INFORMATIQUE
TECHNIQUE



Expert

Django

Développez vos applications web en Python

(fonctionnalités essentielles et bonnes pratiques)

En téléchargement

</> le code source

+ QUIZ

Version en ligne **OFFERTE !** pendant 1 an

Patrick SAMSON

eni

Expert IT

Django

Développez vos applications web en Python

(fonctionnalités essentielles et bonnes pratiques)

En téléchargement



le code source

Table des matières

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **EIDJAN** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

1. Public concerné 7
2. Prérequis 8
3. Objectifs 9
4. Organisation 10

Chapitre 1
Installation

1. Introduction 11
2. Installation de Python 12
   - 2.1 Python 2 ou 3 ? 12
   - 2.2 32 ou 64 bits ? 13
   - 2.3 Déroulé de l'installation 14
   - 2.4 Après l'installation 21
3. Installation d'un moteur de base de données 23
   - 3.1 Installation de PostgreSQL 24
   - 3.2 Installation du pilote 29
4. Installation d'un outillage pour traduction 31
5. Installation d'un gestionnaire de fuseaux horaires 33
6. Installation de Django 34

Chapitre 2
Création de site

1. Objectifs 37
2. Création d'un projet 37
3. Premier lancement du site 40
4. Création d'une première application 45
   4.1 Emplacement d'une application 46
   4.2 Création 49
5. Paramètres de configuration 52
   5.1 Configuration minimale 52
   5.2 Configuration par défaut affinée 55
6. Variations de configuration 61
7. Création de l'application 69
8. Outillage de mise au point 70

Chapitre 3
Routage

1. Présentation 73
2. Configuration des adresses 75
3. Espace de noms 78
4. Instances multiples d'une même application 81

Chapitre 4
Modèles

1. Introduction 85
2. Instanciation de la base de données 86
   2.1 Création de la base de données 86
   2.2 Référencement de la base de données 88

2.3 Première alimentation de la base de données 90
2.4 Création d'un super-utilisateur 93
3. Champs 95
4. Métadonnées 102
5. ORM (Object Relation Mapping) et migrations 105
5.1 Création d'une migration initiale 106
5.2 Exécution d'une migration initiale 108
6. Exploration des métadonnées 109
7. Gestionnaires 114
8. Opérations sur objets 117
8.1 Création 118
8.2 Mise à jour 123
8.3 Lecture 126
8.4 Suppression 128
8.5 Optimisations 130
8.6 Opérations de masse 132
8.6.1 Création 133
8.6.2 Lecture 134
8.6.3 Chargement d'instantané 134
8.6.4 Quelques usages des instantanés 139
9. Migration de structures et données 150

Chapitre 5
Traces et journalisation

1. Requêtes à la base de données 153
1.1 Plan d'exécution 153
1.2 Constitution du code SQL 160
1.3 Journalisation des requêtes SQL 163
1.3.1 Observations manuelles 163
1.3.2 Observations automatiques 166

2. Pose de traces personnalisées 171
2.1 Noms des enregistreurs 172
2.2 Exemple d'enregistreur 172
2.3 Autres exemples de gestionnaires 177
2.3.1 Rotation de journaux 177
2.3.2 Transmission vers une machine distante 179

Chapitre 6
Intergiciels

1. Introduction 183
2. Création d'une application dédiée à l'outillage 184
3. Implémentation d'un intergiciel 185
3.1 Mise en place du cadre 185
3.2 Écriture d'un traitement 186
3.3 Alternance de mise en/hors service 189

Chapitre 7
Vues

1. Fonctions ou classes 191
2. Vues intégrées 193
2.1 Vues de base 193
2.2 Vues génériques 194
3. Greffons 195
4. Données de contexte 197
5. Processeurs de contexte 200
6. Requêtes AJAX 212
6.1 Restriction stricte au mode AJAX 215
6.1.1 Par greffon seul 217
6.1.2 Par greffon et décorateur 217
7. Intégrations en modèles 219

8. Simulation d'authentification 221
9. Écritures des vues 224
9.1 Dossier d'arrivée 224
9.2 Dossier d'envoi 226
9.3 Factorisation des vues de dossier 227
9.4 Encore plus d'optimisation et d'intégration 232
9.5 Contrôle du cache 240
9.6 Lecture de message 242
9.7 Composition de message 251
9.8 Effacement de message 265

Chapitre 8
Pages et gabarits

1. Introduction 275
2. Moteurs 276
2.1 Moteurs intégrés 276
2.2 Moteurs personnalisés 278
2.3 Sélection du moteur 279
3. Principes de fonctionnement 281
4. Pages non dynamiques 282
4.1 Application flatpages 282
4.1.1 Exemple d'usage 284
4.2 Fichiers statiques 290
4.2.1 Déploiement en production 297
4.2.2 Simulations en développement 302
5. Structuration des pages 304
6. Expérimentation rapide et manuelle 304
7. Écritures des gabarits 305
7.1 Base du site 305
7.2 Bases de l'application 310
7.3 Dossier d'arrivée 315

7.4 Dossier d'envoi 317
7.5 Lecture de message 317
7.6 Composition de message 318
7.7 Effacement de message 324
8. Composants de gabarit personnalisés 327
8.1 Balise de gabarit 327
8.2 Filtre 331

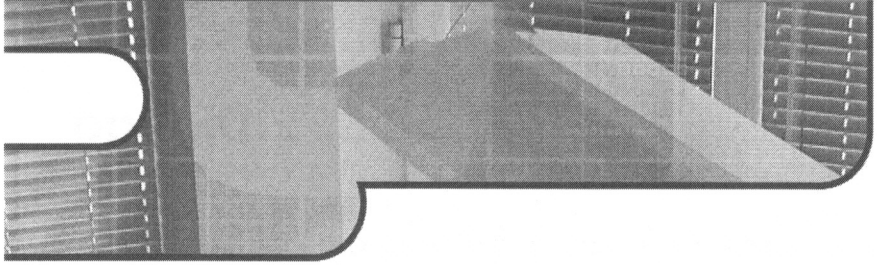
Chapitre 9
Alternatives

1. Propos 335
2. Alternance entre plusieurs versions de Django 335
2.1 Par configuration propre au site 336
2.1.1 Apport d'une ancienne version 338
2.1.2 Apport d'une nouvelle version 339
2.1.3 Emploi d'une instance déjà présente 341
2.2 Par environnement virtuel 342
3. Variations de configurations 348
3.1 Superposition en profondeur 348
3.2 Exécution de code 352

Chapitre 10
Internationalisation

1. Propos 355
2. Configuration 357
3. Détermination de la langue de l'utilisateur 361
4. Limitation de la quantité de langues supportées 363

Index 369



Avant-propos

1. Public concerné

Ce livre s'adresse à une audience volontairement large. De par la nature technique du produit, en tant qu'infrastructure logicielle pour construire des services web, les développeurs forment de toute évidence une catégorie ciblée en particulier. À ceux pour qui le produit est une totale découverte, il est donné toutes les procédures pour se bâtir localement une installation complète et effective du produit, de façon à ne pas se contenter d'une simple lecture passive, mais à effectuer de véritables expérimentations. Faire des essais, se confronter à ses erreurs et les rectifier est un excellent moyen pour prendre connaissance du sujet et pour le mémoriser. Un utilisateur plus averti, bien qu'ayant déjà manipulé le produit, trouvera certainement des thèmes nouveaux, d'autres manières d'aborder des points connus, ou des confirmations d'intuitions. Un développeur déjà expérimenté sur d'autres infrastructures, notamment basées sur d'autres langages de programmation (PHP, Java, JavaScript par exemple) pourra établir des comparaisons, en particulier sur la puissance du produit, sa facilité et rapidité de mise en œuvre initiale, ainsi que sur sa richesse fonctionnelle lui permettant de répondre à des besoins pouvant allant jusqu'à des niveaux de complexité élevés.

Même pour un lecteur plus éloigné de la technique, il sera intéressant de constater combien l'emploi d'une infrastructure, dès lors qu'elle est bien conçue, documentée, gouvernée dans ses objectifs initiaux et futurs, est un facteur de productivité, tout en laissant libre champ à la créativité du développeur pour imaginer et implémenter les spécificités de ses besoins métier.

2. Prérequis

Django est écrit dans le langage **Python**. Une connaissance préalable de ce langage est donc un atout certain, puisque les explications se concentrent sur le produit. Pour autant, un développeur habitué à d'autres langages saura faire les rapprochements nécessaires avec ses connaissances dans la mesure où on ne mentionne que des concepts classiques de classes, d'objets, de listes ou tableaux et d'instructions assez ordinaires.

Une connaissance minimale du langage **SQL** permettra d'apprécier pleinement les relations établies entre un serveur applicatif et un gestionnaire de bases de données.

S'agissant de serveur web, le trio **HTML-CSS-JS** apparaît naturellement, mais avec une dose minimaliste, car il s'agit juste de disposer de pages en état de fonctionner. Toute préoccupation de nature esthétique des rendus graphiques est volontairement écartée.

Des techniques courantes dans le domaine du développement web sont supposées connues, par exemple celles dénommées par ces sigles : AJAX, API, JSON, XML.

Le système d'exploitation utilisé pour les illustrations est **Windows 10**. Il ne s'agit nullement d'un impératif et des opérations équivalentes peuvent se réaliser sous d'autres systèmes d'exploitation, dès lors que les produits employés y sont supportés.

© Editions ENI - All rights reserved

3. Objectifs

On peut identifier deux objectifs principaux visés par ce livre. Le premier objectif est de prendre totalement en main le lecteur dans sa découverte du produit, avec la mise en place complète et effective d'un environnement de développement et la construction des pièces élémentaires d'un site. Le second objectif s'emploie à faire partager de bonnes pratiques, d'une part en matière de structuration du projet et d'autre part en compréhension et en maîtrise du produit.

Pour satisfaire ces objectifs, il a été choisi de se concentrer sur la mise en œuvre d'un cas concret : une application de messagerie interne entre les utilisateurs d'un site. Si ce cas d'usage est suffisamment représentatif pour aborder de nombreux sujets techniques du produit, il est bien entendu que tous ne peuvent pas être balayés.

Le livre ne prétend pas être exhaustif sur toutes les capacités offertes par le produit. Une première raison est qu'il ne devrait pas être dans la vocation d'un livre de se substituer à la documentation officielle, au risque de n'en être qu'un doublon. En effet, la qualité et l'abondance de cette documentation sont telles qu'il n'y a aucun intérêt à une redite. C'est pourquoi il ne faut pas s'attendre à trouver un déballage des caractéristiques détaillées de la syntaxe et des options à l'occasion de l'introduction d'un élément du produit. Une deuxième raison est que ce livre n'aspire pas à être un catalogue de toutes les fonctionnalités. Elles sont nombreuses et présentent naturellement des distinctions en niveaux de popularité, de complexité, de réponses à des usages, de variantes et de vieillissement. Il est donc délibérément choisi de se focaliser sur un certain nombre de fonctionnalités incontournables, de façon à manipuler un cas réel et cohérent. D'autres thèmes, qui ne sont pas naturels avec le fil conducteur adopté ou n'introduisant pas une valeur ajoutée suffisante, sont par conséquent omis.

4. Organisation

La rédaction du livre est orientée pour que le lecteur ne se limite pas à une prise de connaissance théorique, mais pour qu'il fasse des travaux pratiques sur une réelle mise en œuvre du produit. Ceci augmente l'acquisition des connaissances et donne la liberté de pousser plus loin l'exploration de certains points d'intérêt personnel ou d'apporter des réponses immédiates à des interrogations.

Un premier chapitre d'installation donne tous les outils nécessaires pour disposer d'un environnement de travail sur sa machine, en toute autonomie.

Le chapitre suivant établit les fondations pour la construction d'un site. On y découvre quelle est la structure attendue, les outils pour la bâtir, ainsi que les possibilités de personnalisation et de configuration.

Le chapitre consacré au routage est l'étape qui va permettre de mettre en fonctionnement une première page sommaire afin d'avoir la satisfaction de constater le bon fonctionnement immédiat de son site.

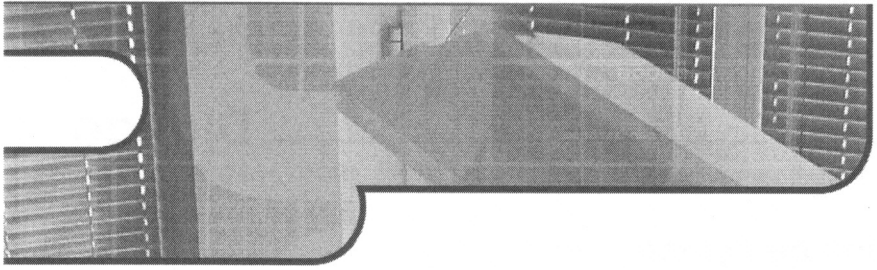
Vient ensuite un chapitre dédié aux modèles de données, le cœur essentiel pour un site applicatif dynamique, dont dérivent de nombreux autres éléments.

La disponibilité de contenu dans la base de données fournit alors de la matière pour procéder à de l'exploration, de l'observation et de la manipulation. Deux chapitres sont consacrés à ces thèmes : l'un pour donner des outils utiles à la visibilité et à la compréhension des traitements déclenchés, l'autre pour démontrer l'injection des traitements, altérants ou neutres, dans le chemin de service d'une requête.

Le vaste sujet des vues, pages et gabarits est traité dans deux chapitres, toujours avec une démarche progressive de construction, avec du code basique pour commencer et des optimisations dans un second temps.

Sur la fin, un chapitre revient sur des points abordés, mais avec des techniques plus élaborées et donc qu'il valait mieux présenter de façon différée et pour des lecteurs plus connaisseurs. Enfin, le dernier chapitre est dédié à l'internationalisation d'un site.

© Editions ENI - All rights reserved



Chapitre 1
Installation

1. Introduction

Django est une infrastructure logicielle très fournie, mais ce n'est pas un produit autosuffisant, dans le sens où, étant écrit en langage Python, il s'appuie sur l'existence préalable d'un interpréteur de ce langage sur la machine. Il faut donc installer cet interpréteur comme un prérequis.

Django a besoin d'un gestionnaire de base de données, ne serait-ce que pour son fonctionnement interne. Le gestionnaire SQLite fait partie de la liste des gestionnaires supportés et Python en contient une implémentation. Il est possible de le constater par la présence, sous le répertoire d'installation de Python, de : `DLLs\sqlite3.dll` et `Lib\sqlite3\`.

Comme son nom l'indique, ce gestionnaire se veut léger, c'est à la fois sa force (il est courant dans l'informatique embarquée, Android par exemple) et sa faiblesse (il a des limites en fonctionnalités et en performances). Autre effet de la légèreté, il n'existe pas d'outil officiel si l'on veut s'épargner les manipulations en ligne de commande (et la connaissance de la syntaxe SQL) par l'utilisation d'une interface graphique. Il faut chercher parmi les outils de la communauté, dont beaucoup ont leur développement abandonné depuis de nombreuses années. Pour la construction du site de l'ouvrage, il sera donc préférable de mettre en place un gestionnaire plus élaboré et plus confortable.

2. Installation de Python

Pour installer Python, rendez-vous tout d'abord à l'adresse :
https://www.python.org/downloads/

2.1 Python 2 ou 3 ?

La série Python 3, le successeur de Python 2, est apparue fin 2008. En raison des changements non rétrocompatibles, Django est resté pendant longtemps lié à la série Python 2. Publiée début 2013, Django 1.5 a été la première version à introduire un fonctionnement possible sous Python 3 tout en continuant à fonctionner aussi en Python 2, avec la même base de code, car les concepteurs ont fait le choix d'une transition progressive plutôt qu'un saut incompatible. Une forte raison en faveur de cette stratégie provient de l'importance incontournable des applications communautaires qui viennent enrichir l'infrastructure logicielle et qui étaient amenées par nature à avoir chacune leur rythme et leur stratégie de migration.

La série Django 1.11 est la dernière à tourner sous Python 2. Fin 2017, la série Django 2 marque une rupture par son numéro majeur de version, mais ce n'est pas en raison d'une quantité inhabituelle de changements non rétrocompatibles. Il s'agit plutôt de symboliser deux événements marquants : d'une part, l'abandon du support de Python 2, avec l'allègement en conséquence de la base de code et, d'autre part, l'adoption d'un nouveau schéma de numérotation, inspiré par le concept de gestion sémantique de version.

Pendant cette phase de transition de Python, il suffisait qu'un paquet additionnel ne soit pas encore mis à niveau pour être le grain de sable qui empêche les montées de version de Django dans son projet. Django 1.11 est estampillée LTS (*Long Term Support*) signifiant qu'elle bénéficie d'un support à long terme, qui s'étend jusqu'en avril 2020 au moins.

Étant donné la quantité de paquets additionnels écrits pour l'infrastructure logicielle, il est inévitable, au cours de recherches, de tomber sur des paquets qui n'ont pas été adaptés à l'évolution. Il reste d'actualité de rester vigilant à ce sujet au moment de fixer son choix pour un paquet.

© Editions ENI - All rights reserved

Ici, pour un nouveau projet, la question du choix des versions de Python et de Django ne se pose plus : il convient de prendre les plus récentes.

2.2 32 ou 64 bits ?

Les distributions binaires laissent le choix entre une architecture 32 bits ou 64 bits. Sachant qu'un binaire 32 bits peut quand même fonctionner sur une machine 64 bits, mais pas l'inverse, l'éventuelle question résiduelle est de savoir quelles seraient les situations où cette combinaison non optimale aurait du sens.

À nouveau, il faut regarder en arrière et remonter dans le temps. Ce n'est pas le moteur Python qui est en cause, puisqu'un installateur dédié au 64 bits est apparu en version 2.4 en fin 2004. Il faut se tourner du côté des paquets additionnels qu'on est amené inévitablement à ajouter au cœur, pour remplir les besoins métier.

Un paquet additionnel n'est pas nécessairement écrit purement en Python. Il doit bien sûr présenter des points d'appels au sens Python pour se rendre compréhensible de l'interpréteur. Mais rien n'empêche que ce ne soit qu'un emballage autour d'un code natif écrit dans un autre langage. Un cas très répandu est d'avoir une bibliothèque écrite en C/C++ qu'on souhaite rendre accessible à partir d'un script Python. Mettre en place une jonction avec un autre langage est une tâche ardue et il est préférable d'utiliser pour cela des outils spécialisés, dont un exemple renommé est le produit SWIG (*Simplified Wrapper and Interface Generator* - http://www.swig.org/).

Dans ces cas hybrides, les difficultés commencent, car il faut disposer au final d'un package binaire, à partir du code source. Or, on n'a pas nécessairement à sa disposition tout l'outillage de compilation et la situation peut encore se complexifier avec une éventuelle cascade de dépendances à d'autres bibliothèques. Pour noircir encore plus le tableau, le résultat à produire doit être individualisé à la fois pour la version de Python et pour son architecture.

Par exemple, pour le même composant :

- `M2Crypto-0.31.0.win-amd64-py3.6.msi` est un binaire pour Python 3.6 et pour 64 bits.
- `M2Crypto-0.31.0.win32-py2.7.msi` est un binaire pour Python 2.7 et pour 32 bits.

Dans la situation la plus favorable, l'auteur du paquet met à disposition des distributions binaires en plus de la distribution source. Mais ce n'est pas toujours le cas et la quantité de combinaisons est parfois restreinte. En alternative ou en complément, avec un peu de bonheur, il peut se trouver que d'autres personnes motivées proposent aussi des distributions binaires. À nouveau, il n'est pas certain de trouver sa bonne combinaison, ou elle peut être obsolète.

On est donc potentiellement confronté au cas où un paquet additionnel n'est distribué en binaire que sous une architecture 32 bits, ce qui rend par conséquent nécessaire d'installer une version de Python également en 32 bits bien que la machine ait un système d'exploitation 64 bits. Heureusement, au fil du temps, ceci est de moins en moins le cas.

Un autre raisonnement a pu consister à se dire qu'il vaut mieux installer un Python 32 bits et qu'on aura ainsi plus de chances de trouver la distribution d'un paquet additionnel, considérant que le 32 bits est plus répandu que le 64 bits. Ce raisonnement était notamment valide à l'occasion d'un changement de matériel, de façon à ne pas déstabiliser un projet en cours. Mais la tendance est passée dans l'autre sens, il est préférable de raisonner en 64 bits prioritairement et d'accepter l'idée qu'un paquet accessible en 32 bits seulement a vécu et doit être écarté.

2.3 Déroulé de l'installation

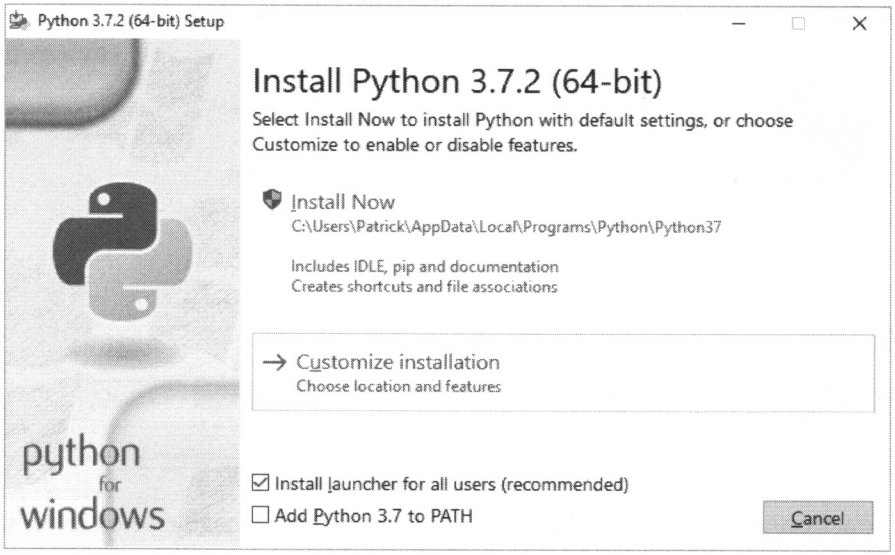
Cette section expose le déroulé d'une installation de Python. La méthode, les options, les répertoires, et de façon générale toutes les possibilités de choix parmi des variations, doivent être pris à titre d'exemple et ne sont jamais impératifs. La seule règle à se donner est d'avoir au final une installation en capacité de faire tourner Django.

© Editions ENI - All rights reserved

La machine est supposée en architecture 64 bits. Les variantes de la forme `embeddable zip file` sont plutôt dédiées à être incluses au sein de distributions de logiciels ayant une base Python, ce qui n'est pas le contexte présent. Les variantes de la forme `web-based installer` pourraient convenir, mais le but est simplement de réduire le volume de données téléchargées, en n'allant chercher que des composants voulus.

Ainsi, la variante prise est : `Windows x86-64 executable installer`.

Le déroulé ci-dessous a été réalisé avec `python-3.7.2-amd64.exe`.



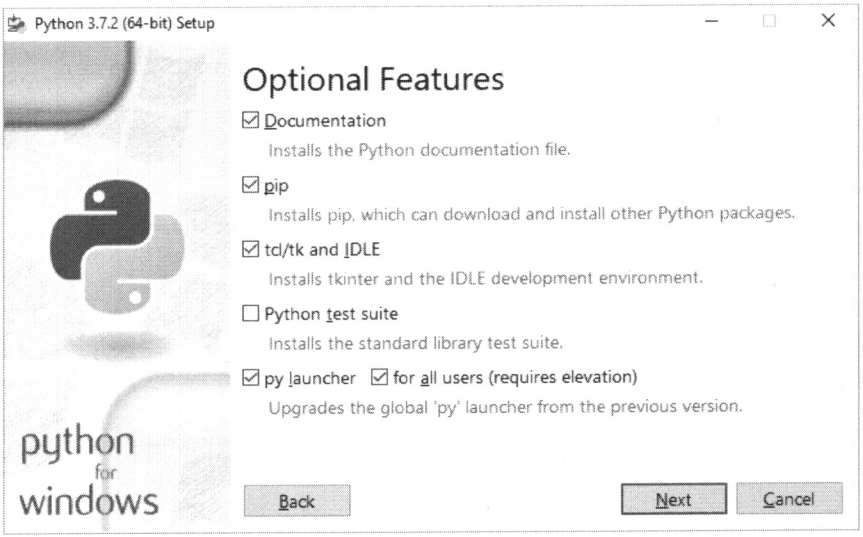
Écran d'accueil

▶ Choisissez l'installation personnalisée : **Customize installation**.

Les deux options du bas de l'écran ne sont définitives ici que pour l'installation immédiate, elles pourront être à nouveau positionnées dans les écrans suivants.

Il est préférable d'éviter l'action d'installation immédiate, car elle est source de confusion.

Si on change les paramètres donnés par défaut dans les écrans de l'installation personnalisée et qu'on revient sur cet écran par le bouton **Back**, certaines mentions reflètent effectivement nos choix, par exemple pour le répertoire d'installation, mais d'autres restent désynchronisées, par exemple avoir décoché **IDLE**, **pip** ou **Documentation** dans la fenêtre suivante, celle des fonctionnalités optionnelles.



Écran « Fonctionnalités optionnelles »

La fonctionnalité **Documentation** est bien sûr indispensable et en avoir un exemplaire sur son poste évite le besoin d'une connexion réseau pour consulter une version en ligne.

L'outil **pip** est un gestionnaire de paquets, pratiquement indispensable.

Le duo **tcl/tk** (*Tool Command Language/ToolKit*) sert de socle aux interfaces graphiques et à ce titre est une dépendance de l'outil **IDLE** (*Integrated Development and Learning Environment*). Il s'agit d'un outil à ne pas confondre avec un véritable IDE, mais il offre un moyen pratique et rapide pour, par exemple, faire tourner des bouts de code en cas de doute, vérifier une syntaxe ou un comportement, constater la levée ou non d'une exception...

© Editions ENI - All rights reserved

- Décochez l'option **Python test suite**. Il s'agit des tests pour Python lui-même et on peut s'en passer.

Les options **py launcher** et **for all users** sont utiles à conserver. On sait que sous Windows il est possible d'associer un exécutable à une extension de fichier. C'est ainsi que les fichiers d'extension `.py`, pour être lancés, ont leur chemin donné en tant que paramètre à un interpréteur, typiquement du genre `python.exe`. Tant que l'interpréteur restait dans la série Python 2.x, les montées de version (la nouvelle version prenant la place de l'ancienne) ne posaient guère de souci étant donné le soin apporté à la compatibilité. L'arrivée de Python 3, avec une refonte de la syntaxe, a marqué une rupture. Il est devenu nécessaire de faire cohabiter plus d'un interpréteur, pour une même extension de fichier. Déjà la question s'était posée pour le passage de Python 1 à Python 2, avec une discussion sur la possibilité d'introduire l'extension `.py2` (et plus généralement `.py3`, etc.), mais cela ne satisfaisait pas toutes les situations d'usage, notamment celles où la source du script n'est pas un fichier. La stratégie adoptée a été, d'une part, d'associer l'extension avec un aiguilleur et, d'autre part, de permettre au script d'inclure un indicateur optionnel sur la version qu'il requiert. Ainsi, cet aiguilleur saura solliciter la bonne version de l'interpréteur, selon un algorithme de détermination alimenté par ses paramètres de lancement et l'indicateur du script.

Ce confort de transition, initialement dédié au passage de Python 2 à Python 3, conserve une valeur même si on travaille dans un environnement purement Python 3. En effet, que ce soit pour une application complète et encore plus pour un paquet redistribuable, un développeur doit s'assurer du bon fonctionnement de son code dans le plus large spectre des versions `3.x`, même s'il est confiant qu'aucune surprise n'est à attendre. L'aiguilleur permet pour cela de préciser le numéro mineur de la version. Cette simple variation ouvre la possibilité de vérifier que ce qui fonctionnait jusqu'à présent en version `3.n` continue à fonctionner en version `3.n+1`, et au besoin alterner des essais dans l'une ou l'autre des versions pour faire de la mise au point.

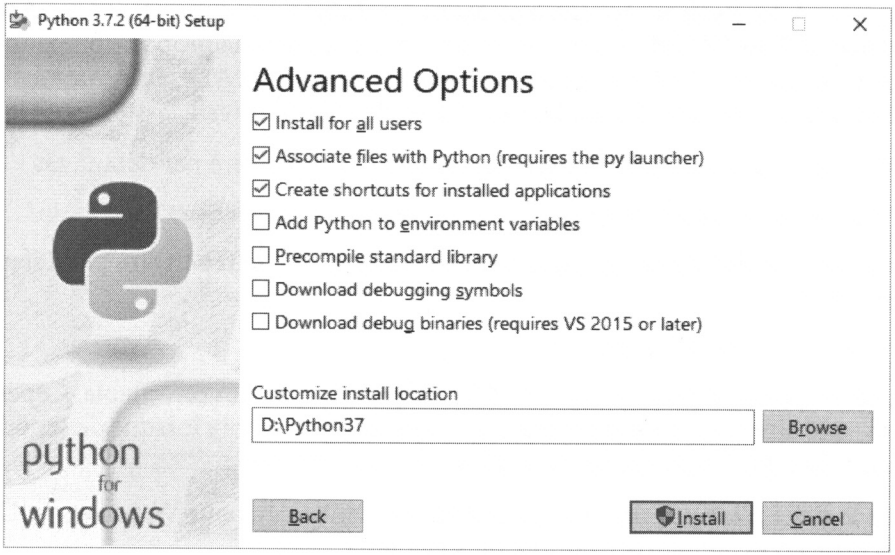
Cet aiguilleur, lorsqu'accessible à tous les utilisateurs comme il est recommandé, est matérialisé par le fichier `C:\Windows\py.exe` (ainsi que `pyw.exe`).

Pour approfondir son usage, on se reportera à l'aide, dont voici un extrait qui en résume l'essentiel :

```
C:\>py --help
Python Launcher for Windows Version 3.7.2150.1013

usage:
py [launcher-args] [python-args] script [script-args]

Launcher arguments:

-2     : Launch the latest Python 2.x version
-3     : Launch the latest Python 3.x version
-X.Y   : Launch the specified Python version
    The above all default to 64 bit if a matching 64 bit python
    is present.
-X.Y-32: Launch the specified 32bit Python version
-X-32 : Launch the latest 32bit Python X version
-X.Y-64: Launch the specified 64bit Python version
-X-64 : Launch the latest 64bit Python X version
-0  --list       : List the available pythons
-0p --list-paths : List with paths
```

- Cliquez sur **Next**. Vous arriverez à la fenêtre des options avancées.

© Editions ENI - All rights reserved



Écran « Options avancées »

▪Cochez l'option **Install for all users**.

Dans le cas le plus courant, il n'y a pas de raison de poser de restriction sur les utilisateurs, alors autant préférer être large.

Le fait de cocher cette case va automatiquement rendre aussi cochée la case **Precompile standard library**. Ceci aura pour incidence de générer, dès la phase d'installation, des exemplaires compilés en `.pyc` de tous les fichiers `.py` fournis en standard. En situation ordinaire, les exemplaires compilés ne sont produits qu'à la volée, c'est-à-dire la première fois que le fichier source est sollicité. L'objectif voulu est un gain en performances, par une simple anticipation de l'étape de compilation. Puisqu'il n'y a aucun discernement, l'opération s'applique à toute la bibliothèque et donc aussi à des pans dont on n'aura jamais l'occasion de faire usage. En d'autres termes ironiques, notamment pour une machine de développement, on peut voir cette action comme une façon efficace d'encombrer le disque.

▪Décochez l'option **Precompile standard library**.

L'option **Add Python to environment variables** est équivalente au **Add Python 3.7 to PATH** du premier écran. En accord avec la proposition par défaut (non coché), il n'est pas indispensable d'enrichir le `PATH`, car, comme il a été vu précédemment, l'aiguilleur se charge de tout le travail.

Pour une installation strictement personnelle, le répertoire par défaut est :

```
C:\Users\Patrick\AppData\Local\Programs\Python\Python37
```

Avec l'option d'installation pour tous les utilisateurs, le répertoire par défaut bascule en :

```
C:\Program Files\Python37
```

Cette proposition pour `Program Files\` est tout à fait convenable. Cependant, une alternative usuelle consiste à faire un autre choix lorsque des conditions telles que les suivantes sont plus ou moins réunies :

- Le logiciel n'est pas un produit spécifiquement écrit pour Windows (l'IDE Eclipse en est un autre exemple plus évident).
- On dispose de disques ou de partitions autres que `C:`.
- Le choix du répertoire est nativement proposé par l'installateur, c'est-à-dire que le produit est pensé dans cet esprit. Sinon, vouloir forcer des chemins dans des données de configuration devinées expose à un fort risque de dysfonctionnement.

Le présent exposé va être poursuivi avec une destination choisie à :

```
D:\Python37
```

Remarque

Il est important de ne pas faire l'économie du numéro de version, dans l'idée d'avoir en parallèle plusieurs installations de niveau différent.

De plus, pour la suite de nos explications, il sera plus aisé de mentionner une référence à un chemin ainsi raccourci.

© Editions ENI - All rights reserved

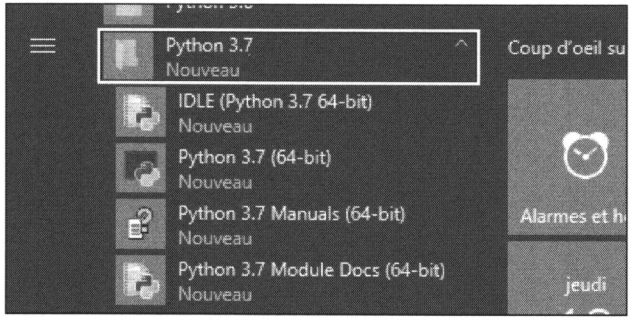
2.4 Après l'installation

Le succès de l'installation peut se confirmer par des interrogations de version.

Exemples pour une machine avec plusieurs installations

```
D:\>py --version
Python 3.7.2

D:\>py -3 --version
Python 3.7.2

D:\>py -3.6 --version
Python 3.6.3

D:\>py -3.8 --version
Python 3.8 not found!
Installed Pythons found by py Launcher for Windows
 -3.7-64 *
 -3.6-32
 -2.7-32

Requested Python version (3.8) not installed, use -0 for
available pythons

D:\>py -2 --version
Python 2.7.7
```

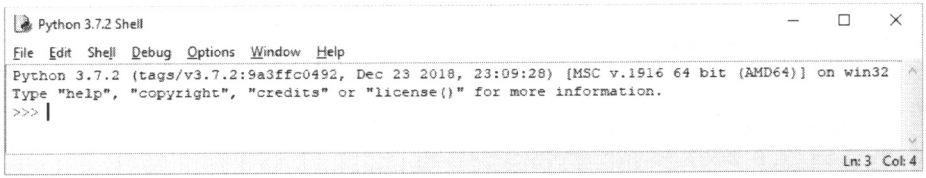
Les raccourcis de démarrage Windows se présentent ainsi :



Parmi les entrées, deux méritent une attention particulière :

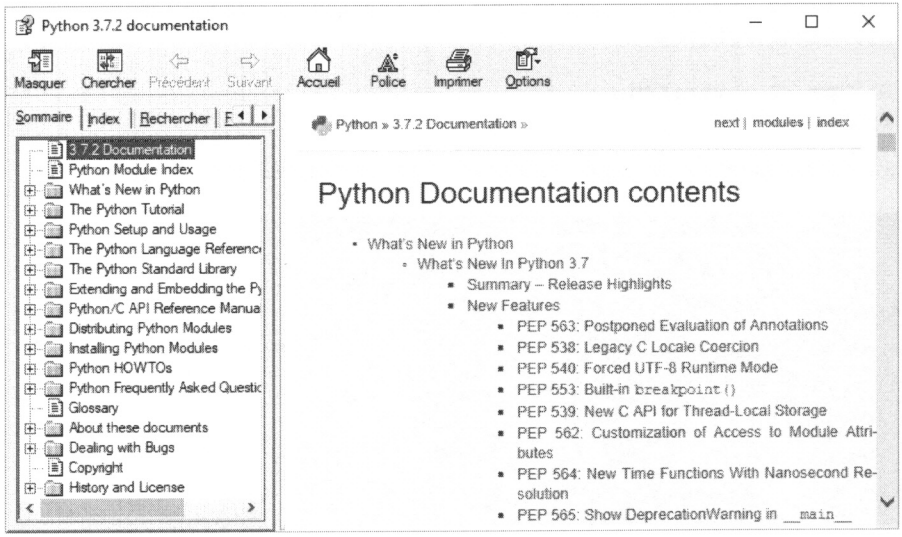
IDLE

IDLE (*Integrated Development and Learning Environment*) est un environnement de travail qui présente un enrobage graphique autour de l'interpréteur en ligne de commande.



On pourra se contenter de cet environnement pour faire des essais de langage purement Python, mais lorsqu'il s'agira de faire des essais dépendants de l'environnement Django, il sera préférable d'employer une console (*shell*), tel qu'il est décrit à la fin du chapitre Création de site.

Python 3.7 Manuals



© Editions ENI - All rights reserved

L'usage de la documentation est tellement indispensable et les raccourcis de démarrage sont tellement pénibles que la manipulation suivante est vivement conseillée :

▶Faites un raccourci sur votre bureau de la cible :

```
D:\Python37\Doc\python372.chm
```

3. Installation d'un moteur de base de données

Django supporte officiellement les systèmes suivants : PostgreSQL, MySQL, Oracle et SQLite. Il est possible de trouver des paquets implémentant l'accès à d'autres moteurs, mais ce sont des initiatives d'individus ou d'organisations à part, qui ont donc chacune leur pérennité, leur couverture de fonctionnalités et leur support des versions de Django. Bien sûr, toute question relative à leur support doit être adressée à ces intervenants externes.

Comme annoncé dans l'introduction, il va dès à présent être mis en place un gestionnaire de base de données, plus étoffé que le SQLite déjà présent dans la distribution Python.

■Remarque

Quel que soit le choix, il est plus sûr d'employer en environnement de développement le moteur qui sera utilisé en environnement de production, peu importe les considérations de confort ou d'outillage.

Pour le projet, le choix arbitraire de PostgreSQL est fait, sans pour autant avoir d'arguments forts de préférence par rapport aux autres possibilités.

3.1 Installation de PostgreSQL

◘Rendez-vous sur le site : https://www.postgresql.org/download/

En suivant le lien dédié à la distribution binaire pour Windows, il est proposé une liste d'installateurs. Faisons ici le choix de **Interactive installer by EnterpriseDB**, qui inclut l'outil graphique d'administration **pgAdmin**.

Le déroulé ci-dessous a été réalisé avec :

```
postgresql-11.1-1-windows-x64.exe
```

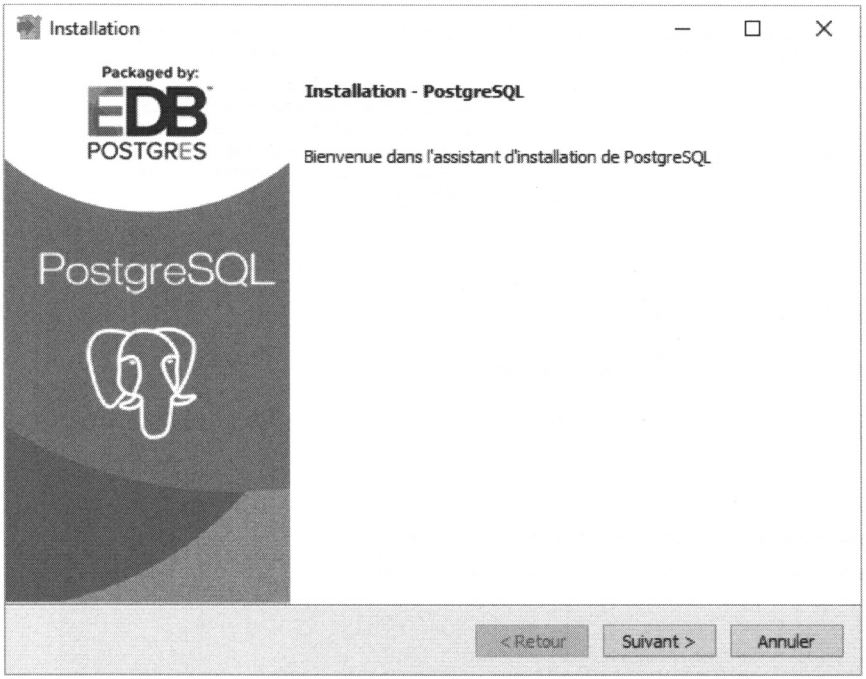
◘Si vous avez déjà installé une instance de serveur, elle peut se trouver en cours de fonctionnement. Dans ce cas, veillez à la stopper.

Avant l'apparition du premier écran, l'installateur va faire de lui-même des installations ou des mises à jour sur la machine, sans même demander confirmation au propriétaire. Il s'agit de composants nécessaires à l'exécution de programmes développés avec l'outil Microsoft Visual C++.

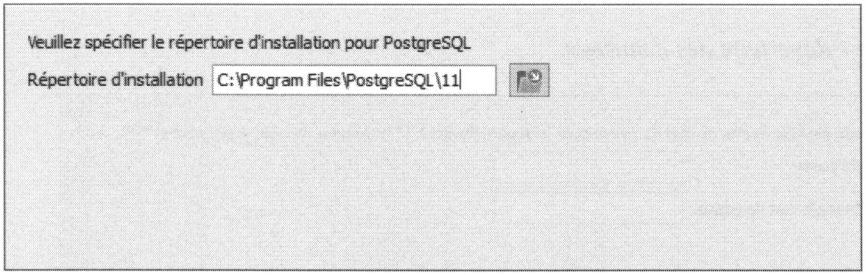
Les actions sont tellement rapides qu'on n'a pas le temps de voir les quelques informations qui défilent à l'écran, mais voici la liste de ces composants :

- Microsoft Visual C++ 2013 x86 Minimum Runtime - 12.0.40660
- Microsoft Visual C++ 2013 x86 Additional Runtime - 12.0.40660
- Microsoft Visual C++ 2017 x86 Minimum Runtime - 14.15.26706
- Microsoft Visual C++ 2017 x86 Additional Runtime - 14.15.26706
- Microsoft Visual C++ 2017 x64 Minimum Runtime - 14.15.26706
- Microsoft Visual C++ 2017 x64 Additional Runtime - 14.15.26706

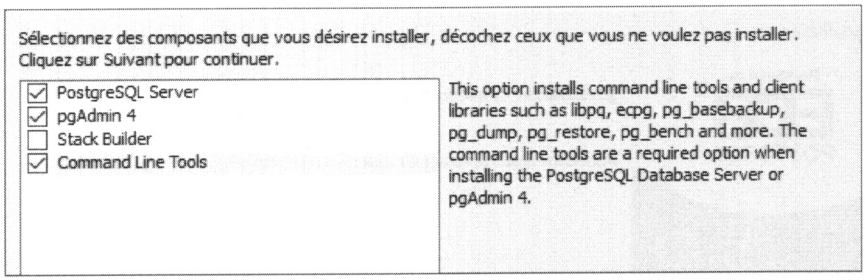
© Editions ENI - All rights reserved



Écran d'accueil



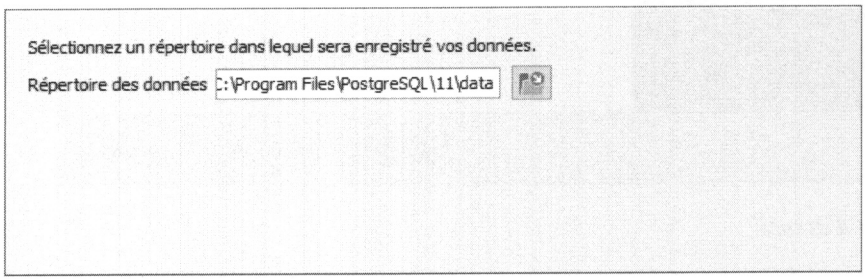
Écran « Répertoire d'installation »



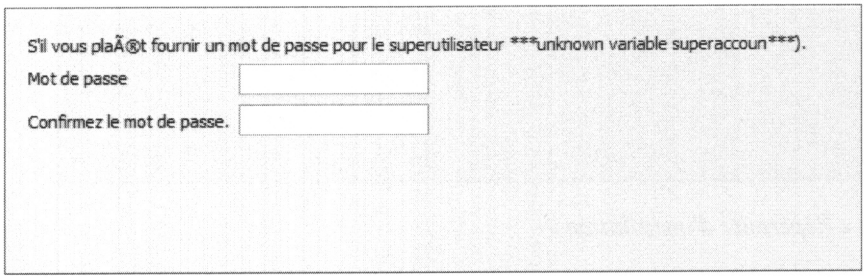
Écran « Sélection des composants »

▶Décochez l'option **Stack Builder**.

Il s'agit d'un outil optionnel pour télécharger et installer par la suite des outils et pilotes supplémentaires.

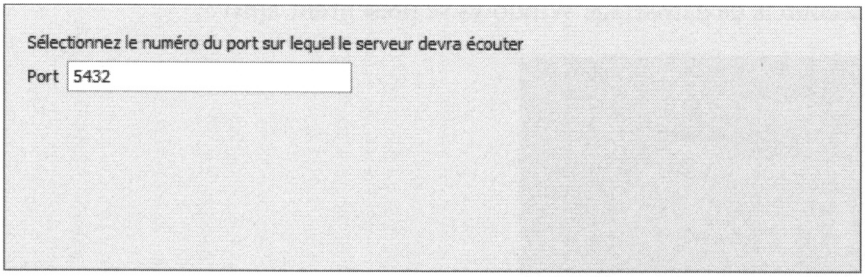


Écran « Répertoire des données »

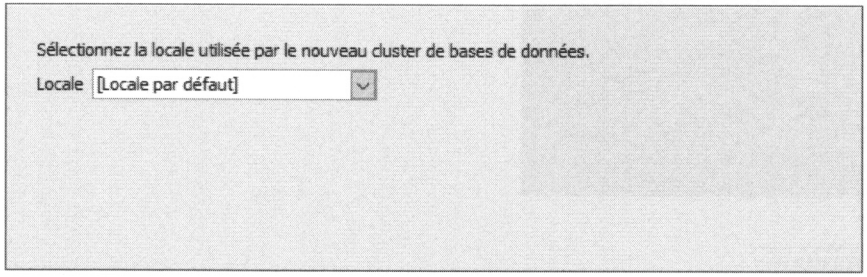


Écran « Mot de passe »

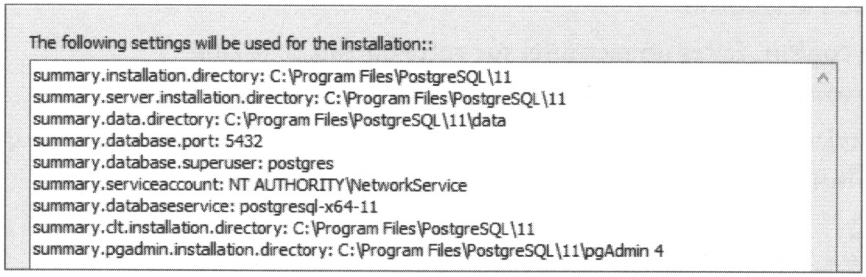
© Editions ENI - All rights reserved



Écran « Port »



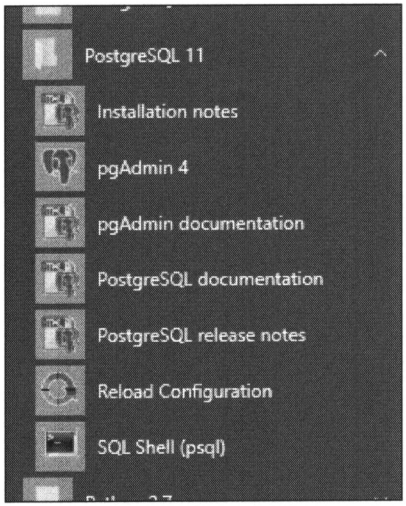
Écran « Options avancées »



Écran « Synthèse »

Après un dernier écran de confirmation, l'installation se réalise.

Les raccourcis de démarrage Windows se présentent ainsi :



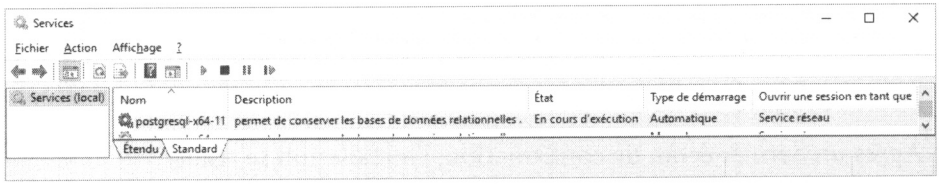
Remarque

Au cas où ce menu ne serait pas apparent à l'issue immédiate de la phase d'installation, le simple fait de clore puis ouvrir sa session Windows permet de rétablir la situation.

Par confort, faites un raccourci sur votre bureau de la cible :

```
C:\Program Files\PostgreSQL\11\pgAdmin 4\bin\pgAdmin4.exe
```

Le service est installé avec un type de démarrage à « Automatique », ce qui signifie qu'il sera démarré en même temps que le système.



© Editions ENI - All rights reserved

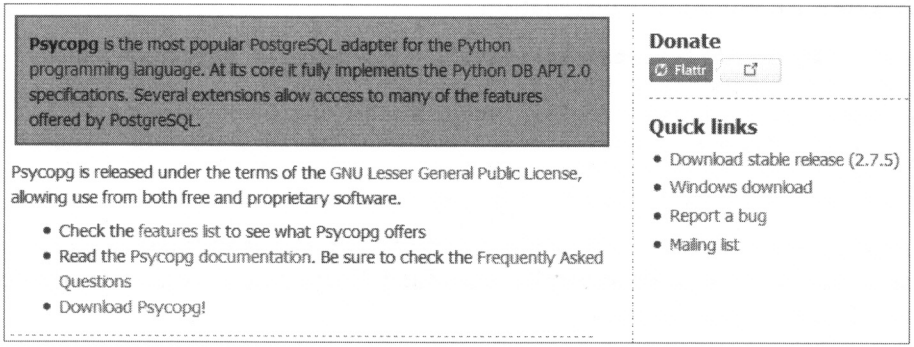
La présence permanente du service n'est pas une nécessité sur un poste de travail. Pour éviter d'occuper de la mémoire lorsque ce n'est pas utile, il est avantageux de passer en type « Manuel » et de ne démarrer le serveur de bases de données que durant les périodes où on travaille sur un projet qui le nécessite.

3.2 Installation du pilote

On comprend facilement que ce n'est pas dans le rôle de la distribution Python d'inclure les paquets pour s'interfacer avec des systèmes périphériques, trop nombreux.

Dans le cas de PostgreSQL, le pilote d'interconnexion additionnel nécessaire se nomme `psycopg2`.

▶Rendez-vous sur le site : http://initd.org/psycopg/



Écran de la page d'accueil

Surtout ne pas se précipiter à vouloir télécharger un fichier *tarball*, dont on ne sait pas quoi faire, ou un installateur binaire en `.exe` assez souvent pas à jour de version.

La section de téléchargement donne une procédure d'installation simple puisque réduite à utiliser l'outil `pip`. Mais les pages du site web ne sont pas détaillées et ne sont pas toujours à jour. Une lecture attentive de la documentation permet d'en apprendre plus sur les instructions et conditions d'installation.

Pour éviter d'avoir à satisfaire les conditions requises pour une installation à partir du code source, il est plus simple de s'orienter vers une distribution binaire toute prête, sous forme d'un fichier *wheel* de type `.whl`.

Dans sa version 2.7, le paquet du nom historique `psycopg2` peut encore cibler la distribution binaire, mais ce ne sera plus le cas dans la version 2.8 qui n'offrira que la distribution source. Pour continuer à obtenir une distribution binaire, il faut pointer le paquet nouvellement nommé `psycopg2-binary`, et c'est ce qui va être fait en anticipation.

Il est inutile de préciser un numéro de version, la plus récente sera prise et cela convient. L'outil saura prendre la bonne distribution en accord avec la combinaison OS-Python-Architecture de la machine hôte.

Le chemin vers l'outil n'est pas dans le `PATH` et n'a pas nécessité de l'être. Il suffit de le préciser au lancement.

▶Passez la commande d'installation :

```
D:\>\Python37\Scripts\pip install --no-compile psycopg2-binary
Collecting psycopg2-binary
  Downloading https://files.pythonhosted.org/packages/5f/75/[...]
  [...]/psycopg2_binary-2.7.6.1-cp37-cp37m-win_amd64.whl (996kB)
Installing collected packages: psycopg2-binary
Successfully installed psycopg2-binary -2.7.6.1
```

L'option `--no-compile` épargne la compilation par avance des fichiers `.py` (se référer à la coche **Precompile standard library** de l'installation Python pour la même justification).

La liste des apports est :

```
D:\>dir \Python37\Lib\site-packages\ps*
<DIR>          psycopg2
<DIR>          psycopg2_binary-2.7.6.1.dist-info
```

© Editions ENI - All rights reserved

4. Installation d'un outillage pour traduction

Au stade consacré à l'internationalisation, il sera nécessaire de mettre en œuvre des outils dédiés à la gestion de la déclinaison des chaînes de caractères en plusieurs langues. Django dispose d'une commande pour collecter ces chaînes et les rassembler dans un fichier de messages. Le fichier répond à un format reconnu, établi par des outils à code source ouvert qui prennent aussi en charge d'autres étapes du processus.

La traduction du contenu du fichier se réalise à part, indépendamment de l'infrastructure logicielle, soit de façon manuelle (simple éditeur de texte), soit avec des outils spécialisés ou des plateformes de services.

La dernière étape consiste à compiler chacun des fichiers de messages, pour que le serveur utilise un format optimisé, meilleur pour ses performances.

Les opérations de collecte et de compilation sont orchestrées par l'infrastructure logicielle. Celle-ci s'appuie pour cela sur l'outillage `GNU gettext`. Contrairement à d'autres OS, cette suite d'outils n'est pas disponible dans Windows, il est nécessaire de procéder à son installation.

La documentation Django ne donne pas de directives quant au choix du dépôt où trouver l'outillage. Pour faciliter le travail, elle cite un lien (non reproduit ici, car il a varié avec la version de la documentation et donc sa stabilité est incertaine) vers un site externe qui propose une distribution.

Un installateur peut éventuellement donner le choix entre deux variantes :

- *shared* : étant donné qu'ils appartiennent à une même famille d'outillage, les fichiers exécutables se basent sur une large base de code commun. Ce code est factorisé et placé en un seul exemplaire dans quelques fichiers `.dll`. On gagne en taille, en contrepartie de devoir conserver groupés tous les fichiers dans un même lieu.
- *static* : chaque fichier exécutable porte tout le code dont il a besoin, sans avoir besoin d'auxiliaire. Il peut être déplacé indépendamment des autres, mais au prix d'un surpoids.

Remarque

Pour l'usage prévu, la portabilité n'est pas utile et il vaut mieux donner la préférence à la déclinaison avec les DLL.

Le distributeur peut offrir le choix entre un programme exécutable d'installation et un fichier dans un format archive. L'archive a l'avantage de laisser l'utilisateur entièrement maître de la façon de procéder.

À titre d'exemple, l'installation est poursuivie avec une distribution que le site mentionné en lien dans la documentation Django a permis de télécharger :

```
gettext0.19.8.1-iconv1.15-shared-64.zip
```

De toute l'archive, on peut se contenter du seul contenu sous le répertoire `bin`.

- Versez le contenu du répertoire `bin/` vers un répertoire de votre choix.

Le nom du répertoire n'a pas d'importance. Dans la continuité d'une époque où il fallait réunir plusieurs morceaux de distribution, typiquement `gettext-runtime` et `gettext-tools`, le nom `gettext-utils` est un choix usuel et expressif.

Exemples

- `C:\Program Files\gettext-utils\`
- `D:\bin\gettext-utils\`

- Ajoutez le chemin vers ce répertoire dans la variable d'environnement système `PATH`.

Cet ajout est nécessaire puisque naturellement les commandes de Django emploient ces outils sans la connaissance de leur emplacement effectif.

Le succès de l'installation se vérifie par une interrogation de la version du principal exécutable.

Exemple

```
D:\>xgettext --version
xgettext (GNU gettext-tools) 0.19.8.1
```

© Editions ENI - All rights reserved

5. Installation d'un gestionnaire de fuseaux horaires

Le paquet `pytz` (comprendre *PYthon TimeZone*) est une dépendance requise par Django. Son installation sera initiée par celle de Django si le paquet n'est pas présent. On pourrait donc se passer de cette étape manuelle. Elle va toutefois être réalisée volontairement, pour son effet didactique.

Ce paquet apporte le support des fuseaux horaires pour les manipulations de dates et heures locales. Il s'appuie sur la base de données Olson des fuseaux horaires.

Remarque

Django a fait ce choix, mais il existe d'autres gestionnaires. Par exemple, la documentation Python recommande le paquet `dateutil.tz`.

Passez la commande d'installation :

```
D:\>\Python37\Scripts\pip install --no-compile pytz
Collecting pytz
 Downloading https://files.pythonhosted.org/packages/61/28/[...]
 [...]/pytz-2018.9-py2.py3-none-any.whl (510kB)
Installing collected packages: pytz
Successfully installed pytz-2018.9
```

La liste des apports est :

```
D:\>dir \Python37\Lib\site-packages\py*
<DIR>          pytz
<DIR>          pytz-2018.9.dist-info
```

6. Installation de Django

Des versions anciennes de la documentation officielle contenaient, sur le sujet de l'installation, une section traitant du retrait préalable d'une éventuelle installation précédente. Elle était justifiée par le besoin d'une intervention manuelle d'effacement dans le cas précis d'une installation elle aussi manuelle, c'est-à-dire sans passer par un outil spécifique. Cette section n'apparaît désormais plus, puisque seul l'usage de l'outil `pip` est recommandé et celui-ci sait gérer proprement une installation par-dessus une version déjà présente. Dans tous les cas, au final, une action d'installation est censée supplanter un exemplaire déjà présent.

Dans ce chapitre dédié aux installations, il est supposé que les produits ne sont pas déjà présents sur le poste de travail ou qu'il n'y a aucune conséquence à remplacer une installation préexistante. S'agissant de l'infrastructure logicielle, il est pourtant assez courant de vouloir confronter son travail et ses dépendances à plus qu'une version, au moins durant une période de transition, pour vérifier la continuité de fonctionnement. Ce cas spécifique est abordé au chapitre Alternatives.

Deux méthodes sont à écarter, car elles ne se prêtent pas au contexte :

- L'installation par un gestionnaire de paquets, soit de sa distribution OS, soit d'un outil auxiliaire.

  C'est plutôt un usage du monde Unix. De plus, on n'est pas certain de disposer de la dernière version ni de rester maître de la manière dont le produit est installé (chemins, dépendances, compléments non désirés, etc.).

- L'installation de la version en cours de développement.

  Elle est destinée à ceux qui veulent suivre au plus près les évolutions correctives et évolutives, grâce à un gestionnaire de sources tel que `git`. En pratique, cet usage est destiné à la participation au développement du produit et à la proposition de correctifs (*pull request* et *patch*).

Il va donc être procédé à l'installation d'une version officielle stable, selon la méthode recommandée.

© Editions ENI - All rights reserved

Passez la commande d'installation :

```
D:\>\Python37\Scripts\pip install --no-compile django
Collecting django
 Downloading https://files.pythonhosted.org/packages/36/50/[...]
 [...]/Django-2.1.5-py3-none-any.whl (7.3MB)
Requirement already satisfied: pytz in
d:\python37\lib\site-packages (from django) (2018.9)
Installing collected packages: django
Successfully installed django-2.1.5
```

Le message d'avertissement suivant est sans gravité et peut être ignoré :

```
The script django-admin.exe is installed in
'd:\python37\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to
suppress this warning, use --no-warn-script-location.
```

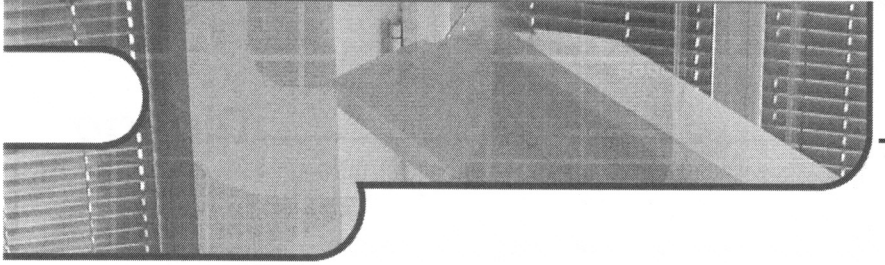
Le succès de l'installation peut se confirmer par des interrogations de version.

Exemples

```
D:\>py -c "import django; print(django.get_version())"
2.1.5

D:\>\Python37\Scripts\django-admin --version
2.1.5
```

La liste des apports est :

```
D:\>dir \Python37\Lib\site-packages\d*
<DIR>          django
<DIR>          Django-2.1.5.dist-info

D:\>dir \Python37\Scripts\d*
102 794 django-admin.exe
    132 django-admin.py
```



Chapitre 2
Création de site

1. Objectifs

Dans le chapitre précédent, les outils et l'infrastructure logicielle nécessaires à la création de site ont été mis en place. Ceci va permettre de poser les bases du projet.

Le choix du répertoire racine d'un site est libre, il n'y a aucune obligation en rapport avec Python ou Django. De même il n'y a pas à se placer sous la racine d'un serveur frontal (Apache, Nginx, etc.) et il vaut d'ailleurs mieux l'éviter pour ne pas s'exposer à un éventuel risque de visibilité du code à cause d'une faille de sécurité. Le présent exposé va être poursuivi avec une destination courte et expressive, choisie arbitrairement à `D:\dj`.

2. Création d'un projet

La mise en œuvre d'une infrastructure logicielle suppose le respect d'un cadre conventionnel de disposition des éléments, ou, dit en d'autres termes, mettre les choses là où l'infrastructure a prévu de les trouver. Plutôt que de tout devoir écrire soi-même, des assistants sont mis à disposition pour poser ce cadre initial, sur la base d'un gabarit, avec des valeurs par défaut supposées convenir dans la majorité des cas.

Créez un nouveau répertoire pour héberger le projet :

```
D:\>md dj
```

Créez un nouveau projet, selon l'une ou l'autre façon :

```
D:\>\Python37\Scripts\django-admin startproject mysite dj
```

Ou :

```
D:\>cd dj
D:\dj>\Python37\Scripts\django-admin startproject mysite .
```

Dans ce cas, notez bien la présence d'un caractère point isolé final.

Cette structure va être créée :

```
dj\
├─ manage.py
└─ mysite\
   ├─ __init__.py
   ├─ settings.py
   ├─ urls.py
   └─ wsgi.py
```

La commande `startproject` a, d'une part, un paramètre obligatoire pour nommer le projet et, d'autre part, un paramètre optionnel pour citer un chemin de répertoire, devant exister préalablement, pour accueillir le projet.

Le nom du projet est un choix libre, employer le mot `mysite` est juste une convention usuelle. En effet, il n'est pas nécessaire d'utiliser le nom du projet puisque le répertoire racine donne déjà son nom (`dj` dans le cas présent). De plus, lorsqu'on passe d'un site à un autre, il est plus facile de comprendre que les paquets désignés sous le préfixe `mysite` concernent spécifiquement le site courant, en contraste évident avec des paquets de provenance externe. L'éventuelle recopie de paquets d'un projet vers un autre est aussi facilitée puisqu'on évite ainsi l'effort fastidieux des mises en accord du nom.

Le choix est fait de citer explicitement le paramètre optionnel, car en son absence un premier répertoire sera créé en reprenant le nom du projet (s'il existe déjà, la commande échouera).

© Editions ENI - All rights reserved

La commande minimale, que l'on voit habituellement dans les documentations et tutoriels, de la forme :

```
django-admin startproject mysite
```

aurait produit une structure telle que :

```
dj\
└ mysite\
  – manage.py
  └ mysite\
    └ ...
```

Comme on a naturellement tendance à s'être déjà placé dans un répertoire vierge, nouvellement créé pour être dédié au projet, la présence d'un premier répertoire `mysite` paraît superflue et on se demande quel peut être son rôle et si son nom doit être préservé. En réalité il ne s'agit que d'un conteneur, dont le nom et la présence n'ont pas d'importance pour Django. Il est donc permis de supprimer cet étage ou de le renommer.

Les fichiers arrivés sont :

- `manage.py` : le point d'entrée pour les commandes en mode ligne destinées à ce projet. Les commandes documentées pour `django-admin` s'appliquent à cet utilitaire puisqu'il joue le même rôle, mais du fait qu'il aménage son environnement, il a l'avantage de cibler ce projet implicitement, sans qu'il soit nécessaire de le préciser.
- `__init__.py` : le traditionnel fichier pour qualifier ce répertoire au rang d'un paquet Python. Ce fichier est ici vide.
- `settings.py` : les paramètres de configuration du projet pour diriger l'infrastructure logicielle.
- `urls.py` : les déclarations de routes du projet.
- `wsgi.py` : le point d'entrée à l'usage des serveurs web frontaux qui implémentent l'interface WSGI, permettant ainsi un déploiement aisé du projet.

3. Premier lancement du site

Afin de se rassurer sur la bonne situation des actions menées jusqu'à présent, il est permis à ce stade de tenter un lancement du serveur de développement de Django.

