Ssx'z
MathX

Zeste de savoir

Développez votre site web avec le framework Django

12 août 2019

Table des matières

I. Présentation de Django 7

1. Créez vos applications web avec Django 9
1.1. Qu'est-ce qu'un framework ? . . . 9
1.1.1. Quels sont les avantages d'un framework ? . . . 9
1.1.2. Quels sont les désavantages d'un framework ? . . . 10
1.2. Qu'est-ce que Django ? . . . 10
1.2.1. Pourquoi ce succès ? . . . 11
1.2.2. Une communauté à votre service . . . 11
1.3. Téléchargement et installation . . . 11
1.3.1. Linux et Mac OS . . . 12
1.3.2. Windows . . . 12
1.3.3. Vérification de l'installation . . . 14
1.4. En résumé . . . 14

2. Le fonctionnement de Django 15
2.1. Un peu de théorie : l'architecture MVC . . . 15
2.2. La spécificité de Django : le modèle MVT . . . 16
2.3. Projets et applications . . . 17
2.4. En résumé . . . 18

3. Gestion d'un projet 20
3.1. Créons notre premier projet . . . 20
3.2. Configurez votre projet . . . 23
3.3. Créons notre première application . . . 24
3.4. En résumé . . . 26

4. Les bases de données et Django 27
4.1. Une base de données, c'est quoi ? . . . 27
4.2. Le langage SQL et les gestionnaires de base de données . . . 28
4.3. La magie des ORM . . . 29
4.4. Le principe des clés étrangères . . . 30
4.5. En résumé . . . 31

II. Premiers pas 32

5. Votre première page grâce aux vues 34
5.1. Hello World ! . . . 34
5.2. Routage d'URL : comment j'accède à ma vue ? . . . 35

5.3. Organiser proprement vos URL . . . 38
5.3.1. Comment procède-t-on ? . . . 38
5.4. Passer des arguments à vos vues . . . 40
5.5. Des réponses spéciales . . . 42
5.5.1. Simuler une page non trouvée . . . 42
5.5.2. Rediriger l'utilisateur . . . 43
5.6. En résumé . . . 45

6. Les templates **46**
6.1. Lier template et vue . . . 46
6.2. Affichons nos variables à l'utilisateur . . . 50
6.2.1. Affichage d'une variable . . . 50
6.2.2. Les filtres . . . 51
6.3. Manipulons nos données avec les tags . . . 52
6.3.1. Les conditions : {% if %} . . . 52
6.3.2. Les boucles : {% for %} . . . 53
6.3.3. Le tag {% block %} . . . 54
6.3.4. Les liens vers les vues : {% url %} . . . 56
6.3.5. Les commentaires : {% comment %} . . . 57
6.4. Ajoutons des fichiers statiques . . . 58
6.5. En résumé . . . 59

7. Les modèles **60**
7.1. Créer un modèle . . . 60
7.2. Jouons avec des données . . . 62
7.3. Les liaisons entre modèles . . . 66
7.4. Les modèles dans les vues . . . 72
7.4.1. Afficher les articles du blog . . . 72
7.4.2. Afficher un article précis . . . 74
7.5. En résumé . . . 77

8. L'administration **78**
8.1. Mise en place de l'administration . . . 78
8.1.1. Les modules `django.contrib` . . . 78
8.1.2. Accédons à cette administration ! . . . 78
8.2. Première prise en main . . . 80
8.3. Administrons nos propres modèles . . . 83
8.4. Personnalisons l'administration . . . 84
8.4.1. Modifier l'aspect des listes . . . 84
8.4.2. Modifier le formulaire d'édition . . . 89
8.4.3. Retour sur notre problème de slug . . . 92
8.5. En résumé . . . 93

9. Les formulaires **94**
9.1. Créer un formulaire . . . 94
9.2. Utiliser un formulaire dans une vue . . . 95
9.3. Créons nos propres règles de validation . . . 98
9.4. Des formulaires à partir de modèles . . . 102
9.5. En résumé . . . 107

10. La gestion des fichiers **108**
10.1. Enregistrer une image . . . 108
10.2. Afficher une image . . . 110
10.3. Encore plus loin . . . 112
10.4. En résumé . . . 113

11. TP : un raccourcisseur d'URL **114**
11.1. Cahier des charges . . . 114
11.2. Correction . . . 116

III. Techniques avancées **122**

12. Les vues génériques **124**
12.1. Premiers pas avec des pages statiques . . . 124
12.2. Lister et afficher des données . . . 126
12.2.1. Une liste d'objets en quelques lignes avec ListView . . . 126
12.2.2. Afficher un article via `DetailView` . . . 129
12.3. Agir sur les données . . . 131
12.3.1. CreateView . . . 131
12.3.2. UpdateView . . . 132
12.3.3. DeleteView . . . 135
12.4. En résumé . . . 137

13. Techniques avancées dans les modèles **139**
13.1. Les requêtes complexes avec Q . . . 139
13.2. L'agrégation . . . 141
13.3. L'héritage de modèles . . . 144
13.3.1. Les modèles parents abstraits . . . 144
13.3.2. Les modèles parents classiques . . . 145
13.3.3. Les modèles proxy . . . 146
13.4. L'application ContentType . . . 147
13.5. En résumé . . . 150

14. Simplifions nos templates : filtres, tags et contextes **151**
14.1. Préparation du terrain : architecture des filtres et tags . . . 151
14.2. Personnaliser l'affichage de données avec nos propres filtres . . . 152
14.2.1. Un premier exemple de filtre sans argument . . . 153
14.2.2. Un filtre avec arguments . . . 155
14.3. Les contextes de templates . . . 157
14.3.1. Un exemple maladroit : afficher la date sur toutes nos pages . . . 157
14.3.2. Factorisons encore et toujours . . . 158
14.4. Des structures plus complexes : les custom tags . . . 160
14.4.1. Première étape : la fonction de compilation . . . 161
14.4.2. Passage de variable dans notre tag . . . 165
14.4.3. Les *simple tags* . . . 167
14.4.4. Quelques points à ne pas négliger . . . 168
14.5. En résumé . . . 168

15. Les signaux et middlewares **170**
15.1. Notifiez avec les signaux . . . 170
15.2. Contrôlez tout avec les middlewares . . . 174
15.3. En résumé . . . 177

IV. Des outils supplémentaires **178**

16. Les utilisateurs **180**
16.1. Commençons par la base . . . 180
16.1.1. L'utilisateur . . . 180
16.1.2. Les mots de passe . . . 181
16.1.3. Étendre le modèle User . . . 182
16.2. Passons aux vues . . . 183
16.2.1. La connexion . . . 183
16.2.2. La déconnexion . . . 184
16.2.3. Intéragir avec le profil utilisateur . . . 185
16.3. Les vues génériques . . . 186
16.3.1. Se connecter . . . 186
16.3.2. Se déconnecter . . . 186
16.3.3. Se déconnecter puis se connecter . . . 187
16.3.4. Changer le mot de passe . . . 187
16.3.5. Confirmation du changement de mot de passe . . . 187
16.3.6. Demande de réinitialisation du mot de passe . . . 187
16.3.7. Confirmation de demande de réinitialisation du mot de passe . . . 188
16.3.8. Réinitialiser le mot de passe . . . 188
16.3.9. Confirmation de la réinitialisation du mot de passe . . . 188
16.4. Les permissions et les groupes . . . 189
16.4.1. Les permissions . . . 189
16.4.2. Les groupes . . . 190
16.5. En résumé . . . 190

17. Les messages **192**
17.1. Les bases . . . 192
17.2. Dans les détails . . . 193
17.3. En résumé . . . 194

18. La mise en cache **195**
18.1. Cachez-vous ! . . . 195
18.1.1. Dans des fichiers . . . 195
18.1.2. Dans la mémoire . . . 196
18.1.3. Dans la base de données . . . 196
18.1.4. En utilisant Memcached . . . 196
18.1.5. Pour le développement . . . 197
18.2. Quand les données jouent à cache-cache . . . 197
18.2.1. Cache par vue . . . 197
18.2.2. Dans les templates . . . 198
18.2.3. La mise en cache de bas niveau . . . 198
18.3. En résumé . . . 199

19. La pagination **201**
19.1. Exerçons-nous en console . . . 201
19.2. Utilisation concrète dans une vue . . . 202
19.3. En résumé . . . 203

20. L'internationalisation **204**
20.1. Qu'est-ce que le i18n et comment s'en servir ? . . . 204
20.2. Traduire les chaînes dans nos vues et modèles . . . 206
20.2.1. Cas des modèles . . . 209
20.3. Traduire les chaînes dans nos templates . . . 209
20.3.1. Le tag {% trans %} . . . 209
20.3.2. Le tag {% blocktrans %} . . . 210
20.3.3. Aidez les traducteurs en laissant des notes ! . . . 210
20.4. Sortez vos dictionnaires, place à la traduction ! . . . 211
20.4.1. Génération des fichiers .po . . . 211
20.4.2. Génération des fichiers .mo . . . 213
20.5. Le changement de langue . . . 213
20.6. En résumé . . . 214

21. Les tests unitaires **215**
21.1. Nos premiers tests . . . 215
21.1.1. Ecrire un test unitaire . . . 215
21.1.2. Lançons notre test en console . . . 217
21.1.3. Initialisation de données pour nos tests . . . 217
21.2. Testons des vues . . . 218
21.3. En résumé . . . 219

22. Ouverture vers de nouveaux horizons : django.contrib **220**
22.1. Vers l'infini et au-delà . . . 220
22.2. Dynamisons nos pages statiques avec flatpages ! . . . 221
22.2.1. Installation du module . . . 222
22.2.2. Gestion et affichage des pages . . . 223
22.2.3. Lister les pages statiques disponibles . . . 224
22.3. Rendons nos données plus lisibles avec humanize . . . 225
22.3.1. `apnumber` . . . 225
22.3.2. `intcomma` . . . 225
22.3.3. `intword` . . . 226
22.3.4. `naturalday` . . . 226
22.3.5. `naturaltime` . . . 226
22.3.6. `ordinal` . . . 226
22.4. En résumé . . . 227

V. Annexes **228**

23. Déployer votre application en production **229**
23.1. Le déploiement . . . 229
23.1.1. Configuration du projet . . . 229
23.1.2. Installation avec Apache2 . . . 230

23.1.3. Installation avec nginx et gunicorn . . . 231
23.2. Gardez un œil sur le projet . . . 232
23.2.1. Activer l'envoi d'e-mails . . . 233
23.2.2. Quelques options utiles… . . . 233
23.2.3. Introduction à Sentry, pour garder un oeil encore plus attentif . . . 234
23.3. Hébergeurs supportant Django . . . 235
23.4. En résumé . . . 236

24. L'utilitaire manage.py **237**
24.1. Les commandes de base . . . 237
24.1.1. Prérequis . . . 237
24.1.2. Liste des commandes . . . 237
24.2. La gestion de la base de données . . . 241
24.3. Les commandes d'applications . . . 245

« ***Le framework web pour les perfectionnistes sous pression*** »

Vous connaissez les langages HTML et Python et souhaitez créer des sites web dynamiques ? Ce cours est fait pour vous ! Il vous apprendra pas à pas comment prendre en main Django, un framework très populaire permettant de créer des sites web à l'aide du langage Python.

Comparable aux frameworks Ruby on Rails et Symfony2, Django s'occupe de gérer les couches basses d'un site (sessions, sécurité...) et peut même générer une interface d'administration tout seul ! L'objectif de Django est de proposer un développement plus efficace et plus rapide d'une application dynamique web tout en conservant la qualité.

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 des bases du HTML, pour comprendre les pages présentées dans ce cours.

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

À l'issue de ce cours, qui porte sur la version *à support long* (1.8) de Django, vous saurez construire des sites web complexes et élégants en un temps record.

Première partie

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.

1. 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 ?

1.1. 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 !

1.1.1. 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 !

1.1.2. 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.

1.2. 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 , Tornado ou encore Flask . 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.

FIGURE 1.1. – 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 !

FIGURE 1.2. – Logos d'Instagram, de la NASA et de Pinterest

1.2.1. 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.

1.2.2. 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 et une traduction en français est disponible. 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. Le site djangosnippets.org 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 !
Dans le même esprit, de nombreuses applications, prêtes à être utilisées, sont disponible via Django Packages notamment.

1.3. 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 3.4**. Il est possible d'utiliser Django avec Python 2.7 également, mais nous vous **recommandons fortement d'utiliser Python 3**.

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 1.8, soit d'exécuter vos deux projets avec deux versions de Django bien indépendantes.

Quoi qu'il arrive, si vous avez deux projets django, nous vous conseillons d'utiliser virtualenv pour éviter les conflits de dépendances.

1.3.1. Linux et Mac OS

Sous Linux et Mac OS, la méthode la plus simple et universelle consiste à installer Django depuis les répertoires de Pypi .

Pypi est l'index des paquets Python de référence. Il contient les paquets de dizaines de milliers de projets Python, prêt à être utilisé pour vos propres projets. Grâce à `pip`, il est possible d'installer ces paquets en une seule commande mais aussi gérer les dépendances de vos projets facilement.

Si vous n'avez jamais utilisé pip, il vous faut d'abord que vous l'installiez. Utilisez votre gestionnaire de paquets préféré et installez le paquet `python-pip`. Par exemple, sous Debian et Ubuntu :

```
1 apt-get install python-pip
```

Ensuite, vous n'avez plus qu'à installer le paquet Django, via pip :

```
1 pip install django==1.8
```

Nous spécifions ici que nous souhaitons la version 1.8 de Django et il va donc chercher la dernière version de la branche 1.8. L'index Pypi conserve toutes les versions de Django, et il est donc possible d'installer d'anciennes versions si vous le souhaitez.

1.3.2. 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:\Python34\;C:\Python34\Lib\site-packages\django`

5. Cherchez la variable système (deuxième liste) `Path` et ajoutez ceci en fin de ligne (faites attention à votre version de Python) : `;C:\Python34\;C:\Python34\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.in‘. Respectez bien le point-virgule permettant de séparer le répertoire de ceux déjà présents, comme indiqué à la figure suivante.

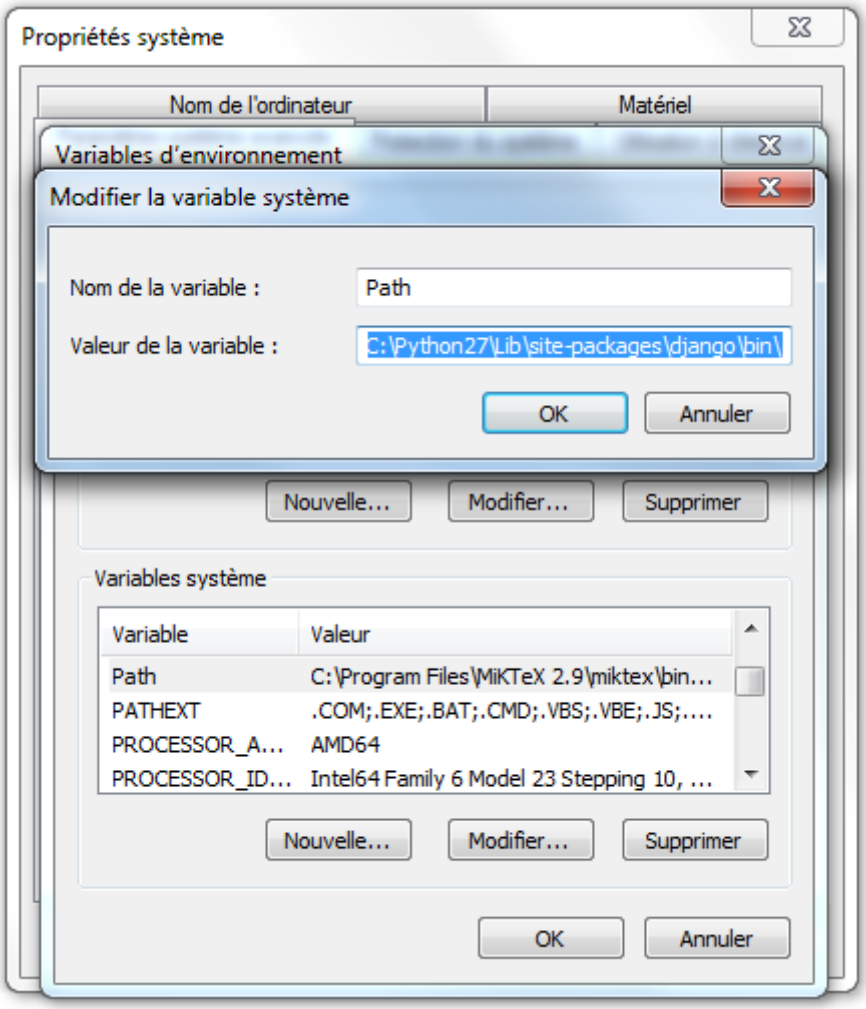


FIGURE 1.3. – Édition du Path sous Windows 7

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

```
cd C:\Users\<nom_d'utilisateur>\Downloads\django1.8  # A adapter à votre réper
python setup.py install
```

Les fichiers sont ensuite copiés dans votre dossier d'installation Python (ici `C:\Python34`).

1.3.3. 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 :

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

Si vous obtenez également 1.8 comme réponse, félicitations, vous avez correctement installé Django !

Il se peut que vous obteniez un numéro de version légèrement différent (du type 1.8.10). 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.).

1.4. 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 1.8, sortie en septembre 2014. Nous ne garantissons pas que les exemples donnés soient compatibles avec des versions antérieures et postérieures.

2. 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.

2.1. 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.

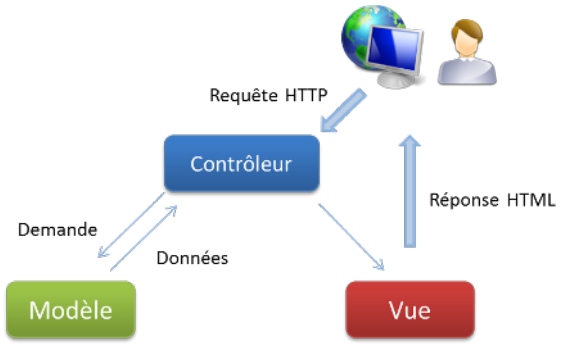


FIGURE 2.1. – Schéma de l'architecture MVC

2.2. 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.

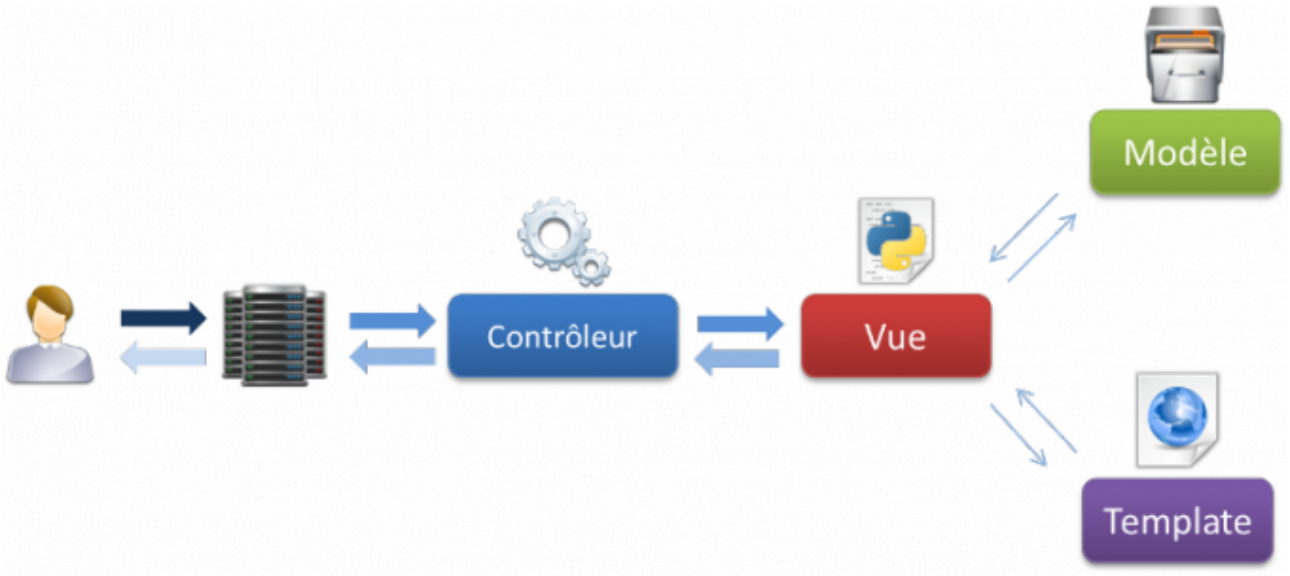


FIGURE 2.2. – Schéma d'exécution d'une requête

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.

2.3. 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.

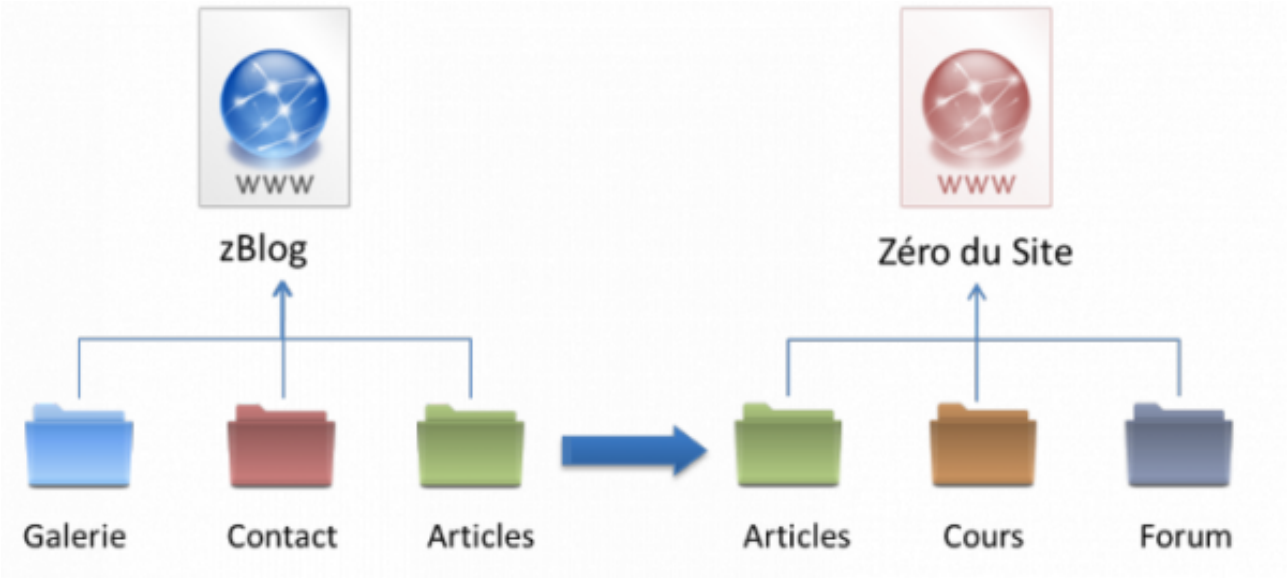


FIGURE 2.3. – Organisation d'un projet Django et réutilisation d'une application

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.

Si vous souhaitez ajouter un blog sur votre projet, vous pouvez même télécharger une application déjà existante (via `pip` par exemple), comme Django Blog Zinnia , et personnaliser les templates en les surchargeant. Nous verrons ce principe plus loin.Il existe des milliers d'applications open source que vous pouvez télécharger et intégrer à vos projets. N'hésitez pas à faire un tour sur Django Packages pour trouver votre bonheur.

