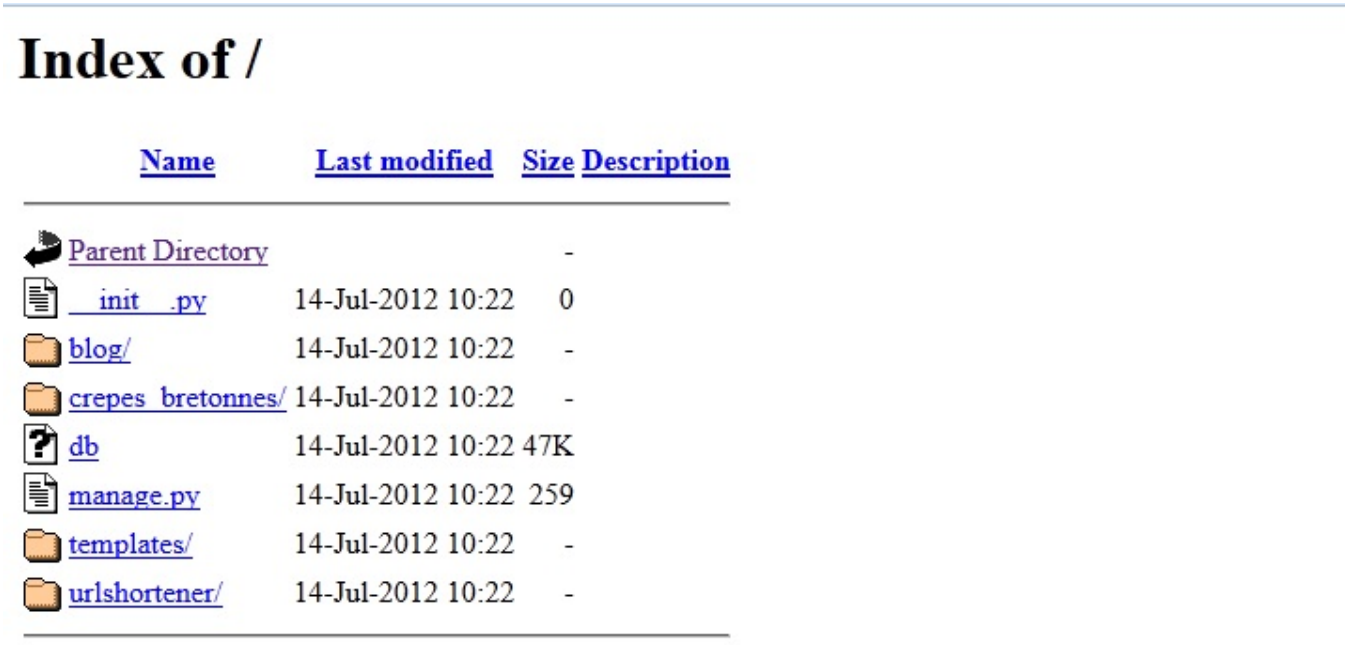
Par défaut, Django fournit un fichier `wsgi.py` qui s'occupera de cette liaison. Pour rappel :

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, voici le résultat :



Une liste de fichiers Python que nous ne pouvons que télécharger

Non seulement votre code n'est pas exécuté, mais il est lisible par tous. Il 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 ! Nous ne couvrirons pas l'installation d'Apache2, ni sa configuration basique. Sachez bien évidemment que vous pouvez utiliser d'autres serveurs HTTP : nginx, lighttpd, etc.

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
WSGIPythonPath /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 deuxième ligne, `WSGIPythonPath`, permet de rendre accessible votre projet via la commande `import` en Python. 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 avions 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. Il est très important de le faire, sans quoi les erreurs et des données sensibles seront affichées !
- Remplir la variable `ALLOWED_HOSTS` qui doit contenir les différentes adresses depuis lesquelles le site peut être accédé. Exemple : `ALLOWED_HOSTS = ['www.crepes-bretonnes.com', '.super-crepes.fr']`. Le point au début du deuxième élément de la liste permet d'indiquer que tous les sous-domaines sont acceptés, autrement dit, les domaines suivants seront accessibles : super-crepes.fr, www.super-crepes.fr, media.super-crepes.fr, coucou.super-crepes.fr, etc.
- Adaptez la connexion à la base de données en fonction de ce que vous souhaitez utiliser en production. Nous vous conseillons d'utiliser MySQL ou PostgreSQL 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 regardez l'exception lancée lors du chargement d'une page.