▶ Passez la commande de lancement du serveur intégré :

```
D:\dj>py manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).

You have 15 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): admin, auth,
contenttypes, sessions.
Run 'python manage.py migrate' to apply them.
<Mois JJ, AAAA - hh:mm:ss>
Django version 2.1.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

On constate qu'à l'occasion de ce lancement, il est préalablement procédé à un balayage du projet, à la recherche d'erreurs usuelles. Il s'agit des mêmes contrôles que ceux de la commande `check`.

Le message d'avertissement à propos de migrations en attente d'application est attendu dans le contexte actuel. Il suffit de l'ignorer et il n'est pas nécessaire de passer la commande mentionnée.

Comme il est dit dans la sortie de la console et dans la documentation de la commande `runserver`, il ne s'agit aucunement d'un serveur utilisable pour un environnement de production.

© Editions ENI - All rights reserved

C'est un outil dédié aux développeurs pour faciliter leur travail de conception, car il présente les avantages suivants :

- Il est disponible sur la machine locale, immédiatement et sans effort. L'interface réseau et le port sont par défaut à 127.0.0.1 et 8000, mais il est possible de spécifier d'autres valeurs en arguments. Par exemple : `manage.py runserver 192.168.0.10:8899` permet de servir aussi des requêtes provenant d'autres postes de travail que sa propre machine (192.168.0.10 étant dans cet exemple une adresse IP privée sur un segment de réseau local). On peut aussi employer l'adresse `0.0.0.0` pour écouter toutes les interfaces réseau de la machine. Son raccourci `0` mentionné par la documentation Django ne semble pas être supporté sur la plateforme Windows et le lancement échoue avec l'erreur « `Error: [Errno 11001] getaddrinfo failed` ». Une autre variante est d'employer le nom de la machine, par exemple `PC-Patrick:8001`, mais cela suppose un complément de configuration concernant `ALLOWED_HOSTS`, en rapport avec des protections de sécurité.
- Un changement dans les fichiers de code amorce automatiquement un rechargement du serveur, sans qu'il soit nécessaire de penser à l'arrêter et le relancer. Cette manipulation d'arrêt-relance reste quand même un passage obligé dans les cas d'ajouts de fichier. Il est aussi à noter que la surveillance a un coût en ressources système, car elle se réalise par la lecture toutes les secondes de la date de modification des fichiers. On peut ainsi constater à l'état de repos du serveur une consommation continuelle de CPU (processeur) de l'ordre de 4 à 7 %. L'argument optionnel `--noreload` permet de désactiver ce comportement et ainsi laisse l'occupation CPU à 0 % au repos, ce qui peut avoir un intérêt, par exemple pour un simple usage de démonstration ou pour épargner la consommation énergétique sur un portable.

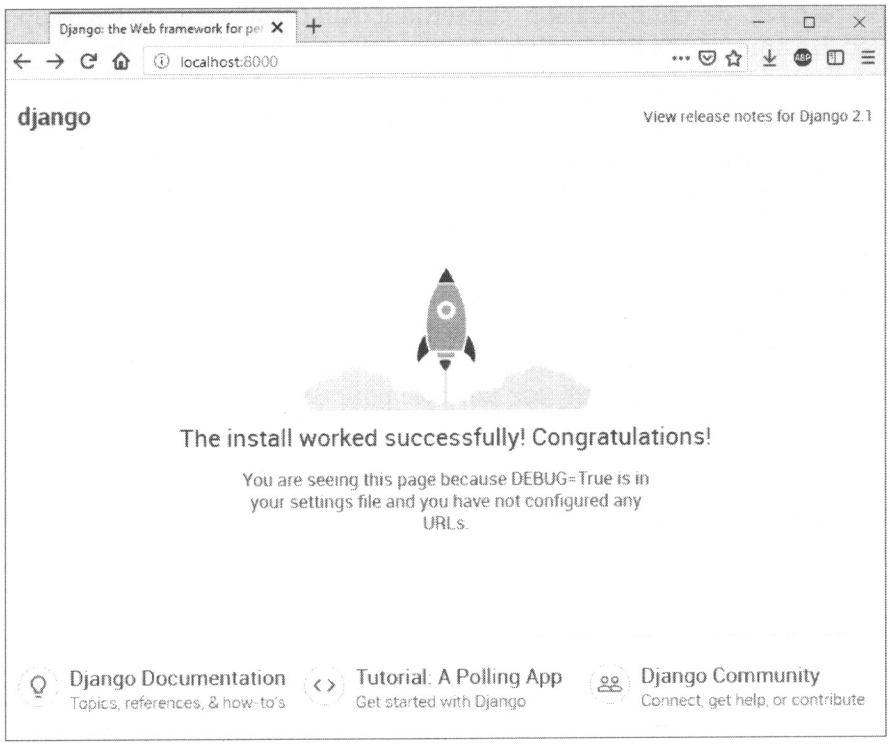
Si nécessaire, il est possible d'avoir plusieurs serveurs actifs en même temps, a priori sur des projets séparés, à la seule condition de leur affecter des numéros de port différents.

Si on observe les répertoires du projet, on constate la présence d'un fichier supplémentaire à la suite du lancement :

```
dj\
├─ db.sqlite3
└─ ...
```

Rien n'a encore été établi relativement à la base de données, mais puisque l'infrastructure logicielle a la nécessité d'avoir une liaison avec une base, la configuration fournie par le gabarit prévoit le plus simple, c'est-à-dire une base `sqlite3` puisque sa forme se réduit à un simple fichier. La configuration a établi que le fichier porte ce nom et soit localisé à cet emplacement. Il est normal que le fichier soit encore totalement vide, aucune action de création de tables n'a eu lieu jusqu'à présent.

▶Allez sur le site http://localhost:8000 pour y constater la page servie :



© Editions ENI - All rights reserved

Sur la console, on constate que le serveur donne les traces des requêtes servies :

```
[...] "GET / HTTP/1.1" 200 16348
[...] "GET /static/admin/css/fonts.css HTTP/1.1" 200 423
Not Found: /favicon.ico
[...] "GET /favicon.ico HTTP/1.1" 404 1972
[...] "GET /static/admin/fonts/Roboto-Regular-webfont.woff
   HTTP/1.1" 200 80304
[...] "GET /static/admin/fonts/Roboto-Bold-webfont.woff
   HTTP/1.1" 200 82564
[...] "GET /static/admin/fonts/Roboto-Light-webfont.woff
   HTTP/1.1" 200 81348
```

Ceci permet par exemple de repérer des ressources manquantes, qui se révèlent par un code de retour HTTP à 404 plutôt que 200. C'est ici le cas avec la ressource graphique `favicon.ico` dont la demande est tentée automatiquement par le navigateur. Il est normal qu'elle ne soit pas trouvée puisque rien n'a été fait pour cela.

Page d'accueil

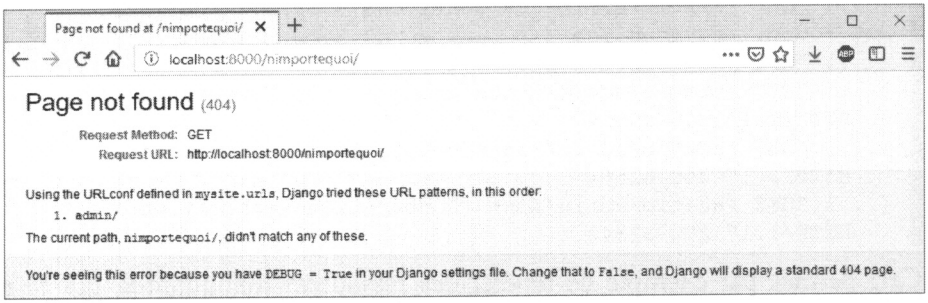
Il nous est indiqué que si nous voyons cette page d'accueil par défaut, c'est parce que nous tournons en mode DEBUG et que nous n'avons pas configuré de routes URL. Mais pourtant nous avons bien un fichier de routes `urls.py`, ce qui semble en contradiction avec l'annonce.

En réalité, nous sommes dans le cas ordinaire d'une page non trouvée qui devrait provoquer une réponse avec le code 404. Mais dans un contexte de fonctionnement en mode DEBUG, il est fait des cas particuliers de ces deux situations :

- 1. Soit véritablement aucune route n'est définie. Dans ce cas, n'importe quelle page demandée se voit servir ce contenu.
- 2. Soit la page demandée est la racine du site et il n'existe qu'une route et celle-ci mène au module d'administration intégré.

Nous sommes actuellement dans la situation n°2, car nous pouvons constater dans le fichier `urls.py`, donné par l'assistant de création de projet, la présence d'une définition d'une route, unique, vers l'administration.

On peut vérifier que toute autre requête invalide, par exemple http://localhost:8000/nimportequoi/, rend vraiment une page 404 :



Ressources statiques

Les traces de la console montrent qu'il n'est pas seulement servi la page HTML, mais aussi des ressources statiques, telles que des fichiers de style CSS et des polices de caractères WOFF. Ce serait également le cas pour des images et des scripts. Or, la commande `runserver` basique du serveur intégré n'est pas capable de délivrer cette nature de données. Le fonctionnement est possible parce que la configuration fournie par le gabarit a opté pour la déclaration de présence de l'application additionnelle `django.contrib.staticfiles` et cette application enrichit la commande de base avec une capacité à servir des fichiers statiques.

Cette capacité peut au besoin être désactivée avec l'option `--nostatic`, comme le montre cette trace :

```
D:\dj>py manage.py runserver --nostatic
[...]
Quit the server with CTRL-BREAK.
[...] "GET / HTTP/1.1" 200 16348
Not Found: /static/admin/css/fonts.css
[...] "GET /static/admin/css/fonts.css HTTP/1.1" 404 2017
Not Found: /favicon.ico
[...] "GET /favicon.ico HTTP/1.1" 404 1972
```

Que le serveur intégré soit à la fois capable de servir des pages dynamiques et des fichiers statiques est bien pratique pour le développeur, dont l'attention doit se concentrer sur l'écriture du code plutôt que sur la mise en fonctionnement d'une panoplie d'outils pour voir ses pages complètes.

© Editions ENI - All rights reserved

En situation de production, l'architecture doit bien sûr être différente, au moins pour des raisons de performances en rapport avec le volume de requêtes.

En pratique, les ressources statiques doivent être délivrées par un serveur frontal, sans que le serveur Django ne soit sollicité, et heureusement, car il n'est pas destiné à ce rôle, ni en matière de volume ni de sécurité. Cependant, il peut arriver de vouloir utiliser le serveur intégré sans être en mode DEBUG et vouloir profiter de sa capacité de serveur statique, par exemple pour une preuve rapide de fonctionnement ou pour une démonstration. Dans ce cas, il faut exprimer clairement qu'on est conscient d'agir en dehors du cadre ordinaire, en employant l'option `--insecure`.

4. Création d'une première application

Maintenant que le cadre du projet est en place, il est possible d'y adjoindre une ou des applications. Le minimum est d'avoir une application. Pour un site très basique, on peut s'en contenter et y loger tout le code. Mais le plus souvent, il est préférable de concevoir une architecture modulaire, en cloisonnant les fonctionnalités par grandes familles, chacune donnant lieu à une application. Certaines applications vont émerger naturellement, par exemple parce qu'elles sont suffisamment génériques pour être employées dans d'autres projets, ou parce qu'elles fournissent une extension optionnelle de fonctionnalités.

Même si tout est estimé spécifique au projet et forme un ensemble technique apparemment compact et indissociable, une découpe présente néanmoins des avantages. Par exemple, pour des raisons de répartition de l'effort de développement puis de maintenance au sein d'une équipe, ou pour se ménager la possible substitution d'une technologie par une autre, éventuellement encore inconnue, quoique pressentie à ce stade.

A contrario, il faut se garder de l'excès inverse et tronçonner en de trop nombreuses applications. Certaines peuvent se révéler ridiculement petites ou la quantité peut devenir ingérable et donner une impression d'absence de maîtrise du sujet. On risque aussi de finir par ne plus savoir qui fait quoi et parfois on se retrouve à écrire ce qui l'a déjà été.

Une application offre des fonctionnalités et éventuellement fait appel aux fonctionnalités d'autres applications, d'où l'existence d'un réseau de dépendances. Dans ce réseau, la situation à éviter est d'aboutir à des boucles de dépendance, c'est-à-dire, par exemple, que l'application A a besoin de l'application B qui elle-même a besoin de l'application A.

Le raisonnement est le même si le chemin comporte plus de maillons. Ce schéma augmente le risque d'exposition au problème Python bien connu d'importation circulaire : pour importer un module, il faut d'abord importer un autre module, qui doit aussi d'abord importer encore un autre module, et ainsi de suite, jusqu'à retomber sur un module déjà rencontré dans le parcours. La boucle est bouclée et il n'y a pas d'autre solution que de casser cette chaîne de dépendances en restructurant l'architecture.

Même si le fonctionnement n'est pas empêché, il y a lieu de s'interroger sur la qualité de la conception lorsque des applications sont tellement imbriquées les unes aux autres qu'au final leurs frontières finissent par ne plus se distinguer. C'est peut-être alors l'occasion de remettre en débat l'architecture, en regroupant des composants interdépendants et en introduisant des éléments pour un meilleur cloisonnement.

Un objectif à viser est d'avoir un schéma de dépendances orienté linéairement, comme une pyramide (du sommet vers la base) ou une sphère (du cœur vers l'extérieur). Dans tous les cas, dessiner un graphe de dépendances entre applications est une aide utile à de multiples égards : la détection ou la prévention d'introduction de boucles, la compréhension pour un nouvel arrivant sur le projet, la maintenance, etc.

4.1 Emplacement d'une application

Une application peut se situer n'importe où sur un système de fichiers, à condition d'être visible par un des chemins de recherche Python. Le choix de l'emplacement est donc assez subjectif.

© Editions ENI - All rights reserved

Le schéma suivant expose quatre principaux emplacements potentiels :

```
Python37\Lib\
└ site-packages\
  └ <appA>\
dj\
├ manage.py
├ <appB>\
├ mysite\
│  └ <appC>\
└ <unRépertoire>\
  └ <appD>\
```

Cas appA

Cet emplacement est avant tout destiné à recevoir des paquets additionnels, de source externe, pour compléter l'installation Python de base. Ces paquets sont alors à la disposition de tous les projets. C'est bien l'intention recherchée lorsque certains paquets ont été apportés dans le chapitre d'installation. Ce n'est de toute évidence pas un lieu privilégié pour l'application d'un projet.

Cas appB et appC

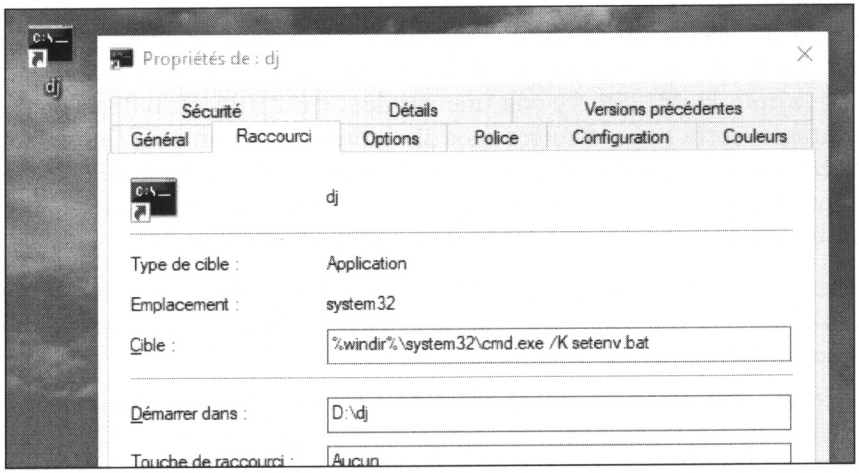
Ces emplacements peuvent être considérés de façon équivalente, pour accueillir les applications du projet. Dans le cas `appC`, il faudra le désigner comme un module de deuxième niveau, en écrivant ses références sous la forme : `mysite.appC`. Plutôt que d'y voir une lourdeur d'écriture, essayons d'en tirer un avantage. Après tout, d'autres modules constitués d'un seul fichier sont déjà référencés ainsi puisqu'on trouve dans le contenu des fichiers du projet les mentions "mysite.settings", "mysite.urls" et "mysite.wsgi.application". Dans ces conditions, il est valable d'adopter ce principe d'organisation :

- Mettre au niveau `mysite.appC` les applications spécifiques au projet, c'est-à-dire celles qui sont écrites spécialement dans le cadre de ce projet et pour en implémenter les besoins sur-mesure.
- Mettre au niveau `appB` les applications développées ou réemployées à l'occasion du projet, car elles sont suffisamment génériques pour être factorisées avec d'autres projets, que ce soit déjà le cas ou qu'elles en présentent le potentiel. Dans cette catégorie, on trouve souvent des utilitaires de journalisation, de prise de traces de mise au point, etc.

Cas appD

L'idée de ce niveau d'emplacement est de rassembler des applications sous un même répertoire, pour faire des catégories ou des familles, et aider à la compréhension de la structure du projet. Un usage est de le considérer comme un enrichissement du cas `appB`. Un autre usage est d'y loger des applications de source extérieure, notamment des applications issues de dépôts de sources. Le nom du répertoire doit être soigneusement choisi pour qualifier clairement les applications contenues. Un exemple typique est d'avoir un répertoire `external-apps` pour héberger les applications d'origine externe.

Comme pour `appC`, il s'agit de module de deuxième niveau, dont il faudrait mentionner le chemin. Mais parfois, référencer le chemin complet n'est pas souhaitable, car le répertoire n'est qu'un confort de structuration et traîner son nom ne fait qu'alourdir l'écriture sans apporter de réelle valeur technique. Un moyen pour remédier à cette situation est d'ajouter le répertoire parmi les chemins de recherche Python. Concrètement, cela peut se réaliser par la définition de la variable d'environnement `PYTHONPATH`, qui va enrichir ainsi la variable `sys.path`. Le positionnement de la définition peut s'automatiser par un script d'ouverture de session de console :



© Editions ENI - All rights reserved

Voici un exemple de contenu du fichier `setenv.bat` :

```
@echo off
set PYTHONPATH=external-apps
```

4.2 Création

Que le projet soit constitué d'une application unique ou qu'il soit un assemblage de plusieurs applications, il se dégage toujours une application principale, qui formera le cœur du projet, les autres applications étant des auxiliaires ou des périphériques. Il faut donner un nom à cette application, qui devrait permettre de la distinguer aisément parmi toutes les autres.

Un usage répandu est de reprendre le nom du projet ou du site comme nom de l'application principale. Ce n'est nullement une obligation. On peut éventuellement reprocher à ce choix d'exposer par inattention à une confusion parmi des répertoires de même nom ou de noms similaires.

Une convention alternative va être employée ici, en choisissant le terme `main`, par simple analogie avec le `main()` du langage C ou encore avec la notion de `__main__` en Python.

De cette manière, quel que soit le projet sur lequel on intervient, on repère immédiatement l'application qui joue le rôle principal, équivalent à un point d'entrée dans un programme.

En application des raisons exposées à la section précédente, l'application sera placée dans le répertoire `mysite`.

▶Créez une application :

```
D:\dj>cd mysite
D:\dj\mysite>py ..\manage.py startapp main
```

Notez que le fichier `manage.py` doit être pointé dans le répertoire parent du répertoire courant.

Cette structure est arrivée :

```
mysite\
└─ main\
   ├─ migrations\
   │  └─ __init__.py
   ├─ __init__.py
   ├─ admin.py
   ├─ apps.py
   ├─ models.py
   ├─ tests.py
   └─ views.py
```

Juste pour prouver l'état de bon fonctionnement, une première vue est mise en place, pour servir une simple page d'accueil.

- Créez le répertoire `main\templates`.

Le terme exact `templates` doit être respecté, car c'est une convention de nommage de l'infrastructure logicielle.

- Créez la page `main\templates\index.html`, avec ce contenu minimaliste :

```
<html>
Bienvenue
</html>
```

- Ajoutez ce contenu dans le fichier des vues `main\views.py` :

```
def index(request):
    return render(request, 'index.html')
```

Seul le nom du fichier suffit, sans plus de précision sur son emplacement. Ce fait s'explique plus loin par la déclaration de l'application dans la configuration du projet.

© Editions ENI - All rights reserved

- Complétez le fichier des routes pour détenir l'extrait de contenu `mysite\urls.py`.

```
from mysite.main import views

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.index),
]
```

Dans une écriture plus régulière et recommandée, le circuit serait passé par le maillon intermédiaire constitué d'un fichier `main\urls.py` à créer, qui aurait été référencé par une fonction `include()`, mais cette approche simplifiée peut être tolérée ici.

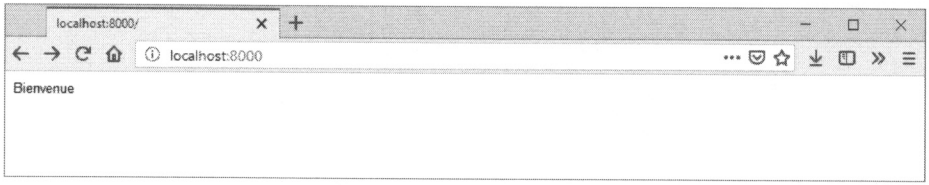
- Ajoutez dans le fichier de configuration `mysite\settings.py` la ligne mise en valeur :

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'mysite.main',
]
```

Parmi tous les effets induits par cet ajout, celui qui est intéressant ici est de permettre au moteur de gabarit d'explorer l'emplacement conventionnel dans l'arborescence de fichiers de cette application, lors de sa phase de recherche des fichiers de gabarits.

Il ne reste plus qu'à vérifier que cette vue fonctionne.

- Allez sur le site http://localhost:8000 pour y constater la page servie :



5. Paramètres de configuration

Un embryon de site est désormais disponible, suffisant pour servir de terrain d'essais. Les capacités de configuration vont à présent être revues. Elles sont offertes via le fichier `mysite\settings.py`.

5.1 Configuration minimale

La documentation montre qu'il existe de l'ordre de 150 paramètres de configuration. Ces paramètres ont pratiquement tous une valeur par défaut, au cas où le fichier ne positionne pas un choix. Il existe cependant quelques rares exceptions pour lesquelles soit la valeur par défaut soit une combinaison avec la valeur d'un autre paramètre empêchent le démarrage du serveur.

`SECRET_KEY` : la chaîne vide par défaut doit être supplantée par une véritable valeur, censée être unique à un projet. Sinon, cela provoque l'erreur :

```
django.core.exceptions.ImproperlyConfigured:
 The SECRET_KEY setting must  not be empty.
```

`ALLOWED_HOSTS` : la liste vide par défaut n'est acceptée que si `DEBUG` est positionné à `True`, qui n'est pas sa valeur par défaut. Dans le cas contraire, il faudra fournir une liste valorisée, sinon le refus se traduit par le message :

```
CommandError:
 You must set settings.ALLOWED_HOSTS if DEBUG is False.
```

L'assistant de création de projets a fourni un socle minimal de départ, avec le positionnement de certains des paramètres à une valeur estimée usuelle.

Quelques valeurs sont pourtant déjà celles par défaut, mais cela peut se comprendre lorsque ce n'est pas la situation pour d'autres paramètres et que la réunion de ces paramètres forme un lot cohérent, à lire et interpréter ensemble.

© Editions ENI - All rights reserved

Même lorsque le serveur a pu démarrer, on pourrait vite être confronté à des valeurs insuffisantes au moment de servir une requête. Par exemple :

```
AttributeError: 'Settings' object has no attribute 'ROOT_URLCONF'
```

ROOT_URLCONF : en effet, sa valeur par défaut est non définie. Il ne faut pas y voir une erreur par omission, car une configuration de routage pourrait être adjointe dynamiquement à la requête par un intergiciel et aurait priorité. Il s'agit d'un mode de fonctionnement très avancé, le cas ordinaire reste de définir l'arbre de routage à l'aide du fichier de configuration.

Dès l'instant où ce paramètre sera positionné, d'autres chargements vont vouloir se faire en cascade, ce qui nécessite encore le positionnement d'autres paramètres à des valeurs significatives :

```
LookupError: No installed app with label 'admin'.
```

INSTALLED_APPS : la liste vide par défaut ne suffit pas. Puisque le routage actuel établit une route vers le module intégré d'administration, il est nécessaire de mentionner `django.contrib.admin` et, par dépendances, `django.contrib.auth` et `django.contrib.contenttypes`. D'autre part, puisque le but est de servir une page d'une application, il faut bien sûr mentionner celle-ci : `mysite.main`. Le rang de classement d'une application dans la liste peut avoir une influence, car si une ressource est potentiellement livrable par plus d'une application, celle citée en début de liste a la priorité. La pratique conventionnelle est de mettre dans l'ordre : les applications de l'infrastructure logicielle (celles paquetées en `django.contrib`), les applications externes, et enfin les applications de son projet. Dans la plupart des cas, il s'agit d'un conflit involontaire de nommage qu'on résout plus facilement en choisissant un autre nom dans son application. Dans de rares cas, cette priorité peut être mise à profit pour substituer une ressource dans une application de source externe sans devoir modifier son code.

```
django.template.exceptions.TemplateDoesNotExist: index.html
```

TEMPLATES : la liste par défaut étant vide, aucun moteur de rendu n'est en fonction. Il faut au minimum un moteur, déclaré avec les sous-éléments suivants.

BACKEND : parmi les deux moteurs intégrés, `DjangoTemplates` est le moteur historique, donc le plus répandu.

`APP_DIRS` : la valeur par défaut est `False`. Il faut donc la positionner à `True` pour susciter la découverte du fichier gabarit à l'intérieur de l'application.

```
'django.contrib.auth.context_processors.auth' must be in TEMPLATES
in order to use the admin application.
```

Ce dernier détail se résout avec la présence du processeur mentionné parmi la liste `context_processors` sous `OPTIONS`.

Pour résumer, et en admettant l'état actuel du site comme une simple base de départ qui devra être enrichie, la page d'accueil de démonstration peut être délivrée avec cette configuration réduite :

```
SECRET_KEY = 'x%5a(9qaa8<SECRET_A_NE_JAMAIS_PUBLIER>cxrtg%=)&&'
DEBUG = True
INSTALLED_APPS = [
 'django.contrib.admin',   # pour urls.py
 'django.contrib.auth',    # dépendance de admin
 'django.contrib.contenttypes',    # dépendance de admin
 'mysite.main',
]
ROOT_URLCONF = 'mysite.urls'
TEMPLATES = [
 {
   'BACKEND': 'django.template.backends.django.DjangoTemplates',
   'APP_DIRS': True,
   'OPTIONS': {
     'context_processors': [
       'django.contrib.auth.context_processors.auth',
     ],
   },
 },
]
```

On aurait éventuellement pu pousser encore plus loin la simplification en éliminant dans le projet toute référence à l'administration, mais l'idée se limitait à citer une configuration minimale viable, sans avoir à intervenir sur d'autres portions du projet.

© Editions ENI - All rights reserved

5.2 Configuration par défaut affinée

L'objectif ici est de parcourir à nouveau le fichier de configuration tel qu'il est donné par l'assistant de création de projet, pour revenir sur certains paramètres et les expliquer.

La documentation mentionne la valeur par défaut, lorsqu'il y en a une, de chacun des paramètres. Attention toutefois à rester vigilant sur sa pertinence. Ce n'est pas nécessairement la meilleure valeur, car lorsqu'un paramètre est introduit par une nouvelle version, il peut avoir été donné priorité au mode de fonctionnement antérieur de façon à garantir la compatibilité arrière, alors que pour un nouveau projet, un autre choix, plus moderne, peut apporter un net progrès.

`BASE_DIR`

Cette entrée ne correspond nullement à un paramètre de configuration. Il s'agit de la définition d'une valeur, amenée certainement à être utilisée plusieurs fois dans la suite du fichier et donc établie une seule fois. S'agissant d'une valeur constante, il est dans les conventions Python de nommer la variable en majuscules et avec le caractère de soulignement. Par contre, pour signifier que la variable n'a pas vocation à être publique, mais qu'elle est réservée à un usage interne, il est plus explicite de préfixer son nom par un caractère de soulignement. Les occurrences de la variable seraient alors renommées en `_BASE_DIR`. Mais on s'aperçoit que le seul usage actuel de la variable est situé dans le paramètre `DATABASES`, pour le moteur `sqlite3` qui n'est pas nécessaire jusqu'à présent et qui sera écarté plus tard au profit d'un autre moteur. Finalement, le plus simple est de mettre hors service les éléments `BASE_DIR` et `DATABASES` en les basculant en commentaire.

Le répertoire ciblé par la variable correspondait à `D:\dj\`, dont l'intérêt est faible, car la pratique révèle que les ressources visées sont plutôt situées sous le répertoire du site. En conséquence il va être fait usage d'une autre variable, qui désignera `D:\dj\mysite\`, appelée `_SITE_ROOT` pour bien signifier la racine du site.

▶Modifiez le contenu du fichier de configuration pour aboutir à :

```
# BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(_[...]
_SITE_ROOT = os.path.dirname(os.path.abspath(__file__))
```

`ALLOWED_HOSTS`

Il a été vu précédemment qu'une liste vide est déjà la valeur par défaut, tolérée dès lors que `DEBUG` est à `True`. Passer la ligne en commentaire ne dégradera rien.

`INSTALLED_APPS`

L'infrastructure logicielle intègre une quinzaine d'applications, car bien qu'étant optionnelles, elles ont été estimées d'un usage suffisamment courant pour faire partie intégrante du produit. On a pu constater précédemment que quelques-unes étaient nécessaires pour démarrer le serveur intégré. Les autres sont traditionnellement employées et c'est pourquoi elles font partie du gabarit :

- `staticfiles` pour la gestion des fichiers auxiliaires que sont les feuilles de style CSS, les images et les bibliothèques de code JavaScript. Le but est essentiellement de faciliter et d'automatiser, par de l'outillage, le déploiement de ces ressources sur un serveur frontal de médias. Il n'est pas de servir ces ressources, même si, pour rendre la vie facile aux développeurs, l'application enrichit le serveur de développement intégré pour qu'il puisse directement trouver les ressources à leur source.
- `messages` pour des messages de notification, typiquement de succès ou d'échec, suite à une soumission de la part de l'utilisateur, un formulaire par exemple. Ces messages sont mémorisés transitoirement pour être affichés sur une page à suivre.
- `sessions` pour maintenir des informations relatives à la session de l'utilisateur, autrement perdues d'une requête à l'autre. Les informations restent sur le serveur et seul un cookie, détenant un identifiant, est échangé entre le navigateur et le serveur. L'application intégrée d'authentification des utilisateurs, `auth`, habituelle lorsque le site a besoin d'un mécanisme d'inscription d'internautes, a une dépendance envers cette application. Lorsque le site ne fait pas usage de `auth` et qu'aucune application par ailleurs n'utilise une conservation de données de session, on peut sans crainte mettre en commentaire cette application et ainsi économiser au serveur un peu de travail inutile.

© Editions ENI - All rights reserved

`MIDDLEWARE`

Le chapitre Intergiciels est dédié à ce sujet. Il donne plus de détails sur le rôle de ces éléments et propose d'en écrire un.

Les intergiciels standards disponibles sont de l'ordre d'une vingtaine et sont situés à deux endroits :

```
django\middleware\
django\contrib\<nom>\middleware.py
```

La liste établie dans le gabarit en mentionne environ un tiers d'entre eux. Ce sont les plus courants, en particulier ceux relatifs à la sécurité, qu'il est bon de laisser en place si on n'a pas une parfaite connaissance de leur rôle. Toutefois, ceux sous la catégorie `contrib` peuvent a priori être désactivés lorsque l'application correspondante l'est aussi dans `INSTALLED_APPS`.

Bien que n'étant pas donnés dans la liste par défaut, d'autres intergiciels présentent un intérêt certain :

- `django.middleware.gzip.GZipMiddleware`

  Il permet la compression des données échangées avec le navigateur pour gagner en performances. Sa mise en place pourra par exemple combler une recommandation d'un outil d'analyse de la vitesse d'un site, tel que PageSpeed Insights.

- `django.middleware.locale.LocaleMiddleware`

  Il permet la détermination de la langue préférentielle du navigateur. Pour plus de détails, se référer au chapitre consacré à l'internationalisation.

- `django.middleware.common.BrokenLinkEmailsMiddleware`

  Il permet l'envoi d'un courriel aux administrateurs à la détection d'un lien vers une page inexistante. Cette remontée d'alerte donne la possibilité d'être plus réactif sur la correction de liens abîmés.

Le rang de classement des entrées dans la liste a son importance, pour assurer un fonctionnement cohérent de l'ensemble. Les règles sont assez compliquées, heureusement, la documentation fournit une section qui synthétise un classement préférentiel avec ses justifications.

`TEMPLATES.DIRS`

Une liste des répertoires que le moteur doit balayer dans sa recherche de fichiers gabarits. L'usage typique est d'y mentionner un répertoire servant aux gabarits de base du site, notamment ceux qui établissent la structure globale des pages (menus, barres, en-tête, pied de page, etc.).

- Créez un répertoire `templates` sous `D:\dj\mysite\`.
- Modifiez le contenu pour aboutir à :

```
'DIRS': [os.path.join(_SITE_ROOT, 'templates')],
```

`TEMPLATES.OPTIONS.context_processors`

En plus de celui de `auth` cité précédemment, d'autres processeurs sont proposés par défaut :

- `django.template.context_processors.debug`
  Les variables mises dans le contexte sont destinées à conditionner des choix de présentation au fait d'être en mode `DEBUG`. À ce stade, il n'est pas dit que cela sera exploité, donc autant désactiver ce processeur en le passant en commentaire.
- `django.template.context_processors.request`
  Ce processeur est un classique, car il met à disposition la variable `request`, porteuse d'informations utiles sur les caractéristiques de la requête. Une caractéristique est soit présente par nature, par exemple `request.POST`, soit ajoutée par un intergiciel d'une application, par exemple `request.user` par l'application `auth`.
- `django.contrib.messages.context_processors.messages`
  Ce processeur est un élément de la fonctionnalité offerte par l'application `messages`, utile à conserver.

© Editions ENI - All rights reserved

`WSGI_APPLICATION`

Ce paramètre n'a d'usage que pour le serveur intégré, de façon à ce que celui-ci se conforme au protocole Python WSGI (spécifié dans le document *Python Enhancement Proposal* PEP 3333). L'assistant de projet a fait pointer le paramètre vers un objet dans le fichier `wsgi.py`, dont l'implémentation par défaut désigne précisément l'application WSGI par défaut lorsque le paramètre n'est pas défini. Le mettre en commentaire procure donc un raccourci et on constate un fonctionnement apparemment identique.

Il existe pourtant une petite nuance : le passage par `wsgi.py` positionne la variable d'environnement `DJANGO_SETTINGS_MODULE`, indispensable pour la suite des chargements de modules, mais comme `manage.py` positionne également cette variable auparavant, il n'y a globalement pas de différence. Il n'a vraiment que si on a personnalisé le point d'entrée procuré par `wsgi.py`, ce qui est inusuel, qu'il est bon de le faire utiliser par `runserver` pour coller au plus près au contexte du serveur de production.

`DATABASES`

Tant que le besoin de la base de données ne se fait pas sentir, le paramètre peut soit rester tel qu'il est proposé par l'assistant, soit être désactivé.

`AUTH_PASSWORD_VALIDATORS`

Ce paramètre est lié à l'application `auth` et liste des composants en charge d'estimer la force d'un mot de passe (longueur, chiffres, mots courants, etc.). L'assistant de projet a renseigné la liste avec tous les validateurs intégrés disponibles pour maximiser la sécurité et c'est très bien ainsi.

`LANGUAGE_CODE`
`TIME_ZONE`

L'assistant n'ayant pas de source d'information pour personnaliser ces paramètres, ils sont fournis avec une valeur arbitraire.

▶Ajustez les valeurs pour respectivement `'fr'` et `'Europe/Paris'`.

`USE_I18N`

`USE_L10N`

Ces paramètres sont mis par défaut à `True`, mais ils ont trait à l'internationalisation, ce qui n'est pas nécessaire à ce stade. L'occasion sera donnée de revenir dessus puisque le traitement de ce sujet fait l'objet d'un chapitre ultérieur. Dans l'immédiat, il est préférable de mettre hors service cette capacité.

▶Basculez les valeurs sur `False`.

`USE_TZ`

Ce paramètre demande à l'infrastructure logicielle de manipuler par défaut des objets date et heure porteurs d'un fuseau horaire. Il est surtout là par raison de compatibilité arrière (il a été introduit en version 1.4), d'où une valeur par défaut à `False`.

Vouloir revenir à un fonctionnement en objet dit naïf, c'est-à-dire ignorant de son fuseau, n'est pas la situation usuelle. Il est préférable de laisser par principe ce paramètre activé à `True`, même si on est convaincu d'utiliser les heures locales d'un unique fuseau.

`STATIC_URL`

Si l'application `staticfiles` est employée, alors le paramètre doit avoir une valeur et celle établie par l'assistant convient bien. En cas d'incohérence, un message d'erreur est produit :

```
django.core.exceptions.ImproperlyConfigured: You're using the
staticfiles  app without having set the required STATIC_URL setting.
```

© Editions ENI - All rights reserved

6. Variations de configuration

Le site est amené à tourner selon différents contextes, par exemple : développement, tests, intégration, production. Ces contextes induisent inévitablement des paramètres de configuration adaptés précisément à chaque situation, ne serait-ce que pour la connexion à la base de données.

A priori, toutes les variations de paramétrage doivent être concentrées dans le fichier `settings.py`, puisque c'est son rôle. La question se ramène donc à viser des exemplaires différents de ce fichier ou à influencer son contenu.

Parmi les nombreuses façons de traiter le sujet, quelques-unes vont être exposées succinctement, réparties par famille.

Les techniques par désignation

Dans cette famille, l'objectif est d'aboutir à cibler précisément un contenu de fichier, final et complet, soit parce que c'est le seul, soit en suivant des critères de sélection.

La solution la plus rustique, à condition que la situation s'y prête, est simplement de désigner toujours un fichier de même nom, au même emplacement, mais dans lequel on a mis un contenu différent, spécifique à la machine hôte, qu'elle soit réelle ou virtuelle. Ceci suppose de dédier un environnement hôte à un contexte, de façon stable dans le temps.

La contrepartie de cette apparente simplicité est le conflit d'intégration avec un système de gestion de versions, puisqu'une même ressource devrait accueillir plusieurs contenus. On est alors obligé de se replier sur une gestion manuelle ou annexe, sujette à des risques d'oubli, d'écrasement, de mélange, etc. Encore d'autres difficultés apparaissent avec l'emploi d'outils d'automatisation pour les tests et les déploiements.

Pour remédier au souci de ressource unique, une solution consiste à associer à chaque environnement son fichier dédié. Le problème se déplace alors sur le moyen de déterminer, d'une part, l'environnement de lancement et, d'autre part, d'en déduire le bon fichier parmi une collection.

Les façons d'évaluer l'environnement sous lequel le code tourne sont nombreuses, en voici quelques exemples :

– Selon le chemin du répertoire :

```
import os
_SITE_ROOT = os.path.dirname(os.path.abspath(__file__))
_PATH_ENV_MAP = {
    'D:\\dj\\mysite': 'dev',
    '/path/to/prod/mysite': 'prod',
}
_env = _PATH_ENV_MAP[_SITE_ROOT]
```

– Selon le compte utilisateur :

```
import getpass
_USER_ENV_MAP = {
    'Patrick': 'dev',
    'www': 'prod',
}
_env = _USER_ENV_MAP[getpass.getuser()]
```

– Selon le nom de la machine :

```
import platform
_NODE_ENV_MAP = {
    'PC-Patrick': 'dev',
    'web-srv': 'prod',
}
_env = _NODE_ENV_MAP[platform.node()]
```

– Selon une variable d'environnement de son choix :

```
import os
_env = os.environ.get('APP_ENV', 'dev')
```

Avec un positionnement de la variable tel que (sous Windows) :

```
> set APP_ENV=prod
```

Ou (sous Unix) :

```
$ export APP_ENV=prod
```

© Editions ENI - All rights reserved

Cette clé d'environnement sert ensuite pour viser le chargement de tel ou tel fichier de configuration. Dans cet exemple, on suppose que ces fichiers sont situés dans le même répertoire que `settings.py` et se nomment selon le motif `settings_<environnement>.py`, d'où ce schéma :

```
mysite\
├ __init__.py
├ settings.py
├ settings_dev.py
└ settings_prod.py
```

```
from importlib import import_module
_module = import_module('.settings_{}'.format(_env), __package__)
for _name in dir(_module):
    if _name[0].isupper():
        locals()[_name] = getattr(_module, _name)
```

Le travail de ce code consiste à importer localement les attributs du module ciblé. Sachant que tous les paramètres de Django ont leur nom tout en majuscules, on ne retient que ceux dont le nom débute par une majuscule, ce qui suffit pour exclure les éventuels attributs purement internes, notamment les variables de travail débutant par un caractère de soulignement (cf. conventions établies par le document Python Enhancement Proposal PEP 8).

En phase de mise au point, on peut vérifier si un résultat correct est produit, en activant cette ligne de traces en fin de code, dotée d'un filtre pour réduire le volume à ce qui est intéressant à observer :

```
print(dict([(k,v)for(k,v)in locals().items() if k[0].isupper()]))
```

En accordant le code en conséquence, d'autres répartitions sont valables, par exemple :

```
mysite\
├ __init__.py
└ settings\
  ├ __init__.py
  ├ dev.py
  └ prod.py
```

Dans cette disposition, le code anciennement dans `settings.py` est migré dans `settings\__init__.py` et la mise au format du nom de module devient `'.{}'`.

Lorsque l'emploi de variables d'environnement est une technique admise pour le projet, il existe une possibilité encore plus directe de désigner la configuration, puisque l'infrastructure logicielle prévoit une variable optionnelle à cet usage.

Il s'agit de la variable nommée `DJANGO_SETTINGS_MODULE`, avec pour valeur par défaut : `'mysite.settings'`.

Plutôt que de chercher à alimenter dynamiquement le module par défaut comme cela a été fait précédemment, il suffit de positionner cette variable dans son environnement de travail.

Par exemple, pour travailler en contexte de développement :

```
> set DJANGO_SETTINGS_MODULE=mysite.settings_dev
```

Ce mode est supporté aussi bien dans l'interfaçage avec un serveur web frontal, à travers le module `wsgi.py`, que pour les commandes en `manage.py`.

De plus, les commandes en `manage.py` disposent de l'option `--settings` pour imposer prioritairement une valeur. Par exemple :

```
D:\dj>py manage.py runserver --settings=mysite.settings_tests
```

Si on n'y prend pas garde, un désavantage évident de ces techniques est l'aspect répétitif des déclarations : de nombreux paramètres sont dupliqués à l'identique dans chacun des environnements, ce qui n'a aucune valeur en soi et au contraire est une charge en termes de maintenance, avec un potentiel risque de divergence en cas d'oubli. La famille de techniques suivante apporte un gain sur ce point.

Sinon, il est possible de concentrer les paramètres communs dans un fichier de base, par exemple `defaults_settings.py`, et le référencer dans chacune des configurations :

```
from mysite.defaults_settings import *
# ... suite
```

© Editions ENI - All rights reserved

Les techniques par superposition

En réalité, les différences de paramétrage d'un environnement à l'autre ne sont pas si nombreuses. Maintenir des exemplaires séparés engendre une forte duplication d'informations pourtant destinées à rester durablement identiques. L'optimisation recherchée est une factorisation : concentrer ces paramètres communs en un contenant unique servant de base et apporter seulement un différentiel, par nature en faible quantité, pour les éléments vraiment spécifiques à l'environnement. Ce complément peut se faire par addition, mais cela supposerait que chacun des environnements valorise chacune des pièces d'information, d'où encore une possible source de duplication. Il est en général plus efficace de procéder par écrasement d'une valeur par défaut.

Il est donc supposé la présence du fichier `settings.py` ordinaire, servant de base commune à tous les environnements, dont voici un extrait pour exemple :

```
"""
Django settings for mysite project.
"""

# ...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'dbname_prod',
        'USER': 'user_prod',
        'PASSWORD': 'pwd_prod',
    }
}
# ...
# [ici, en fin de fichier, un complément à venir]
```

Ce fichier de base est censé héberger le maximum de définitions, si possible toutes les définitions, avec pour valeurs par défaut celles qui comptent le plus, c'est-à-dire celles de l'environnement de production.

La formulation la plus basique du complément à placer en fin du fichier s'exprime ainsi :

```
# ...
from mysite.settings_local import *
```

Chaque environnement détient son exemplaire personnel du fichier `settings_local.py`. Dans l'idéal, celui de l'environnement de production est simplement vide, puisque pour lui tout est déjà dans le contenu commun.

Pour un environnement de développement, on pourra alimenter la configuration locale comme dans cet exemple :

`settings_local.py`

```
DEBUG = True
DATABASES = {
   'default': {
       'ENGINE': 'django.db.backends.mysql',
       'NAME': 'project_dev',
       'USER': 'user_dev',
       'PASSWORD': 'pwd_dev',
   }
}
# ... et d'autres
```

Le fichier général `settings.py` peut être géré sous un système de gestion de versions, mais cela ne pourra pas être le cas pour les multiples fichiers `settings_local.py`. Au mieux, pour documenter la mise en place d'un nouvel environnement (aide à l'arrivée sur le projet d'un nouveau développeur, par exemple), on peut placer sous la gestion de versions un fichier du genre `settings_local.readme`, contenant toutes les instructions utiles à la mise en œuvre du mécanisme et à son fonctionnement.

En contrepartie de la simplicité, l'instruction d'importation avec le caractère `'*'` a le petit tort d'éventuellement ramener trop de définitions, au mieux inutiles, au pire indésirables.

Dans la plupart des cas, l'excédent est sans conséquence. Mais si on veut être plus pur, le mécanisme peut être amélioré.

Une première amélioration consiste à ajouter un filtre, pour ne considérer que les véritables définitions de paramètres, en sachant que leurs noms sont en lettres majuscules.

© Editions ENI - All rights reserved

La formulation de la fin du fichier setting.py devient alors :

```
# ...
from mysite import settings_local
for attr in dir(settings_local):
 if not attr[0].isupper(): continue
 globals()[attr] = getattr(settings_local, attr)
```

Ceci est bien suffisant pour les paramètres à valeur unique, mais lorsqu'il s'agit de valeurs multiples sous forme de listes, il est dommage de devoir redéfinir toute la collection pour simplement ajouter une entrée. Il serait avantageux de se contenter de fournir des valeurs additionnelles.

La technique va donc être élaborée à l'aide d'une convention de nommage : préfixer le nom du paramètre par le motif `EXTRA_` signifie que les valeurs données doivent venir en addition aux valeurs de base.

Il est pris comme exemple la présence d'outils de mise au point dans un contexte de développement :

`settings_local.py`

```
EXTRA_INSTALLED_APPS = [
   'tools',
]
EXTRA_MIDDLEWARE = [
   'tools.middleware.DebugConnectionsMiddleware',
]
```

L'implémentation est enrichie ; la fin du fichier `settings.py` devient :

```
# ...
from mysite import settings_local
import re
for attr in dir(settings_local):
 if not attr[0].isupper(): continue
 value = getattr(settings_local, attr)
 match = re.search('^EXTRA_(\w+)', attr)
 if match:
   globals()[match.group(1)] += value
 else:
   globals()[attr] = value
```

Cette technique est restée longtemps suffisante, tant que les paramètres visés étaient constitués de simples valeurs de type primitif (chaînes, nombres) ou de type séquence (tuples, listes). La situation est devenue plus compliquée à partir de la version 1.8 avec notamment le cas du regroupement des paramètres individuels `TEMPLATE_CONTEXT_PROCESSORS`, `TEMPLATE_DEBUG`, `TEMPLATE_DIRS`, `TEMPLATE_LOADERS` et `TEMPLATE_STRING_IF_INVALID` dans un seul `TEMPLATES` selon une structure imbriquée de listes et dictionnaires.

Afin de traiter ces situations, la technique d'intégration doit être complétée. Des implémentations en sont proposées dans le chapitre Alternatives.

Il reste encore un sujet d'amélioration : si possible, il est toujours mieux d'anticiper les cas d'erreur et de donner explicitement des indications pour remédier au problème. Ici, l'erreur typique est d'oublier l'apport du fichier local. Elle peut se gérer par ce complément à la fin du fichier `settings.py`, pour imprimer un message explicatif et sortir proprement avec un code d'erreur :

```
# ...
try:
 from mysite import settings_local
except ImportError:
 print("""
   ------------------------------------------------------
    Vous devez créer un fichier settings_local.py
    et y mettre vos paramètres de configuration locale.
   ------------------------------------------------------
   """)
 import sys
 sys.exit(1)
else:
 # ...
```

© Editions ENI - All rights reserved

7. Création de l'application

Dans un premier temps, une structure minimale est mise en place, sachant que parfois il s'agit de contenus factices qui seront remplacés par la suite.

Le choix du nom de l'application n'est pas critique. Le terme `messages` est un candidat potentiel, mais il est préférable de l'éviter, car il est déjà employé par l'application intégrée `django.contrib.messages`. Pour rester dans le thème des communications, il est fait le choix du terme `messenger`.

Comme pour la première application, la nouvelle application sera placée dans le répertoire `mysite`.

▪Créez une application :

```
D:\dj>cd mysite
D:\dj\mysite>py ..\manage.py startapp messenger
```

La situation doit maintenant être :

```
mysite\
├─ main\
└─ messenger\
```

▪Sous `messenger\`, créez la hiérarchie de répertoires `templates\messenger\`.

Pourquoi créer à nouveau un sous-répertoire dans le répertoire des gabarits de l'application ? Parce que le moteur de gabarit, dans sa phase de recherche de fichier, va explorer les répertoires potentiels de chacune des applications et prendra la première correspondance trouvée. Or, il n'est pas rare que plusieurs applications emploient un même nom de fichier, usuel. C'est le cas avec le fichier `index.html`, destiné à être utilisé sous l'application `messenger`, mais qui est aussi déjà employé dans l'application `main`. Celle-ci étant mentionnée en tête dans la liste des applications, elle masque un éventuel homonyme dans les applications suivantes. Il est techniquement possible de faire en sorte que tous les gabarits aient un nom distinct, en utilisant des préfixes par exemple, mais cela alourdit l'écriture. L'emploi d'un sous-répertoire, faisant office d'espace de noms, est une convention répandue, on peut d'ailleurs le constater parmi les applications sous `django\contrib\`.

- Sous ce répertoire, créez cette page, avec ce contenu passe-partout :

`mysite\messenger\templates\messenger\index.html`

```
<html>
Bienvenue dans l'application messenger.
</html>
```

- Ajoutez ce contenu dans le fichier des vues :

`mysite\messenger\views.py`

```
def index(request):
    return render(request, 'messenger/index.html')
```

- Ajoutez dans le fichier de configuration `mysite\settings.py` la ligne mise en valeur :

```
INSTALLED_APPS = [
    # ...
    'mysite.main',
    'mysite.messenger',
]
```

Dans l'état, le serveur de développement n'est pas en mesure d'exploiter cette nouvelle application, car il manque des éléments de routage, objet du chapitre suivant.

8. Outillage de mise au point

Comme annoncé à l'occasion de la référence à l'outil IDLE (cf. chapitre Installation - Après l'installation), maintenant qu'un projet est créé et disponible, il est intéressant de pouvoir faire quelques expérimentations sur ce projet, directement avec des instructions Python en ligne de commande. Ceci permet d'évaluer unitairement une portion de code, de lever un doute sur sa compréhension du comportement d'un élément, d'explorer en profondeur une structure de données, etc.

© Editions ENI - All rights reserved

L'infrastructure logicielle propose justement pour cela une commande pour disposer d'un interpréteur, avec un environnement bien positionné, à l'image de celui du serveur de développement :

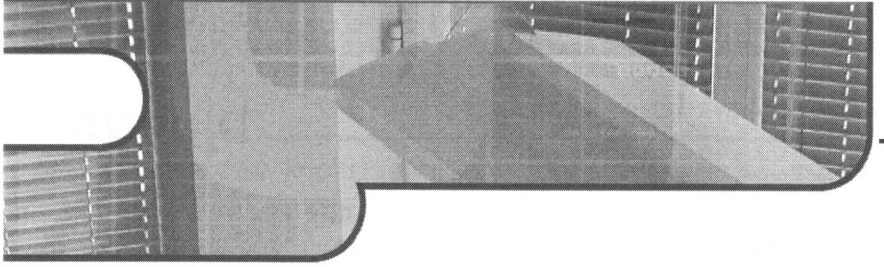
```
D:\dj>py manage.py shell
Python 3.7.2 (tags/v3[...]) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more i[...]
(InteractiveConsole)
>>>
```

Cette console permet de facilement vérifier une vue, avant même qu'elle soit associée à une URL :

```
>>> from mysite.messenger import views
>>>
>>> views.index(None)
<HttpResponse status_code=200, "text/html; charset=utf-8">
>>> views.index(None).content
b"<html>\nBienvenue dans l'application messenger.\n</html>"
>>>
```

Un objet `request` est normalement passé en paramètre des appels aux vues. N'en disposant pas ici, mais n'en faisant pas non plus usage, il suffit de passer n'importe quoi, comme `None`.

Le contenu de la réponse est bien celui espéré, encodé en chaîne binaire.



Chapitre 3
Routage

1. Présentation

Précédemment, pour avoir rapidement une preuve de fonctionnement du site, une page d'accueil a été mise en place, au plus simple. Ce point va maintenant être revu afin d'être développé.

En quelques mots, le routage consiste d'une part, dans le sens entrant, à établir une carte de correspondance entre les URL et des processus de traitement, et d'autre part, dans le sens sortant, à pouvoir aisément générer des URL bien formées.

Le système de routage peut être vu comme un arbre dont le point d'entrée est cité parmi les paramètres de configuration sous le nom `ROOT_URLCONF`. La notion de ramification est apportée par un mécanisme d'inclusion, qui fait que les définitions peuvent être réparties de façon hiérarchique.

Si on observe le fichier `mysite\urls.py`, on voit que l'assistant de création de projet propose d'office un routage vers les vues du module d'administration intégrée, avec l'emploi du terme `admin` en premier maillon des chemins d'URL :

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

Les maillons suivants des chemins sont établis par le paquet mentionné. Il est vain de vouloir aller plus avant dans l'intérieur de ce paquet, car il n'est pas représentatif de ce que nous avons personnellement à faire dans nos applications. En effet, `admin` étant un module devant découvrir son environnement, il construit essentiellement son routage de façon dynamique.

Jusqu'à présent le routage du site se limite à :

`mysite\urls.py`

```
path('', views.index),
```

Il est suffisant pour servir la vue à l'adresse `http://localhost:8000`.

A contrario, ce routage ne permet pas de servir autre chose que cette adresse. Si on essaye `http://localhost:8000/autre`, on déclenche une erreur « Page not found ».

Ce n'est pas une situation habituelle, mais si on souhaitait avoir une vue de repli pour toutes les URL indéterminées, il faudrait l'exprimer par une expression rationnelle :

`mysite\urls.py`

```
from django.urls import re_path
#...
 #...
 re_path('.*', views.index),
```

Puisque le parcours des variables `urlpatterns` est ordonné, ce motif d'URL doit bien sûr être placé en toute fin de liste.

Si l'intention était d'avoir un traitement des appels à des pages inconnues, il est alors plus régulier de laisser la requête échouer en code HTTP 404 et de référencer un traitement personnalisé, par exemple par une vue `vue404()`, en remplacement du traitement par défaut :

`mysite\urls.py`

```
from django.conf.urls import handler404
#...
handler404 = views.vue404
#...
```

© Editions ENI - All rights reserved

2. Configuration des adresses

L'objectif est de continuer la mise en place du routage, avec une capacité d'adresser les vues à venir de l'application `messenger`. Une façon traditionnelle est d'attribuer un maillon du chemin à une application, souvent avec le même terme que le nom de l'application pour faciliter la compréhension. Dans le cas présent, ce maillon s'écrit ainsi :

`mysite\urls.py`

```
from django.urls import include, path
#...
 #...
 path('messenger/', include('mysite.messenger.urls')),
```

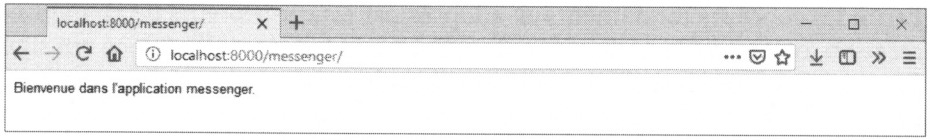
La fonction `include()` permet de déléguer la suite de la correspondance au module mentionné, qui lui-même peut faire un nouvel étage de répartition et ainsi de suite.

Une configuration minimale d'URL est préparée pour le module `messenger`, en créant son fichier de routage :

`mysite\messenger\urls.py`

```
from django.urls import path

from .views import index

urlpatterns = [
    path('', index),
]
```

- Allez à l'adresse http://localhost:8000/messenger/ pour y constater la page servie :



Le plan de routage va être enrichi en préparant des URL pour répondre aux opérations de base, à savoir : lister les dossiers de réception et d'envoi, écrire, voir et effacer un message.

En attendant de disposer des vues effectives, toutes ces URL mènent temporairement sur la seule vue de démonstration.

▶Complétez le fichier `mysite\messenger\urls.py`, pour contenir :

```
from django.urls import path
from django.views.generic import RedirectView

from .views import index

urlpatterns = [
    # dossiers
    path('inbox/', index, name='inbox'),
    path('sent/', index, name='sent'),
    # actions
    path('write/', index, name='write'),
    path('view/<int:id>/', index, name='view'),
    path('delete/', index, name='delete'),

    path('', Redirect.as_view(
      pattern_name='inbox', permanent=True)),
]
```

Un nom est adjoint à la plupart des URL. Ceci est destiné à la construction des hyperliens dans les gabarits de vues, en respectant le principe de ne définir le chemin de l'URL qu'une fois.

Toutes les URL finissent par le caractère barre oblique. Ce n'est ni une obligation ni une nécessité, mais un choix encouragé par l'infrastructure logicielle. En effet, d'un point de vue technique, les URL avec et sans barre oblique sont bien considérées comme deux objets différents, notamment par les moteurs de recherche ou les systèmes de cache. Il n'y a pas de raison forte pour dire qu'un système est meilleur que l'autre.

Parmi les concepts fondamentaux adoptés par les concepteurs de Django à sa création, on trouve la notion d'URL uniformisée (ou normalisée), ce qui signifie que l'important est d'être cohérent dans leur forme d'écriture, c'est-à-dire ne pas avoir tantôt l'une tantôt l'autre selon les pages, ni admettre indifféremment l'une et l'autre comme identiques.

© Editions ENI - All rights reserved

Pour cela, la terminaison par une barre oblique a été privilégiée, avec la mise à disposition de facilités pour garantir sa présence. C'est le rôle du paramètre de configuration `APPEND_SLASH`, ayant `True` comme valeur par défaut, d'activer une procédure de rattrapage : si l'URL de la requête ne correspond à aucun motif déclaré et si elle ne se termine pas par une barre oblique, alors un intergiciel (`CommonMiddleware`) se charge de remplacer ce qui devrait être une erreur de type HTTP 404 (*Page not found*) par une réponse de type redirection, vers la même URL avec une barre oblique ajoutée en fin. Il n'est pas dit que cette URL ainsi normalisée aboutisse à une ressource, mais au moins on lui aura donné une seconde chance.

Un effet auxiliaire d'adhérer à ce choix est de ne pas avoir d'interrogation sur la présence et la place de la barre oblique en tant que séparateur pour le mécanisme à base de fonctions `include()` ou, encore plus obscur, de construction dynamique : il suffit de toujours la mettre. Dans tous les cas, Django n'oblige pas à la présence conventionnelle de la barre oblique, le développeur reste maître de ses URL.

L'URL de consultation d'un message possède un paramètre capturé qui donnera l'identifiant unique de l'objet ciblé. Le convertisseur de type `int` y est précisé puisque seule une valeur numérique est attendue. Sinon, le convertisseur implicite est `str`, qui revient à capturer n'importe quelle chaîne non vide, à l'exception du caractère '/'.

Pour finir, il est établi une redirection permanente (code HTTP 301 plutôt que 302 pour le temporaire par défaut) vers la page de réception lorsque le motif n'est pas précisé. On peut le voir comme un raccourci.

On peut essayer ces URL :

- `localhost:8000/messenger/`
- `localhost:8000/messenger/inbox` (noter l'absence de barre finale)

et constater qu'elles sont ramenées par le jeu des redirections à :

- `localhost:8000/messenger/inbox/`

3. Espace de noms

Dans la section précédente, des noms ont été donnés aux chemins d'URL. Ceci permet de reconstruire une URL dans l'autre sens, à partir d'une référence à son nom. Cette faculté, appelée résolution inversée, est largement utilisée dans les gabarits de page. On en voit aussi ici un usage, dans le fichier de routage au sein de la définition de la redirection du motif vide vers la page de réception.

Se pose alors la question de l'unicité des noms. Dans la grande majorité des cas, on voudra que la référence à un nom désigne sans ambiguïté une URL bien précise. Le problème est que les noms employés peuvent être assez ordinaires, comme `index` ou `home`, pour se retrouver dans plus d'une application. Or, lorsqu'un nom est utilisé plusieurs fois, soit au sein d'une même application (mauvaise conception), soit dans plusieurs applications (conflit de nommage), le mécanisme de recherche retient la dernière occurrence trouvée dans la liste ordonnée `urlpatterns`.

Une solution basique serait d'ajouter un préfixe par convention, dans le même esprit que l'idée citée à l'occasion du problème de conflit de nommage des gabarits de vues, avec le même défaut d'alourdissement des noms. Mais il faut plutôt voir cette convention d'écriture comme un artifice tendant à réduire les risques de conflit, sans s'en protéger avec certitude. De plus, elle n'est pas adaptée à l'emploi d'applications externes, qu'on n'est pas censé devoir déformer pour les rendre compatibles avec son environnement.

À l'inverse de ce qui vient d'être dit, on peut aussi mettre à profit ce potentiel d'écrasement de nom pour des cas particuliers de surcharge volontaire, notamment s'agissant d'applications externes ou de celles sous `django.contrib`. Un exemple est la vue `LogoutView`, de `django.contrib.auth.views`, qui est associée au nom `logout`. On suppose vouloir intercepter les appels à cette vue de la part de Django ou d'autres applications, sans devoir intervenir sur le code source de ces appels. Sachant que les appels doivent normalement être initiés à l'aide d'une résolution inversée du nom, on peut surcharger l'association de ce nom avec une vue personnalisée. La condition à respecter est qu'elle soit déclarée après l'inclusion des URL originales.

© Editions ENI - All rights reserved

Par exemple :

mysite\urls.py

```
    path('normal/', include('django.contrib.auth.urls')),
    path('surcharge/', include('mysite.myauth.urls')),
```

mysite\myauth\urls.py

```
from .views import LogoutView
#...
    path('logout/', LogoutView.as_view(), name='logout'),
```

La bonne résolution des noms est constatée :

```
D:\dj>py manage.py shell
>>> from django.urls import reverse
>>>
>>> reverse('login')
'/normal/login/'
>>> reverse('logout')
'/surcharge/logout/'
```

Il est alors possible d'implémenter un traitement personnalisé dans la nouvelle vue, pour par exemple enregistrer des événements d'historique, de statistique, ou libérer des allocations de ressources, ou se garantir contre une perte d'informations en cas d'oubli de sauvegarde sur une page d'édition. Rien n'empêche ensuite de finir l'interception en déléguant le rendu final à la vue de déconnexion originale de `auth`.

Pour revenir au sujet d'unicité des noms d'URL, il existe une meilleure solution que d'ajouter des préfixes : utiliser des espaces de noms. Un espace de noms permet de contextualiser un nom utilisé, au moins potentiellement, à plus d'une reprise, pour désigner pourtant des cibles différentes. Ce concept donne accès à deux possibilités : 1) faire la distinction entre plusieurs applications qui utiliseraient un même nom ; 2) mettre en place plusieurs instances d'une même application. C'est pourquoi on parle d'espace de noms d'application et d'espace de noms d'instance. Il faut admettre qu'il est parfois un peu difficile de s'y retrouver dans la documentation et qu'on a vite fait de s'embrouiller entre les deux, parce qu'en réalité on trouve une alternance entre trois expressions proches : *application namespace*, *application instance*, *instance namespace*. Le tableau se complique avec des règles de repli sur des valeurs par défaut et de priorité de recherche inverse.

Il est normal de devoir lire et relire à de multiples reprises la documentation pour tenter de comprendre ces algorithmes.

L'emploi d'espace de noms est hautement conseillé et devrait toujours être le cas pour une application externe puisque, par définition, son auteur ne sait pas parmi quelles autres applications elle sera utilisée.

Heureusement, le cas présent va rester simple : il y a une seule instance de l'application et il suffira de se limiter à mentionner un espace de noms d'application.

Ce n'est pas une obligation, mais pour rester au plus simple, il est courant de prendre pour la chaîne nommant l'espace de noms d'application le nom même de l'application. Donc, sans surprise, l'espace de noms va être nommé par la chaîne de caractères `'messenger'`.

Dans le cas d'une inclusion d'un module de configuration d'URL, le nom se déclare par l'attribut `app_name` du module :

`mysite\messenger\urls.py`

```
#...
app_name = 'messenger'
urlpatterns = [
#...
```

Dans l'inclusion, il pourrait être précisé un espace de noms d'instance. Pour simplifier, cela ne va pas être fait, pour laisser prendre par défaut le nom de l'espace de noms d'application. Le fait d'avoir le même nom implique aussi qu'il s'agit de l'instance par défaut.