2.4. 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.

3. 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.

3.1. 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 :

```
1 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 :

```
1 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 :

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

Un nouveau dossier nommé `crepes_bretonnes` est apparu et possède la structure suivante :

```
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`
— un autre sous-dossier nommé également `crepes_bretonnes`.

Créez dans le dossier principal un dossier nommé `templates`, lequel contiendra vos templates HTML.

Sachez toutefois qu'il est possible d'intégrer un dossier `templates` **au sein de chaque application** : ceci permet une meilleure modularité et un partage d'applications entre projets plus facile. En théorie, il est recommandé d'agir tel quel, afin d'avoir tout le code concernant une application au même endroit. En effet, Django cherche en priorité le template demandé dans le dossier `templates` des applications installées, puis après dans ceux listés dans une variable du fichier de configuration, `TEMPLATE_DIRS`, que l'on va voir juste après.. De plus pour éviter les collisions, il est conseillé de garder la structure suivante : `<APP>/templates/<APP>/mon_template.html`, où `<APP>` est le nom de votre application. Le dossier `templates` à la racine de votre projet **ne contiendra que les templates spécifique à votre projet**, qui ne peuvent être rangés autre part. Dans ce cours nous allons, par simplicité pédagogique, tout placer dans le dossier `templates` à la racine. Pensez cependant à prendre la bonne habitude de bien **séparer vos dossiers `templates`** dans de vrais projets ! La correction du TP disponible à la fin de la partie 2 vous montrera un exemple d'application avec un dossier templates indépendant.

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` :

```
$ 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 :

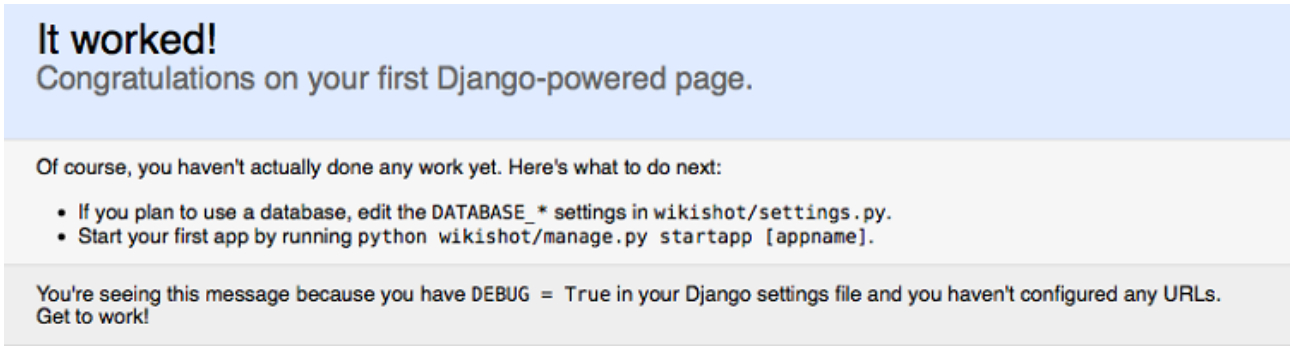


FIGURE 3.1. – Votre première page Django

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` :

```
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.

3.2. 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.

Pour commencer, nous avons tout en haut la variable `BASE_DIR`, définie automatiquement selon le chemin du fichier courant (qui est `settings.py` pour rappel) :

```
1 import os
2 BASE_DIR = os.path.dirname(os.path.dirname(__file__))
```

La fonction `dirname` est appelée deux fois, afin de remonter d'un dossier et ainsi avoir le chemin vers la racine du projet. La variable `BASE_DIR` peut être ainsi utilisée pour référencer des chemins vers des fichiers au sein du projet (ressources CSS, Javascript, fichiers de test, base de données SQLite...)

Ensuite, nous allons expliquer les variables les plus importantes. Tout d'abord nous avons les variables de debug :

```
1 # /!\ n'utilisez pas ça sur votre hébergement final, cela peut
2 # causer des problèmes de sécurité
3 DEBUG = True
4 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, comme le commentaire le précise !

Nous avons ensuite la variable `ALLOWED_HOSTS` que nous allons laisser de côté pour le moment. Elle permet de renseigner les noms de domaine et IP par lesquels le projet peut être vu lorsque le mode `DEBUG` est désactivé. Nous allons également laisser de côte plusieurs variables, que l'on aura l'occasion de recroiser par la suite.

La configuration de la base de données se fait dans le dictionnaire `DATABASES`, déjà rempli pour gérer une base de données *SQLite* (notez l'utilisation de `BASE_DIR` par ailleurs).

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

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

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',  # Backends disponibles : 'postgresql_psycopg2', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': 'crepes_bretonnes',  # Nom de la base de données
        'USER': '<user>',            # Utilisateur
        'PASSWORD': '<pswd>',        # Mot de passe si nécessaire
        'HOST': '127.0.0.1',         # Utile si votre base de données est sur une autre machine
        'PORT': '',                  # ... et si elle utilise un autre port que celui par défaut
    }
}
```

Listing 1 – Configuration des connecteurs de base de données

Juste après la base de données, nous avons quelques variables pour définir la langue et le fuseau horaire de votre projet :

```
# Langage utilisé au sein de Django, pour afficher les messages
# d'information et d'erreurs notamment
LANGUAGE_CODE = 'fr-FR'
# Fuseau horaire, pour l'enregistrement et l'affichage des dates
# Laissez UTC dans le cas de l'Europe de l'Ouest, c'est notre fuseau ;)
TIME_ZONE = 'UTC'
```

Listing 2 – Configuration des langues et fuseaux horaires

Il y a ensuite les variables `USE_I18N` , `USE_L10N` et `USE_TZ` qui sont toutes à `True`. Elles permettent entre autres d'activer l'internationalisation[1] et ainsi avoir l'interface en français, la répresentation des jours au format `dd/mm/YYYY` et d'autres détails.

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.

3.3. Créons notre première application

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

1. l'*internationalisation* est abregée *i18n* car il y a 18 lettres entre le i et le dernier n

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

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

```
crepes_bretonnes/
    blog/
        __init__.py
        admin.py
        migrations/
            __init__.py
        models.py
        tests.py
        views.py
    crepes_bretonnes/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    db.sqlite3
    manage.py
```

Listing 3 – Architecture du projet

Si vous avez suivi, certains noms de fichiers sont relativement évidents. Nous allons les voir un à un dans les prochains chapitres :

- `admin.py` va permettre de définir ce que souhaitez afficher et modifier dans l'administration de l'application ;
- `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.

Le dossier migrations nous servira plus tard, quand nous parlerons des modèles. Il permet de retracer l'évolution de vos modèles dans le temps et d'appliquer les modifications à votre base de données. Nous verrons ça plus en détails également, laissons-le de côté pour le moment.

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, que l'on a ignoré plus tôt. Tout en conservant les autres applications installées, ajoutez une chaîne de caractères avec le nom de votre application. Votre variable devrait ressembler à quelque chose comme ceci :

```
INSTALLED_APPS = (
    'django.contrib.admin',
```

```
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
)
```

Listing 4 – la liste des applications utilisées

Nous sommes désormais prêts pour attaquer le développement de notre application de blog !

3.4. 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 start project 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.

4. 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.

4.1. 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.

4.2. Le langage SQL et les gestionnaires de base de données

Il existe plusieurs programmes qui s'occupent de gérer des bases de données. Nous les appelons, tout naturellement, des gestionnaires de bases de données (ou « SGBD » pour « systèmes de gestion de bases de données »). Ces derniers s'occupent de tout : création de nouvelles tables, ajout de nouvelles entrées dans une table, mise à jour des données, renvoi des entrées déjà enregistrées, etc. Il y a énormément de SGBD, chacun avec des caractéristiques particulières. Néanmoins, ils se divisent en deux grandes catégories : les bases de données SQL et les bases de données non-SQL. Nous allons nous intéresser à la première catégorie (celle que Django utilise majoritairement).
Les gestionnaires de bases de données SQL sont les plus populaires et les plus utilisés pour le moment. Ceux-ci reprennent l'utilisation du classement par tableau tel que nous l'avons vu. L'acronyme « SQL » signifie « Structured Query Language », ou en français « langage de requêtes structurées ». En effet, lorsque vous souhaitez demander au SGBD toutes les entrées d'une table, vous devez communiquer avec le serveur (le programme qui sert les données) dans un langage qu'il comprend. Ainsi, si pour commander un café vous devez parler en français, pour demander les données au gestionnaire vous devez parler en SQL.
Voici 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 :

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

Listing 5 – une requete SQL pour obtenir les films de Q.Tarantino

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.

4.3. 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 :

```
1 class Article(models.Model):
2     titre = models.CharField(max_length=100)
3     auteur = models.CharField(max_length=42)
4     contenu = models.TextField(null=True)
5     date = models.DateTimeField(auto_now_add=True, auto_now=False,
6                                 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…

```
1 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…

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

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

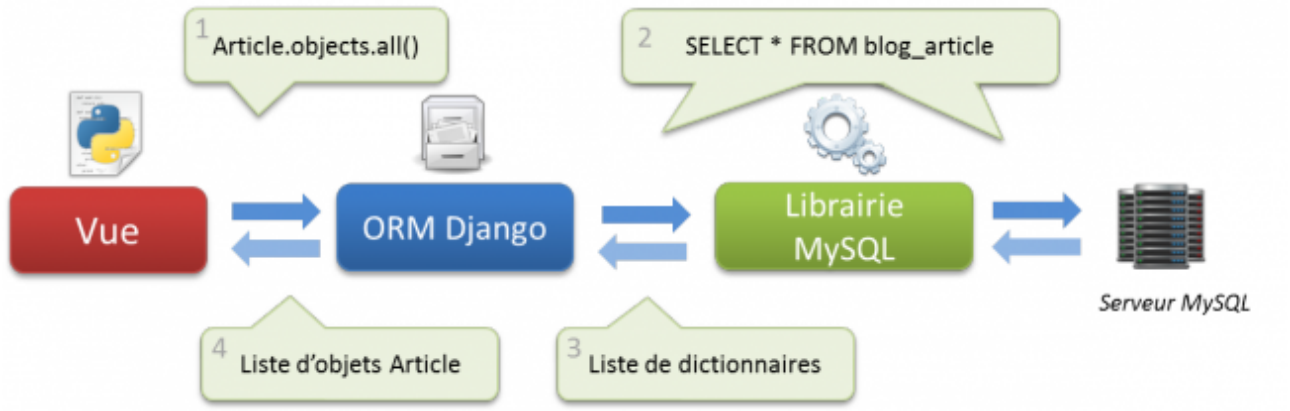


FIGURE 4.1. – Fonctionnement de l'ORM de Django

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.

4.4. 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.

4.5. 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.

Deuxième partie

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 !

5. 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 !

5.1. Hello World !

Commençons enfin notre blog sur les bonnes crêpes bretonnes. 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, prenant en paramètre une requête HTTP et renvoyant une réponse HTTP. 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 :

```
1 from django.shortcuts import render
2 # Create your views here.
```

Si vous utilisez encore Python 2, 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 : `#-*- 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 ! Ceci ne concerne que Python 2 et est facultatif pour Python 3, qui utilise l'UTF-8 de base.

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. On va pour le moment laisser de côté la méthode render déjà importé et utiliser HttpResponse , pour comprendre la base :

```
1  from django.http import HttpResponse
2  from django.shortcuts import render
3
4  def home(request):
5
           """ Exemple de page HTML, non valide pour que l'exemple soit concis """
6      text = """<h1>Bienvenue sur mon blog !</h1>
7                 <p>Les crêpes bretonnes ça tue des mouettes en plein vol !</p>"""
8      return HttpResponse(text)
```

Si vous utilisez Python 2, vous pourriez avoir besoin de préfixer les chaines contenant des accents avec u : text = u"""..""" pour forcer la chaine en Unicode.
Ce code se divise en trois parties :

1. 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.
2. Une fonction home, avec comme argument une instance de `HttpRequest`. Nous avons nommé ici (et c'est partout le cas par convention) sobrement cet argument request. Celui-ci contient des informations sur la méthode de la requête (`GET`, `POST`), les données des formulaires, la session du client, etc. Nous y reviendrons plus tard.
3. 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.

5.2. 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 :

```python
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    # Examples:
    # url(r'^$', 'crepes_bretonnes.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),
    url(r'^admin/', include(admin.site.urls)),
]
```

Listing 6 – Définition des routes

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.

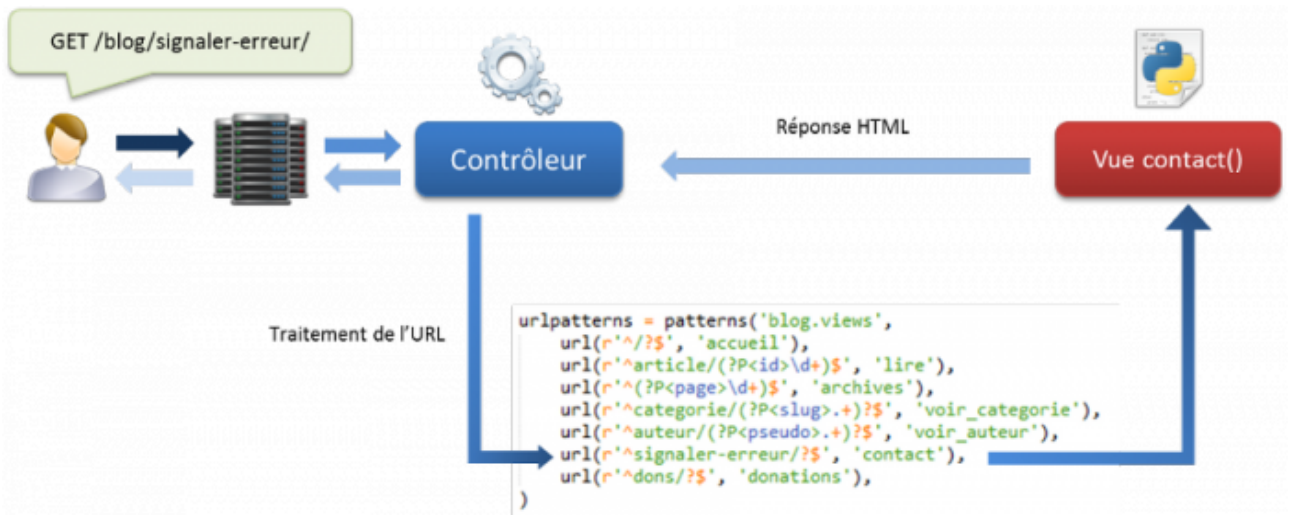


FIGURE 5.1. – Schéma d'exécution d'une requête (nous travaillons pour le moment sans templates et sans modèles)

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` :

```
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 routes en fonction de "gabarits" ou de "motifs" d'URL. 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 ».

Le second argument doit être la vue, qui est la fonction home que l'on créé dans le fichier `blog/views.py` . On y accède donc via le module views importé juste au-dessus. Il est également possible d'utiliser directement la chaine de caractère `'blog.views.home'` pour s'affranchir de l'import, mais en cas d'erreur les exceptions seront plus difficiles à comprendre.
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.

FIGURE 5.2. – L'affichage de votre première vue

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

5.3. 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.

5.3.1. 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 !) :

```
1 from django.conf.urls import url
2 from . import views
3
4 urlpatterns = [
```

```
5       url(r'^accueil$', views.home),
6   ]
```

Listing 7 – Définition d'un motif d'url

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` :

```
1   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 :

```
1   urlpatterns = patterns('',
2       url(r'^accueil/$', 'blog.views.home'),
3       url(r'^truc/$', 'blog.views.truc'),
4       url(r'^chose/$', 'blog.views.chose'),
5       url(r'^machin/$', 'blog.views.machin'),
6       url(r'^foo/$', 'blog.views.foo'),
7       url(r'^bar/$', 'blog.views.bar'),
8   )
```

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 :

```
1   urlpatterns = patterns('blog.views',
2       url(r'^accueil/$', 'home'),
3       url(r'^truc/$', 'truc'),
4       url(r'^chose/$', 'chose'),
5       url(r'^machin/$', '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 :

```
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.

5.4. 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://instagram.com/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 faut 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` :

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

Listing 8 – Des urls avec des paramètres

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 :

```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 #{0} !".format(id_article)
    return HttpResponse(text)
```

Listing 9 – Définiition de la vue avec un paramètre

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 septembre 2014 : `/blog/articles/2014/09`.
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/09/2014`. 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 :

```python
urlpatterns = [
    url(r'^home$', views.home),
    url(r'^article/(?P<id_article>\d+)$', views.view_article),
    url(r'^articles/(?P<year>\d{4})/(?P<month>\d{2})$',
        views.list_articles),
]
```

Listing 10 – Utilisaion de paramètres nommés

Et la vue correspondante :

```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 :
`list_articles(request, year=2014, month=9)`
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, via le dictionnaire `request.GET` dans la vue. Ici, `request.GET.get('ref', None)` retournerait 'twitter' .

5.5. 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.

5.5.1. 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 :

```
1 from django.http import HttpResponse, Http404
2
3 def view_article(request, id_article):
4     if int(id_article) > 100: #Si l'ID est supérieur à 100, nous
          considérons que l'article n'existe pas
5         raise Http404
6
7     return HttpResponse('<h1>Mon article ici</h1>')
```

Listing 11 – Une page 404 quand on ne trouve pas l'article

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.

FIGURE 5.3. – Erreur 404, page introuvable

5.5.2. Rediriger l'utilisateur

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

```
1 from django.shortcuts import redirect
2
3 def list_articles(request, year, month):
4     # Il veut des articles ? Soyons fourbe et redirigeons le vers
          djangoproject.com
5     return redirect("https://www.djangoproject.com")
```

Listing 12 – Une redirection vers djangoproject

N'oubliez pas qu'une URL valide pour accéder à cette vue serait `/blog/articles/2014/09`.
Cependant, si vous souhaitez rediriger votre visiteur vers une autre page de votre site web, il est plus intéressant de privilégier l'autre méthode, qui permet de garder indépendante la configuration des URL et des vues. Nous devons donc passer en argument le nom de la vue vers laquelle nous voulons rediriger l'utilisateur, avec éventuellement des arguments destinés à celle-ci.

```
1 from django.http import HttpResponse, Http404
2 from django.shortcuts import redirect
3
4 def view_article(request, id_article):
5     if int(id_article) > 100:
```

```
6           raise Http404
7
8       return redirect(view_redirection)
9
10  def view_redirection(request):
11      return HttpResponse("Vous avez été redirigé.")
12
13  url(r'^redirection$', views.view_redirection),
```

Ici, si l'utilisateur accède à l'URL `/blog/article/101`, il aura toujours une page 404. Par contre, s'il choisit un ID inférieur à 100, alors il sera redirigé vers la seconde vue, qui affiche un simple message.
Il est également possible de préciser si la redirection est temporaire ou définitive en ajoutant le paramètre permanent=True. L'utilisateur ne verra aucune différence, mais ce sont des détails que les moteurs de recherche prennent en compte lors du référencement de votre site web.
Si nous souhaitions rediriger un visiteur vers la vue view_article définie précédemment par un ID d'article spécifique, il suffirait simplement d'utiliser la méthode redirect ainsi :

```
1  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 ?

En fait, il est possible d'indiquer une vue de trois manières différentes :
- En passant directement la fonction Python, comme nous l'avons vu au début ;
- En donnant le chemin vers la fonction, dans une chaîne de caractères (ce qui évite de l'importer si elle se situe dans un autre fichier) ;
- En indiquant le nom de la vue tel qu'indiqué dans un urls.py (voir l'exemple 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 :

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

Listing 13 – une vue nommée

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 :

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

Listing 14 – Redirection vers une vue nommée

Pour terminer, sachez qu'il existe également une fonction qui permet de générer simplement l'URL et s'utilise de la même façon que redirect ; il s'agit de reverse (`django.core.urlresolvers.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.

5.6. 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.

6. 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.

6.1. 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.

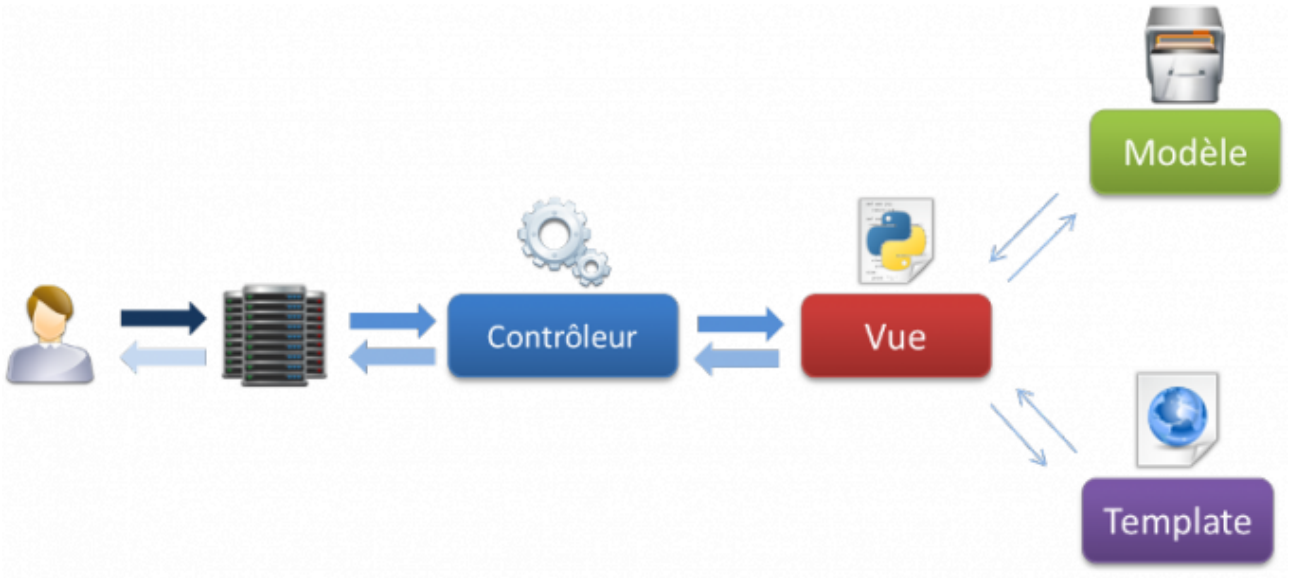


FIGURE 6.1. – Schéma d'exécution d'une requête

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](https ://docs.djangoproject.com/en/stable/ref/templates/api/#using-the-template-syst | 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 :

```
1  from datetime import datetime
2  from django.shortcuts import render
3
4  def date_actuelle(request):
5      return render(request, 'blog/date.html', {'date':
           datetime.now()})
6
7
8  def addition(request, nombre1, nombre2):
9      total = int(nombre1) + int(nombre2)
10     # Retourne nombre1, nombre2 et la somme des deux au tpl
11
12     return render(request, 'blog/addition.html', locals())
```

Listing 15 – blog/views.py