Gardez un œil 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 e-mail à chaque erreur rencontrée avec le contexte de celle-ci !

Cet e-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 l'e-mail).

Activer l'envoi d'e-mails

Dans un premier temps, assurez-vous qu'un serveur d'e-mails est installé sur votre machine, permettant d'envoyer des e-mails via le protocole SMTP.
Pour pouvoir recevoir ces alertes, assurez-vous que votre variable DEBUG est à `False`. Les e-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` est correcte et à jour. En effet, ce sont les administrateurs présents dans cette liste qui recevront les e-mails d'erreurs. Pour rappel, lors de la création de notre projet, nous avions mis ceci :

Code : Python

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

Ici, les e-mails d'erreurs sont envoyés aux deux personnes, en même temps.

Par défaut, Django envoie les e-mails depuis l'adresse `root@localhost`. Cependant, certaines boîtes e-mail rejettent cette adresse, ou tout simplement vous souhaiteriez 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 avertis des pages 404

Par défaut, les pages non trouvées ne sont pas signalées par e-mail. Si vous voulez toutefois les recevoir, ajoutez les lignes suivantes dans votre `settings.py` :

Code : Python

```
SEND_BROKEN_LINK_EMAILS = True
MANAGERS = ADMINS # À 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 enverra un e-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 survienne lors de la saisie de données sensibles : saisie d'un mot de passe, d'un numéro de carte bleue, etc. Pour des raisons de sécurité, il est nécessaire de cacher ces informations dans les e-mails d'erreurs ! Pour ce faire, nous devons 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 e-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 personne n'est jamais à l'abri d'une fuite

(vol de compte e-mail, écoute de paquets…).

Hébergeurs supportant Django

Nous avons vu comment installer un projet Django sur un serveur dédié. Cependant, tout le monde n'a pas la chance d'avoir un serveur à soi. Il existe toutefois d'autres possibilités. 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.	Gratuit et Payant
Heroku	Un hébergeur devenu très à la mode. Très flexible et puissant, il permet de réaliser énormément d'opérations différentes et gère parfaitement Django.	Gratuit et Payant
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 liste plus exhaustive est disponible ici (site officiel en anglais). Comme vous pouvez le voir, la majorité de ces hébergeurs sont payants.

En résumé

- Il ne faut pas utiliser le serveur `python manage.py runserver` en production.
- Une des méthodes d'installation possible passe par Apache2 avec le `mod_wsgi`, en exécutant le script `wsgi.py` contenu dans le répertoire du projet.
- Si l'on désactive le mode `DEBUG`, Django enverra un e-mail à toutes les personnes listées dans le tuple `ADMINS` en cas d'erreur 500 sur le site. Il est possible d'être averti en cas d'erreurs 404, et de filtrer les données sensibles envoyées (telles que les mots de passe).

L'utilitaire manage.py

Tout au long de ce cours, vous avez utilisé la commande `manage.py` fournie par Django pour différents besoins : créer un projet, créer une application, mettre à jour la structure de la base de données, ajouter un super-utilisateur, enregistrer et compiler des traductions, etc. Pour chaque tâche différente, `manage.py` possède une commande adaptée. Bien évidemment, nous n'avons vu qu'une fraction des possibilités offertes par cet outil. Dans ce chapitre, nous aborderons toutes les commandes une par une, avec leurs options, qu'elles aient déjà été introduites auparavant ou non.

N'hésitez pas à survoler cette liste, vous découvrirez peut-être une commande qui pourrait vous être utile à un moment donné, et vous n'aurez qu'à revenir dans ce chapitre pour découvrir comment elle marche exactement.

Les commandes de base

Prérequis

La plupart des commandes acceptent des arguments. Leur utilisation est propre à chaque commande et est indiquée dans le titre de celle-ci. Un argument entre chevrons `< ... >` indique qu'il est obligatiore, tandis qu'un argument entre crochets `[ ... ]` indique qu'il est optionnel. Référez-vous ensuite à l'explication de la commande pour déterminer comment ces arguments sont utilisés.

Certaines commandes possèdent également des options pour modifier leur fonctionnement. Celles-ci commencent généralement toutes par deux tirets « `--` » et s'ajoutent avant ou après les arguments de la commande, s'il y en a. Les options de chaque commande seront présentées après l'explication du fonctionnement global de la commande.

Les commandes seront introduites par thème : nous commencerons par les outils de base dont la plupart ont déjà été expliqués, puis nous expliquerons toutes les commandes liées à la gestion de la base de données, et nous finirons par les commandes spécifiques à certains modules introduits auparavant, comme le système utilisateurs.

Liste des commandes

runserver [port ou adresse:port]

Lance un serveur de développement local pour le projet en cours. Par défaut, ce serveur est accessible depuis le port 8000 et l'adresse IP 127.0.0.1, qui est l'adresse locale de votre machine. Le serveur redémarre à chaque modification du code dans le projet. Bien évidemment, ce serveur n'est pas destiné à la production. Le framework ne garantit ni les performances, ni la sécurité du serveur de développement pour une utilisation en production.

À chaque démarrage du serveur et modification du code, la commande `validate` (expliquée par la suite) est lancée afin de vérifier si les modèles sont corrects.

Il est possible de spécifier un port d'écoute différent de celui par défaut en spécifiant le numéro de port. Ici, le serveur écoutera sur le port 9000 :

Code : Console