`mysite\urls.py`

```
#...
# path('messenger/', include(
#    'mysite.messenger.urls', namespace='messenger'
# )),
path('messenger/', include('mysite.messenger.urls')),
```

À partir de là, les URL vont pouvoir être référencées par un nom au sein d'un espace de noms, la syntaxe étant de la forme : `'espace:nom'`. Il peut éventuellement y avoir une succession d'espaces, avec toujours le caractère `':'` servant de séparateur.

© Editions ENI - All rights reserved

Par exemple, l'URL pour la boîte de réception se désigne par `'messenger:inbox'`, ce qui lève l'ambiguïté avec un possible autre `inbox` ailleurs.

Il faut ajuster en conséquence la référence mentionnée pour la redirection :

`mysite\messenger\urls.py`

```
    #...
    path('', Redirect.as_view(
      pattern_name='messenger:inbox', permanent=True)),
```

Encore un point à propos de l'algorithme de recherche inverse : dans notre configuration, supposons que nous déclarions un espace de noms d'instance autre que celui par défaut :

`mysite\urls.py`

```
#...
path('messenger/', include(
    'mysite.messenger.urls', namespace='nimportequoi'
)),
```

La référence avec l'espace de noms d'application aurait quand même ciblé notre instance en raison d'être la dernière déployée (au sens mentionnée dans le routage), et évidemment parce que c'est aussi la seule.

Notre besoin se contente d'une seule instance de l'application et c'est très bien ainsi, car en réalité notre routage comporte une faille si l'application est déployée plus d'une fois. La section suivante détaille cette faiblesse et donne une solution pour y remédier.

4. Instances multiples d'une même application

Pour rappel, nous n'avons pas le besoin de déployer plus d'une fois notre application, mais l'exercice va quand même être fait pour expérimenter le comportement de notre routage.

Pour mettre en œuvre l'application à deux reprises, il faut naturellement définir deux chemins d'URL distincts. On peut assez facilement se douter que si on ne définit aucun des espaces de noms des deux instances, sachant que toutes deux vont prendre par défaut l'espace de noms de l'application, cela crée une source d'ambiguïté.

En effet, les vérifications d'intégrité au démarrage du serveur détectent le caractère anormal de cette situation et un avertissement est produit :

```
WARNINGS:
?: (urls.W005) URL namespace 'messenger' isn't unique. You may
not be able to reverse all URLs in this namespace
```

On pourrait laisser cette attribution par défaut à l'une des instances, mais il est préférable de nommer explicitement les deux instances. L'analyse sera plus facile à conduire et les conclusions seront les mêmes.

mysite\urls.py

```
#...
path('messenger1/', include(
 'mysite.messenger.urls', namespace='inst1'
)),
path('messenger2/', include(
 'mysite.messenger.urls', namespace='inst2'
)),
```

Avec un navigateur on peut vérifier qu'une requête vers l'accueil de l'instance 2 est correctement redirigée, comme le montrent les traces du serveur :

```
[...] "GET /messenger2/ HTTP/1.1" 301 0
[...] "GET /messenger2/inbox/ HTTP/1.1" 200 54
```

La mauvaise surprise apparaît pour une requête vers l'accueil de l'instance 1, où on est redirigé vers l'instance 2 :

```
[...] "GET /messenger1/ HTTP/1.1" 301 0
[...] "GET /messenger2/inbox/ HTTP/1.1" 200 54
```

Ceci est tout à fait normal, car la définition de la redirection référence l'espace de noms de l'application, ce qui donne un potentiel de deux instances. Ici, en l'absence d'instance par défaut, l'instance retenue est la dernière déployée, soit `inst2`. Toutefois, définir une instance comme celle par défaut en lui donnant soit pas de nom, soit le nom de l'application, ne ferait toujours que satisfaire un chemin, mais pas l'autre.

© Editions ENI - All rights reserved

La solution au problème vient avec la possibilité d'assister le résolveur en lui disant quelle est l'instance courante. Pour cela il faut remplir deux conditions : premièrement, déterminer quelle est cette instance courante, à partir de l'objet `request`, et deuxièmement, alimenter l'attribut `current_app` de la fonction `reverse()`. Or, dans le fichier de routage il n'y a accès à aucune de ces informations et la vue générique `RedirectView` n'offre pas de tels réglages.

Il faut écrire une vue personnalisée pour reprendre la main sur le fonctionnement :

`mysite\messenger\urls.py`

```
#...
from .views import index, IndexView

   #...
   # path('', Redirect.as_view(
   #    pattern_name='messenger:inbox', permanent=True)),
   # Remplacé par :
   path('', IndexView.as_view()),  # redirige sur inbox
```

Ensuite, la vue supplémentaire est écrite, sur un héritage de `RedirectView` puisque seule une surcharge est nécessaire.

`mysite\messenger\views.py`

```
#...
from django.urls import reverse
from django.views.generic import RedirectView
#...
class IndexView(RedirectView):
   pattern_name = 'messenger:inbox'
   permanent = True

   def get_redirect_url(self, *args, **kwargs):
       return reverse(self.pattern_name,
           args=args, kwargs=kwargs,
           current_app=self.request.resolver_match.namespace)
```

Par rapport à la vue originale, on peut se permettre d'intégrer les deux attributs puisqu'on n'est plus dans le cadre d'une vue générique. La méthode `get_redirect_url()` est réécrite de façon simplifiée, car on ne tient plus compte des attributs `url` et `query_string`, mais surtout on a ajouté l'argument `current_app` qui permet d'assurer la continuité d'instance.

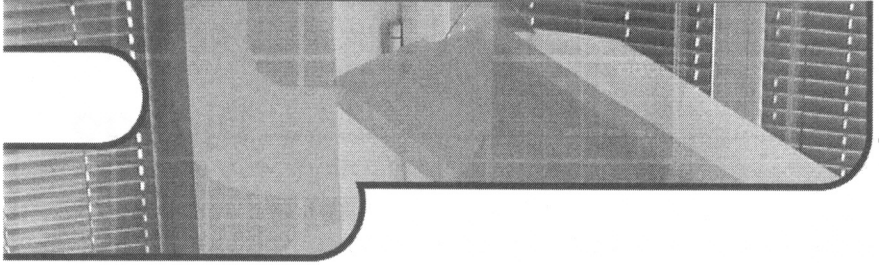
Remarque

*Veillez à cocher la case **Désactiver le cache** dans les outils de développement du navigateur, ou à forcer un rechargement complet de la page, pour ne pas récupérer d'anciennes redirections restées en cache.*

Avec un navigateur on peut vérifier que cette fois chacune des requêtes vers l'accueil d'une instance est correctement redirigée au sein de la même instance :

```
[...] "GET /messenger1/ HTTP/1.1" 301 0
[...] "GET /messenger1/inbox/ HTTP/1.1" 200 54
[...] "GET /messenger2/ HTTP/1.1" 301 0
[...] "GET /messenger2/inbox/ HTTP/1.1" 200 54
```

© Editions ENI - All rights reserved



Chapitre 4
Modèles

1. Introduction

Ce chapitre est consacré à la modélisation et au stockage des objets des applications. En dehors des objets intrinsèques à l'infrastructure logicielle, il va s'agir en l'occurrence de messages échangés entre des utilisateurs.

Chaîne de caractères : simple ou double guillemets ?

En Python, les chaînes de caractères littérales peuvent indifféremment être encadrées par des couples de guillemets simples ou doubles.

Le guide de style (cf. conventions établies par le document *Python Enhancement Proposal* PEP 8) ne donne pas de recommandation particulière d'emploi de l'une ou l'autre manière. Il indique juste que si la chaîne contient un guillemet d'un genre, utiliser l'autre genre pour l'encadrement épargne l'emploi du caractère d'échappement, ce qui est bénéfique à la lisibilité.

Les normes de style qu'il est demandé de respecter aux contributeurs du code de Django ne sont pas non plus directives, à part une préférence pour les simples guillemets, sauf si la chaîne en contient.

Dans ce chapitre et par la suite, il est tenté de suivre les conventions préférentielles suivantes, qui semblent correspondre aux exemples vus dans la documentation de Django et qui sont aussi communément adoptées :

- Les simples guillemets pour les noms à signification technique, tels que les identifiants, les noms de fichier ou de module.
- Les doubles guillemets pour les textes de langage (d'autant plus susceptibles de contenir le simple guillemet en langue française).

2. Instanciation de la base de données

Jusqu'à présent, il n'a pas été nécessaire de disposer d'une base de données. Au plus, un fichier `dj\db.sqlite3` existe, mais de taille nulle, généré en conséquence de la configuration par défaut donnée par l'assistant de création de projet. Il a été vu qu'on pouvait même désactiver le paramètre de configuration `DATABASES` et éliminer ce fichier.

Pour autant, il a été possible de préparer des vues et donc de servir des pages. Ceci démontre qu'il est tout à fait possible d'employer Django sans nécessairement devoir mettre en œuvre une base de données, notamment pour de simples sites délivrant de l'information statique, sans interaction avec les utilisateurs.

Pour l'application à réaliser, il va falloir un espace de persistance des messages et une base de données est le support privilégié pour satisfaire ce besoin.

2.1 Création de la base de données

▶ Ouvrez l'outil `pgAdmin4`.

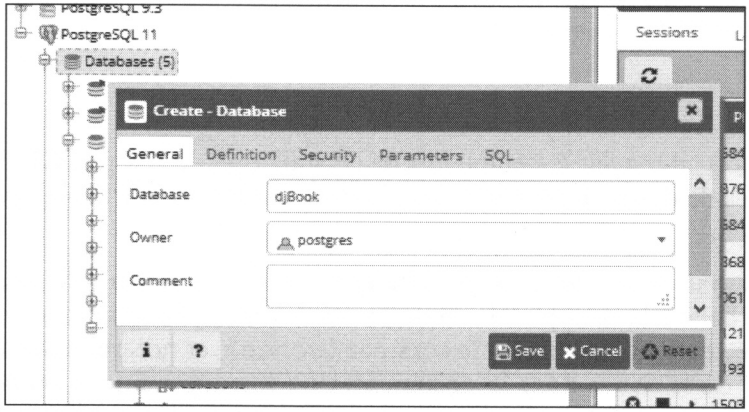
Pour rappel, il s'agit d'un outil d'administration en mode graphique, empaqueté avec l'installation de la base de données PostgreSQL, dont nous avions recommandé de déposer un raccourci sur le bureau. Au lancement, son interface est présentée dans un onglet du navigateur par défaut.

© Editions ENI - All rights reserved

À condition que, dans le menu hiérarchique de gauche, le focus soit détenu par un nœud compatible, à savoir un **Server Group** (comme **PostgreSQL 11** sur la copie d'écran plus loin) ou son descendant **Databases**, alors la section **Object** de la barre de menus, ainsi que le menu contextuel par clic droit, proposent une entrée **Create / Database...** pour amorcer la création d'une base.

Le nom de la base est quelconque, il est fait le choix de `djBook`.

Pour faire simple, on peut laisser ici le propriétaire de la base à l'utilisateur `postgres`, qui est le super-utilisateur établi par l'installateur. Mais concernant le site de production, pour des raisons de sécurité, il faut prévoir un compte utilisateur dédié à cet usage.



Dans l'onglet **Definition**, on peut constater que l'encodage par défaut est `UTF8`, ce qui convient parfaitement.

Les autres options restant à leur valeur par défaut, dans l'onglet **SQL** on peut observer la commande résultante :

```
CREATE DATABASE djBook
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    CONNECTION LIMIT = -1;
```

2.2 Référencement de la base de données

Maintenant que la base existe, elle peut être référencée dans les paramètres de configuration. Parmi les nombreuses options, les caractéristiques obligatoires à fournir sont ceux-ci :

settings.py

```
# ...
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'djBook,
        'USER': 'postgres',
        'PASSWORD': 'postgres',
    }
}
# ...
```

Le mot de passe à citer avait été établi lors de l'installation de PostgreSQL. Le serveur de bases de données est par défaut la machine locale, sur le port par défaut du moteur.

Il serait intéressant de vérifier tout de suite que la connexion à la base peut s'établir. Si on balaye la panoplie de commandes de l'outil d'administration, une idée prometteuse est d'exploiter la commande `check`, et notamment son étiquette (*tag*) `database`, dont la documentation mentionne deux points intéressants : 1) cela vérifie des problèmes de configuration liés aux bases de données ; 2) ce point de contrôle n'est pas lancé par défaut (lorsqu'on ne précise aucune étiquette), car il fait plus que de l'analyse statique de code.

En effet, tout semble bien se passer :

```
D:\dj>py manage.py check --tag database
System check identified no issues (0 silenced).
```

Sauf que, dans le cas du moteur PostgreSQL, le contrôle ne fait rien et donc aucune connexion à la base n'est tentée (en comparaison, dans le cas du moteur MySQL, un ordre est passé). Donc cette commande n'est pas en mesure d'alerter sur une erreur dans les attributs `NAME`, `USER` ou `PASSWORD`.

© Editions ENI - All rights reserved

Une autre commande est accessible, en rapport avec le sujet : `dbshell`. Son rôle est de lancer le logiciel client en ligne de commande propre au moteur choisi, mais surtout avec les paramètres de connexion déclarés, ce qui est justement le but recherché. La documentation informe que l'exécutable du logiciel client doit être dans le `PATH`, car il n'existe pas la possibilité de le spécifier explicitement. L'installateur du moteur ne prévoit pas de s'ajouter de façon permanente dans le `PATH` système.

La permanence n'est en effet pas indispensable et on peut se contenter de faire manuellement ce complément de définition de PATH, pour le temps d'emploi de la commande dans cette session en console :

```
D:\dj>path=%path%;C:\Program Files\PostgreSQL\11\bin
```

Avec la configuration par défaut de la console, le client fera apparaître cet avertissement au démarrage et sur certains affichages les caractères accentués seront mal reproduits :

```
Attention : l'encodage console (850) diffère de l'encodage
 Windows (1252).
 Les caractères 8 bits peuvent ne pas fonctionner correctement.
 Voir la section « Notes aux utilisateurs de Windows » de la page
 référence de psql pour les détails.
```

Pour éviter ce désagrément, il suffit de changer l'encodage console pour lui donner la valeur espérée par le client, à l'aide de la commande `chcp` (*CHangeCodePage*) :

```
D:\dj>chcp 1252
Page de codes active : 1252
```

Le client est accessible et la commande peut aboutir :

```
D:\dj>py manage.py dbshell
psql (11.1)
Saisissez « help » pour l'aide.

djBook=#
```

Cette commande confirme l'encodage attendu par le client :

```
djBook=# \encoding
WIN1252
```

Une commande permet de constater les caractéristiques de la connexion :

```
djBook=# \conninfo
Vous êtes connecté à la base de données « djBook » en tant
qu'utilisateur « postgres » sur l'hôte « localhost » via le port « 5432 ».
```

Pour quitter proprement :

```
djBook=# \q

D:\dj>
```

2.3 Première alimentation de la base de données

Normalement, la création de la base est la seule opération à faire manuellement, à part éventuellement la création d'un compte utilisateur. À partir de là, l'infrastructure logicielle se charge de tout le reste à suivre, dont la création des tables.

Avant d'avancer sur la construction de l'application, il est utile de mettre à niveau le contenu de la base avec ce que le site gère comme objets jusqu'à maintenant. Cette synchronisation se nomme migration dans la documentation.

Pour rappel, les applications déclarées sont :

settings.py

```
INSTALLED_APPS = [
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'mysite.main',
 'mysite.messenger',
]
```

Le serveur de développement ne se prive pas de nous rappeler nos devoirs à chaque démarrage :

```
D:\dj>py manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
```

© Editions ENI - All rights reserved

```
You have 14 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): admin, auth,
contenttypes.
Run 'python manage.py migrate' to apply them.
```

Pour commencer, il est bon de faire de l'observation pour identifier ces migrations. La commande administrative `showmigrations` est dédiée à cela et n'engage aucune modification :

```
D:\dj>py manage.py showmigrations
admin
[ ] 0001_initial
[ ] 0002_logentry_remove_auto_add
[ ] 0003_logentry_add_action_flag_choices
auth
[ ] 0001_initial
[ ] 0002_alter_permission_name_max_length
[ ] 0003_alter_user_email_max_length
[ ] 0004_alter_user_username_opts
[ ] 0005_alter_user_last_login_null
[ ] 0006_require_contenttypes_0002
[ ] 0007_alter_validators_add_error_messages
[ ] 0008_alter_user_username_max_length
[ ] 0009_alter_user_last_name_max_length
contenttypes
[ ] 0001_initial
[ ] 0002_remove_content_type_name
main
(no migrations)
messenger
(no migrations)
```

On visualise bien ces 14 migrations en attente. Si on a la curiosité de savoir ce que fait telle migration, on peut aller consulter son code, par exemple `django\contrib\admin\migrations\0001_initial.py`, mais c'est un peu pénible à lire, car il s'agit de code généré par un algorithme.

Une alternative est d'utiliser la commande `sqlmigrate` pour visualiser le code SQL qui sera appliqué. Les arguments de la commande sont le nom de l'application et le nom de la migration (lignes volontairement tronquées dans cet extrait d'affichage) :

```
D:\dj>py manage.py sqlmigrate admin 0001_initial
BEGIN;
--
-- Create model LogEntry
--
CREATE TABLE "django_admin_log" ("id" serial NOT NULL PRIMARY [...]
ALTER TABLE "django_admin_log" ADD CONSTRAINT "django_admin_l [...]
ALTER TABLE "django_admin_log" ADD CONSTRAINT "django_admin_l [...]
CREATE INDEX "django_admin_log_content_type_id_c4bce8eb" ON " [...]
CREATE INDEX "django_admin_log_user_id_c564eba6" ON "django_a [...]
COMMIT;
```

Sans avoir besoin de comprendre tous les détails, on devine aisément ce que cette migration est destinée à créer.

La synchronisation s'applique avec la commande `migrate` (on trouve encore dans des documentations âgées le nom de la commande utilisée auparavant : `syncdb`). L'opération pourrait être limitée à certaines applications et à une certaine profondeur de migration, mais ce n'est pas utile ici. Il est même possible de défaire des migrations pour revenir à un état antérieur.

```
D:\dj>py manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes
Running migrations:
 Applying contenttypes.0001_initial... OK
 Applying auth.0001_initial... OK
 Applying admin.0001_initial... OK
 Applying admin.0002_logentry_remove_auto_add... OK
 Applying admin.0003_logentry_add_action_flag_choices... OK
 Applying contenttypes.0002_remove_content_type_name... OK
 Applying auth.0002_alter_permission_name_max_length... OK
 Applying auth.0003_alter_user_email_max_length... OK
 Applying auth.0004_alter_user_username_opts... OK
 Applying auth.0005_alter_user_last_login_null... OK
 Applying auth.0006_require_contenttypes_0002... OK
 Applying auth.0007_alter_validators_add_error_messages... OK
 Applying auth.0008_alter_user_username_max_length... OK
 Applying auth.0009_alter_user_last_name_max_length... OK
```

© Editions ENI - All rights reserved

La commande `showmigrations` montre les mêmes lignes que précédemment, mais cette fois avec tous les indicateurs cochés : `[X]`.

Du côté de la base de données, sous pgAdmin, on observe l'arrivée de neuf tables : six tables pour `auth`, une table pour `admin`, une table pour `contenttypes` et une table pour les migrations.

Parmi ces tables, celle-ci présente un intérêt particulier : `django_migrations`.

- Visualisez son contenu : menu **View/Edit Data ▶ All Rows**, ou raccourci-clavier : [Shift][Alt][V].

Sans surprise, on y constate la présence de 14 enregistrements pour mémoriser le fait que ces migrations ont été appliquées et à quel moment.

L'occasion est donnée plus loin de fabriquer des migrations. Au cours du développement, il faut s'attendre à parfois devoir faire machine arrière sur une migration qui finalement ne donne pas satisfaction et vouloir repartir sur une nouvelle piste, ou bien tâtonner sur une partie des caractéristiques du changement donnant lieu à migration. Dans ces cas, plutôt que de se débattre dans le système des migrations pour défaire les montages, il peut être plus efficace de faire soi-même un nettoyage manuel de bas niveau. C'est pourquoi il a été mis en évidence les éléments qui entrent en jeu en plus des modifications structurelles de la base : fichier sous `migrations\`, et enregistrement dans une table.

2.4 Création d'un super-utilisateur

Le paquet `django.contrib.auth` permet de gérer des utilisateurs ainsi que des droits, appelés aussi permissions. En fonction des droits attribués, nominativement à un utilisateur ou à travers un groupe d'utilisateurs auquel il appartient, cet utilisateur a plus ou moins de visibilité et de possibilités d'actions sur le site. Pour faciliter l'exploitation, le modèle intégré `User` (Utilisateur) comporte un indicateur booléen qui, lorsqu'il est monté, donne implicitement à ce compte toutes les permissions sans qu'il soit nécessaire d'avoir la collection complète des enregistrements pour cela.

Créer un utilisateur avec un tel privilège n'est aucunement un impératif, mais son aspect si utile fait que c'est une opération courante et presque un réflexe à la création d'un projet.

Rien n'empêche de créer plus d'un super-utilisateur, mais dans la pratique un seul devrait suffire au besoin dès lors que ce privilège n'est pas donné au compte d'une personne physique, mais plutôt à un compte impersonnel dédié à un rôle d'administration.

Bien que le gestionnaire du modèle Utilisateur dispose d'une méthode `create_superuser()` pour simplifier la création d'un compte de cette catégorie, il est encore plus simple d'employer la commande administrative offerte par le paquet pour ce besoin.

Le nom d'utilisateur est libre, mais un choix représentatif de l'intention est meilleur pour la compréhension et il est bon de privilégier des termes comme `superuser` en référence à l'attribut du modèle nommé `is_superuser`, ou `root` en référence au nom conventionnel du compte des systèmes d'exploitation de type Unix qui dispose de toutes les permissions.

L'adresse de messagerie doit être syntaxiquement valide, mais peut tout à fait être fictive si on sait que ce compte n'a pas vocation à recevoir des messages. C'est d'ailleurs préférable puisqu'un compte super-utilisateur devrait être réservé à la maintenance ou au support de niveau élevé. Si des messages devaient entrer en jeu, il est probablement plus pertinent d'employer un compte d'exploitation ou de support de premier niveau, quitte à lui attribuer plus de permissions qu'un compte ordinaire.

▶ Créez le compte :

```
D:\dj>py manage.py createsuperuser
Username (leave blank to use 'patrick'): root
Email address: fake@fake.tld
Password:
Password (again):
Superuser created successfully.
```

© Editions ENI - All rights reserved

Le mot de passe doit évidemment être choisi avec soin. La valeur saisie est soumise à évaluation auprès des validateurs listés dans le paramètre de configuration `AUTH_PASSWORD_VALIDATORS`. Parmi ces validateurs, il y en a un qui vérifie que le nombre de caractères est supérieur à un minimum, et un autre qui regarde si le mot ne fait pas partie d'une liste de valeurs trop ordinaires (de l'ordre de 20000 mots).

C'est ainsi qu'un mot de passe tel que `root`, `admin`, `user` ou `super` est signalé (mais étonnamment pas `superuser`, à exclure quand même) :

```
Password:
Password (again):
This password is too short. It must contain at least 8
 characters.
This password is too common.
Bypass password validation and create user anyway? [y/N]:
Password:
Password (again):
```

3. Champs

La modélisation des objets d'une application est un fondement crucial du processus de développement, car c'est une phase structurante, c'est-à-dire qu'une grande part des travaux de codage est ensuite gouvernée par la façon dont les instances d'objets se présentent et sont liées les unes aux autres. La pertinence du modèle conceptuel de données a une incidence sur de nombreuses qualités espérées du code et de l'application : fluidité, lisibilité, maintenance, rapidité, montée en charge, etc. Des méthodes et des outils existent pour assister la démarche de conception, comme Merise ou UML (*Unified Modeling Language*).

Quelle que soit la manière, il faudra s'interroger sur les questions typiques : Quelles sont les natures d'objets à considérer ? Classes concrètes ou abstraites ? Quelle hiérarchie d'héritage ? Quelles relations entre les instances et leurs orientations : faut-il dire « A possède B » ou « B appartient à A » et où matérialiser une cardinalité dans le cas de multipropriété?

Pour rester simple, l'application se contente d'une seule nature d'objets : des messages. De même, le nombre de propriétés d'un message est volontairement réduit.

Les objets, formalisés par une classe, se définissent dans le fichier ayant pour nom conventionnel `models.py`.

▶Complétez le fichier, pour contenir les extraits successifs :

`mysite\messenger\models.py`

```
from django.conf import settings
from django.db import models
from django.utils.timezone import now


class Message(models.Model):
    subject = models.CharField("sujet", max_length=120)
    body = models.TextField("corps", blank=True)
    # ...
```

L'essentiel des pièces pour constituer un modèle est fourni par le module `django.db.models` : la classe de base pour établir le socle et les types de champs couramment employés. En réalité, les types de champs sont hébergés sous le module `django.db.models.fields`, mais cette possibilité d'écriture centralisée est voulue par un fil continu d'importations et c'est la convention officiellement recommandée.

De façon traditionnelle, les messages ont un sujet et un corps.

Pour le sujet, un champ de nature chaîne de caractères de longueur bornée correspond au besoin. L'argument de la longueur maximale admise est obligatoire. Il est garanti d'être respecté au niveau de la base de données. Par contre, dans sa correspondance avec un type de colonne d'une table (typiquement un `VARCHAR()`), chaque moteur de base de données a sa politique de plafonnement. Par exemple, MySQL admet un maximum de 65535 caractères (maximum théorique, car d'une part c'est aussi le plafond de l'enregistrement en entier, toutes colonnes confondues, et d'autre part certains caractères occupent plus ou moins d'octets selon l'encodage adopté), alors que pour PostgreSQL le plafond n'est pas précisément déterminé, mais tellement élevé (de l'ordre de 1 Go) qu'il n'a plus de réel sens.

Donner la valeur 120 à la longueur maximale est ici un choix arbitraire, il suffit de rester raisonnable en prenant une valeur confortable sans tomber dans l'excès.

© Editions ENI - All rights reserved

Il n'est spécifié que quelques options, car pour les nombreuses autres options, leur valeur par défaut convient. C'est ainsi, par exemple, que le sujet est un champ requis, c'est-à-dire qu'il ne peut pas rester vide lorsqu'utilisé dans un formulaire.

On remarquera que le premier argument de ces deux champs correspond à l'option `verbose_name`, mais fourni ici en format d'argument positionnel alors que la documentation ne mentionne que des arguments nommés.

Il s'agit d'une habitude ancienne et conventionnelle d'écriture, sachant qu'il se trouve que cet argument est le premier des arguments. La remarque est valable aussi pour d'autres types de champs similaires, mais attention, ce n'est pas le cas pour tous : pour les champs de relation, tels que `ForeignKey`, le premier argument est positionnel et désigne la cible de la relation.

L'option permet de fournir un nom humainement compréhensible. Elle est facultative et sa valeur de repli est le nom de l'attribut (avec remplacement des caractères de soulignement par des caractères d'espace). Il n'y a pas d'obligation, mais la convention est de ne pas mettre la première lettre en majuscule puisque ceci est fait automatiquement lorsque la chaîne est utilisée en affichage.

En pratique, plutôt qu'une valeur fixe, il est courant d'enrober cette spécification par une fonction de marquage pour permettre une traduction différée dans la langue de l'utilisateur. Conventionnellement, on emploie cette forme d'écriture « `_("subject")` » (voir aussi à ce propos le chapitre Internationalisation).

Pour le corps, contrairement au sujet, le contenu peut prendre un volume important et il n'y a pas lieu fonctionnellement de fixer un plafond de longueur, considérant qu'il s'agit d'une saisie de texte venant d'une personne. C'est pourquoi un champ de nature texte illimité est choisi, d'autant plus que son composant de formulaire par défaut est rendu par une balise HTML `<textarea>`, donc en plusieurs lignes. Il existe cependant la possibilité de spécifier une longueur maximale, mais elle ne sera pas mise en application au niveau de la base de données puisque le type de colonne correspondant est du genre `TEXT` (PostgreSQL) ou `LONGTEXT` (MySQL). Une longueur maximale, en limitant la quantité en saisie, peut tout de même trouver un rôle en termes de sécurité en empêchant un déni de service par une attaque visant une occupation complète de l'espace disque. La valeur devrait être suffisamment haute pour ne pas pénaliser un usage légitime tout en restant contrainte afin de rester dans les capacités d'absorption du serveur.

Laisser le corps vide devant rester une situation valide, l'option `blank` est positionnée à `True` de façon à admettre que le champ demeure vide dans un formulaire.

▶Complétez la définition du message avec un expéditeur et un destinataire :

```
    # ...
    sender = models.ForeignKey(settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='sent_messages',
        verbose_name="expéditeur")
    recipient = models.ForeignKey(settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
        related_name='received_messages',
        verbose_name="destinataire")
    # ...
```

Puisqu'il s'agit d'une messagerie interne au site, les personnes impliquées sont des utilisateurs enregistrés. Les champs pointent donc vers des entrées du modèle de l'application `auth` pour établir une relation de type plusieurs-à-un. Les deux premiers paramètres mentionnés sont obligatoires.

© Editions ENI - All rights reserved

Le premier paramètre d'un champ `ForeignKey`, dont le nom est `to`, doit désigner la cible de la relation. Il est techniquement possible d'écrire un référencement apparemment plus direct, de cette façon :

```
from django.contrib.auth.models import User
    # ...
    sender = models.ForeignKey(User,
    # ...
```

Mais ceci est déconseillé, car on s'impose le modèle intégré fourni par l'infrastructure logicielle, or celle-ci offre la possibilité à un projet de personnaliser le modèle d'un utilisateur à travers le paramètre de configuration `AUTH_USER_MODEL`. Si ce raccourci est acceptable pour une application dédiée à un projet pour lequel on a la certitude de rester sur le modèle intégré, il ne l'est pas pour une application qui se veut réutilisable.

Le deuxième paramètre spécifie le comportement lorsque la cible est amenée à disparaître. Parmi les choix, on trouve : ne rien faire de particulier (avec une exposition au risque que le gestionnaire de base de données lève une erreur pour perte d'intégrité), empêcher la suppression de la cible, remplacer la valeur (par `null`, par la valeur par défaut, par une valeur de repli fixe ou calculée). Ici, le choix retenu est de propager l'effacement, considérant que si l'utilisateur disparaît, les messages dans lesquels il est impliqué n'ont plus lieu d'être conservés.

Normalement, le retrait du message doit sembler préjudiciable à l'autre partenaire, qui lui existe encore. C'est pourquoi, dans la plupart des cas, on préfère empêcher toute suppression d'objet central sensible, comme un utilisateur, et employer un marquage sur l'objet pour signifier qu'il ne peut plus être impliqué dans de nouvelles relations. D'ailleurs, le modèle intégré offre un attribut `is_active` qui convient bien à cette signification puisque justement la documentation recommande de positionner ce drapeau plutôt que d'effacer le compte.

Le paramètre `related_name` est en principe optionnel, car il a une valeur par défaut. Celle-ci est constituée par le motif `<model_name>_set` où `<model_name>` est le nom en minuscules du modèle à la source de la relation. Dans le cas actuel, cela aboutit à la valeur `message_set`. Ce nom correspond à un gestionnaire de relation inverse qui permet, à partir d'une instance d'objet du côté destination de la relation, d'accéder aux instances d'objet du côté source de la relation.

Par exemple :

```
from django.db import models


class Auteur(models.Model):
    # ...

class Livre(models.Model):
    auteur = models.ForeignKey(Auteur, on_delete=models.CASCADE)
    # ...
```

```
>>> a = Auteur.objects.get(id=1)
>>> a.livre_set.all()  # tous les livres de cet auteur
```

Il s'agit de la structure traditionnelle d'un objet livre, avec sa relation vers un objet auteur (unique par simplification). À partir d'une instance d'auteur, la requête non moins traditionnelle est d'obtenir la liste de ses livres.

D'une part, ce nom par défaut, issu d'une composition automatique, peut paraître pauvre en sens (encore plus si on emploie des noms de classe en français, comme ici avec ce mélange de langues : `livre_set`) et on peut lui préférer un nom plus significatif.

D'autre part, il n'est pas possible de se reposer sur cette valeur par défaut plus d'une fois puisqu'il y aura une ambiguïté.

© Editions ENI - All rights reserved

C'est précisément le cas avec deux relations de `Message` vers `User` (ou une déclinaison personnalisée). Sans le positionnement explicite des noms, l'ambiguïté est reportée ainsi au lancement du serveur :

```
messenger.Message.recipient: (fields.E304) Reverse accessor for
'Message.recipient' clashes with reverse accessor for
'Message.sender'.
   HINT: Add or change a related_name argument to the
   definition for 'Message.recipient' or 'Message.sender'.
messenger.Message.sender: (fields.E304) Reverse accessor for
'Message.sender' clashes with reverse accessor for
'Message.recipient'.
   HINT: Add or change a related_name argument to the
   definition for 'Message.sender' or 'Message.recipient'.
```

Puisqu'il faut nommer explicitement au moins une des deux relations, autant le faire pour les deux avec des noms expressifs. À partir d'un utilisateur, il sera aisé d'accéder à ses messages, soit en tant qu'expéditeur, soit en tant que destinataire :

```
>>> from django.contrib.auth import get_user_model
>>>
>>> u = get_user_model().objects.get(id=1)
>>> sent_msgs = u.sent_messages.all()  # en expéditeur
>>> received_msgs = u.received_messages.all()  # en destinataire
```

Pour tenir compte, par principe, de la possibilité que le modèle d'utilisateur puisse être personnalisé au sein d'un projet, il est fait appel à `get_user_model()` plutôt que de référencer directement `User`.

Bien que l'ordre n'ait pas d'importance, placer l'option `verbose_name` en fin de la définition est une habitude d'écriture répandue dans la communauté.

▶Pour finir, ajoutez un champ de nature temporelle :

```
    # ...
    sent_at = models.DateTimeField("envoyé le", default=now)
    # ...
```

Par simplicité, l'instant d'envoi est assimilé à celui de la création de l'instance. Dans ces conditions, plutôt que de gérer soi-même l'alimentation du champ, il est plus efficace de l'automatiser ici en précisant une valeur par défaut.

Cette valeur n'étant pas fixe, elle doit être fournie à l'aide d'un objet appelable, en l'occurrence la fonction `django.utils.timezone.now()`.

À la place d'une valeur par défaut, on pourrait être tenté d'utiliser l'option `auto_now_add=True` qui positionne automatiquement le champ à la valeur de l'instant présent lorsque l'objet est créé.

Le résultat est identique, mais la différence vient après la création : il n'est plus possible de modifier la valeur du champ. Ce comportement est spécialement dédié aux champs dont le rôle est un horodatage de la création : ne pas pouvoir modifier la valeur par la suite apporte une garantie d'intégrité. L'option similaire `auto_now` joue le même rôle, mais à chaque mise à jour de l'enregistrement. Elle est donc dédiée aux champs d'horodatage de dernière modification. La situation n'est pas véritablement celle d'un tampon d'horodatage de création, c'est pourquoi l'option `default` est préférable, car elle laisse la possibilité de créer et de modifier un objet avec un instant d'envoi de son choix.

4. Métadonnées

Une métadonnée est une donnée qui sert à qualifier une autre donnée. Dans le cas d'un modèle, il va s'agir de spécifier des informations complémentaires, telles que des caractéristiques, des préférences, des options, des libellés, etc. N'étant pas à mélanger avec les champs, ces définitions doivent être placées dans une classe interne portant le nom `Meta`.

▶Complétez le fichier, pour contenir les extraits successifs :

`mysite\messenger\models.py`

```
# ...
class Message(models.Model):
    # ... les champs

    class Meta:
        verbose_name = "message"
        verbose_name_plural = "messages"
        ordering = ['-sent_at', '-id']

    # ...
```

© Editions ENI - All rights reserved

Aucune métadonnée ni la classe `Meta` ne sont obligatoires, car des valeurs par défaut sont prévues. Beaucoup d'options correspondent à des usages avancés, comme modifier des noms ou des conventions, qu'il n'est même pas nécessaire de connaître lorsqu'on reste dans une situation ordinaire. Quelques options ont toutefois une valeur par défaut, fixe ou issue d'une déduction, qui, bien que fonctionnant dans toutes les situations, n'est pas nécessairement la mieux adaptée à l'application. C'est la situation avec les trois cas mentionnés ici.

De la même manière que l'option équivalente vue précédemment pour les champs, les options `verbose_name` et `verbose_name_plural` permettent de fournir un nom humainement compréhensible dans la représentation textuelle, respectivement au singulier et au pluriel, de la classe du modèle ou de ses instances. Par défaut, le nom au singulier est déduit du nom de la classe par une transformation algorithmique (« CamelCase » devient « camel case ») et le nom au pluriel est composé du nom au singulier avec l'ajout d'un caractère « s » final. Il se trouve que ces déductions pourraient convenir pour la classe `Message`, mais il faut plutôt le voir comme une coïncidence. Il est courant qu'elles ne soient pas suffisantes : des noms de classe peuvent mal se prêter à ce genre de conversion ; la règle de mise au pluriel est différente pour certains mots. De toute façon, pour une variation internationale, on est amené à écrire sous cette forme :

```
        verbose_name = _("message")
        verbose_name_plural = _("messages")
```

Si on ne précise rien, les requêtes d'interrogation de la base de données rendent les objets dans un ordre indéterminé. De plus, cet ordre peut être variable d'une exécution à l'autre de la même requête. Il reste néanmoins préférable de ne pas demander un classement des résultats lorsque cela n'a pas d'utilité, car cette option à l'opération représente un coût supplémentaire pour la base de données. Pour autant, il ne faudrait pas tomber dans l'excès inverse en faisant soi-même en Python le classement sur la collection des résultats.

Dans le cas présent, lorsqu'on récupère une liste de messages, il y a tout lieu de penser qu'il faudra les présenter dans un certain ordre. Dans la plupart des cas, l'ordre attendu est celui du temps, basé sur l'instant d'envoi en l'occurrence, d'où la citation du champ `sent_at`.

Le caractère « - » additionnel est un préfixe qui précise un sens descendant du classement, car il est en général préférable de présenter les objets les plus récents en tête de liste. L'absence de préfixe induit un sens ascendant.

Remarque

La chaîne '?' demande un classement volontairement aléatoire, à ne pas confondre avec le fait de ne rien spécifier car, dans ce cas, un classement est généralement induit par le gestionnaire de base de données et il est relativement stable (d'autant plus si un cache intervient), bien qu'indéterminé et non garanti.

Avec seulement un classement selon l'instant d'envoi, le caractère indéterminé du classement est susceptible d'intervenir à nouveau au sein d'ensembles de messages s'ils sont émis au même instant. Cette réflexion semble théorique au regard de la précision de l'instant qui peut aller jusqu'à la microseconde, si on considère que des personnes sont à l'initiative des actions. En pratique, on peut trouver des raisons pour devoir quand même se protéger contre cette mauvaise situation.

Une première raison provient d'un principe de base qui veut qu'on doive s'attendre à tous les mauvais coups, même les plus impensables. Il n'est jusqu'à présent pas établi qui alimente ce champ ni de quelle façon. Supposons que la fonction `django.utils.timezone.now()` soit utilisée. Celle-ci s'appuie sur la méthode Python `datetime.datetime.now()` d'une classe pourvue d'une précision de la microseconde. Pourtant, cette précision peut ne pas être respectée en fonction de l'environnement d'exécution : par exemple, avec Python 2.7 en plateforme Windows, la précision obtenue ne peut pas être meilleure que la milliseconde (les chiffres de microsecondes sont là, mais à trois zéros). La milliseconde reste d'une finesse acceptable, mais cet exemple montre que rien ne garantit contre le fait de tomber sur un environnement d'une précision outrageusement plus grossière.

Une deuxième raison de ne pas se fier à l'unicité de l'instant d'envoi vient de la possibilité volontairement laissée, malgré la fourniture d'une valeur par défaut, d'imposer dès la création ou de modifier après création la valeur du champ. Dans le cas d'une saisie d'instant par une personne il est vain d'espérer une précision meilleure que la seconde.

© Editions ENI - All rights reserved

Dans le cas où ce n'est pas une personne mais un processus informatisé qui est à l'origine de l'émission, par exemple une API avec des déclenchements par lots, la rapidité d'action fait que des instants identiques peuvent apparaître, voire peut-être naturellement si l'émetteur ne se préoccupe pas d'une grande précision des valeurs d'instant qu'il impose.

Quoi qu'il en soit, il n'y a pas en principe de raison forte pour déclarer invalide la simultanéité d'envoi de plusieurs messages. C'est pourquoi il faut donner un critère de tri secondaire. L'identifiant de l'enregistrement est un candidat adéquat, considérant qu'il s'agit d'un nombre croissant, unique et immuable (on exclut la mauvaise conception d'une édition par effacement de l'ancien enregistrement et création d'un nouvel enregistrement).

Comme pour le critère primaire, le préfixe « - » est employé, toujours pour avoir les objets les plus récents en début de liste.

5. ORM (Object Relation Mapping) et migrations

Un des points forts de Django est de placer les modèles au cœur de la définition d'une application, c'est-à-dire d'y concentrer le maximum d'informations, pour ensuite en déduire des composants dérivés, par exemple les éléments d'un formulaire HTML, leur validité (telle que : valeur requise ou optionnelle, longueur minimale/maximale, jeu de caractères), etc.

Il s'agit d'un des concepts fondamentaux posés à la création du produit par ses développeurs. Cela signifie qu'un modèle doit porter non seulement les données d'une instance, mais aussi les comportements de l'objet qu'il représente : les moyens d'y accéder en base de données et sa logique métier.

C'est ainsi que, ayant défini précédemment le modèle, cela doit suffire à l'infrastructure logicielle pour se débrouiller dans la manipulation de ce modèle et de ses instances d'objet. En conséquence, on peut engager une étape de création en base de données des structures relatives à ce modèle.

Dans le jargon de l'infrastructure logicielle, cette opération qui consiste à transposer les nouveautés des modèles vers les schémas de la base de données porte le nom de « migration ».

Ce terme semble un peu ésotérique puisque sa signification ordinaire exprime le déplacement physique d'éléments d'une zone vers une autre. Ici, il faut l'entendre dans un sens plus large, comme un changement d'état ou le parcours d'un état A vers un état B.

Deux commandes ont été précédemment explorées, relatives à ce sujet, et se limitant à de l'observation :

- `showmigrations`, pour afficher une liste des migrations répertoriées avec leur état d'exécution.
- `sqlmigrate`, pour afficher le code SQL prévu pour l'exécution de la migration.

La réalisation d'une migration se déroule en deux phases : une phase de génération du matériel relatif à une migration et une phase d'exécution de migration. Cette répartition ouvre ensuite des facilités de gestion et d'action : mémoriser les changements dans le système de contrôle de versions ; déployer seulement la phase exécutoire à un ou plusieurs serveurs; faire des retours arrière en annulant une migration.

5.1 Création d'une migration initiale

La commande administrative `makemigrations` est destinée à faire une analyse du différentiel de l'état des modèles par rapport au dernier état connu et à en codifier les changements. Si on ne restreint pas la portée de la commande à une ou plusieurs applications, toutes les applications déclarées en `INSTALLED_APPS` de la configuration sont concernées, ce qui est rarement souhaité, car on a plutôt tendance à travailler une application à la fois. Il est donc préférable d'être explicite sur la cible, d'autant plus si les autres applications sont à la charge d'autres intervenants au sein d'une équipe.

```
D:\dj>py manage.py makemigrations messenger
Migrations for 'messenger':
 mysite\messenger\migrations\0001_initial.py
   - Create model Message
```

© Editions ENI - All rights reserved

On constate la génération d'un fichier dans le répertoire des migrations de l'application. La partie après le caractère de soulignement dans le nom du fichier peut être imposée par une option à la ligne de commande, mais le plus simple est de laisser son nom être issu d'une composition automatique, pour ne pas avoir à s'en préoccuper. Le nom suggéré est, dans quelques cas simples, un assemblage basé sur la nature de l'opération et les noms des objets impliqués (modèle et champ), avec des cas particuliers, par exemple `initial` pour la toute première migration. Dans la plupart des cas, le motif par défaut avec horodatage `auto_<aaaammdd>_<hhmm>` s'applique, notamment pour des opérations multiples ou pour une opération de modification. Le numéro séquentiel en préfixe n'a pas de réelle importance sur le plan technique, mais il procure quand même des avantages : 1) l'unicité du nom des migrations au cas où elles concernent une même opération sur une même cible ; 2) un confort de lecture pour le développeur pour constater la séquence des changements dans le temps.

Le comparatif d'état tient aussi compte des migrations préparées et non encore exécutées. Cela se voit si on relance la commande une seconde fois immédiatement :

```
D:\dj>py manage.py makemigrations messenger
No changes detected in app 'messenger'
```

On peut souhaiter simplement faire une observation ponctuelle des actions de changement, par exemple pour lever des doutes ou faire des vérifications avant de poursuivre avec des modifications additionnelles. Dans ce cas, plutôt que de devoir à chaque fois effacer les fichiers de sortie, il est plus efficace d'employer l'option `--dry-run` qui permet d'éviter l'écriture de ces fichiers. Il est ainsi possible d'alterner un ajustement du code et une relance de la commande jusqu'à ce que la liste des actions corresponde aux attentes. Si on souhaite vraiment avoir en plus la visibilité du contenu des fichiers de migration, il faut ajouter l'option `--verbosity 3` (les niveaux 1 et 2 n'apportent rien de plus) et ces contenus sont alors produits sur la console :

```
D:\dj>py manage.py makemigrations --dry-run --verbosity 3 messenger
Migrations for 'messenger':
 mysite\messenger\migrations\0001_initial.py
   - Create model Message
Full migrations file '0001_initial.py':
# Generated by Django 2.1.5 on 20[...]
```

```
from django.conf import settings
[...]
```

Expliquer le contenu du fichier produit n'est pas un objectif ici puisque ce contenu est dédié au moteur d'exécution qui sait comment l'exploiter.

Il faut malgré tout retenir qu'il s'agit bien de code Python, donc lisible si on en a la curiosité. Une lecture permet en outre de constater que l'étendue de l'analyse des changements des modèles et des champs ne se limite pas aux seules informations destinées au système de bases de données, mais couvre toute modification. Par exemple, si on passe le libellé du champ `subject` de `sujet` à `sujet2`, la migration engendrée est :

```
py manage.py makemigrations --dry-run --verbosity 3 messenger
Migrations for 'messenger':
 mysite\messenger\migrations\0002_auto_20[...]_[...].py
   - Alter field subject on message
Full migrations file '0002_auto_20[...]_[...].py':
# Generated by Django 2.1.5 on 20[...]
[...]
migrations.AlterField(
 model_name='message',
 name='subject',
 field=models.CharField(max_length=120, verbose_name='sujet2'),
[...]
```

On peut expérimenter les commandes `showmigrations` et `sqlmigrate` sur la migration disponible, mais elles ne vont rien montrer de nouveau par rapport aux observations faites précédemment dans ce chapitre.

5.2 Exécution d'une migration initiale

La migration préparée à la section précédente s'applique à l'aide de la commande administrative `migrate`, qui a déjà été employée en début de chapitre :

```
D:\dj>py manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, messenger
Running migrations:
 Applying messenger.0001_initial... OK
```

© Editions ENI - All rights reserved

Au début du chapitre, à l'occasion de la toute première migration pour mise en place des éléments initiaux de la base de données, il a déjà été souligné l'existence de la table `django_migrations` et la nature de ses enregistrements. Sans surprise, on y constate donc l'arrivée d'un enregistrement supplémentaire.

Ce qui intéresse le plus est l'arrivée de la table `messenger_message`. Par défaut, si on ne cherche pas à exploiter l'option `db_table` de la classe interne `Meta` du modèle, le nom de la table est composé par un assemblage du nom de l'application et du nom de classe du modèle.

6. Exploration des métadonnées

Une API est à disposition pour explorer les caractéristiques des classes de modèles, ce qu'on peut aussi appeler introspection de modèles. Elle est accessible par l'attribut `_meta` de la classe du modèle. Cet attribut porte une instance d'objet de classe `django.db.models.options.Options`. Si on compte l'usage de cet attribut parmi les sources des applications sous `django.contrib`, on constate qu'il est mentionné par six applications sur quinze et surtout que la moitié des références sont dans le module d'administration intégré, sans surprise puisque justement son rôle est de construire des pages sur base d'un balayage des applications gérées et de leurs modèles.

La diversité des termes relatifs au concept « meta » semble inutilement déconcertante. Elle s'explique par des raisons historiques. L'attribut existe sous ce nom depuis l'origine du produit, mais comme l'indique son caractère de soulignement en préfixe, il était conçu en tant qu'interface privée. En Python, on sait qu'il s'agit d'une convention, car techniquement rien ne restreint l'accès. L'emploi de la qualification « non public » essaye d'apporter la signification qu'il s'agit moins d'une notion de permission qu'une mise en garde : l'implémentation derrière le nom doit être considérée comme instable, non officielle et pouvant changer à tout moment sans préavis ni engagement de continuité de compatibilité.

Très tôt, les développeurs ont trouvé cette interface bien pratique pour obtenir les informations nécessaires à leur code applicatif, malgré le risque encouru de cessation brutale de fonctionnement à l'occasion d'une montée de version.

De plus, en n'étant pas publique, l'implémentation n'était pas couverte par des tests unitaires.

Constatant cet état de fait, les concepteurs de Django ont admis qu'il fallait apporter un support officiel à cette interface et celui-ci a été introduit dans la version 1.8. À l'occasion de ce nouveau statut, l'API a été révisée, pour une meilleure formalisation. Étant donné la vaste diffusion de son usage, il a été porté une attention particulière au maintien d'un maximum de rétrocompatibilité de cette interface, notamment grâce à des solutions de remplacement. Par ailleurs, un bon nombre de méthodes et de propriétés sont passées par une phase transitoire de dépréciation pour laisser le temps aux développeurs d'adapter leur code aux nouvelles pratiques.

En réalité, l'API n'a pas entièrement quitté son caractère privé. Seulement une portion a acquis un niveau public et le nom `_meta` a été conservé. Si on se fie à la documentation, seules deux méthodes ont été promues au niveau public :

- `get_fields(include_parents=True, include_hidden=False)`, pour obtenir une collection des champs du modèle.
- `get_field(field_name)`, pour obtenir une instance de champ à partir de son nom.

Il faut donc considérer le reste comme étant toujours à caractère privé. Pourtant, l'examen des propriétés de l'attribut révèle une grande quantité d'informations. Certaines pourraient être intéressantes à exploiter pour une application ou pour de la mise au point. L'extrait ci-dessous mentionne des propriétés à la signification aisément abordable :

```
>>> from mysite.messenger.models import Message
>>> from pprint import pprint
>>>
>>> pprint(vars(Message._meta))
{[...]
 'abstract': False,
 'app_label': 'messenger',
[...]
 'auto_created': False,
 'auto_field': <django.db.models.fields.AutoField: id>,
[...]
 'db_table': 'messenger_message',
[...]
```

© Editions ENI - All rights reserved

```
 'managed': True,
 'model': <class 'mysite.messenger.models.Message'>,
 'model_name': 'message',
 'object_name': 'Message',
[...]
 'ordering': ['-sent_at', '-id'],
[...]
 'pk': <django.db.models.fields.AutoField: id>,
 'private_fields': [],
 'proxy': False,
[...]
 'swappable': None,
[...]
 'verbose_name': 'message',
 'verbose_name_plural': 'messages'}
```

Revenons maintenant sur l'exploration des champs.

La méthode `get_fields([...])` rend un tuple de champs :

```
>>> pprint(Message._meta.get_fields())
(<django.db.models.fields.AutoField: id>,
<django.db.models.fields.CharField: subject>,
<django.db.models.fields.TextField: body>,
<django.db.models.fields.related.ForeignKey: sender>,
<django.db.models.fields.related.ForeignKey: recipient>,
<django.db.models.fields.DateTimeField: sent_at>)
```

En pratique, l'usage est plutôt de balayer la collection avec un filtre de façon à n'obtenir que des champs de nature homogène. Par exemple :

```
>>> [f.name for f in Message._meta.get_fields()
... if not f.auto_created and not f.is_relation]
['subject', 'body', 'sent_at']
```

Plusieurs attributs permettent de savoir s'il s'agit d'une relation et, si c'est le cas, quelle est sa nature :

```
>>> for f in Message._meta.get_fields():
...     print('{0.name:10}{0.is_relation!s:6}'
...     '{0.many_to_many!s:6}{0.many_to_one!s:6}'
...     '{0.one_to_many!s:6}{0.one_to_one!s:6}'
...     '{0.related_model}'.format(f))
...
id        False None  None  None   None   None
subject   False None  None  None   None   None
```

```
body      False None  None  None   None   None
sender    True  False True  False  False  <cla[...]auth.models.User'>
recipient True  False True  False  False  <cla[...]auth.models.User'>
sent_at   False None  None  None   None   None
```

La méthode `get_field(field_name)` permet d'en savoir plus sur un champ, à condition de connaître son nom. En voici un exemple complet :

```
>>> pprint(vars(Message._meta.get_field('subject')))
{'_db_tablespace': None,
 '_error_messages': None,
 '_unique': False,
 '_validators': [],
 '_verbose_name': 'sujet',
 'attname': 'subject',
 'auto_created': False,
 'blank': False,
 'choices': [],
 'column': 'subject',
 'concrete': True,
 'creation_counter': 51,
 'db_column': None,
 'db_index': False,
 'default': <class 'django.db.models.fields.NOT_PROVIDED'>,
 'editable': True,
 'error_messages': {'blank': 'This field cannot be blank.',
                    'invalid_choice': 'Value %(value)r is not a
valid choice.',
                    'null': 'This field cannot be null.',
                    'unique': '%(model_name)s with this
%(field_label)s already exists.',
                    'unique_for_date': '%(field_label)s must be
unique for %(date_field_label)s %(lookup_type)s.'},
 'help_text': '',
 'is_relation': False,
 'max_length': 120,
 'model': <class 'mysite.messenger.models.Message'>,
 'name': 'subject',
 'null': False,
 'primary_key': False,
 'remote_field': None,
 'serialize': True,
 'unique_for_date': None,
 'unique_for_month': None,
```

© Editions ENI - All rights reserved

```
'unique_for_year': None,
'validators': [<django.core.validators.MaxLengthValidator object
 at 0x0000017BA7FD5898>],
'verbose_name': 'sujet'}
```

Parmi les propriétés du champ, on voit par exemple la présence de `help_text` et `max_length`. L'accès à la valeur de telles propriétés peut être mis à profit, par exemple pour composer un formulaire sur mesure à partir du modèle d'une application externe si on n'est pas satisfait de la génération automatique de formulaire ou de formulaires proposés par l'application. L'introspection est préférable à la duplication si on tient à minimiser les efforts de maintien à jour et les risques d'incohérence à l'occasion de montées de version.

Il est aussi possible de citer un nom de champ correspondant à l'attribut `related_name` d'un autre modèle qui pointe vers ce modèle.

L'application en présente deux cas :

```
>>> from django.contrib.auth.models import User
>>>
>>> User._meta.get_field('sent_messages')
<ManyToOneRel: messenger.message>
>>> User._meta.get_field('received_messages')
<ManyToOneRel: messenger.message>
```

Voici une observation des détails d'un tel champ :

```
>>> pprint(vars(User._meta.get_field('sent_messages')))
{'field': <django.db.models.fields.related.ForeignKey: sender>,
 'field_name': 'id',
 'limit_choices_to': {},
 'model': <class 'django.contrib.auth.models.User'>,
 'multiple': True,
 'name': 'sent_messages',
 'on_delete': <function CASCADE at 0x0000017BA73BA158>,
 'parent_link': False,
 'related_model': <class 'mysite.messenger.models.Message'>,
 'related_name': 'sent_messages',
 'related_query_name': None,
 'symmetrical': False}
```

Il est possible de suivre les liens plus profondément et ainsi d'explorer de proche en proche les ramifications. Dans cet exemple, on part du modèle `User` pour découvrir le champ `sender` du modèle `Message` :

```
>>> pprint(vars(User._meta.get_field('sent_messages').field))
{'_db_tablespace': None,
[...]
 'attname': 'sender_id',
[...]
 'name': 'sender',
[...]
 'verbose_name': 'expéditeur'}
```

Ceci peut parfois être utile dans quelques situations : en mise au point ; en maintenance ; pour prendre connaissance de la structure d'applications externes dont on ne maîtrise pas le code.

7. Gestionnaires

Un gestionnaire de modèles est un objet fondamental parmi les fonctionnalités offertes par une infrastructure logicielle. Il en a été fait un survol à l'occasion d'une section précédente au sujet de l'ORM et des migrations. Le gestionnaire se pose en intermédiaire facilitateur entre le code applicatif qui préfère manipuler confortablement des instances d'objet Python et la base de données qui comprend essentiellement des requêtes en langage SQL. Cette couche d'abstraction est également bien utile en prenant à sa charge les subtiles différences de syntaxe ou de fonctionnalités entre les systèmes de base de données supportés officiellement. Ainsi, les opérations usuelles d'interrogation, de création, de modification et de suppression d'enregistrement se font par de simples appels aux méthodes du gestionnaire.

D'office, avec la définition d'un modèle, un gestionnaire natif est mis à disposition sous un attribut de la classe. Cet attribut se nomme `objects`. Il se pourrait qu'on souhaite utiliser ce nom en tant que champ du modèle, bien que ce soit rare et mal avisé au regard du potentiel de confusion avec les habitudes. On pourrait juste vouloir donner un autre nom à l'attribut du gestionnaire, bien que là encore l'intérêt est probablement très faible, comparé à la rupture des conventions. Le cas d'usage ciblé est plutôt la possibilité de personnaliser le gestionnaire.

© Editions ENI - All rights reserved

Dans tous les cas, la façon de faire reste la même : définir un attribut avec une instance du type `models.Manager` ou d'un type hérité.

Dans l'application, il n'y a pas de nécessité ni de raison valable de changer le nom de l'attribut et il est donc conservé. Par contre, le terrain va être préparé pour un gestionnaire personnalisé :

```
# ...
class MessageManager(models.Manager):
    # à compléter
    pass


class Message(models.Model):
    # ...
    sent_at = models.DateTimeField("envoyé le", default=now)

    objects = MessageManager()

    class Meta:
    # ...
```

La personnalisation sera implémentée par la suite au sein de la classe de gestionnaire. Les deux grandes motivations pour faire l'effort de construire son gestionnaire sont : modifier le comportement par défaut en surchargeant des méthodes du gestionnaire natif et ajouter des fonctionnalités en définissant des méthodes additionnelles.

Le cas typique du premier cas est la situation où il faut naturellement restreindre la collection des objets. Plutôt que de devoir à chaque emploi du gestionnaire poser toujours le même filtre, il est plus efficace d'intégrer ce filtre dans le gestionnaire.

Prenons l'exemple fictif où on ne veut pas exploiter des messages d'une ancienneté supérieure à un an. Un filtre est à intégrer dans la méthode qui fournit l'objet `QuerySet` de base, en charge de rendre une collection des objets de la base de données.

```
from datetime import timedelta

# ...
class MessageManager(models.Manager):
    def get_queryset(self):
        max = now() - timedelta(days=365)
        return super().get_queryset().filter(sent_at__gte=max)
```

Avec cette surcharge, l'écriture habituelle `Message.objects.all()` ou tout autre chaînage d'autres méthodes rendra un ensemble intrinsèquement limité par un critère temporel. Ceci peut être perçu comme un bien puisqu'on écrit la contrainte une fois pour toutes et donc sans risque d'oubli au sein du projet. A contrario, d'une part l'écriture n'est pas porteuse d'indices pour le développeur de la présence d'une contrainte personnalisée et d'autre part on peut avoir besoin de formuler des requêtes non soumises à cette contrainte.

Heureusement, il est aussi possible de définir plus d'un gestionnaire pour un même modèle. On peut alors définir un autre gestionnaire, personnalisé, qui n'aura pas la contrainte.

En pratique, on adopte un schéma inverse, plus naturel : le caractère non restreint du gestionnaire natif est conservé et on définit des gestionnaires additionnels si on veut des contraintes, avec des noms significatifs.

Pour reprendre l'exemple précédent, une meilleure écriture est :

```
class Message(models.Model):
    # ...
    objects = models.Manager()
    recent_objects = MessageManager()
    # ...
```

Il est alors possible de faire appel soit à `Message.recent_objects.all()` pour les opérations ordinaires du site, soit à `Message.objects.all()` pour les opérations d'administration.

© Editions ENI - All rights reserved

On remarque que le gestionnaire natif, sous le nom conventionnel `objects`, est défini explicitement. En effet, ce gestionnaire n'est implicitement défini qu'en l'absence de tout autre gestionnaire. À cela s'ajoute une notion de gestionnaire par défaut en cas de présence de plusieurs gestionnaires, ciblé par des actions internes à l'infrastructure logicielle, par exemple la commande administrative `dumpdata` d'exportation d'objets de la base de données. Sauf à préciser quel est le nom de ce gestionnaire préférentiel par l'option `Meta.default_manager_name`, le premier rencontré dans la définition de la classe est retenu. C'est pourquoi il est essentiel non seulement de mentionner `objects`, mais aussi de le placer avant les autres gestionnaires.

Il existe un autre moyen de contourner les restrictions des gestionnaires personnalisés : par l'utilisation du gestionnaire de base du modèle. Optionnellement, comme vu précédemment, on peut définir une classe de gestionnaire, en attribuer une instance sous un attribut, et finalement mentionner le nom de cet attribut par l'option `Meta.base_manager_name`. En l'absence de précision, une instance native `models.Manager()` est créée. Si un gestionnaire personnalisé est imposé, il n'est pas censé opérer un filtrage des enregistrements (bien que techniquement rien n'empêche de le faire), car ce qu'on attend de lui est précisément d'avoir un accès totalement libre aux données, notamment pour parcourir les relations entre objets.

Dans la situation de la première écriture de notre exemple fictif, le filtrage temporel est évité en formulant la requête sous la forme `Message._base_manager.all()`. Cette possibilité peut en particulier être intéressante à exploiter face à des applications externes excessivement restrictives, et dans tous les cas pour de l'analyse de dysfonctionnements.

8. Opérations sur objets

À ce stade, les expérimentations des opérations sur les objets vont se faire de façon manuelle, en mode console. C'est assez rustique, mais avec le mérite de mieux appréhender les étapes pas à pas. Savoir ensuite intégrer ces instructions dans un véritable code applicatif en sera d'autant plus facile.

8.1 Création

Pour pouvoir créer des messages entre utilisateurs, il faut d'abord disposer d'utilisateurs. Il n'est pas question d'utiliser le compte super-utilisateur, ce n'est pas son propos.

Le gestionnaire du modèle Utilisateur dispose de la méthode `create_user([...])` dédiée à cette opération. Il suffit de lui passer en paramètres les valeurs essentielles et elle s'occupe du travail, notamment de la codification du mot de passe. Ce mot de passe n'est évidemment pas stocké en clair ni véritablement chiffré puisqu'il s'agit plutôt du résultat d'une fonction à sens unique basé sur un algorithme de hachage.

L'irréversibilité de la fonction assure qu'il n'est pas possible de remonter au mot de passe en clair même si on obtient la connaissance du produit. Si un utilisateur oublie son mot de passe, on ne peut donc pas le lui rappeler, mais seulement lui permettre d'en établir un nouveau.

Au minimum, la méthode requiert seulement le paramètre `username`. Il est donc possible de créer un compte sans mot de passe, mais alors le compte se voit attribuer malgré tout un mot de passe fictif, selon un motif qui rend reconnaissable par le système la nature à part de ce compte. En conséquence, ce compte n'est pas autorisé à se connecter au site. Il faut le voir comme un compte spécial, à vocation technique et non destiné à une personne physique. Par exemple, on peut imaginer des comptes fictifs d'attente, de façon à ne pas être empêché de créer des objets ayant une clé étrangère obligatoire vers un compte utilisateur, mais dont la cible définitive ne sera connue que plus tard dans le cycle de vie de l'objet. Un autre cas d'usage transitoire vient de la volonté de ne jamais transmettre un mot de passe de quelque manière que ce soit à un utilisateur, notamment par courrier électronique, par principe de sécurité, et donc de ne pas l'attribuer à la création. À la place, il est mis en place une procédure sécurisée permettant à l'utilisateur de positionner un mot de passe de son choix. Pour cette procédure, on peut simplement réutiliser celle qui peut exister pour gérer l'oubli de mot de passe, en considérant qu'avoir oublié ce qu'on n'a jamais su est un cas compatible.