```
1  from django.conf.urls import url
2  from . import views
3
4  urlspatterns = [
5      url(r'^date$', views.date_actuelle),
6      url(r'^addition/(?P<nombre1>\d+)/(?P<nombre2>\d+)/$',
           views.addition)
7  ]
```

Listing 16 – Extrait de blog/urls.py

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.

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

— Dans la liste des dossiers fournis dans la variable de configuration TEMPLATES_DIR ;
— S'il ne l'a pas trouvé, dans le dossier templates de l'application en cours.

Nous n'avons pas parlé de `TEMPLATES_DIR` lors de la configuration de notre projet, car cette variable n'est pas présente par défaut dans le fichier `settings.py`. En réalité, il vous est possible de définir une liste de dossiers où Django ira chercher les templates en priorité. Classiquement, cette liste est composée d'un dossier templates à la racine de votre projet :

```
TEMPLATE_DIRS = (
    os.path.join(BASE_DIR, 'templates'),
)
```

Nous vous conseillons de créer un dossier templates à la racine du projet. Vous pourrez y déposer des templates plutôt propres à votre projet (erreurs 404, squelette de votre design, pages statiques...).
Pour nos applications, nous allons utiliser la deuxième catégorie : le dossier templates de l'application actuelle. En effet, il est préférable de conserver les templates propres à une application dans son dossier, afin de permettre la réusabilité de l'application. Si vous souhaitez partager votre application, tout marchera sans devoir déplacer les templates en fonction de l'installation.
Enfin, pour éviter les conflits, l'usage est de créer un dossier du nom de l'application au sein du dossier templates. On obtient alors la hiérarchie suivante :

```
crepes_bretonnes/
    blog/
        __init__.py
        admin.py
        migrations/
            __init__.py
        models.py
        templates/
            blog/
                addition.html
                date.html
        tests.py
        views.py
    crepes_bretonnes/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    templates/
    db.sqlite3
```

```
21      manage.py
```

Listing 17 – L'arborescence du projet

L'avantage de cette structure est que les personnes qui utiliseraient votre application pourrait surcharger votre template en écrivant un nouveau dans leur propre dossier `templates/` global ! Maintenant que nous avons notre structure, créeons nos fichiers templates. Occupons-nous d'abord de `date.html`, le plus simple. Créez ce nouveau fichier dans `blog/templates/blog/` :

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

Nous retrouvons `date`, comme passé dans `render()`. Si vous accédez à cette page (après lui avoir assigné une URL), le `{{ date }}` est bel et bien remplacé par la date actuelle ! Dans ce fichier, nous avons accès qu'à cette variable, puisque c'est la seule que nous avons passé au template via la fonction render .

i Si vous obtenez l'exception `TemplateDoesNotExist`, spécifiant que `blog/date.html` n'existe pas, vérifiez que l'application blog est bien dans `INSTALLED_APPS`, dans le fichier `settings.py`. Django ne va chercher les templates que dans les applications installées !

On peut rapidement observer le second exemple avec le template `addition.html` :

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

Listing 18 – Faire une addition

Nous expliquerons bientôt les structures présentes dans ce template, ne vous inquiétez pas. Cependant, en allant sur `http://127.0.0.1:8000/blog/addition/5/3` , vous obtiendrez le résultat de l'addition `5 + 3`.

FIGURE 6.2. – Résultat de notre petite calculatrice

Encore une fois, un piège peut se glisser ici. Du fait des accents, faites attention à ce que votre fichier de templates soit encodé en UTF8, sinon vous pouvez obtenir l'erreur UnicodeDecodeError.
Ceci est dû à la présence d'accent dans le template, cher à notre langue française.

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. Le dictionnaire pour notre seconde vue est semblable à ceci :

```
1 {"total": 8, "nombre1": 5, "nombre2": 3, "request": <WSGIRequest
      ...>}
```

La fonction `locals()` inclut également request, puisque c'est un argument de notre vue et donc une variable locale de la fonction.

Nous utilisons ici la fonction `locals` car nous avons des vues très courtes et pour que vous puissiez vous concentrer sur l'essentiel. Dans un projet plus conséquent, comme Zeste de Savoir par exemple, il vaut mieux utiliser un dictionnaire. Cela rend le code plus lisible et cela évite les effets de bord.

6.2. Affichons nos variables à l'utilisateur

6.2.1. 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.

```
1 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 « Clem » et `date` « 28 décembre », le moteur de templates affichera « Bonjour Clem, 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 :

```
{# 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.

6.2.2. 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 :

```
{{ 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` :

```
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 :

```
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 !

```
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.

6.3. 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.

6.3.1. 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 :

```
Bonjour
{% if sexe == "Femme" %}
    Madame
{% else %}
    Monsieur
{% endif %} !
```

Listing 19 – Une condition dans le template

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 :

```
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 :

```
{% 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 %}
```

6.3.2. 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 :

```
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 :

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

Listing 20 – Afficher une liste.

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 :

```
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 %}` :

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

```
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 :

```
<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.

6.3.3. 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 :

```
<!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>
```

Listing 21 – Notre template de base.

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 :

```
{% 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 %}
```

Listing 22 – Une page qui utilise le template de base.

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 :

FIGURE 6.3. – Fonctionnement du tag {% block %}

6.3.4. 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 :

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

… générera le code HTML suivant :

```
<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 :

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

6.3.5. 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 #}`.

```
<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 %}`.

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

6.4. 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.
Comme pour les templates, les fichiers statiques peuvent se trouver à deux endroits : au sein du dossier static de l'application courante ou alors dans un ou plusieurs dossiers définis dans la configuration du projet.
Tout d'abord, créons un dossier à la racine du projet, dans lequel vous enregistrerez les fichiers liés globalement au projet : CSS et Javascript global, images du design... Nous l'appellerons `static`.
Il faut ensuite renseigner ce dossier et lui assigner un préfixe d'URL dans votre `settings.py`.
Voilà les deux variables qu'il faudra modifier, ici selon notre exemple :

```
STATIC_URL = '/static/'   # Qui devrait déjà être la configuration
    par défaut

STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

Listing 23 – Configuration des urls statiques.

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. En plus de ce dossier, tous les dossiers static de toutes les applications dans INSTALLED_APPS pourront être utilisés.
De la même façon, on tiendra à respecter la convention du dossier du nom de l'application au sein du dossier static, afin d'éviter les conflits. On obtient la structure de projet suivante (nous avons omis volontairement les fichiers Python de l'application blog) :

```
crepes_bretonnes/
    blog/
        static/
            blog/
                crepes.jpg
        templates/
            blog/
                addition.html
                date.html
    crepes_bretonnes/
        __init__.py
        settings.py
        urls.py
        wsgi.py
    static/
        css/
        img/
            header.png
        js/
```

```
    templates/
        base.html
```

Listing 24 – Notre architecture.

Vous pouvez ainsi inclure l'image de crêpes de l'application blog dans votre template de la façon suivante

```
{% 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 `static.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, le serveur Django vous permet d'utiliser les fichiers statiques tout de même. Cette méthode n'est pas considéré comme efficace et sécurisée donc elle ne doit pas être utilisée en production ! Pour le déploiement des fichiers statiques en production, référez-vous à l'annexe consacrée à ce sujet.

6.5. 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 %}`.

7. 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.

7.1. 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 :

```
1  from django.db import models
2
3  class Article(models.Model):
4      titre = models.CharField(max_length=100)
5      auteur = models.CharField(max_length=42)
6      contenu = models.TextField(null=True)
7      date = models.DateTimeField(auto_now_add=True, auto_now=False,
8                                  verbose_name="Date de parution")
9
10     def __str__(self):
11         """
12         Cette méthode que nous définirons dans tous les modèles
13         nous permettra de reconnaître facilement les différents objets que
14         nous traiterons plus tard et dans l'administration
15         """
16         return self.titre
```

Listing 25 – Le modèle représentant un article.

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 associera automatiquement à l'objet la date courante lorsque ce dernier sera créé pour la première fois. 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 : `IntegerField`, `BooleanField`... 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 faut lancer la commande `makemigrations` via l'utilitaire `manage.py`. Cette commande va déterminer quelles modifications ont été apportées aux modèles et va détecter quels changements il faut opérer en conséquence sur la structure de la base de données. Ensuite, il faut utiliser la commande `migrate` qui va réaliser ces changements dans la base de données. Pour rajouter votre nouveau modèle, il faut donc lancer :

```
python manage.py makemigrations
python manage.py migrate
```

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.

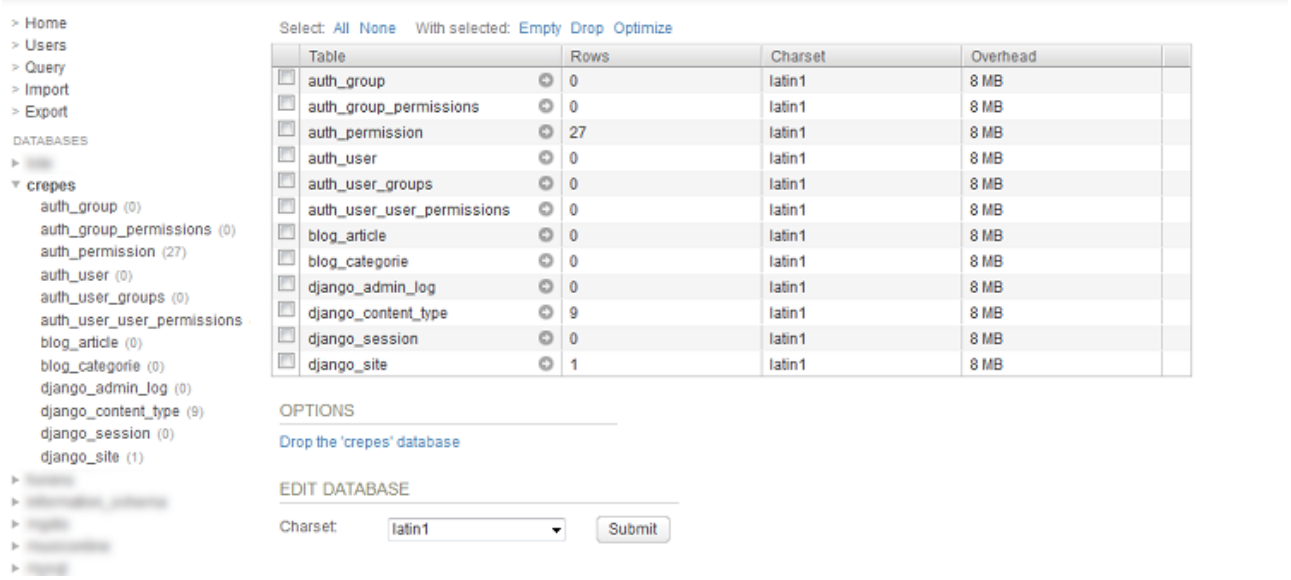


FIGURE 7.1. – Aperçu des tables créées dans un outil de gestion de base de données

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

7.2. 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` :

```
$ python manage.py shell
Python 3.4.0 (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 :

```
>>> 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 :

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

? 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 :

```
>>> 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 :

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

L'entrée a été créée et enregistrée !
Bien évidemment, il est toujours possible de modifier l'objet par la suite :

```
>>> 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 :

```
>>> 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()` :

```
>>> 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 :

```
>>> 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` :

```
>>> 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 :

```
>>> 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 :

```
>>> 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 :

```
>>> 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 :

```
>>> 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) :

```
>>> 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 :

```
>>> 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 « - » :

```
>>> 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 :

```
>>> Article.objects.filter(date__lt=datetime.now()).order_by('date','titre').re
[<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 :

```
>>> 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 :

```
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)
```

7.3. 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.

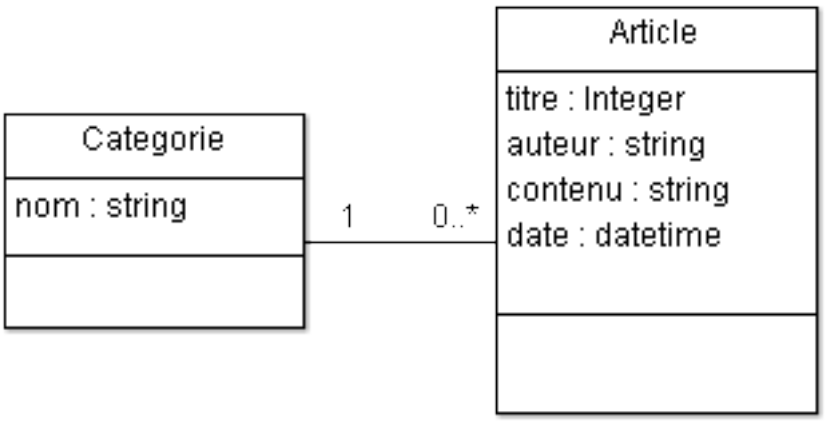


FIGURE 7.2. – Ici, un article peut être lié à une et une seule catégorie, et une catégorie peut être attributée à une infinité d'articles

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

```
1 class Categorie(models.Model):
2     nom = models.CharField(max_length=30)
3
4     def __str__(self):
5         return self.nom
```

Listing 26 – Notre nouvelle classe : `Categorie`.

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

```
1 class Article(models.Model):
2     titre = models.CharField(max_length=100)
3     auteur = models.CharField(max_length=42)
4     contenu = models.TextField(null=True)
5     date = models.DateTimeField(auto_now_add=True, auto_now=False,
          verbose_name="Date de parution")
6     categorie = models.ForeignKey('Categorie')
7
8     def __str__(self):
9         return self.titre
```

Listing 27 – Ajout de la référence à `Categorie`.

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.
Utilisez `makemigrations` et `migrate` comme décrits précédemment pour mettre à jour votre base de données.

Si vous avez déjà des articles présents dans votre base de données, Django va vous demander que faire pour ces lignes déjà existantes.
On va supposer que chaque article doit avoir une catégorie et on va donc attribuer, à la volée, une catégorie par défaut pour les articles déjà créés.
Vous pouvez donc faire l'option "1" et ensuite entrez la valeur 1, pour donner la première catégorie à ces articles.

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

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

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

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

Listing 28 – Lier une article à sa catégorie.

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

```
>>> 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`.

```
>>> 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 :

```
>>> Article.objects.filter(categorie__nom__contains="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 :

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

    def __str__(self):
        return self.nom

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

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

Listing 29 – Par exemple : une voiture a un moteur et un moteur ne peut se trouver dans deux voitures différentes.

N'oubliez pas de mettre à jour votre base de données.
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` :

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

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

Ici, le `OneToOneField` a créé une relation en sens inverse qui ne va plus renvoyer un `QuerySet`,

mais directement l'élément concerné (ce qui est logique, celui-ci étant unique). Cette relation inverse prendra simplement le nom du modèle, 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 :

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

    def __str__(self):
        return self.nom

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

    def __str__(self):
        return self.nom

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

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

Listing 30 – Un vendeur peut vendre plusieurs produits et chaque produit est vendu par plusieurs vendeurs.

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.
Mettez à jour la base de données (`makemigrations` puis `migrate`) et lancez un shell. Enregistrons un vendeur et deux produits :

```
>>> 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 :

```
>>> 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 :

```
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` :

```
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 `ManyToMany Field`, ici `Vendeur`) associés au modèle destinataire (ici `Produit`), rien de plus simple, nous obtenons à nouveau un `QuerySet` :

```
>>> 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 :

```
>>> 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 :

```
>>> 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` :

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

Et tout a disparu !

7.4. 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.

7.4.1. 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 sqlmigrate blog 0001_initial`, qui renvoie la structure SQL des tables créées :

```
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;
```

Listing 31 – Le SQL qui permet de créer nos tables.

Nous n'avons sélectionné ici que les modèles `Categorie` et `Article`. Le `0001_initial` correspond à la première migration créée pour cette application.

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 :

```
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.
```

Listing 32 – blog/views.py

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

Listing 33 – Extrait de blog/urls.py

```
<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>
```

```
8       </div>
9   {% empty %}
10      <p>Aucun article.</p>
11  {% endfor %}
```

Listing 34 – templates/blog/accueil.html

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.

7.4.2. 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 :

```
1  def lire(request, id):
2      try:
3          article = Article.objects.get(id=id)
4      except Article.DoesNotExist:
5          raise Http404
6
7      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`.

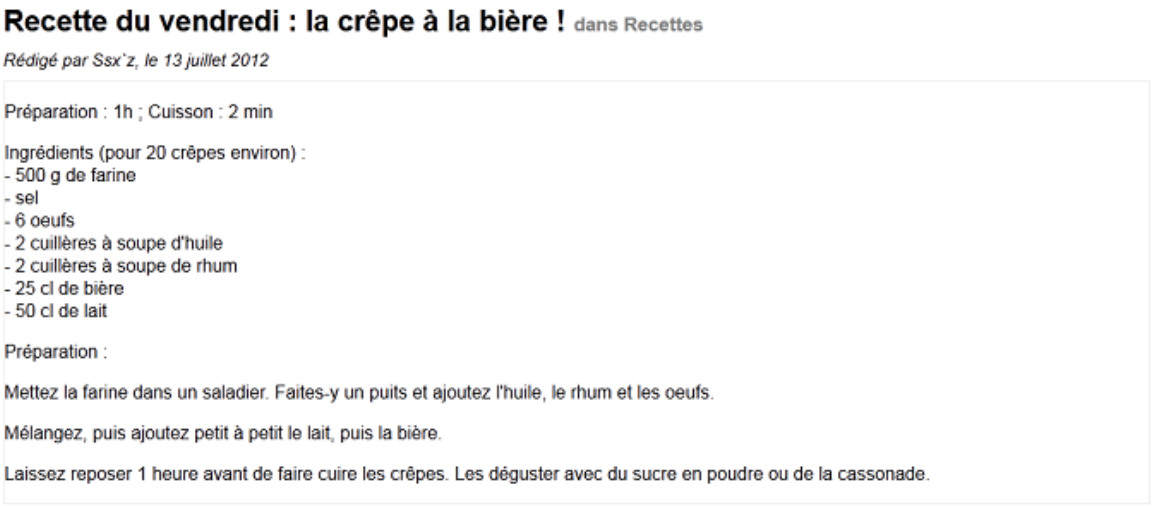
```
1  # Il faut ajouter l'import get_object_or_404, attention !
2  from django.shortcuts import render, get_object_or_404
3
4  def lire(request, id):
5      article = get_object_or_404(Article, id=id)
6      return render(request, 'blog/lire.html', {'article': article})
```

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

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

```
3 <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.

FIGURE 7.3. – 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

7.4.2.1. 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 :

> 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.
>
> *Wikipédia - Slug (journalisme)*

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 :

```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 __str__(self):
        return self.titre
```

Listing 35 – Ajout du **slug** à notre article.

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 :

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

Listing 36 – Extrait de blog/urls.py

```python
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})
```

Listing 37 – Extrait de blog/views.py

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

Listing 38 – templates/blog/accueil.html

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.

7.5. 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.

8. 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.

8.1. Mise en place de l'administration

8.1.1. 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.

8.1.2. Accédons à cette administration !

Ce module est optionnel mais il est tellement utilisé que la configuration par défaut l'intègre désormais. Cependant, pour ceux qui auraient modifié la configuration, voici un petit rappel.

8.1.2.1. Import des modules

Tout d'abord, ayez dans votre liste `INSTALLED_APPS` les applications suivantes, déjà présent au début de la liste par défaut :

```
1 'django.contrib.admin',
2 'django.contrib.auth',
3 'django.contrib.contenttypes',
4 'django.contrib.sessions',
```

Listing 39 – Les INSTALLED_APPS par défaut

Ensuite, le module d'administration nécessite aussi l'import de middlewares, normalement inclus par défaut également :

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

Listing 40 – Les middlewares

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

8.1.2.2. Mise à jour de la base de données

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. Si vous ne l'avez pas déjà fait au moins une fois, il faut exécuter python manage.py migrate .Ensuite, il nous faut créer un compte super-utilisateur. Il sera au début le seul compte à pouvoir accéder à l'administration.
Pour ce faire, vous devez taper la commande suivante :

```
python manage.py createsuperuser
```

Insérez les informations utilisateur que Django vous demande (nom d'utilisateur, e-mail, mot de passe).

8.1.2.3. 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 une ligne pour l'administration, mais celles-ci sont commentées. Votre fichier devrait ressembler à ceci :

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    # D'autres éventuelles directives.
    # Celles de notre application blog notamment
    # ...
    url(r'^admin/', include(admin.site.urls)),
]
```

Nous voyons que par défaut, l'administration sera disponible à l'adresse `http://localhost:8000/admin/`. Nous pouvons donc tester en lancant le serveur Django. 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 via la commande `createsuperuser`.

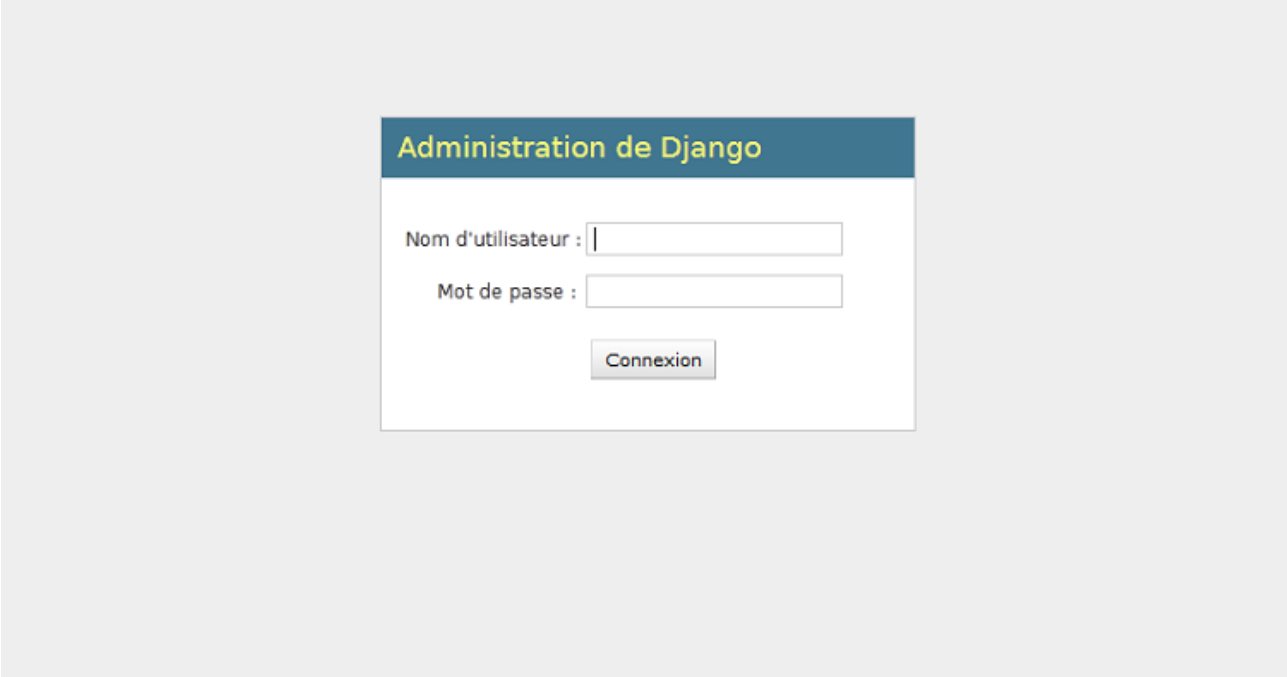


FIGURE 8.1. – L'écran de connexion de l'administration

8.2. 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.