```
python manage.py runserver 9000
```

De même, il est possible de spécifier une adresse IP différente de 127.0.0.1 (pour l'accessibilité depuis le réseau local par exemple). Il n'est pas possible de spécifier une adresse IP sans spécifier le port :

Code : Console

```
python manage.py runserver 192.168.1.6:7000
```

Le serveur de développement prend également en charge l'IPv6. Nous pouvons dès lors spécifier une adresse IPv6, tout comme nous spécifierions une adresse IPv4, à condition de la mettre entre crochets :

Code : Console

```
python manage.py runserver [2001:0db8:1234:5678::9]:7000
```

--ipv6, -6
Il est également possible de remplacer l'adresse locale IPv4 127.0.0.1 par l'adresse locale IPv6::1 en spécifiant l'option `--ipv6` ou `-6` :

Code : Console

```
python manage.py runserver -6
```

--noreload
Empêche le serveur de développement de redémarrer à chaque modification du code. Il faudra procéder à un redémarrage manuel pour que les changements soient pris en compte.

--nothreading
Le serveur utilise par défaut des threads. En utilisant cette option, ceux-ci ne seront pas utilisés.

shell

Lance un interpréteur interactif Python. L'interpréteur sera configuré pour le projet et des modules de ce dernier pourront être importés directement.

version

Indique la version de Django installée :

Code : Console

```
python manage.py version
1.5
```

help <commande>

Affiche de l'aide pour l'utilisation de `manage.py`. Utilisez `manage.py help <commande>` pour accéder à la description d'une commande avec ses options.

startproject <nom> [destination]

Cette commande s'utilise obligatoirement avec `django-admin.py` : vous ne disposez pas encore de `manage.py` vu que le projet n'existe pas.

Crée un nouveau projet utilisant le nom donné en paramètre. Un nouveau dossier dans le répertoire actuel utilisant le nom du projet sera créé et les fichiers de base y seront insérés (`manage.py`, le sous-dossier contenant le `settings.py` notamment, etc.).

Il est possible d'indiquer un répertoire spécifique pour accueillir le nouveau projet :

Code : Console

```
django-admin.py startproject crepes_bretonnes /home/crepes/projets/crepes
```

Ici, tous les fichiers seront directement insérés dans le dossier `/home/crepes/projets/crepes`.

--template
Cette option permet d'indiquer un modèle de projet à copier, plutôt que d'utiliser celui par défaut. Il est possible de spécifier un chemin vers le dossier contenant les fichiers ou une archive (.tar.gz, .tar.bz2, .tgz, .tbz, .zip) contenant également le modèle.
Exemple :

Code : Console