© Editions ENI - All rights reserved

Le paramètre `password` étant en troisième position, après le paramètre optionnel `email`, on peut créer un compte, au plus simple, de cette façon :

```
>>> from django.contrib.auth.models import User
>>>
>>> foo = User.objects.create_user('foo', '', 'passoire')
>>> # ou  User.objects.create_user('foo', password='passoire')
```

Contrairement à la commande administrative `createsuperuser` vue précédemment, les validateurs de qualité du mot de passe ne sont pas mis en œuvre ici, considérant qu'il s'agit de code sous la maîtrise d'un développeur et non en lien direct avec la saisie d'une personne. La valeur fournie du paramètre est donc supposée valide dans cet appel. Si on souhaite soumettre une valeur à validation dans son code, une fonction est disponible, qui rend `None` comme résultat positif et lève une erreur dans les cas négatifs :

```
>>> from django.contrib.auth.password_validation \
... import validate_password
>>>
>>> validate_password('passoire')
>>> validate_password('motdepasse')
Traceback (most recent call last):
 File "<console>", line 1, in <module>
 File "[...]password_validation.py", line 51, in validate_password
   raise ValidationError(errors)
django.core.exceptions.ValidationError:
 ['This password is too common.']
```

Voici l'observation du compte nouvellement créé :

```
>>> pprint(vars(foo))
{[...],
 'date_joined': datetime.datetime([…]),
 'email': '',
 'first_name': '',
 'id': 2,
 'is_active': True,
 'is_staff': False,
 'is_superuser': False,
 'last_login': None,
 'last_name': '',
 'password':
 'pbkdf2_sha256$120000$fgBYsFAnyZ72$54XwwZDpjTk3[...]',
 'username': 'foo'}
```

L'attribut `id` est à 2 puisque le numéro 1 a été pris lors de la création du super-utilisateur effectuée en début de chapitre.

On constate que les attributs `first_name` et `last_name` sont vides, ce qui est techniquement valide, mais peu fonctionnel lorsqu'on a affaire à des individus. Il est donc naturel de compléter l'objet aussitôt :

```
>>> foo.first_name = 'John'; foo.last_name = 'Doe'
>>> foo.save()
```

Toutes les informations auraient tout de même pu être fournies dès la création grâce à la possibilité de véhiculer des attributs quelconques en tant que paramètres additionnels. Cette variante est appliquée à la création d'un deuxième compte :

```
>>> bar = User.objects.create_user('bar', password='passoire',\
... first_name='Alan', last_name='Smithee')
```

Maintenant que des comptes d'utilisateur sont disponibles, il devient possible de créer des messages. La création d'un nouvel exemplaire d'un modèle commence par l'instanciation d'un objet Python :

```
>>> msg = Message()
>>> pprint(vars(msg))
{[...],
'body': '',
'id': None,
'recipient_id': None,
'sender_id': None,
'sent_at': datetime.datetime(20[...], 800809),
'subject': ''}
```

Si on met en comparaison cet affichage et la définition de la classe du modèle, on peut faire les constats suivants :

- Les champs `body` et `subject` sont bien des chaînes vides.
- Un attribut de nom `id` est présent pour servir de clé primaire dans la table de la base de données. En effet, dans le modèle, il n'a pas été défini explicitement un champ comme devant servir de clé primaire. Ceci aurait été indiqué avec le paramètre `primary_key`. En conséquence, il a été implicitement donné au modèle un champ de nom `id` et de type `AutoField`, ce qui donnera en base de données un champ numérique entier, non nul et auto-incrémenté.

© Editions ENI - All rights reserved

- L'attribut `sent_at` a bien reçu l'instant présent en tant que valeur par défaut.
- Les attributs `recipient_id` et `sender_id` sont nommés ainsi, car le suffixe `_id` est ajouté aux champs de type `ForeignKey` pour former le nom de la colonne dans la table de base de données. Mais c'est un détail d'implémentation interne qu'on peut ignorer puisqu'on manipule toujours le champ avec son nom dans le modèle, sauf cas très particuliers où on composerait soi-même des ordres SQL personnalisés.

Pour l'instant, l'objet n'a d'existence qu'en mémoire. Si on veut lui donner aussi une existence en base de données, il faut demander sa sauvegarde par l'appel d'une méthode. Une tentative est lancée par curiosité :

```
>>> msg.save()
Traceback (most recent call last):
 File "[...]\django\db\backends\utils.py", line 85, in _execute
   return self.cursor.execute(sql, params)
psycopg2.IntegrityError: ERREUR:  une valeur NULL viole
 la contrainte NOT NULL de la colonne « recipient_id »
DETAIL:  La ligne en échec contient
 (1, , , 20[...].800809+00, null, null)
```

C'est un échec, mais il était prévisible. En effet, selon le modèle, les champs `sender` et `receiver` ont par défaut l'option `null=False`, ce qui signifie que les champs correspondants en base de données n'admettent pas la valeur `NULL` issue de la transposition de la valeur Python `None`. Il faut affecter de véritables valeurs à ces attributs, c'est une contrainte intentionnelle, sinon le modèle n'a pas de sens.

```
>>> msg.sender = foo; msg.recipient = bar
>>> pprint(vars(msg))
{[...],
'recipient_id': 3,
'sender_id': 2,
[...]}
```

On constate bien que le code manipule des champs ou des instances et l'infrastructure s'occupe de traduire cela en identifiants lorsque c'est nécessaire pour la technique, notamment dans les échanges avec la base de données.

Désormais, la sauvegarde peut aboutir ; la méthode ne fournit pas de valeur de retour :

```
>>> msg.save()
>>> pprint(vars(msg))
{[...],
 'id': 2,
 [...]}
```

L'attribut `id` a été valorisé à l'occasion de l'insertion de l'enregistrement. Il lui a été affecté la valeur 2, car la valeur 1 a été consommée par la tentative précédente, indépendamment du fait qu'elle ait échoué.

Si on trouve étrange de voir que le champ `subject` puisse rester vide alors que le champ du modèle a l'option par défaut `blank=False`, c'est oublier que l'option agit seulement sur la validation des champs de saisie d'un formulaire et non sur le contenu de la colonne en table.

Le positionnement des attributs peut se faire dès l'instanciation de l'objet. Il suffit d'employer le nom de l'attribut comme mot-clé de l'argument :

```
>>> msg3 = Message(sender=foo, recipient=bar, subject='sujet',
... body='texte')
>>> msg3.save()
```

Les étapes de création d'instance et de sauvegarde peuvent aussi se réaliser en une seule instruction par l'intermédiaire d'une méthode du gestionnaire :

```
>>> msg4 = Message.objects.create(sender=foo, recipient=bar,
... subject='sujet', body='texte')
>>> pprint(vars(msg4))
{[...],
 'id': 4,
 [...]}
```

© Editions ENI - All rights reserved

8.2 Mise à jour

La mise à jour d'un objet se demande également par la même méthode `save()` :

```
>>> msg4.body = 'texte révisé'
>>> msg4.save()
>>> pprint(vars(msg4))
{[...],
 'body': 'texte révisé',
 'id': 4,
 [...]}
```

Cette fois-ci, l'instruction SQL sous-jacente a été un `UPDATE` plutôt qu'un `INSERT`. Il est inutile de s'en préoccuper, un algorithme interne détermine le bon choix sur la base de la valeur de la clé primaire de l'instance.

Attention toutefois à bien prendre connaissance de cet algorithme si on est tenté d'intervenir sur l'affectation de la clé. Il s'agit d'un usage avancé à éviter, car il est très facile de tomber dans la confusion entre objet existant ou à créer. Le danger potentiel vient du fait que la présence d'une clé dans l'objet fait privilégier une mise à jour, avec un repli sur une insertion en cas d'échec, ce qui est voulu, mais malheureusement aussi en cas de retour incorrect (faux négatif) dans de rares circonstances. Si la clé est positionnée manuellement, le respect de l'intention n'est plus garanti.

À cela s'ajoute, pour PostgreSQL, une complication avec l'avancement des clés auto-incrémentées. Une illustration de ce propos est apportée par l'exemple suivant :

```
>>> msg5 = Message(sender=foo, recipient=bar, id=5)
>>> msg5.save()
```

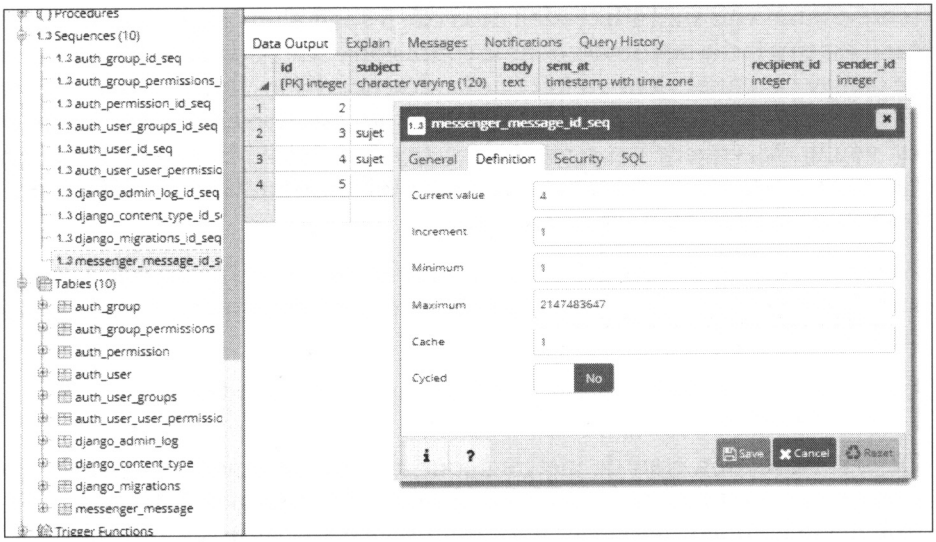
Déjà, si l'intention était de mettre à jour un enregistrement, soit qu'on croit exister, soit dont l'existence n'est pas déterminée, mais avec l'idée que l'opération devra poliment retourner une erreur si ce n'est pas le cas, c'est raté.

Si l'intention était de créer un nouvel enregistrement, c'est une opération réussie, dans un premier temps. Mais on s'expose à des ennuis, comme on le voit avec l'opération immédiatement à suivre :

```
>>> msg6 = Message(sender=foo, recipient=bar)
>>> msg6.save()
Traceback (most recent call last):
 File "[...]\django\db\backends\utils.py", line 85, in _execute
   return self.cursor.execute(sql, params)
psycopg2.IntegrityError: ERREUR:  la valeur d'une clé dupliquée
 rompt la contrainte unique « messenger_message_pkey »
DETAIL:  La clé « (id)=(5) » existe déjà.
```

L'opération qui semblait pourtant légitime échoue ici, car dans ce cycle, PostgreSQL n'a pas pu conserver la synchronisation de l'incrémentation.

Voici un instantané de l'état du compteur juste après l'insertion de l'enregistrement n°5 :



Le compteur n'a pas progressé : le contenu du champ **Current value** est toujours à 4. Donc, à l'insertion suivante, sans précision de numéro, un `INSERT` est lancé avec naturellement le prochain numéro de séquence, soit le 5, alors qu'il n'est pas libre. L'opération échouée a fait progresser la séquence à 5 et le prochain numéro sera le 6.

© Editions ENI - All rights reserved

Une nouvelle tentative du `save()` passera donc, mais le mal est déjà fait :

```
>>> msg6.save()
```

Il n'est pas nécessaire d'approfondir plus avant le sujet, mais seulement de préciser succinctement que des moyens existent pour obtenir une meilleure maîtrise des opérations :

- La commande administrative `sqlsequencereset` affiche l'ordre SQL à passer pour rétablir la synchronisation entre la séquence et les valeurs actuelles de la colonne (elle n'évalue pas si l'ordre est réellement nécessaire) :

```
D:\dj>py manage.py sqlsequencereset messenger
BEGIN;
SELECT setval(pg_get_serial_sequence('"messenger_message"','id'),
 coalesce(max("id"), 1),
 max("id") IS NOT null) FROM "messenger_message";
COMMIT;
```

On devine en voyant l'usage de la fonction `max()` que le but de l'instruction est de positionner le compteur à la valeur la plus élevée de la colonne `id`. Lorsqu'on sait cela, il est aussi possible de modifier manuellement sous pgAdmin le champ **Current value** de la séquence. Si cet ajustement avait été réalisé après l'insertion de l'enregistrement numéroté 5, la création suivante aurait réussi.

- Dans la méthode `save()`, il existe les paramètres optionnels booléens mutuellement exclusifs `force_insert` et `force_update` pour demander explicitement à être limité à un cas de figure précis et surtout sans repli vers l'autre, quitte à finir en erreur.
- L'option `select_on_save` de la classe `Meta` du modèle permet de revenir à un ancien algorithme (datant de Django 1.5 et avant) dans lequel un `UPDATE` potentiel devait être préalablement confirmé par un `SELECT` pour s'assurer de la réelle existence de l'enregistrement. Ceci évite un repli indésirable sur une insertion dans de rares contextes de faux retours d'opérations. Mais le coût systématique d'une pré-requête quasi inutile a fait abandonner ce principe.

Dans le cas d'une mise à jour, la méthode `save()` va reporter en table la totalité des champs puisque chacun d'eux a pu potentiellement changer. Pourtant, il est usuel que les changements concernent un petit nombre d'attributs. Sur un modèle avec un nombre considérable d'attributs, il y a matière à optimiser la requête SQL pour éviter le superflu. Le paramètre `update_fields` permet de donner la liste des seuls champs à considérer dans l'opération.

C'est difficilement le cas avec des formulaires, mais si on a connaissance sans effort des champs affectés, il y a matière à optimisation. L'exemple de révision du début pouvait s'écrire de manière plus performante ainsi :

```
>>> msg4.body = 'texte révisé'
>>> msg4.save(update_fields=['body'])
```

Un effet secondaire bénéfique de la présence du paramètre est l'assurance qu'un ordre `UPDATE` sera imposé, sans possibilité de repli sur une insertion.

8.3 Lecture

Le chargement d'instances d'objets à partir du stock dans la base de données se réalise par des demandes auprès du gestionnaire. Le principe général est de bâtir une requête à l'aide d'un objet `QuerySet`, plus ou moins complexe, en l'affinant pas à pas par des appels successifs aux méthodes qu'il supporte. Tant qu'on reste dans cette phase de pure construction (un motif de conception couramment rencontré sous le nom de `builder()` dans le monde Java), il n'y a pas d'évaluation, donc la base de données n'est pas sollicitée. Une grande part des méthodes rend l'objet lui-même, ce qui facilite l'écriture et la lecture du code en permettant d'enchaîner les méthodes les unes derrière les autres. Une autre part des méthodes va provoquer l'exécution de la requête et mettre un résultat à disposition. Il n'est pas obligatoire d'appeler une méthode terminale pour provoquer l'interrogation de la base de données, certaines opérations basiques du langage vont le faire, comme obtenir la quantité d'objets (`len()`), amorcer leur itération (`for..in`) ou les demander sous une certaine forme (`list()`).

La quantité de méthodes et de critères sur les champs est conséquente. Elle permet la construction de requêtes d'une complexité qui peut aller très loin lorsqu'on maîtrise le catalogue.

© Editions ENI - All rights reserved

Cela s'acquiert avec le temps, l'expérience, une bonne dose de persévérance, et l'emploi de quelques outils pour voir le rendu SQL et s'assurer que l'intention est bien respectée, mais aussi le plus souvent pour comprendre pourquoi cela ne fonctionne pas.

Cette section se limite à formuler des requêtes de base, avec les méthodes les plus couramment employées.

La méthode `get([...])` est destinée à la récupération d'un objet et un seul. Les arguments d'appel, combinés éventuellement avec d'autres critères de filtrage posés par des filtres précédents, doivent permettre d'identifier un enregistrement unique. Une erreur est levée en dehors de ce cas : `MultipleObjectsReturned` si plusieurs enregistrements correspondent aux critères, `DoesNotExist` si aucun enregistrement n'est trouvé. Naturellement le critère de sélection privilégié est l'identifiant de la clé primaire, mais tous les critères peuvent être utilisés.

```
>>> msg = Message.objects.get(pk=3)
>>> print(msg)
Message object (3)
```

Ici, le terme `pk` est introduit comme mot-clé de recherche dans les champs, à la place de `id`. Ce terme, aux initiales de *primary key*, est un alias de commodité pour désigner la clé primaire du modèle. Le plus souvent, il s'agit de l'attribut automatique `id` et les deux formes d'écritures sont alors équivalentes. Utiliser l'alias est quand même plus significatif de l'intention, particulièrement lorsque la clé sort du cas classique, et permet surtout de ne pas se préoccuper de son véritable nom.

Remarque

Django n'est pas prévu pour admettre les clés primaires composites, c'est-à-dire issues d'un assemblage de plusieurs colonnes (référence : ticket 373).

Si l'idée est de se contenter de capter un seul objet quand on sait que potentiellement plusieurs peuvent satisfaire aux critères, on peut procéder par index après un filtrage :

```
>>> msg = Message.objects.filter(sender=foo)[0]
>>> print(msg)
Message object (6)
```

Une différence par rapport à l'usage d'un `get()` est que l'erreur produite en cas d'absence d'objet est ici `IndexError`.

Pour ce cas d'usage, il est probablement plus simple d'exploiter des méthodes de commodité telles que `first()` ou `last()`, avec ces avantages : la valeur `None` est directement rendue en cas d'absence d'objet; si aucun classement n'est établi, ni par la requête ni par le modèle, alors un classement selon la clé primaire est automatiquement induit. Ceci est d'autant plus vrai que l'index, qui n'est qu'un cas simplifié de la syntaxe de segmentation (`[i:j:k]`), ne peut pas être négatif ici : `[-1]` pour espérer obtenir le dernier élément n'est pas supporté.

La méthode `get_or_create([...])` est aussi d'un usage fréquent. C'est également une méthode dite de commodité, car elle rassemble en un appel unique la capacité de récupérer un objet selon ses caractéristiques, s'il existe, et la capacité de créer un tel objet dans le cas contraire (optionnellement avec la fourniture de valeurs à d'autres attributs).

Par exemple :

```
>>> result = Message.objects.get_or_create(body='texte')
>>> print(result)
(<Message: Message object (3)>, False)
```

Le résultat rendu est un tuple `(object, created)`, avec `object` l'objet existant ou créé, et `created` un booléen indiquant s'il y a eu création.

On remarquera la présence d'une méthode similaire dans le contexte de modification plutôt que de lecture : `update_or_create([...])`.

8.4 Suppression

Si on détient l'instance Python, supprimer l'enregistrement correspondant de la base de données se fait simplement en appelant la méthode `delete([...])` sur l'objet :

```
>>> result = msg6.delete()
>>> print(result)
(1, {'messenger.Message': 1})
```

© Editions ENI - All rights reserved

Le résultat renvoyé est un tuple donnant le nombre d'objets supprimés et un dictionnaire donnant la ventilation de cette quantité par type d'objet. Cette finesse de rapport prend plus de sens avec une suppression en cascade d'objets liés ou en cas d'objets répartis sur plusieurs tables du fait d'une construction de modèle par héritage.

L'objet Python demeure existant, à ceci près que son attribut `id` a été réinitialisé :

```
>>> pprint(vars(msg6))
{[...],
 'body': '',  'id': None,
 'recipient_id': 3,
 'sender_id': 2,
 'sent_at': datetime.datetime(20[...], 772565),
 'subject': ''}
```

Supprimer une collection d'objets se réalise efficacement par l'intermédiaire du gestionnaire du modèle, puisqu'une action unique est demandée au système de base de données sans passer par un chargement en mémoire de ces objets (sauf situations moins ordinaires avec des `on_delete` sur des clés étrangères ou des signaux à émettre avant ou après suppression). En contrepartie de ce gain de performances, il ne pourra pas y avoir de passage par la méthode `delete()` pour chaque objet, ce qui pourrait potentiellement être un manque si cette méthode a été personnalisée pour effectuer une action indispensable. Dans ce genre de situation, il faut en revenir à la bonne vieille boucle itérative et supprimer les objets à l'unité.

Par exemple :

```
>>> result = Message.objects.filter(pk=5).delete()
>>> print(result)
(1, {'messenger.Message': 1})
```

Dans cet exemple, le fait que la collection ne contienne qu'un seul élément ne nuit pas à la généralité du propos. Par contre, il ne faudrait pas imaginer remplacer indifféremment la méthode `filter()` par la méthode `get()`, car cette dernière rend un objet et non un `QuerySet` dont on cible la méthode `delete()`.

Il se trouve que par coïncidence et homonymie des noms de méthode, dans ce cas précis on aboutira finalement au même résultat, mais de manières différentes : avec le `get()`, deux requêtes SQL seront produites : un `SELECT` puis un `DELETE`, avec de plus un chargement en mémoire d'un objet inutile puisqu'immédiatement oublié.

8.5 Optimisations

Les méthodes de l'objet `QuerySet` sont bien utiles et suffisamment évoluées pour s'épargner la peine d'écrire en langage SQL. Il reste néanmoins pertinent d'avoir au minimum une idée de la forme de l'ordre SQL qui sera élaboré par l'infrastructure et soumis à exécution auprès du système de base de données.

Cette connaissance, même superficielle, peut permettre de composer intelligemment sa requête afin d'éviter des interrogations inutilement lourdes en traitement ou volumineuses en quantité de données transmises. En voici quelques cas :

Utiliser count() plutôt que len()

Pour connaître la quantité d'une collection d'objets, une approche basique consiste à récupérer la collection pour en compter ses éléments :

```
>>> len(Message.objects.all())
3
```

Cette façon de faire est évidemment dispendieuse en transmission de données puisque les objets eux-mêmes ne servent à rien ici. Une meilleure conception revient à demander le comptage au niveau de la base de données pour n'avoir plus qu'à transmettre la charge utile de l'information :

```
>>> Message.objects.all().count()
3
>>> Message.objects.count()  # syntaxe suffisante
3
```

© Editions ENI - All rights reserved

Utiliser exists() plutôt que des opérateurs booléens

Pour connaître la présence ou l'absence d'enregistrements satisfaisant des critères, un contexte booléen est un déclencheur suffisant pour l'évaluation de la requête :

```
>>> if Message.objects.filter(sender=foo):
...     print('Au moins un msg envoyé par Foo.')
...
Au moins un msg envoyé par Foo.
```

À nouveau, cette formulation provoque un gâchis par la transmission d'un volume de données non exploitées, en l'occurrence un ensemble de messages. Une manière plus efficace est de demander l'évaluation booléenne en amont :

```
>>> superuser = User.objects.get(pk=1)
>>> Message.objects.filter(sender=foo).exists() and 'ok'
'ok'
>>> Message.objects.filter(sender=superuser).exists() and 'ok'
False
```

Utiliser update() plutôt que get() + save()

Si le but d'un traitement est de mettre à jour un enregistrement alors qu'on ne détient pas au préalable son instance Python représentative, il n'est pas pour autant indispensable de créer une telle instance.

Cette façon d'écrire donne lieu à deux ordres SQL :

```
>>> msg = Message.objects.get(pk=4)
>>> msg.subject = 'sujet4'
>>> msg.save()
```

Alors que l'action peut être accomplie en un seul ordre SQL :

```
>>> Message.objects.filter(pk=4).update(subject='sujet4')
1
```

La méthode donne en retour le nombre d'enregistrements retenus pour la mise à jour, à ne pas confondre avec le nombre d'enregistrements réellement modifiés qui n'est pas connu et peut lui être inférieur.

Un autre avantage d'un ordre unique est d'éviter l'improbable mais potentielle faille de fonctionnement si le champ est modifié par une action parallèle dans l'intervalle de temps entre le `get()` et le `save()`, aussi court soit-il.

On comprend encore mieux la sensibilité du problème quand il s'agit de réaliser une incrémentation, pour un simple compteur ou pour attribuer un numéro de série unique.

8.6 Opérations de masse

Dans cette section sont rassemblées quelques opérations, particulières, car leur action cible une quantité d'objets, le but étant d'agir sur un lot de façon plus performante que par une boucle parcourant chacun des objets individuels.

Les actions citées portent sur la création et la lecture d'objets, ainsi que l'entrée et la sortie de jeux de données.

Des actions de modification et d'effacement, respectivement par les méthodes `update(**kwargs)` et `delete()`, existent aussi et sont suffisamment explicites pour ne pas être détaillées ici. Il est toutefois souligné que même lorsqu'on traite un seul objet, passer par ces méthodes doit normalement faire bénéficier d'une meilleure performance par rapport à agir sur l'objet lui-même, car l'action se réalise si possible directement au stade de l'ordre SQL. Si le gain est affirmé pour une modification, il n'est pas systématique pour un effacement, car selon la présence ou non d'écouteurs sur des signaux de pré-effacement ou de post-effacement et le besoin de propager l'effacement à des objets liés, il y aura ou pas chargement en mémoire de l'objet (détail interne d'implémentation appartenant à Django).

Plutôt que d'écrire :

```
# fonctionne, mais peu efficace
m = Message.objects.get(id=12)
m.subject = 'foo'
m.save()
```

Il est préférable de faire :

```
# mieux
Message.objects.filter(id=12).update(subject='foo')
```

© Editions ENI - All rights reserved

De même, remplacer :

```
# fonctionne, mais peu efficace
m = Message.objects.get(id=12)
m.delete()
```

Par :

```
# mieux
Message.objects.filter(id=12).delete()
```

Un avantage supplémentaire à ce que l'action soit réalisée au plus près de la base de données est d'éviter la période critique entre le chargement de l'objet et la mémorisation de sa nouvelle situation. Aussi courte soit cette période, elle laisse la possibilité d'une potentielle altération de l'objet en base de données par un éventuel autre processus tout autant légitime, d'où un risque de conflit.

8.6.1 Création

La méthode `bulk_create(objs, batch_size=None)` permet de réaliser l'insertion d'une collection d'objets en une seule requête à la base de données, d'où normalement un gain en performances comparé à une boucle ordinaire sur une requête unitaire. Toutefois, si la quantité est vraiment énorme et pose des soucis sur les ressources (mémoire, capacité de la base de données ou de son pilote, besoin de voir un taux de progression), un paramètre optionnel permet de fixer un plafond de façon à introduire une répartition par lots de taille raisonnable.

Exemple d'envoi d'une annonce à tous les utilisateurs du site :

```
>>> msgs = [Message(subject='Annonce', sender=superuser,
... body='Demain, le site sera fermé pour maintenance.',
... recipient=user) for user in User.objects.all()]
>>> Message.objects.bulk_create(msgs)
[<Message: Message object (7)>, <Message: Message object (8)>,
<Message: Message object (9)>]
```

Le retour est la liste des mêmes objets, éventuellement enrichis avec la clé primaire lorsque la base de données permet la récupération de cette information (c'est le cas ici avec PostgreSQL).

8.6.2 Lecture

La méthode `in_bulk(id_list=None, field_name='pk')` permet la récupération en bloc d'un ensemble d'objets à partir d'une liste des valeurs pour un champ. En général, le champ le plus approprié est la clé primaire du modèle, ce qui correspond à la valeur par défaut du paramètre pouvant préciser le champ de sélection.

Exemple

```
>>> result = Message.objects.in_bulk([7, 8, 9])
>>> pprint(result)
{7: <Message: Message object (7)>,
 8: <Message: Message object (8)>,
 9: <Message: Message object (9)>}
```

Le résultat rendu n'est pas une liste comme on a l'habitude pour des opérations traitant de collections, mais un dictionnaire, avec le bénéfice de formaliser une correspondance directe entre une valeur du champ et son objet.

Le champ de sélection doit être de type unique, sinon une erreur se produit :

```
>>> Message.objects.in_bulk(['Annonce'], field_name='subject')
Traceback (most recent call last):
 File "<console>", line 1, in <module>
 File "[...]ango\db\models\manager.py", line 82, in manager_method
   return getattr(self.get_queryset(), name)(*args, **kwargs)
 File "[...]\django\db\models\query.py", line 621, in in_bulk
   raise ValueError("in_bulk()'s field_name must be a unique
     field but %r isn't." % field_name)
ValueError: in_bulk()'s field_name must be a unique field but
 'subject' isn't.
```

8.6.3 Chargement d'instantané

Le terme « instantané » est celui de la documentation en français pour traduire le mot anglais d'origine : « *fixture* ». En réalité, l'un comme l'autre est nébuleux, mais il faut admettre qu'il n'est pas évident de donner un nom au mécanisme mis en œuvre et on attachera peu d'importance au mot.

© Editions ENI - All rights reserved

Un instantané est une collection de fichiers porteurs de contenus sérialisés de la base de données. Il se désigne par un nom et se manipule par des commandes administratives, essentiellement `loaddata` et `dumpdata` pour respectivement importer et exporter des données, mais aussi `testserver` pour alimenter une base de données à des fins de tests.

Pour avoir du matériel à disposition, le mieux est de commencer par produire un instantané. La façon la plus évidente est de demander une telle production avec la commande administrative `dumpdata`. Sans paramètre, toutes les instances de tous les modèles des applications sont sorties, ce qui est probablement excessif, et les informations sont fournies sur la sortie standard de la console, au format JSON.

Pour illustration, en voici une copie (raccourcie pour limiter le volume) :

```
D:\dj>py manage.py dumpdata
s[{"model": "contenttypes.contenttype", "pk": 1, "fields":
{"app_label": "admin", "model": "logentry"}}, {"model":
"contenttypes.contenttype", "pk": 2, "fields": {"app_label":
"auth", "model": "permission"}}, [...] {"model":
"contenttypes.contenttype", "pk": 6, "fields": {"app_label":
"messenger", "model": "message"}}, {"model": "auth.permission",
"pk": 1, "fields": {"name": "Can add log entry", "content_type":
1, "codename": "add_logentry"}}, [...] {"model": "auth.user", "pk":
1, "fields": {"password": "pbkdf2_sha256$[...]", "last_login":
null, "is_superuser": true, "username": "root", "first_name": "",
"last_name": "", "email": "fake@fake.tld", "is_staff": true,
"is_active": true, "date_joined": "20[...].177", "groups": [],
"user_permissions": []}}, [...] {"model": "messenger.message",
"pk": 2, "fields": {"subject": "", "body": "", "sender": 2,
"recipient": 3, "sent_at": "20[...].800"}}, [...]]
```

Pour la suite des manipulations, il vaut mieux se limiter seulement aux objets de l'application et demander une indentation pour rendre la lecture plus confortable. Pour récupérer les informations dans un fichier, on peut soit préciser son nom par une option, soit utiliser la redirection de flux du système d'exploitation :

```
D:\dj>py manage.py dumpdata --indent 2 messenger > dumpMsgs.json
D:\dj>type dumpMsgs.json
[
{
 "model": "messenger.message",
```

```
 "pk": 2,
 "fields": {
   "subject": "",
   "body": "",
   "sender": 2,
   "recipient": 3,
   "sent_at": "20[...].800"
 }
},
[...]
{
 "model": "messenger.message",
 "pk": 9,
 "fields": {
   "subject": "Annonce",
   "body": "Demain, le site sera ferm\u00e9 pour maintenance.",
   "sender": 1,
   "recipient": 3,
   "sent_at": "20[...].237"
 }
}
]
```

Sans rien changer au contenu du fichier pour commencer, une réinjection immédiate est expérimentée avec la commande administrative `loaddata`. Le niveau de traces est positionné afin d'avoir plus de détails sur le déroulement des traitements :

```
D:\dj>py manage.py loaddata --verbosity 2 dumpMsgs
Loading 'dumpMsgs' fixtures...
Checking 'D:\dj' for fixtures...
Installing json fixture 'dumpMsgs' from 'D:\dj'.
Resetting sequences
Installed 6 object(s) from 1 fixture(s)
```

Il n'est pas utile de détailler toutes les possibilités de l'algorithme de recherche, mais à partir du nom de l'instantané, la commande est capable d'explorer plusieurs répertoires conventionnels potentiels, pour tenter d'y trouver des fichiers compatibles, avec un contenu au format de sérialisation JSON ou XML, éventuellement compressés. La détermination se base sur des motifs dans les extensions cumulées du nom de fichier (par exemple `.xml.zip`), non dans le contenu du fichier.

© Editions ENI - All rights reserved

Les traces de la dernière commande aident à comprendre ce qui s'est passé : le répertoire courant a été exploré, un fichier `dumpMsgs.json` y a été trouvé. Il en est déduit que l'interprétation doit se faire au format JSON et le contenu du fichier a été injecté dans la base de données.

La trace indiquant la réinitialisation des séquences correspond au sujet déjà abordé précédemment avec l'emploi de la commande administrative `sqlsequencereset`.

En effet, le contenu peut éventuellement fournir des valeurs à des champs de type auto-incrémenté, typiquement la clé primaire `id`. Il faut donc s'assurer de la synchronisation de ces compteurs en fin d'opération.

Enfin, la dernière ligne de trace est une synthèse de ce qui a été découvert et réalisé.

Chacun des objets extraits de l'instantané passe par un traitement de mémorisation en base de données qui est basé sur le même socle que la méthode `save()` qu'on a l'habitude d'employer, mais avec quelques différences du fait que certains attributs peuvent être absents ou sous une autre forme, notamment pour les champs de relation. Il reste néanmoins encore valable que ce sont la présence et la valeur de la clé primaire qui feront aboutir à une création d'enregistrement ou à une mise à jour d'un enregistrement existant. Dans la trace de la commande, le terme *Installed* est volontairement neutre pour s'appliquer à toutes les situations, mais il s'agissait de mises à jour.

Pour mettre en évidence le comportement en création de nouveaux objets, un instantané `loadMsgs.json` est tout d'abord créé manuellement sur la base d'une copie du fichier précédent.

La reprise des seuls deux derniers messages (pk 8 et 9) suffit pour la démonstration. Ensuite, il est fait les modifications suivantes : 1. Mettre un sujet et un corps facilitant leur distinction par rapport aux originaux ; 2. Retirer l'attribut `pk` pour inciter à une création :

```
D:\dj>type loadMsgs.json
[
{
 "model": "messenger.message",
 "fields": {
   "subject": "Via loaddata - A",
```

```
    "body": "Mon beau corps - A.",
    "sender": 1,
    "recipient": 2,
    "sent_at": "20[...].237"
  }
},
{
 "model": "messenger.message",
 "fields": {
    "subject": "Via loaddata - B",
    "body": "Mon beau corps - B.",
    "sender": 1,
    "recipient": 3,
    "sent_at": "20[...].237"
  }
}
]
```

Voici le passage de la commande d'importation :

```
D:\dj>py manage.py loaddata --verbosity 2 loadMsgs
Loading 'loadMsgs' fixtures...
Checking 'D:\dj' for fixtures...
Installing json fixture 'loadMsgs' from 'D:\dj'.
Resetting sequences
Installed 2 object(s) from 1 fixture(s)
```

La seule différence dans cette trace par rapport à la précédente tient au nombre d'objets installés. Une observation de la table des messages permet de constater l'arrivée des deux enregistrements nouveaux, sous les identifiants 10 et 11.

La création aurait aussi pu se faire en conservant les attributs `pk` et en les valorisant avec ces valeurs prédictibles, mais il est bien plus simple de s'épargner cette peine et de laisser le système s'en occuper.

© Editions ENI - All rights reserved

8.6.4 Quelques usages des instantanés

Alimentation initiale en données

Les instantanés fournissent un moyen pratique pour alimenter la base de données avec des jeux de données initiales, telles que des données de configuration ou des catalogues par défaut.

Un peu d'historique aide à mieux comprendre la raison d'être des instantanés : avant d'être remplacée par la commande `migrate` de plus grande envergure, la commande `syncdb`, conçue pour la seule étape de la phase d'installation d'une application, avait la capacité d'installer automatiquement, par sa seule présence, un instantané du nom conventionnel `initial_data`.

Le comportement ressemblait à ceci :

```
> py manage.py syncdb
Loading 'initial_data' fixtures...
Installing json fixture 'initial_data' from '/myapp/fixtures/'.
Installed 2 object(s) from 1 fixture(s)
```

Cette fonctionnalité a été rendue obsolète en version 1.7 et a été retirée en version 1.9. Désormais, pour disposer d'un mécanisme équivalent, une façon de faire est de le mettre en place soi-même sous la forme d'une migration de données.

Un exercice de cette mise en place va aider à comprendre les étapes. Pour commencer, il faut créer un squelette de migration destiné à être édité manuellement, grâce à une option :

```
D:\dj>py manage.py makemigrations --empty messenger
Migrations for 'messenger':
 mysite\messenger\migrations\0002_auto_20[...].py
```

Le fichier offre ce contenu préparatoire :

```
D:\dj>type mysite\messenger\migrations\0002_auto_20[...].py
# Generated by Django 2.1.5 on 20[...]

from django.db import migrations


class Migration(migrations.Migration):
```

```
    dependencies = [
        ('messenger', '0001_initial'),
    ]

    operations = [
    ]
```

Comme on le voit avec `dependencies`, cette migration suivra la toute première migration, celle en charge de créer le modèle dans la base de données, c'est-à-dire la table correspondante. C'est très bien ainsi puisque d'une part il faut bien cette table pour espérer y placer des données et d'autre part faire l'opération dès le début pour que ces données aient la qualification d'initiales.

Le fichier doit être modifié pour y mettre l'opération souhaitée. Dans la panoplie des opérations, une quinzaine sont relatives aux migrations de schémas. Parmi les trois opérations restantes, celle qui a du sens pour une migration de données est `RunPython`, même si, comme son nom l'indique, elle est généraliste puisqu'il s'agit d'exécuter du code à fournir en arguments. Selon les préconisations de la documentation, le code est fourni sous la forme d'une fonction, définie juste avant la classe :

```
# ...
from django.core.management import call_command
# ...

def load_initial_data(apps, schema_editor):
    db_alias = schema_editor.connection.alias
    call_command('loaddata', 'initial_data',
        database=db_alias,
        app_label='messenger',
    )

class Migration(migrations.Migration):
    # ...
    operations = [
        migrations.RunPython(load_initial_data),
    ]
```

© Editions ENI - All rights reserved

Seul le nom du fichier de données est fourni, il n'est pas nécessaire de mentionner un chemin d'accès, car par convention trois emplacements sont explorés cumulativement pour y rechercher la présence d'un ou plusieurs fichiers en concordance avec le motif du nom :

- L'éventuel répertoire `fixtures` de toutes les applications installées. Avec une option, il est toutefois possible de cibler une seule application, ce qui a été fait ici avec le paramètre `app_label`.
- Les éventuels répertoires cités dans le paramètre de configuration `FIXTURE_DIRS` (vide par défaut).
- Le répertoire de travail.

Note 1 : Il est permis de préciser des sous-répertoires dans un nom d'instantané si on a vraiment besoin d'une répartition. Par exemple : `initial/initial_data`.

Note 2 : Plusieurs fichiers de données peuvent être chargés, à condition qu'ils soient trouvés dans des répertoires différents. Pour des raisons historiques mal connues, la présence de deux concordances ou plus pour un même répertoire (par exemple `initial_data.json.zip` et `initial_data.yaml`) n'est pas admise et génère une erreur.

Le fichier de données est maintenant créé dans le répertoire le plus approprié, c'est-à-dire `messenger\fixtures`. Un simple format JSON non compressé convient.

`mysite\messenger\fixtures\initial_data.json`

```
[
{
 "model": "messenger.message",
 "fields": {
   "subject": "Bienvenue",
   "body": " Texte d'accueil.",
   "sender": 1,
   "recipient": 2
 }
},
{
 "model": "messenger.message",
 "fields": {
```

```
    "subject": "Bienvenue",
    "body": "Texte d'accueil.",
    "sender": 1,
    "recipient": 3
  }
 }
 ]
```

Avec le besoin de présence de relations vers un autre modèle, il est entendu que le modèle `message` n'est pas un bon candidat pour des données initiales. Il s'agit plutôt ici de faire une démonstration de la technique.

Il ne reste plus qu'à faire exécuter la migration de données :

```
D:\dj>py manage.py migrate messenger
Operations to perform:
 Apply all migrations: messenger
Running migrations:
 Applying messenger.0002_auto_20[...]...
 Installed 2 object(s) from 1 fixture(s)
OK
```

Par pgAdmin on peut vérifier l'arrivée des deux objets, avec les identifiants 12 et 13.

Si la commande administrative `flush` est utilisée dans un projet où des données initiales sont mises en œuvre, il faudra sans doute un peu plus d'attention dans les manipulations. En effet, cette commande permet de vider les tables de la base de données, ce qui peut être bien pratique en développement pour repartir d'une situation propre, sauf que la table des migrations réalisées (`django_migrations`) est préservée. Des données initiales injectées par migration sont donc éliminées comme les autres et la migration ne sera pas rejouée.

Pour remédier à cela, une première solution consiste à repasser manuellement le chargement des données initiales à l'aide de commandes `loaddata`. Ce n'est pas compliqué à réaliser, mais il faut avoir la connaissance de l'existence de ces données et ne pas en oublier. Dans cette situation, il est avantageux de se créer un script pour enrober cette séquence de commandes. Une deuxième façon pour réinitialiser la base de données est plus radicale, mais a son efficacité : il suffit de détruire la base (`drop database`) et de la recréer (`create database`), puis de passer une commande `migrate`.

© Editions ENI - All rights reserved

Alimentation pour suite de tests

Lorsqu'il s'agit de réaliser les tests d'une application, on en vient très vite à avoir besoin de données dans la base, ne serait-ce qu'un échantillon minimal. Si la quantité de ces données est réduite, il est habituel de les créer par du code avec l'utilisation ordinaire de l'ORM. Ceci peut s'implémenter dans la méthode `setUp()` ou, mieux, dans la méthode de classe `setUpTestData()`. Quand les volumes de données à mettre en jeu deviennent importants, les instantanés fournissent une solution : il suffit de spécifier l'attribut de classe `fixtures` avec une liste d'instantanés à charger.

Exemple :

```
from django.test import TestCase

class MessengerTestCase(TestCase):
    fixtures = ['tests/users.json', 'tests/messages.xml']
    # ...
```

Par rapport à sa classe de base `TransactionTestCase`, la classe `TestCase` est optimisée pour ne réaliser le chargement qu'une fois, tout en assurant que les données restent disponibles pour tous les tests de la classe grâce à un mécanisme de nettoyage basé sur une annulation de transaction.

Alimentation d'un serveur de tests

Habituellement la commande administrative `runserver` est utilisée pour faire tourner un serveur de développement sur la base de données courante. La commande administrative `testserver` procure un fonctionnement similaire, mais en utilisant une base de données éphémère créée à cette occasion et pouvant être remplie avec les instantanés cités en paramètres.

Cas d'usage : Un dysfonctionnement se manifeste en site de production, qu'on ne parvient pas à reproduire en site de développement. Il est soupçonné que la raison du problème soit plus dans les données que dans le code. Le diagnostic est difficile à établir en raison d'un manque de données réelles, en nature ou en volume, ou encore à cause d'une malheureuse combinaison insoupçonnée. Ces données qui exposent le problème peuvent être extraites par une commande `dumpdata`. Il n'est d'ailleurs peut-être pas indispensable de capter l'intégralité de la base de données et dans ce cas on peut limiter l'amplitude de l'exportation à certaines applications, voire certains modèles.

L'instantané est ensuite chargé par le serveur de tests et toutes les manipulations nécessaires au diagnostic, même destructives, peuvent être effectuées sans crainte.

Cas d'usage : Vérifier ou déterminer par une expérimentation manuelle face à un serveur les données adéquates à utiliser pour la suite de tests unitaires.

Transfert entre bases

Un instantané peut servir de support pour transférer des données d'une base de données vers une autre. Il est ainsi possible d'enrichir une base de production avec des données qui auront été préparées d'avance dans une base auxiliaire. Le fait de pouvoir mettre en ligne un volume important d'objets en une courte opération massive est d'autant plus intéressant si la constitution de l'information prend beaucoup de temps alors qu'il est important de mettre à disposition une collection suffisamment complète.

Un autre exemple, dans l'autre sens, est de vouloir reverser des données de production vers une autre base de travail, pour disposer de données représentatives de l'activité, afin d'en tirer des enseignements, d'analyser un dysfonctionnement, etc.

Par le biais des entrée et sortie standards, le passage par un stockage sur disque n'est même pas nécessaire si les bases de données sont accessibles simultanément au sein de l'infrastructure. La commande `dumpdata` envoie déjà par nature son résultat sur la sortie standard. La commande `loaddata` est capable de lire l'entrée standard si un caractère tiret est employé comme nom de l'instantané. Dans ce cas, il est nécessaire de préciser le format de sérialisation, puisque rien ne permet de le déduire.

Par exemple, avec une configuration déclarant les bases de données `prod` et `preprod` :

```
D:\dj>py manage.py dumpdata --database=prod mon_app | py
manage.py loaddata --format=json --database=preprod -
```

Le transfert d'informations par instantané apporte un avantage encore plus appréciable lorsque les gestionnaires de bases de données sont différents. Grâce à la sérialisation, on n'a plus à se préoccuper de trouver une combinaison d'outils d'import/export compatibles pour les deux parties.

© Editions ENI - All rights reserved

Il ne faudrait pourtant pas considérer ce moyen comme un outil de synchronisation de bases. Si les ajouts et modifications d'objets sont pris en compte, ce n'est pas le cas des suppressions. L'idée de passer par un vidage préalable est probablement périlleuse (disparition transitoire de données, rupture potentielle d'intégrité), voire impossible (perte de relations).

Mise à jour de base de données

Dans certains cas favorables, l'instantané offre un moyen assez simple pour tenir à jour une partie des informations d'une base de données. Pour illustrer cette capacité, supposons que nous avons un site basé sur l'exploitation d'un catalogue d'objets (des points géographiques d'intérêt, un annuaire de personnes, une liste de livres, etc.). Les conditions favorables sont que ces objets sont en lecture seule en production et qu'ils ne sont jamais supprimés, mais éventuellement rendus inactifs par un champ de statut.

Supposons aussi qu'il n'est pas souhaité que les changements (ajouts, modifications) soient faits en temps réel sur la base de production. Peu importe pourquoi, mais on peut trouver de multiples raisons à cela : collecte parcellaire de caractéristiques, cohésion entre objets, besoin de validation et approbation, etc. Un moyen simple pour gérer ce mode de fonctionnement est de saisir les nouveautés dans une base de données de travail (une base de données de développement peut jouer ce rôle) pour ensuite les copier en lots vers la base de données de production à des moments de synchronisation choisis.

Le transfert d'informations se passe en deux phases, dont la seconde est réalisable sans effort : il suffit d'injecter un instantané dans la base de données ciblée par une commande `loaddata` ordinaire. Tout le travail réside dans la première phase, celle de génération de l'instantané.

Puisqu'il est question de ne transférer que les objets ayant muté depuis le point de synchronisation précédent, la commande ordinaire `dumpdata` n'est pas suffisante, car elle ne propose pas de critères de sélection portant sur les instances d'objets. Le modèle doit donc permettre de repérer les nouveautés apparues au-delà d'une certaine date. Une façon simple de gérer ce besoin est de disposer d'un attribut donnant la date de dernière modification de l'enregistrement.

Pour servir d'illustration, supposons une application nommée `some_app` avec le modèle simplifié suivant :

```
from django.db import models
from django.utils.timezone import now


class SomeItem(models.Model):
    some_attr = models.CharField("some attr", max_length=10)
    # ...
    last_update = models.DateField("last update", default=now)
```

L'attribut `last_update` est un champ à usage interne pour administrer le cycle de vie de l'objet, en l'occurrence pour mémoriser à quelle date il a été créé ou modifié.

Pour produire l'instantané, une commande administrative personnalisée est mise en place. Par convention, l'implémentation d'une telle commande doit être logée sous le répertoire `management/commands/` d'une application.

Le nom d'un fichier `.py` est le nom d'une commande, sauf dans le cas particulier où le nom commence par le caractère de soulignement `'_'`, ce qui autorise la présence de modules auxiliaires.

Dans sa forme basique minimale, une commande personnalisée a la structure suivante :

`mysite\some_app\management\commands\some_cmd.py`

```
from django.core.management.base import BaseCommand


class Command(BaseCommand):
    # def add_arguments(self, parser):
        # éventuelles définitions d'arguments ...

    def handle(self, *args, **options):
        # traitements à réaliser ...
        pass
```

© Editions ENI - All rights reserved

Son lancement se fait de la même façon qu'une commande non personnalisée :

```
D:\dj>py manage.py some_cmd

D:\dj>
```

Sur cette base, il va être dans un premier temps ajouté des messages d'aide à la compréhension du rôle de la commande et le support d'un argument positionnel requis pour fournir une date :

```
# ...
class Command(BaseCommand):
    help = 'Exporte les objets modifiés.'

    def add_arguments(self, parser):
        parser.add_argument('last_update_date',
            help='dernière modification >= à jj/mm/aaaa'
        )

    # ...
```

Pour constater l'effet de ces additions :

```
D:\dj>py manage.py some_cmd --help
usage: manage.py some_cmd [-h] [--version] [-v {0,1,2,3}]
                  [--settings SETTINGS] [--pythonpath PYTHONPATH]
                          [--traceback] [--no-color]
                          last_update_date
Exporte les objets modifiés.

positional arguments:
 last_update_date       dernière modification >= à jj/mm/aaaa

optional arguments:
 -h, --help             show this help message and exit
 [...]
```

```
D:\dj>py manage.py some_cmd
usage: manage.py some_cmd [-h] [--version] [-v {0,1,2,3}]
                  [--settings SETTINGS] [--pythonpath PYTHONPATH]
                          [--traceback] [--no-color]
                          last_update_date
manage.py some_cmd: error: the following arguments are required:
last_update_date
```

Il reste à compléter la méthode pour réaliser le traitement voulu :

```
import datetime

from django.core import serializers
# ...

from mysite.some_app.models import SomeItem

class Command(BaseCommand):
    # ...
    def handle(self, *args, **options):
        date = options['last_update_date'].split('/')
        date.reverse()
        date = datetime.date(*map(int, date))
        return serializers.serialize("json",
            SomeItem.objects.filter(last_update__gte=date))
```

Les manipulations sur la date sont simplement destinées à extraire d'une chaîne de caractères `'day/month/year'` les éléments pour alimenter en nombres un constructeur de la forme `date(year, month, day)`.

La dernière instruction rend au format JSON les instances ayant leur valeur de date de dernière modification supérieure ou égale à la date fournie.

Pour mettre en évidence sans effort le fonctionnement de la commande, en se contentant de ce dont on dispose déjà dans le projet, on peut faire l'exercice avec deux substitutions :

```
from mysite.some_app.models import SomeItem
```

par :

```
from django.contrib.auth.models import User
```

et :

```
SomeItem.objects.filter(last_update__gte=date)
```

par :

```
User.objects.filter(date_joined__gte=date)
```

© Editions ENI - All rights reserved

Dotée de ces contournements temporaires, la commande est capable de prouver qu'elle peut produire un résultat :

```
D:\dj>py manage.py some_cmd 20/10/2000
[{
   "model": "auth.user", "pk": 1, "fields": {
     "username": "root",
     "first_name": "", "last_name": "",
     [...]
 }, {
   "model": "auth.user", "pk": 3, "fields": {
     "username": "bar",
     "first_name": "Alan", "last_name": "Smithee",
     [...]
 }, {
   "model": "auth.user", "pk": 2, "fields": {
     "username": "foo",
     "first_name": "John", "last_name": "Foo",
     [...]
 }
]
```

Sauvegardes et migrations

Rien ne semble empêcher de vouloir considérer le couple `dumpdata/loaddata` comme un moyen de sauvegarde/restauration, mais en pratique les gestionnaires de bases de données offrent normalement des outils dédiés à ces opérations, plus complets et performants.

Il faut aussi tenir compte de certaines particularités, comme l'absence de modèle et donc de données produites pour la table `django_migrations`. D'une manière générale, quel que soit l'outil, une sauvegarde a tendance à devenir obsolète avec l'arrivée de migrations de schémas.

Éventuellement, dans le cas particulier d'une migration d'une nature de gestionnaire vers une autre, l'instantané peut apporter une solution de compatibilité par le fait d'employer un format de sérialisation neutre, et, comme on l'a vu précédemment, les informations de séquence pour les colonnes auto-incrémentées sont correctement ajustées.

9. Migration de structures et données

Jusqu'à présent, les migrations de structures pratiquées étaient des opérations dites initiales, c'est-à-dire pour créer les tables, notamment celle de l'application. Il est temps maintenant d'expérimenter une évolution ordinaire au cours du développement d'une application, à savoir enrichir ou modifier un modèle. Ceci a donc naturellement des répercussions sur les tables de la base de données, mais l'outil de migration est encore là pour aider en se chargeant de la basse besogne.

Par commencer, le modèle reçoit un complément de champs bien utiles à la gestion de l'état d'un message :

- Un champ `read_at` pour mémoriser le fait que le message a été lu. On pourrait aussi se contenter d'un champ booléen, mais cela ne coûte pas grand-chose d'avoir cette information plus précise, même si elle n'est pas exploitée par la suite.
- Des champs `sender_deleted_at` et `recipient_deleted_at` pour mémoriser la manifestation de chacune des parties à vouloir effacer le message, puisqu'il ne correspond qu'à un unique enregistrement. À nouveau, on pourrait être plus rustique avec des champs de nature booléenne, mais disposer de dates ouvre une possibilité : le message pourra être définitivement nettoyé de la table, non seulement à condition que les deux parties aient marqué cette volonté, mais en plus à condition qu'il se soit écoulé un délai minimal de conservation, dans l'idée de maintenir un historique afin de récupérer un message sur un changement d'avis (genre corbeille).

▶ Ajoutez les champs suivants au modèle :

`mysite\messenger\models.py`

```
# ...
class Message(models.Model):
    # ...
    read_at = models.DateTimeField(
        "lu le", null=True, blank=True)
    sender_deleted_at = models.DateTimeField(
        "effacé par expéditeur le", null=True, blank=True)
    recipient_deleted_at = models.DateTimeField(
        "effacé par destinataire le", null=True, blank=True)
    # ...
```

© Editions ENI - All rights reserved

À la naissance d'un message, ces champs doivent être sans valeur. La façon ordinaire pour exprimer une absence de valeur est admettre une codification par un NULL au niveau SQL. Les champs de caractères sont des cas particuliers où par convention Django recommande de ne pas employer ce code, car il aurait la même signification que la chaîne vide, ce qui est une redondance à éviter. Ici, avec de l'horodatage, le NULL est admissible en positionnant l'option `null` à `True`. L'option `blank` est positionnée à `True` de façon à permettre que le champ demeure vide dans un formulaire.

▶Créez la migration :

```
D:\dj>py manage.py makemigrations messenger
Migrations for 'messenger':
 mysite\messenger\migrations\0003_auto_20[...]_[...].py
   - Add field read_at to message
   - Add field recipient_deleted_at to message
   - Add field sender_deleted_at to message
```

Dans le cas présent, la commande génère le fichier de migration sans poser de question puisque les enregistrements existants pourront prendre la valeur NULL dans les colonnes ajoutées. Ce n'est pas ainsi lorsqu'une valeur est requise et qu'elle ne peut pas être déterminée par défaut. À titre d'illustration, supposons que nous ayons eu un champ du genre :

```
    fake_at = models.DateTimeField("fake", blank=True)
```

La création de migration aurait imposé de fournir une valeur :

```
D:\dj>py manage.py makemigrations messenger
You are trying to add a non-nullable field 'fake_at' to message
without a default; we can't do that (the database needs something
to populate existing rows).
Please select a fix:
1) Provide a one-off default now (will be set on all existing
rows with a null value for this column)
2) Quit, and let me add a default in models.py
Select an option: 2
```

La solution n'est pas nécessairement de préciser une valeur par défaut dans le modèle. L'option 1 du correctif est tout à fait valable dans certaines situations, où une valeur par défaut n'a pas de sens pour de futurs objets et pourtant il faut bien affecter une valeur aux anciens objets. On peut qualifier cette valeur de « non-choix » ou de « compatibilité ».

Pour le cas d'une date-heure comme dans l'exemple précédent, on pourrait par convention adopter l'instant de départ des tampons d'horodatage, à savoir le 1er janvier 1970 :

```
D:\dj>py manage.py makemigrations messenger
[...]
Select an option: 1
Please enter the default value now, as valid Python
The datetime and django.utils.timezone modules are available,
so you can do e.g. timezone.now
Type 'exit' to exit this prompt
>>> datetime.datetime(1970,1,1)
Migrations for 'messenger':
 mysite\messenger\migrations\0004_message_fake_at.py
   - Add field fake_at to message
```

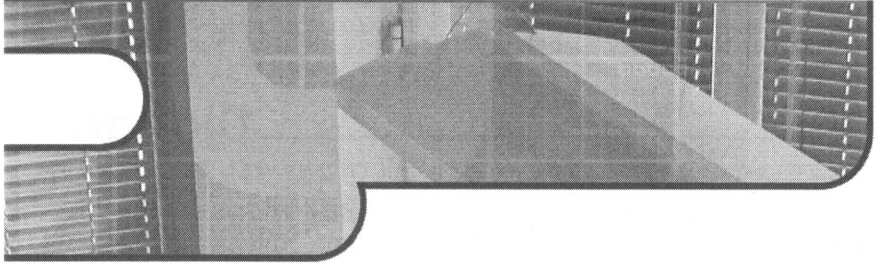
Remarque

La démonstration de ce type de fabrication de migration, avec demande de renseignement, ne peut pas être réalisée avec l'option `--dry-run`, car la fourniture imposée d'une valeur y est court-circuitée et la valeur est considérée comme `None`. Cette valeur est invalide au final, mais admise par simplification puisqu'il ne s'agit pas de produire un fichier de migration réel, mais seulement de montrer les migrations potentielles.

- En final, appliquez la migration relative aux champs ajoutés :

```
D:\dj>py manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, messenger
Running migrations:
 Applying messenger.0003_auto_20[...]_[...]... OK
```

© Editions ENI - All rights reserved



Chapitre 5
Traces et journalisation

1. Requêtes à la base de données

D'une façon générale, les outils d'ORM peuvent paraître parfois obscurs sur leur façon de produire les ordres SQL, ou bien, même si on pense avoir compris la documentation et écrit un code correct, on voudrait se rassurer sur son interprétation en ayant une visibilité sur l'ordre SQL construit. Ce chapitre va donner quelques moyens pour parvenir à voir cette construction et pour mieux comprendre comment son exécution est planifiée. Accéder à cette connaissance peut parfois aider à diagnostiquer des dysfonctionnements ou des faiblesses de performances.

1.1 Plan d'exécution

Le plan d'exécution d'un ordre SQL est un rapport fourni par le gestionnaire de base de données pour détailler de quelle façon celui-ci compte exécuter cet ordre. La plupart du temps, on ne se préoccupe pas d'orienter ses choix et on se fie à ses algorithmes pour faire au mieux, même s'il faut reconnaître que c'est plutôt par absence de maîtrise des options qu'il faudrait poser. L'observation d'un plan doit permettre de comprendre par exemple : si un index est mis à contribution et lequel lorsque plusieurs peuvent satisfaire au besoin ; quelles sont éventuellement les jointures mises en œuvre ; quels sont les volumes estimés et les efforts de traitement (plus difficiles à comprendre).

Il faut admettre que c'est par nature un rapport technique, souvent abscons et qui tourne au jargon pour un lecteur occasionnel.

Hors Django

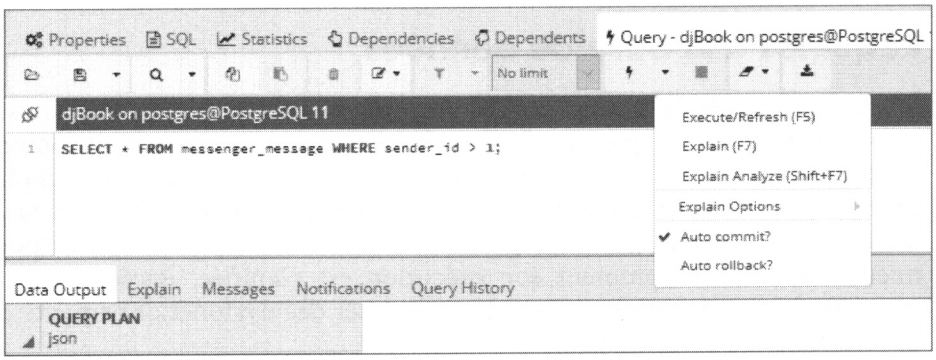
Pour aborder le sujet, commençons par faire des manipulations avec les outils disponibles directement sous le gestionnaire PostgreSQL et son pgAdmin.

Avec pgAdmin, sur la table `messenger_message`, il faut ouvrir un onglet d'interrogation (menu **Tools/Query Tool** ou clic droit et **Query Tool...**), car l'onglet habituel **Edit Data** ne donne pas accès à l'outil visé.

▶Copiez cet ordre SQL :

```
SELECT * FROM messenger_message WHERE sender_id > 1;
```

▶Demandez un **Explain (F7)** dans le sous-menu des exécutions :



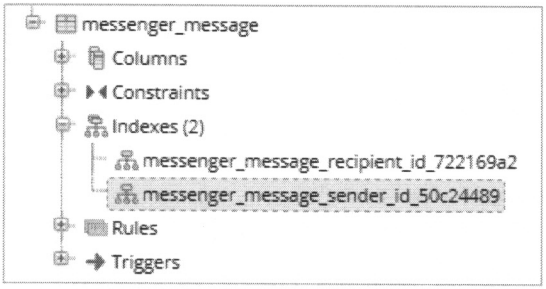
On peut se passer des **Explain Options > Verbose, Costs, Buffers, Timing**, car elles ne vont pas être déterminantes dans cette modeste expérimentation. Il en est de même pour la variante **Explain Analyze (Shift+F7)**.

© Editions ENI - All rights reserved

L'affichage bascule automatiquement sur l'onglet **Explain** pour présenter un affichage graphique des résultats. On va lui préférer l'onglet **Data Output** pour y observer le **`QUERY PLAN`**. Celui-ci est exprimé au format JSON, sur plusieurs lignes, mais seule la première ligne est visible. Pour visualiser l'ensemble, il est nécessaire soit de double-cliquer sur la cellule pour obtenir une fenêtre en surimpression, soit de faire un copier-coller vers un autre éditeur. Le résultat obtenu est le suivant :

```
[
 {
   "Plan": {
     "Node Type": "Seq Scan",
     "Parallel Aware": false,
     "Relation Name": "messenger_message",
     "Alias": "messenger_message",
     "Filter": "(sender_id > 1)"
   }
 }
]
```

L'information qui attire l'attention est le motif **`"Seq Scan"`**, pour en déduire que l'optimiseur du moteur SQL a choisi de procéder à un balayage séquentiel de la table pour en retenir les enregistrements satisfaisant au **`"Filter"`**. Ce champ dispose pourtant bien d'un index, selon cet écran :



Les deux index visibles sont par défaut automatiquement créés par Django par le fait qu'il s'agit de champs `ForeignKey`, candidats naturels à être impliqués dans des jointures de tables. L'optimiseur ne met pas en œuvre l'index dédié au champ d'expéditeur, car il a jugé que la balance entre le gain de performances espéré et le coût de traitement n'est pas favorable, étant donné le peu d'enregistrements. Un simple balayage est finalement plus économique.

Pour poursuivre la démonstration, plutôt que de remplir les tables avec des données fictives pour faire du volume, il suffit de poliment imposer l'utilisation de l'index.

Il n'est pas possible d'orienter PostgreSQL à utiliser un certain index par une mention dans l'ordre SQL, contrairement à MySQL qui supporte un indice sous la forme `USE INDEX (index_list)`. En contournement, un paramètre de configuration de la session est positionné pour décourager le planificateur à utiliser le balayage séquentiel s'il dispose d'une autre méthode.

La commande doit être passée dans le même onglet d'interrogation, sinon elle s'appliquera à une autre session. Par des mises en commentaires, on peut facilement mettre en jeu l'une ou l'autre des commandes :

```
SET enable_seqscan = off;
--SELECT * FROM messenger_message WHERE sender_id > 1;
```

- Passez un **Execute/Refresh (F5)** sur la commande SET ; alternez les commentaires pour revenir sur le `SELECT` et passez un **Explain (F7)**

Le **`QUERY PLAN`** devient :

```
[
 {
   "Plan": {
     "Node Type": "Bitmap Heap Scan",
     "Parallel Aware": false,
     "Relation Name": "messenger_message",
     "Alias": "messenger_message",
     "Recheck Cond": "(sender_id > 1)",
     "Plans": [
       {
         "Node Type": "Bitmap Index Scan",
         "Parent Relationship": "Outer",
         "Parallel Aware": false,
         "Index Name": "messenger_message_sender_id_50c24489",
         "Index Cond": "(sender_id > 1)"
       }
     ]
   }
 }
]
```

© Editions ENI - All rights reserved

Sans nécessairement tout comprendre, on constate notamment avec le motif **"Bitmap Index Scan"** qu'un index a bien été exploité et dans **"Index Name"** qu'il s'agit de celui attendu.

Pour revenir à la normale, la commande à passer est :

```
SET enable_seqscan = on;
```

Avec Django

L'objectif est maintenant de faire une manipulation comparable avec Django. La méthode `explain` (récente, car apparue en version 2.1) de l'objet `QuerySet` offre ce moyen.

```
>>> from mysite.messenger.models import Message
>>>
>>> print(Message.objects.filter(sender_id__gt=1).explain())
Sort  (cost=15.01..15.19 rows=73 width=334)
 Sort Key: sent_at DESC, id DESC
 ->  Seq Scan on messenger_message  (cost=0.00..12.75 rows=73
                                                    width=334)
       Filter: (sender_id > 1)
```

Le résultat est similaire à celui sous pgAdmin, avec les nuances suivantes : a) une étape de classement intervient puisque le modèle définit des critères de tri ; b) par défaut, le format de présentation est celui du système de gestion de bases de données, en l'occurrence du texte pour PostgreSQL ; c) des informations de coûts sont présentes, car par défaut l'option est `ON`. Un contenu identique, à part quelques effets de mise en forme, pouvait être retrouvé avec ces compléments :

```
>>> print(Message.objects
...      .order_by()
...      .filter(sender_id__gt=1)
...      .explain(format='JSON', costs=False))
[{'Plan': {'Node Type': 'Seq Scan', 'Parallel Aware': False,
'Relation Name':  'messenger_message', 'Alias':
'messenger_message', 'Filter': '(sender_id > 1)'}}]
```

Tout comme précédemment et pour les mêmes raisons, l'index disponible n'est pas utilisé. Son usage doit être à nouveau forcé, mais passer ce genre d'ordre SQL n'est pas une opération courante et donc aucune facilité n'est prévue par l'ORM pour cela. Deux façons de faire vont être explorées, plus ou moins propres et aisées.