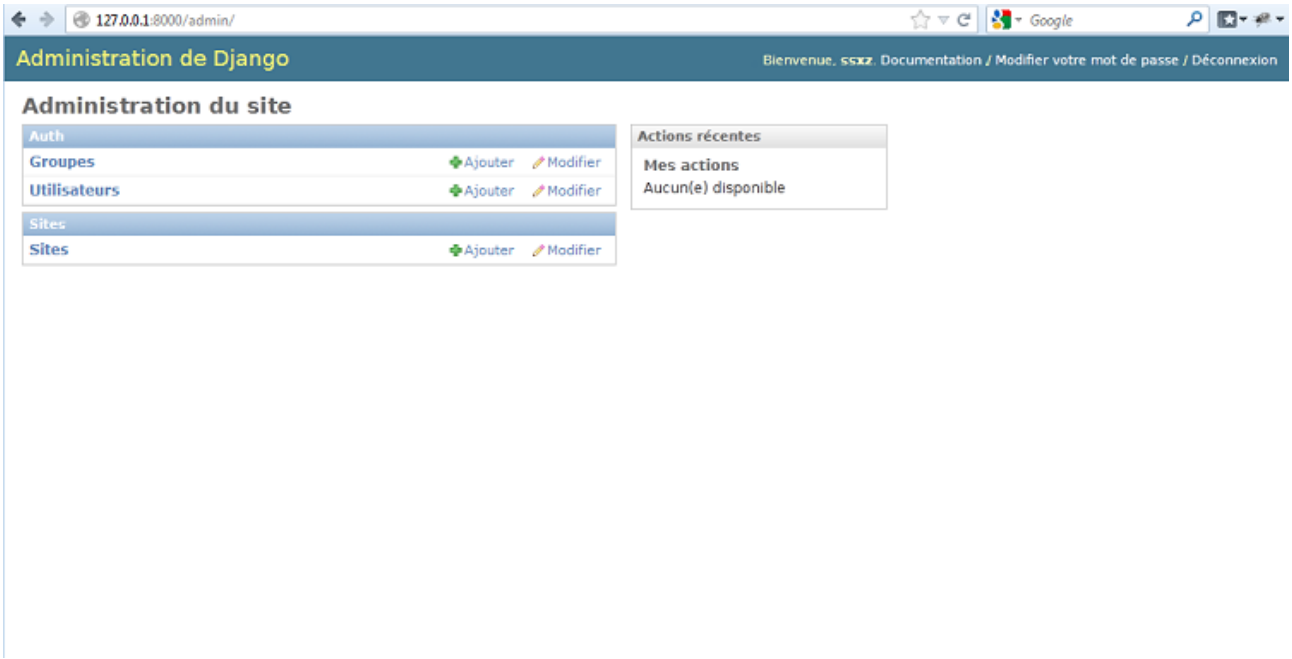


FIGURE 8.2. – Accueil de l'administration

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.

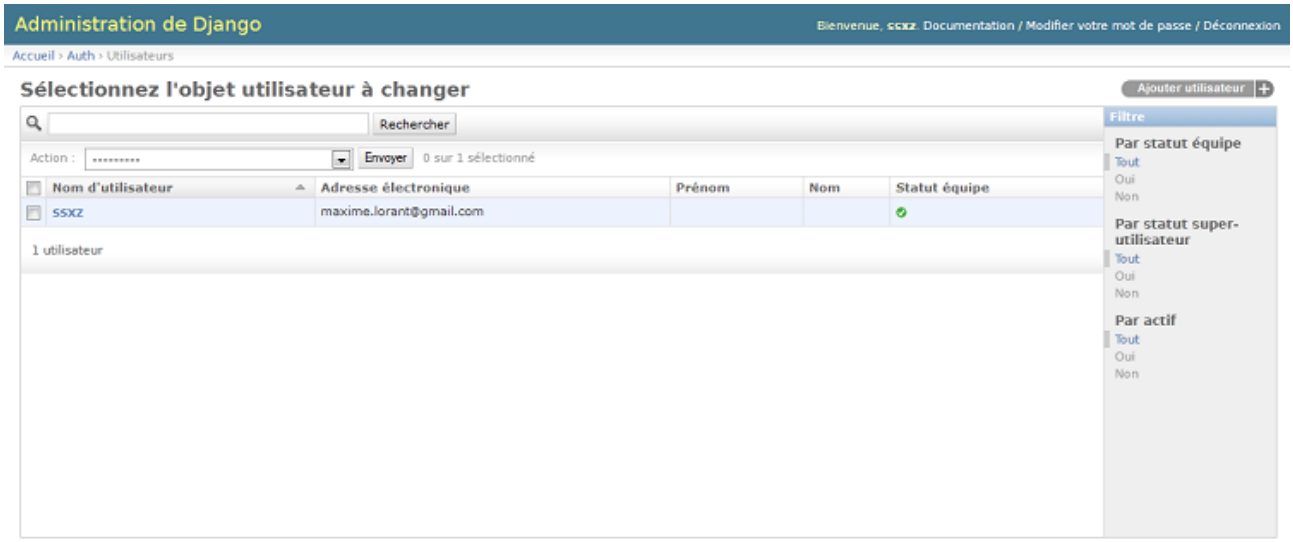


FIGURE 8.3. – Liste des utilisateurs

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.

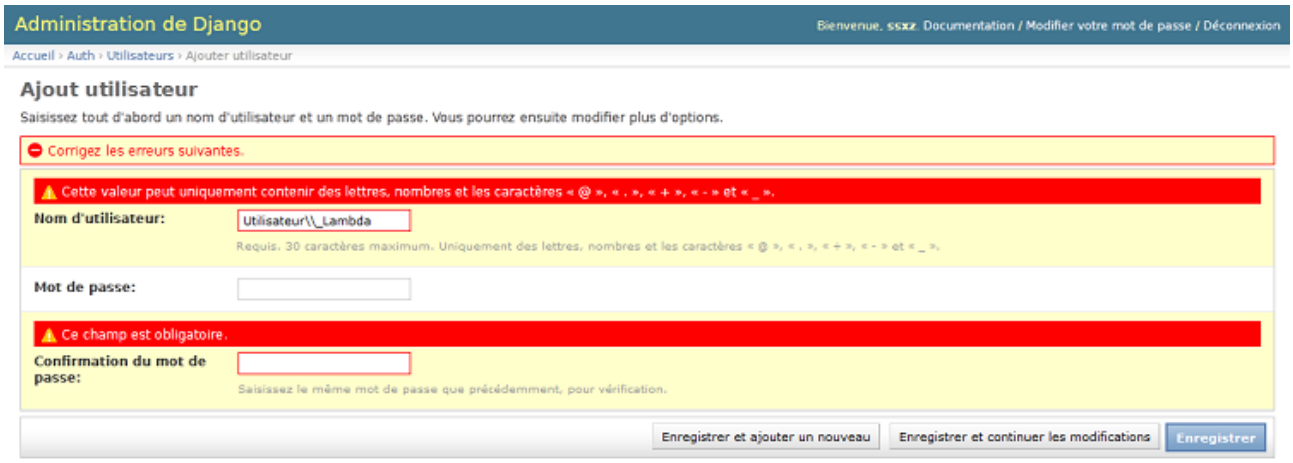


FIGURE 8.4. – Formulaire de création de comptes, après le validateur avec erreur

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).

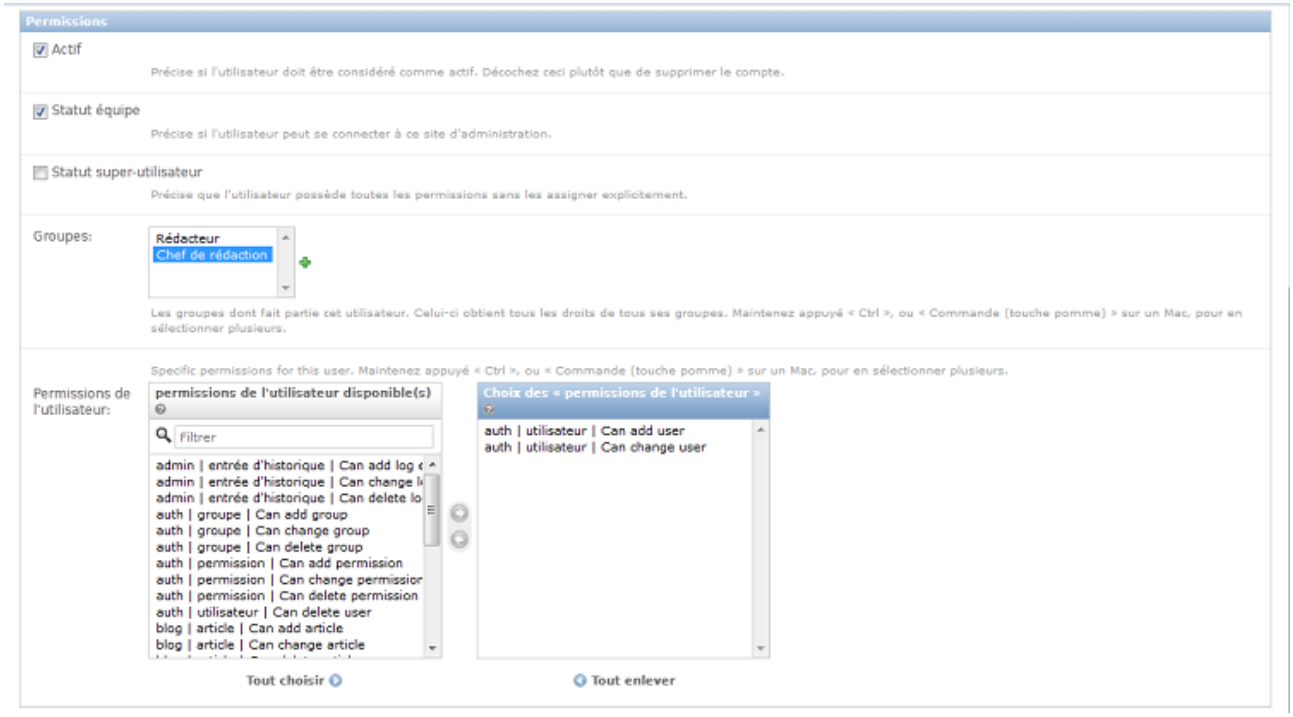


FIGURE 8.5. – Exemple d'édition des permissions, ici j'ai créé deux groupes avant d'éditer l'utilisateur

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.*

FIGURE 8.6. – Historique des modifications d'un objet utilisateur

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).

8.3. 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 :

```
1 from django.contrib import admin
2 from blog.models import Categorie, Article
3
4 admin.site.register(Categorie)
5 admin.site.register(Article)
```

Listing 41 – Ajouter nos propres modèles à l'interface d'admin.

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.

FIGURE 8.7. – 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.

8.3.0.1. Comment cela fonctionne-t-il ?

Au lancement du serveur, le framework va 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.

8.4. 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.

8.4.1. 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.

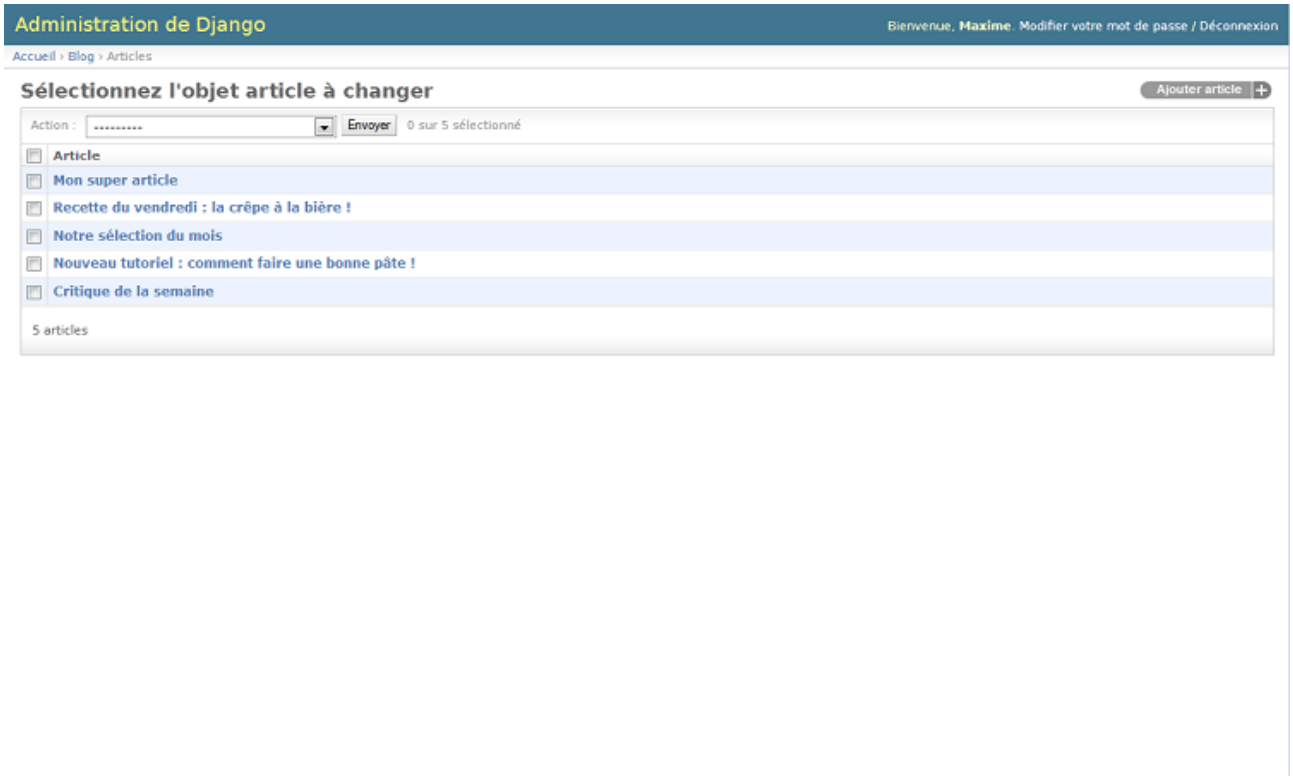


FIGURE 8.8. – Notre liste d'articles, avec uniquement le titre comme colonne

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 `__str__` que nous avons définie dans notre modèle.

Dans les anciennes versions de python, c'était en fait la fonction `__unicode__` qui était appelée. Avec son retrait de la norme python3, django a proposé d'ajouter un *décorateur de classe* (`@python_2_unicode_compatible`) pour s'assurer de la compatibilité avec python2

```
1  class Article(models.Model):
2      titre = models.CharField(max_length=100)
3      auteur = models.CharField(max_length=42)
4      slug = models.SlugField(max_length=100)
5      contenu = models.TextField()
6      date = models.DateTimeField(auto_now_add=True, auto_now=False,
7                                  verbose_name="Date de parution")
8      categorie = models.ForeignKey(Categorie)
9
10     def __str__(self):
11         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 :

```
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` :

```
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 :

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

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

```
1  from django.contrib import admin
2  from blog.models import Categorie, Article
3
4  class ArticleAdmin(admin.ModelAdmin):
5      list_display   = ('titre', 'auteur', 'date')
6      list_filter    = ('auteur', 'categorie')
7      date_hierarchy = 'date'
8      ordering       = ('date',)
9      search_fields  = ('titre', 'contenu')
10
11 admin.site.register(Categorie)
12 admin.site.register(Article, ArticleAdmin)
```

Vous pouvez maintenant observer le résultat sur la figure suivante :

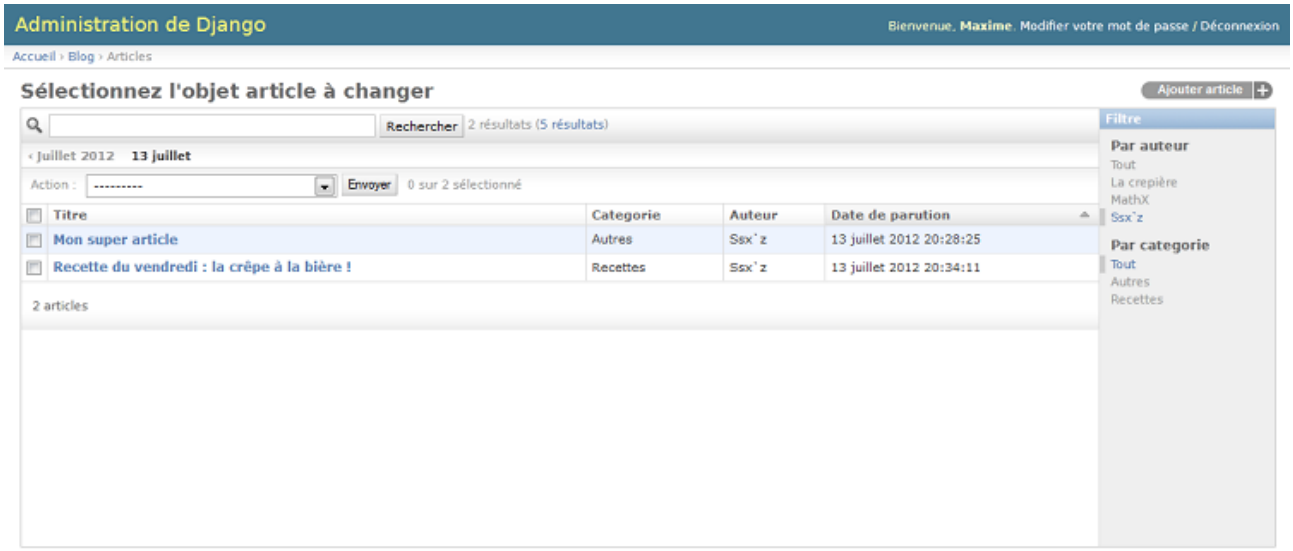


FIGURE 8.9. – La même liste, bien plus complète, et plus pratique !

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 :

```
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[:40]
    if len(article.contenu) > 40:
        return '{}...'.format(text)
    else:
        return text
```

Listing 42 – Sélectionner les 40 premiers caractères.

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 :

```
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[:40]
        if len(article.contenu) > 40:
            return '{}...'.format(text)
        else:
            return text

```

```
22      # En-tête de notre colonne
23      apercu_contenu.short_description = 'Aperçu du contenu'
24
25  admin.site.register(Categorie)
26  admin.site.register(Article, ArticleAdmin)
```

Listing 43 – Mettre un aperçu du contenu dans la page d'administration.

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

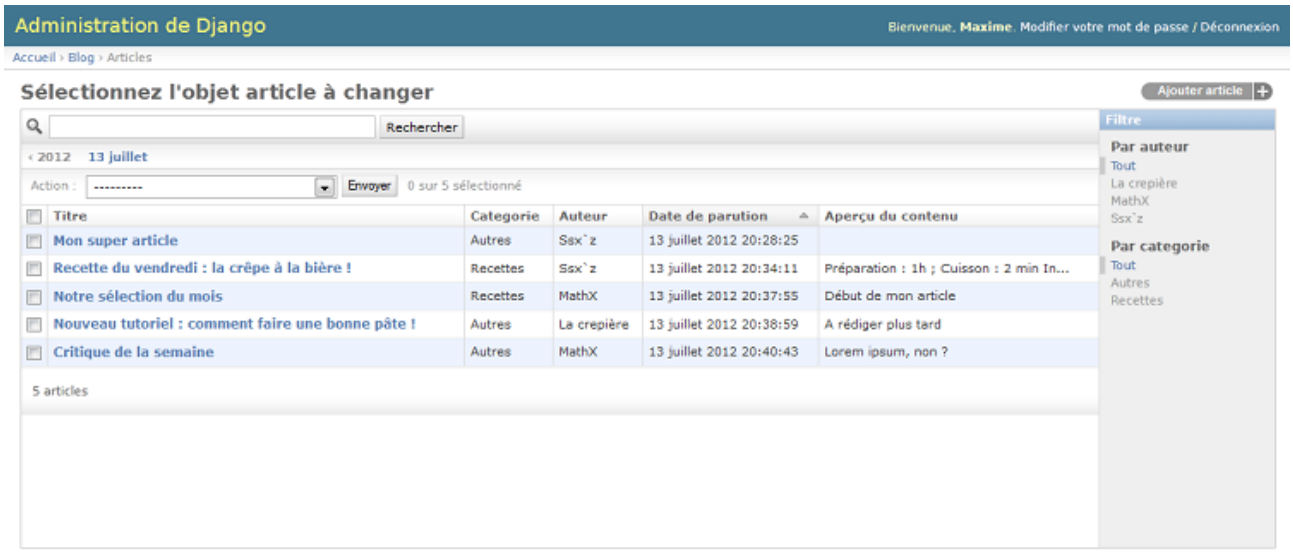


FIGURE 8.10. – La liste des articles est accessible depuis l'administration

8.4.2. 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 qui masque le champ dans l'administration par défaut).

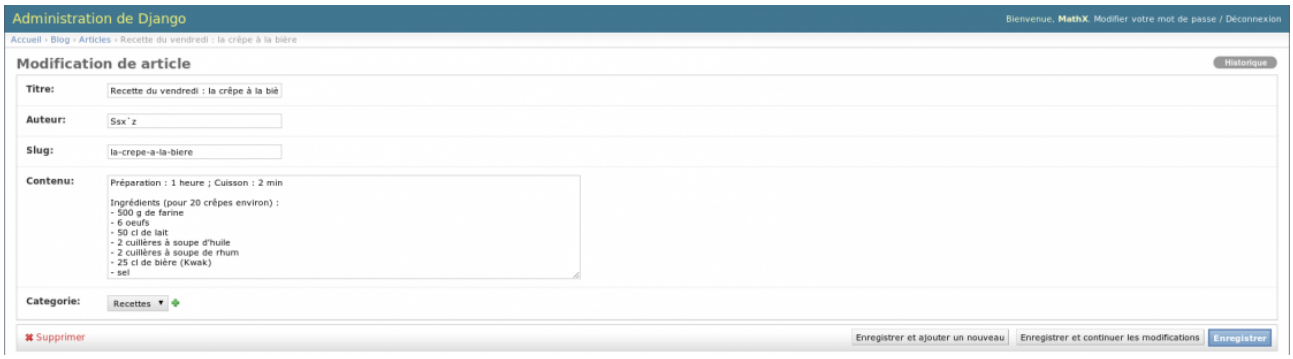


FIGURE 8.11. – Le formulaire d'édition d'un article par défaut

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 :

```
1 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).

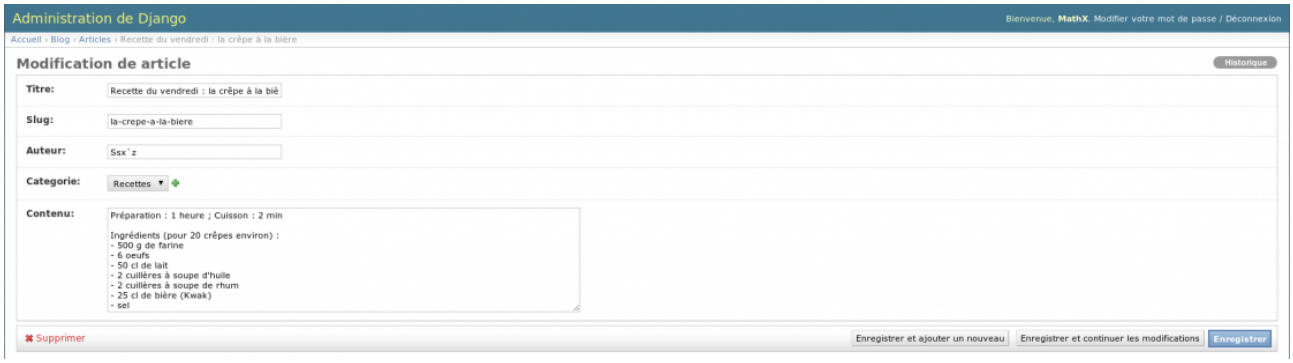


FIGURE 8.12. – Notre formulaire, avec une nouvelle organisation des champs

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.

```
1 fieldsets = (
2     # Fieldset 1 : meta-info (titre, auteur…)
3    ('Général', {
4         'classes': ['collapse',],
5         'fields': ('titre', 'slug', 'auteur', 'categorie')
6     }),
7     # Fieldset 2 : contenu de l'article
```

```
    ('Contenu de l\'article', {
       'description':
           'Le formulaire accepte les balises HTML. Utilisez-les à bon escient
       'fields': ('contenu', )
    }),
)
```

Listing 44 – Hiérarchisation de nos champs.