```
django-admin.py startproject --
template=/home/mathx/projets/modele_projet crepes_bretonnes
```

Indiquer une URL (http, https ou ftp) vers une archive est également possible :

Code : Console

```
django-admin.py startproject --
template=http://monsite.com/modele_projet.zip crepes_bretonnes
```

Django se chargera de télécharger l'archive, de l'extraire, et de la copier dans le nouveau projet.

startapp <nom> [destination]

Crée une nouvelle application dans un projet. L'application sera nommée selon le nom passé en paramètre. Un nouveau dossier sera créé dans le projet avec les fichiers de base (`models.py`, `views.py`, ...).

Si vous le souhaitez, vous pouvez indiquer un répertoire spécifique pour accueillir l'application, sinon l'application sera ajoutée dans le répertoire actuel :

Code : Console

```
python manage.py startapp blog /home/crepes/projets/crepes/global/blog
```

`--template`
Tout comme `startproject`, cette option permet d'indiquer un modèle d'application à copier, plutôt que d'utiliser celui par défaut. Il est possible de spécifier un chemin vers le dossier contenant les fichiers ou une archive (.tar.gz, .tar.bz2, .tgz, .tbz, .zip) contenant le modèle. Exemple :

Code : Console

```
python manage.py startapp --
template=/home/mathx/projets/modele_app crepes_bretonnes/blog
```

Indiquer une URL (http, https ou ftp) vers une archive est également possible :

Code : Console

```
python manage.py startapp --
template=http://monsite.com/modele_app.zip crepes_bretonnes
```

Django se chargera de télécharger l'archive, de l'extraire, et de la copier dans la nouvelle application.

diffsettings

Indique les variables de votre `settings.py` qui ne correspondent pas à la configuration par défaut d'un projet neuf. Les variables se terminant par `###` sont des variables qui n'apparaissent pas dans la configuration par défaut.

validate

Vérifie et valide tous les modèles des applications installées. Si un modèle est invalide, une erreur est affichée.

test <application ou identifiant de test>

Lance les tests unitaires d'un projet, d'une application, d'un test ou d'une méthode en particulier. Pour lancer tous les tests d'un projet, il ne faut spécifier aucun argument. Pour une application, il suffit d'indiquer son nom (sans inclure le nom du projet) :

Code : Console

```
python manage.py test blog
```

Pour ne lancer qu'un seul test unitaire, il suffit d'ajouter le nom du test après le nom de l'application :

Code : Console

```
python manage.py test blog.BlogUnitTest
```

Finalement, pour ne lancer qu'une seule méthode d'un test unitaire, il faut également la spécifier après l'identifiant du test :

Code : Console

```
python manage.py test blog.BlogUnitTest.test_lecture_article
```

`--failfast`
Arrête le processus de vérification de tous les tests dès qu'un seul test a échoué et rapporte l'échec en question.

testserver <fixture fixture …>

Lance un serveur de développement avec les données des fixtures indiquées. Les fixtures sont des fichiers contenant des données pour remplir une base de données. Pour mieux comprendre le fonctionnement des fixtures, référez-vous à la commande `loaddata`.

Cette commande effectue trois actions :

1. Elle crée une base de données vide.
2. Une fois la base de données créée, elle la remplit avec les données des fixtures passées en paramètre.
3. Un serveur de développement est lancé avec la base de données venant d'être remplie.

Cette commande peut se révéler particulièrement utile lorsque vous devez régulièrement changer de données pour tester la vue que vous venez d'écrire. Au lieu de devoir à chaque fois créer et supprimer des données manuellement dans votre base pour vérifier chaque situation possible, il vous suffira de lancer le serveur de développement avec des fixtures adaptées à chaque situation.

`--addrport [port ou adresse:port]`
Tout comme pour `runserver`, le serveur de développement sera accessible par défaut depuis 127.0.0.1:8000. Il est également possible de spécifier un port d'écoute ou une adresse spécifique :

Code : Console

```
python manage.py testserver --addrport 7000 fixture.json
```

Code : Console

```
python manage.py testserver --addrport 192.168.1.7:9000 fixture.json
```

Toutes les commandes de cette section ont une option commune : `--database` qui permet de spécifier l'alias (indiqué dans votre `settings.py`) de la base de données sur laquelle la commande doit travailler si vous disposez de plusieurs bases de données. Exemple :

Code : Console