Option 1 : Par requête brute

Le gestionnaire de modèle offre la méthode `raw()` pour construire des requêtes SQL dites brutes, c'est-à-dire prêtes à l'emploi, sous forme de chaîne de caractères.

```
>>> rqs = Message.objects.raw('SET enable_seqscan = off')
```

L'objet rendu est une instance de classe `RawQuerySet`, dont le fonctionnement se veut le plus proche de celui de la classe `QuerySet`. À ce titre, il faut provoquer une évaluation de la requête pour obtenir réellement son action au niveau de la base de données. Lorsqu'on essaye les façons documentées de l'évaluation pour `QuerySet`, elles finissent toutes en erreur, car la méthode est prévue pour un ordre qui va rendre des instances du modèle, ce qui n'est pas respecté ici. Ainsi les manières suivantes : `list(rqs)`, `len(rqs)`, `bool(rqs)`, `for x in rqs`, vont aboutir à ce type d'erreur :

```
TypeError: 'NoneType' object is not iterable
```

La documentation pour `QuerySet` mentionne aussi le déclenchement de l'évaluation par la façon `repr()`, mais ce cas n'est pas applicable à `RawQuerySet`, qui se limite à un affichage.

Pour des manipulations manuelles, l'erreur n'est pas une catastrophe puisqu'en réalité l'ordre SQL a tout de même été réalisé. Ce n'est que le retour qui est mal supporté, mais on n'en a pas besoin. Si on souhaite malgré tout éviter la remontée de l'erreur, des solutions possibles sont :

a) Capter l'erreur pour la passer sous silence :

```
>>> try:
...     list(rqs)
... except TypeError:
...     pass
...
```

© Editions ENI - All rights reserved

Ou :

```
>>> from contextlib import suppress
>>>
>>> with suppress(TypeError):
...     list(rqs)
...
```

b) Faire semblant de respecter l'attendu, avec l'ajout d'un minimum viable :

```
>>> raw_query = 'SET enable_seqscan = off; select 1 as id;'
>>> list(Message.objects.raw(raw_query))
[<Message: Message object (1)>]
```

c) Tricher en amorçant l'évaluation, mais sans aborder le traitement du retour :

```
>>> rqs.query._execute_query()
```

On notera l'appel à une méthode interne d'API (repérable à un caractère de soulignement en préfixe), donc à considérer avec les précautions d'usage.

Option 2 : Par exécution directe d'ordre SQL

Il ne s'agit plus de passer par un objet `QuerySet` ou `Manager`, mais de s'adresser en direct à la base de données, ce qui permet de s'affranchir de la couche modèle et de ne pas être contraint quant à la nature de l'ordre passé.

Le point d'entrée le plus simple est l'objet `django.db.connection`, qui représente une connexion à la base de données par défaut (celle donnée en `settings.py` par la clé `'default'` de `DATABASES`). Un objet curseur obtenu de la connexion permet ensuite de faire exécuter un ordre :

```
>>> from django.db import connection
>>>
>>> with connection.cursor() as cursor:
...     cursor.execute('SET enable_seqscan = off')
...
```

En cas de doute sur le caractère effectif du positionnement, quelle que soit la manière employée, le code suivant permet de le vérifier :

```
>>> with connection.cursor() as cursor:
...     cursor.execute('SHOW enable_seqscan')
...     print(cursor.fetchone())
...
('on',)
```

Maintenant que le paramètre est positionné, on peut constater que les nouveaux choix du planificateur sont bien ceux attendus, avec l'emploi de l'index :

```
>>> print(Message.objects.filter(sender_id__gt=1).explain())
Sort  (cost=17.88..18.07 rows=73 width=334)
 Sort Key: sent_at DESC, id DESC
 -> Bitmap Heap Scan on messenger_message
                              (cost=4.71..15.62 rows=73 width=334)
    Recheck Cond: (sender_id > 1)
    -> Bitmap Index Scan on messenger_message_sender_id_50c24489
                                 (cost=0.00..4.69 rows=73 width=0)
        Index Cond: (sender_id > 1)
```

1.2 Constitution du code SQL

Former une requête consiste à bâtir un objet `QuerySet`. Cet objet comporte un attribut nommé `query`, qui héberge un objet auxiliaire à usage interne. S'il apparaît parfois dans la documentation ou en tant que paramètre dans des méthodes, il est dit que c'est seulement dans le but de faire transiter ce bagage d'un objet à un autre. On n'est pas censé connaître la teneur de cet objet, d'autant plus qu'il est précisé qu'il ne fait pas partie de l'API publique. Respecter ces consignes n'empêche pas d'être curieux et de voir ce que cet objet peut révéler d'intéressant. En pratique, seulement son rendu en chaîne de caractères suffira.

Pour commencer, un exemple basique :

```
>>> qs = Message.objects.all()
>>> qs.query
<django.db.models.sql.query.Query object at 0x000001C86E159748>
>>> print(qs.query)
SELECT "messenger_message"."id", "messenger_message"."subject",
"messenger_message"."body", "messenger_message"."sender_id",
```

© Editions ENI - All rights reserved

```
"messenger_message"."recipient_id",
"messenger_message"."sent_at",  "messenger_message"."read_at",
"messenger_message"."sender_deleted_at",
"messenger_message"."recipient_deleted_at"
FROM "messenger_message"
ORDER BY "messenger_message"."sent_at" DESC,
         "messenger_message"."id" DESC
```

Cet affichage donne un moyen simple et immédiat pour faire un rapprochement entre une requête et sa transposition en SQL. La lecture est facile pour cette requête minimale. Le texte produit peut rapidement devenir volumineux pour des requêtes plus élaborées, avec des filtres et de nombreuses jointures. On peut être amené à faire un copier-coller de cette masse vers un éditeur, pour remettre en forme et faire des découpages en lignes pour rendre la chose humainement lisible.

Pour continuer, une requête plus affinée :

```
>>> qs = Message.objects.order_by().only('subject')\
...      .filter(sender_id__gt=1)
>>> print(qs.query)
SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message"
WHERE "messenger_message"."sender_id" > 1
```

Les critères sont : écarter le classement, limiter l'acquisition immédiate des champs au seul `subject`, et poser un critère de sélection.

Attention toutefois à l'interprétation du résultat de l'affichage, ce n'est pas nécessairement du code SQL final et valide. Ce n'est pas un dysfonctionnement, car ce n'est pas l'intention visée. En effet, en interne, il est établi séparément une chaîne de caractères avec des points de substitution et une collection des valeurs des paramètres. Pour produire l'affichage, il est fait un simple assemblage de textes, sans tenir compte d'éventuelles nécessités de guillemets, qui n'entrent en jeu que plus tard dans l'exécution. Il n'y a pas d'occasion de voir le phénomène dans les exemples précédents, car ils sont sans paramètre ou avec des nombres. Un autre exemple, avec un paramètre de texte, va le mettre en évidence :

```
>>> qs = Message.objects.order_by().only('subject')\
...      .filter(body='foo')
>>> print(qs.query)
```

```
SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message" WHERE "messenger_message"."body" = foo
```

Si on copie tel quel le texte SQL pour le coller sous pgAdmin et tenter son l'exécution, voici l'erreur qui en résulte :

```
ERROR: ERREUR: la colonne « foo » n'existe pas
LINE 1:
...M "messenger_message" WHERE "messenger_message"."body" = foo
                                                            ^

SQL state: 42703
Character: 124
```

Il faut en effet un encadrement du paramètre par des guillemets, en version simple guillemet pour PostgreSQL :

```
SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message" WHERE "messenger_message"."body" = 'foo'
```

C'est encore plus flagrant avec des dates :

```
>>> from datetime import date
>>>
>>> dt = date(2000, 12, 31)
>>> qs = Message.objects.order_by().only('subject')\
...      .filter(body='foo', sent_at__gt=dt)
>>> print(qs.query)
SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message" WHERE ("messenger_message"."body" = foo
AND "messenger_message"."sent_at" > 2000-12-31 00:00:00)
```

Quand on veut ajuster le code SQL pour le rendre valide afin de le faire exécuter ailleurs pour évaluation, le repérage des points de rectification peut devenir laborieux sur des ordres riches en paramètres. La méthode `sql_with_params()` est un moyen de se faire aider, car elle met en évidence d'une part les points de substitution au motif `%s` et d'autre part la collection de paramètres :

```
>>> print(qs.query.sql_with_params())
('SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message" WHERE ("messenger_message"."body" = %s
AND "messenger_message"."sent_at" > %s)',
('foo', datetime.datetime(2000, 12, 31, 0, 0)))
```

© Editions ENI - All rights reserved

1.3 Journalisation des requêtes SQL

Mettre en fonctionnement une journalisation des requêtes est encore un autre moyen d'observer ce qui se passe au niveau des échanges avec la base de données. Django intègre quelques facilités pour activer une telle observation.

Tout d'abord, un point important à considérer : la capture des traces des requêtes n'est mise en œuvre qu'en situation de développement, pas de production, c'est-à-dire que le paramètre de configuration `DEBUG` doit être positionné à `True`. Cette contrainte est essentiellement justifiée par des raisons de performances, car la capture représente un coût alors même qu'elle ne serait pas exploitée ensuite.

1.3.1 Observations manuelles

Les traces s'accumulent au sein d'un objet `connection` et sont accessibles sous son attribut `queries` sous forme d'une liste. Voici un exemple avec la connexion par défaut :

```
>>> from django.db import connection
>>>
>>> len(connection.queries)
12
```

Le nombre rendu varie selon le nombre d'essais antérieurs dans la session. Le stockage interne est implémenté à travers un objet qui se comporte comme une liste plafonnée : lorsque le nombre maximal d'entrées est atteint, l'arrivée de nouvelles entrées est admise, mais en fait évacuer autant à l'autre bout. Sachant que le maximum est positionné à 9000, on se dit qu'on a de la marge avant de saturer le tuyau, ceci d'autant plus que ce stockage est remis à vide au début de chaque requête HTTP.

Contrairement aux moyens d'observation évoqués précédemment, il va falloir nécessairement passer par une exécution effective de la requête SQL, pour disposer ensuite de sa trace.

Reprenons le même style de requête, en provoquant son évaluation. Peu importe si aucun enregistrement ne correspond :

```
>>> from datetime import date
>>>
>>> dt = date(2000, 12, 31)
>>> qs = Message.objects.order_by().only('subject')\
...      .filter(body='foo', sent_at__gt=dt)
>>> len(qs)  # évaluation
0
```

Puisque les traces accumulées jusqu'à présent peuvent être nombreuses, il vaut mieux limiter la visualisation à la seule dernière entrée :

```
>>> print(connection.queries[-1])
{
 'sql': 'SELECT "messenger_message"."id",
   "messenger_message"."subject" FROM "messenger_message"
   WHERE ("messenger_message"."body" = \'foo\'
   AND "messenger_message"."sent_at" > \'2000-12-31T00:00:00\'
   ::timestamp)',
 'time': '0.015'
}
```

Pour rester actuel à chaque affichage, il est important de repartir de l'objet `connection` car l'attribut `queries` rend une simple copie du stockage tel qu'il est au moment de la demande. Sinon, pour conserver une référence sur le stockage interne, on peut utiliser le code suivant en veillant à rester neutre et ne faire que de la lecture :

```
>>> ql = connection.queries_log
>>> print(ql[-1])
```

Le rendu est un dictionnaire avec deux éléments :

- `time` : le temps d'exécution de la requête par la base de données, exprimé en secondes arrondies à trois décimales.
- `sql` : la commande SQL dans son format final, donc avec les caractères guillemets adéquats.

© Editions ENI - All rights reserved

Si on souhaite prendre une copie de l'instruction SQL sans s'encombrer de l'échappement des guillemets, il est plus efficace de faire :

```
>>> print(connection.queries[-1]['sql'])
SELECT "messenger_message"."id", "messenger_message"."subject"
FROM "messenger_message"
WHERE ("messenger_message"."body" = 'foo' AND
"messenger_message"."sent_at" > '2000-12-31T00:00:00'::timestamp)
```

Visibilité des requêtes dans les vues

Dans le chapitre Création de site, des processeurs de contexte avaient été cités, parmi lesquels celui-ci :

`django.template.context_processors.debug`

À ce moment-là, il n'avait pas été jugé utile de le mettre en œuvre. Mais examinons maintenant de plus près ce qu'il apporterait.

Parce qu'il donne accès à des informations qui peuvent être sensibles en matière de sécurité, son usage obéit à des conditions restrictives : non seulement il faut être en mode `DEBUG`, mais en plus il faut que l'adresse IP du demandeur soit autorisée en étant citée dans le paramètre de configuration `INTERNAL_IPS`.

Lorsque le processeur est activé et que les conditions d'accès sont respectées, ces variables sont mises à disposition pour la vue et son gabarit :

- `debug` : un booléen à `True` pour affirmer le fonctionnement effectif dans ce mode.
- `sql_queries` : une liste de dictionnaires correspondant aux requêtes SQL réalisées jusqu'au moment où cette information est accédée.

Comme cela a déjà été vu, ces dictionnaires sont également issus de l'attribut `queries` de la connexion à une base de données. La seule différence par rapport aux manipulations précédentes est qu'il s'agit d'un cumul sur toutes les connexions déclarées.

Reste ensuite au développeur à savoir quoi faire de cette liste dans le gabarit de la vue, comment présenter les informations, comment éventuellement n'en rendre visible qu'une portion selon des filtres, etc.

Il n'est pas certain que l'effort de travail que cela induit vaille la peine comparativement aux autres moyens de prise de traces, plus simples à mettre en place.

1.3.2 Observations automatiques

Les observations qui ont été faites manuellement précédemment ont un équivalent automatisé au niveau du serveur, par l'activation d'un niveau de journalisation.

La manière simple et recommandée pour définir ses souhaits en matière de journalisation est de le faire dans le fichier de configuration avec le paramètre `LOGGING`. Par défaut, ce paramètre est un dictionnaire vide, mais cela ne signifie pas pour autant une absence totale de journalisation. Plus exactement, le paramètre définit un complément qui sera fusionné avec une base par défaut enfouie dans le code. La configuration de base est décrite dans la documentation et on peut aussi explorer sa définition sous l'attribut `DEFAULT_LOGGING` du module en `django/utils/log.py`. Ainsi, on bénéficie d'un socle, qu'il suffit d'amender, pour l'enrichir ou le raffiner.

▶ Ajoutez cette définition en fin du fichier `settings.py` :

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console_debug': {  # inspiré du 'console' par défaut
            'level': 'DEBUG',
            # pas nécessaire car implicite
            # 'filters': ['require_debug_true'],
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'handlers': ['console_debug'],
            # DEBUG pour voir les requêtes
            # INFO pour revenir à la normale
            'level': 'DEBUG',
            'propagate': False,
        },
    }
}
```

© Editions ENI - All rights reserved

La clé `version` est un impératif d'implémentation et la seule valeur valide est 1.

La clé `disable_existing_loggers` est à `True` par défaut, en raison de compatibilité antérieure, mais l'intérêt est au contraire de conserver les enregistreurs définis par le socle.

Un gestionnaire `console_debug` est ajouté. Il existe bien un gestionnaire `console` dans le socle, mais il est de niveau `INFO` et il est préférable ne pas y toucher pour ne pas avoir d'impact sur d'autres pans de la journalisation. Ce gestionnaire, avec son niveau à `DEBUG`, aura ainsi la visibilité des traces espérées. Pour faire simple, il est choisi de diriger les traces sur la console (par défaut le flux *sys.stderr*), mais toute autre voie est valable.

La clé `filters` est là en imitation de la définition du gestionnaire `console`, mais elle n'est pas indispensable puisque la condition sera forcément satisfaite, sinon les traces ne sont pas générées et donc l'enregistreur utilisateur de ce gestionnaire n'est même pas sollicité. Elle est mise en commentaires, car le mécanisme de fusion ne permet pas de référencer le filtre `require_debug_true` existant dans le socle :

```
ValueError: Unable to add filter 'require_debug_true'
```

Chaque configuration de journalisation doit être autonome et il faudrait reproduire ce filtre.

`django.db.backends` est un enregistreur intégré, utilisé pour l'émission des traces concernant les requêtes SQL à la base de données. Ces messages sont émis avec le niveau `DEBUG`. Le socle ne définit rien de particulier pour cet enregistreur, si bien que les définitions de l'enregistreur racine `django` s'appliquent et celles-ci ne considèrent que les messages à partir du niveau `INFO`. Comme il n'est pas souhaité de modifier la racine, car cela aurait un impact sur toute la hiérarchie d'enregistreurs, la préférence va à l'ajout d'une définition dédiée à l'enregistreur visé, de façon à permettre la prise en compte des messages à partir du niveau `DEBUG`. L'option `propagate` est descendue, pour éviter un doublon de sortie sur console d'éventuels messages à partir du niveau `INFO`.

Il faut repartir d'un shell renouvelé pour disposer des nouveaux paramètres de configuration :

```
D:\dj>py manage.py shell
[...]
>>> from mysite.messenger.models import Message
>>> from datetime import date
>>>
>>> dt = date(2000, 12, 31)
>>> qs = Message.objects.order_by().only('subject')\
...      .filter(body='foo', sent_at__gt=dt)
>>> len(qs)  # évaluation
(0.025) SELECT "messenger_message"."id",
"messenger_message"."subject" FROM "messenger_message"
WHERE ("messenger_message"."body" = 'foo' AND
"messenger_message"."sent_at" > '2000-12-31T00:00:00'::timestamp)
;args=('foo', datetime.datetime(2000, 12, 31, 0, 0))
0
```

Entre la ligne lançant l'évaluation et la ligne donnant son résultat se sont présentées les lignes de traces. Le format de la trace est :

```
(<time>) <sql>; args=<params>
```

Les motifs `time` et `sql` sont les mêmes que ceux expliqués à la section précédente. Par contre, le motif `params` est en plus et rappelle les arguments mis en jeu, sous forme d'un tuple Python.

Dans la situation de fonctionnement avec un serveur de développement `runserver`, on aurait de la même manière l'apparition de ce genre de traces sur la console du serveur, au fur et à mesure du traitement d'une requête HTTP, pour chacune des requêtes SQL soumises.

Le format de la trace est celui par défaut et il se contente de montrer le contenu du message journalisé. Si les traces sont nombreuses et variées, il devient utile de voir plus de caractéristiques pour faire des distinctions à la lecture, en mettant en place un metteur en forme personnalisé. C'est encore plus vrai lorsque les traces sont orientées vers un fichier, pour une éventuelle consultation différée à l'occasion de l'analyse d'un problème de fonctionnement.

© Editions ENI - All rights reserved

▶Complétez la définition :

```
LOGGING = {
    # [...]
    'formatters': {  # default is '%(message)s'
        'simple': {
            'format': '{asctime}_{name}_{levelname} - {message}',
            'style': '{',
        },
    },
    'handlers': {
        'console_debug': {  # inspiré du 'console' par défaut
            # [...]
            'formatter': 'simple',
        },
    },
    # [...]
}
```

Le style a historiquement la valeur par défaut `'%'`, pour une mise en forme dite *printf-style* (par exemple `'%(message)s'`). Cette forme est devenue désuète et il est préférable d'employer une forme plus moderne, telle que celle de `str.format()`, en l'indiquant par le choix codifié `'{'`.

La clé `format` fournit le gabarit voulu pour la trace, en citant les mots-clés des morceaux d'information qu'on veut faire apparaître. Dans un catalogue de près d'une vingtaine de noms d'attribut possibles, il en est retenu quatre, probablement les plus utiles : un horodatage, le nom de l'enregistreur, le niveau de journalisation et bien sûr le contenu du message.

En rejouant les instructions précédentes, on obtient cette fois-ci une sortie enrichie, plus facile à situer dans son contexte :

```
20nn-nn-nn nn:nn:nn,nnn_django.db.backends_DEBUG - (0.005) SELECT
"messenger_message"."id", "messenger_message"."subject" FROM
"messenger_message" WHERE ("messenger_message"."body" = 'foo' AND
"messenger_message"."sent_at" > '2000-12-31T00:00:00'::timestamp)
; args=('foo', datetime.datetime(2000, 12, 31, 0, 0))
```

Il ne faut pas être surpris si les traces sont également actives pour des requêtes à l'initiative de l'infrastructure logicielle. Par exemple, à l'occasion du lancement du serveur de développement :

```
D:\dj>py manage.py runserver
Performing system checks...

System check identified no issues (0 silenced).
20nn-nn-nn nn:nn:nn,nnn_django.db.backends_DEBUG - (0.055)
   SELECT c.relname, c.relkind
   FROM pg_catalog.pg_class c
   LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
       WHERE c.relkind IN ('r', 'v')
           AND n.nspname NOT IN ('pg_catalog', 'pg_toast')
           AND pg_catalog.pg_table_is_visible(c.oid); args=None
20nn-nn-nn nn:nn:nn,nnn_django.db.backends_DEBUG - (0.015) SELECT
"django_migrations"."app", "django_migrations"."name" FROM
"django_migrations"; args=()
Lllllll nn, 20nn - nn:nn:nn
Django version 2.1.5, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

- Pour mettre en sommeil les traces dans la suite des travaux, restaurez le niveau de journalisation de l'enregistreur :

```
LOGGING = {
   # [...]
   'loggers': {
       'django.db.backends': {
           # [...]
           'level': INFO',
           # [...]
       },
   }
}
```

Complément

Un résultat comparable à cette journalisation peut être obtenu en mettant en œuvre un intergiciel. Cette solution est déportée dans le chapitre Intergiciels pour lui servir d'exemple d'implémentation.

© Editions ENI - All rights reserved

2. Pose de traces personnalisées

Dans la section précédente, il a été exploité des informations mises à disposition par un enregistreur intégré, mais tout est basé sur le module standard de journalisation de Python. Il suffit donc de suivre ce schéma d'implémentation pour parsemer son propre code avec des points de traces.

Des traces aident à remplir des objectifs variés, dont voici quelques exemples :

- S'assurer en cours de développement que les comportements se réalisent conformément aux prévisions.
- Mémoriser des situations exceptionnelles dont on ne sait pas quoi faire et qui en principe ne devraient jamais arriver.
- Déclencher des actions de communication à propos d'événements notables, typiquement par un envoi de courriel. On pense naturellement à des cas de défauts attendus, mais il peut aussi bien s'agir de circonstances positives.
- Disposer de métriques sur l'usage de l'application.
- Garder un œil sur la bonne santé de l'application et sur son environnement (par exemple ressources disponibles, quotas, qualité du réseau, etc.).

Pour disposer d'un système de traces efficace, maîtrisé et qui apporte une réelle valeur, il faut notamment réfléchir à ces points :

- Placer les observations à des endroits stratégiques.
- Doser les quantités : pas assez de traces engendre de la frustration, car on n'a pas la visibilité espérée, et trop de traces noie le lecteur dans du superflu pouvant aller jusqu'à le décourager.
- Être à la fois concis et ne pas basculer dans le sibyllin.
- Faire une savante répartition entre les niveaux de gravité, autant pour les productions des messages de trace que pour leur filtrage et leur orientation vers des consommateurs.
- En cas d'écriture sur un espace de stockage, prévoir un mécanisme automatique pour faire du nettoyage afin d'éviter de tomber dans le piège du disque plein.

2.1 Noms des enregistreurs

Un enregistreur est identifié par un nom, à la fois pour sa configuration et pour son emploi. Prendre le nom fourni par la variable `__name__`, c'est-à-dire le nom du module courant, est une convention promue par la documentation Python et relayée par la documentation Django, avec l'idée de coller à la hiérarchie des modules, mais ce n'est qu'une suggestion, et d'ailleurs Django ne l'applique pas pour ses enregistreurs internes. Le choix du nom est donc libre. Un simple mot suffit, mais il est d'usage de tirer profit de la possibilité de construire une hiérarchie pour classifier à sa façon les messages de journalisation. Le caractère point `'.'` revêt la signification spéciale de caractère séparateur pour formaliser cette structure parent/enfant.

Le bénéfice de ce système de relation se révèle avec la faculté des messages à être propagés aux enregistreurs en amont, de façon à factoriser des traitements. Il est néanmoins possible de désactiver la propagation au stade d'un enregistreur.

Tous les enregistreurs internes de Django, émetteurs de messages, ont un nom qui commence par `'django.'`. Respecter un cloisonnement par des espaces de noms est une bonne pratique pour éviter un risque de conflit. De nombreux exemples dans des documentations donnent le nom du projet comme préfixe aux noms des enregistreurs : `project.something` ou `myproject.something`. L'idée est tout à fait valable pour des modules dédiés au projet. Elle se prête mal à des modules réutilisables puisque, dans ce cas, le niveau du projet ne doit pas intervenir. C'est pourquoi dans l'application `messenger` il sera préférable de prendre directement le nom de l'application comme racine de l'espace de noms des enregistreurs.

2.2 Exemple d'enregistreur

Pour les besoins de la démonstration, un message quelconque est produit dans une vue. Toujours pour être démonstratif, un enregistreur fils est employé, ce qui permet de faire une distinction avec l'enregistreur racine. Pour se conformer au choix d'espace de noms justifié à la section précédente et pour rester intuitif, l'enregistreur porte le nom `messenger.views`.

© Editions ENI - All rights reserved

▶Complétez le fichier des vues :

`mysite\messenger\views.py`

```
import logging
# [...]

logger = logging.getLogger('messenger.views')


def index(request):
    logger.debug('message de démonstration')
    # [...]
# [...]
```

Du côté de la configuration, un premier enregistreur est ajouté en tant que racine pour l'application. Le souhait est de lui donner pour rôle d'être un collecteur général, c'est-à-dire sans filtre pour ne rien perdre en tant que dernier maillon. Pour surtout ne pas inonder un opérateur avec un flot continu de texte, les données sont dirigées vers un simple fichier en accumulation.

▶Complétez le fichier de configuration :

`mysite\settings.py`

```
LOGGING = {
    [...]
    'handlers': {
        [...]
        'file_appending': {
            'class': 'logging.FileHandler',
            'formatter': 'simple',
            'filename': os.path.join(_SITE_ROOT, 'logs/app.log'),
        },
    },
    'loggers': {
        [...]
        'messenger': {
            'handlers': ['file_appending'],
            'level': 'DEBUG',
        },
    }
}
```

Un gestionnaire de nature fichier est défini. Par défaut, le fichier est ouvert en mode écriture, avec ajout à la fin, sans limite de taille. Par défaut, s'il n'existe pas déjà, il est créé et vide au prochain démarrage du serveur.

Le chemin vers le répertoire de stockage du fichier journal doit être existant, car il ne sera pas automatiquement créé.

Il n'est pas nécessaire de préciser de niveau de gravité sur le gestionnaire. En effet, la valeur par défaut qui s'applique est `NOTSET`, causant le traitement de tous les messages, comme il est souhaité.

▶Créez le répertoire d'hébergement des journaux :

```
D:\dj>cd mysite
D:\dj\mysite>md logs
```

Un enregistreur fils doit ensuite être ajouté. Pour faire au plus simple, son gestionnaire sera celui déjà présent pour sortir sur la console.

▶Complétez le fichier de configuration :

`mysite\settings.py`

```
LOGGING = {
    [...]
    'loggers': {
        [...]
        'messenger.views': {
            'handlers': ['console_debug'],
            'level': 'DEBUG',
        },
    }
}
```

Sur lancement du serveur de développement, on constate l'arrivée du fichier journal, initialement vide :

```
D:\dj>dir mysite\logs
   <DIR>          .
   <DIR>          ..
                0 app.log
```

© Editions ENI - All rights reserved

Sur un accès à la page `/messenger/inbox/`, on constate la sortie d'une trace sur la console :

```
D:\dj>py manage.py runserver
[...]
Quit the server with CTRL-BREAK.
[...]_messenger.views_DEBUG - message de démonstration
[...] "GET /messenger/inbox/ HTTP/1.1" 200 54
```

Par contre, le journal n'est pas immédiatement alimenté et reste vide pour l'instant. Il y a bien un système de tampon qui amasse les données pour optimiser les écritures sur le flux, mais il n'est pas en cause, car un vidage est provoqué après chaque émission d'un message. La retenue se situe dans des couches plus basses, en lien avec le système d'exploitation, pour des raisons d'optimisation des accès au disque.

Le phénomène n'est pas propre à la journalisation, mais concerne les accès aux fichiers en général sous Python.

Une expérimentation permet de le prouver. Attention, le code présenté n'est pas une implémentation utilisable, mais un bricolage, conditionné par la connaissance de la configuration de la journalisation du projet et de l'implémentation non publique du module de journalisation Python. Le principe est de forcer une synchronisation du flux auprès du système de fichiers, juste après la génération du message :

```
    logger.debug('message de démonstration')
    # pour démo, NE PAS employer pour du code réel :
    os.fsync(logger.parent.handlers[0].stream.fileno())
```

Avec cet ajout de démonstration, le fichier est mis à jour en continu, preuve que le restant du code est correctement écrit. Avoir des écritures sur disque différées n'est de toute façon pas un handicap, car ce mode de journalisation n'est pas fait pour être suivi en direct.

Un contournement propre du problème s'obtient en arrêtant le serveur, ce qui entraîne la synchronisation. Le message est alors disponible :

```
D:\dj>py manage.py runserver
[...]
^C
D:\dj>dir mysite\logs
   <DIR>          .
```

```
   <DIR>          ..
                  74 app.log

D:\dj>type mysite\logs\app.log
[...]_messenger.views_DEBUG - message de dÚmonstration
```

À la sortie sur console du contenu du fichier, un petit souci apparaît pour le caractère accentué. Le contenu même du fichier n'est pas en cause, car si on l'ouvre sous un éditeur de texte, la lettre est respectée et on voit que l'encodage est de type ANSI.

Dans la définition du gestionnaire, il n'a pas été cherché à imposer un encodage, ce qui aurait pu être fait en précisant par exemple :

```
'encoding': 'utf8',
```

L'encodage par défaut de la plateforme s'applique. Il peut être découvert par ce code :

```
D:\dj>py -c "import locale; print(locale.getpreferredencoding())"
cp1252
```

En comparaison, une fenêtre d'invite de commandes a par défaut un autre encodage, comme le montre le code suivant :

```
D:\dj>chcp
Page de codes active : 850
```

La commande `chcp` (*CHange Code Page*) sans paramètre affiche le numéro de la page de codes active.

La page de numéro 850 est une antiquité qui traîne pour des motifs de maintien de compatibilité. Le numéro 1252 a pris sa place sous Windows (pour les langues d'Europe occidentale). Il suffit de changer la page de codes pour accorder la console avec les contenus :

```
D:\dj>chcp 1252
Page de codes active : 1252

D:\dj>type mysite\logs\app.log
[...]_messenger.views_DEBUG - message de démonstration
```

L'affichage des caractères accentués devient correct.

© Editions ENI - All rights reserved

2.3 Autres exemples de gestionnaires

Parmi les natures de gestionnaire les plus couramment utilisées, il a déjà été exploré :

- La sortie sur console, soit en mentionnant l'existence du gestionnaire intégré `console`, valable pour le niveau `INFO` et en mode Développement, soit avec un gestionnaire personnalisé `console_debug` valable pour le niveau `DEBUG`.
- L'accumulation infinie dans un fichier, avec un gestionnaire personnalisé `file_appending`.

Un autre cas traditionnel est disponible dans le socle intégré :

- L'envoi d'un courriel aux administrateurs du site (paramètre de configuration `ADMINS`), avec `mail_admins`, pour un niveau à partir de `ERROR` et en mode Production.

Des constructions plus élaborées ou moins courantes sont maintenant présentées dans cette section.

2.3.1 Rotation de journaux

Le gestionnaire `file_appending`, utilisé précédemment, présente l'avantage d'être basique. Il a pourtant un désavantage : le fichier unique a la faculté de grossir indéfiniment et potentiellement causer un problème de saturation d'espace de stockage. Ceci ne satisfait pas à l'un des points énoncés au début pour juger de la qualité d'un système de traces. Éluder la question en disant qu'on a de l'espace plus qu'il n'en faut avec des disques récents n'est pas un argument recevable par principe, et dans tous les cas personne n'a envie ensuite de manipuler un fichier de taille astronomique. La solution typique consiste à régulièrement mettre de côté le journal en cours et continuer sur un nouveau fichier vierge. Le critère pour décider du renouvellement est habituellement basé soit sur la quantité de données, soit sur une découpe du temps. De plus, le nombre de fichiers conservés est plafonné, ce qui s'apparente à une notion de profondeur d'historique. L'ensemble du mécanisme apporte ainsi une certaine maîtrise du volume de stockage : elle est garantie avec un plafond de taille ; elle n'est pas infaillible avec une répartition temporelle, mais normalement très improbable.

Le module de journalisation Python propose deux gestionnaires de nature rotatoire :

– `RotatingFileHandler`, basé sur la quantité de données.

– `TimedRotatingFileHandler`, basé sur le temps.

Une répartition des journaux fondée sur le temps est généralement plus abordable pour un opérateur, alors que le volume de données est plutôt un critère orienté vers la technique. En effet, avec une idée de la fenêtre de temps où une recherche doit se faire, il est plus naturel et rapide de cibler un fichier en particulier dans le lot.

Pour la mise en pratique à suivre, la version temporelle est donc préférée.

▶ Complétez le fichier de configuration :

`mysite\settings.py`

```
[...]
'handlers': {
    [...]
    'file_rotating': {
        'class': 'logging.handlers.TimedRotatingFileHandler',
        'formatter': 'simple',
        'filename': os.path.join(_SITE_ROOT, 'logs/time.log'),
        'when': 's',
        'interval': 15,
        'backupCount': 3,
    },
},
'loggers': {
    [...]
    'messenger':{
        #'handlers': ['file_appending'],
        'handlers': ['file_rotating'],
        [...]
},
```

La fréquence de rotation est ici volontairement très élevée, quinze secondes, de façon à pouvoir observer le fonctionnement sans délai. Des valeurs plus réalistes sont plutôt de l'ordre de la journée ou de la semaine. Trois fichiers au plus sont maintenus : lorsqu'une rotation se produit et si ce chiffre est déjà atteint, le fichier le plus ancien est détruit pour faire de la place.

© Editions ENI - All rights reserved

Le nouveau gestionnaire est affecté à l'enregistreur `messenger`, en remplacement de l'ancien, mais il pourrait être mis en complément si cela avait du sens, par exemple si les espaces de stockage reposent sur des technologies particulières ou visent des intentions particulières (archivage, conservation légale, disques réseau de très haute capacité, etc.).

Avec le navigateur, des actualisations répétées de la page, plus ou moins espacées en jouant sur le délai des quinze secondes, produisent les fichiers journaux successifs :

```
D:\dj>dir mysite\logs
nn/nn/20nn  20:12                 0 time.log
nn/nn/20nn  20:10               518 time.log.20nn-nn-nn_20-10-46
nn/nn/20nn  20:11               222 time.log.20nn-nn-nn_20-11-09
nn/nn/20nn  20:11               148 time.log.20nn-nn-nn_20-11-29
```

À la rotation du journal, le fichier courant est renommé avec un suffixe construit sur la date et l'heure correspondant à l'instant où le fichier a été créé, non celui de la rotation. On le voit aussi grâce au contenu :

```
D:\dj>type mysite\logs\time.log.20nn-nn-nn_20-11-29
20nn-nn-nn 20:11:29,360_messenger.views_DEBUG - message de dém[...]
20nn-nn-nn 20:11:34,169_messenger.views_DEBUG - message de dém[...]
```

Comme expliqué précédemment avec le fichier `app.log` et pour la même raison de synchronisation du système de fichiers, le fichier `time.log` reste présenté avec une taille nulle.

2.3.2 Transmission vers une machine distante

Dans cette partie, l'objectif est de transporter les messages de journalisation vers un site distant. Le cas d'usage va être l'observation du flux par un opérateur, à l'occasion d'une session d'assistance et de diagnostic à distance.

Le module de journalisation Python propose deux gestionnaires de base pour communiquer avec une machine à travers une connexion réseau :

- `SocketHandler`, basé sur une prise TCP (*Transmission Control Protocol*).
- `DatagramHandler`, basé sur une prise UDP (*User Datagram Protocol*).

TCP est un protocole de transport fiable, en mode connecté, servant de socle à de nombreux protocoles applicatifs (web, transfert de fichier, messagerie, etc.). UDP est un protocole non fiable, en mode non connecté, servant à l'envoi de messages courts, appelés datagrammes. Il est dit non fiable car il n'y a pas de garantie sur la livraison des datagrammes face à la qualité du réseau, c'est-à-dire pas de protection contre la perte ou la duplication des datagrammes, ni contre un ordre d'arrivée différent de celui d'envoi. En contrepartie, ce protocole est plus léger et rapide que son frère TCP. Il convient dans les situations où les compromis de fiabilité n'ont pas de conséquence ou sont compensés par la couche applicative.

La version UDP est bien adaptée au contexte, car la fiabilité n'est pas une nécessité : la perte d'un message est admissible et aucune réponse n'est attendue.

Face à l'émission des messages réalisée par une instance de gestionnaire `DatagramHandler`, il faut un récepteur. En voici une implémentation rustique pour commencer, écrite en quelques lignes de Python :

`dj\logsUDPServer.py`

```
import socketserver

class MyHandler(socketserver.BaseRequestHandler):
    def handle(self):
        print(self.request[0])

if __name__ == "__main__":
    HOST, PORT = "0.0.0.0", 9696
    with socketserver.UDPServer((HOST, PORT), MyHandler) as srv:
        srv.serve_forever()
```

Le serveur se met à l'écoute, sur toutes ses interfaces réseau pour être le plus ouvert. Le port est quelconque parmi les plages de ports publics et libres. Le serveur tourne indéfiniment. Il est arrêté proprement par un Ctrl-Break. Le gestionnaire de requête se contente d'afficher les données brutes reçues.

En théorie le serveur est censé être hébergé sur une autre machine. Pour l'exercice, on se contentera de la machine locale pour jouer le rôle.

© Editions ENI - All rights reserved

- Ouvrez une console à part pour faire tourner ce serveur :

```
D:\dj>py logsUDPServer.py
```

- Complétez le fichier de configuration `mysite\settings.py` :

```
[...]
'handlers': {
    [...]
    'remote': {
        'class': 'logging.handlers.DatagramHandler',
        'host': 'localhost',
        'port': 9696,
    },
},
'loggers': {
    [...]
    'messenger': {
        # 'handlers': ['file_appending'],
        # 'handlers': ['file_rotating'],
        'handlers': ['remote'],
        [...]
},
```

Il est inutile de spécifier une clé `formatter`, ce type de gestionnaire n'en tient pas compte.

Avec une actualisation de la page du navigateur, une trace apparaît :

```
D:\dj>py logsUDPServer.py
b'\x00\x00\x02-}q\x00(X\x04\x00\x00\x00nameq\x01X\x0f\x00\x00\x00
messenger.viewsq\x02X\x03\x00\x00\x00msgq\x03X\x19\x00\x00\x00
message de d\xc3\xa9monstrationq\x04X\x04\x00\x00\x00argsq\x05NX\[...]
```

Grossièrement, tout fonctionne, sauf que ce contenu est illisible. En effet, une lecture attentive de la documentation du gestionnaire enseigne que ce n'est pas simplement le texte journalisé qui est transmis, mais une représentation sérialisée de l'enregistrement. De plus, il s'agit d'un format de sérialisation propre à Python, mais heureusement le code de réception est aussi du Python. Il est nécessaire de procéder à un déballage propre inverse à la réception.

▶Modifiez le serveur comme suit :

dj\logsUDPServer.py

```
import pickle
[...]

    def handle(self):
        ip = self.client_address[0]
        d = pickle.loads(self.request[0][4:])  # préfixe ignoré
        print(f"{ip}\t{d['asctime']}_"
              f"{d['name']}_{d['levelname']} - {d['msg']}")
```

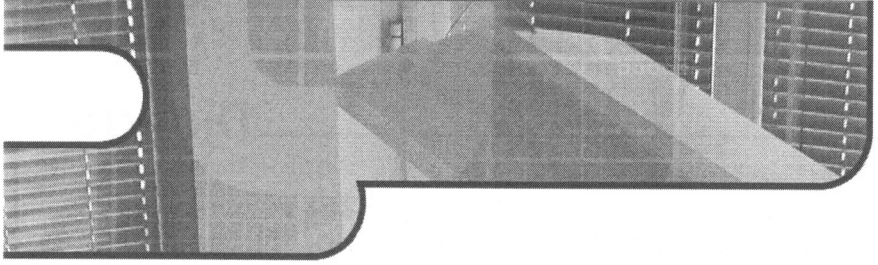
La documentation (v3.7.2) donne peu de précision sur le format binaire de transmission, en particulier sur la présence, en préfixe à la charge utile, de la longueur de cette charge, codifiée sur quatre octets. Ceci explique le besoin d'écarter ces premiers octets avant de considérer la sérialisation.

Désormais, la trace est affichée dans un aspect lisible, en imitation du format employé jusqu'à présent avec le metteur en forme `simple`. Elle est enrichie de l'adresse IP de l'émetteur, nécessaire si on est dans une situation d'émetteurs multiples :

```
D:\dj>py logsUDPServer.py
127.0.0.1  20nn-nn-[...]_messenger.views_DEBUG - message de démo[...]
```

Le dictionnaire offre d'autres attributs de l'enregistrement de journalisation, mais de moindre valeur informative.

© Editions ENI - All rights reserved



Chapitre 6
Intergiciels

1. Introduction

Le terme « intergiciel » sera utilisé pour désigner ce que la documentation en langue anglaise appelle *middleware* et ce que d'autres peuvent appeler « logiciel médiateur ». Le préfixe inter ou *middle* souligne clairement qu'il s'agit d'un morceau de logiciel placé à la frontière entre deux blocs importants dans la chaîne de traitement de l'information. S'agissant de Django, la chaîne dans laquelle le but est d'insérer les maillons est celle du traitement d'un verbe HTTP, à la fois dans le sens entrant, pour la requête, mais aussi dans le sens sortant, pour la réponse.

Le thème a été succinctement abordé dans le chapitre Création de site à l'occasion du parcours du fichier de configuration. Il n'y était mentionné que des composants déjà inclus dans l'infrastructure logicielle, mais il est aussi possible d'écrire ses propres composants.

Pour expérimenter la création d'un intergiciel, le travail va porter à nouveau sur le sujet de la prise de traces des requêtes SQL étudié dans le chapitre Traces et journalisation. Imaginons que la solution par journalisation ne soit pas proposée ; hypothèse tout à fait plausible puisque les facultés de journalisation ont été une nouveauté de la version 1.3.

2. Création d'une application dédiée à l'outillage

L'intergiciel à créer pourrait être logé au sein de l'application `messenger`, mais puisque le service rendu n'est pas spécifique à cette application, il est plus propre de le déporter dans une application générique. Ainsi, celle-ci pourra être réutilisable pour d'autres projets. Elle pourra aussi être enrichie ultérieurement avec d'autres fonctionnalités potentielles. Puisque la réflexion s'oriente vers une boîte à outils, nommons cette application `toolbox`.

▶Créez cette application :

```
D:\dj>cd mysite
D:\dj\mysite>py ..\manage.py startapp toolbox
```

L'assistant a joué son rôle en apportant des squelettes de fichiers, mais la plupart ne seront pas utilisés étant donné le rôle spécial assigné à cette application. Rien n'empêche de les laisser et de les ignorer. Pourtant, ils peuvent être perçus comme du bruit sur le plan de la maintenance, car ils pourraient soulever des questions et des doutes sur la raison ou la nécessité de leur présence. Pour couper court à ces incertitudes, le plus pragmatique est de les éliminer tout de suite. Au pire, il sera toujours possible d'amener un exemplaire d'un fichier dont l'utilité apparaîtrait plus tard.

▶Éliminez les répertoires et fichiers superflus :

```
D:\dj\mysite>cd toolbox
D:\dj\mysite\toolbox>rmdir /s/q migrations
[...]site\toolbox>del admin.py apps.py models.py tests.py views.py
```

Finalement, il ne reste que :

```
mysite\
└ toolbox\
   └ __init__.py
```

© Editions ENI - All rights reserved

3. Implémentation d'un intergiciel

3.1 Mise en place du cadre

Les intergiciels peuvent se situer n'importe où, du moment qu'ils sont visibles du chemin Python. Il n'y a pas de bénéfice à faire l'original, alors faisons comme tout le monde et écrivons les composants dans un fichier nommé `middleware.py` (au singulier même s'il héberge plusieurs composants).

La façon d'écrire un intergiciel et le nom du paramètre de configuration ont été entièrement révisés dans la version 1.10.

Pour assurer une continuité de compatibilité, il a suffi que les composants écrits dans l'ancien style héritent de ce nouvel auxiliaire pour ne pas nécessiter d'autres refontes de codage :

```
django.utils.deprecation.MiddlewareMixin
```

Django a adopté cette technique pour les intergiciels fournis avec son code, ainsi que beaucoup d'auteurs de paquets tiers. Ceci explique pourquoi on n'y trouve pas d'écriture dans le nouveau style. N'étant pas contraint à ce devoir, le code présenté sera exprimé avec la syntaxe la plus récente.

La première démarche consiste à poser le squelette, tel qu'il est donné dans la documentation, en ajoutant seulement deux instructions de trace pour temporairement faire la preuve du bon fonctionnement :

▶Créez le module de l'intergiciel :

```
mysite\toolbox\middleware.py
```

```
class WatcherMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        print('avant la vue')

        response = self.get_response(request)

        print('après la vue')

        return response
```

Ajoutez l'intergiciel à la liste, dans les paramètres de configuration :

`mysite\settings.py`

```
MIDDLEWARE = [
    'mysite.toolbox.middleware.WatcherMiddleware',
    # [...]
]
```

L'ordre des éléments dans la liste a son importance, car ces éléments doivent être considérés comme des cercles concentriques, avec la vue au centre. Ainsi, le premier de la liste est le premier à voir passer la requête et aussi le dernier à rendre la réponse.

L'intérêt de notre intergiciel, en tant qu'observateur, est de ne pas manquer une soumission à la base de données, bien sûr de la part de la vue, mais aussi de la part d'un autre intergiciel. C'est pourquoi, en le situant en tête de liste, il aura la main en dernier dans la phase de remontée de la réponse.

Il n'est pas nécessaire de mentionner l'application dans la liste `INSTALLED_APPS`, pour l'usage que l'on fait de son composant.

Sur un accès à `localhost:8000/` face au serveur de développement, on constate les traces :

```
avant la vue
après la vue
[...] "GET / HTTP/1.1" 200 24
```

3.2 Écriture d'un traitement

Maintenant que le cadre est posé et que son fonctionnement est mis en évidence, on peut se recentrer sur le véritable traitement à implémenter dans l'intergiciel.

Les instructions temporaires d'affichage peuvent être retirées. À la place, il est mis l'affichage des traces de requêtes.

Complétez le fichier :

`mysite\toolbox\middleware.py`

```
from django.db import connection
```

© Editions ENI - All rights reserved

```
class WatcherMiddleware:
    # [...]
    def __call__(self, request):
        response = self.get_response(request)

        print(connection.queries)

        return response
```

Plutôt que la page d'accueil du site, il est plus réaliste de demander une page de l'application, telle que `localhost:8000/messenger/inbox/`. Les lignes suivantes apparaissent sur la console :

```
[]
[...] "GET /messenger/inbox/ HTTP/1.1" 200 54
```

Tout fonctionne bien, sauf que la liste est vide puisqu'il n'est fait aucun accès à la base de données. Pour aller au bout de la démonstration, des interrogations à la base de données doivent être introduites quelque part, par exemple ces demandes purement fictives dans la vue :

`mysite\messenger\views.py`

```
# [...]
from .models import Message


def index(request):
    demo = list(Message.objects.all())  # démo middleware
    demo = Message.objects.only('body').get(pk=2)  # démo
    return render(request, 'messenger/index.html')
# [...]
```

Cette fois-ci, avec un rafraîchissement de la page, des requêtes apparaissent dans la trace :

```
[{'sql': 'SELECT "messenger_message"."id",
"messenger_message"."subject", "messenger_message"."body",
"messenger_message"."sender_id",
"messenger_message"."recipient_id",
"messenger_message"."sent_at", "messenger_message"."read_at",
"messenger_message"."sender_deleted_at",
"messenger_message"."recipient_deleted_at" FROM
```

```
"messenger_message" ORDER BY "messenger_message"."sent_at" DESC,
"messenger_message"."id" DESC', 'time': '0.030'}, {'sql': 'SELECT
"messenger_message"."id", "messenger_message"."body" FROM
"messenger_message" WHERE "messenger_message"."id" = 2',
'time': '0.005'}]
[...] "GET /messenger/inbox/ HTTP/1.1" 200 54
```

Pour faciliter la lecture, il est profitable de faire appel au module `pprint` :

`mysite\toolbox\middleware.py`

```
from pprint import pprint
# [...]
class WatcherMiddleware:
    # [...]
    def __call__(self, request):
        # [...]
        pprint(connection.queries)
        # [...]
```

Ainsi, les traces deviennent :

```
[{'sql': 'SELECT "messenger_message"."id", "messenger_message"[...]
         '"messenger_message"."body", "messenger_message"."sen[...]
         '"messenger_message"."recipient_id", "messenger_messa[...]
         '"messenger_message"."read_at", '
         '"messenger_message"."sender_deleted_at", '
         '"messenger_message"."recipient_deleted_at" FROM "mes[...]
         'ORDER BY "messenger_message"."sent_at" DESC, '
         '"messenger_message"."id" DESC',
  'time': '0.000'},
{'sql': 'SELECT "messenger_message"."id", "messenger_message"[...]
         '"messenger_message" WHERE "messenger_message"."id" =[...]
  'time': '0.000'}]
[...] "GET /messenger/inbox/ HTTP/1.1" 200 54
```

© Editions ENI - All rights reserved

3.3 Alternance de mise en/hors service

Avoir ces traces et peut-être celles d'autres sources peut ne pas être souhaité en permanence. Ici, la situation est simple : il suffirait de basculer en commentaire la déclaration de l'intergiciel dans le fichier de configuration. Mais si on imagine qu'il existe de nombreux points à basculer, il faudrait les répertorier et la manipulation devient fastidieuse. Avoir en quelque sorte un interrupteur central serait appréciable. Une solution possible est la mise en place d'un paramètre (éventuellement plusieurs si des distinctions sont nécessaires) dans le fichier de configuration.

Pour la démonstration, une solution alternative va être présentée, basée sur l'examen d'une variable d'environnement. De cette façon, il n'est même plus nécessaire de toucher au code pour tourner avec ou sans les traces. Le nom `DJANGO_DEBUG_VERBOSE` est choisi pour la variable d'environnement.

Il est permis aux intergiciels de décider de leur propre mise hors service au moyen de la levée de l'exception `MiddlewareNotUsed` lors de leur initialisation.

Cette capacité est mise à profit dans la méthode `__init__()` :

`mysite\toolbox\middleware.py`

```
import os
# [...]

from django.core.exceptions import MiddlewareNotUsed
# [...]
class WatcherMiddleware:
    def __init__(self, get_response):
        if not os.getenv('DJANGO_DEBUG_VERBOSE'):
            raise MiddlewareNotUsed
        self.get_response = get_response
    # [...]
```

Immédiatement, les traces ne sont plus là sur un nouvel accès à la page. Pour les faire revenir, le serveur doit être relancé après avoir positionné la variable d'environnement. Sa valeur est sans importance, il suffit d'une chaîne de caractères non vide :

```
D:\dj>set DJANGO_DEBUG_VERBOSE=set

D:\dj>py manage.py runserver
[... avec traces ]
```

Pour désactiver les traces, la variable doit être retirée. Un retrait se réalise en affectant une valeur vide à la variable :

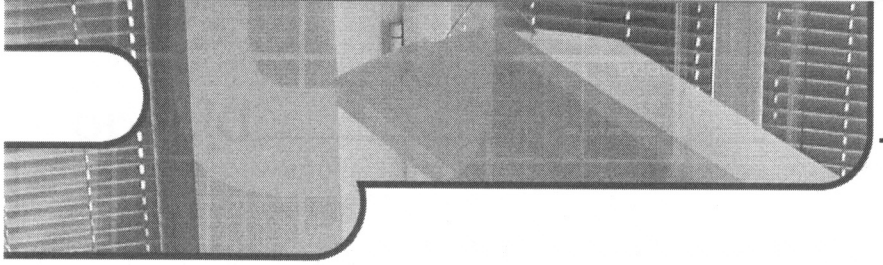
```
D:\dj>set DJANGO_DEBUG_VERBOSE=

D:\dj>py manage.py runserver
[... sans traces ]
```

Ceci explique pourquoi la valeur `set` est préférée à d'autres valeurs telles que `true`, `1`, ou `on`, car cela pourrait laisser supposer à tort que la désactivation se fait par les valeurs symétriques `false`, `0` ou `off`.

L'intergiciel présenté doit être considéré comme un exemple d'implémentation. En effet, si l'intention se réduit à une visibilité sur les ordres SQL, alors il offre peu d'avantages par rapport à l'activation d'un enregistreur natif comme cela est expliqué dans le chapitre Traces et journalisation.

On peut aussi lui reprocher de montrer les ordres, en bloc, à la fin de la requête, alors que l'enregistreur les sort au fil du traitement donc entrelacés avec d'autres traces, ce qui aide à suivre le parcours du processus. Le raisonnement est différent s'il s'agit de poursuivre d'autres buts, comme par exemple faire des comptages, des statistiques, de la détection de situations anormales, etc.

© Editions ENI - All rights reserved



Chapitre 7
Vues

1. Fonctions ou classes

Une vue peut s'écrire soit sous forme d'une fonction, soit sous forme d'une classe. L'application a déjà un exemplaire de chacune des formes : la fonction `index`, introduite dans le chapitre Création de site, et la classe `IndexView`, introduite dans le chapitre Routage.

Avoir deux façons pour un même but est inhabituel sous Django, mais l'explication est historique : à l'origine, les vues devaient être implémentées par des fonctions. Au fil du temps, ce schéma a montré ses limites : beaucoup de logique conditionnelle pour traiter les variations (les verbes HTTP par exemple) ; code trop verbeux ; difficulté à factoriser d'où des répétitions ou, pour les éviter, une organisation du code confuse. Une autre architecture, fondée sur des classes, a été introduite en version 1.3. L'intention n'est pas de prendre la place de l'écriture en fonctions, mais d'offrir une alternative avec des avantages. Ce changement de paradigme a pu être vécu comme une épreuve pour des développeurs habitués depuis longtemps à garder une totale visibilité du fonctionnement à travers leurs fonctions. En effet, avec des classes, on raisonne plutôt en assemblage et imbrication, en héritant de composants de base et en surchargeant des attributs et des méthodes seulement là où c'est nécessaire, d'où un certain effort mental pour garder une compréhension du tableau entier dont finalement on ne remplit que quelques cases et l'essentiel du canevas reste invisible en arrière-plan.

Il n'est donc pas surprenant d'avoir à consulter le code source des composants de base pour se remettre en tête la séquence des appels et repérer les endroits à personnaliser. On peut aussi sentir ce besoin d'accompagnement à la migration dans la documentation, puisqu'à de multiples reprises elle décrit la transposition de la forme fonction vers la forme classe.

Remarque

Certains utilisent les sigles CBV et FBV en écriture abrégée pour respectivement « class-based views » et « function-based views », mais il ne s'agit pas d'une dénomination régulière et elle ne fait pas partie de la documentation.

Évidemment, la syntaxe en classe a la faveur par rapport à la syntaxe en fonction. Pour autant, il ne serait pas juste d'utiliser les mots « ancienne forme » et « nouvelle forme ». On verra plutôt les termes « autre forme » ou « forme alternative ». Les deux façons ont chacune leurs avantages et inconvénients. Il est tout autant absurde de s'obstiner à continuer à écrire seulement des fonctions que de se contraindre à tout écrire en classes.

Le contrat d'une vue est d'être une entité appelable, d'accepter au minimum un paramètre de nature `HttpRequest`, et de renvoyer un objet `HttpResponse` ou de lever une exception.

L'infrastructure logicielle offre une panoplie de vues intégrées, dites aussi génériques, pour remplir les rôles les plus communément rencontrés. Ces composants sont présentés dans les sections suivantes. Dans l'immédiat, le point majeur à connaître est que toutes les vues auront une chaîne d'héritage qui aboutit à la classe mère `View`. Du fait de son statut particulier, bien que résidant sous le module `django.views.generic.base`, elle est rendue aussi importable du module `django.views` et c'est la seule avec cette faculté. Alors que le code source de la classe est en réalité hébergé plus bas dans un sous-module, on écrit donc communément :

```
from django.views import View
```

Cette vue racine est porteuse de la méthode `as_view()`, qui rend une entité appelable servant de point d'entrée au processus de traitement de la requête, et qu'on fournit aux appels à `path()` dans `urls.py`.

© Editions ENI - All rights reserved

Elle fournit aussi une implémentation par défaut pour la méthode `dispatch()`, en déléguant le traitement de la requête à une méthode spécialisée dont le nom est directement induit du verbe HTTP sollicité, en minuscules (par exemple `GET` se traduit en `get()`). Le refus des verbes qu'on ne souhaite pas supporter et une implémentation pour le verbe `OPTIONS` sont encore d'autres services pris en charge par cette vue.

2. Vues intégrées

2.1 Vues de base

La section précédente a introduit la classe `View` en tant que base des vues. Sur cette base, d'autres vues sont mises à disposition pour satisfaire les besoins classiquement rencontrés. Ainsi, la vue `RedirectView` a déjà été utilisée dans le chapitre Routage.

La vue `TemplateView` est probablement la plus élémentaire, puisqu'il s'agit de faire le rendu d'un gabarit. La vue `index()`, introduite dans le chapitre Création de site au plus simple, c'est-à-dire sous forme de fonction dans le fichier `main\views.py`, peut être éliminée en la remplaçant par une vue générique, directement citée dans le fichier de routage :

`mysite\urls.py`

```
[...]
from django.views.generic import TemplateView

#from mysite.main import views

urlpatterns = [
    [...]
    # path('', views.index),  # remplacé par :
    path('', TemplateView.as_view(template_name='index.html')),
    [...]
```

Les paramètres dans l'appel à `as_view()` sont passés au constructeur de la vue et celui-ci les traite comme des affectations de valeurs aux attributs de l'instance. Ce mécanisme permet donc de personnaliser à son goût une vue générique et au minimum de fournir les attributs obligatoires.

Dans le cas de `TemplateView`, il est indispensable de lui dire quel est le gabarit à employer, car il n'y a pas de valeur par défaut. Les paramètres désignant les mêmes objets pour chaque appel à la vue, il ne faudrait pas commettre l'erreur de fournir un objet mutable (un dictionnaire ou une liste par exemple) et de permettre la modification de cet objet au sein de la vue, sous peine d'interférences entre utilisateurs.

Puisque l'application `main` est ainsi déchargée du rendu de la page d'accueil, il n'est pas cohérent de lui laisser le fichier gabarit `index.html` sous son arborescence. Son nouvel emplacement naturel doit être de portée plus générale et cité dans le paramètre de configuration `TEMPLATES.DIRS`. Ceci a justement déjà été préparé dans le chapitre Création de site, avec le dossier `mysite\templates`.

- Déplacez le gabarit :

```
D:\dj>move mysite\main\templates\index.html mysite\templates
        1 fichier(s) déplacé(s).
```

- Actualisez la page http://localhost:8000 pour constater que la page servie est toujours correcte.

Globalement, tout fonctionne comme avant, avec le bénéfice qu'utiliser une vue générique a épargné la charge d'écrire son propre exemplaire.

2.2 Vues génériques

Les autres vues génériques se répartissent dans trois catégories :

- Les vues d'affichage remplissent le besoin classique de présenter d'une part une liste d'objets et d'autre part le détail d'un objet.
- Les vues d'édition proposent la gestion de formulaire (validation des champs et retour à l'affichage en cas d'erreur) ainsi que les cas usuels de création, modification ou destruction d'un objet.

© Editions ENI - All rights reserved

- Les vues dédiées aux dates, essentiellement orientées sur l'exploration d'archives, permettent de naviguer dans une masse d'objets selon un axe temporel de plus en plus fin : année, mois, semaine, jour. Cette catégorie est assez spécialisée et d'usage beaucoup moins courant que les deux précédentes.

3. Greffons

Les vues génériques ont une valeur appréciable en mettant à disposition une implémentation déjà écrite pour des traitements récurrents. Elles ont un premier degré de personnalisation, facile d'accès à travers le positionnement d'attributs. Un exemple déjà vu est l'attribut `template_name` de la vue `TemplateView`. Malgré tout, il arrive qu'on ait besoin d'une vue très similaire à une vue générique et pourtant aucune ne colle parfaitement au besoin. Il serait dommage d'avoir à tout écrire depuis la base `View`. Heureusement, les vues sont conçues sur une architecture modulaire, qui privilégie l'assemblage de pièces élémentaires. L'héritage multiple de classes, possible en langage Python, est le principe qui permet cette construction. Une vue se construit sur la base d'une autre vue, avec des enrichissements apportés par héritage complémentaire d'une ou plusieurs classes auxiliaires. Cet auxiliaire est désigné par la documentation Django (anglaise et française) sous le terme *mixin*. Dans cet ouvrage, il est adopté une traduction par le terme français greffon.

Par exemple, la vue `TemplateView` est construite ainsi :

```
class TemplateView(TemplateResponseMixin, ContextMixin, View):
```

Elle est basée directement sur la vue de base et deux greffons lui sont adjoints. Par convention, pour faciliter la distinction, le nom donné à un greffon est suffixé par le motif `Mixin`, de la même manière que le nom d'une vue est suffixé par le motif `View`. Les greffons doivent se placer avant la vue parente, de façon à pouvoir apporter une surcharge qui soit prioritaire par rapport au comportement établi par la vue parente. Pour rappel, l'algorithme de recherche d'attributs fait une exploration selon ces règles : le plus profond d'abord, de gauche à droite. On retrouve dans la documentation des vues une référence à cette notion importante d'ordre sous le sigle anglais MRO (*Method Resolution Order*).

La classe `TemplateView` définit la méthode `get()` pour satisfaire la délégation de traitement attendue par la vue de base. Pour réaliser son travail, elle fait appel à deux autres méthodes qu'elle n'implémente pas elle-même : `get_context_data()` et `render_to_response()`. Ces deux méthodes sont respectivement apportées par les greffons `ContextMixin` et `TemplateResponseMixin`.

Cette répartition révèle les avantages donnés à un greffon : a) il peut être réutilisé pour apporter son concours à toute autre vue ; b) il peut être hérité par un autre greffon. Si on prend l'exemple du greffon `TemplateResponseMixin`, il sert à la fois :

– Pour bâtir une autre vue :

```
class FormView(TemplateResponseMixin, BaseFormView):
```

– Comme parent pour un autre greffon :

```
class MultipleObjectTemplateResponseMixin(TemplateResponseMixin):
```

En l'occurrence, par rapport aux deux méthodes apportées par la classe parente, la classe fille se contente d'en surcharger une, pour réaliser un complément en s'appuyant sur le travail déjà assuré par la méthode parente, avec un appel via `super()`.

En synthèse, construire sa vue est souvent une savante composition : il est bon de piocher dans les vues et greffons intégrés les éléments intéressants pour s'en servir comme base et ainsi éviter les redites, et en même temps il faut bien apporter des compléments métier à l'aide de personnalisations et de greffons à soi.

© Editions ENI - All rights reserved

4. Données de contexte

Dans le processus de rendu d'une page, l'objet des données de contexte est un support de passation de données : le moteur de gabarit instancie la page en remplaçant les variables mentionnées dans le gabarit par les valeurs piochées dans l'objet des données de contexte, lui-même ayant été alimenté au préalable par la vue.

Dans les usages ordinaires des vues, un objet des données de contexte est un simple dictionnaire primitif et il est désigné dans la documentation par le mot anglais `context`, avec un caractère minuscule. Il ne faut pas le confondre avec le mot `Context`, écrit avec un caractère majuscule, qui désigne une classe du module `django.template`. Cette classe et sa sous-classe `RequestContext` sont des objets employés au niveau des interfaces de plus bas niveau, avec les moteurs de rendu.

Si on reprend l'exemple précédent de `TemplateView`, l'objet des données de contexte est initialisé avec les éventuels paramètres extraits de l'URL. Son enrichissement est ensuite sous-traité au greffon `ContextMixin` par sa méthode `get_context_data()`. Celui-ci fait deux compléments : il ajoute l'objet vue sous la clé `view` et il actualise le dictionnaire avec le contenu de l'éventuel paramètre `extra_context` passé dans `as_view()`.

Un exemple va illustrer ces capacités. Pour éviter de modifier la configuration du site, les opérations font se faire manuellement par simulation.