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 : 1. `fields` : liste des champs à afficher dans le fieldset ;
   a) `description` : une description qui sera affichée en haut du fieldset, avant le premier champ ;
   b) `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 :

```
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
```

```
21          ('Contenu de l\'article', {
22              'description':
                    'Le formulaire accepte les balises HTML. Utilisez-les à bon esc
23              'fields': ('contenu', )
24          }),
25      )
26
27      # Colonnes personnalisées
28      def apercu_contenu(self, article):
29          """
30          Retourne les 40 premiers caractères du contenu de l'article. S'il
31          y a plus de 40 caractères, il faut ajouter des points de suspension.
32          """
33          text = article.contenu[:40]
34          if len(article.contenu) > 40:
35              return '{}...'.format(text)
36          else:
37              return text
38
39      apercu_contenu.short_description = 'Aperçu du contenu'
40
41  admin.site.register(Categorie)
42  admin.site.register(Article, ArticleAdmin)
```

… qui donne la figure suivante.

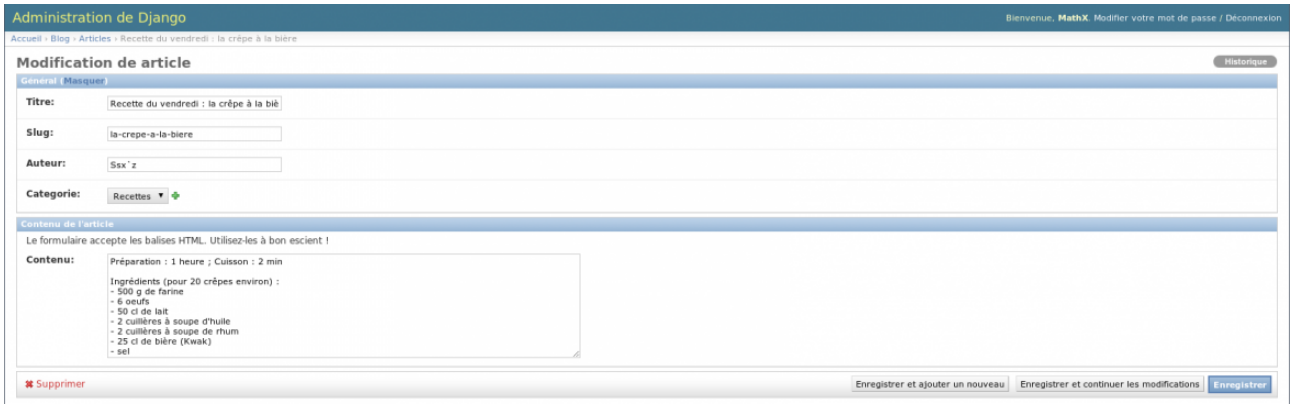


FIGURE 8.13. – Notre formulaire, mieux présenté qu'avant

[[information]]| 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.

8.4.3. 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 :

```
1 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).

FIGURE 8.14. – Exemple d'utilisation de `prepopulated_fields`

8.5. 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.

9. 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 !

9.1. 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 :

```
1 from django import forms
2
3 class ContactForm(forms.Form):
4     sujet = forms.CharField(max_length=100)
5     message = forms.CharField(widget=forms.Textarea)
6     envoyeur = forms.EmailField(label="Votre adresse mail")
7     renvoi =
          forms.BooleanField(help_text="Cochez si vous souhaitez obtenir une cop
          required=False)
```

Listing 45 – blog/forms.py

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.

9.2. 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 :
La voici :

```
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())
```

Listing 46 – Extrait de blog/views.py

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

Listing 47 – Extrait de blog/urls.py

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 :

```
>>> 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 :

```
{% 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>
```

Listing 48 – templates/blog/contact.html

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) :

```
<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>
```

Listing 49 – Le code généré par le template.

Et voici à la figure suivante l'image du rendu (bien entendu, libre à vous de l'améliorer avec un peu de CSS).

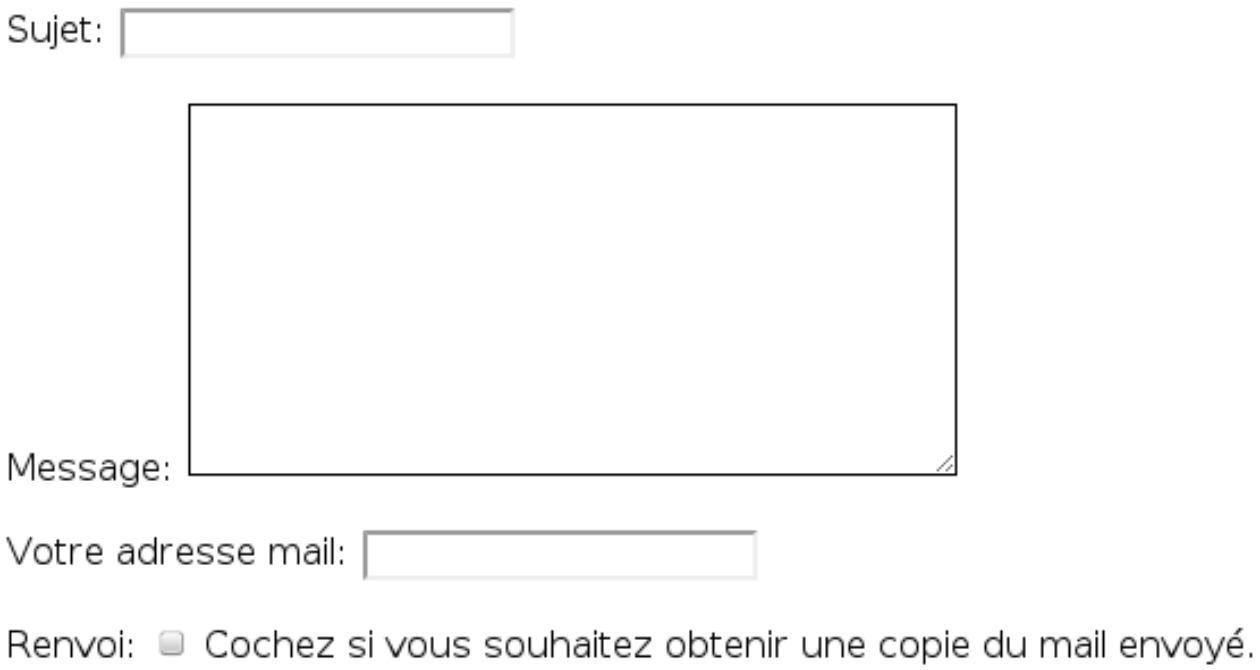


FIGURE 9.1. – Rendu du formulaire

Pour finir, parlons du tag `{% csrf_token %}` qui n'a pas été glissé par hasard. 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 !

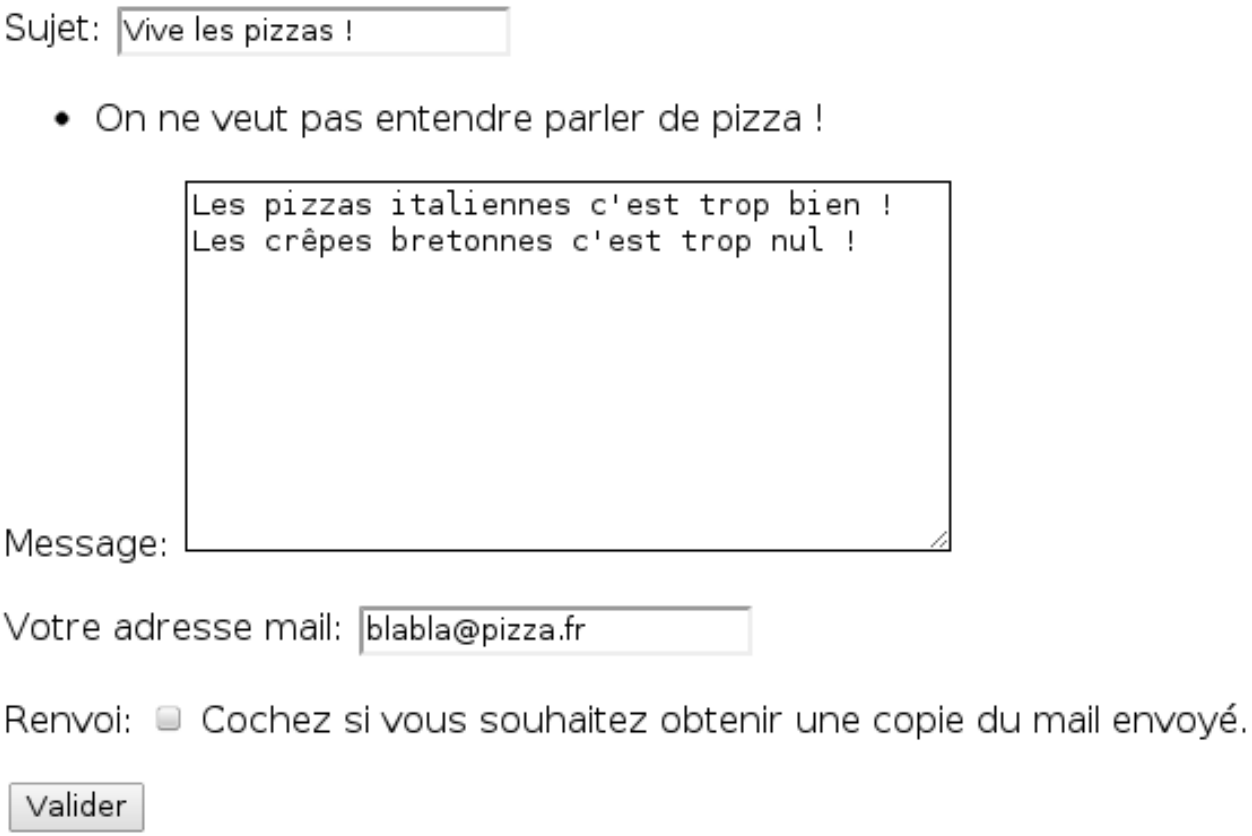
9.3. Créons nos propres règles de validation

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 :

```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.



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 :

```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 ma
            )

    return cleaned_data  # N'oublions pas de renvoyer les données si tout est OK
```

Listing 50 – Notre fonction `clean`

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.

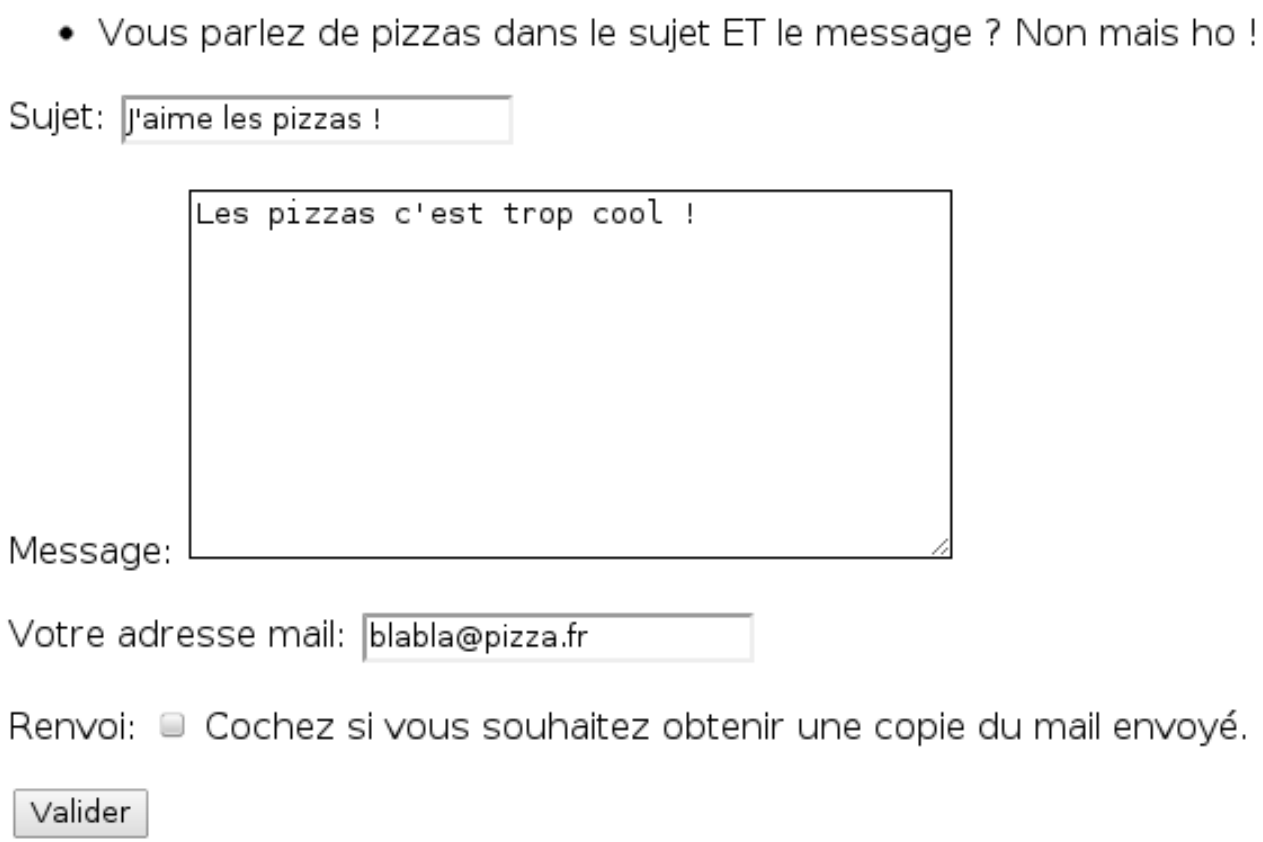


FIGURE 9.2. – Formulaire invalide

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 :

```
1  def clean(self):
2      cleaned_data = super(ContactForm, self).clean()
3      sujet = cleaned_data.get('sujet')
4      message = cleaned_data.get('message')
5
6      if sujet and message:  # Est-ce que sujet et message sont
           valides ?
7          if "pizza" in sujet and "pizza" in message:
8              msg =
                   "Vous parlez déjà de pizzas dans le sujet, n'en parlez plus da
9              self.add_error("message", msg)
10
11     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.

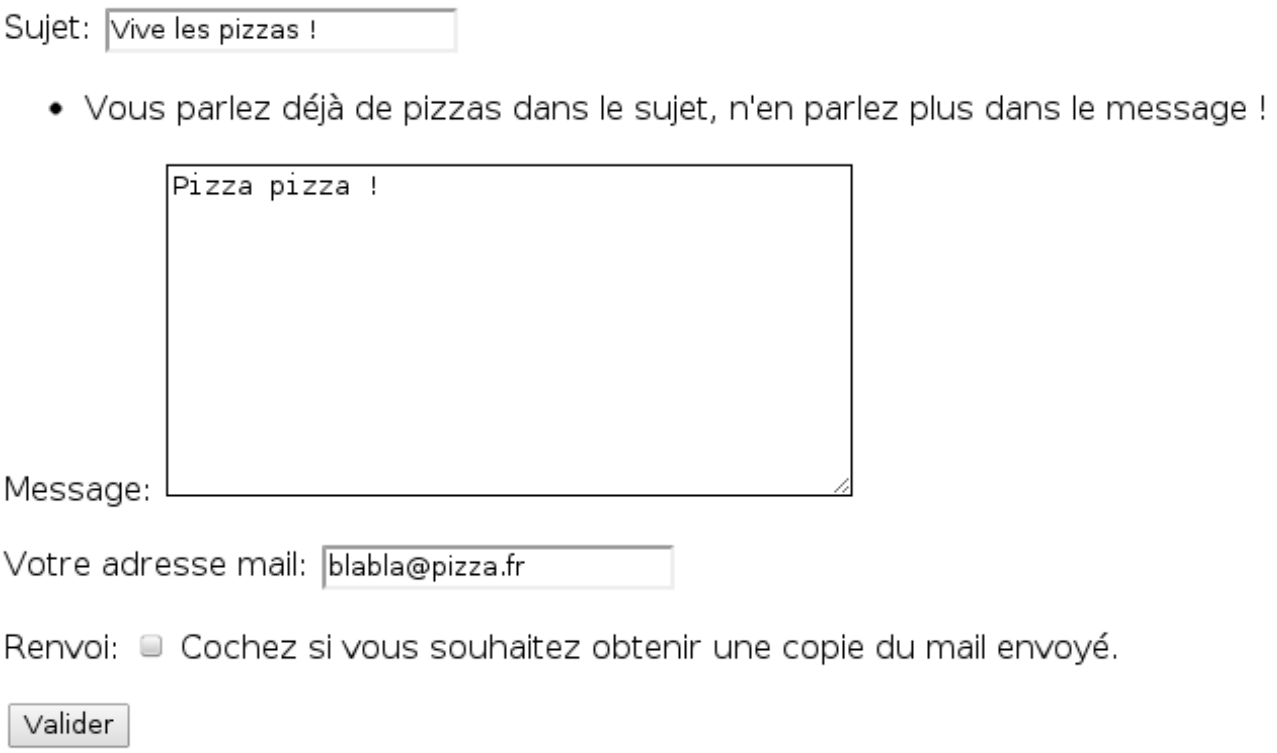


FIGURE 9.3. – Le message d'erreur est bien adapté

9.4. 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 :

```
1  class Article(models.Model):
2      titre = models.CharField(max_length=100)
3      auteur = models.CharField(max_length=42)
4      slug = models.SlugField(max_length=100)
5      contenu = models.TextField(null=True)
6      date = models.DateTimeField(auto_now_add=True, auto_now=False,
           verbose_name="Date de parution")
7      categorie = models.ForeignKey(Categorie)
8
9      def __str__(self):
10         return self.titre
```

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

```python
from django import forms
from models import Article

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

Listing 51 – Extrait de blog/forms.py

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 :

FIGURE 9.4. – 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` :

```
>>> from blog.models import Article, Categorie
>>> from blog.forms import ArticleForm
>>> donnees = {
... 'titre':"Les crêpes c'est trop bon",
... 'slug':"les-crepes-cest-trop-bon",
... 'auteur':"Maxime",
...
    'contenu':"Vous saviez que les crêpes bretonnes c'est trop bon ? La pêche
... '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 :

```
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 !

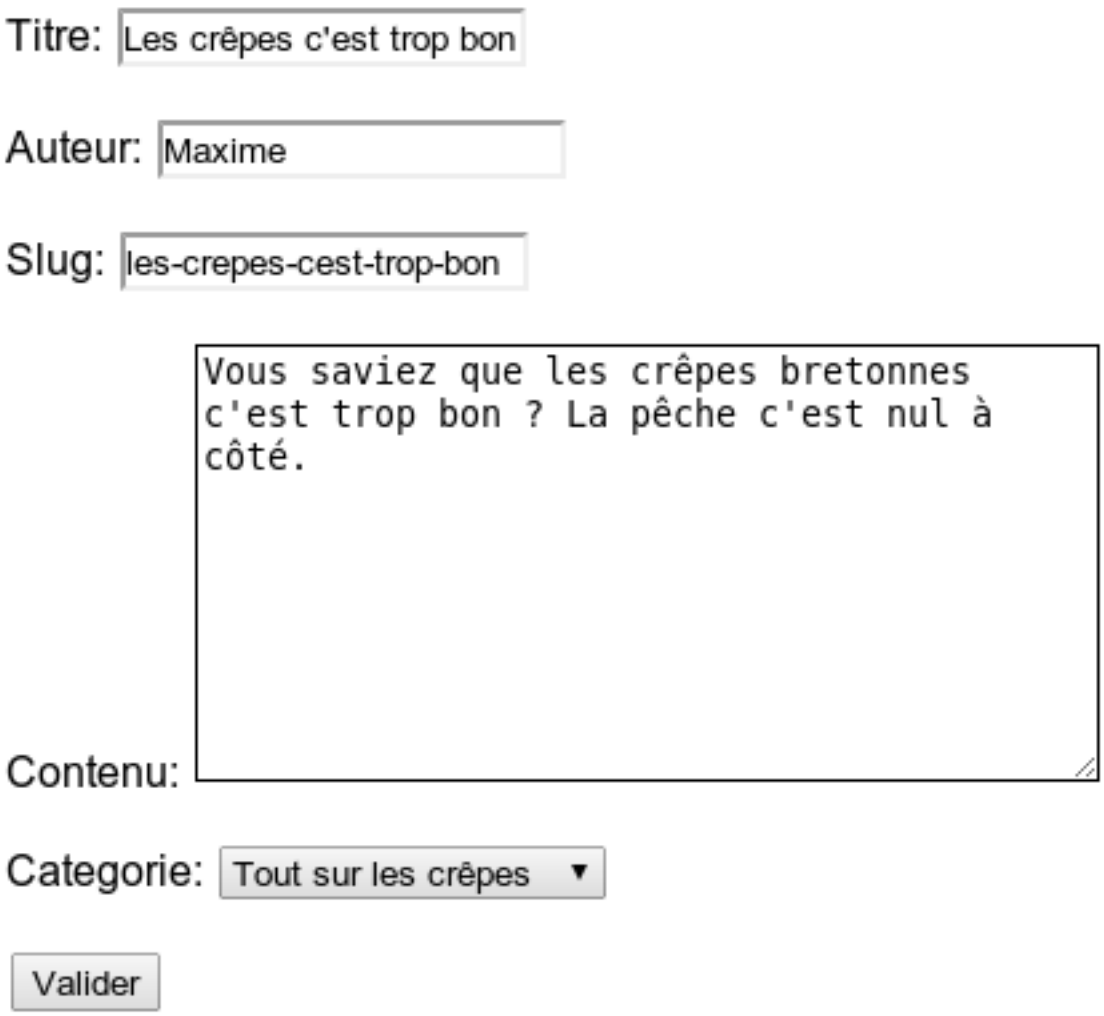


FIGURE 9.5. – L'entrée se met automatiquement à jour !

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 :

```
1 form = ArticleForm(request.POST, instance=article)
2 if form.is_valid():
3     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` :

```
1 class ArticleForm(forms.ModelForm):
2     class Meta:
3         model = Article
4         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 :

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

i 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.

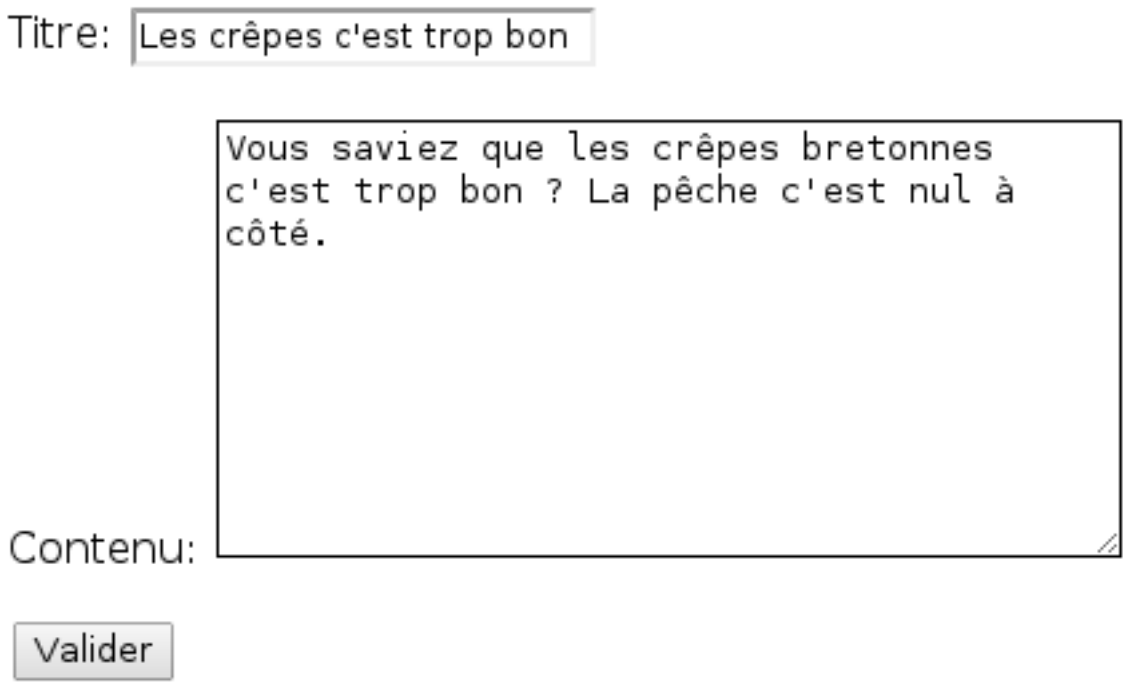


FIGURE 9.6. – Seuls les champs « titre » et « contenu » sont éditables

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 :

```
>>> 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.