```
python manage.py syncdb --database=master
```

syncdb

Crée une table dans la base de données pour chaque modèle issu des applications installées (depuis `INSTALLED_APPS` dans `settings.py`). Cet outil crée une table uniquement si elle n'existe pas déjà dans la base de données. À noter que si vous avez modifié votre modèle, `syncdb` ne mettra pas à jour la structure de la table correspondante. Vous devrez faire cela manuellement.

Lors de l'installation de l'application `django.contrib.auth`, `syncdb` permet de créer un super-utilisateur directement. De plus, l'outil cherchera des fixtures commençant par `initial_data` avec une extension appropriée (typiquement json ou xml, vos fixtures doivent donc s'appeler `initial_data.json` par exemple). Pour mieux comprendre le fonctionnement des fixtures, référez-vous à la commande `loaddata`.

`--noinput`
Supprime toutes les confirmations adressées à l'utilisateur.

dbshell

Lance le client de gestion de votre base de données en ligne de commande, selon les paramètres spécifiés dans votre `settings.py`. Cette commande présume que vous avez installé le client adapté à votre base de données et que celui-ci est accessible depuis votre console.

- Pour PostgreSQL, l'utilitaire psql sera lancé ;
- Pour MySQL, l'utilitaire mysql sera lancé ;
- Pour SQLite, l'utilitaire sqlite3 sera lancé.

dumpdata <application application.Modele …>

Affiche toutes les données d'applications ou de modèles spécifiques contenues dans votre base de données dans un format texte sérialisé. Par défaut, le format utilisé sera le JSON. Ces données sont des fixtures qui pourront être utilisées par la suite dans des commandes comme `loaddata`, `syncdb` ou `testserver`. Exemple :

Code : Console

```
python manage.py dumpdata blog.Article
[{"pk": 1, "model": "blog.article", "fields": {"date": "2012-07-
11T15:51:08.607Z", "titre": "Les crêpes c'est trop bon", "categorie": 1,
 "auteur": "Maxime", "contenu": "Vous saviez que les crêpes bretonnes c'est trop bo
 La pêche c'est nul."}},
 {"pk": 2, "model": "blog.article", "fields": {"date": "2012-07-
11T16:25:53.262Z", "titre": "Un super titre d'article !",
 "categorie": 1, "auteur": "Mathieu", "contenu": "Un super contenu ! (ou pas)"}}]
```

Il est possible de spécifier plusieurs applications et modèles directement :

Code : Console

```
python manage.py dumpdata blog.Article blog.Categorie auth
```

Ici, les modèles `Article` et `Categorie` de l'application `blog` et tous les modèles de l'application `django.contrib.auth` seront sélectionnés et leurs données affichées. Si vous ne spécifiez aucune application ou modèle,

tous les modèles du projet seront repris.

`--all`
Force l'affichage de tous les modèles, à utiliser si certains modèles sont filtrés.

`--format <fmt>`
Change le format utilisé. Utilisez `--format xml` pour utiliser le format XML par exemple.

`--indent <nombre d'espace>`
Par défaut, comme vu plus tôt, toutes les données sont affichées sur une seule et même ligne. Cela ne facilite pas la lecture pour un humain, vous pouvez donc utiliser cette option pour indenter le rendu avec le nombre d'espaces spécifié pour chaque indentation.

`--exclude`
Empêche certains modèles ou applications d'être affichés. Par exemple, si vous souhaitez afficher toutes les données de l'application `blog`, sauf celles du modèle `blog.Article`, vous pouvez procéder ainsi :

Code : Console