Une configuration de routage est préparée. Une seule URL est suffisante et elle est porteuse d'un paramètre, appelé `id`. Le paramètre `extra_context` apporte également une donnée de contexte :

```
>>> from django.urls import path
>>> from django.views.generic import TemplateView
>>>
>>> urlpatterns = [
...     path('démo/<int:id>/', TemplateView.as_view(
...         template_name='index.html',
...         extra_context={'extra': 'plus'}
...     ))
... ]
```

Parmi les utilitaires en rapport avec le routage et les URL, la fonction `resolve()` permet de faire le lien entre un chemin URL et la vue en charge de son traitement :

```
>>> from django.urls import resolve
>>>
>>> resolver_match = resolve('/démo/12/',
...     urlconf=tuple(urlpatterns))
```

Habituellement, on ne se préoccupe pas de fournir le paramètre `urlconf` à cette fonction, car sa valeur par défaut est obtenue de la configuration du site. Cela est fait ici pour imposer le routage particulier préparé auparavant. Plusieurs formats sont admis pour le paramètre. Le type `list` ne convient pas ; par contre, son équivalent en `tuple` est supporté.

Un objet de classe `ResolverMatch` présente un ensemble d'attributs, mais la récupération de ceux les plus utiles est facilitée par une affectation à un 3-tuple :

```
>>> print(resolver_match)
ResolverMatch(func=django.views.generic.base.TemplateView,
args=(), kwargs={'id': 12}, url_name=None, app_names=[],
namespaces=[])
>>> func, args, kwargs = resolver_match
```

L'élément `func` est la fonction rendue par `as_view()` dans l'usage ordinaire. On peut le mettre en évidence en imitant le traitement d'une requête, comme si elle venait d'un navigateur. Pour cela, il faut tricher un peu et construire un objet requête de toutes pièces. Heureusement, il suffit de positionner son attribut `method` pour le rendre utilisable dans cette démonstration :

```
>>> from django.http.request import HttpRequest
>>>
>>> request = HttpRequest()
>>> request.method = 'GET'
```

© Editions ENI - All rights reserved

À l'aide de tous ces éléments, la fonction de vue peut être exécutée :

```
>>> response = func(request, *args, **kwargs)
>>> print(response)
<TemplateResponse status_code=200, "text/html; charset=utf-8">
>>> response.render()
<TemplateResponse status_code=200, "text/html; charset=utf-8">
>>> response.content
b'<html>\nBienvenue\n</html>'
```

Il n'est pas possible d'accéder plus finement aux attributs de l'instance de `TemplateView` mis en jeu dans ces essais. Pour contourner le problème et aller au bout des observations, on peut reproduire une partie des actions internes, en commençant par la construction de l'instance :

```
>>> view = TemplateView(
...     template_name='index.html',
...     extra_context={'extra': 'plus'})
```

Les paramètres à donner sont ceux donnés à l'appel à `as_view()`.

L'objet `view` passe ensuite par des étapes qu'il n'est pas nécessaire d'appliquer ici, car le point intéressant dans la démonstration est la constitution du contexte. Celui-ci est obtenu par l'appel suivant, avec les paramètres conservés de la résolution de l'URL pour coller fidèlement à l'implémentation réelle :

```
>>> context = view.get_context_data(**kwargs)
>>> context
{'id': 12,
 'view': <django.views.generic.base.TemplateView object at 0[...]>,
 'extra': 'plus'}
```

Preuve est faite que le contexte contient bien les paramètres capturés de l'URL (venus avec `**kwargs`), et les ajouts de `ContextMixin`, à savoir : la clé `view` et les éléments de `extra_context`.

5. Processeurs de contexte

Les processeurs de contexte ont été effleurés dans le chapitre Création de site, juste à l'occasion de l'exploration de la configuration, et il n'était question à ce stade que des processeurs intégrés disponibles.

Le rôle de ces composants va être illustré avec l'écriture d'un processeur personnalisé pour l'application.

Un processeur de contexte est une entité appelable destinée à alimenter l'objet de données de contexte. Il revêt un caractère systématique par son activation au niveau de la configuration et sa sollicitation à chaque rendu d'un gabarit.

Son contrat d'interface est de recevoir un objet requête et de retourner un dictionnaire, éventuellement vide, d'entrées qui seront mises à disposition dans le contexte. Traditionnellement, un processeur s'écrit sous la forme d'une fonction, au plus simple.

Pour une application de messagerie, il est habituel d'informer spontanément l'utilisateur du nombre de ses messages non lus et cette information a des chances d'être présentée sur toutes les pages, par exemple dans un bandeau en en-tête. Plutôt que de devoir répéter l'ajout au contexte de cette valeur pour chacune des vues, il est plus efficace de mettre en place un processeur de contexte qui fera le travail une fois pour toutes.

Le composant peut être logé n'importe où, mais il est conventionnel d'employer un module de nom `context_processors.py` sous la racine de l'application.

▶ Créez le fichier des processeurs de contexte :

`mysite\messenger\context_processors.py`

```
from .models import Message


def inbox(request):
    if request.user.is_authenticated:
        return {'messenger_unread_count': Message.objects.filter(
                recipient=request.user,
                read_at__isnull=True,
                recipient_deleted_at__isnull=True).count()
        }
```

© Editions ENI - All rights reserved

```
    else:
        return {}
```

Dès l'entrée, un test s'assure qu'on a affaire à un utilisateur connu. En effet, l'information recherchée n'a pas de sens pour un utilisateur anonyme et dans ce cas il est préférable de rendre un objet vide plutôt que de renvoyer l'information valorisée à la valeur zéro : ultérieurement, notamment au moment du rendu de la page, un vide permettra une distinction par rapport à un zéro, nombre légitime dans l'autre situation.

Pour obtenir la quantité de messages non lus dans la boîte d'arrivée, il n'est retenu que les messages qui satisfont à tous ces critères : l'utilisateur en est le destinataire, la date de lecture n'est pas positionnée, l'utilisateur n'a pas effacé le message.

- Pour qu'il produise son effet, déclarez ce processeur de contexte dans la configuration :

`mysite\settings.py`

```
[...]
TEMPLATES = [
    [...]
    'context_processors': [
      # 'django.template.context_processors.debug',
      # 'django.template.context_processors.request',
      'django.contrib.auth.context_processors.auth',
      # 'django.contrib.messages.context_processors.messages',
      'mysite.messenger.context_processors.inbox',
    ],
[...]
```

L'ordre de classement des processeurs n'est pas aussi essentiel qu'on a pu le voir avec les intergiciels, puisque chacun d'eux ne fait qu'enrichir un conteneur, a priori indépendamment des autres intervenants.

L'éventuel souci pourrait venir de l'emploi d'un même nom dans plusieurs maillons. Concernant les processeurs de contexte entre eux, chacun est censé utiliser des noms uniques, mais si ce n'est pas le cas, le dernier de la liste l'emporte. S'agissant d'une variable positionnée à la fois par la vue et par un processeur de contexte, il est donné volontairement la priorité à la valeur établie par la vue. Par exemple, dans une vue, il serait mal choisi de positionner une entrée avec le nom `user` ou `perms` lorsque le processeur de contexte de `auth` est activé, car celui-ci renseigne précisément des entrées ainsi dénommées et elles seraient perdues, à moins que l'intention soit d'imposer exceptionnellement ses valeurs. Un autre exemple encore plus pertinent dans le cadre de cette application de messagerie : si on met en fonctionnement le processeur de contexte de `contrib.messages`, celui-ci s'accapare l'entrée de contexte dénommée `messages`, qu'il aurait été pourtant bien intéressant d'employer. Les significations sont différentes puisqu'il s'agit de textes gradués en gravité à restituer à l'utilisateur, mais il n'en reste pas moins que le nom de variable est pris. Pour éviter, ou au moins largement limiter, le risque de conflits de nom avec d'autres processeurs de contexte, présents ou à venir, une solution simple est de préfixer ses variables. Ceci est donc fait avec le préfixe `messenger_`.

La mise en œuvre des notions d'authentification dans le processeur de contexte amène à devoir mettre en service l'intergiciel `AuthenticationMiddleware` et par dépendance l'intergiciel `SessionMiddleware`, si ce n'était pas déjà le cas, dans la configuration :

`mysite\settings.py`

```
[...]
MIDDLEWARE = [
    # [...]
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    # [...]
]
[...]
```

© Editions ENI - All rights reserved

À nouveau, pour réaliser une expérimentation par un exemple, les opérations vont se faire manuellement, en reproduisant le minimum nécessaire pris de l'implémentation de l'infrastructure logicielle.

Remarque

L'interpréteur devant être renouvelé à chaque actualisation des sources et de la configuration, on a avantage à conserver la saisie des commandes dans un éditeur à côté, de façon à rejouer des blocs de commandes par des copier/coller.

Tout d'abord, un objet de données de contexte est construit, comme précédemment, mais au plus simple, c'est-à-dire sans paramètre :

```
D:\dj>py manage.py shell
>>> from django.views.generic import TemplateView
>>>
>>> view = TemplateView(template_name='index.html')
>>> context = view.get_context_data()
>>> context
{'view': <django.views.generic.base.TemplateView object at 0[...]>}
```

Un objet requête est nécessaire. Deux instances sont préparées de façon à évaluer les deux situations : utilisateur anonyme et utilisateur authentifié. Le minimum suffisant pour utiliser la requête par la suite est de lui affecter un objet utilisateur. Pour une requête ordinaire, en provenance d'un navigateur, cette affectation est faite par l'intergiciel `AuthenticationMiddleware` :

```
>>> from django.http.request import HttpRequest
>>> from django.contrib.auth.models import AnonymousUser, User
>>>
>>> request_anon = HttpRequest()
>>> request_auth = HttpRequest()
>>> request_anon.user = AnonymousUser()
>>> request_auth.user = User.objects.get(pk=2)
```

La fonction non publique `make_context()` permet de construire un objet `RequestContext` à partir des données de contexte et d'une requête :

```
>>> from django.template.context import make_context
>>>
>>> req_ctx_anon = make_context(context, request_anon)
>>> req_ctx_auth = make_context(context, request_auth)
```

L'entrée en jeu des processeurs de contexte ayant lieu au cours du rendu d'un gabarit, il faut avoir en main une instance de gabarit :

```
>>> from django.template.loader import get_template
>>>
>>> tpl = get_template('index.html')
>>> tpl
<django.template.backends.django.Template object at 0x000002C[...]>
>>> tpl.template
<django.template.base.Template object at 0x000002C480749828>
```

Django a évolué vers le support de plus d'un moteur de rendu de gabarit, en complément à son moteur historique, d'où l'introduction d'un module `backends`, sous lequel on trouve à nouveau une branche `django` et sa classe `Template`, qui elle-même détient une référence à un autre `Template`, de niveau plus général. Attention donc à savoir quelle nature d'objet gabarit on doit manipuler dans les reproductions d'exécution.

Maintenant que tous les éléments sont constitués, il reste à faire leur assemblage. L'emploi des processeurs de contexte a une particularité : ce qu'ils fournissent est éphémère dans le contexte. En effet, un gestionnaire de contexte, au sens pur Python `contextlib.contextmanager`, est mis en œuvre par la méthode `bind_template()`. Les variables en provenance des processeurs de contexte ne sont donc présentes en contexte que le temps de l'exécution du corps d'une instruction `with`.

L'affichage du contexte, hors et dans un `with`, le met en évidence.

Une fois préparée, une instance de `RequestContext` ressemble à ceci :

```
>>> req_ctx_anon
[{'True': True, 'False': False, 'None': None},
 {},
 {},
 {'view': <django.views.generic.base.TemplateView object at[...]>}]
```

Il se trouve que la structure est conçue en forme de liste de dictionnaires, mais c'est un détail d'implémentation interne, totalement transparent pour le développeur.

Le premier dictionnaire est un ensemble de trois variables intégrées, offertes par défaut pour tout contexte car très couramment utilisées. Les deux dictionnaires suivants sont des emplacements réservés pour remplissage ultérieur.

© Editions ENI - All rights reserved

Le dernier dictionnaire est celui des données de contexte, fourni à la construction.

Lorsqu'elle est réalisée au sein de l'exécution du gestionnaire de contexte, l'observation montre un apport :

```
>>> with req_ctx_anon.bind_template(tpl.template):
...     print(req_ctx_anon)
...
[{'True': True, 'False': False, 'None': None},
 {'csrf_token':
    <SimpleLazyObject: <function csrf.<locals>._get_val at[...]>>,
  'user': <django[...].auth.models.AnonymousUser object at  0x[...]>,
  'perms': <[...]auth.context_processors.PermWrapper object a[...]>},
 {},
 {'view': <django.views.generic.base.TemplateView object at[...]>}]
```

Le deuxième dictionnaire est un emplacement dédié à recevoir les variables fournies par les processeurs de contexte :

– `django.template.context_processors.csrf` :

Ce processeur n'est pourtant pas mentionné dans la configuration. C'est un cas spécial : en raison de son rôle primordial en matière de sécurité, il est imposé en interne et il n'y a pas d'option pour y échapper. Il apporte la variable `csrf_token`.

– `django.contrib.auth.context_processors.auth`

Ce processeur contribue au contexte avec les variables `user` et `perms`.

Le processeur `inbox` est bien sollicité également, mais il ne fait aucun apport puisqu'il s'agit d'un utilisateur anonyme. La situation se différencie dans la même observation avec un utilisateur authentifié :

```
>>> with req_ctx_auth.bind_template(tpl.template):
...     print(req_ctx_auth)
...
[{'True': True, 'False': False, 'None': None},
 {'csrf_token':
   <SimpleLazyObject: <function csrf.<locals>._get_val at [...]>>,
  'user': <User: foo>,
  'perms': <[...]auth.context_processors.PermWrapper object at[...]>,
  'messenger_unread_count': 3},
 {},
 {'view': <django.views.generic.base.TemplateView object at[...]>}]
```

```
>>> req_ctx_auth
[{'True': True, 'False': False, 'None': None},
 {},
 {},
 {'view': <django.views.generic.base.TemplateView object at[...]>}]
```

La variable `user` traduit un utilisateur authentifié, mais surtout on voit cette fois la variable `messenger_unread_count`. Le nouvel affichage du contexte immédiatement après sortie du gestionnaire de contexte montre que le conteneur des variables des processeurs est repassé à une entité vide.

Il a été écrit plus haut que les données de contexte, situées dans le dernier dictionnaire, ont priorité sur les variables établies par les processeurs de contexte, or on constate que ces dernières apparaissent en amont dans la liste. Tout reste cohérent quand on sait que l'algorithme de résolution des variables parcourt la liste à l'envers.

Optimisation

Dans son écriture actuelle, le processeur de contexte `inbox` présente potentiellement un inconvénient : il mobilise une requête SQL à chaque traitement d'une requête HTTP. Ceci représente un coût en ressources, qu'on peut considérer faible, mais qu'il n'y a pas de raison de négliger.

Si vraiment toutes les pages font usage de la variable, la consommation se justifie. Par contre, il est plus probable que l'information ne soit pas nécessaire pour toutes les pages. Or, un processeur de contexte déclaré dans la configuration ne peut pas être écarté, à la demande, par ou pour telle ou telle vue.

Pour une mise en évidence, il suffit d'activer les traces des accès SQL, ainsi qu'il est décrit dans le chapitre Traces et journalisation. Sur la dernière manipulation, on obtient cette sortie étendue :

```
>>> with req_ctx_auth.bind_template(tpl.template):
...     print(req_ctx_auth)
...
20[...]_django.db.backends_DEBUG - (0.005) SELECT COUNT(*) AS
 "__count" FROM "messenger_message" WHERE
 ("messenger_message"."read_at" IS NULL AND
 "messenger_message"."recipient_id" = 2 AND
 "messenger_message"."recipient_deleted_at" IS NULL); args=(2,)
[{'True': True, 'False': False, 'None': None},
 {'csrf_token': [...], 'user': <User: foo>, 'perms': [...],
```

© Editions ENI - All rights reserved

```
    'messenger_unread_count': 3},
  [...]
```

D'une manière générale, une réévaluation de la situation pourrait amener à la conclusion que l'application ne se prête finalement pas à l'emploi d'un processeur de contexte et qu'il vaudrait mieux en réalité que l'alimentation du contexte soit faite individuellement par les seules vues qui en ont besoin. Une implémentation sous forme de greffon est un moyen pour éviter de tomber dans la duplication de code.

Une autre voie consiste à définir un moteur de rendu de gabarit alternatif, particulièrement allégé en processeurs de contexte, voire sans aucun processeur. En voici un exemple :

mysite\settings.py

```
[...]
TEMPLATES = [
    {
        'BACKEND':'[...]template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(_SITE_ROOT, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # 'django.template.context_processors.debug',
                # 'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                # '[...]trib.messages.context_processors.messages',
                'mysite.messenger.context_processors.inbox',
            ],
        },
    },
    {
        'NAME': 'django_light',
        'BACKEND': '[...]template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(_SITE_ROOT, 'templates')],
        'APP_DIRS': True,
    },
]
```

L'autre moteur déclaré est une copie de celui par défaut sauf qu'il n'a pas de processeurs de contexte. Chaque moteur doit avoir un nom unique. S'il n'est pas précisé, comme c'est le cas pour le premier, la valeur par défaut est tirée du nom de module de la classe déclarée en `BACKEND`. Ici, la classe `DjangoTemplates` est dans le module `django.template.backends.django`, d'où le nom par défaut `django`. Un nom tel que `django_light` est attribué au second moteur pour faire la distinction avec le premier et être significatif de sa particularité.

L'interpréteur doit être renouvelé pour disposer de la nouvelle configuration. Les commandes suivantes permettent de se replacer dans la continuité des opérations précédentes :

```
D:\dj>py manage.py shell
>>> from django.views.generic import TemplateView
>>> view = TemplateView(template_name='index.html')
>>> context = view.get_context_data()
>>>
>>> from django.http.request import HttpRequest
>>> request_auth = HttpRequest()
>>> from django.contrib.auth.models import User
>>> request_auth.user = User.objects.get(pk=2)
>>>
>>> from django.template.context import make_context
>>> req_ctx_auth = make_context(context, request_auth)
```

La nouveauté vient maintenant, dans l'obtention du gabarit où le paramètre optionnel `using` permet de préciser le moteur exclusif à utiliser, en fournissant son nom :

```
>>> from django.template.loader import get_template
>>>
>>> tpl_light = get_template('index.html', using='django_light')
```

Lorsque le paramètre est omis, les moteurs sont sollicités successivement dans l'ordre de la liste de configuration pour une tentative de chargement et le premier qui répond positivement est retenu.

© Editions ENI - All rights reserved

Avec cette configuration adaptée, l'observation devient :

```
>>> with req_ctx_auth.bind_template(tpl_light.template):
...     print(req_ctx_auth)
...
[{'True': True, 'False': False, 'None': None},
 {'csrf_token':
   <SimpleLazyObject: <function csrf.<locals>._get_val at[...]>>},
 {},
 {'view': <django.views.generic.base.TemplateView object at[...]>}]
```

On constate bien la seule présence de la variable apportée par le processeur de contexte `csrf`, inévitable comme il a été dit juste auparavant.

Pour une mise en œuvre ordinaire, le moteur à employer peut s'indiquer comme on le fait pour le gabarit, au moment de l'obtention de la vue. En voici un exemple au niveau du routage :

`mysite\urls.py`

```
[...]
urlpatterns = [
    [...]
    path('light', TemplateView.as_view(
        template_name='index.html',
        template_engine='django_light')),
    [...]
```

Une autre solution au problème, certainement plus légère que l'emploi d'un moteur additionnel, se base sur un constat simple : le résultat de la requête SQL n'est pas réellement utile au sein du processeur de contexte, mais plutôt à celui qui va par la suite exploiter la variable dédiée au résultat.

On en déduit qu'il est judicieux de différer son exécution effective jusqu'au moment où l'information sera concrètement réclamée. Des utilitaires existent pour faciliter l'implémentation d'actions différées, dont l'objet `SimpleLazyObject`, non documenté, auquel on donne une entité appelable en charge de délivrer l'information finale.

Modifiez le processeur de contexte pour qu'il se présente ainsi :

mysite\messenger\context_processors.py

```
from django.utils.functional import SimpleLazyObject
[...]
def inbox(request):
    if request.user.is_authenticated:
        return {'messenger_unread_count': SimpleLazyObject(
            lambda: Message.objects.filter(
                [...]
        )}
    [...]
```

Une fonction anonyme suffit pour respecter le contrat et aucun paramètre n'est passé. L'appel ne se fera éventuellement que plus tard, si l'objet enveloppant doit être évalué. Si personne ne demande la valeur de la variable de contexte, le gestionnaire de modèle n'est pas sollicité et l'accès superflu à la base de données épargné.

Sur le moteur de rendu de gabarits avec les processeurs de contexte, l'observation révèle maintenant :

```
>>> tpl = get_template('index.html')
>>> with req_ctx_auth.bind_template(tpl.template):
...     print(req_ctx_auth)
...     print(req_ctx_auth.get('messenger_unread_count'))
...
[{'True': True, 'False': False, 'None': None},
 {'csrf_token': [...], 'user': <User: foo>, 'perms': [...],
  'messenger_unread_count':
   <SimpleLazyObject: <function inbox.<locals>.<lambda>a[...]>>},
 {},
 {'view': [...]}]
20[...]_django.db.backends_DEBUG - (0.035) SELECT COUNT(*) AS [...]
3
```

Les affichages à la console montrent que le contexte détient un objet différé, avec une fonction sans nom (lambda) située sous l'objet `inbox`. Surtout, à en juger la séquence, l'endroit où la trace SQL apparaît confirme que la requête SQL n'est soumise qu'au moment de la lecture de la variable.

Parmi les utilitaires pour actions différées, il existe aussi la fonction `lazy(func, *resultclasses)`, tout autant non documentée, qui semble suffire au besoin et être plus légère.

© Editions ENI - All rights reserved

▶Modifiez le processeur de contexte pour qu'il se présente ainsi :

mysite\messenger\context_processors.py

```
from django.utils.functional import lazy
[...]
def inbox(request):
    if request.user.is_authenticated:
        return {'messenger_unread_count': lazy(
            lambda: Message.objects.filter(
                [...],
            int
        )}
    [...]
```

Le paramètre `func` est la même fonction anonyme que dans la version précédente. Pour les paramètres de typage du résultat, il est cité le type entier `int` puisqu'un nombre est rendu.

L'observation se fait un peu différemment :

```
>>> with req_ctx_auth.bind_template(tpl.template):
...     myvar = req_ctx_auth.get('messenger_unread_count')
...     print(myvar)
...     print(myvar())
...
<function inbox.<locals>.<lambda> at 0x0000012ECDBBC1E0>
20[...]_django.db.backends_DEBUG - (0.030) SELECT COUNT(*) AS [...]
3
```

La variable du contexte n'héberge plus une instance d'objet comme dans la version précédente, mais une entité appelable, ainsi rendue par la fonction `lazy()`, qui agit en tant qu'enveloppe. L'évaluation de la variable doit donc être provoquée non plus par une simple lecture, mais par un appel.

Ceci ne pose aucun souci quand la variable est mentionnée dans un gabarit, car la distinction n'apparaît pas dans la syntaxe et l'algorithme de résolution des variables s'occupe de savoir s'il a affaire à une entité appelable ou d'un autre genre.

6. Requêtes AJAX

L'infrastructure logicielle traite les requêtes de nature AJAX avec les mêmes mécanismes qu'une requête ordinaire. La distinction est facilitée par une méthode sur l'objet requête, pour rendre un simple booléen. Le traitement de la requête se fait avec une vue, comme d'habitude, sauf que très souvent la réponse est au format JSON plutôt que HTML.

Pour une application de messagerie, on peut imaginer un besoin de maintenir actuelle une pièce d'information dans la page du navigateur, autrement que par des rechargements complets de la page, automatiquement ou à l'initiative de l'utilisateur. Un exemple typique est un compteur de messages non lus naturellement à jour. Diverses techniques, plus ou moins réactives et laborieuses, existent pour remplir la fonctionnalité. Par nature, la messagerie n'a pas vocation à être hautement réactive, donc une technique par interrogation régulière à basse fréquence est suffisante dans le cas présent.

Supposons que la page soit pourvue d'un code pour interroger en AJAX le serveur toutes les N minutes afin d'obtenir l'état actuel du nombre de messages non lus dans la boîte d'arrivée de l'utilisateur.

Pour commencer, une route est établie :

`mysite\messenger\urls.py`

```
[...]
from .views import index, IndexView
from .views import AjaxUnreadCountView  # démo AJAX
[...]
urlpatterns = [
    [...]
    # démo AJAX
    path('unread-count/', AjaxUnreadCountView.as_view()),
    [...]
```

Une ligne isolée est utilisée pour l'import de la vue, pour être plus facilement retirée ou désactivée par une mise en commentaire, si cette route n'est pas conservée après la démonstration.

Le nom de la vue est volontairement préfixé avec le motif `Ajax`, par commodité de lecture, pour bien rappeler la nature qui lui est assignée.

© Editions ENI - All rights reserved

Ensuite, une vue est construite :

mysite\messenger\views.py

```
[...]
from django.http import JsonResponse
[...]
from django.views import View
[...]
class AjaxUnreadCountView(View):
    http_method_names = ['get']

    def get(self, request, *args, **kwargs):
        if request.user.is_authenticated:
            data = {'unread_count': Message.objects.filter(
                recipient=request.user,
                read_at__isnull=True,
                recipient_deleted_at__isnull=True).count()}
        else:
            data = {}
        return JsonResponse(data)
```

La vue hérite du socle de base `View`, il n'existe pas de vue générique particulière à la situation. Il est plus propre de restreindre, avec l'attribut `http_method_names`, le jeu de verbes admis. S'il est absent, la différence n'est pas grande puisque la même gestion d'erreur est appelée pour les verbes non supportés et pour ceux dont la méthode correspondante n'est pas implémentée, mais d'une part la lecture est plus confortable en étant explicite, et d'autre part cela écarte les verbes `HEAD` et `OPTIONS` supportés par défaut par `View`, sans nécessité ici.

La classe `JsonResponse` est une variante de `HttpResponse` adaptée au cas de renvoi d'un contenu au format JSON. C'est donc un choix privilégié pour une vue AJAX. Si le demandeur n'est pas connu, par nature, mais aussi si son authentification a expiré depuis son dernier accès, un objet vide est retourné. Une réponse positive reste préférée à une réponse d'erreur, car cela devrait être plus facile ensuite au niveau du navigateur de distinguer les situations, attendues ou réellement anormales. Il n'y a pas de bénéfice à fournir une chaîne vide ou `None` plutôt qu'un dictionnaire, ni en volume ni en effort de traitement à l'arrivée; au contraire d'ailleurs, puisqu'une contrainte historique de sécurité, qui n'est plus d'actualité, oblige à passer un paramètre dérogatoire (`safe`) pour fournir autre chose qu'un dictionnaire.

Pour l'expérimentation, il faut renouveler l'interpréteur pour disposer des nouveautés et lui passer les commandes suivantes pour reconstruire les objets de base des manipulations :

```
D:\dj>py manage.py shell
>>> from django.http.request import HttpRequest
>>> from django.contrib.auth.models import AnonymousUser, User
>>>
>>> request_anon = HttpRequest()
>>> request_auth = HttpRequest()
>>> request_anon.user = AnonymousUser()
>>> request_auth.user = User.objects.get(pk=2)
>>> request_anon.method = 'GET'
>>> request_auth.method = 'GET'
```

Pour aller au plus court, inutile de passer par la résolution d'URL, car la vue peut être acquise directement :

```
>>> from mysite.messenger.views import AjaxUnreadCountView
>>>
>>> view = AjaxUnreadCountView.as_view()
```

Une requête de la part d'un visiteur anonyme donne les traces suivantes :

```
>>> resp_anon = view(request_anon)
>>> resp_anon
<JsonResponse status_code=200, "application/json">
>>> resp_anon.content
b'{}'
```

La représentation de l'objet de réponse confirme qu'il est conforme au résultat attendu. L'attribut `content` montre le contenu de la réponse, un objet JSON vide comme attendu.

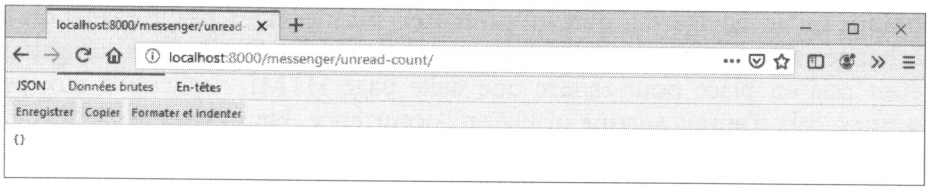
En revanche, la requête équivalente de la part d'un utilisateur connu donne un résultat valorisé :

```
>>> resp_auth = view(request_auth)
>>> resp_auth.content
b'{"unread_count": 3}'
```

© Editions ENI - All rights reserved

6.1 Restriction stricte au mode AJAX

Dans les essais précédents, à aucun moment le fait d'être une requête de nature AJAX n'apparaît, à part dans le nom de la vue, ce qui ne change rien au constat. On peut le vérifier par un accès ordinaire par navigateur :



Il est tout à fait possible de rester dans ce mode de fonctionnement. Si toutefois on souhaite imposer que l'accès à la vue soit limité au mode AJAX, le contrôle de cette caractéristique est facilité par la disponibilité de la méthode `is_ajax()` sur l'objet requête.

Modifiez la vue pour ajouter le test de contrainte :

`mysite\messenger\views.py`

```
[...]
from django.http import JsonResponse, HttpResponseForbidden
[...]
class AjaxUnreadCountView(View):
    [...]
    def get(self, request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseForbidden()
        [...]
```

Pour un refus lié à des notions de permissions, il est fréquent de voir une implémentation de vue dans le style :

```
from django.core.exceptions import PermissionDenied
[...]
        if not some_condition:
            raise PermissionDenied
```

En effet, cette exception est interceptée par l'infrastructure logicielle et traduite en une vue intégrée du type 403 *Forbidden* (comme il en existe pour les cas 400 *Bad Request*, 404 *Not Found*, et 500 *Internal Server Error*).

Le gabarit de cette vue est personnalisable, ce qui facilite une gestion unifiée des erreurs de page sur la globalité du site. En absence de fourniture d'un gabarit personnalisé, la charge de la réponse renvoyée au navigateur est :

```
<h1>403 Forbidden</h1>
```

Ce minimum peut encore avoir un sens face à un utilisateur et l'interface graphique qu'un navigateur doit lui présenter, mais ce n'est pas le cas ici où les échanges sont purement techniques, entre machines. D'ailleurs, si un gabarit était mis en place pour rendre une belle page HTML explicative aux personnes, cela n'aurait aucune utilité en l'occurrence. Pour éviter le gâchis de la transmission d'un corps inutile, il est préférable de garder la main et de retourner une réponse de même type, mais parfaitement vide.

Désormais, l'accès tel qu'il était fait auparavant est refusé :

```
>>> request_auth.is_ajax()
False
>>> resp_auth = view(request_auth)
>>> resp_auth
<HttpResponseForbidden status_code=403, "text/html; charset=u[...]>
```

Pour prouver que le code fonctionne aussi en conditions favorables, il va suffire de forger la requête pour qu'elle paraisse comme en fonctionnement réel :

```
>>> request_auth.META['HTTP_X_REQUESTED_WITH'] = 'XMLHttpRequest'
```

L'accès est alors à nouveau permis :

```
>>> request_auth.is_ajax()
True
>>> resp_auth = view(request_auth)
>>> resp_auth
<JsonResponse status_code=200, "application/json">
>>> resp_auth.content
b'{"unread_count": 3}'
```

Si on imagine avoir d'autres vues bâties sur le même schéma, c'est-à-dire à l'accès contraint au mode AJAX, dupliquer ce code de condition dans chacune d'elles ne serait pas conforme à un haut niveau de qualité. C'est une excellente occasion de factoriser ce code dans un greffon, avec des nuances que les sous-sections suivantes vont décrire.

© Editions ENI - All rights reserved

6.1.1 Par greffon seul

◘Modifiez la vue pour introduire un greffon et y migrer le test :

`mysite\messenger\views.py`

```
[...]
class AjaxMixin:
    def dispatch(self, request, *args, **kwargs):
        if not request.is_ajax():
            return HttpResponseForbidden()
        return super().dispatch(request, *args, **kwargs)


class AjaxUnreadCountView(AjaxMixin, View):
    [...]
    def get(self, request, *args, **kwargs):
        if request.user.is_authenticated:
        [...]
```

Cette écriture apporte de multiples bénéfices. La lecture et par conséquent la compréhension fonctionnelle sont améliorées par le découplage, c'est-à-dire l'isolation claire de la contrainte d'accès par rapport à la vue. Celle-ci est recentrée sur l'essence de son travail de vue, à savoir délivrer un rendu ou plus généralement une charge utile.

Le contrôle, écrit qu'une seule fois, est réutilisable à volonté dans de multiples autres vues, par simple héritage. Même si, en l'occurrence, seule la méthode `get()` intervient, il est avantageux de loger le contrôle dans la méthode `dispatch()`, car il s'appliquera ainsi à n'importe quel verbe potentiel.

6.1.2 Par greffon et décorateur

La factorisation peut être poussée encore plus loin avec l'emploi d'un décorateur. En effet, même placé dans un greffon, le contrôle reste quelque chose qui empêche plutôt qu'une fonctionnalité positive.

Ce genre de rôle se prête bien à une implémentation sous forme de décorateur. Un exemple de décorateur parmi les plus utilisés est `login_required` du paquet `auth`, pour protéger l'accès à des vues réservées aux personnes inscrites au site. On trouve aussi `permission_required` dans ce paquet `auth`. Ces références incitent à l'écriture d'un décorateur, qu'on va appeler `ajax_required`, par franche analogie.

Il faut admettre que l'écriture d'un décorateur est un peu complexe, par sa nature d'enveloppe, pour intercaler des traitements autour de l'exécution éventuelle de l'objet enveloppé. C'est pourquoi il ne faut pas hésiter à reprendre le squelette d'un décorateur existant. De plus, l'affaire se complique encore un peu lorsqu'il s'agit de mettre un décorateur sur une méthode de classe plutôt qu'une simple fonction. Heureusement, Django offre l'utilitaire `method_decorator` pour en faciliter l'écriture.

▶ Modifiez la vue pour introduire un décorateur et y migrer le test :

`mysite\messenger\views.py`

```
[...]
from functools import wraps
[...]
from django.utils.decorators import method_decorator
[...]
def ajax_required(func):
    @wraps(func)
    def wrapper(request, *args, **kwargs):
        if request.is_ajax():
            return func(request, *args, **kwargs)
        return HttpResponseForbidden()
    return wrapper


class AjaxMixin:
    @method_decorator(ajax_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
[...]
```

Pour la méthode `dispatch()`, il n'est plus nécessaire d'isoler le paramètre `request` de la collection `*args`. Dans le décorateur, le test est basculé en positif, car la logique y est plutôt tournée ainsi : si les conditions sont satisfaites, on aboutit à appeler la fonction enveloppée, sinon à la fin on laisse échouer avec un retour négatif.

Avec ce montage par décorateur, on pourrait avoir l'impression d'avoir complexifié l'écriture par rapport à la version précédente. Un bénéfice se retrouve pourtant si le décorateur peut être réutilisé ailleurs que dans une vue de forme de classe. C'est notamment le cas avec des vues ayant le même besoin de contrôle, mais codées sous forme de fonction.

© Editions ENI - All rights reserved

Pour illustrer ce propos, voici une vue fictive implémentée par une fonction, dont on veut l'usage restreint au mode AJAX :

```
@ajax_required
def ajax_whoami_view(request):
    return JsonResponse({'username': request.user.get_username()}
        if request.method == 'GET'
            and request.user.is_authenticated
        else {})
```

On a bien une fois de plus d'un côté la vue qui ne sait que rendre le résultat de son travail et d'un autre côté le code pour assurer le respect d'une contrainte, toujours écrit en un exemplaire.

Les observations suivantes montrent les trois retours possibles :

```
>>> from mysite.messenger.views import ajax_whoami_view
>>>
>>> resp_anon = ajax_whoami_view(request_anon)
>>> resp_anon
<HttpResponseForbidden status_code=403, "text/html charset=u[...]>
>>>
>>> request_anon.META['HTTP_X_REQUESTED_WITH'] = 'XMLHttpRequest'
>>> ajax_whoami_view(request_anon).content
b'{}'
>>>
>>> request_auth.META['HTTP_X_REQUESTED_WITH'] = 'XMLHttpRequest'
>>> ajax_whoami_view(request_auth).content
b'{"username": "foo"}'
```

7. Intégrations en modèles

Avec les sections précédentes, il est à remarquer que le besoin d'obtenir le compte des messages non lus est apparu à deux occasions : dans un processeur de contexte pour mettre le compteur à disposition des gabarits et dans une vue AJAX pour permettre à la page de s'actualiser. Dans le chapitre consacré aux pages et gabarits, le besoin apparaît une fois de plus. Dans tous les cas, la requête est construite de la même façon. Il n'est évidemment pas question de supporter cette duplication de code, un remaniement s'impose.

Le lieu préférentiel pour la centralisation d'un traitement portant sur les instances d'un modèle est le gestionnaire de ce modèle. Un gestionnaire personnalisé a déjà été préparé dans le chapitre consacré aux modèles, mais laissé vide jusqu'à présent.

▶Complétez le gestionnaire de modèle :

mysite\messenger\models.py

```
[...]
class MessageManager(models.Manager):
    def unread_count(self, user):
        return self.filter(
            recipient=user,
            read_at__isnull=True,
            recipient_deleted_at__isnull=True).count()
[...]
```

▶Remaniez les anciennes écritures pour exploiter cette nouvelle méthode :

mysite\messenger\context_processors.py

```
[...]
def inbox(request):
    if request.user.is_authenticated:
        return {'messenger_unread_count': lazy(
            lambda: Message.objects.unread_count(request.user),
            [...]
```

mysite\messenger\views.py

```
[...]
class AjaxUnreadCountView(AjaxMixin, View):
[...]
    def get(self, request, *args, **kwargs):
        return JsonResponse({'unread_count':
                Message.objects.unread_count(request.user)}
            if request.user.is_authenticated
            else {})
[...]
```

■Remarque

On a également profité de l'occasion pour adopter une rédaction plus compacte sous forme d'expression conditionnelle.

© Editions ENI - All rights reserved

8. Simulation d'authentification

Pour la suite des travaux, il devient trop difficile de rester non authentifié, car il faudra cibler les messages se rapportant à un utilisateur. Forger les requêtes comme on a pu le faire jusqu'à maintenant pour faire croire à un utilisateur authentifié devient trop laborieux. Par ailleurs, l'authentification est un domaine suffisamment vaste pour mériter d'être traité à part. Pour éluder le point dans l'immédiat, de simples émulations vont être mises en place, ce qui permet d'avancer en restant concentré sur le sujet du chapitre.

Il est préférable de ne pas polluer l'application `messenger` avec ces considérations de travail et donc on va plutôt tirer profit de la présence de l'application `main` pour lui faire héberger ces besoins auxiliaires.

Tout d'abord, un point d'intendance est à régler : l'application `auth` s'appuie sur l'application `sessions` pour faire une partie du travail et doit être activée dans la configuration si ce n'est pas déjà le cas.

▶ Activez la ligne pour `sessions` :

`mysite\settings.py`

```
INSTALLED_APPS = [
    [...]
    'django.contrib.contenttypes',  # dépendance de admin
    'django.contrib.sessions',  # opt, pour utiliser auth
    # 'django.contrib.messages',
    [...]
]
```

Lorsqu'on met une application en activité, il est banal d'oublier la possible présence de ses migrations à appliquer. Pourtant le serveur de développement le rappelle :

```
You have 1 unapplied migration(s). Your project may not work
properly until you apply the migrations for app(s): sessions.
Run 'python manage.py migrate' to apply them.
```

Mais la trace peut passer inaperçue, et lorsqu'on lance une action qui a besoin de l'application, cela mène à un échec avec ce genre de traces :

```
psycopg2.ProgrammingError: ERREUR:
 la relation «django_session» n'existe pas
LINE 1: SELECT (1) AS "a" FROM "django_session" WHERE
"django_sessio...
```

▶Réalisez les migrations, pour l'application nouvellement activée :

```
D:\dj>py manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, messenger, sessions
Running migrations:
 Applying sessions.0001_initial... OK
```

La migration initiale crée la table, dont l'éventuelle trace d'échec précédente révèle la nécessité.

Des vues pour actionner la connexion et la déconnexion peuvent maintenant être envisagées. S'agissant d'émulation, on ne va pas s'embarrasser de généricité ni de réalisme, et donc coder au plus simple et en dur. Ainsi, l'utilisateur est fixé dans l'implémentation, puisque le seul objectif est de monter une session authentifiée, au plus économique.

▶Complétez le routage :

`mysite\urls.py`

```
[...]
from mysite.main import views

urlpatterns = [
   [...]

   # émulations de vues d'authentification
   path('login/', views.emulated_login),
   path('logout/', views.emulated_logout),
]
```

© Editions ENI - All rights reserved

▶Ajoutez les vues :

`main\views.py`

```
from django.contrib.auth import login, logout
from django.contrib.auth.models import User
from django.http import HttpResponse
[...]

def emulated_login(request):
    login(request, User.objects.get(pk=2))
    return HttpResponse('logged in')

def emulated_logout(request):
    logout(request)
    return HttpResponse('logged out')
```

Avec ce simple code minimal, on pourra se placer en situation d'utilisateur connu ou anonyme, en pointant le navigateur sur l'une ou l'autre de ces adresses :

- `http://localhost:8000/login/` pour établir une session authentifiée.
- `http://localhost:8000/logout/` pour en sortir.

Les outils de développement du navigateur permettent de vérifier les conditions d'authentification. En particulier :

- La réponse à l'accès en connexion montre la pose d'un cookie :

```
Set-Cookie: sessionid=8iuvf2v3mcid4oj9zxtxkql1sabqvu4s;
expires=Fri, 15 Nov 20[...] 17:34:39 GMT; HttpOnly;
Max-Age=1209600; Path=/; SameSite=Lax
```

Par défaut, la durée de conservation d'une session est de quatorze jours.

- Le stockage montre en section cookies une entrée du genre :

```
sessionid :"8iuvf2v3mcid4oj9zxtxkql1sabqvu4s" [...]
```

- La réponse à l'accès en déconnexion montre le retrait du cookie :

```
Set-Cookie: sessionid=""; expires=Thu, 01 Jan 1970 00:00:00 GMT;
Max-Age=0; Path=/
```

9. Écritures des vues

Cette section va être consacrée à la construction des vues de l'application. Le travail de rendu effectif des pages est volontairement différé au chapitre suivant, relatif aux pages et gabarits.

Jusqu'à maintenant, le fichier de routage contient quelques routes, mais elles mènent vers une même vue `index` d'attente, à but démonstratif. Il est temps de les reprendre une à une pour leur donner leur vrai rôle.

9.1 Dossier d'arrivée

Modifiez le fichier de routage :

`mysite\messenger\urls.py`

```
[...]
from .views import index
from .views import (InboxView,
    IndexView,
)
[...]
urlpatterns = [
    [...]
    path('inbox/', InboxView.as_view(), name='inbox'),
    [...]
```

L'import de la vue `index` est isolé sur une ligne en prévision de sa future élimination, puisque les usages de cette vue transitoire vont être progressivement remplacés. L'instruction d'import des vues est préparée pour être répartie sur plusieurs lignes, qui vont se remplir au fil des adaptations.

Modifiez le fichier des vues pour créer cette nouvelle vue :

`mysite\messenger\views.py`

```
[...]
from django.contrib.auth.decorators import login_required
[...]
from django.views.generic import RedirectView, TemplateView
[...]
```

© Editions ENI - All rights reserved

```
class InboxView(TemplateView):
    http_method_names = ['get']
    template_name = 'messenger/inbox.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        msgs = Message.objects.filter(
            recipient=self.request.user,
            recipient_deleted_at__isnull=True)
        # 'messages' est déjà pris par contrib.messages
        context['msg_messages'] = msgs
        return context
[...]
```

La vue est bâtie sur la vue générique basique `TemplateView` en charge de rendre un gabarit, à l'aide d'informations piochées dans un contexte. Il suffit dans ce cas de surcharger la méthode `get_context_data()` permettant d'alimenter ce contexte. Le schéma classique est de s'intercaler, c'est-à-dire d'appeler la même méthode sur le parent pour récupérer un objet de contexte, vide ou pas, et de le compléter ou le modifier, avant de le rendre.

Seul le verbe `GET` est admis et le nom du gabarit est indiqué par l'attribut `template_name`.

La vue n'a pas de sens si on ne sait pas quel est l'utilisateur. Elle est donc protégée à l'aide du décorateur intégré `login_required`, attaché à la méthode `dispatch()` pour être applicable de façon générale à tout accès à la vue.

- Créez un gabarit temporaire par copie d'un existant :

```
D:\dj>cd mysite\messenger\templates\messenger
D:\dj\mysite\messenger\templates\me[...]>copy index.html inbox.html
        1 fichier(s) copié(s).
```

- Si un supplément de visibilité est souhaité, ajoutez-y cette ligne pour constater l'acquisition des messages (format brut, sans mise en forme) :

```
[...]
<div>{{ msg_messages }}</div>
[...]
```

- Vérifiez la conservation du fonctionnement du site, en accédant à la page `/messenger/inbox/`.

9.2 Dossier d'envoi

- Modifiez le fichier de routage :

`mysite\messenger\urls.py`

```
[...]
from .views import (InboxView, SentView,
[...]
urlpatterns = [
    [...]
    path('sent/', SentView.as_view(), name='sent'),
    [...]
```

- Modifiez le fichier des vues pour créer la nouvelle vue :

`mysite\messenger\views.py`

```
[...]
class SentView(TemplateView):
    http_method_names = ['get']
    template_name = 'messenger/sent.html'

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        msgs = Message.objects.filter(
            sender=self.request.user,
            sender_deleted_at__isnull=True)
        # 'messages' est déjà pris par contrib.messages
        context['msg_messages'] = msgs
        return context
[...]
```

- Créez un gabarit temporaire par copie du gabarit précédent :

```
D:\dj>cd mysite\messenger\templates\messenger
D:\dj\mysite\messenger\templates\me[...]>copy inbox.html sent.html
        1 fichier(s) copié(s).
```

© Editions ENI - All rights reserved

- Vérifiez la conservation du fonctionnement du site, en accédant à la page `/messenger/sent/`.

Fort bien, cette vue fonctionne tout autant que la précédente et il n'y a rien d'étonnant puisqu'elle lui ressemble tellement. Une telle similitude de code engendre irrésistiblement un intense désir de remaniement, objet de la section suivante.

9.3 Factorisation des vues de dossier

Les vues des deux dossiers obéissent à un même principe de traitement et ont donc naturellement beaucoup de code identique. Heureusement, il n'y a rien de tel qu'un greffon pour factoriser ces doublons.

- Créez un greffon et commencez à y migrer le code évident :

`mysite\messenger\views.py`

```
[...]
class FolderMixin:
    """Code commun aux dossiers."""
    http_method_names = ['get']

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
[...]
```

Il reste la méthode `get_context_data()`, non strictement identique, mais conçue sur le même schéma. Puisque ce sont seulement les critères de filtrage qui font la distinction, ils doivent être clairement isolés. Une façon basique et habituelle pour gérer ce cas est d'employer un attribut, comme cela est fait avec `template_name`, en imaginant un attribut `filters`.

Cette solution simple suppose que l'attribut soit statique, ce qui n'est pas le cas ici puisque l'un des critères a sa valeur liée à l'utilisateur, donc dynamique et issue de la requête. Plusieurs solutions permettant de résoudre le problème vont être exposées, selon un degré d'élaboration s'élevant progressivement.

Ci-dessous, leur implémentation sur une des deux vues, `InboxView`, suffit à la démonstration.

La généralisation de la dernière solution à l'autre vue peut se faire à la fin.

Solution 1

Le passage par une méthode apporte la possibilité de construire une représentation des filtres comme on le souhaite.

▶ Créez la méthode dans la vue :

```
[...]
class InboxView(FolderMixin, TemplateView):
    [...]
    def _get_filters(self, user):
        return {
            'recipient': user,
            'recipient_deleted_at__isnull': True,
        }
[...]
```

Il est choisi de préfixer le nom de la méthode par un caractère de soulignement pour rappeler qu'il s'agit d'un auxiliaire de travail interne.

▶ Déplacez cette méthode depuis la vue vers le greffon et adaptez-la :

```
[...]
class FolderMixin:
    [...]

    def get_context_data(self, **kwargs):
        [...]
        msgs = Message.objects.filter (
            **self._get_filters(self.request.user))
        [...]
[...]
```

La syntaxe en `**` permet la restitution du dictionnaire obtenu, sous forme d'arguments en mots-clés.

© Editions ENI - All rights reserved

Solution 2

Plus exactement, il s'agit d'un raffinement de la solution précédente, dans l'idée de revenir à l'idée première d'avoir une écriture fondée sur un attribut. Un décorateur Python permet justement de tourner une méthode en un attribut à lecture seule.

La passation du paramètre `user` n'est pas un souci puisque l'information reste accessible partout au sein de la vue (ceci pouvait déjà être fait dans la solution précédente).

- Décorez et transformez en `filters()` la méthode `_get_filters()` de la vue :

```
[...]
    @property
    def filters(self):
        return {
            'recipient': self.request.user,
            'recipient_deleted_at__isnull': True,
        }
[...]
```

- Ajustez en conséquence la méthode du greffon :

```
[...]
class FolderMixin:
    [...]
    def get_context_data(self, **kwargs):
        [...]
        msgs = Message.objects.filter(**self.filters)
        [...]
[...]
```

Au prix d'une petite construction syntaxique pour le concevoir, on a quand même l'impression de manipuler un simple attribut par ailleurs.

Solution 3

Les solutions précédentes ont permis à la fois de factoriser du code de traitement et à chaque vue de le personnaliser, en l'occurrence par la pose de leurs critères de filtrage. Cette architecture risque cependant de montrer ses limites de simplicité et de lisibilité si on veut disposer de plus de facteurs de personnalisation.

Il faudrait sans doute multiplier les attributs pour couvrir plus de variations et de combinaisons.

Quand, dans le module des vues, on en arrive à complexifier les requêtes pour demander une collection d'instance de modèles, il est temps de se poser la question de savoir s'il ne serait pas mieux de transférer cette charge de travail au module des modèles. Après tout, ce sont des concepts plus en rapport avec le modèle qu'avec la vue.

Il est plus sain de recentrer la vue sur sa tâche de composition d'un résultat, en la bornant à un rôle de consommateur d'informations auprès d'un producteur d'informations. La structuration est plus lisible ainsi et la maintenance y gagne aussi. Un gestionnaire de modèle est justement fait pour ce rôle de fournisseur d'objets. Si celui disponible par défaut ne suffit pas aux besoins, il ne faut pas hésiter à écrire un gestionnaire personnalisé.

Ici, un gestionnaire `MessageManager` est déjà en place, il attend juste d'être complété. Pour garder le plus de souplesse, une méthode par dossier est prévue, à charge de chacune des vues d'indiquer la méthode qu'elle vise. Fournir le nom de la méthode dans un attribut suffit à cela.

- Mettez pour l'instant de côté la méthode `filters()` de la vue et remplacez-la par un attribut :

```
[...]
class InboxView(FolderMixin, TemplateView):
    # pour FolderMixin:
    folder_name = 'inbox'
    # pour TemplateView:
    template_name = 'messenger/inbox.html'
[...]
```

© Editions ENI - All rights reserved

En prévision d'une future lecture, des commentaires pour étiqueter les ensembles d'attributs aident le développeur à comprendre immédiatement ce qui est standard et ce qui est personnalisé.

▶Ajustez en conséquence la méthode du greffon :

```
[...]
class FolderMixin:
    [...]
    def get_context_data(self, **kwargs):
        [...]
        msgs = getattr(Message.objects, self.folder_name)(
            self.request.user)
        [...]
[...]
```

La demande se réalise en deux étapes : la première procure l'attribut appelable du gestionnaire ayant le nom souhaité par la vue, la seconde réalise l'appel à la méthode et obtient les messages. Les paramètres nécessaires à l'exécution doivent être fournis ; dans ce cas, il s'agit de l'objet utilisateur.

Finalement, tout le travail effectif est migré dans le gestionnaire de modèle et c'était bien le but recherché.

▶Créez la nouvelle méthode dans le gestionnaire :

`mysite\messenger\models.py`

```
[...]
class MessageManager(models.Manager):
    def inbox(self, user):
        filters = {
            'recipient': user,
            'recipient_deleted_at__isnull': True,
        }
        return self.filter(**filters)
    [...]
```

On remarquera un bénéfice supplémentaire non négligeable apporté par cette nouvelle organisation : il devient largement plus facile d'envisager des tests unitaires en rapport avec le jeu de messages attendu dans un dossier pour un certain utilisateur. Auparavant, l'information à contrôler était enfouie au fond d'une vue dont la durée d'instanciation est éphémère au cours d'une chaîne de traitement. Maintenant, il suffit d'appeler une méthode de gestionnaire.

Le même principe d'intégration appliqué à l'autre dossier, celui d'envoi, donne ce codage :

mysite\messenger\views.py

```
[...]
class SentView(FolderMixin, TemplateView):
    # pour FolderMixin:
    folder_name = 'sent'
    # pour TemplateView:
    template_name = 'messenger/sent.html'
[...]
```

mysite\messenger\models.py

```
[...]
class MessageManager(models.Manager):
    [...]
    def sent(self, user):
        filters = {
            'sender': user,
            'sender_deleted_at__isnull': True,
        }
        return self.filter(**filters)
    [...]
```

9.4 Encore plus d'optimisation et d'intégration

Avant l'arrivée de la méthode `inbox()` dans le gestionnaire de modèle, il y avait la méthode `unread_count()`, déjà le fruit d'une intégration. Si on compare attentivement ces deux méthodes, une certaine similitude est manifeste. Effectivement, obtenir le nombre de messages non lus revient à considérer les messages du dossier d'arrivée, ne retenir que ceux qui ne sont pas lus et en faire le comptage, mais enchaîner une à une de telles opérations représente des charges inutiles. Heureusement, avec le principe d'évaluation différée, la construction d'une requête n'engendre pas de coûts intermédiaires.

© Editions ENI - All rights reserved

▶Reformulez la méthode `unread_count()` en prenant appui sur la méthode `inbox()` :

`mysite\messenger\models.py`

```
    [...]
    def unread_count(self, user):
        return self.inbox(user).filter(
            read_at__isnull=True).count()
        # return self.filter(  # ancienne écriture
            # recipient=user,
            # read_at__isnull=True,
            # recipient_deleted_at__isnull=True).count()
    [...]
```

L'appel à `inbox()` rend un objet `QuerySet`, non évalué, qu'il est possible de continuer à affiner. Un filtre complémentaire est posé et en dernier lieu le compte est demandé.

Pour rappel, cette méthode est exploitée à deux occasions : dans un processeur de contexte et dans un appel AJAX. La seconde est choisie pour vérifier le bon fonctionnement de cette nouvelle écriture et deux façons vont être présentées pour en faire la démonstration.

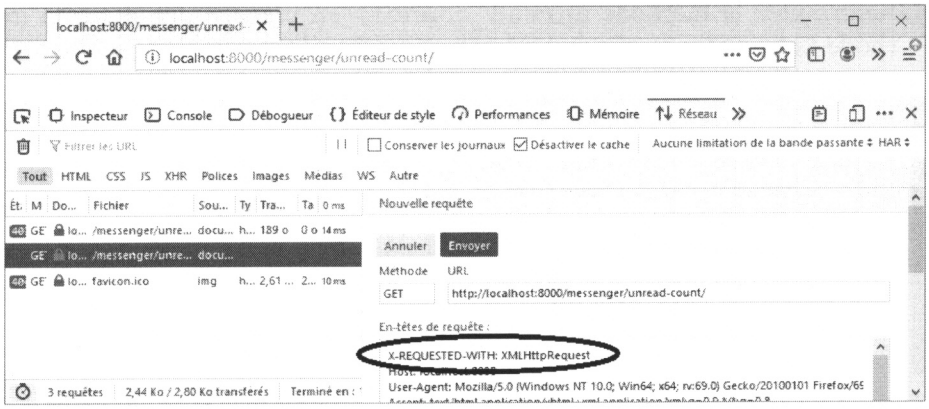
Solution 1 : par navigateur

Les outils de développement du navigateur sont utilisés pour la manipulation, avec l'onglet **Réseau**.

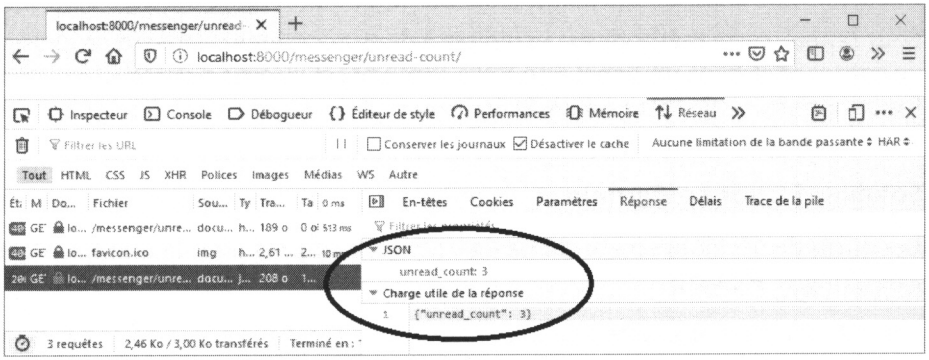
Un premier essai ordinaire à l'adresse `/messenger/unread-count/` va naturellement échouer en `403 Forbidden`, du fait de la restriction de la vue à devoir être appelée en mode AJAX. Mais cela donne une requête pouvant être modifiée et renvoyée.

▶Éditez la requête pour lui ajouter l'en-tête supplémentaire nécessaire :

```
X-REQUESTED-WITH: XMLHttpRequest
```



Après envoi, le résultat est conforme :



Solution 2 : par ligne de commande

Le principe est de se placer en tant que client et d'interroger le serveur de développement. La démonstration est réalisable avec un interpréteur Python ordinaire, il n'y a pas de nécessité à employer un shell de Django et l'outil IDLE convient très bien à l'affaire.

Inutile d'installer des paquets externes, le module standard `urllib.request` est l'outil nécessaire et suffisant. La petite complexité par rapport à son utilisation basique vient du besoin de mettre en place une gestion de cookies pour l'authentification.

© Editions ENI - All rights reserved

Tout d'abord, voici les imports et la préparation des composants de base de la recette :

```
Python 3.7.2 (tags/v3.7.[...]) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more in[...]
>>> import urllib.request
>>> from http.cookiejar import CookieJar

>>> ck_handler = urllib.request.HTTPCookieProcessor(CookieJar())
>>> opener = urllib.request.build_opener(ck_handler)
>>> HOST = http://localhost:8000/
>>> LOGIN_URL = HOST + "login/"
>>> COUNT_URL = HOST + "messenger/unread-count/"
>>> AJAX_HEADER = {'X-REQUESTED-WITH': 'XMLHttpRequest'}
```

Une étape préliminaire consiste à s'authentifier auprès du site, pour récupérer un cookie qui sera ensuite automatiquement transmis dans les requêtes ultérieures :

```
>>> req = urllib.request.Request(LOGIN_URL)
>>> with opener.open(req) as f:
        print(f.read().decode('utf-8'))

logged in
>>>
```

L'étape porteuse de valeur est la demande AJAX :

```
>>> req = urllib.request.Request(COUNT_URL, headers=AJAX_HEADER)
>>> with opener.open(req) as f:
        print(f.read().decode('utf-8'))

{"unread_count": 3}
>>>
```

À chacun des dossiers correspond maintenant une méthode sur le gestionnaire de modèle pour obtenir une collection d'instances du modèle. Cette collection est normalement destinée à être affichée, de façon plus ou moins détaillée. Un format très classique est une présentation tabulaire, avec typiquement le sujet, mais très probablement aussi l'indication de l'expéditeur ou du destinataire, voire les deux. Or, ces informations proviennent d'autres modèles et sont stockées dans d'autres tables de la base de données, mais le mécanisme de relations et de clés étrangères établit des liens permettant d'abstraire cette répartition et d'assurer une apparence de continuité. On n'a pas forcément conscience de la charge que cela représente pour l'infrastructure, en contrepartie de cette apparente fluidité. Il est vrai aussi que le fait d'utiliser un ORM réduit la visibilité sur les efforts déployés en arrière-plan pour délivrer le résultat.

C'est pourquoi il est bon parfois d'activer les traces des requêtes à la base de données pour constater que, si effectivement le résultat est produit et qu'il est correct, son coût peut parfois être démesuré.

On va mieux s'en rendre compte avec un exemple simple : présenter la liste des messages du dossier d'arrivée avec des attributs, parmi lesquels le nom de l'expéditeur. Les traces de `django.db.backends` sont au niveau `DEBUG`.

Le code se contente d'obtenir les messages en dossier d'arrivée d'un utilisateur et de faire une boucle pour les exposer :

```
D:\dj>py manage.py shell
>>> from django.contrib.auth.models import User
>>>
>>> user = User.objects.get(pk=2)
[...]_django.db.backends_DEBUG - (0.025) SELECT "auth_user"."id",
[...]oined" FROM "auth_user" WHERE "auth_user"."id" = 2; args=(2,)
>>>
>>> from mysite.messenger.models import Message
>>>
>>> msgs = Message.objects.inbox(user)
>>> for m in msgs:
...     print(m.subject, "/", m.sender.get_username())
...
[...]_django.db.backends_DEBUG - (0.030) SELECT "messenger_message"
[...]nger_message"."recipient_deleted_at" FROM "messenger_message"
WHERE ("messenger_message"."recipient_id" = 2
AND "messenger_message"."recipient_deleted_at" IS NULL)
```

© Editions ENI - All rights reserved

```
ORDER BY "messenger_message"."sent_at" DESC,
"messenger_message"."id" DESC; args=(2,)
```

Jusque-là, les requêtes collent aux besoins : une sélection d'un objet utilisateur et une sélection de messages. La suite est plus discutable :

```
[...]_django.db.backends_DEBUG - (0.000) SELECT "auth_user"."id",
[...]oined" FROM "auth_user" WHERE "auth_user"."id" = 1; args=(1,)
Bienvenue / root
[...]_django.db.backends_DEBUG - (0.000) SELECT "auth_user"."id",
[...]oined" FROM "auth_user" WHERE "auth_user"."id" = 1; args=(1,)
Via loaddata - A / root
[...]_django.db.backends_DEBUG - (0.000) SELECT "auth_user"."id",
[...]oined" FROM "auth_user" WHERE "auth_user"."id" = 1; args=(1,)
Annonce / root
>>>
```

Il y a autant de requêtes qu'il y a de messages. Ce serait même le double si on avait demandé à afficher le destinataire en plus. Manifestement, ce code manque d'efficience et sa conception doit être revue. Sachant d'avance que la relation devra être suivie pour lire la table liée, autant le faire en une seule fois dès la requête d'origine. Cette solution est apportée par la méthode `select_related()` de `QuerySet`.

L'exemple doit donc être révisé pour s'écrire :

```
>>> msgs = Message.objects.inbox(user).select_related('sender')
```

Les traces deviennent plus raisonnables :

```
>>> for m in msgs:
...     print(m.subject, "/", m.sender.get_username())
...
[...]_django.db.backends_DEBUG - (0.010) SELECT "messenger_message"
[...]", "messenger_message"."subject", "messenger_message"."body",
[...]", T3."id", T3."password", T3."last_login",
[...]te_joined" FROM "messenger_message" INNER JOIN "auth_user" T3
ON ("messenger_message"."sender_id" = T3."id")
WHERE ("messenger_message"."recipient_id" = 2
AND "messenger_message"."recipient_deleted_at" IS NULL)
ORDER BY "messenger_message"."sent_at" DESC,
"messenger_message"."id" DESC; args=(2,)
Bienvenue / root
Via loaddata - A / root
Annonce / root
```

La requête est plus grosse, mais elle présente l'avantage d'être suffisante pour produire le résultat souhaité, sans un flot de requêtes complémentaires comme précédemment.

Il reste à intégrer cette technique dans le gestionnaire de modèle. Elle est la même pour `inbox()` et pour `sent()`, sauf que l'objet auxiliaire est différent selon le cas : pour un dossier d'arrivée, on aura naturellement tendance à présenter seulement l'expéditeur puisqu'il n'y a aucun intérêt à présenter le destinataire (en l'occurrence soi-même) dans ce contexte, et inversement pour le dossier d'envoi.

Pour d'éventuels autres dossiers, plus généraux comme une corbeille ou un archivage, la présentation simultanée des deux intervenants retrouve du sens.

Pour gérer ces nuances, tout en s'interdisant de dupliquer du code, une méthode de travail est introduite, de façon à pouvoir centraliser un comportement de base commun aux dossiers :

mysite\messenger\models.py

```
    [...]
    def inbox(self, user):
        related = ('sender',)
        filters = {
            'recipient': user,
            'recipient_deleted_at__isnull': True,
        }
        return self._folder(related, filters)
    [...]
    def sent(self, user):
        related = ('recipient',)
        filters = {
            'sender': user,
            'sender_deleted_at__isnull': True,
        }
        return self._folder(related, filters)
    [...]
```

© Editions ENI - All rights reserved

Cette méthode s'écrit initialement ainsi :

```
[...]
def _folder(self, related, filters):
    """Code de base, en commun aux dossiers."""
    return self.select_related(*related).filter(**filters)
[...]
```

On pourra vérifier que l'exemple précédent, sous la forme de sa première écriture, retrouve maintenant une quantité maîtrisée de requêtes SQL.