9.5. 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.

10. 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.

10.1. Enregistrer une image

Préambule : avant de commencer à jouer avec des images, il est nécessaire d'installer la bibliothèque *Pillow*. Django se sert en effet de cette dernière pour faire ses traitements sur les images. Vous pouvez télécharger la bibliothèque en utilisant `pip install pillow` ou via 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 ») :

```
1 class Contact(models.Model):
2     nom = models.CharField(max_length=255)
3     adresse = models.TextField()
4     photo = models.ImageField(upload_to="photos/")
5 
6     def __str__(self):
7             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 en autre autre l'argument : `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, avec par exemple `MEDIA_ROOT = os.path.join(BASE_DIR, '/media/')`. Si vous ne spécifiez pas de valeur à `upload_to`, les images seront enregistrées à la racine de `MEDIA_ROOT`.
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 :

```
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 :

```
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 :

```
>>> 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 :

```
<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.

10.2. 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 :

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

```

```
4 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 :

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

Côté template :

```
1 <h1>Liste des contacts</h1>
2 {% for contact in contacts %}
3     <h2>{{ contact.nom }}</h2>
4     Adresse : {{ contact.adresse|linebreaks }}<br/>
5     <img src="{{ contact.photo.url }}"/>
6 {% 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.
Le résultat est plutôt simple, comme vous pouvez le constater sur la figure suivante.

FIGURE 10.1. – 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.

10.3. 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) :

```
>>> from blog.models import Contact
>>> c = Contact.objects.get(nom="Chuck Norris")
>>> c.photo.name
```

```
'photos/chuck_norris.jpg'  # Chemin relatif vers le fichier à partir de MEDIA_ROOT
>>> c.photo.path
'/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 :

```
def renommage(instance, nom):
    nom_fichier = os.path.splitext(nom)[0] # on retire l'extension
    return "{}-{}".format(instance.id, nom_fichier)
```

Ici, notre fonction préfixe le nom de fichier par l'identifiant unique de l'instance de modèle en cours. Un exemple de modèle utilisant cette fonction serait donc simplement :

```
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 !

10.4. 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`.

11. 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).

11.1. 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 :

```
1 import random
2 import string
3
4 def generer(nb_caracteres):
5     caracteres = string.ascii_letters + string.digits
```

```
6       aleatoire = [random.choice(caracteres) for _ in
            range(nb_caracteres)]
7
8       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.

11.2. Correction

Normalement, cela ne devrait pas avoir posé de problèmes !
Il fallait donc créer une nouvelle application et l'inclure dans votre `settings.py` :

```
python manage.py startapp mini_url
```

Votre `models.py` devrait ressembler à ceci :

```
from django.db import models
import random
import string


class MiniURL(models.Model):
    url = models.URLField(verbose_name="URL à réduire",
        unique=True)
    code = models.CharField(max_length=6, unique=True)
    date = models.DateTimeField(auto_now_add=True,

                                        verbose_name="Date d'enregistrement")
    pseudo = models.CharField(max_length=255, blank=True,
        null=True)
    nb_acces = models.IntegerField(default=0,

                                           verbose_name="Nombre d'accès à l'URL")

    def __str__(self):
        return "[{0}] {1}".format(self.code, self.url)

    def save(self, *args, **kwargs):
        if self.pk is None:
```

```
20              self.generer(6)
21
22          super(MiniURL, self).save(*args, **kwargs)
23
24      def generer(nb_caracteres):
25          caracteres = string.ascii_letters + string.digits
26          aleatoire = [random.choice(caracteres) for _ in
                range(nb_caracteres)]
27
28          self.code = ''.join(aleatoire)
29
30      class Meta:
31          verbose_name = "Mini URL"
32          verbose_name_plural = "Minis URL"
```

Listing 52 – models.py

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` !) :

```
$ python3 manage.py makemigrations
Migrations for 'mini_url':
  0001_initial.py:
    - Create model MiniURL
$ python3 manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: admin, contenttypes, auth, sessions
  Apply all migrations: blog, mini_url
Synchronizing apps without migrations:
  Creating tables...
  Installing custom SQL...
  Installing indexes...
Running migrations:
  Applying mini_url.0001_initial... OK
```

Le `forms.py` est tout à fait classique :

```python
from django import forms
from models import MiniURL

class MiniURLForm(forms.ModelForm):
    class Meta:
        model = MiniURL
        fields = ('url', 'pseudo')
```

Listing 53 – mini_url/forms.py

De même pour `admin.py` :

```python
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)
```

Listing 54 – mini_url/admin.py

Voici `mini_url/urls.py`. N'oubliez pas de l'importer dans votre `urls.py` principal. Rien de spécial non plus :

```python
from django.conf.urls import url
from . import views

urlpatterns = [
    # Une string vide indique la racine
    url(r'^$', views.liste, name='url_liste'),
    url(r'^nouveau$', views.nouveau, name='url_nouveau'),
    # (?P<code>\w{6}) capturera 6 caractères alphanumériques.
    url(r'^(?P<code>\w{6})/$', views.redirection,
        name='url_redirection'),
)
```

Listing 55 – mini_url/urls.py

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

```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` :

```
1  from django.shortcuts import redirect, get_object_or_404, render
2  from mini_url.models import MiniURL
3  from mini_url.forms import MiniURLForm
4
5
6  def liste(request):
7      """ Affichage des redirections """
8      minis = MiniURL.objects.order_by('-nb_acces')
9
10     return render(request, 'mini_url/liste.html', locals())
11
12
13 def nouveau(request):
14     """ Ajout d'une redirection """
15     if request.method == "POST":
16         form = MiniURLForm(request.POST)
17         if form.is_valid():
18             form.save()
19             return redirect(liste)
20     else:
21         form = MiniURLForm()
22
23     return render(request, 'mini_url/nouveau.html', {'form': form})
24
25
26 def redirection(request, code):
27     """ Redirection vers l'URL enregistrée """
28     mini = get_object_or_404(MiniURL, code=code)
29     mini.nb_acces += 1
30     mini.save()
31
32     return redirect(mini.url, permanent=True)
```

Listing 56 – mini_url/views.py

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).

```
<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>
```

Listing 57 – liste.html

```
<h1>Raccourcir une URL</h1>

<form method="post" action="{% url 'url_nouveau' %}">
    {% csrf_token %}
    {{ form.as_p }}
    <input type="submit"/>
</form>
```

Listing 58 – nouveau.html

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.

Troisième partie

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.

12. 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 information dynamique par exemple, ou encore de simple listes d'objets sans traitement particulier.
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.

12.1. 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 :

```
1 from django.shortcuts import render
2
3 def faq(request):
4     return render(request, 'blog/faq.html', {}))
```

Listing 59 – Exemple de vue mal conçue

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

Listing 60 – ... avec sa définition d'URL associée

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 :

```
1 from django.views.generic import TemplateView
2
3 class FAQView(TemplateView):
4     template_name = "blog/faq.html"  # chemin vers le template à
          afficher
```

Listing 61 – Dans un fichier views.py

Dès lors, il suffit de router notre URL vers une méthode héritée de la classe `TemplateView`, ici `as_view` :

```
1 from django.conf.urls import patterns, url, include
2 from blog.views import FAQView  # N'oubliez pas d'importer la
      classe mère
3
4 urlpatterns = patterns('',
5     (r'^faq/$', FAQView.as_view()),    # Nous demandons la vue
          correspondant à la classe FAQView créée précédemment
6 )
```

Listing 62 – blog/urls.py

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` :

```
1 from django.conf.urls import patterns, url, include
2 from django.views.generic import TemplateView  # L'import a changé,
      attention !
3
4 urlpatterns = patterns('',
5     url(r'^faq/',
          TemplateView.as_view(template_name='blog/faq.html')),
6 )
```

Listing 63 – blog/urls.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`.

12.2. 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 de la partie précédente 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.

12.2.1. 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 :

```
1  from django.conf.urls import patterns, url, include
2  from django.views.generic import ListView
3
4  from . import views
5  from .models import Article
6
7  urlpatterns = [
8      # Nous allons réécrire l'URL de l'accueil
9      url(r'^$', ListView.as_view(model=Article,)),
10
11     # Et nous avons toujours nos autres pages…
12     url(r'^article/(?P<id>\d+)$', views.lire),
13     url(r'^(?P<page>\d+)$', views.archives),
14     url(r'^categorie/(?P<slug>.+)$', views.voir_categorie),
15 ]
```

Listing 64 – blog/urls.py

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` :

```
urlpatterns = [
    url(r'^$', ListView.as_view(model=Article,
                    context_object_name="derniers_articles",
                    template_name="blog/accueil.html")),
    ...
]
```

Listing 65 – blog/urls.py

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 :

```
import views

urlpatterns = [
    # Via la fonction as_view, comme vu tout à l'heure
    url(r'^$', views.ListeArticles.as_view(), name="blog_liste"),
    ...
]
```

Listing 66 – urls.py

C'est l'occasion d'expliquer l'interet de 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 :

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
```

Listing 67 – blog/views.py

Désormais, nous souhaitons paginer nos résultats, afin de n'afficher que 5 articles par page, par exemple. Il existe un attribut adapté :

```
class ListeArticles(ListView):
    model = Article
    context_object_name = "derniers_articles"
    template_name = "blog/accueil.html"
    paginate_by = 5
```

Listing 68 – blog/views.py

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.

Nous parlerons plus loin du fonctionnement de la pagination, dans la partie IV.

Listing 69 – blog/accueil.html

Allons plus loin ! Nous pouvons également *surcharger la sélection des objets* à récupérer, et ainsi soumettre nos propres filtres :

```
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)
```

Listing 70 – views.py

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.

```
from django.conf.urls import url
import views

urlpatterns = [
    url(r'^categorie/(\d+)$', views.ListeArticles.as_view(),
        name='blog_categorie'),
    url(r'^article/(?P<id>\d+)$', views.lire),
    url(r'^(?P<page>\d+)$', views.archives)
```

```
]
```

Listing 71 – blog/urls.py

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.

Listing 72 – views.py

```python
def get_queryset(self):
    return Article.objects.filter(categorie__id=self.kwargs['id'])
```

```python
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
```

Listing 73 – blog/views.py

```
<h3>Catégories disponibles</h3>
<ul>
{% for categorie in categories %}
    <li><a href="{% url "blog_categorie" categorie.id %}">{{ categorie.nom }}</a></li>
{% endfor %}
</ul>
```

Listing 74 – Extrait de blog/accueil.html

12.2.2. Afficher un article via `DetailView`

pk

```
import views

urlpatterns = [
    url(r'^categorie/(\w+)$', views.ListeArticles.as_view()),
    url(r'^article/(?P<pk>\d+)$', views.LireArticle.as_view(),
        name='blog_lire'),
]
```

Listing 75 – blog/urls.py

```
class LireArticle(DetailView):
    context_object_name = "article"
    model = Article
    template_name = "blog/lire.html"
```

Listing 76 – blog/views.py

LireView

article

```
<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>
```

ListView

get_queryset

get_object

```
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()
```

```
9
10          article.nb_vues += 1  # Imaginons un attribut « Nombre de
                vues »
11          article.save()
12
13          return article  # Et nous retournons l'objet à afficher
```

Listing 77 – blog/views.py

`request`

`self.request`

12.3. Agir sur les données

CRUD

- `Create`
- `Read`
- `Update`
- `Delete`

12.3.1. CreateView

`CreateView`

```
1 from django.views.generic import CreateView
2 from django.core.urlresolvers import reverse_lazy
3
4 class URLCreate(CreateView):
5     model = MiniURL
6     template_name = 'mini_url/nouveau.html'
7     form_class = MiniURLForm
8     success_url = reverse_lazy(liste)
```

Listing 78 – blog/views.py

`model`

`template_name`

`<app>/<model>_create_form.html`

```python
from django.conf.urls import url
from . import views

urlpatterns = [
    # Une string vide indique la racine
    url(r'^$', views.liste, name='url_liste'),
    url(r'^nouveau$', views.URLCreate.as_view(),
        name='url_nouveau'),
    # (?P<code>\w{6}) capturera 6 caractères alphanumériques.
    url(r'^(?P<code>\w{6})$', views.redirection,
        name='url_redirection'),
]
```

Listing 79 – urls.py

12.3.2. UpdateView

```python
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)
```

Listing 80 – blog/views.py

CreateView UpdateView

UpdateView <app>/<model>_update_form.html

```
<form method="post" action="{% url "url_nouveau" %}">
```

```
<form method="post" action="">
```

urls.py DetailView pk

```
url(r'^edition/(?P<pk>\d)/$', URLUpdate.as_view(),
    name='url_update'),
```

MiniURL /url/edition/1 /url/edition/2

FIGURE 12.1. – Exemple de formulaire de mise à jour, reprenant le même template que l'ajout

12.3.2.1. Améliorons nos URL avec la méthode `get_object()`

pk

FIGURE 12.2. – Ce que nous avons actuellement et ce que nous souhaitons avoir

get_object

```python
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)
```

Listing 81 – blog/views.py

get_object_or_404

self.kwargs

urls.py code

```
1 url(r'^edition/(?P<code>\w{6})/$', views.URLUpdate.as_view(),
      name='url_update'),
```

12.3.2.2. Effectuer une action lorsque le formulaire est validé avec `form_valid()`

form_valid

```
1 def form_valid(self, form):
2             self.object = form.save()
3             messages.success(self.request,
                  "Votre profil a été mis à jour avec succès.")  #
                  Envoi d'un message à l'utilisateur
4             return HttpResponseRedirect(self.get_success_url())
```

Listing 82 – Un `form_valid()` personnalisé

12.3.3. DeleteView

UpdateView

```
1 class URLDelete(DeleteView):
2     model = MiniURL
3     context_object_name = "mini_url"
4     template_name = 'mini_url/supprimer.html'
5     success_url = reverse_lazy(liste)
6
7     def get_object(self, queryset=None):
8         code = self.kwargs.get('code', None)
9         return get_object_or_404(MiniURL, code=code)
```

Listing 83 – mini_url/views.py

supprimer.html

```
<h1>Êtes-vous sûr de vouloir supprimer cette URL ?</h1>

<p>{{ mini_url.code }} -&gt; {{ 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>
```

Listing 84 – supprimer.html

`urls.py`

`URLUpdate`

```
url(r'^supprimer/(?P<code>\w{6})/$', views.URLDelete.as_view(), name='url_delete'),
```

Listing 85 – mini_url/urls.py

`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 %}
```

```
17 </ul>
```

Listing 86 – liste.html

FIGURE 12.3. – Notre vue, après avoir cliqué sur un des liens « Supprimer » qui apparaissent dans la liste

ccbv.co.uk

— Documentation officielle sur les vues génériques
— Documentation non officielle mais très complète, listant les attributs et méthodes de chaque classe

12.4. En résumé

—

— urls.py

TemplateView views.py

—

—

— django.views.generic

13. Techniques avancées dans les modèles

13.1. Les requêtes complexes avec Q

Q

Q

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

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

manage.py shell

```
>>> 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()
```

Q

```
>>> from django.db.models import Q
>>> Q(nom="Maxime")
# 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)>]
```

Listing 87 – Utilisation basique de Q.

```
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)>]
```

Listing 88 – Une requête avec un OU.

SQL

```
>>> Eleve.objects.filter(Q(moyenne=10) & Q(nom="Sofiane"))
[<Eleve: Élève Sofiane (10/20 de moyenne)>]
```

```
>>> Eleve.objects.filter(Q(moyenne=10), Q(nom="Sofiane"))
[<Eleve: Élève Sofiane (10/20 de moyenne)>]
```

SQL

```
>>> Eleve.objects.filter(Q(moyenne=10), ~Q(nom="Sofiane"))
[<Eleve: Élève Thibault (10/20 de moyenne)>]
```

```
conditions = [('moyenne', 15), ('nom', 'Thibault'), ('moyenne',
    18)]
```

Q

```
objets_q = [Q(x) for x in conditions]
```

```
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

reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])

operator.or_

```
Eleve.objects.filter(objets_q[0] | objets_q[1] | objets_q[2])
```

13.2. L'agrégation

aggregate

```
from django.db.models import Avg
>>> Eleve.objects.aggregate(Avg('moyenne'))
{'moyenne__avg': 11.25}
```

Avg

Avg 'moyenne' moyenne__avg

Avg django.db.models

— Max
— Min
— Count

```
>>> Eleve.objects.aggregate(Avg('moyenne'), Min('moyenne'),
    Max('moyenne'))
{'moyenne__max': 18, 'moyenne__avg': 11.25, 'moyenne__min': 7}
```

```
>>> Eleve.objects.aggregate(Moyenne=Avg('moyenne'),
    Minimum=Min('moyenne'), Maximum=Max('moyenne'))
{'Minimum': 7, 'Moyenne': 11.25, 'Maximum': 18}
```

QuerySet

filter

```
>>>
    Eleve.objects.filter(nom__startswith="Ma").aggregate(Avg('moyenne'),
    Count('moyenne'))
{'moyenne__count': 2, 'moyenne__avg': 12.5}
```

moyenne__count

Count

QuerySet

```
>>> Eleve.objects.filter(nom__startswith="Ma").count()
2
```

```python
class Cours(models.Model):
    nom = models.CharField(max_length=31)
    eleves = models.ManyToManyField(Eleve)

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

```
>>> 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"))
```

ForeignKey

ManyToManyField

```
>>> Cours.objects.aggregate(Max("eleves__moyenne"))
{'eleves__moyenne__max': 18}
```

```
>>> Cours.objects.aggregate(Count("eleves"))
{'eleves__count': 6}
```

```
>>>
    Cours.objects.annotate(Avg("eleves__moyenne"))[0].eleves__moyenne__avg
11.25
```

```
>>>
    Cours.objects.annotate(Moyenne=Avg("eleves__moyenne"))[1].Moyenne
12.5
```

QuerySet filter exclude order_by

```
>>> Cours.objects.annotate(Moyenne=Avg("eleves__moyenne")).filter(Moyenne__gte=
[<Cours: Anglais>]
```

13.3. L'héritage de modèles

13.3.1. Les modèles parents abstraits

abstract=True

Meta

```
class Document(models.Model):
    titre = models.CharField(max_length=255)
    date_ajout = models.DateTimeField(auto_now_add=True,
                                      verbose_name="Date d'ajout du documer
    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")
```

Listing 89 – Une chaine d'héritages.

13.3.2. Les modèles parents classiques

```python
class Lieu(models.Model):
    nom = models.CharField(max_length=50)
    adresse = models.CharField(max_length=100)

    def __str__(self):
        return self.nom

class Restaurant(Lieu):
    menu = models.TextField()
```

Listing 90 – Héritage sans modèle abstrait.

```
>>>
    Restaurant(nom=u"La crêperie bretonne",adresse="42 Rue de la crêpe 35000 Re
    menu=u"Des crêpes !").save()
>>> Restaurant.objects.all()
[<Restaurant: La crêperie bretonne>]
>>> Lieu.objects.all()
[<Lieu: La crêperie bretonne>]
```

Lieu Restau
rant

```
>>> resto = Restaurant.objects.all()[0]
>>> print resto.nom+", "+resto.menu
La crêperie bretonne, Des crêpes !
```

Restaurant
Lieu

```
>>> 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'
```

Restaurant Lieu

```
>>> print type(lieu.restaurant)
<class 'blog.models.Restaurant'>
>>> print(lieu.restaurant.menu)
Des crêpes !
```

13.3.3. Les modèles proxy

Meta

```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
```

```
>>> 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
```

13.4. L'application ContentType

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

ct Eleve
ContentType
— model_class
— get_object_for_this_type
ct.model_class().objects.get(attr=arg)

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

ContentType

Document
ForeignKey Document Article Image
Video

ContentTypes

Commentaire
Image Video Commentaire

Commentaire

```
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fieds import GenericForeignKey

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

    def __str__(self):
```

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

Listing 91 – Un commentaire générique.

```
1  >>> from blog.models import Commentaire, Eleve
2  >>> e = Eleve.objects.get(nom="Sofiane")
3  >>> c =
       Commentaire.objects.create(auteur="Le professeur",contenu="Sofiane ne trava
       content_object=e)
4  >>> c.content_object
5  <Eleve: Élève Sofiane (10/20 de moyenne)>
6  >>> c.object_id
7  4
8  >>> c.content_type.model_class()
9  <class 'blog.models.Eleve'>
```

```python
from django.contrib.contenttypes.fields import GenericRelation

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

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

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

```
commentaires = GenericRelation(Commentaire,
    content_type_field="le_champ_du_content_type",
    object_id_field="le champ_de_l_id")
```

13.5. En résumé

—

— Avg Max Min

Count

—

— ContentType

14. Simplifions nos templates : filtres, tags et contextes

14.1. Préparation du terrain : architecture des filtres et tags

—

—

`templatetags` `s`

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.

```
blog/
   __init__.py
   models.py
   templatetags/
       __init__.py    # À ne pas oublier
      blog_extras.py
   views.py
```

Listing 92 – Architecture avec les templatetags

`admin.py`

admin.site.register()

blog_extras.py

```
1 from django import template
2
3 register = template.Library()
```

templatetags INSTALLED_APPS

settings.py

```
1 {% 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.