```
python manage.py dumpdata blog --exclude blog.Article
```

`--natural`
Utilise une représentation différente pour les relations `ForeignKey` et `ManyToMany` pour d'éventuelles situations à problèmes. Si vous affichez `contrib.auth.Permission` ou utilisez des `ContentType` dans vos modèles, vous devriez utiliser cette option.

`loaddata <fixture fixture …>`

Enregistre dans la base de données les fixtures passées en argument. Les fixtures sont des fichiers contenant des données de votre base de données. Ces données sont enregistrées dans un format texte spécifique, généralement en JSON ou en XML. Django peut dès lors directement lire ces fichiers et ajouter leur contenu à la base de données. Vous pouvez créer des fixtures à partir de la commande `dumpdata`.

La commande `loaddata` ira chercher les fixtures dans trois endroits différents :

- Dans un dossier `fixtures` dans chaque application ;
- Dans un dossier indiqué par la variable `FIXTURE_DIRS` dans votre `settings.py` ;
- À partir du chemin vers le fichier donné en argument, absolu ou relatif.

Vous pouvez omettre d'indiquer l'extension de la fixture :

Code : Console

```
manage.py loaddata ma_fixture
```

Dans ce cas, Django ira chercher dans tous les endroits susmentionnés et prendra toutes les fixtures ayant une terminaison correspondant à un format de fixtures (.json pour le JSON ou .xml pour le XML par exemple). Dès lors, si vous avez un fichier `ma_fixture.json` dans un dossier `fixtures` d'une application, celui-ci sera sélectionné.

Bien entendu, vous pouvez également spécifier un fichier avec l'extension :

Code : Console

```
manage.py loaddata ma_fixture.xml
```

Dans ce cas, le fichier devra obligatoirement s'appeler `ma_fixture.xml` pour être sélectionné.

Django peut également gérer des fixtures compressées. Si vous indiquez `ma_fixture.json` comme fixture à utiliser, Django cherchera `ma_fixture.json`, `ma_fixture.json.zip`, `ma_fixture.json.gz` ou `ma_fixture.json.bz2`. S'il

tombe sur une fixture compressée, il la décompressera, puis lancera le processus de copie.

inspectdb

Inspecte la base de données spécifiée dans votre `settings.py` et crée à partir de sa structure un `models.py`. Pour chaque table dans la base de données, un modèle correspondant sera créé. Cette commande construit donc des modèles à partir de tables, il s'agit de l'opération inverse de la commande `syncdb`.

Notons qu'il ne s'agit ici que d'un raccourci. Les modèles créés automatiquement doivent être relus et vérifiés manuellement. Certains champs nommés avec des mots-clés de Python (comme `class` ou `while` par exemple) peuvent avoir été renommés pour éviter d'éventuelles collisions. Il se peut également que Django n'ait pas réussi à identifier le type d'un champ et le remplace par un `TextField`. Il convient également d'être particulièrement attentif aux relations `ForeignKey`, `ManyToManyField`, etc.

Voici un extrait d'un `inspectdb`, reprenant le modèle `Article` de notre application `blog` créée dans le cours :

Code : Python

```
class BlogArticle(models.Model):
    id = models.IntegerField(primary_key=True)
    titre = models.CharField(max_length=100)
    auteur = models.CharField(max_length=42)
    contenu = models.TextField()
    date = models.DateTimeField()
    categorie = models.ForeignKey(BlogCategorie)
    class Meta:
        db_table = u'blog_article'
```

flush

Réinitialise la base de données. Celle-ci retrouvera l'état dans laquelle elle était après un `syncdb` : les tables seront recréées, toutes les données seront perdues et les fixtures `initial_data` seront réinsérées.

sql <application application …>

Construit et affiche les requêtes SQL permettant de créer les tables dans la base de données à partir des modèles d'une ou des applications indiquées. Exemple :

Code : Console

```
python manage.py sql blog auth
```

sqlcustom <application application …>

Affiche des requêtes SQL contenues dans des fichiers. Django affiche les requêtes contenues dans les fichiers `<application>/sql/<modele>.sql` où **<application>** est le nom de l'application donné en paramètre et **<modele>** un modèle quelconque de l'application. Si nous avons l'application `blog` incluant le modèle `Article`, la commande `manage.py sqlcustom blog` affichera les requêtes du fichier `blog/sql/article.sql` s'il existe, et y ajoutera toutes les autres requêtes des autres modèles de l'application.

À chaque fois que vous réinitialisez votre base de données par exemple, cette commande vous permet d'exécuter facilement des requêtes SQL spécifiques en les joignant à la commande `dbshell` via des pipes sous Linux ou Mac OS X :

Code : Console