Une évolution vient d'être apportée à la méthode `inbox()`. Or, celle-ci servant de base à la méthode `unread_count()`, il faut en examiner les incidences. La méthode reste compatible puisqu'une éventuelle remontée des propriétés de l'expéditeur ne remet pas en cause son fonctionnement. Mais cela ne sert à rien, donc il y a matière à améliorer ce point.

Un paramètre est ajouté, dont la valeur par défaut correspond au cas d'usage majoritaire. Il conditionne ensuite la mise en œuvre ou pas de la récupération de l'objet lié :

```
[...]
def inbox(self, user, related=True):
    related = ('sender',) if related else None
    [...]
    return self._folder(related, filters)
[...]
```

En aval, la méthode de base doit être accordée à la possible absence de réclamation d'objets liés :

```
[...]
def _folder(self, related, filters):
    [...]
    qs = self.all()
    if related:
        qs = qs.select_related(*related)
    return qs.filter(**filters)
[...]
```

En amont, la méthode qui se limite à un comptage doit aussi être adaptée, pour ne pas demander d'objet lié :

```
    [...]
    def unread_count(self, user):
        return self.inbox(user, related=False).filter(
            read_at__isnull=True).count()
    [...]
```

En réalité, il se trouve que demander ou pas des objets liés est sans effet lorsque la méthode `count()` est employée : dans ce cas les relations ne sont pas suivies si elles ne concourent pas au filtrage des enregistrements. Mais il vaut mieux considérer ce fait comme un détail d'implémentation ou une optimisation, et écrire son code proprement en toutes circonstances.

Pour la démonstration, ces deux constructions donnent la même trace :

```
>>> Message.objects.unread_count(user)
[...]
>>> Message.objects.inbox(user).filter(
...     read_at__isnull=True).count()
[...]_django.db.backends_DEBUG - (0.005) SELECT COUNT(*)
 AS "__count" FROM "messenger_message"
 WHERE ("messenger_message"."recipient_id" = 2
 AND "messenger_message"."recipient_deleted_at" IS NULL
 AND "messenger_message"."read_at" IS NULL); args=(2,)
3
```

9.5 Contrôle du cache

Les vues, tout au moins celles des dossiers, présentées jusqu'à maintenant, se doivent de refléter l'état actuel des contenus. En effet, ces contenus peuvent changer à tout moment et on s'exposerait à un fort risque d'inexactitude à laisser montrer une page issue d'un système de cache.

Même si l'application ou le site ne mettent pas en œuvre un tel mécanisme, il ne manque pas d'autres maillons dans la chaîne de transmission pour le faire, ne serait-ce que le navigateur.

© Editions ENI - All rights reserved

Garder la main sur la mise en cache a donc son importance. Si on observe les en-têtes de la réponse au navigateur, dans l'implémentation actuelle de l'application, on ne trouve rien en rapport avec la mise en cache.

Des décorateurs sont disponibles pour faciliter cette tâche. Il est possible de choisir finement les mots-clés d'en-tête avec le décorateur `cache_control(**kwargs)`, mais le plus simple est d'utiliser le décorateur `never_cache` pour tout désactiver.

▶Ajoutez le décorateur de désactivation de cache :

`mysite\messenger\views.py`

```
[...]
from django.views.decorators.cache import never_cache
[...]
never_cache_m = method_decorator(never_cache)
[...]
class FolderMixin:
    [...]
    @never_cache_m
    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
    [...]
[...]
```

Le décorateur devant être appliqué à une méthode, il est enveloppé dans le décorateur `method_decorator`. De plus, en prévision d'usages répétés ultérieurement, il en est fait un attribut au niveau du module. Par pure convention, le suffixe `_m` (significatif de `method` mais en version courte) est adjoint au nom d'origine.

▶Dans le même esprit, factorisez le décorateur pour authentification, il sera utilisé à nouveau plus tard :

`mysite\messenger\views.py`

```
[...]
login_required_m = method_decorator(login_required)
never_cache_m = method_decorator(never_cache)
[...]
class FolderMixin:
    [...]
    @never_cache_m
    @login_required_m
```

```
    def dispatch(self, *args, **kwargs):
    [...]
[...]
```

L'observation des en-têtes de réponse arrivant au navigateur montre maintenant des compléments de ce genre :

```
Expires: Tue, 05 [...] 20[...] 17:30:23 GMT
Cache-Control: max-age=0, no-cache, no-store, must-revalidate
```

9.6 Lecture de message

Pour aborder le sujet, un minimum nécessaire va être établi pour produire un résultat. Ceci permet de découvrir qu'avec les conventions implicites d'une infrastructure logicielle, il peut être facile et rapide de monter une implémentation. Il est entendu que ce minimum devra ensuite être complété, mais plutôt pour mettre des restrictions, notamment celles relatives à la sécurité.

Classiquement, on accède à un message en particulier par le suivi d'un hyperlien, notamment à partir d'une liste des messages telle que celle des dossiers vus précédemment.

▶Modifiez le fichier de routage :

`mysite\messenger\urls.py`

```
[...]
from .views import (InboxView, SentView,
    MessageView,
[...]
urlpatterns = [
    [...]
    path('view/<int:pk>/', MessageView.as_view(), name='view'),
    [...]
```

▶Modifiez le fichier des vues pour créer la nouvelle vue :

`mysite\messenger\views.py`

```
[...]
from django.views.generic import (DetailView,
    RedirectView, TemplateView,
)
[...]
```

© Editions ENI - All rights reserved

```
class MessageView(DetailView):
    # pour DetailView
    model = Message
[...]
```

- Créez un gabarit temporaire par copie d'un existant :

```
D:\dj>cd mysite\messenger\templates\messenger
D:\[...]ssenger\templates\me[...]>copy index.html message_detail.html
        1 fichier(s) copié(s).
```

- Si un supplément de visibilité est souhaité, ajoutez-y cette ligne pour constater l'acquisition du message (format brut, sans mise en forme) :

```
[...]
<div>{{ object }}</div>
[...]
```

- Vérifiez la conservation du fonctionnement du site, en accédant à la page d'un message existant, telle que : `/messenger/view/2/`.

Un fonctionnement avec si peu de code mérite quelques explications. En réalité, tout repose sur la vue générique utilisée : `DetailView`. Comme elle est elle-même un assemblage de vues de base et de greffons, les informations sur son mode de fonctionnement se retrouvent un peu éparpillées dans la documentation.

Cette vue générique est prévue pour afficher une page spécifiquement pour un objet, plus précisément une instance d'un modèle. Par le jeu des conventions, elle saura se débrouiller pour acquérir l'instance et faire le reste du travail, sauf sur un point où il faut être directif : quel est le modèle. Le plus simple est encore de le dire clairement, ce qui a été fait avec l'attribut `model`. Un autre moyen est d'employer l'attribut `queryset`, dont l'équivalent à l'emploi de `model` devrait être écrit ainsi :

```
queryset = Message._default_manager.all()
```

Cette variante d'écriture ne présente un avantage que si on souhaite remplacer le `all()`, afin d'affiner l'interrogation de la table avec des critères restrictifs comme des filtres ou un classement. Une dernière possibilité consiste à surcharger la méthode `get_queryset()`. Si aucune de ces trois façons n'est donnée, on s'expose à la levée d'une exception, à moins de surcharger à un niveau plus haut la méthode `get_object()`.

À partir du jeu des candidats, il faut cibler un enregistrement et un seul. La vue générique supporte deux identifiants potentiels à extraire de l'URL : soit la clé primaire, soit un label unique (*slug* en anglais). Le nom de l'identifiant peut être précisé par un attribut, mais par défaut ce sont respectivement `pk` et `slug`. La clé primaire est un choix qui convient à l'application, voilà pourquoi il a été choisi pour le motif de l'URL de nommer `pk` son paramètre capturé.

L'étape suivante est l'instanciation d'un gabarit pour construire la page. À nouveau, on pourrait imposer son choix de nom de gabarit avec l'attribut `template_name`, mais par défaut il est composé selon le motif `<app>/<model>_detail.html`, en caractères minuscules. Généralement, c'est bien suffisant, mais la personnalisation peut encore être poussée plus loin, avec deux possibilités : a) la partie du motif `_detail` est une valeur par défaut, qu'on peut surcharger par l'attribut `template_name_suffix` ; b) summum de l'individualisation, le nom du gabarit peut être obtenu de l'enregistrement en base de données de l'objet, pioché dans une colonne dont on donne le nom par l'attribut `template_name_field`.

Appliqué à la situation, cela donne `messenger/message_detail.html`, ce qui justifie le choix de ce nom lors de la copie de fichier.

Enfin, l'objet à montrer est mis à disposition dans les données de contexte à la fois sous deux variables. L'une porte le nom invariable `object`. Voilà ce qui donne la source de la citation dans le gabarit. L'autre est nommée selon l'attribut `context_object_name`, dont la valeur par défaut est le nom du modèle en minuscules. On aurait donc pu écrire `{{ message }}` dans le gabarit avec le même effet.

Cela suggère une mise en garde : attention à ne pas passer à côté d'une collision de noms entre ces variables et une autre qui pourrait être apportée par un processeur de contexte, que ce soit un processeur du projet, un processeur d'une application tierce ou un processeur intégré activé. La variable de la vue a silencieusement priorité sur celle d'un processeur. La remarque est valable aussi bien pour l'emploi par défaut du nom du modèle que pour un nom mal approprié fourni par l'attribut.

L'implémentation à ce stade pourrait peut-être suffire pour un site où tout est public et à accès libre. Ce n'est pas le cas ici, il faut donc ajouter des restrictions.

© Editions ENI - All rights reserved

Pour commencer, comme pour les vues précédentes, l'utilisateur doit être authentifié. De même, il est préférable de limiter les verbes supportés et d'empêcher une mise en cache :

▶Complétez la vue :

`mysite\messenger\views.py`

```
[...]
class MessageView(DetailView):
    http_method_names = ['get']
    [...]

    @never_cache_m
    @login_required_m
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
[...]
```

Le point de sécurité suivant concerne l'autorisation pour un utilisateur à ne visualiser que les messages qui le concernent et pas ceux du voisin. Le fait qu'il ne lui soit présenté dans des listes que les hyperliens vers ses messages dûment autorisés ne l'empêche nullement de taper un identifiant quelconque dans l'adresse de l'URL pour tenter de voir ailleurs. Un message est considéré accessible à un utilisateur s'il en est l'expéditeur ou le destinataire, voire les deux puisque s'écrire à soi-même est a priori valide.

Un simple attribut fixe, tel que `queryset`, ne suffira pas puisque la contrainte doit être personnalisée à l'utilisateur connecté. Le passage par une méthode est nécessaire.

Une première idée consiste à faire un contrôle a posteriori, c'est-à-dire récupérer l'objet sans condition et vérifier ensuite qu'il satisfait aux permissions. Ceci se réalise par surcharge de la méthode `get_object()` :

```
from django.http import Http404
[...]
    def get_object(self):
        object = super().get_object()
        user = self.request.user
        if not (object.sender == user
                or object.recipient == user):
            raise Http404
        return object
```

Si l'objet existe mais n'est pas permis pour l'utilisateur, une exception est levée, pour aboutir au retour d'une page standard de type 404 *Not Found*. On pourrait être tenté par l'exception `PermissionDenied`, ce qui renverrait une page standard de type 403 *Forbidden*, mais c'est à éviter au moins pour deux raisons : la première raison stipule qu'en matière de sécurité, moins on en dit et mieux on se protège ; la seconde raison est d'adopter un comportement similaire à celui de l'inexistence de l'objet demandé et, dans ce cas, il s'agit précisément de cette exception `Http404`, avec un texte.

Ici, on peut se passer de donner une raison, pour rester discret puisque la requête est forgée et illégale.

Si on y réfléchit bien, cette solution a tout de même quelques désavantages :

- C'est un gâchis de rapatrier un enregistrement de la base de données, construire un objet et finalement le jeter immédiatement, même s'il est vrai que ce mauvais scénario ne doit normalement jamais arriver.
- Sauf à chercher à optimiser, ce code génère pour une partie de son travail deux à trois requêtes SQL : une première pour lire le message, une deuxième pour lire l'utilisateur expéditeur, et éventuellement une troisième pour lire l'utilisateur destinataire si la condition sur l'expéditeur n'a pas conclu le test par anticipation.
- Il est fort probable d'être amené à exposer l'expéditeur et le destinataire dans l'affichage détaillé d'un message, d'où la troisième requête SQL si éventuellement elle avait pu être épargnée dans le contrôle, pour la raison donnée dans le point précédent.

Pour gommer ces inconvénients, il faut regarder parmi les méthodes mises en œuvre dans des niveaux plus profonds. La plus appropriée est sans doute la méthode `get_queryset()`.

Une écriture envisageable prendrait cette forme :

```
from django.db.models import Q
[...]
    def get_queryset(self):
        user = self.request.user
        return super().get_queryset()\
            .filter(Q(recipient=user) | Q(sender=user))
```

© Editions ENI - All rights reserved

La classe `Q` permet de construire des objets à visée SQL à partir d'expressions comme on les utilise habituellement pour la méthode `filter()`. Cela permet ensuite de les passer comme paramètres, de les combiner pour former des formulations plus complexes, etc. C'est en particulier l'outil nécessaire pour réaliser des requêtes devant comporter une logique de OU, puisque tous les autres moyens obéissent implicitement à une logique de ET. Les objets Q peuvent être associés avec des opérateurs :

- `&` pour une logique additive (`AND`),
- `|` pour une logique alternative (`OR`),
- `~` pour une logique renversée (`NOT`).

Par exemple :

```
Q(fld1__startswith='_') | (Q(fld2=12) & ~Q(fld3__contains='X'))
```

Un bénéfice immédiat est de n'avoir plus qu'une requête SQL, porteuse du contrôle d'autorisation. Ainsi, le message ne transite qu'à bon escient, c'est-à-dire s'il est réellement utilisable de façon légitime.

En revanche, il reste la question des requêtes supplémentaires pour l'expéditeur et le destinataire. On a déjà vu qu'un `select_related()` est la solution.

Deux options de placement sont possibles :

Option A : Directement dans la requête personnalisée à construire :

```
    def get_queryset(self):
        [...]
        return super().get_queryset()\
            .select_related('sender', 'recipient')\
            .filter(Q(recipient=user) | Q(sender=user))
```

Option B : Comme faisant partie naturelle et inconditionnelle de la requête de base :

```
class MessageView(DetailView):
    [...]
    # model = Message
    queryset = Message.objects\
        .select_related('sender', 'recipient')
```

Lorsqu'on exploite l'attribut `queryset`, l'attribut `model` ne gêne pas, mais il n'est plus utilisé et il vaut mieux l'écarter pour ne pas introduire de confusion.

Techniquement, les deux options sont équivalentes et au final la requête générée est la même. L'option B a un léger avantage sémantique, en séparant d'un côté une optimisation et d'un autre côté une contrainte de sécurité.

Même si on a progressé, cette écriture n'est pas exempte de tout reproche. On trouve enfouies au sein d'une certaine vue des notions, d'une part ayant trait directement et uniquement au modèle, et d'autre part qui ne sont pas vraiment exclusives à cette vue. Un besoin identique pourra tout à fait se retrouver à d'autres occasions, comme une autre vue ou pour une API.

Étant donné qu'il n'est pas bon de dupliquer du code, autant chercher tout de suite à l'externaliser.

Comme cela a été fait au moment du travail avec les dossiers, une intégration dans le gestionnaire de modèle est une excellente solution pour obtenir la factorisation souhaitée. D'ailleurs, la présence dans un module de vues, d'objets de classe `Q`, provenant donc d'un module de modèles (`django.db.models`), n'est pas un interdit, mais c'est un indice supplémentaire pour se poser la question de l'opportunité d'un remaniement.

L'idée est donc de migrer vers une méthode de gestionnaire de modèle la formulation d'une requête optimisée d'obtention des messages autorisés pour tel utilisateur. Le nom `allowed()` est choisi pour cette méthode et elle doit recevoir un utilisateur en paramètre.

▶ Complétez le gestionnaire :

`mysite\messenger\models.py`

```
[...]
class MessageManager(models.Manager):
    [...]
    def allowed(self, user):
        return self.select_related('sender', 'recipient')\
            .filter(models.Q(recipient=user)
                    | models.Q(sender=user))
```

© Editions ENI - All rights reserved

Il s'agit maintenant de réorganiser la vue pour tirer profit de cette nouvelle fonctionnalité du gestionnaire de modèle. L'intervention va à nouveau concerner la méthode `get_queryset()`, mais il va falloir tenir compte d'un point technique : prendre appui sur `super().get_queryset()` ne va plus convenir, car il est rendu un objet `QuerySet` alors que la méthode `allowed()` visée est portée par un objet de classe `Manager`. Or, à partir de `QuerySet`, il n'est pas possible de remonter à un `Manager`, au mieux est-il possible de remonter à son modèle. Pour exploiter le parent, il faudrait utiliser cette construction :

```
super().get_queryset().model.objects.allowed(user)
```

Puisque l'on travaille dans un environnement spécifique, il est inutile de vouloir être générique et l'expression peut se raccourcir pour un accès direct.

▶Modifiez la vue :

`mysite\messenger\views.py`

```
[...]
class MessageView(DetailView):
    [...]
    #model = Message
    [...]
    def get_queryset(self):
        return Message.objects.allowed(self.request.user)
```

Du fait d'avoir entièrement établi la requête, l'attribut `model` n'a plus d'utilité. Il est préférable de l'enlever pour ne pas laisser de doute à ce sujet.

Le dernier point à régler concerne l'alimentation du champ `read_at` pour signifier que le message a été consulté, si l'utilisateur en est le destinataire. Initialement, la valeur du champ est `NULL`. À la lecture du message, on souhaite mémoriser ce fait en mettant l'instant dans ce champ.

Il s'agit seulement d'une bascule à la première lecture, pas une mémorisation de la toute dernière consultation.

L'endroit adéquat pour travailler l'objet se situe au retour de la méthode `get_object()`. Il faut donc surcharger cette méthode pour intercaler un traitement.

▶Modifiez la vue pour apporter cette méthode :

mysite\messenger\views.py

```
[...]
from django.utils.timezone import now
[...]
class MessageView(DetailView):
    [...]
    def get_object(self):
        object = super().get_object()
        if object.read_at is None\
                and object.recipient == self.request.user:
            object.read_at = now()
            object.save(update_fields=['read_at'])
        return object
```

Les deux conditions sur le message sont :

1) qu'il n'ait jamais été lu ;

2) que ce soit un message en réception, c'est-à-dire que son destinataire est l'utilisateur authentifié.

Pour une optimisation des performances, une légère préférence d'ordre est accordée au critère relatif à la lecture, car il est plus probable qu'il ne soit pas satisfait que l'inverse et cela épargne en conséquence l'évaluation du second critère.

Quand les conditions sont réunies, le champ est renseigné avec l'instant présent. On demande ensuite une sauvegarde de l'objet pour mettre à jour son enregistrement en base de données. Une optimisation est réalisée à cette occasion : par défaut, la méthode `save()` reporte en base la totalité des champs de l'objet sans discernement, qu'ils aient été modifiés ou non depuis l'acquisition de l'objet. Or, ici, on sait avec certitude que seul un certain champ a été modifié, donc il est totalement inutile de transmettre les autres. Le paramètre `update_fields` permet de préciser cette finesse en listant le ou les noms des seuls champs à considérer dans l'opération.

© Editions ENI - All rights reserved

Les traces confirment l'économie des champs dans la requête soumise :

```
[...]_django.db.backends_DEBUG - (0.005) UPDATE "messenger_message"
SET "read_at" = '20nn-nn-nnTnn:nn:nn.nnnnnn'::timestamp
WHERE "messenger_message"."id" = 8;
args=(datetime.datetime(20nn, nn, nn, nn, nn, nn, nnnnnn), 8)
```

9.7 Composition de message

- Modifiez le fichier de routage :

`mysite\messenger\urls.py`

```
[...]
from .views import (InboxView, SentView,
    MessageView, WriteView,
[...]
urlpatterns = [
    [...]
    path('write/', WriteView.as_view(), name='write'),
    [...]
```

- Modifiez le fichier des vues pour créer la nouvelle vue :

`mysite\messenger\views.py`

```
[...]
from django.views.generic import (CreateView, DetailView,
    RedirectView, TemplateView,
)
[...]
class WriteView(CreateView):
    # pour CreateView
    model = Message
    fields = ['subject', 'body', 'recipient']
[...]
```

- Créez un gabarit temporaire par copie d'un existant :

```
D:\dj>cd mysite\messenger\templates\messenger
D:\[...]ssenger\templates\me[...]>copy index.html message_form.html
        1 fichier(s) copié(s).
```

- Si un supplément de visibilité est souhaité, ajoutez-y cette ligne pour constater la disponibilité d'un formulaire (format brut, sans mise en forme) :

```
[...]
<div>{{ form }}</div>
[...]
```

- Vérifiez la conservation du fonctionnement du site, en accédant à la page de création d'un message : `/messenger/write/`.

Comme pour la vue de lecture, la vue pour composer un message va être basée sur une vue générique : `CreateView`. Son nom l'exprime clairement, cette vue sert à créer un objet. Le principe de fonctionnement repose sur ces grandes étapes, de façon générale pour du traitement de formulaire :

- 1. Une action HTTP `GET` pour se faire servir un formulaire initial vierge.
- 2. Une action HTTP `POST` pour soumettre le formulaire.
- 3. Une validation du formulaire soumis. Si elle est négative, le formulaire est à nouveau servi et retour à l'étape 2. Si elle est positive, traitement des données et passage à l'étape 4.
- 4. Acquittement du traitement par retour d'un ordre de redirection de page.

Pour l'instant, le minimum de renseignement a été donné à la vue générique pour qu'elle commence à travailler. L'attribut `model` donne la classe de l'objet dont il faudra créer une instance. L'attribut `fields` liste les noms des champs devant faire partie du formulaire automatique proposé. Si la composition automatique n'est pas à son goût, on peut implémenter son propre formulaire et utiliser à la place l'attribut `form_class` pour fournir le nom de sa classe.

Dans le cas où la génération automatique est employée, l'obligation de mentionner chacun des noms des champs peut sembler pénible, mais elle est justifiée par des raisons de sécurité. Dans d'anciennes versions de Django, l'attribut était optionnel et il était convenu qu'en son absence tous les champs étaient mis dans le formulaire. Cette situation pouvait être conforme et parfaitement assumée par un développeur vigilant à la conception initiale du projet.

© Editions ENI - All rights reserved

Mais il était trop facile par la suite, notamment à l'occasion d'une maintenance par un intervenant moins familier du projet, d'introduire de nouveaux champs, non publics, dans un modèle et d'oublier qu'ils allaient automatiquement se retrouver dans les formulaires, d'où un potentiel de faille de sécurité. L'attribut a donc été rendu obligatoire, mais pour conserver de la rétrocompatibilité, il est permis de citer une valeur spéciale : `__all__`, interprétée comme un raccourci par lequel le développeur affirme qu'il a connaissance du risque encouru et en prend toute la responsabilité. Cette option est évidemment déconseillée et ne saurait en aucun cas être justifiée par une volonté d'en écrire moins. Ici, la question ne se pose pas puisque de toute évidence certains champs n'ont pas à être proposés à la saisie d'un utilisateur.

Pour la partie gabarit, les variantes sont les mêmes que celles citées dans la section concernant la lecture de message, puisqu'un même greffon se charge de ce travail. La seule différence tient à la valeur par défaut du suffixe : `_form` pour une création, d'où au plus simple : `messenger/message_form.html`.

Le formulaire est mis à disposition dans les données de contexte sous le nom fixe `form`, d'où la variable suggérée dans le gabarit pour une observation brute.

Sur cette base de départ trop ouverte, il faut comme d'habitude ajouter des restrictions. Au minimum comme pour les autres vues, il faut que l'utilisateur soit authentifié, et il est préférable de limiter les verbes supportés et d'empêcher une mise en cache.

▶ Complétez la vue :

`mysite\messenger\views.py`

```
[...]
from django.views.decorators.csrf import csrf_protect
from django.views.decorators.debug import\
    sensitive_post_parameters
[...]
login_required_m = method_decorator(login_required)
csrf_protect_m = method_decorator(csrf_protect)
never_cache_m = method_decorator(never_cache)
sensitive_post_parameters_m = method_decorator(
    sensitive_post_parameters('subject', 'body'))
[...]
```

```
class WriteView(CreateView):
    http_method_names = ['get', 'post']
    [...]

    @sensitive_post_parameters_m
    @never_cache_m
    @csrf_protect_m
    @login_required_m
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)
[...]
```

Deux nouveaux décorateurs sont introduits pour cette vue : `csrf_protect` et `sensitive_post_parameters`.

csrf_protect

Le nom est basé sur le sigle de *Cross Site Request Forgery*. Il s'agit d'une technique d'attaque par laquelle un site malintentionné tente de profiter du navigateur pour provoquer silencieusement des requêtes forgées vers un site vulnérable. La discrétion de l'attaque tient beaucoup à la conservation normale au sein du navigateur d'un cookie d'authentification auprès du site ciblé par l'attaque.

Parmi les techniques de défense, il en est une qui consiste à protéger une requête entraînant une action avec effet, dite non sûre, telle qu'initiée par `POST`, par une nécessaire requête préalable de type sûr, telle que `GET`, afin de récupérer une pièce d'information non prédictible à refournir ensuite. Le serveur effectue alors un contrôle de cohérence pour s'assurer de la légitimité du demandeur avant d'engager l'action. La sûreté du mécanisme repose sur le fait que le site malveillant, bien que pouvant pousser le navigateur à interroger un autre site, ne se verra pas donner accès à sa réponse de la part du navigateur. Si donc il est malgré tout possible de provoquer un `POST` en aveugle, il manquera la pièce d'information non lisible du `GET`, tout autant lancé en aveugle. Par exemple, voici une trace d'un navigateur sur une tentative de lecture de propriété d'un objet `XMLHttpRequest`, à l'initiative d'une requête pourtant traitée avec succès par le serveur :

```
Blocage d'une requête multiorigines (Cross-Origin Request) :
la politique «Same Origin» ne permet pas de consulter la
ressource distante située sur http://[...]. Raison : l'en-tête
CORS «Access-Control-Allow-Origin» est manquant.
```

© Editions ENI - All rights reserved

Pour des usages légitimes et maîtrisés, des techniques complémentaires permettent de poser des dérogations pour autoriser ce genre de trafic intersites.

La protection est habituellement apportée par la présence de cet intergiciel dans la configuration du site : `django.middleware.csrf.CsrfViewMiddleware`.

Du fait de son importance pour la sécurité, ce composant est activé par défaut par l'assistant de création de site. Il a le mérite d'apporter la protection à toutes les requêtes non sûres.

Si une telle mise en œuvre globale ne convient pas, par exemple parce que la protection ne doit pas s'appliquer à certaines vues particulières, deux options sont possibles : soit conserver l'intergiciel mais décorer ces vues avec une logique d'exclusion par `@csrf_exempt`, soit désactiver l'intergiciel, mais décorer toutes les autres vues avec une logique d'inclusion par `@csrf_protect`. Un critère pour juger de la meilleure logique peut être la proportion entre les deux familles de vues. La présence à la fois de l'intergiciel et du décorateur de protection semble une redondance, mais elle n'est pas nuisible.

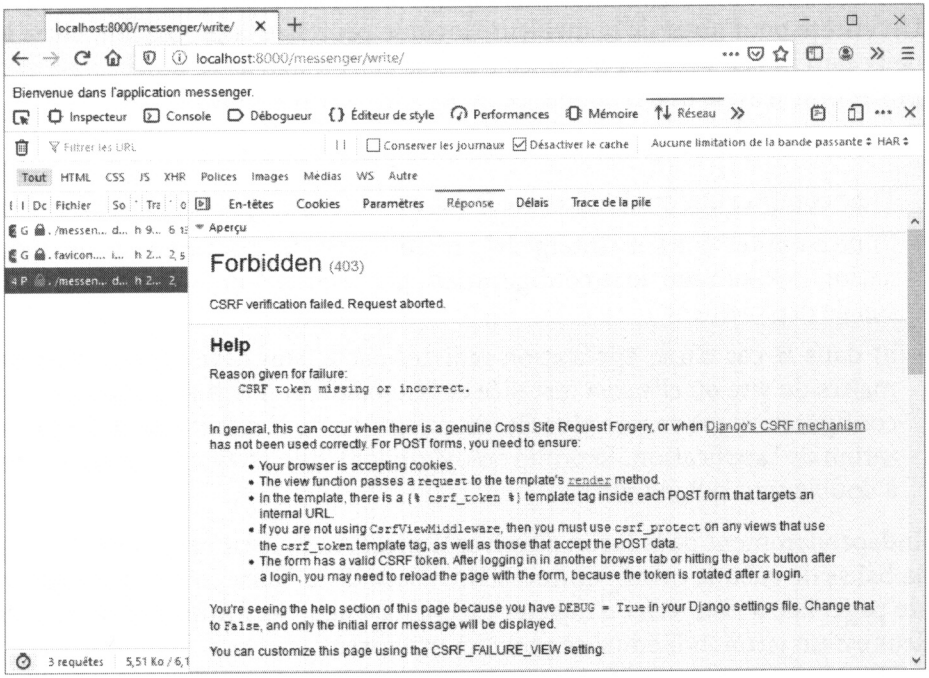
Au contraire, elle est recommandable pour plusieurs raisons :

- 1) ne compter que sur le décorateur expose le risque de l'oublier sur une vue ;
- 2) ne compter que sur l'intergiciel présente aussi un risque en cas de dégradation involontaire de la configuration, par exemple un raté dans le processus de déploiement ;
- 3) dans le cas d'une application redistribuable, son développeur n'est pas maître du site où elle sera installée et l'exposition au risque d'une mauvaise configuration est encore plus élevée, malgré toute la qualité de la documentation de l'application. Exprimé en métaphore, mieux vaut un verrou fermé à double tour qu'un seul.

Indépendamment de la façon d'activer la protection, il est nécessaire d'inclure la balise de gabarit `csrf_token` à l'intérieur de la balise `<form>` du gabarit de page. Elle a pour effet d'injecter un champ caché de formulaire dont la valeur est un jeton individualisé pour cet utilisateur.

Ce jeton est stable puisqu'il a une durée de vie d'un an (52 semaines précisément), mais il est habillé différemment à chaque emploi à l'aide d'un sel cryptographique et son contenu enveloppé est tout de même renouvelé à chaque phase de connexion. À la soumission du formulaire, le contenu de ce jeton est comparé avec un contenu de référence. Par défaut, ce contenu de référence est véhiculé par un cookie. Il existe bien une alternative pour qu'il soit conservé dans la session, en positionnant le paramètre de configuration `CSRF_USE_SESSIONS`, mais cet usage est moins générique puisqu'il ne pourrait pas s'appliquer à un site ouvert aux anonymes, au moins partiellement par exemple pour une page de contact.

La mise en évidence du besoin de présence de la balise de gabarit, qui n'y est actuellement pas, peut se faire facilement en provoquant un `POST` grâce aux outils de développement du navigateur :



© Editions ENI - All rights reserved

La trace du serveur de développement dit la même chose succinctement :

```
Forbidden (CSRF token missing or incorrect.): /messenger/write/
[...] "POST /messenger/write/ HTTP/1.1" 403 2513
```

La ligne de trace est due à l'enregistreur nommé `django.security.csrf`. Il n'a pas de configuration particulière et donc, par propagation, celle de la racine s'applique avec une sortie sur la console à partir du niveau `INFO`.

Les cas de rejet de la requête se traduisent par une réponse dont le statut (5xx, 4xx, etc.) fixe le niveau de gravité de la trace, mais toujours égal ou supérieur à `INFO`, ce qui explique l'apparition de la ligne.

Le sujet se réglera proprement dans le chapitre suivant dédié aux gabarits.

sensitive_post_parameters

En cas d'erreur, un rapport est construit pour aider le développeur à localiser sa cause. C'est typiquement ce qu'on voit dans les pages à fond jaune lorsqu'on tourne en mode `DEBUG`. Il est notamment présenté la pile des appels avec leurs variables locales, les attributs de l'objet requête et le listing complet des paramètres de configuration. Mais certaines informations à caractère sensible y sont tout de même masquées, dont celles ayant trait à la sécurité : mots de passe, secrets, algorithmes, etc. Leurs contenus sont alors remplacés par une suite de caractères étoile. Par exemple :

```
EMAIL_HOST_PASSWORD         '**********'
```

En mode production, les rapports ne sont plus présentés à l'écran, mais, en général et avec la configuration par défaut, ils sont émis par courriel aux administrateurs du site. La confidentialité de certaines informations peut être mise en cause par cette diffusion. Il est alors préférable de marquer ces informations pour qu'elles soient filtrées dans les rapports. Ce décorateur est prévu pour réaliser ce travail sur les paramètres soumis dans une requête `POST`.

La notion de « caractère sensible » aux informations doit être considérée au sens large et ne se réduit pas seulement à un critère de sécurité. Il faut y inclure les données relatives à la vie privée, celles dites à « caractère personnel », et plus précisément ici le secret de la correspondance privée.

Ainsi, le sujet et le corps (au minimum, car il faudrait probablement aussi prendre l'identité des correspondants) sont des contenus confidentiels et n'ont aucunement vocation à paraître dans un rapport.

Un essai en mode production, non détaillé ici, montrerait l'extrait suivant dans le courriel envoyé à l'occasion d'une erreur de serveur de type 5xx :

```
[...]
Request information:
USER: foo

GET: No GET data

POST:
csrfmiddlewaretoken =3D'uZljmv8xFBYlPn3odUpTFYQhmDZCDTJzRS3G[...]'
subject =3D '**********'
body =3D '**********'
recipient =3D '1'
[...]
```

Les deux paramètres sont effectivement masqués et leur confidentialité ainsi respectée.

Concernant maintenant la création d'une instance d'un message, si effectivement le formulaire apporte certains champs et si d'autres attributs sont pourvus d'une valeur par défaut, il reste néanmoins un attribut qui n'entre pas dans ces catégories : l'expéditeur. Il est requis et sans choix puisqu'il faut que ce soit l'utilisateur connecté. Sa valeur doit donc être imposée par le code et pour cela il faut trouver une méthode à surcharger. Dans les vues et greffons génériques et avec une classe de formulaire générée automatiquement à partir du modèle, la création de l'enregistrement est engagée sur un appel ordinaire à la méthode `save()` du formulaire. Dans l'état actuel du code de la vue, cet appel est voué à l'échec à cause de l'absence de l'expéditeur. Pour preuve, voici un morceau de trace d'un essai forcé :

```
Internal Server Error: /messenger/write/
Traceback (most recent call last):
 File "D:\Python3[...]\db\backends\utils.py", line 85, in _execute
   return self.cursor.execute(sql, params)
psycopg2.IntegrityError: ERREUR:  une valeur NULL viole la
 contrainte NOT NULL de la colonne « sender_id »
DETAIL:  La ligne en échec contient (14, s, , 20[...], 1, null,
 null, null, null)
```

© Editions ENI - All rights reserved

De façon générale, il existe deux options pour remédier à un déficit de champs sur un formulaire : brider la sauvegarde pour agir après coup et la finaliser, ou agir avant la sauvegarde. Pour mieux les comprendre et les comparer, ces deux stratégies vont être présentées sous la forme de trois techniques d'implémentation.

Solution 1 : intervenir en aval

Le principe reste d'appeler la méthode de sauvegarde sur le formulaire, mais avec un paramètre complémentaire : `save(commit=False)`. Le travail est réalisé à ceci près que l'étape d'insertion en base de données est sautée. L'objet créé est rendu et n'a d'existence qu'en mémoire ; charge à l'appelant de provoquer sa réelle écriture en base, si et quand il le souhaite.

La surcharge doit influencer la méthode du générique qui fait appel à la sauvegarde. L'identification du composant en cause dans la chaîne d'héritage se fait à l'aide de la documentation (répartie entre les vues et les greffons) ou en fouillant dans les sources. On localise dans le cas présent que la sauvegarde est portée par le greffon `ModelFormMixin`, dans sa méthode `form_valid()` qui, comme son l'indique, apporte son action après que le formulaire a été jugé valide. Il faut veiller à s'insérer en respectant la signature et les affectations d'attribut de la méthode originale :

```
def form_valid(self, form):
    self.object = form.save(commit=False)
    self.object.sender = self.request.user
    self.object.save()
    return super().form_valid(form)
```

L'objet rendu par le formulaire avec sauvegarde différée peut ainsi être retravaillé. Ici, il est complété en imposant l'expéditeur, avant d'être définitivement sauvegardé en base.

L'appel à `super()` assure la continuité de la chaîne de traitement dans le maillage des vues et greffons. Cette suite est nécessaire, car elle fournit la réponse HTTP.

Cette formulation a le mérite de pouvoir fonctionner, mais avec un excédent qui peut passer inaperçu si on ne pousse pas la réflexion dans les moindres détails. En effet, cette surcharge n'est qu'un ajout de code et ne supplante pas le traitement générique. Tel qu'il est écrit, l'appel à `super()` maintient l'exécution du `form.save()` générique quelque part dans la hiérarchie d'héritage. On pourrait craindre la création d'un objet en doublon. Telle qu'est faite l'implémentation du formulaire, il se trouve qu'à la première sauvegarde l'instance créée est conservée au sein du formulaire et donc employée à nouveau à la seconde sauvegarde. Un seul et même objet est manipulé, d'abord en création, puis en mise à jour.

Une fois de plus, l'activation de traces est un bon révélateur :

```
[...] django.db.backends_DEBUG - (0.010) INSERT INTO
 "messenger_message" ("subject", "body", "sender_id",
 "recipient_id", "sent_at", "read_at", "sender_deleted_at",
 "recipient_deleted_at")
 VALUES ('s', '', 2, 1, '20[...]'::timestamp, NULL, NULL, NULL)
 RETURNING "messenger_message"."id";
 args=('s', '', 2, 1, datetime.datetime(20[...]), None, None, None)
[...] django.db.backends_DEBUG - (0.005) UPDATE  "messenger_message"
 SET "subject" = 's', "body" = '', "sender_id" = 2,
 "recipient_id" = 1, "sent_at" = '20[...]'::timestamp, "read_at" =
 NULL, "sender_deleted_at" = NULL, "recipient_deleted_at" = NULL
 WHERE "messenger_message"."id" = 14;
 args=('s', '', 2, 1, datetime.datetime(20[...]), 14)
```

L'ordre `UPDATE` est un superflu à écarter. Pour l'éviter, le raccordement dans la chaîne doit viser l'ancêtre du composant porteur, par une mention explicite dans l'appel à `super()` :

```
[...]
from django.views.generic.edit import ModelFormMixin
[...]
   def form_valid(self, form):
       [...]
       return super(ModelFormMixin, self).form_valid(form)
```

On remarquera que cette manière de faire est viable ici, car il n'y a pas de traitements portés par une méthode de ce nom dans des composants antérieurs dans la chaîne et qu'il aurait fallu préserver.

© Editions ENI - All rights reserved

Solution 2 : intervenir en amont

Des attributs d'instance de modèle, pourtant requis, restent non valorisés parce que cette instance est automatiquement initiée de façon vierge, par le traitement qui en a besoin, typiquement le formulaire au moment de sa création. Le principe de cette solution est donc de contourner cette instanciation par défaut en fournissant d'entrée une instance préparée au préalable avec les valeurs souhaitées. Il est naturel pour une vue de création comme `CreateView` de ne pas compter sur l'existence préalable d'une instance d'un objet, mais on comprend aisément l'inverse pour une vue de mise à jour comme `UpdateView`. Or, ces deux vues partagent une large base de composants en commun, ce qui fait que la fourniture d'une instance de modèle, nécessaire pour l'une, est simplement neutralisée dans l'autre et on devrait pouvoir facilement en tirer profit.

La passation d'une instance de modèle se fait à la construction du formulaire, avec un paramètre de mot-clé `instance`. L'observation de cette construction dans les classes génériques révèle que tout est déjà prévu et que ce paramètre est alimenté pour peu que la vue ait un attribut nommé `object`. Dans le cas de `CreateView`, cet attribut est mis par défaut à `None`. Il suffit alors de positionner cet attribut avec une véritable instance pré-renseignée et tout le reste suivra.

L'injection de l'instance personnalisée peut se faire à plus d'un endroit : `get_form()`, `get_form_class()` ou `get_form_kwargs()`. Intervenir au niveau des méthodes plus fines n'a pas de bénéfice et nuirait plutôt à la compréhension, alors il est préférable de s'intercaler dans le premier point d'entrée :

```
def get_form(self, form_class=None):
    self.object = Message(sender=self.request.user)
    return super().get_form(form_class)
```

Cette méthode est aussi appelée à l'occasion du `GET`, pour alimenter les données de contexte avec le formulaire à rendre sur la page. La présence d'un objet instancié reste neutre.

Si ce n'était pas le cas, il serait possible de conditionner l'injection avec un test tel que :

```
if self.request.method in ('POST', 'PUT'):
```

Avec cette implémentation, l'attribut `model` de la vue n'est plus indispensable, car les éléments génériques tirent l'information nécessaire de l'attribut `object`, prioritairement analysé :

```
class WriteView(CreateView):
    [...]
    # model = Message
    [...]
```

Solution 3 : intervenir au milieu

Cette solution se situe entre les deux premières : on laisse le formulaire instancier lui-même un objet vierge et on va quand même y toucher avant de le laisser faire la sauvegarde. Cette manière paraît un peu illicite, car elle repose sur la connaissance de l'implémentation interne des formulaires pour modèle.

On peut quand même l'estimer officielle, car même si elle est extrêmement rare dans la documentation, un exemple y est présent. La clé est de savoir que l'objet construit par le formulaire est porté par l'attribut nommé `instance` et rien n'empêche d'intervenir dessus. La surcharge se place au dernier moment avant la sauvegarde de l'objet :

```
    def form_valid(self, form):
        form.instance.sender = self.request.user
        return super().form_valid(form)
```

Quelle solution choisir ?

De manière générale pour les formulaires, la préférence va au principe de la solution 1, c'est-à-dire obtenir d'abord l'instance du nouvel objet de la part du formulaire, mais sans écriture en base de données, et de faire ensuite des compléments. Il est ainsi possible de modifier à volonté l'objet, et éventuellement d'affecter certains de ses attributs en fonction d'autres attributs issus du formulaire. Cela suppose d'être maître de l'agencement du code, ce qui est facilement le cas pour des vues propres à l'application, mais devient plus difficile avec des vues génériques, où par nature la liberté de personnalisation est moindre. On l'a bien vu dans la solution exposée, avec la nécessité de remplacer du code plutôt que d'en ajouter. C'est pourquoi, dans le cas présent, la solution 3 semble mieux convenir, car plus lisible.

© Editions ENI - All rights reserved

Le dernier sujet à traiter concerne la réponse à rendre en cas de succès de création. Le comportement par défaut se voit si on fait une tentative dans un but de découverte :

```
Internal Server Error: /messenger/write/
Traceback (most recent call last):
 File "D:[...]views\generic\edit.py", line 116, in get_success_url
   url = self.object.get_absolute_url()
AttributeError: 'Message' object has no attribute
 'get_absolute_url'
[...]
django.core.exceptions.ImproperlyConfigured: No URL to redirect
 to.  Either provide a url or define a get_absolute_url method on
 the Model.
[...] "POST /messenger/write/ HTTP/1.1" 500 147625
```

Si rien d'autre n'est précisé, le code générique espère savoir vers quelle page rediriger le navigateur auprès de la méthode `get_absolute_url()` du modèle. Cette méthode n'a pas d'implémentation par défaut, et puisque rien n'a été fait jusqu'à présent, l'échec est attendu.

▶Complétez le modèle :

```
[...]
from django.urls import reverse
[...]
class Message(models.Model):
    [...]
    def get_absolute_url(self):
        return reverse('messenger:view', args=[self.pk])
```

Avec ceci, le minimum est en place pour que la vue de composition de message puisse fonctionner, avec en cas de succès la visualisation du message venant d'être envoyé. Ce n'est pas nécessairement la page la plus pertinente après un envoi. Dans ce cas, on peut indiquer quelle est la page souhaitée par l'attribut `success_url`. Il peut être plus opportun de revenir sur la page du dossier d'arrivée.

Complétez la vue :

`mysite\messenger\views.py`

```
[...]
from django.urls import reverse, reverse_lazy
[...]
class WriteView(CreateView):
    [...]
    success_url = reverse_lazy('messenger:inbox')
    [...]
```

L'emploi de `reverse_lazy()` plutôt que `reverse()` est nécessaire de façon à différer l'évaluation, car cette ligne est lue dès l'importation du fichier, soit à un moment où les URL ne sont pas encore entièrement chargées et accessibles.

La chaîne URL peut être personnalisée encore plus, car, avant son emploi, elle passe par une opération de mise en forme dont les arguments en mots-clés sont les attributs de l'objet créé. La chaîne de caractères peut donc être exploitée en tant que motif doté de champs de remplacement.

À titre d'illustration, supposons que la redirection doit se faire vers le dossier d'envoi, avec une mise en valeur de l'objet venant d'être créé. La vue du dossier doit être capable de distinguer cet objet parmi la collection des objets. Un moyen simple est de fournir son identifiant à l'aide d'un paramètre dans l'URL. Le nom de ce paramètre est par exemple `highlight`, et l'identifiant d'un objet message est `id`.

La formulation se complique un peu, mais il faut bien tout à la fois : conserver la déduction de l'URL à partir du nom de la vue, faire un complément de chaîne de caractères et respecter l'aspect différé de l'opération :

```
[...]
from django.utils.functional import lazy
[...]
    success_url = lazy(
        lambda: reverse('messenger:sent') + "?highlight={id}",
        str)()
```

© Editions ENI - All rights reserved

Si cela paraît plus lisible, et encore plus si on emploie ce montage plus d'une fois, on peut le rationaliser sous forme de fonction :

```
[...]
def reverse_with_id(paramname):
    return lazy(
        lambda viewname: "{}?{}={{id}}".format(
            reverse(viewname), paramname),
        str)
[...]
    success_url = reverse_with_id('highlight')('messenger:sent')
```

Ou avec une intégration plus poussée :

```
[...]
def reverse_with_id(paramname):
    query = "?{}={{id}}".format(paramname)
    return lazy(lambda viewname: reverse(viewname) + query, str)
reverse_highlight = reverse_with_id('highlight')
[...]
    success_url = reverse_highlight('messenger:sent')
```

9.8 Effacement de message

Cette vue porte un nom un peu trompeur, car en réalité rien ne va être effacé, mais seulement un marquage est apporté. En effet, le modèle de message a été conçu pour comporter des champs de marquage pour effacement, distribués à chacune des parties de l'échange.

▶Modifiez le fichier de routage :

`mysite\messenger\urls.py`

```
[...]
from .views import (InboxView, SentView,
    MessageView, WriteView, DeleteView,
[...]
urlpatterns = [
    [...]
    path('delete/', DeleteView.as_view(), name='delete'),
    [...]
```

La volonté d'effacer un message peut naturellement surgir à l'occasion de chaque présentation à l'écran de ce message, que ce soit en détail ou dans une liste. Il est préférable d'être en mesure d'y répondre de façon la plus immédiate. Ce serait alourdir inutilement la navigation que de devoir aller sur une page spéciale pour dans un second temps réaliser la sélection et l'action. Dans ces conditions, la vue n'a pas spontanément à servir une action `GET`. Elle n'est prévue que pour une action `POST`, charge aux autres vues de prévoir dans leur gabarit la capacité à initier cette requête, sans nécessairement se baser sur un traditionnel formulaire taillé pour les deux verbes.

Avec ce mode de fonctionnement, une vue générique telle que `DeleteView` ne va évidemment être d'aucun secours. D'autant plus qu'elle ne gère qu'un objet à la fois, ce qui n'est pas compatible avec le point suivant à propos de la gestion multiple.

Ce n'est pas confortable pour un utilisateur de devoir répéter une même action, individuellement pour chacun des objets d'un ensemble, là où il suffirait de demander l'action une seule fois de façon collective. La vue doit donc être conçue pour agir sur un ou plusieurs messages en une requête.

Parmi les vues génériques, la vue `FormView` est trop dédiée au processus en deux étapes d'un formulaire, avec validation et retour au formulaire en cas d'erreur de saisie.

Si on cherche quelque chose d'intéressant du côté des greffons, on voit `MultipleObjectMixin`, mais son but est de gérer de la pagination. `MultipleObjectTemplateResponseMixin` est fait pour de la présentation de liste, donc hors sujet. `DeletionMixin` est prévu pour traiter un objet unique.

En absence de mieux, il ne reste plus qu'à baser la vue sur le socle de base `View`.

▶Modifiez le fichier des vues pour créer le squelette de la nouvelle vue :

`mysite\messenger\views.py`

```
[...]
from django.http import HttpResponseRedirect
[...]
class DeleteView(View):
    http_method_names = ['post']
```

© Editions ENI - All rights reserved

```
    @csrf_protect_m
    @login_required_m
    def dispatch(self, *args, **kwargs):
        return super().dispatch(*args, **kwargs)

    def post(self, request, *args, **kwargs):
        pks = request.POST.getlist('pks')
        if pks:
            self._action(request.user, pks)
        return HttpResponseRedirect(request.META['HTTP_REFERER'])

    def _action(self, user, pks):
        """...à compléter..."""
```

Avec les mêmes raisons déjà données pour les vues précédentes, la méthode `dispatch()` porte les décorateurs de sécurité.

Le seul verbe à implémenter est le `POST`. La désignation des messages concernés se fait par la transmission de leur clé primaire dans un paramètre. Par choix, le nom du paramètre est mis au pluriel pour souligner le fait qu'il faut s'attendre à des valeurs multiples. L'objet `request.POST` est une instance de classe `QueryDict`, comme l'est `request.GET` pour les paramètres passés en URL, qui reproduit le comportement d'un dictionnaire, mais avec des extensions, notamment pour gérer de multiples valeurs sous une même clé puisque c'est un cas normal pour certains éléments de formulaire HTML tels que des boîtes de sélection ou des cases à cocher. Parmi les nombreuses méthodes de cette classe, `getlist()` est celle qui permet de récupérer l'ensemble des valeurs pour une clé, alors que `get(key)` ou `['key']` ne rendent que la dernière valeur. Par défaut, une liste vide est rendue si la clé n'existe pas.

La situation d'absence d'identifiants de messages est un cas particulier. D'un point de vue fonctionnel, ce cas n'a pas de sens et ne devrait pas arriver, mais il faut pourtant le supporter proprement. Selon la gravité qu'on veut lui donner, il peut soit faire l'objet d'une gestion d'erreur plus ou moins sévère (simple message d'avertissement à l'utilisateur, renvoi sur une page spéciale, etc.), soit être absorbé discrètement. Dans tous les cas, il est inutile d'engager l'action : au mieux, celle-ci n'aura aucun effet, tout en représentant au minimum un risque de coût de traitement ; au pire, elle n'est pas réalisable et c'est le cas ici comme on le verra expliqué plus loin.

Le traitement à réaliser est sous-traité à une méthode auxiliaire à usage privé, avec les paramètres nécessaires à l'accomplissement de son travail. Cette répartition n'est pas strictement nécessaire pour le code actuel. C'est un découpage en anticipation d'une probable introduction de greffons si le code est enrichi de commandes supplémentaires. En effet, il y a fort à parier que leur fonctionnement adoptera le même schéma, à savoir : récupérer une liste d'identifiants, agir dessus et rendre une réponse d'acquittement. On devine que la méthode `_action()` peut être logée dans un greffon personnalisé pour chaque commande ou famille de commandes et tout le restant de la mécanique peut être factorisé dans un greffon unique commun. Le retour arrière d'un effacement (`UndeleteView`) ou la mise en archive (`ArchiveView`) sont des exemples de futures commandes potentielles.

Une fois l'action réalisée, il ne reste plus qu'à donner un acquittement sous la forme d'une redirection. Cette forme est traditionnelle après un `POST`, selon les bonnes pratiques de développement web pour éviter le risque d'une nouvelle soumission intempestive à cause d'une navigation en arrière, car en procédant ainsi, la requête de soumission est supplantée dans l'historique du navigateur. La requête d'effacement ayant pu être initiée à partir de plus d'une page, quelconque mais significative, il semble inopportun de mener l'utilisateur dans tous les cas vers une même page imposée, probablement par nature générique et neutre. Pour débuter, le choix est fait de renvoyer l'utilisateur vers sa page de départ, information normalement accessible par un en-tête HTTP. On verra plus loin que ce comportement ne peut pas rester aussi simplifié et il devra s'adapter aux circonstances. Le dictionnaire `request.META` détient des entrées pour les en-têtes et d'autres entrées relatives aux variables d'environnement, positionnées par le système d'exploitation de la machine et par le serveur WGSI frontal. Avec quelques exceptions, la règle générale pour former les entrées d'en-tête est : une mise en caractères majuscules, le remplacement du trait d'union par le caractère de soulignement et l'ajout du préfixe `HTTP_`. Un en-tête `"Referer: http://[...]"` se traduit donc par une entrée avec la clé `HTTP_REFERER`.

© Editions ENI - All rights reserved

Dans cette commande d'effacement, l'action consiste à mettre à jour un des champs de marquage, sauf que dans la requête rien ne dit s'il s'agit de messages reçus ou envoyés pour déterminer sur lequel des champs agir. De toute façon, il ne faut pas compter sur cette classification puisque potentiellement la demande pourrait être exprimée à partir d'une sélection dans une liste de messages mixtes si on imagine une extension future de l'application avec la présence de dossiers auxiliaires, comme un dossier d'archivage. Prévoir deux ordres de mise à jour, avec les filtres adéquats, suffit pour contourner cette indétermination. De plus, il doit être prévu les protections contre des tentatives d'intervention sur des messages auxquels on n'a pas droit, par une transmission forgée de clés illégitimes.

▶Modifiez le fichier des vues pour l'implémentation de l'action :

`mysite\messenger\views.py`

```
[...]
    def _action(self, user, pks):
        at = now()
        recipient_rows = Message.objects\
            .filter(pk__in=pks,
                recipient=user,
                recipient_deleted_at__isnull=True)\
            .update(recipient_deleted_at=at)
        sender_rows = Message.objects\
            .filter(pk__in=pks,
                sender=user,
                sender_deleted_at__isnull=True)\
            .update(sender_deleted_at=at)
        if not (recipient_rows or sender_rows):
            raise Http404
```

Une variable de travail `at`, en n'appelant qu'une fois l'horloge, assure d'avoir un même instant pour chaque besoin de tampon temporel dans le traitement. Ici, ce n'est pas une nécessité fonctionnelle, mais le respect d'un principe de bon codage.

Chacune des requêtes SQL est bâtie sur le même schéma, en l'occurrence positionner le champ de marquage dans les enregistrements satisfaisant à l'ensemble des critères suivants :

- 1. Être parmi les messages ciblés par la demande d'effacement.
- 2. Se limiter à un côté de l'échange : soit expéditeur, soit destinataire.
- 3. Ne pas déjà être effacé.

Le critère n°1 met en œuvre l'opérateur `in`, qui juge dans son étape de construction initiée par l'évaluation de la requête, donc sans même parler de son exécution, que par nature aucun résultat ne peut être espéré avec une liste vide en paramètre. Cela se traduit par la levée d'une exception `EmptyResultSet`. Voilà ce qui justifie le besoin en amont de ne pas laisser passer ce genre de situation, ce qui a été implémenté plus tôt dans l'appelant avec une condition de validité sur `pks`.

Le critère n°2 sert aussi de protection contre une attaque qui viserait des messages n'appartenant pas à l'utilisateur.

Le critère n°3 écarte l'éventuel cas inutile de remplacement d'une marque par une autre.

La méthode `update()` rend le nombre d'enregistrements candidats à la mise à jour. Si on écarte le cas improbable de vouloir effacer des messages déjà effacés, le fait de n'avoir aucun message d'un côté ou de l'autre de l'échange peut être considéré comme suffisamment anormal pour provoquer une réponse négative comme une page de type 404 *Not Found*. Le cas est plutôt révélateur d'une attaque avec des `pks` forgés.

L'implémentation actuelle de la vue a ce qu'il faut pour fonctionner, mais un point faible doit être retravaillé : celui de l'adresse de redirection en fin de traitement. Le travail concerne trois thèmes : l'absence de l'information, la sécurisation et la disparition de la cible de retour.

© Editions ENI - All rights reserved

La non-communication par un navigateur de l'en-tête `Referer` est un comportement admis par le protocole HTTP. Par exemple, le paramètre de configuration `network.http.sendRefererHeader` du navigateur Firefox permet de limiter ou d'interdire l'envoi de cet en-tête. Cela risque cependant de nuire au fonctionnement face à certains sites. C'est le cas avec Django qui renforce ses contrôles de sécurité lorsque le protocole est HTTPS, en vérifiant la présence de l'en-tête et la cohérence de son contenu face à l'en-tête `Host` ou éventuellement d'autres sources déclarées autorisées par le paramètre de configuration `CSRF_TRUSTED_ORIGINS`. Mais peu importe les raisons, il faut dans tous les cas avoir un fonctionnement propre en cas d'absence de cet en-tête et donc prévoir une vue de repli par défaut. Le code de gestion de l'en-tête peut tout de suite être factorisé dans une fonction pour plusieurs raisons : 1) elle a de bonnes chances d'être utilisée à d'autres occasions ; 2) elle permet d'y placer la sécurisation abordée ensuite ; 3) la séquence priorisée de replis, exprimée en opérateurs logiques, est plus facile à percevoir.

Le dossier d'arrivée convient pour la vue de repli et il est plus confortable de mentionner cette vue par son nom. La logique combinatoire sur des chaînes de caractères se gère facilement, mais le problème est que la réponse `HttpResponseRedirect` attend une URL en paramètre. Le besoin d'exprimer la cible d'une redirection sous différentes formes est suffisamment récurrent pour avoir donné lieu à la disponibilité d'un raccourci. Dans le module `django.shortcuts`, on trouve la fonction `redirect()`, avec un premier paramètre qui supporte la désignation de la cible de trois façons : 1) en nom de vue ; 2) en URL absolue ou relative ; 3) en instance de modèle, avec l'emploi de son `get_absolute_url()`. Les variantes n°1 et n°2 correspondent à la situation présente.

▶Modifiez le fichier des vues pour la première part de l'amélioration :

`mysite\messenger\views.py`

```
[...]
# from django.http import HttpResponseRedirect
from django.shortcuts import redirect
[...]
    def post(self, request, *args, **kwargs):
        next_url = _get_referer(request) or 'messenger:inbox'
        [...]
        return redirect(next_url)
```

Le deuxième thème a trait à la sécurité : ce n'est pas parce qu'un en-tête est donné qu'il faut le croire aveuglément. La sollicitation de la vue d'effacement est censée être exclusivement initiée à partir d'une page du site, jamais d'un site externe. Il n'y a donc aucune légitimité à sortir du site à l'occasion de la redirection d'acquittement et toute tentative dans ce sens doit être filtrée. Une solution simple consiste à dégrader l'URL vers une forme relative, par élimination des portions protocole et domaine.

▶Modifiez le fichier des vues pour la deuxième part de l'amélioration :

`mysite\messenger\views.py`

```
[...]
from urllib.parse import urlsplit, urlunsplit
[...]
def _get_referer(request):
    if 'HTTP_REFERER' in request.META:
        sr = urlsplit(request.META['HTTP_REFERER'])
        return urlunsplit(
            ('', '', sr.path, sr.query, sr.fragment)
        )
```

Désormais, un exemple d'observation de trafic pour une soumission à la vue montrerait les en-entêtes suivants.

Pour la requête :

```
Host: localhost:8000
[...]
Origin: http://localhost:8000
Referer: http://localhost:8000/messenger/sent/
[...]
```

Pour la réponse, issue de la redirection :

```
HTTP/1.1 302 Found
[...]
Location: /messenger/sent/
[...]
```

© Editions ENI - All rights reserved

Le troisième point concerne l'incapacité de revenir à son point de départ après la réalisation de l'action. En effet, la page à l'initiative de l'effacement et référencée en `Referer` peut être dédiée à l'instance et n'a alors plus d'existence. C'est le cas avec la page de lecture de message, où un bouton sera proposé pour effacer le message actuel.

La solution choisie est de permettre optionnellement d'indiquer une « URL de suite » lors de la soumission d'effacement. Cette information est transmise dans un paramètre de l'URL et le nom choisi est `next`. Elle est récupérable d'une façon similaire aux paramètres `pks` mais dans l'attribut `GET` :

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

Si la clé n'est pas présente, le retour est `None`. Comme pour l'information de `Referer`, cette information d'entrée doit être sécurisée, d'où la création d'une fonction par clarté et éventuelle réutilisation.

▶Complétez le fichier des vues avec cette fonction :

`mysite\messenger\views.py`

```
[...]
def _get_referer(request):
[...]

def _get_safe_internal_url(urlstring):
    if urlstring:
        sr = urlsplit(urlstring)
        return urlunsplit(
            ('', '', sr.path, sr.query, sr.fragment)
        )
```

La priorité est donnée à cette adresse de redirection lorsqu'elle est fournie.

▶Modifiez le fichier des vues :

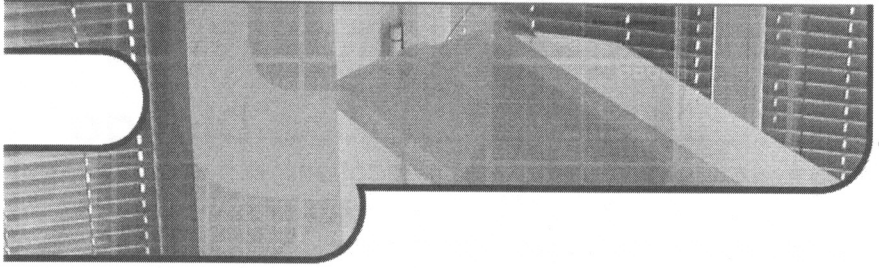
`mysite\messenger\views.py`

```
    def post(self, request, *args, **kwargs):
        [...]
        if pks:
            [...]
            return redirect(
                _get_safe_internal_url(request.GET.get('next'))
                or next_url)
        else:
            return redirect(next_url)
```

Une branche `else` est ajoutée, puisqu'en cas d'absence d'identifiants de messages, l'action n'est pas engagée et il n'y a pas de raison de faire autrement que la redirection ordinaire par défaut.

Au terme de ce chapitre, l'ancienne vue `index()` n'est plus utilisée. Elle peut être mise hors service par une mise en commentaire. De même, son importation dans `urls.py` peut être commentée.

© Editions ENI - All rights reserved



Chapitre 8
Pages et gabarits

1. Introduction

Le chapitre précédent, relatif aux vues, a montré un travail portant essentiellement sur la donnée entrante ou interne, sur de nombreux aspects : sécurité, contrôle, validité, stockage, actions, etc., jusqu'à le plus souvent rassembler de la donnée sortante. Dans les situations de rendus purement techniques, la production de la sortie est en général totalement prise en charge par le code de la vue, éventuellement avec l'aide d'assistants de mise en forme. L'exemple typique est un ensemble de vues pour servir une API, avec des échanges au format JSON. Un autre usage similaire a été expérimenté avec les vues en mode AJAX, qui rendaient aussi dans ce cas du JSON.

Mais l'usage le plus courant est de vouloir produire des pages au format HTML de façon dynamique, c'est-à-dire que la page est construite à la volée au moment de la demande. Pour autant, de larges parts d'une page sont de nature statique, donc servies sans variation possible, et se retrouvent à l'identique sur plusieurs pages de façon à présenter un site cohérent au niveau parcours et sur le plan graphique. Une infrastructure logicielle digne de ce nom se doit de rationaliser ce genre de composition et ne pas laisser le développeur s'embourber dans un codage inextricable formé de sorties de miettes de HTML par des instructions du genre `print` ou `echo`.

Le principe du gabarit est commun pour apporter une solution élégante au besoin : le contenu d'un fichier est exprimé prioritairement de façon statique, dans le format de sortie voulu, et en complément sont disséminées des balises localisant des remplacements et des comportements dynamiques tels qu'une boucle ou une logique conditionnelle. Bien sûr, tout le travail de fusion, composition et assemblage doit être confié à un moteur de rendu.

2. Moteurs

L'emploi de plus d'un moteur de gabarit au sein d'un projet est supporté. On a quand même plutôt tendance à éviter la dispersion et à rester uniforme en faisant le choix d'un moteur unique.

2.1 Moteurs intégrés

Historiquement, Django était fourni avec son seul moteur de gabarit maison, dénommé *Django Template Language* (parfois désigné par le sigle DTL). En version 1.8 est arrivée l'ouverture à l'intégration d'autres moteurs ainsi que le support d'un moteur alternatif : Jinja2 (ou Jinja version 2, en distinction d'une version 1).

Jinja est un moteur de rendu de gabarit, écrit en Python, inspiré de celui de Django, mais issu d'une communauté autre que la *Django Software Foundation*. Il revendique de donner aux auteurs de gabarits plus de fonctionnalités et d'outils, et d'être plus rapide, notamment par le fait de produire du code Python compilé lors du premier usage, bénéfique aux usages suivants. C'est un moteur pour Python et donc il ne se réduit pas à un usage sous Django.

La popularité de Jinja et ses performances ont fait naître des envies chez certains de le voir supplanter le moteur DTL. Le sujet a été plusieurs fois évoqué, discuté, évalué dans ses pour et contre, et remis en sommeil (pour les raisons récurrentes à ce genre de débat : le manque de ressources humaines pour s'atteler à la réalisation ; les arbitrages avec d'autres projets).