{% load %}

```
1 {% load blog_extras static i18n %}
```

14.2. Personnaliser l'affichage de données avec nos propres filtres

—

—

```
1 {{ texte|upper }}            -> Filtre upper sur la variable
    "texte"
```

```
{{ texte|truncatewords:80 }} -> Filtre truncatewords, avec comme
    argument "80" sur la variable "texte"
```

14.2.1. Un premier exemple de filtre sans argument

« Citation » de Wikipédia

{{ "Bonjour le monde !"|cita
tion }}

citation blog_extras.py

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

citation

—

@register.filter

name

—

register.filter('citation', citation)

filtre_citation

```
from django import template

register = template.Library()

@register.filter
def citation(texte):
    return "« %s »" % texte


@register.filter(name='mon_filtre_citation')
def citation2(texte):
    return "« %s »" % texte

```

```
14 def citation3(texte):
15     return "« %s »" % texte
16
17 register.filter('un_autre_filtre_citation', 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.

`load` `citation`

```
1 {% load blog_extras %}
2 Un jour, une certaine personne m'a dit : {{
      "Bonjour le monde !"|citation }}
```

Un jour, une certaine personne m'a dit : « Bonjour le monde ! ».

FIGURE 14.1. – Le résultat incorrect de notre filtre

`{{ "<strong>Bonjour</strong> le monde !"|citation }}`

`escape` `django.utils.html`

```
1  from django import template
2  from django.utils.html import escape
3  from django.utils.safestring import mark_safe
4
5  register = template.Library()
6
7  @register.filter(is_safe=True)
8  def citation(texte):
9      """
10     Affiche le texte passé en paramètre, encadré de guillemets
11     français doubles et d'espaces insécables.
```

```
12      """
13      res = "« %s »" % escape(texte)
14      return mark_safe(res)
```

Listing 93 – notre filtre, maintenant `safe`

14.2.2. Un filtre avec arguments

```
1  {{ ma_chaine|smart_truncate:40 }}
```

`smart_truncate` `ma_chaine`

```
1  def smart_truncate(texte, nb_caracteres):
2
3      # Nous vérifions tout d'abord que l'argument passé est bien un
           nombre
4      try:
5              nb_caracteres = int(nb_caracteres)
6      except ValueError:
7              return texte  # Retour de la chaîne originale sinon
8
9      # Si la chaîne est plus petite que le nombre de caractères
           maximum voulus,
10     # nous renvoyons directement la chaîne telle quelle.
11     if len(texte) <= nb_caracteres:
12             return texte
13
14     # […]
```

```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 + '…'
```

Listing 94 – smart truncate

```
@register.filter
def smart_truncate(texte, nb_caracteres):
```

```
<p>
   {{ "Bonjour"|smart_truncate:14 }}<br />
```

```
3     {{ "Bonjour tout le monde"|smart_truncate:15 }}<br />
4     {{ "Bonjour tout le monde, c'est bientôt Noël"|smart_truncate:18
          }}<br />
5     {{ "To be or not to be, that's the question"|smart_truncate:16
          }}<br />
6 </p>
```

> Bonjour Bonjour tout le... Bonjour tout le... To be or not to...

```
1 def smart_truncate(texte, nb_caracteres=20):
```

```
1 {{ "To be or not to be, that's the question"|smart_truncate }}
```

14.3. Les contextes de templates

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

news date_actuelle

news date

14.3.1. Un exemple maladroit : afficher la date sur toutes nos pages

```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())
```

`{{ date_actuelle }}`

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.

14.3.2. Factorisons encore et toujours

`context_processors.py`

`crepes_bretonnes`

`request`

`render`

```python
from datetime import datetime

def get_infos(request):
    date_actuelle = datetime.now()
    return {'date_actuelle': date_actuelle}
```

`date_actuelle`

`settings.py`

`TEMPLATE_CONTEXT_PROCESSORS`
`settings.py`

```
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"
)
```

```
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",
)
```

`date_actuelle`

```
<p>Bonjour à tous, nous sommes le {{ date_actuelle }} et il fait
    beau en Bretagne !</p>
```

`date_actuelle`

```
Bonjour à tous, nous sommes le 15 novembre 2012 23:58:16 et il fait
    beau en Bretagne !
```

14.3.2.1. Petit point technique sur l'initialisation du contexte

`render` `HttpResponse` `render` `django.shortcut` `render_to_response`

```
from django.shortcuts import render_to_response
[...]
return render_to_response('blog/archives.html', locals())
```

`render_to_response` `TEMPLATE_CONTEXT_PROCESSOR`

```
return render_to_response('blog/archives.html', locals(),
    context_instance=RequestContext(request))
```

`render`

14.4. Des structures plus complexes : les custom tags

```
Bonjour, nous sommes le {% now %}. Je suis {{ prenom }} {{
    nom|upper }}
```

— `TextNode` `"Bonjour, nous sommes le "`
— `Now node`

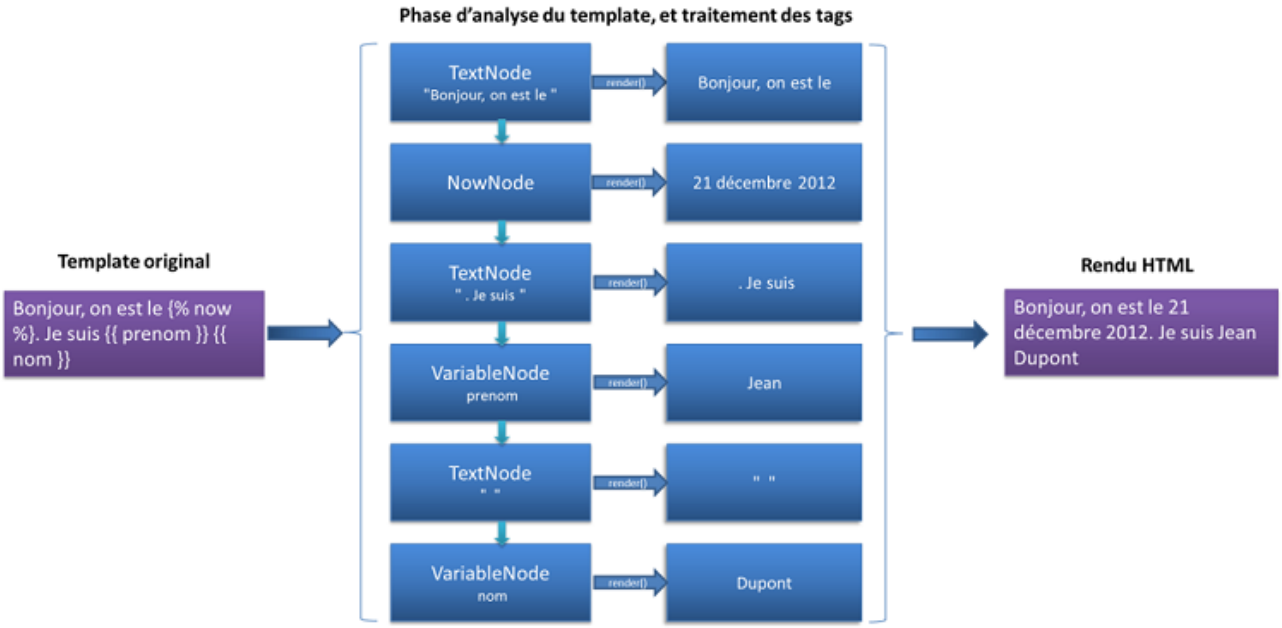


FIGURE 14.2. – Schéma d'exécution

14.4.1. Première étape : la fonction de compilation

`split_contents()`

`token.contents.split(' ')`

```
from django import template

def random(parser, token):

        """ Tag générant un nombre aléatoire, entre les bornes données en argu
    # 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 = '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 = '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 = 'L\'argument "begin" doit obligatoirement être inférieur à l\'argu
        % nom_tag
        raise template.TemplateSyntaxError(msg)

    return RandomNode(begin, end)
```

`msg`

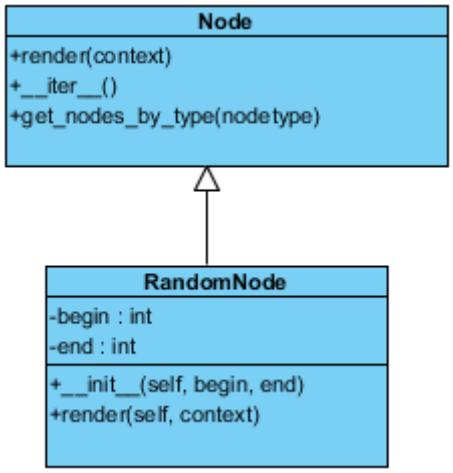


FIGURE 14.3. – Diagramme UML de notre classe RandomNode

RandomNode __init__ self begin end render(self, context)

```
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))
```

render

- @register.tag()
- @register.tag(name='nom_du_tag')
- register.tag('nom_du_tag', random)

```
from django import template
from random import randint

```

```
4  register = template.Library()
5
6  @register.tag
7  def random(parser, token):
8
           """ Tag générant un nombre aléatoire, entre les bornes données en argu
9      try:
10         nom_tag, begin, end = token.split_contents()
11     except ValueError:
12         msg = 'Le tag %s doit prendre exactement deux arguments.' %
               token.split_contents()[0]
13         raise template.TemplateSyntaxError(msg)
14
15     # Nous vérifions que nos deux paramètres sont bien des entiers
16     try:
17         begin, end = int(begin), int(end)
18     except ValueError:
19         msg =
               'Les arguments du tag %s sont obligatoirement des entiers.'
               % nom_tag
20         raise template.TemplateSyntaxError(msg)
21
22     # Nous vérifions si le premier est bien inférieur au second
23     if begin > end:
24         msg =
               'L\'argument "begin" doit obligatoirement être inférieur à l\'argu
               % nom_tag
25         raise template.TemplateSyntaxError(msg)
26
27     return RandomNode(begin, end)
28
29
30 class RandomNode(template.Node):
31     def __init__(self, begin, end):
32         self.begin = begin
33         self.end = end
34
35     def render(self, context):
36         return str(randint(self.begin, self.end))
```

Listing 95 – Notre tag random

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'`.

FIGURE 14.4. – Erreur 500 lorsque le tag est mal utilisé

14.4.2. Passage de variable dans notre tag

```
{% url %}
```

```
{% random %}
```

```
1 def ma_vue(request):
2     return render(request, 'template.html', {'begin': 1, 'end':
          42})
```

Listing 96 – views.py

```
1 {% random begin end %}
```

Listing 97 – templates/template.html

```
begin    end
```

```
@register.tag
def random(parser, token):

        """ Tag générant un nombre aléatoire, entre les bornes en arguments """
    try:
        nom_tag, begin, end = token.split_contents()
    except ValueError:
        msg =
            'Le tag random doit prendre exactement deux arguments.'
        raise template.TemplateSyntaxError(msg)

    return RandomNode(begin, end)
```

render RandomNode

template Variable

```
from django.template.base import VariableDoesNotExist

class RandomNode(template.Node):
    def __init__(self, begin, end):
        self.begin = begin
        self.end = end

    def render(self, context):
        not_exist = False

        try:
            begin = template.Variable(self.begin).resolve(context)
            self.begin = int(begin)
        except (VariableDoesNotExist, ValueError):
            not_exist = self.begin
        try:
            end = template.Variable(self.end).resolve(context)
            self.end = int(end)
        except (VariableDoesNotExist, ValueError):
            not_exist = self.end

        if not_exist:
            msg =
                'L\'argument "%s" n\'existe pas, ou n\'est pas un entier.'
                % not_exist
            raise template.TemplateSyntaxError(msg)

```

```
26          # Nous vérifions si le premier entier est bien inférieur au
                second
27          if self.begin > self.end:
28              msg =
                    'L\'argument "begin" doit obligatoirement être inférieur à l\'a
29              raise template.TemplateSyntaxError(msg)
30
31          return str(randint(self.begin, self.end))
```

__init__

render() template

{% random 1 10 %} VariableDoesNotExist

```
{% random 0 42 %}
{% random a b %} avec a = 0 et b = 42
{% random a 42 %}
```

```
{% random a 42 %} avec a = "Bonjour"
{% random a 42 %} où a n'existe pas
```

14.4.3. Les *simple tags*

random

```
@register.simple_tag(name='random')  # L'argument name est encore
    une fois facultatif
def random(begin, end):
    try:
        return randint(int(begin), int(end))
    except ValueError:
```

```
6           raise
                template.TemplateSyntaxError('Les arguments doivent nécessairement é
```

```
1 @register.simple_tag(takes_context=True)
2 def random(context, begin, end):
3     # …
```

?

Pourquoi avoir fait toute cette partie si au final nous pouvons faire un tag en moins de lignes, et plus simplement ?

`cycle`

`SimpleNode` `render`

`random`

—

`{% if %} {% endif %}`

—

—

14.4.4. Quelques points à ne pas négliger

`escape`

`render`

la documentation officielle

14.5. En résumé

—

—

```
{% load nom_module %}
```

—

—

```
TEM
```

```
PLATE_CONTEXT_PROCESSORS
```

—

15. Les signaux et middlewares

15.1. Notifiez avec les signaux

```
from django.models.signals import post_delete

post_delete.connect(ma_fonction_de_suppression, sender=MonModele)
```

connect

post_delete

ma_fonc

tion_de_suppression

sender

MonModele

sender

post_delete sender

post_delete

— sender
— instance

— using

ma_fonction_de_suppression

```
def ma_fonction_de_suppression(sender, instance, **kwargs):
        # processus de suppression selon les données fournies par
            instance
```

?

Pourquoi spécifier un `**kwargs` ?

?

Où faut-il mettre l'enregistrement des signaux ?

models.py urls.py

models.py

```
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
```

`django.db.models.signals.pre_save`

— `sender`

— `instance`

— `using`

— `raw` `True`

`django.db.models.signals.post_save`

— `sender`

— `instance`

— `using`

— `raw` `True`

— `created` `True`

`django.db.models.signals.pre_delete`

— `sender`

— `instance`

— `using`

`django.db.models.signals.post_delete`

— `sender`

— `instance`

— `using`

`django.core.signals.request_started` — `sender`

`django.core.handlers.wsgi.WsgiHandler`

`django.core.signals.request_finished` — `sender`

`django.core.handlers.wsgi.WsgiHandler`

https ://docs.djangoproject.com/en/1.4/ref/signals/

`print`

`django.dispatch.Signal`

```
import django.dispatch

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

`crepe_finie`

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

`send`

```
class Crepe(models.Model):
        nom_recette = models.CharField(max_length=255)
        prix = models.IntegerField()
```

```
4         #d'autres attributs
5
6         def preparer(self, adresse):
7                 # Nous préparons la crêpe pour l'expédier à
                      l'adresse transmise
8                 crepe_finie.send(sender=self, adresse=adresse,
                      prix=self.prix)
```

preparer faire_livraison sender send send (receveur, retour) faire_livraison crepe_finie True send [(faire_livraison, True)] disconnect connect

```
1 crepe_finie.disconnect(faire_livraison)
```

crepe_finie faire_livraison sender

15.2. Contrôlez tout avec les middlewares

```
1 MIDDLEWARE_CLASSES = (
2     'django.contrib.sessions.middleware.SessionMiddleware',
```

```
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
        'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
```

settings.py

— process_request(self, request)
request HttpRequest

— process_view(self, request, view_func, view_args, view_kwargs)
view_func
view_args view_kwargs

— process_template_response(self,request,response)
TemplateResponse response HttpResponse

— process_response(self, request, response)

— process_exception(self, request, exception)
exception Exception
None HttpResponse

HttpResponse

HttpResponse

process_response

HttpResponse

request process_request HttpRequest

setting.py
process_request process_view

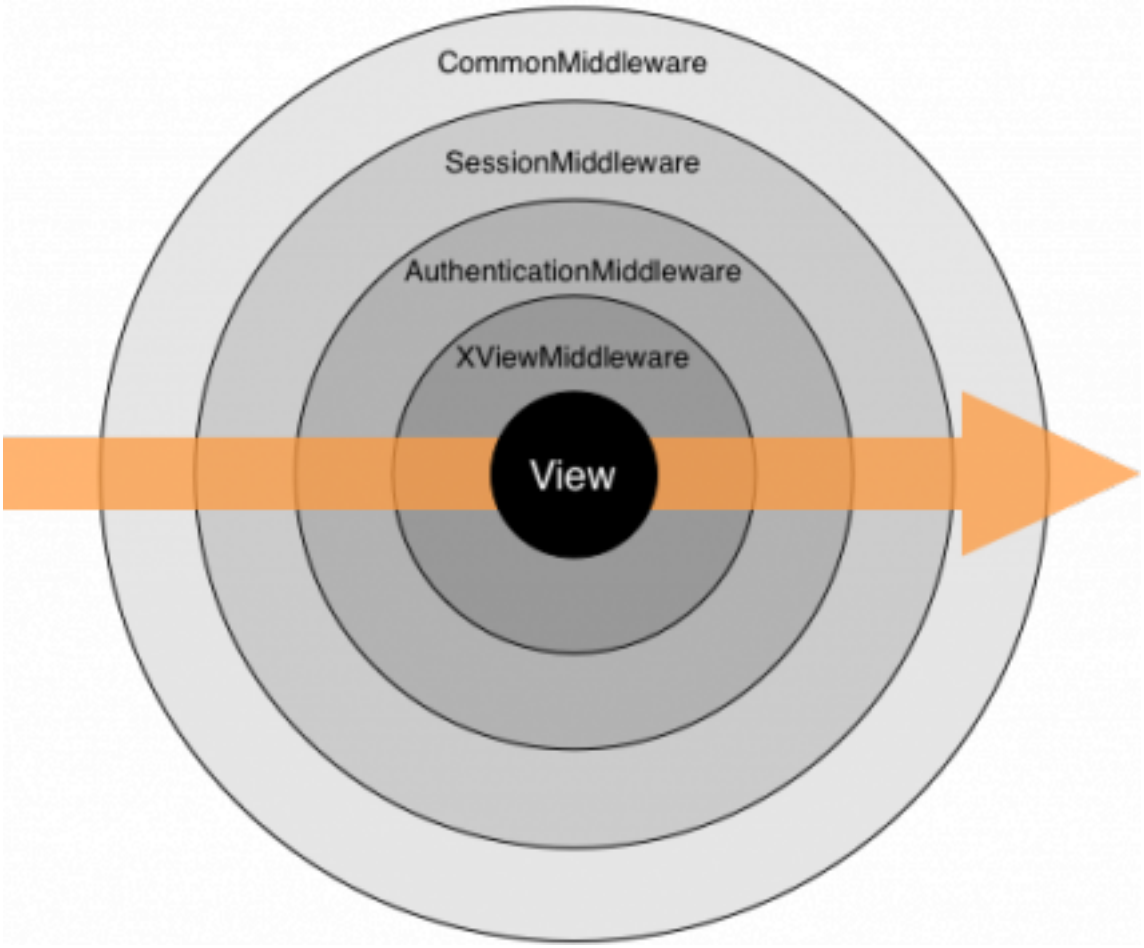


FIGURE 15.1. – Ordre d'exécution des middlewares

settings.py manage.py syncdb
stats/middleware.py

```

```

`MIDDLEWARE_CLASSES` `settings.py`

```

```

`pro`
`cess_view`

`request.path`
`HttpRequest`

`response.content`
`HttpResponse`

15.3. En résumé

—

—

—

`MIDDLEWARE_CLASSES`

—

Quatrième partie

Des outils supplémentaires

16. Les utilisateurs

16.1. Commençons par la base

'django.contrib.auth' 'django.contrib.contenttypes'
INSTALLED_APPS settings.py

- 'django.contrib.sessions.middleware.SessionMiddleware'
- 'django.contrib.auth.middleware.AuthenticationMiddleware'

16.1.1. L'utilisateur

django.contrib.auth.models.User

User
- username

- first_name
- last_name
- email
- password

- is_staff

- is_active True False

ForeignKey
- is_superuser True

- last_login datetime

- date_joined datetime
- user_permissions ManyToMany

— `groups` `ManyToMany`

`create_user`

Listing 98 – Créer un utilisateur

`maxime@crepes-bretonnes.com`

`password`

16.1.2. Les mots de passe

`algorithme$iterations$sel$empreinte`

— `pbkdf2_sha256`

—

—

cRu9mKvGzMzW

—

user.password

password

User

— set_password(mot_de_passe)

.save()

— check_password(mot_de_passe)

True

False

— set_unusable_password()

check_password

False

— has_usable_password()

True

False

set_unusable_password

16.1.3. Étendre le modèle User

User

User

User

User

OneToOneField

blog/models.py

Profil

User

Profil

User

OneToOneField

16.2. Passons aux vues

16.2.1. La connexion

Listing 99 – Notre formulaire de connexion (`blog/forms.py`)

Listing 100 – Notre template de connexion (`templates/login.html`)

1.
2.
3.
4.
5.

`authenticate` `login` `django.contrib.auth`

— `authenticate(username=nom, password=mdp)`
`authenticate` `User`
`None`

— `login(request, user)`
`HttpRequest` `User`

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.

Listing 101 – La vue de connexion (`blog/views.py`)
`crepes_bretonnes/urls.py`

`/connexion`

`manage.py createsuperuser`

16.2.2. La déconnexion

`logout` `django.contrib.auth`

Listing 102 – Notre vue de déconnexion (`blog/views.py`)

```
```

16.2.3. Intéragir avec le profil utilisateur

```
```

`django.contrib.auth.decorators`

```
```

Listing 103 – une vue qui nécessite de s'authentifier.

`LOGIN_URL` `settings.py` `'/accounts/login/'` `'/connexion/'`

```
```

`'/connexion/'` `'/connexion/?next=/bonjour/'` `GET`

`redirect_field_name`

```
```

`LOGIN_URL` `settings.py`

```
```

— `user_logged_in` `request` `user`

— `user_logged_out` `request` `user`

— `user_login_failed`

16.3. Les vues génériques

`django.contrib.auth`

?

Pourquoi alors nous avoir expliqué comment gérer manuellement la (dé)connexion ?