```
python manage.py sqlcustom blog | python manage.py dbshell
```

sqlall <application application …>

Combinaison des commandes `sql` et `sqlcustom`. Cette commande affichera d'abord les requêtes pour créer les tables, puis les requêtes personnalisées. Référez-vous à ces deux commandes pour connaître le fonctionnement exact de cette commande.

sqlclear <application application …>

Construit et affiche les requêtes SQL permettant de supprimer les tables dans la base de données à partir des modèles d'une ou des applications indiquées.

sqlflush <application application …>

Construit et affiche les requêtes SQL permettant de vider les tables dans la base de données à partir des modèles d'une ou des applications indiquées. Les requêtes SQL affichées agissent exactement comme la commande `flush`.

sqlindexes <application application …>

Construit et affiche les requêtes SQL permettant de créer les index des tables dans la base de données à partir des modèles d'une ou des applications indiquées.

sqlsequencereset <application application …>

Construit et affiche les requêtes SQL permettant de réinitialiser les séquences des tables dans la base de données à partir des modèles d'une ou des applications indiquées. Les séquences sont des index permettant de déterminer l'index à assigner à la prochaine entrée créée.

Les commandes d'applications

clearsessions

Supprime les sessions expirées de `django.contrib.sessions`.

changepassword [pseudo]

Permet de changer le mot de passe d'un utilisateur en spécifiant son pseudo :

Code : Console

```
python manage.py changepassword Mathieu
```

Si aucun nom d'utilisateur n'est spécifié, Django prendra le nom d'utilisateur de la session actuelle. Cette commande n'est disponible que si le système d'utilisateurs (`django.contrib.auth`) est installé.

createsuperuser

Permet de créer un super-utilisateur (un utilisateur avec tous les pouvoirs). Cette commande demandera le pseudo, l'adresse e-mail — si ceux-ci n'ont pas été spécifiés à partir des options — et le mot de passe.

`--username` et `--email`
Permettent de spécifier directement le nom et l'adresse e-mail de l'utilisateur.

Si ces deux options sont indiquées, vous devrez spécifier le mot de passe manuellement par la suite afin que l'utilisateur puisse se connecter.

makemessages

Parcourt tous les fichiers de l'arborescence à partir du dossier actuel pour déterminer les chaînes de caractères à traduire et crée ou met à jour les fichiers de traduction. Référez-vous au chapitre sur l'internationalisation pour plus d'informations.

`--all, -a`
Met à jour les chaînes à traduire pour tous les langages.

`--extensions`
Indique de ne sélectionner que les fichiers qui ont une extension spécifique :

Code : Console

```
python manage.py makemessages --extension xhtml
```

… ne prendra que les fichiers xHTML.

Code : Console

```
python manage.py --extension=html,txt --extension xml
```

…prendra les fichiers HTML, TXT et XML.

`--locale`
Permet de ne mettre à jour qu'une seule langue :

Code : Console

```
python manage.py makemessages --locale fr_FR
```

`--symlinks`
Autorise Django à suivre les liens symboliques en explorant les fichiers.

`--ignore, -i`
Permet d'ignorer certains fichiers :

Code : Console

```
python manage.py makemessages --ignore=blog/* --ignore=*.html
```

Tous les fichiers HTML et du dossier `blog` seront ignorés.

`--no-wrap`
Empêche Django de répartir les longues chaînes de caractères en plusieurs lignes dans les fichiers de traduction.

`--no-location`
Empêche Django d'indiquer la source de la chaîne de caractères dans les fichiers de traduction (nom du fichier et ligne dans celui-ci).

compilemessages

Compile les fichiers de traduction `.po` vers des fichiers `.mo` afin que gettext puisse les utiliser. Référez-vous au chapitre sur l'internationalisation pour plus d'informations.

`--locale`
Permet de ne compiler qu'une seule langue :

Code : Console

```
python manage.py compilemessages --locale fr_FR
```

createcachetable <nom de la table>

Crée une table de cache pour le système de cache. Référez-vous au chapitre sur le système de cache pour comprendre son fonctionnement. La table sera nommée selon le nom donné en paramètre.