16.3.1. Se connecter

`django.contrib.auth.views.login`

— `template_name` `registration/login.html`

— `form`

— `next`

`settings.LOGIN_REDIRECT_URL` `next`

16.3.2. Se déconnecter

`django.contrib.auth.views.logout`

— `next_page`

— `template_name` `registration/logged_out.html`

— `redirect_field_name`

— `title`

16.3.3. Se déconnecter puis se connecter

`django.contrib.auth.views.logout_then_login`

— `login_url` `settings.LOGIN_URL`

16.3.4. Changer le mot de passe

`django.contrib.auth.views.password_change`

— `template_name` `registration/password_change_form.html`

— `post_change_redirect`

— `password_change_form`

— `form`

16.3.5. Confirmation du changement de mot de passe

`django.contrib.auth.views.password_change_done`

— `template_name` `registration/password_change_done.html`

`django.contrib.auth.views.password_change`

16.3.6. Demande de réinitialisation du mot de passe

`django.contrib.auth.views.password_reset`

— `template_name` `registration/password_reset_form.html`

— `email_template_name` `registration/password_reset_email.html`

— `subject_template_name` `registration/password_reset_subject.txt`

— `password_reset_form`

— `post_reset_direct`

— from_email
settings.DEFAULT_FROM_EMAIL

— form

— user
— email user.email
— domain re
quest.get_host()
— protocol http https
— uid
— token token

16.3.7. Confirmation de demande de réinitialisation du mot de passe

django.contrib.auth.views.password_reset_done
— template_name registration/pass
word_reset_done.html

django.contrib.auth.views.password_reset

16.3.8. Réinitialiser le mot de passe

django.contrib.auth.views.password_reset_confirm
— template_name registration/pass
word_reset_confirm.html
— set_password_form
— post_reset_redirect

— form
— validlink True

16.3.9. Confirmation de la réinitialisation du mot de passe

django.contrib.auth.views.password_reset_complete

— `template_name` `registration/password_reset_complete.html`

`django.contrib.auth.views.password_reset_confirm`

16.4. Les permissions et les groupes

16.4.1. Les permissions

`nom_application.nom_permission`

`Article`

`blog`

— `blog.add_article`
— `blog.change_article`
— `blog.delete_article`

`Meta`

`Article`

```
```

`user.has_perm("blog.commenter_article")` `True` `False`

```
```

`login_required`

`django.contrib.auth.decorators.permission_required`

```
```

`Permission` `django.contrib.auth.models`

— `name`

— `content_type` `content_type`
— `codename`

```
```

```
```

`user_permissions` `ManyToMany`

16.4.2. Les groupes

`django.contrib.auth.models.Group`

— `name`
— `permissions` `ManyToMany` `user_permissions`

`ManyToMany` `groups` `User`

```
```

16.5. En résumé

—

`make_password` `check_password`

—
—

— `@login_required`

17. Les messages

17.1. Les bases

settings.py
— MIDDLEWARE_CLASSES 'django.contrib.messages.middleware.MessageMiddleware'
— INSTALLED_APPS 'django.contrib.messages'
— TEMPLATE_CONTEXT_PROCESSORS
settings.py 'django.contrib.messages.context_processors.messages'

— DEBUG
DEBUG=True settings.py
— INFO
— SUCCESS
— WARNING
— ERROR

INFO
request HttpRequest

`messages`

`context_processors`

`messages`

`tags`

`{% extends %}`

17.2. Dans les détails

- `DEBUG`
- `INFO`
- `SUCCESS`
- `WARNING`
- `ERROR`

`settings.py`

`MESSAGE_LEVEL`

`DEBUG` `INFO`

`messages.DEBUG`

`settings.py`

`SUCCESS`

```

```

17.3. En résumé

—

—

—

—

`messages.set_level`

18. La mise en cache

Un tutoriel complet à propos de la mise en cache existe ici . Il vous donnera des informations plus poussées pour améliorer les performances de votre application django.

18.1. Cachez-vous !

SQL

CACHES settings.py

18.1.1. Dans des fichiers

'LOCATION'

'LOCATION' 'c:/mon/dossier'

c:/ 'BACKEND'

pickle

18.1.2. Dans la mémoire

```
```

`'LOCATION'`

`'LOCATION'`

18.1.3. Dans la base de données

`manage.py`

```
```

`nom_table_cache`

`settings.py`

```
```

18.1.4. En utilisant Memcached

`apt-get install memcached`

`memcached -d -m 512 -l 127.0.0.1 -p 11211` `-d` `-m`

18.1.5. Pour le développement

?

Au final, quel système choisir ?

18.2. Quand les données jouent à cache-cache

18.2.1. Cache par vue

Listing 104 – Une vue mise en cache pour 15 minutes

`/article/42` `/article/1337`

`URLconf`

```
```

`@cache_page`

18.2.2. Dans les templates

`{% cache %}` `{% load cache %}`
`cache`

```
```

`carrousel`

```
```

18.2.3. La mise en cache de bas niveau

`cache`
`django.core.cache` `cache`

```
```

`'ma_cle'` `'Coucou !'`

`get`

`None`

```
```

```
```

`add`

```
```

`set_many`
`get_many`

```
```

```
```

```
```

`clear`

```
```

`incr` `decr`

```
```

consultez la documentation

18.3. En résumé

—

—

— `@cache_page`

`urls.py` `{% cache %}`

—

19. La pagination

19.1. Exerçons-nous en console

`QuerySet`

`count` `__len__`

`python manage.py shell`

`Paginator`

`django.core.paginator`

```

```

```

```

`Paginator`

```

```

`Paginator`

```

```

`page()`

`Page`

```

```

```

```

`Paginator` `Paginator`

`orphans`

`Gauteng`

`Teheran` `Pekin`

`allow_empty_first_page`

19.2. Utilisation concrète dans une vue

`page`

— `page` `GET` `/url/?page=1`

— `/url/1`

`request.GET.get('page')`

`page`

Listing 105 – mini_url/views.py

`urls.py`

Listing 106 – mini_url/urls.py

`Paginator`

`Paginator`

```

```

`http://127.0.0.1:8000/m/`

`http://127.0.0.1:8000/m/2`

```

```

`has_next` `has_previous`

`num_pages`

`Paginator`

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" %}`.

`Paginator`

```

```

19.3. En résumé

— `django.core.paginator.Paginator`

— `Paginator` `p.num_pages` `p.page()` `Page` `has_next()` `has_previous()`

— `orphans` `Paginator`

—

20. L'internationalisation

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

20.1.0.1. Sous Mac OS X

télécharger le code source

20.1.0.2. Sous Linux

20.1.0.3. Sous Windows

1. depuis le serveur GNOME un de ses miroirs
   `gettext-runtime-X.zip`
   — `gettext-tools-X.zip`
2. `\bin`
   `C:\Program Files\gettext\bin`
3. `PATH`
   `;C:\Program Files\gettext\bin`

   `xgettext --version`

1.

2.

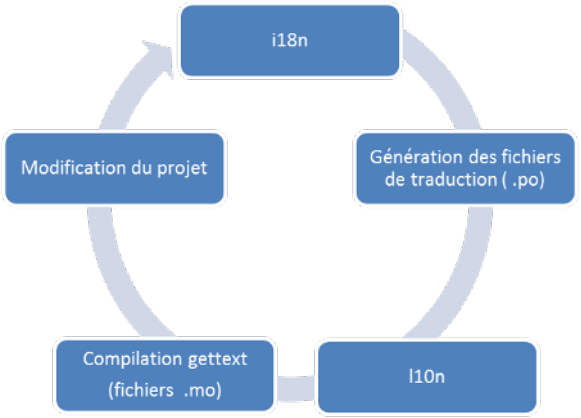


FIGURE 20.1. – Cycle de traduction d'un logiciel

settings.py

LANGUAGE_CODE

USE_I18N

False

USE_L10N

{% now "DATETIME_FORMAT" %}

USE_L10N LANGUAGE_CODE

LANGUAGE_CODE USE_L10N False True

```

```

`gettext`

`django.utils.translation`

`settings.py`

```

```

1.
2.
3. `_language`
4. `Accept-Language`
5. `LANGUAGE_CODE`

```

```

20.2. Traduire les chaînes dans nos vues et modèles

— `gettext` `ugettext`
— `gettext_lazy` `ugettext_lazy`
— `ngettext` `ungettext`
— `ngettext_lazy` `ungettext_lazy`
—

```

```

`test_i18n.html`

```

```

Listing 107 – template `test_i18n.html`

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.

`ugettext`

```

```

`ugettext(…)`

```
```

```
```

`gettext`

`nb_chats`

`gettext` `ungettext`

```
```

`nb_chats`

```
```

`%(nb)s` `%(color)s`

```
```

`Translators`

```
```

homographes

pgettext

20.2.1. Cas des modèles

uget

text_lazy gettext

reverse_lazy

pgettext

ungettext

_lazy

ugettext_lazy _

models.py mini_url

20.3. Traduire les chaînes dans nos templates

django.utils.translation

{% load i18n %}

{% trans %} {% blocktrans %}

20.3.1. Le tag {% trans %}

ugettext()

{% trans ma_variable %}

```
                                                                                {%
trans %}
```

```

```

20.3.2. Le tag {% blocktrans %}

{% trans %} {% blocktrans %} {% endblocktrans %}

```

```

with

```

```

ungettext {% blocktrans %}

—

count nombre=ma_variable nombre ma_variable

—

{% plural %}

nb

```

with

```

```

{% trans %} {% blocktrans %}

```

```

```

20.3.3. Aidez les traducteurs en laissant des notes !

Transla

tors: {# #} {% comment %}

20.4. Sortez vos dictionnaires, place à la traduction !

20.4.1. Génération des fichiers .po

.po

msgid

msgstr

.po

locale

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

settings.py

locale

.po manage.py
locale

```
```

locale/en/LC_MESSAGES
django.po

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

.po

```
```

msgid msgid

msgid
en msgstr
msgid ""

```
```

.po

```
```

msgstr[0]
msgid_plurial
.po

20.4.2. Génération des fichiers .mo

.po

.mo

makemessages

```
```

.mo

.po

20.5. Le changement de langue

django.utils.translation.activate

```
```

urls.py

```
```

```
```

Listing 108 – Le template de la page de choix d'une langue

next

/

20.6. En résumé

—

—

—

—

—

21. Les tests unitaires

21.1. Nos premiers tests

?

Pourquoi faire des tests unitaires ?

21.1.1. Ecrire un test unitaire

Article
est_recent
True
False

```

```

tests.py

Listing 109 – Le teste de notre cas anormal

Listing 110 – assertRaises

21.1.2. Lançons notre test en console

est_recent False True

python manage.py test

21.1.3. Initialisation de données pour nos tests

manage.py dumpdata blog fixtures/test.json

setUp

21.2. Testons des vues

mini_url

à reprendre le code

mini_url/tests.py

creer_url

test_liste

views.liste

creer_url

liste

get

self.client

reverse

get

- status_code
- content
- context

self.assertEqual(reponse.status_code, 200)

assertContains

get

QuerySet minis

re

ponse.context

QuerySet

QuerySet

repr

repr(premier_argument) ==

deuxieme_argument

['<MiniURL: [ALSWM0] http://foo.bar>']

post

self.client

nouveau

```
post                    get

                        assertRedirects

nouveau                                 liste

follow=True   get    post
```

Listing 111 – Teste d'une vue POST

```
```

Listing 112 – Se connecter avec un login et un mot de passe pour les tests.

21.3. En résumé

—

—

—

—

22. Ouverture vers de nouveaux horizons : django.contrib

22.1. Vers l'infini et au-delà

`django.contrib`

`django.contrib`
`django.contrib`

`django.contrib`

`admin`

`admindocs`

`auth`

`comments`

`contenttypes`

`flatpages`

`formtools`

`gis`

http ://www.geodjango.org

`humanize`

`messages`

`redirects`

`sessions`

`sitemaps`

`sites`

`staticfiles`

`syndication`

`webdesign`

`{% lorem [nb] [method] [random] %}`

`flatpages` `humanize`

la documentation officielle

22.2. Dynamisons nos pages statiques avec flatpages !

`TemplateView`

`flatpage` `flatpage`

22.2.1. Installation du module

`flatpages`
`settings.py`

— `'django.contrib.sites'` `'django.contrib.flatpages'`
`INSTALLED_APPS`

— `settings.py` `SITE_ID`

`SITE_ID = 1`

— `python manage.py migrate`

`flatpage`

22.2.1.1. Le cas des URL explicites

`urls.py`
`flatpages`
`/pages/`

```
```

`flatpage`
`urls.py`

```
```

22.2.1.2. Utilisation du middleware FlatpageFallbackMiddleware

`FlatpageFallbackMiddleware`

`process_view()`

`FlatpageFallback`

`flatpage`

`'django.contrib.flatpages.middleware.FlatpageFallbackMiddleware'`
`MIDDLEWARE_CLASSES`

Quelle que soit la méthode choisie, la suite de ce cours ne change pas.

22.2.2. Gestion et affichage des pages

`flatpages`

FIGURE 22.1. – Le module « flatpages » dans l'administration de Django

`/contact/`

`flatpages`

`flatpages/default.html`

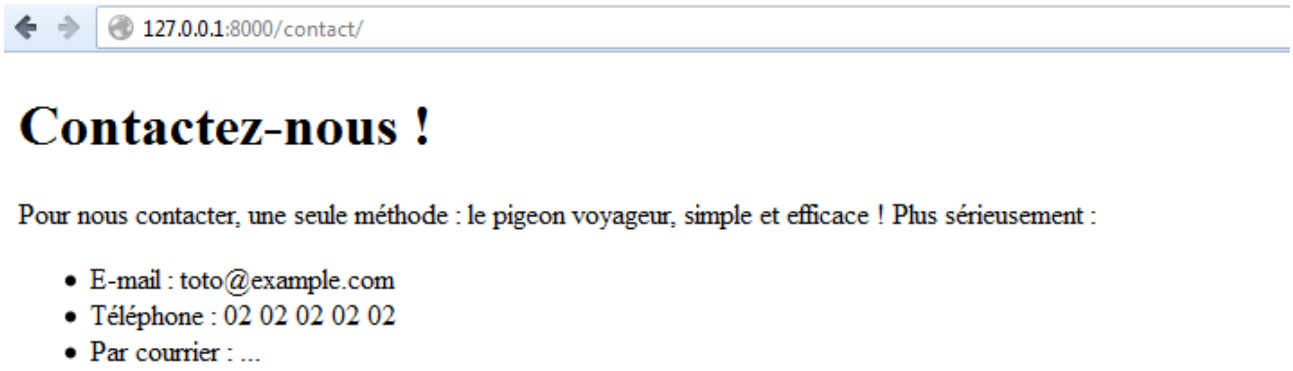


FIGURE 22.2. – Exemple de page de contact utilisant « Flatpages »

`content`

—

`django.contrib.auth`

— `flatpages/default.html`

22.2.3. Lister les pages statiques disponibles

`flatpages`

`{% load flatpages %}`

Listing 113 – Affiche la liste des pages statiques de notre application

for

```
user
{% get_flatpages as flatpages %}
flatpages
for as
```

22.3. Rendons nos données plus lisibles avec humanize

django.contrib.humanize

humanize

'django.contrib.humanize' INSTALLED_APPS settings.py

templatetags {% load humanize %}

22.3.1. apnumber

Associated Press

22.3.2. intcomma

settings.py

22.3.3. `intword`

Listing 114 – Les grands nombres avec `intword`

22.3.4. `naturalday`

Listing 115 – `views.py` Définitions de quelques dates grâce au module `datetime`

22.3.5. `naturaltime`

Listing 116 – `views.py` Définitions de quelques dates et heures grâce au module `datetime`

22.3.6. `ordinal`

`humanize`

22.4. En résumé

—

—

—

— `humanize`
`flatpages`

Cinquième partie

Annexes

23. Déployer votre application en production

23.1. Le déploiement

mod_wsgi

`WSGI`

23.1.1. Configuration du projet

`settings.py`

`DEBUG`

— `DEBUG` `False`

— `ALLOWED_HOSTS`

`ALLOWED_HOSTS = ['www.crepes-bre`
`tonnes.com', '.super-crepes.fr']`

`super-crepes.fr` `www.super-crepes.fr`
`mobile.super-crepes.fr`

—

— `SECRET_KEY` cet outil en ligne

23.1.2. Installation avec Apache2

```
```

Index of /

Name	Last modified	Size	Description
Parent Directory		-	
__init__.py	14-Jul-2012 10:22	0	
blog/	14-Jul-2012 10:22	-	
crepes_bretonnes/	14-Jul-2012 10:22	-	
db	14-Jul-2012 10:22	47K	
manage.py	14-Jul-2012 10:22	259	
templates/	14-Jul-2012 10:22	-	
urlshortener/	14-Jul-2012 10:22	-	

FIGURE 23.1. – Une liste de fichiers Python que nous ne pouvons que télécharger

environnement virtuel (venv)

Listing 117 – /etc/apache2/sites-enabled/crepes_bretonnes.com

23.1.3. Installation avec nginx et gunicorn

Listing 118 – /home/votre_user/start_gunicorn.sh

Ctrl Z

une fois que le fonctionnement de base sera assuré, vous pourrez créer un service à mettre dans le dossier `/etc/init.d` et qui pourrait être relancé par `service crepe-bretonnes restart` par exemple.

Listing 119 – /etc/nginx/sites-available/crepes-bretonnes.com

`/etc/nginx/sites-available/crepes_bretonnes.com`

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.

23.2. Gardez un œil sur le projet

FIGURE 23.2. – Exemple de mail reçu en cas d'erreur

23.2.1. Activer l'envoi d'e-mails

False

ADMINS settings.py

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'`, par exemple.

23.2.2. Quelques options utiles...

23.2.2.1. Être avertis des pages 404

settings.py

CommonMiddleware MIDDLEWARE_CLASSES

MANAGERS

IGNORABLE_404_URLS

Listing 120 – Définir les page 404 courantes pour ne pas être averti.

.php phpmyadmin/

23.2.2.2. Filtrer les données sensibles

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…).

23.2.3. Introduction à Sentry, pour garder un oeil encore plus attentif

projet Sentry

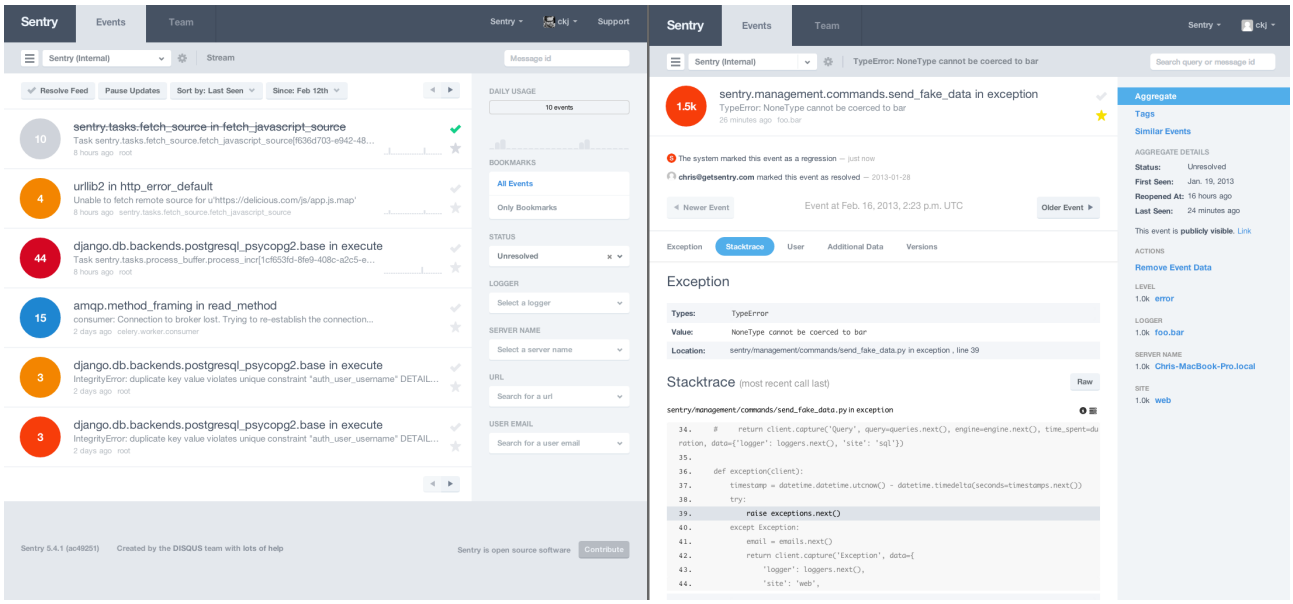


FIGURE 23.3. – Aperçu de quelques pages de Sentry

Zeste de savoir utilise son propre sentry pour que les développeurs soient immédiatement prévenus en cas de bug. Le canal IRC est immédiatement notifié.

23.3. Hébergeurs supportant Django

Alwaysdata

détails ici

Heroku

WebFaction

DjangoEurope

A2 Hosting

disponible ici

23.4. En résumé

—

—

—

24. L'utilitaire manage.py

`manage.py`

`manage.py`

24.1. Les commandes de base

24.1.1. Prérequis

`[...]`

`--`

24.1.2. Liste des commandes

24.1.2.1. `runserver` [port ou adresse :port]

`check`

```

```

```

```

```

```

`--ipv6, -6`

`--ipv6 ou -6`

```

```

`--noreload`

`--nothreading`

24.1.2.2. `shell`

24.1.2.3. `version`

```

```

24.1.2.4. help <commande>

`manage.py` `manage.py help <commande>`

24.1.2.5. 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.

`manage.py` `settings.py`

```
```

`/home/crepes/pro`
`jets/crepes`
`--template`

```
```

```
```

24.1.2.6. `startapp` <nom> [destination]

`models.py`
`views.py`

```
```

`--template` `startproject`

```
```

```
```

24.1.2.7. `diffsettings`

`settings.py`
`###`

24.1.2.8. `check`

—
— `admin.py`
—

24.1.2.9. `test` <application ou identifiant de test>

```
```

```
```

```
```

`--failfast`

24.1.2.10. `testserver` <fixture fixture ...>

`loaddata`

1.
2.
3.

`--addrport [port ou adresse:port]`
`runserver`

```
```

24.2. La gestion de la base de données

`--database`

`settings.py`

24.2.0.1. `makemigrations` [app1 app2...]

`models.py`

`--empty`

`--dry-run`

`--merge`

24.2.0.2. `migrate` [app1 [nom_migration]]

SQL

- `migrate`
- `migrate app`
- `migrate app migration`

`--fake`

SQL

`--list, -l`

24.2.0.3. dbshell

settings.py

—
—
—

24.2.0.4. dumpdata <application application.modele ...>

loaddata

testserver

Article Categorie blog
django.contrib.auth

--all
--format <fmt>
--format xml
--indent <nombre d'espace>

--exclude

blog blog.Article

--natural

ForeignKey ManyToMany
contrib.auth.Permission

ContentType

24.2.0.5. loaddata <fixture fixture ...>

dumpdata
loaddata

—
—
—

fixtures
FIXTURE_DIRS
settings.py

```
```

ma_fixture.json
fixtures

```
```

ma_fixture.xml
ma_fixture.json
ma_fixture.json
ma_fixture.json.zip
ma_fixture.json.gz
ma_fixture.json.bz2

24.2.0.6. inspectdb

settings.py
models.py

syncdb

class
while

TextField
ForeignKey
ManyToManyField
inspectdb
Article
blog

```
```

24.2.0.7. flush

syncdb
initial_data

24.2.0.8. sql <application application ...>

SQL

```
```

24.2.0.9. sqlcustom <application application ...>

SQL
`<application>/sql/<modele>.sql` `<application>`
`<modele>`
`blog` `Article` `manage.py sqlcustom blog`
`blog/sql/article.sql`

SQL
`dbshell`

```
```

24.2.0.10. sqlall <application application ...>

`sql` `sqlcustom`

24.2.0.11. sqlclear <application application ...>

SQL

24.2.0.12. sqlflush <application application ...>

SQL

SQL

`flush`

24.2.0.13. sqlindexes <application application ...>

SQL

24.2.0.14. sqlsequencereset <application application ...>

SQL

24.3. Les commandes d'applications

24.3.0.1. clearsessions

`django.contrib.sessions`

24.3.0.2. changepassword [pseudo]

```
```

`django.contrib.auth`

24.3.0.3. createsuperuser

`--username` `--email`

24.3.0.4. makemessages

`--all, -a`
`--extensions`

```
```

```
```

`--locale`

```
```

`--symlinks`
`--ignore, -i`

```
```

`blog`

`--no-wrap`

`--no-location`

24.3.0.5. compilemessages

`.po` `.mo`

`--locale`

```
```

24.3.0.6. createcachetable <nom de la table>

Liste des abréviations

mod_wsgi 229

SGBD 28 29

SQL 1 23 27 31 60 66 72 74 140 195 241 244