© Editions ENI - All rights reserved

Les discussions n'ont pas été vaines puisqu'elles ont fait ressortir des objectifs et des points à respecter et ont finalement incité un développeur motivé (et aidé d'un financement participatif) à prendre l'initiative d'une réalisation sur ces propositions d'engagement :

- Ne pas remplacer ni déprécier DTL (respect de ses partisans, base historique), mais admettre des moteurs alternatifs optionnels, avec Jinja2 pour ouvrir la voie (mais pas d'autre candidat à l'époque). En corollaire, DTL reste le moteur par défaut. Ce principe de composants à la carte se veut dans le même esprit que ce qui se fait déjà pour d'autres genres de moteurs de l'infrastructure logicielle : bases de données, maintien de session HTTP, serveur de fichiers statiques, serveur de cache, envoi de courriels, authentification, etc.
- Établir une API permettant une intégration facilitée de moteurs tiers (intégration ne signifie pas inclusion, cf. paragraphe suivant).
- Supporter l'emploi de multiples moteurs dans un même projet. Non seulement cela laisse une plus grande liberté de choix au développeur, mais facilite aussi la planification d'une migration progressive d'un moteur vers un autre.

Il est plus exact de parler de support intégré que de moteur intégré, car s'agissant d'un produit externe et optionnel, il ne fait pas partie du paquet Django, ni en tant que dépendance ni en tant que copie embarquée. Son installation en tant que paquet Python doit se faire indépendamment, de façon ordinaire (`pip install`).

Le choix de tel ou tel moteur de rendu peut naturellement être orienté par des préférences personnelles et subjectives de la part des intervenants du projet. Il est seulement attiré l'attention du lecteur sur un aspect parfois négligé face aux attraits purement techniques : une dépendance à un produit externe est introduite dans le projet et, par conséquent, aussi un risque relatif à la pérennité de ce produit.

Par exemple, le sujet apparaît dans les discussions pour savoir s'il était opportun de remplacer définitivement DTL par un moteur tiers, car on peut en penser du bien ou du mal : certains argueront que c'est tout bénéfice de se reposer sur les développeurs d'un produit tiers pour transférer la charge de maintenance et se contenter de leur refiler les tickets de dysfonctionnement, d'autres objecteront qu'on fragilise la pérennité de son projet en le liant à un autre qui peut-être ne tient qu'à une seule personne (pour s'en faire une idée, voir les statistiques relatives aux contributeurs et l'intensité de leur activité dans l'entrepôt du produit).

Dans le cas présent, pour rester basique et sur le terrain le plus employé, la suite du chapitre présente des gabarits écrits pour le moteur interne de Django.

2.2 Moteurs personnalisés

Les évolutions apportées à l'infrastructure logicielle à l'occasion du support d'un second moteur intégré ont ouvert la possibilité de mettre en jeu des systèmes de gabarit tiers. Que ce système soit une implémentation entièrement privée ou qu'il soit un produit public et reconnu, le principe reste le même et passe par le respect d'une API par l'implémentation d'un adaptateur (aussi appelé *backend*).

Dans des recherches d'alternatives à DTL, il est difficile de faire le tri entre les produits qui adhèrent au nouvel interfaçage standard introduit avec Django 1.8 et ceux qui en sont restés aux seules anciennes façons, plus ou moins intrusives et qui procèdent souvent par des substitutions. Comparer les dates d'activité du produit par rapport à la sortie de la version 1.8, avril 2015, est déjà un indice pour refroidir quelques espérances.

© Editions ENI - All rights reserved

2.3 Sélection du moteur

Les moteurs admis par le projet sont déclarés dans la configuration sous le paramètre `TEMPLATES`. Son contenu est une liste, ce qui induit que l'ordre de citation a son importance : l'algorithme de chargement d'un gabarit soumet la proposition successivement à chaque moteur jusqu'à ce qu'un moteur se déclare compétent pour prendre en charge la demande.

Il n'est pas prévu de convention particulière dans le nom ou le type d'extension du gabarit qui permettrait de donner un indice au chargeur ou pour l'orienter selon des préférences.

Au mieux, il existe dans les fonctions impliquées un paramètre optionnel `using` pour limiter la recherche à une instance de moteur particulière, mais son usage est pensé pour des cas qui sortent de l'ordinaire et il est assez contraignant puisque cela suppose d'intervenir au niveau du code de la vue, ce qui ne convient pas aux applications tierces, par exemple.

Puisque le nom du gabarit ne permet pas à lui seul à savoir à quel moteur il est destiné, sa localisation dans le système de fichiers apporte la solution au problème : si chaque moteur explore des répertoires dédiés sans qu'il y ait de lieu commun, chacun est capable de savoir s'il détient le gabarit ou pas. Cela ne signifie pas une absence de collision, mais la situation est alors voulue : une application réutilisable peut mettre à disposition ses gabarits sous plusieurs déclinaisons et laisser le choix à l'intégrateur. Ce choix trouve son expression dans l'ordre adopté dans la configuration; le seul souci éventuel est que ce choix s'applique globalement à toutes les applications du site et ne peut pas être individualisé par application.

Le cloisonnement des répertoires de gabarits est naturellement en place sous l'arborescence d'une application du fait d'une convention de nommage : DTL conserve son nom historique `/templates/`, tandis que pour les autres moteurs, il est simplement employé leur nom. Ainsi, au moteur Jinja2 est attribué le nom de sous-répertoire `/jinja2/`.

Si cela était mis en œuvre pour l'application, on aurait une structure de ce genre :

```
mysite\
├─ [...]\
├─ messenger\
│  ├─ fixtures\
│  ├─ jinja2\
│  │  └─ messenger\
│  ├─ migrations\
│  └─ templates\
│     └─ messenger\
└─ [...]\
```

De même, il faudrait prévoir une répartition des gabarits globaux dans des répertoires spécialisés par moteur. Le choix des emplacements est libre, mais sachant que l'habitude antérieure était d'employer un répertoire `/templates/` sous la racine du projet, deux options viennent naturellement à l'esprit :

- Option 1 : reproduire le même schéma que sous les applications, avec des répertoires parallèles :

```
mysite\
├─ [...]\
├─ jinja2\
├─ [...]\
├─ templates\
└─ [...]\
```

- Option 2 : faire des branches sous le lieu habituel :

```
mysite\
├─ [...]\
├─ templates\
│  ├─ jinja2\
│  └─ django\
└─ [...]\
```

On peut aussi vouloir omettre le sous-répertoire `/django/` pour ne rien changer à l'emplacement conventionnel des gabarits DTL.

© Editions ENI - All rights reserved

3. Principes de fonctionnement

Le système de gabarits a été pensé pour répondre d'abord à des objectifs de présentation, ensuite à une certaine dose de logique dédiée à la présentation, mais pas pour héberger de la logique métier. C'est une volonté manifestée de ne pas permettre d'injecter directement du code métier à partir d'un gabarit, comme certains autres systèmes/langages le permettent avec une alternance continuelle entre du contenu de sortie et des instructions en langage de programmation. Il est estimé que l'écriture d'un gabarit est confiée à un concepteur, pas à un programmeur. Donc il n'est pas prévu de balisage pour basculer temporairement vers du code Python, ni d'inventer un pseudo-langage en imitation.

Par contre, l'utilité est reconnue d'avoir la possibilité de formuler des logiques, mais devant rester très basiques, comme des conditions ou des répétitions, et toujours justifiées par des décisions de présentation. En d'autres termes, tout le travail d'élaboration doit être réalisé dans la couche vue, et la couche gabarit doit se limiter à en exposer le fruit. Le revers de cette rigidité est que parfois on est amené à placer au niveau de la vue des traitements purement à but d'affichage, faute de pouvoir faire autrement.

La sécurité est une raison supplémentaire pour ne pas permettre l'exécution directe de code dans une couche de présentation. Cela ferme la porte aux tentations d'attaques dites par injection.

La cohabitation entre le contenu statique de sortie et les directives de mise en forme est formalisée de façon traditionnelle par du balisage. Une bibliothèque standard de balises de gabarit (*template tags*) et de filtres est disponible, pour couvrir les besoins les plus communs.

L'extensibilité du système n'est pas oubliée, puisqu'il est possible de créer des balises de gabarit et des filtres personnalisés.

Les concepteurs veillent à ne pas dériver vers des capacités de programmation pour freiner l'introduction de trop de logique dans une couche de présentation. Avec ce raisonnement, on pourrait s'étonner de constater la présence d'un filtre intégré nommé `add`, qui permet une opération mathématique de base. En effet, mais il s'agit d'une entorse historique qui remonte très loin dans le temps, avant même la libération du code source. Cet élément n'a pas pu être rendu obsolète par la suite, car cela aurait provoqué beaucoup de tort en matière de compatibilité comparé à un maigre bénéfice. Depuis longtemps, la proposition d'ajout d'un tel filtre dans la collection des filtres intégrés n'aurait aucune chance d'être acceptée, alors n'espérez pas voir ou proposer un filtre comme `sub` pour une opération de soustraction, qui pourtant semble tout autant légitime. De façon plus générale, tout élément à but mathématique n'a pas vocation à faire partie de la collection intégrée, sous prétexte que sa place normale est dans la vue. Savoir si une manipulation doit être logée dans la vue ou dans le gabarit est un débat récurrent parce que la frontière n'est pas évidente dans certains cas. C'est parfois un reproche qui est fait à DTL de ne pas offrir suffisamment de possibilités et de devoir se replier sur le développement de ses propres extensions pour compenser.

4. Pages non dynamiques

Tout ce qu'un site délivre n'a pas nécessairement à être servi par des vues. Il existe d'autres moyens mieux adaptés lorsque le contenu est statique. Les sections suivantes mentionnent certains de ces moyens, succinctement car il s'agit juste d'en faire tout de suite la distinction avec la composition dynamique de pages, sujet principal du chapitre.

4.1 Application flatpages

Le module `django.contrib.flatpages` est un outil mis à disposition pour répondre à un cas d'usage courant : servir des pages stables, établies d'avance et à lecture seule, telles que « Mentions légales » ou « Conditions d'utilisation ».

© Editions ENI - All rights reserved

Comme son nom l'indique (« page plate » en traduction littérale), les pages concernées doivent rester très simples, avec pour but de délivrer un contenu informatif, sans interaction.

L'application a manifestement des avantages par rapport à l'emploi de simples fichiers statiques :

- Les pages sont supportées par la base de données. Une page correspond donc à une instance d'un modèle, avec des champs : non seulement un corps HTML, mais aussi un titre HTML. Un autre bénéfice considérable d'avoir ce stockage en base est l'ouverture à une administration de contenu directement par des gestionnaires de contenu, dans un esprit CMS (*Content Management System*), plutôt que de devoir passer par des administrateurs système ou mettre en jeu une circuiterie sur mesure. Le module d'administration intégré peut suffire à la gestion du cycle de vie de ces pages.
- L'URL associée à la page peut être choisie à volonté.
- Le système de gabarit reste mis en œuvre, ce qui assure le respect d'une charte graphique cohérente pour un ensemble de pages, éventuellement différente par rapport au reste du site.
- Un drapeau permet de restreindre la visibilité de la page aux seuls utilisateurs authentifiés.

Cette application requiert l'installation de l'application `django.contrib.sites`. Cette dépendance a son explication dans l'ancienneté de ces applications : l'initiateur de l'infrastructure logicielle était le service informatique d'un éditeur de presse (journaux papier), détenant deux titres et leurs sites respectifs. Certains articles avaient pourtant vocation à être mis en commun et donc diffusés sur les deux sites. Plutôt que de se résigner à faire de la duplication, une solution plus efficace a été de gérer une base d'information commune et d'établir des relations entre les objets éditoriaux et des objets représentant les sites. Naturellement, selon la même idée que pour des articles, les `flatpages` sont prévues pour devoir être reliées à un ou plusieurs sites.

Au fil du temps, la fonctionnalité multisites a perdu de son importance et il en a été tenu compte avec un remaniement du code dans la version 1.7 permettant de référencer la fonction `get_current_site()` et d'obtenir un objet de la classe `RequestSite`, sans pour autant obliger à l'installation de l'application `sites`. Ainsi, les applications réutilisables modernes peuvent être conçues pour être compatibles multi-sites sans imposer un impact sur la base de données.

4.1.1 Exemple d'usage

Une simple page pour « Mentions légales va servir d'illustration à la mise en œuvre de l'application.

- Installez l'application et sa dépendance dans la configuration, et donnez l'identifiant du site, a priori avec la clé primaire 1 :

`mysite\settings.py`

```
[...]
SITE_ID = 1

# Application definition

INSTALLED_APPS = [
    [...]
    # 'django.contrib.staticfiles',
    'django.contrib.sites',  # dépendance de flatpages
    'django.contrib.flatpages',  # démo
    'mysite.main',
    [...]
]
```

- Passez une migration, pour les applications nouvellement activées :

```
D:\dj>py manage.py migrate
Operations to perform:
 Apply all migrations: admin, auth, contenttypes, flatpages, messenger,
sessions, sites
Running migrations:
 Applying sites.0001_initial... OK
 Applying flatpages.0001_initial... OK
 Applying sites.0002_alter_domain_unique... OK
```

© Editions ENI - All rights reserved

Les tables créées sont :

- `django_flatpage`
- `django_flatpage_sites`
- `django_site`

La table de `sites` est pré-remplie avec un enregistrement par défaut, avec la clé fournie dans la configuration (valeur 1 si absente) :

```
>>> from django.contrib.sites.models import Site
>>>
>>> [vars(s) for s in Site.objects.all()]
[{[...], 'id': 1, 'domain': 'example.com', 'name': 'example.com'}]
```

L'application est capable de faire son travail selon deux politiques. L'une est plutôt basée sur le rattrapage d'échecs en 404 *Not Found* par son intergiciel d'interception `FlatpageFallbackMiddleware`. Celui-ci s'occupe alors de voir si l'URL demandée ne ferait pas partie du catalogue des `flatpages`, auquel cas il sert la page correspondante. Cette solution a l'avantage de réduire la configuration à la seule installation de l'intergiciel, mais comporte aussi quelques points faibles : 1) les méthodes `process_view()` des intergiciels, pour ceux qui en sont dotés, ne sont pas appelées car la non-résolution d'URL interrompt prématurément le traitement ; 2) le rang de placement de l'intergiciel parmi les autres a de l'importance : il est préférable de le mettre en fin de liste pour être le premier dans le chemin de remontée de la réponse et substituer au plus tôt le 404 par une bonne réponse ; 3) si pour une raison quelconque la formation de la réponse 404 n'aboutit pas et dérive vers une réponse de type 500, l'interception sera manquée.

La seconde manière, plus traditionnelle et plus explicite, va être préférée, avec l'établissement d'un routage. La spécification du routage peut être choisie plus ou moins fine.

Du côté du plus fin, des routes individuelles sont configurées :

```
from django.contrib.flatpages import views

urlpatterns += [
    path('mentions-legales/', views.flatpage,
         {'url': '/mentions-legales/'}, name='mentions'),
    path([...]),
    [...]
]
```

L'avantage est de pouvoir donner un nom à la vue, pouvant être ainsi référencé dans d'autres gabarits de page afin d'établir des hyperliens. L'inconvénient est une certaine redondance puisque l'URL doit être à nouveau fournie en paramètre puisque non capturée dans le motif, sans compter que l'URL est encore une fois citée en table de la base de données.

À l'autre extrémité, un motif universel est configuré pour correspondre avec n'importe quelle URL. Ce motif doit alors être placé en toute fin de routage :

```
from django.contrib.flatpages import views

[...]
urlpatterns += [
    path('<path:url>', views.flatpage),
]
```

Entre les deux extrêmes, un compromis est de dédier une racine de branche de chemins uniquement pour ces pages.

▶ Complétez le routage :

`mysite\urls.py`

```
urlpatterns = [
    [...]
    path('pages/', include('django.contrib.flatpages.urls')),
    [...]
]
```

Une `flatpage` étant un modèle comme un autre, sa manipulation (création, modification, suppression) se fait de manière ordinaire. Il faut simplement ne pas oublier de lier la page à un site.

© Editions ENI - All rights reserved

Créez une page :

```
>>> from django.contrib.flatpages.models import FlatPage
>>>
>>> fp = FlatPage.objects.create(url='/mentions-legales/',
... title='Mentions légales',
... content='<h1>Mentions Légales</h1>')
>>> fp.sites.add(1)
```

Une instance est tout d'abord créée et conservée sous la variable `fp`. L'attribut `sites` de l'instance porte une relation de type *many-to-many* avec le modèle `Site`. Il donne accès à un gestionnaire de relation dont l'API propose les méthodes suivantes : `add()`, `create()`, `remove()`, `clear()` et `set()`.

La méthode `add()` a été employée. Elle accepte en un ou plusieurs arguments aussi bien l'objet pointé (ici un objet de classe `Site`) que la valeur du champ visé (ici une clé primaire).

Un gestionnaire de relation inverse est aussi disponible à l'autre bout de la relation. Par défaut, son nom est composé à partir du nom de la classe reliée, mis en minuscules, auquel le suffixe `_set` est ajouté (cf. chapitre Modèles). Les créations de la page et de sa liaison pouvaient également se faire ainsi :

```
>>> from django.contrib.sites.models import Site
>>>
>>> fp = Site.objects.get().flatpage_set.create(
... url='/mentions-legales/', title='Mentions légales',
... content='<h1>Mentions Légales</h1>')
>>> fp
<FlatPage: /mentions-legales/ -- Mentions légales>
```

Ici, une syntaxe au plus court est utilisée avec un `get()` sans argument pour obtenir le site. On a pu se le permettre, car il est su qu'il n'existe qu'un enregistrement dans la table. Il ne faut pas l'interpréter comme prendre le premier enregistrement ou un enregistrement au hasard ; dans toute autre situation, une exception est levée, soit pour absence d'objet, soit pour présence de plus d'un objet.

Les manières précédentes de créer les pages étant de bas niveau, elles permettent potentiellement de faire entrer dans la table des enregistrements incorrects ou conflictuels entre eux. Il y a notamment conflit si on se retrouve avec plus d'un enregistrement de même URL et pour le même site. Il est facile de le mettre en évidence, si les manipulations proposées ont été faites à de multiples reprises, avec cet exemple :

```
>>> from django.contrib.flatpages.views import flatpage
>>>
>>> flatpage(None, '/mentions-legales/')
Traceback (most recent call last):
 File "<console>", line 1, in <module>
 [...]
django.contrib.flatpages.models.FlatPage.MultipleObjectsReturned:
 get() returned more than one FlatPage -- it returned 2!
```

Le premier paramètre de la vue est normalement l'objet `request`, mais une valeur nulle suffit ici pour aboutir à la démonstration voulue.

Pour éviter ce genre de souci, on peut mettre à profit un formulaire mis à disposition par l'application, qui réalise certains contrôles de validité : conformité syntaxique du champ URL, non-duplication.

L'idée générale est d'imiter le cheminement du traitement d'une soumission et donc de simuler un objet `request.POST`. Voici tout d'abord une tentative qui démontre comment on aurait pu être protégé contre le problème d'intégrité introduit précédemment :

```
>>> from django.contrib.flatpages.forms import FlatpageForm
>>> from django.http import QueryDict
>>>
>>> post = QueryDict(mutable=True)
>>> post.update({
... 'url': '/mentions-legales/', 'sites': 1,
... 'title':'Mentions légales',
... 'content': '<h1>Mentions Légales</h1>'})
>>> form = FlatpageForm(post)
>>> form.is_valid()
False
>>> form.errors
{'__all__': ['Flatpage with url /mentions-legales/ already exists
for site example.com']}
```

© Editions ENI - All rights reserved

Un objet `QueryDict` est par défaut immutable, ce qui explique la présence du paramètre `mutable` dans son constructeur pour pouvoir l'alimenter ensuite. La désignation du site est ici une clé de dictionnaire comme les autres.

En respectant les contraintes avec une URL unique, on peut poursuivre jusqu'à la sauvegarde du formulaire, ce qui créera la page et sa liaison au site :

```
>>> post['url'] = '/mentions-legales-v2/'
>>> form = FlatpageForm(post)
>>> form.is_valid()
True
>>> fp = form.save()
>>> fp
<FlatPage: /mentions-legales-v2/ -- Mentions légales>
```

Le champ `template_name`, donnant la possibilité d'associer un gabarit particulier à la page, n'a pas été employé. Dans cas, la valeur par défaut `flatpages/default.html` s'applique, mais le fichier gabarit lui-même n'est pas fourni par l'application. Il ne faut évidemment pas créer de répertoire `templates/` sous `django/contrib/flatpages/`, la bonne place est le répertoire de gabarits global au site, déclaré par le paramètre de configuration `TEMPLATES.DIRS`.

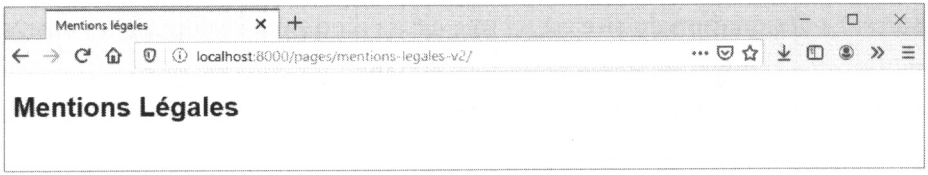
■Créez le gabarit par défaut :

`mysite\templates\flatpages\default.html`

```
<!DOCTYPE html>
<html>
 <head>
   <title>{{ flatpage.title }}</title>
 </head>
 <body>
   {{ flatpage.content }}
 </body>
</html>
```

L'instance de l'objet page est disponible dans les données de contexte sous le nom de variable `flatpage`. L'exemple de contenu du gabarit donné ici est une expression minimale.

Tout est maintenant en place pour servir les pages :



L'application est ancienne et n'est pas exempte de reproches en facilités d'utilisation, en particulier pour la gestion des hyperliens :

- D'une part, l'intégration avec la balise de gabarit traditionnelle de formation d'hyperlien est délicate. Pour bénéficier de la résolution complète et obtenir le chemin correct `/pages/mentions-legales/`, il faudrait l'écrire :

```
{% url 'django.contrib.flatpages.views.flatpage'
'mentions-legales/' %}
```

  On a donc deux lourdeurs : taper l'URL sans faute et omettre le caractère / de tête.

- D'autre part, il existe une balise de gabarit `get_flatpages` pour faciliter la construction d'une liste des pages, qu'on devine surtout destinée à un plan de site ou en pied de page, mais son usage tel qu'il est documenté ne s'accorde pas avec un routage démarrant à une racine dédiée comme cela a été fait avec `'pages/'`. On s'en sort avec cette écriture, mais elle est loin d'être conviviale :

```
<a href="{% url 'django.contrib.flatpages.views.flatpage'
page.url|slice:'1:' %}">{{ page.title }}</a>
```

4.2 Fichiers statiques

On désigne par « fichiers statiques » des ressources nécessaires au projet, mais qui n'ont pas à être composées, elles sont déjà prêtes à être servies. Les cas typiques sont les fichiers d'images, de feuilles de style et de JavaScript. Un serveur applicatif tel que Django prend toute sa valeur à servir du contenu dynamique, c'est-à-dire en composant des pages à la demande. Servir des fichiers est une tâche qu'il est préférable de confier à un serveur frontal ad hoc, conçu et optimisé pour ce rôle.

© Editions ENI - All rights reserved

D'un côté, on a les ressources d'un projet, disséminées à la fois dans de multiples applications et dans des emplacements centraux. D'un autre côté, on a un serveur frontal, souvent un produit réputé sur le marché, mais bien sûr ignorant de Django. Il s'agit alors de faire l'intégration entre ces deux domaines.

Le module `django.contrib.staticfiles` est proposé pour faciliter cette jonction, avec des manières de fonctionner prévues différemment selon le mode : développement ou production.

Pour rappel, car le sujet a déjà été évoqué en chapitre Création de site, le fait de mettre en activité cette application de ressources statiques supplante le serveur de développement ordinaire en :

`\django\core\management\commands\runserver.py`

par celui en :

`\django\contrib\staticfiles\management\commands\runserver.py`

De simples fichiers de style vont servir d'exemples pour expérimenter la mise en œuvre de cette application.

▶Mettez en fonctionnement le module :

`mysite\settings.py`

```
[...]
INSTALLED_APPS = [
    [...]
    # 'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',  # dépendance de flatpages
    [...]
]
```

Pour rappel, la définition suivante, nécessaire à l'application, est déjà présente, car elle fait partie du gabarit apporté par l'assistant de création de projet :

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

Le choix de la valeur est libre, mais la proposition par défaut est en général conservée pour rester dans les habitudes. Elle représente le préfixe ajouté dans la composition des URL par la balise de gabarit présentée juste après.

Le système de configuration prévoit un paramètre `STATICFILES_FINDERS` pour déclarer des moteurs chargés de localiser les ressources selon diverses stratégies, chacun ayant sa spécialité concernant le lieu et le mode d'exploration. Par défaut, si on ne précise pas ce paramètre dans son fichier de configuration, les moteurs activés sont :

- `django.contrib.staticfiles.finders.FileSystemFinder` :
  La recherche est faite sous les répertoires mentionnés par le paramètre `STATICFILES_DIRS`. L'objectif manifeste est de cibler des ressources globales au site, comme des bibliothèques JavaScript, des icônes et logos, des infrastructures de style, des polices de caractères, etc.
- `django.contrib.staticfiles.finders.AppDirectoriesFinder` :
  L'espace de recherche correspond aux applications installées. Les ressources visées sont normalement propres à chaque application, comme des traitements JavaScript spécialisés.

Pour un premier exemple, l'accès à une ressource globale au site est souhaité, indépendamment de toute application.

▶ Ajoutez dans le gabarit d'accueil un lien vers une ressource de style :

`mysite\templates\index.html`

```
{% load static %}
<html>
<head>
<link rel="stylesheet" href="{% static 'css/site.css' %}">
</head>
<body>
Bienvenue
</body>
</html>
```

La balise de gabarit nommée `static` compose une URL en préfixant son paramètre avec la configuration de `STATIC_URL`.

© Editions ENI - All rights reserved

Si à la place on souhaite faire des assemblages soi-même, les options pour disposer de la valeur du paramètre sont :

– Soit activer ce processeur de contexte qui met une variable nommée `STATIC_URL` dans les données de contexte :

```
django.template.context_processors.static
```

– Soit utiliser cette balise de gabarit :

```
{% get_static_prefix %}
```

▶Définissez au moins un emplacement où chercher des ressources globales :

`mysite\settings.py`

```
[...]
STATIC_URL = '/static/'

STATICFILES_DIRS = [
   os.path.join(_SITE_ROOT, 'static'),
]
[...]
```

Par défaut, cette définition contient une liste vide. De manière conventionnelle, le choix est fait ici de loger les ressources dans un sous-répertoire du projet et de rester sur le nom habituel `static`.

▶Créez l'arborescence de répertoires en accord :

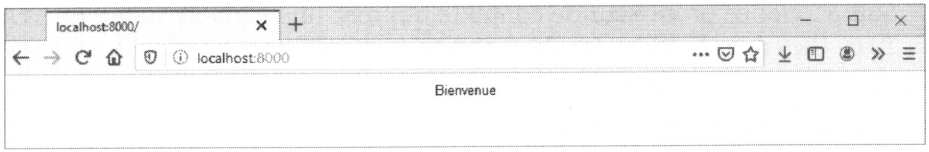
```
mysite\
├─ [...]\
├─ static\
│  └─ css\
└─ [...]\
```

▶Créez le fichier de style :

`mysite\static\css\site.css`

```
body {
 text-align: center;
}
```

Avec le serveur de développement et en mode DEBUG, cela suffit au fonctionnement du site :



Sur la console du serveur, il apparaît :

```
[...] "GET / HTTP/1.1" 200 107
[...] "GET /static/css/site.css HTTP/1.1" 200 32
```

Dans l'URL ci-dessus, la part `/static/` correspond au paramètre de configuration `STATIC_URL`, pas au chemin cité dans le paramètre `STATICFILES_DIRS`, qui reste un détail interne d'implémentation.

Le second exemple concerne l'accessibilité d'une ressource attachée à une application. L'exercice est fait, dans la continuité du précédent, avec aussi une feuille de style, mais le raisonnement se comprend probablement encore mieux avec une image.

La convention est de localiser les ressources sous le répertoire `static/` de chaque application installée. Ce nom est fixe, il serait facile d'en choisir un autre à condition d'implémenter son propre moteur sur un héritage du moteur standard et de surcharger son attribut `source_dir` (mais personne n'a envie de s'embêter avec ça).

Dans l'application `messenger`, le gabarit de visualisation de message va servir d'exemple.

▶ Ajoutez dans le gabarit un lien vers une ressource de style :

`mysite\messenger\templates\messenger\message_detail.html`

```
{% load static %}
<html>
<link rel="stylesheet"
 href="{% static 'messenger/css/style.css' %}">
Bienvenue dans l'application messenger.
<div class="msg_message">{{ object }}</div>
<div>{{ message }}</div>
</html>
```

© Editions ENI - All rights reserved

Ce n'est pas bien grave si on ne se préoccupe pas ici d'écrire un code HTML correct, le navigateur est tolérant et restructure la page avec les balises adéquates.

Il est nécessaire de préciser à nouveau le nom de l'application dans le chemin vers une ressource, comme cela a été expliqué et fait dans le chapitre Création de site pour les gabarits. La raison est identique : la ressource recherchée est prise de la première application qui en détient un exemplaire.

Si on se contentait de mentionner un banal `'css/style.css'`, le risque est élevé d'entrer en conflit avec une autre application qui emploierait la même écriture. Une solution élégante est de cloisonner selon un principe d'espace de noms matérialisé par un niveau de répertoire.

▶Créez l'arborescence de répertoires en accord :

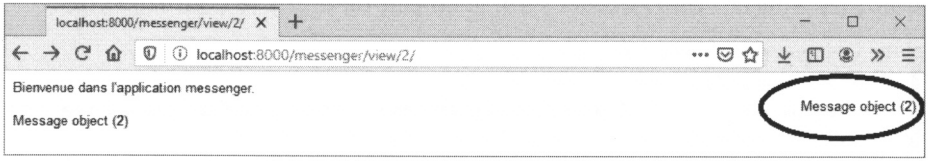
```
mysite\
├─ [...]\
├─ messenger\
│  ├─ [...]\
│  ├─ static\
│  │  └─ messenger\
│  │     └─ css\
│  └─ [...]\
└─ [...]\
```

▶Créez le fichier de style :

`mysite\messenger\static\messenger\css\style.css`

```
.msg_message {
 text-align: right;
}
```

Avec le serveur de développement et en mode `DEBUG`, le fonctionnement du site se vérifie :



Sur la console du serveur, il apparaît :

```
[...] "GET /messenger/view/2/ HTTP/1.1" 200 199
[...] "GET /static/messenger/css/style.css HTTP/1.1" 200 39
```

L'application `staticfiles` fait plus que servir des fichiers statiques, en mode développement grâce à sa commande administrative `runserver`, déjà mentionnée. Elle propose deux autres commandes administratives.

La commande `findstatic` est un outil d'aide au diagnostic. Elle permet de révéler si des ressources statiques existent et où elles sont situées.

En cas d'incompréhension ou de doute sur le fonctionnement des pages du site en rapport avec les ressources, l'outil peut rapidement mettre en évidence des manques ou des conflits.

Voici un premier exemple sur une ressource valide, existante et unique :

```
D:\dj>py manage.py findstatic --verbosity 2 css/site.css
Found 'css/site.css' here:
 D:\dj\mysite\static\css\site.css
Looking in the following locations:
 D:\dj\mysite\static
 D:\Python37\lib\site-packages\Dj[...]\django\contrib\admin\static
 D:\dj\mysite\messenger\static
```

Positionner l'option `--verbosity` à 2 permet d'afficher en plus la liste de répertoires explorés. On constate en particulier l'ordre de recherche : en premier niveau celui donné par l'ordre des moteurs de recherche déclarés, soit par défaut les répertoires globaux de `STATICFILES_DIRS`, ensuite les répertoires des applications installées, et en second niveau l'ordre de déclaration des éléments dans chacune des listes.

Supposons, comme ici, l'application intégrée d'administration activée. Si on a par inadvertance idée de mettre en œuvre dans son application une ressource nommée `admin/css/base.css`, le conflit de noms se confirme vite :

```
D:\dj>py manage.py findstatic admin/css/base.css
Found 'admin/css/base.css' here:
 D:\Python37\[...]\django\contrib\admin\static\admin\css\base.css
 D:\dj\mysite\messenger\static\admin\css\base.css
```

© Editions ENI - All rights reserved

On voit que la ressource est obtenue prioritairement de l'autre application. À l'inverse, on peut tirer profit de la mécanique de priorité pour remplacer des ressources d'une application, sans devoir aller dégrader son contenu :

```
D:\dj>py manage.py findstatic admin/css/base.css
Found 'admin/css/base.css' here:
  D:\dj\mysite\static\admin\css\base.css
  D:\Python37\[...]\django\contrib\admin\static\admin\css\base.css
```

Enfin, la commande permet de consolider un diagnostic supposé d'erreur de saisie de nom, d'oubli de déclaration de répertoire ou de mauvais placement, si une ressource attendue n'est pas détectée.

Le rapport produit équivaut à :

```
D:\dj>py manage.py findstatic --verbosity 2 js/jquery.min.js
No matching file found for 'js/jquery.min.js'.

Looking in the following locations:
 D:\dj\mysite\static
 D:\Python37\lib\site-packages\Dj[...]\django\contrib\admin\static
 D:\dj\mysite\messenger\static
```

4.2.1 Déploiement en production

La commande `collectstatic` est destinée à faciliter le déploiement des ressources statiques dans un environnement de production, dans lequel est normalement prévu un serveur frontal ayant parmi ses tâches celle de délivrer ces ressources. Son objectif est d'automatiser le travail fastidieux de rassembler tous les fichiers, répartis dans de multiples répertoires d'origine, vers un répertoire central, lisible du serveur frontal.

Le point de concentration est donné par le paramètre de configuration `STATIC_ROOT`, qui doit être établi, car il n'a pas de valeur par défaut. Ce lieu ne doit jamais servir de source de stockage de données originales, car il faut imaginer qu'il doit pouvoir être nettoyé à tout instant, sans préjudice. L'option `--clear` est justement prévue pour vider préalablement l'espace avant d'agir dessus. Le chemin doit être un nom absolu et se terminer par un caractère séparateur de chemin.

Pour l'exercice, le lieu choisi est sous la racine de l'espace de travail, et l'appeler `static` lui donne l'avantage de bien exprimer son rôle :

```
dj\
├─ mysite\
├─ static\
└─ [...]
```

▶Établissez le lieu de collecte :

`mysite\settings.py`

```
[...]
STATIC_URL = '/static/'
STATIC_ROOT = 'D:/dj/static/'
[...]
```

▶Passez un premier essai pour une observation sans effet :

```
D:\dj>py manage.py collectstatic --dry-run
Pretending to copy 'D:\dj\mysite\static\css\site.css'
P[...]'[...]\django\contrib\admin\static\admin\css\autocomplete.css
[...]
P[...] contrib\admin\static\admin\js\vendor\xregexp\xregexp.min.js'
P[...] copy 'D:\dj\mysite\messenger\static\messenger\css\style.css'

121 static files copied to 'D:\dj\static'.
```

Le volume de traces provient essentiellement de l'application `admin`. En réelle exécution, les lignes équivalentes « `Copying '...'` » ne sont pas produites, sauf à monter le niveau de verbosité à 2.

Avec un filtre, le flux se tarit et la visibilité est meilleure :

```
D:\dj>py manage.py collectstatic --dry-run --ignore admin
Pretending to copy 'D:\dj\mysite\static\css\site.css'
P[...] copy 'D:\dj\mysite\messenger\static\messenger\css\style.css'

2 static files copied to 'D:\dj\static'.
```

© Editions ENI - All rights reserved

Dans un premier stade, ce sont les deux seuls fichiers vraiment utiles au site et on peut conserver le filtre dans l'exécution de la collecte réelle :

```
D:\dj>py manage.py collectstatic --ignore admin

2 static files copied to 'D:\dj\static'.

D:\dj>tree /f static
[...]
D:\DJ\STATIC
├───css
│       site.css
│
└───messenger
    └───css
            style.css
```

Les lancements suivants de la commande sont plus verbeux, car le lieu de collecte n'est plus vierge. Voici un essai, en ignorant tout pour rester neutre :

```
D:\dj>py manage.py collectstatic --ignore *

You have requested to collect static files at the destination
location as specified in your settings:

   D:\dj\static

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel: yes

0 static files copied to 'D:\dj\static'.
```

Pour un vidage, sans remettre de fichiers :

```
D:\dj>py manage.py collectstatic --clear --ignore *

You have requested to collect static files at the destination
location as specified in your settings:

   D:\dj\static

This will DELETE ALL FILES in this location!
Are you sure you want to do this?
```

```
Type 'yes' to continue, or 'no' to cancel: yes
Deleting 'css\site.css'
Deleting 'messenger\css\style.css'

0 static files copied to 'D:\dj\static'.
```

On notera que seuls les fichiers sont détruits, l'opération laisse des répertoires vides :

```
D:\dj>tree /f static
[...]
D:\DJ\STATIC
├───css
└───messenger
    └───css
```

La commande opère par de la copie brute de contenu, mais avec une optimisation pour éviter des transferts inutiles : si le fichier destination existe et si son contenu est déjà le même que celui du fichier source (sur base de l'horodatage de sa dernière modification), il est sauté :

```
D:\dj>py manage.py collectstatic --noinput --verbosity 2 -i admin
Skipping 'css\site.css' (not modified)
Skipping 'messenger\css\style.css' (not modified)

0 static files copied to 'D:\dj\static', 2 unmodified.
```

L'option `--noinput` est aussi utilisée pour s'épargner les dialogues de confirmation.

Le fait de pouvoir procéder par lien symbolique plutôt que par copie, avec l'option `--link`, est une piste valable à considérer.

Sous Windows, la création de lien symbolique en Python (fonction `os.symlink()`) est assujettie à un droit, qui n'est pas détenu par un utilisateur ordinaire de la machine :

```
D:\dj>py manage.py collectstatic --noinput --link --ignore admin
CommandError: symbolic link privilege not held
```

Le plus simple est de travailler en tant qu'administrateur, en ouvrant une console « Invite de commande (admin) ».

© Editions ENI - All rights reserved

Pour vérification, l'accessibilité au droit nécessaire peut se voir ainsi par la présence de la ligne (ne pas se préoccuper de la colonne `État` à la valeur `Désactivé`) :

```
D:\dj>whoami /priv

Informations de privilèges
----------------------

Nom de privilège                Description                    État
=============================== ============================== ====
[...]
SeCreateSymbolicLinkPrivilege   Créer des liens symboliques   [...]
```

La commande passe alors :

```
D:\dj>py manage.py collectstatic --noinput --link --ignore admin

2 static files symlinked to 'D:\dj\static'.
```

Des liens symboliques sont bien présents :

```
D:\dj>dir /s/n/al static
[...]
 Répertoire de D:\dj\static\css
nn/nn/20nn  nn:nn    <SYMLINK>      site.css
 [D:\dj\mysite\static\css\site.css]
               1 fichier(s)                0 octets

 Répertoire de D:\dj\static\messenger\css
nn/nn/20nn  nn:nn    <SYMLINK>      style.css
 [D:\dj\mysite\messenger\static\messenger\css\style.css]
               1 fichier(s)                0 octets
[...]
```

Si le serveur frontal en charge de délivrer les ressources statiques n'est pas le serveur de déploiement, une phase additionnelle peut être nécessaire pour pousser le nouveau contenu de la collecte vers l'espace de stockage accédé par le serveur frontal. Des outils de synchronisation de répertoires ou des disques réseau partagés sont des exemples de moyens.

Dans la situation d'une distribution par un fournisseur externe, Cloud ou CDN (*Content Delivery Network*), il est possible d'automatiser le processus de diffusion en implémentant un moteur de stockage personnalisé afin de faire le complément de traitement dans une méthode nommée `post_process()`. Des paquets pour les grands noms du stockage en ligne existent dans la communauté.

4.2.2 Simulations en développement

Les deux exemples expérimentés pour servir un fichier CSS ont mis en évidence un fonctionnement immédiat en environnement de développement, par la simple présence de l'application intégrée pour fichiers statiques. Dans ce cas, l'intégration se fait très en amont, au niveau des gestionnaires WSGI, et il s'agit plus d'une interception d'URL que d'une reconnaissance de motif comme nous le faisons d'ordinaire. Dans le cas où cette application n'est pas installée ou qu'un serveur de développement autre que `runserver` est mis en œuvre, alors qu'on veut tout de même servir des fichiers, il est possible de compléter la configuration du site de manière un peu plus manuelle.

Deux réalisations vont être exposées. Toutes deux sont basées sur l'habituelle détection de motif dans l'URL, mais il ne sera pas nécessaire de l'écrire, car des fonctions de haut niveau sont disponibles pour masquer cette mécanique et simplifier l'interfaçage. Sous condition de validité de la configuration (dont le mode `DEBUG`), le principe est d'ajouter au routage l'équivalent de cette route (avec `STATIC_URL` valant `/static/`) :

```
re_path(r'^static/(?P<path>.*)$', serve)
```

La vue `serve` est différente selon l'option :

- Option 1 : imitation de `staticfiles` dans son spectre de découverte.

  Dans cette version, le paramètre de configuration `STATIC_ROOT` et la commande `collectstatic` ne sont pas nécessaires, car les ressources sont recherchées avec l'algorithme de `staticfiles`, c'est-à-dire dans les divers répertoires source.

© Editions ENI - All rights reserved

Le routage se complète ainsi :

mysite\urls.py

```
from django.contrib.staticfiles.urls \
   import staticfiles_urlpatterns
[...]
urlpatterns += staticfiles_urlpatterns()
```

La fonction utilisée rend une route vers la vue de l'application intégrée :

`django.contrib.staticfiles.views.serve`

– Option 2 : ne piocher que dans `STATIC_ROOT`.

Avec cette variante, l'idée est d'être plus proche du comportement d'un serveur de déploiement en étant plus restrictif et en ne regardant que sous le lieu de collecte pour servir les ressources.

Contrairement à l'option précédente, il n'est pas fait usage de la vue de l'application intégrée. Celle-ci est court-circuitée pour appeler directement la vue de base :

`django.views.static.serve`

Le routage se complète ainsi :

mysite\urls.py

```
from django.conf import settings
from django.conf.urls.static import static
[...]
urlpatterns += static(settings.STATIC_URL, \
   document_root=settings.STATIC_ROOT)
```

Aucun composant de l'application intégrée n'est employé dans cette configuration.

5. Structuration des pages

La formation d'une page est en grande partie fondée sur de l'assemblage de blocs, avec trois balises structurantes et selon deux grands principes : la surcharge et l'inclusion.

La surcharge consiste à hériter d'un gabarit de plus grande envergure puis à définir ou compléter des contenus pour ses blocs, qu'ils soient vierges ou qu'ils aient déjà un contenu. Une relation d'héritage simple est établie entre un gabarit fils et son éventuel gabarit parent. Le bénéfice typique de cette composition est d'adopter une disposition constante pour un ensemble de pages ou pour le site entier. Les balises de gabarits utilisées sont `{% extends %}` et `{% block %}`.

Le sens de raisonnement de la surcharge est orienté du plus fin vers le plus englobant, qu'on peut assimiler à un sens montant. Le chemin y est unique et tout tracé.

L'inclusion complète le tableau en œuvrant dans l'autre sens, le sens descendant, pour faire venir autant de morceaux que voulu et ainsi établir des ramifications. On devine tout de suite les avantages pour des portions devant être affichées sur plus d'une page, à l'identique ou de façon très similaire. La balise à employer est `{% include %}`.

6. Expérimentation rapide et manuelle

Il arrive que le fonctionnement ou le résultat d'une balise ou d'un filtre restent obscurs malgré la documentation, ou qu'on tâtonne sur un bon format de conversion. Il est toujours possible d'éditer son gabarit, le sauver, rafraîchir le navigateur, constater le résultat et recommencer. Si le sujet d'expérimentation est limité, le cycle d'essai peut être rendu plus court en faisant des manipulations manuelles en lignes de commande.

© Editions ENI - All rights reserved

Voici un exemple, pour comprendre l'effet de l'échappement automatique, lorsqu'il est actif (par défaut) ou pas :

```
>>> from django.template import Context, Template
>>>
>>> t = Template('{{ v }}'
... '{% autoescape off %}{{ v }}{% endautoescape %}')
>>> t.render(Context({'v': ' un <br> deux '}))
' un &lt;br&gt; deux  un <br> deux '
```

Les commandes peuvent être rapidement retrouvées dans l'historique en naviguant avec les touches fléchées haut et bas, éditées et relancées.

```
>>> from django.utils.safestring import mark_safe
>>>
>>> t.render(Context({'v': mark_safe('<script>danger... ')}))
'<script>danger... <script>danger... '
>>>
>>> from django.utils.html import escape
>>>
>>> t.render(Context({'v': escape("<script>Jtai cassé... ")}))
'&lt;script&gt;Jtai cassé... &lt;script&gt;Jtai cassé... '
```

7. Écritures des gabarits

7.1 Base du site

Il est courant que le site soit conçu pour présenter ses pages selon la même disposition. Le mieux est bien sûr de n'avoir à écrire cette structure qu'une seule fois et qu'elle soit appliquée à toutes les pages. Ce squelette servira de parent à toutes les pages.

Imaginons une disposition à deux colonnes : à gauche un menu de navigation et à droite un espace de contenu à remplir par chacune des pages. La place de ce gabarit est parmi le répertoire global des gabarits du site.

▶Créez le gabarit de base :

`mysite\templates\base.html`

```
{% load static %}
<!DOCTYPE html>
<html>
<head>
 <title>Site - {% block title %}{% endblock %}</title>
 <link rel="stylesheet" href="{% static 'css/site.css' %}">
</head>
<body>
 <div style="float:left; margin-right:20px; width:20%;">
  {# à compléter #}ICI UN MENU
 </div>
 <div style="float:left;">
{% block content %}{% endblock content %}
 </div>
</body>
</html>
```

Cette première étape d'écriture montre trois éléments standards de balisage :

- `{% load static %}`

  Cette balise a été mentionnée auparavant dans des exemples, mais sans être expliquée en détail. Elle demande le chargement de balises et filtres non intégrés et donc fournis par une application installée.

  L'emplacement conventionnel de ces composants est sous un module nommé `templatetags` sous la racine de l'application. Les composants peuvent être rassemblés dans un ou plusieurs fichiers Python de ce répertoire. La répartition est libre, mais devrait être représentative de familles cohérentes. Le nom du fichier est lui aussi libre, mais doit être choisi en tenant compte d'un potentiel risque de conflit de nommage : il s'agit du paramètre à donner à la balise de chargement, sans plus de précision d'origine. Si une autre application nomme son module par le même terme, les règles habituelles du premier trouvé s'appliquent.

© Editions ENI - All rights reserved

Dans le cas présent, il est demandé le chargement des composants ciblés en :

```
django\contrib\staticfiles\
├─ [...]
├─ templatetags\
│  ├─ __init__.py
│  └─ static.py
└─ [...]
```

- `{% block ... %} ... {% endblock %}`

  Ces balises marquent des emplacements qui pourront être retravaillés par un gabarit fils. Deux lieux sont prévus : un premier pour compléter la zone de titre ; un second pour alimenter la page avec une charge utile. Un nom doit être donné dans la balise ouvrante du bloc, pour qu'il puisse être référencé dans les autres gabarits. Par confort de lecture, un nom, de préférence le même, peut aussi être placé dans la balise fermante pour faire le rapprochement lorsque le volume du corps les éloigne trop l'une de l'autre.

- `{# ... #}`

  Il s'agit d'une syntaxe du langage de gabarit pour placer des commentaires. On peut l'utiliser soit pour documenter le code, soit pour neutraliser temporairement du code. En contrepartie de sa simplicité, son usage est limité : l'ouverture et la fermeture doivent tenir sur la même ligne. Pour mettre des commentaires multi-lignes, il faut employer une balise de gabarit :

```
... du code HTML ou autre{% comment %}
Ceci est du commentaire, initié sur une nouvelle ligne,
pour confort de lecture,
et surtout qui s'étend sur plusieurs lignes.
{% endcomment %}suite du HTML ...
```

▶Modifiez la feuille de style du site pour désactiver ce qui n'est plus utile :

`mysite\static\css\site.css`

```
/*
body {
 text-align: center;
}
*/
```

Sur cette base de départ, même inachevée, il devient immédiatement possible de restructurer la page d'accueil général du site. Telle qu'elle est restée jusqu'à présent, elle est parfaitement compatible avec le schéma de base.

▶Remplacez le contenu du gabarit d'accueil par cette nouvelle écriture :

`mysite\templates\index.html`

```
{% extends "base.html" %}
{% block title %}Accueil{% endblock %}
{% block content %}
Bienvenue
{% endblock %}
```

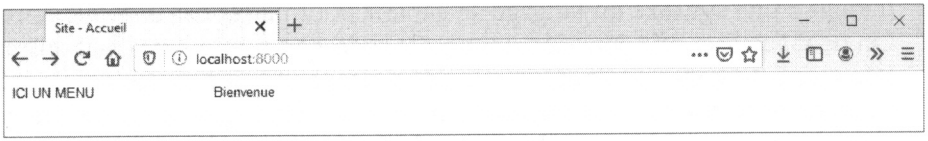
Une balise structurante est introduite ici :

– `{% extends ... %}`

  Le gabarit parent est désigné par la chaîne de caractères de son nom. Alternativement la balise admet l'emploi d'une variable.

Les deux autres balises `block` fournissent le contenu alimentant les emplacements de même nom au sein du parent.

Voici le nouveau rendu :



Il est similaire à l'ancien, avec en plus un titre composé d'un préfixe et du nom de la page. Il reste maintenant à compléter les contenus :

▶Alimentez la section réservée pour le menu de gauche :

`mysite\templates\base.html`

```
[...]
 <div style="float:left; margin-right:20px; width:20%;">

{% if user.is_authenticated %}<ul>
 <li><a href="{% url 'messenger:inbox' %}">Reçu</a></li>
 <li><a href="{% url 'messenger:sent' %}">Envoyé</a></li>
 <li><a href="{% url 'messenger:write' %}">Écrire</a></li>
</ul>{% else %}<i>Identifiez-vous pour disposer de la
messagerie</i>{% endif %}

 </div>
[...]
```

© Editions ENI - All rights reserved

L'affichage du menu n'a de sens que pour les utilisateurs authentifiés. Il est donc conditionné grâce à une structure :

```
{% if <condition> %}...{% endif %}
```

La syntaxe de la structure conditionnelle admet des alternatives par `{% elif <condition> %}` et `{% else %}`.

La condition s'exprime à l'aide d'opérateurs traditionnels de comparaison et de combinaison logique.

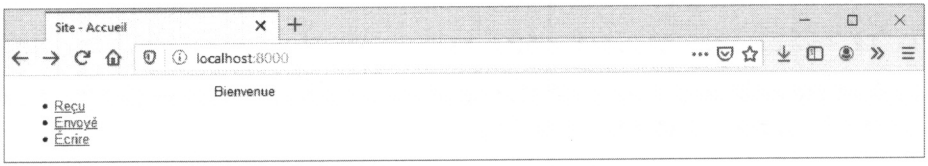
La répartition de l'indentation entre le code HTML et le code de gabarit n'est pas un sujet technique, mais de goût personnel. Plusieurs styles se rencontrent :

- Privilégier l'indentation du HTML puisqu'au final c'est ce code seul qui importe et qui sera émis vers le navigateur. Une consultation de la « charge utile de la réponse » avec les outils de développement du navigateur est rendue plus aisée pour un œil humain. Le code ci-dessus adopte ce choix.
- Privilégier la lecture du code de gabarit, quitte à introduire des caractères d'espacement, des replis de ligne et des lignes vides, inutiles au niveau du navigateur et rendant parfois la lecture humaine assez pénible. Avec cette préférence, le code précédent pouvait s'écrire :

```
{% if user.is_authenticated %}
 <ul>
  <li><a href="{% url 'messenger:inbox' %}">Reçu</a></li>
  <li><a href="{% url 'messenger:sent' %}">Envoyé</a></li>
  <li><a href="{% url 'messenger:write' %}">Écrire</a></li>
 </ul>
{% else %}
 <i>Identifiez-vous pour disposer de la messagerie</i>
{% endif %}
```

- Privilégier l'un ou l'autre choix, mais de temps en temps faire des compromis lorsqu'un respect strict de la convention nuit manifestement à la compréhension de la logique globale du code source.
- N'avantager aucune syntaxe en particulier, soit pour faire au mieux au cas par cas, soit parce qu'on ne veut pas y prêter attention.

Le nouvel aspect de l'accueil devient :

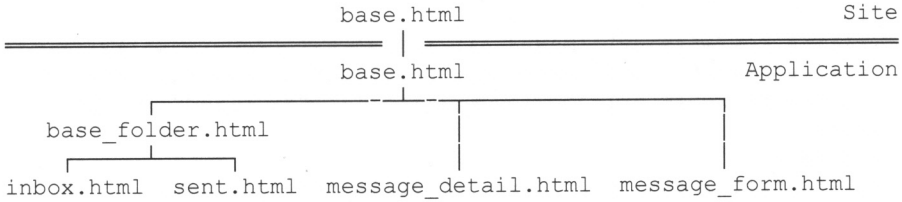


7.2 Bases de l'application

Comme le site, l'application mérite une réflexion sur la structuration de ses pages, pour répertorier les similitudes et rationaliser les points communs, car il est plus probable qu'une application comporte des motifs récurrents que des pages totalement disparates. Par exemple, on a facilement l'intuition que les pages relatives aux dossiers vont avoir de larges ressemblances, dont il n'est pas question de dupliquer le code.

Procéder par inclusion est parfois le réflexe qui vient trop tôt. L'héritage et la redéfinition de blocs demandent un peu plus d'effort de réflexion, mais donnent normalement des constructions mieux conçues.

L'arborescence des pages proposée est la suivante :



Deux nœuds intermédiaires vont être utiles : l'un pour factoriser les éléments de dossiers et l'autre pour factoriser les éléments d'application.

Le gabarit d'application va affiner le gabarit de site sur deux points : compléter le titre et ajouter des définitions de style. Pour le premier point, le bloc nécessaire est déjà en place. En revanche, pour le second, un ajustage est nécessaire dans le gabarit de site pour aménager un conteneur.

© Editions ENI - All rights reserved

▶Ajoutez un bloc en fin de balise <head> :

mysite\templates\base.html

```
[...]
{% block extrahead %}{% endblock %}
</head>
[...]
```

▶Créez le gabarit de base de l'application :

mysite\messenger\templates\messenger\base.html

```
{% extends "base.html" %}{# pas moi-même, mais celui du site #}
{% load static %}
{% block title %}Messagerie{% endblock %}
{% block extrahead %}{{ block.super }}
<link rel="stylesheet"
 href="{% static 'messenger/css/style.css' %}">
{% endblock %}
```

Lorsque la balise `extends` est utilisée, le mécanisme d'héritage impose qu'elle soit la première des balises dans le fichier.

Avoir employé le même nom `base.html` pour le gabarit fils et le gabarit parent n'est pas source de conflit de noms. Dans le gabarit fils, la recherche d'un gabarit parent explore d'abord les répertoires du site (paramètre de configuration `TEMPLATE.DIRS`) avant les répertoires des applications. Certains préféreront employer un nom distinct, comme `base_site.html`, pour se rassurer de toute ambiguïté.

Le bloc `title` donne un contenu qui alimente le parent. Rien ne doit être fait concernant le bloc `content` du parent, pas même une notion de propagation. La fourniture d'un contenu est du ressort des gabarits fils.

Le bloc `extrahead` est surchargé. Mais comme le but est de faire un complément et non un remplacement, l'existant doit être préservé. Le contenu du bloc dans le parent est mis à disposition sous la variable de nom conventionnel `block.super`, il suffit alors de l'utiliser et de faire ses ajouts avant, après ou autour. Il est vrai qu'en l'occurrence l'opération est neutre puisque le contenu issu du parent est vide, mais le principe cumulatif reste de mise et se décline habituellement à tous les étages dans une hiérarchie.

La pyramide se poursuit avec le gabarit commun aux dossiers. Puisque le choix est de privilégier l'héritage plutôt que des inclusions conditionnelles, le principe est fondé sur de l'exclusion : le parent prévoit tous les cas et le fils neutralise les cas qui ne s'appliquent pas à lui. Le bénéfice est que le fils n'a pas à connaître finement la structure du parent et les balises HTML attendues qu'il faudrait injecter. Dans le cas présent, le gabarit de dossier est conçu pour supporter autant la réception que l'envoi ; le choix est pourtant bien exclusif, mais il est différé au fils.

▶Créez le gabarit :

`mysite\messenger\templates\messenger\base_folder.html`

```
{% extends "./base.html" %}
{% block content %}
<h1>{% block msg_folder_title %}{% endblock %}</h1>
{% if msg_messages %}
<table>
 <thead>
  <tr>
{% block msg_sender_header %}
   <th>Expéditeur</th>{% endblock %}
{% block msg_recipient_header %}
   <th>Destinataire</th>{% endblock %}
   <th>Sujet</th>
   <th>{% block msg_date %}{% endblock %}</th>
  </tr>
 </thead>
 <tbody>{% for message in msg_messages %}
  <tr>
{% block msg_sender_cell %}
   <td>{{ message.sender }}</td>{% endblock %}
{% block msg_recipient_cell %}
   <td>{{ message.recipient }}</td>{% endblock %}
   <td>
    <a href="{% url 'messenger:view' message.pk %}">
    {{ message.subject|truncatewords:5 }}
    </a>
   </td>
   <td>{{ message.sent_at }}</td>
  </tr>{% endfor %}
 </tbody>
</table>
{% else %}<p>Aucun message.</p>{% endif %}
{% endblock content %}
```

© Editions ENI - All rights reserved

La balise `extends` donne habituellement des chemins complets par rapport aux racines de recherche des chargeurs, ce qui veut dire ce genre de formulation :

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

Il est toutefois permis de raisonner en relatif par rapport à l'actuel fichier, en débutant le chemin par `./` pour rester dans le même répertoire ou `../` (éventuellement plusieurs fois) pour monter au répertoire parent. Par mesure de sécurité, une tentative de remonter trop haut, au-dessus des racines, provoque une exception.

Le gabarit définit le contenu du bloc `content`, ou plus exactement une structure de contenu préremplie et à affiner, car il va lui-même aménager des blocs internes qui devront être alimentés, enrichis ou neutralisés par ses descendants.

Les blocs `msg_folder_title` et `msg_date` sont des emplacements prévus pour être personnalisés par un fils.

Le nombre de blocs peut devenir élevé, ainsi que les niveaux de hiérarchie, au risque de ne plus bien savoir situer les éléments. Si de plus on raisonne en termes d'application réutilisable, donc potentiellement insérée dans un environnement indéterminé, il est préférable de mettre un peu de discipline dans les noms des blocs. Ici, la convention prise est de préfixer les blocs initiés par l'application avec un motif propre à l'application, choisi à `msg_` pour rester court.

Une balise `if` évalue la variable donnant accès à la liste des instances de messages. L'évaluation d'une variable reproduit le comportement auquel on est habitué en Python, à savoir être vrai si existante, ni vide ni zéro, et pas le booléen `False`. L'objectif ici est d'afficher tout de suite un texte alternatif simple en cas de liste vide, plutôt que d'entrer dans la structure de présentation de messages.

Les paires de blocs, `msg_sender_header` + `msg_recipient_header` en en-tête, et `msg_sender_cell` + `msg_recipient_cell` en corps, sont des paires exclusives : le fils devra neutraliser un des deux cas.

Les balises `for` et `endfor` établissent une boucle ordinaire. L'objet à parcourir doit présenter un caractère itératif. Pour un dictionnaire, on pourra utiliser cette forme d'écriture, sachant que l'algorithme de l'opérateur symbolisé par un caractère point trouvera une correspondance avec une méthode `items()` :

```
{% for key, value in d.items %}
```

Dans le corps de la boucle, en plus de la variable `message`, des variables relatives au mécanisme de boucle sont disponibles, parmi lesquelles on trouve : `forloop.counter` pour un indice de progression, les booléens `forloop.first` et `forloop.last` pour les éléments d'extrémité, bien pratiques pour mettre du style d'encadrement.

La boucle admet une balise `{% empty %}` pour fournir un traitement alternatif si la séquence est vide ou inexistante. Cette faculté aurait pu être utilisée si on avait voulu maintenir la structure en table dans tous les cas :

```
<tbody>{% for message in msg_messages %}
 <tr>
  [...]
 </tr>{% empty %}
 <tr>
  <td colspan="3"style="text-align:center">Aucun message.</td>
 </tr>{% endfor %}
</tbody>
```

L'opérateur `|` est employé pour introduire la notion de filtre, qui consiste simplement à prendre la valeur produite à sa gauche, la transformer ou de manière générale s'en servir pour produire une nouvelle valeur. L'opérateur est applicable aux variables et aux paramètres de balise. Un filtre admet optionnellement un et un seul paramètre, sous forme de constante ou de variable, placé après un caractère séparateur `:`. Si on veut faire passer plusieurs pièces d'information élémentaires, on est amené à les enrober par une enveloppe, avec une codification convenue, comme le montre ce filtre intégré :

```
{{ valeur|yesno:"oui,non,indéterminé" }}
```

L'exemple utilisé ici est le filtre `truncatewords`, qui admet comme paramètre le nombre de mots au-delà duquel la chaîne d'entrée est tronquée. Il est appliqué au sujet pour plafonner l'espace occupé puisque le texte peut être très long et tout afficher n'est pas une nécessité dans un tableau synthétique.

© Editions ENI - All rights reserved

7.3 Dossier d'arrivée

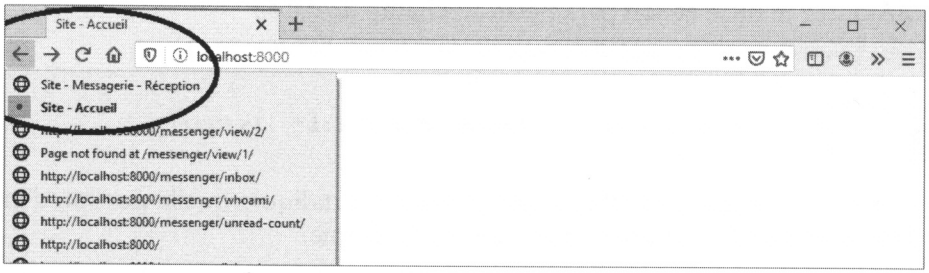
Le gabarit temporaire existant va pouvoir également être ajusté pour s'insérer dans la chaîne d'héritage des gabarits.

▶Remplacez le contenu du gabarit par cette nouvelle écriture :

`mysite\messenger\templates\messenger\inbox.html`

```
{% extends "./base_folder.html" %}
{% block title %}{{ block.super }} - Réception{% endblock %}
{% block msg_folder_title %}Messages reçus{% endblock %}
{% block msg_recipient_header %}{% endblock %}
{% block msg_date %}Reçu{% endblock %}
{% block msg_recipient_cell %}{% endblock %}
```

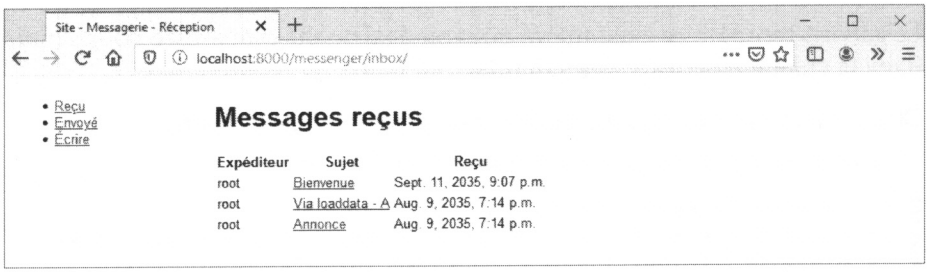
L'héritage vise le gabarit commun aux dossiers, situé au même niveau. On en profite aussi pour étendre le bloc du titre avec une qualification plus précise de la page. Ce point n'est pas négligeable, car cela permet de s'y retrouver dans l'historique de navigation, plutôt que d'avoir toutes les pages avec un même titre indifférencié :



Le travail résiduel se réduit à affiner les blocs définis par le parent direct. S'agissant d'une réception, les personnalisations doivent être les suivantes :

- Les blocs concernant le destinataire sont neutralisés pour que seuls restent les blocs concernant l'expéditeur.
- Le titre du dossier est renseigné.
- La colonne de date est qualifiée en tant que réception, même si le modèle de message n'a qu'un champ `sent_at`, en considérant que l'équivalence des dates est un détail d'implémentation.

Voici la copie d'écran du résultat attendu :



On remarque tout de suite que les dates n'ont pas un bel aspect. Effectivement, la mise en forme n'a pas été travaillée et on se retrouve avec un format par défaut, qui n'est manifestement pas français. Le sujet de l'adaptation aux conventions régionales est traité dans le chapitre Internationalisation.

Pour apporter une réponse simple et immédiate au problème, le rendu de la variable va passer par un filtre intégré prévu pour travailler les informations d'horodatage, en donnant explicitement le format désiré.

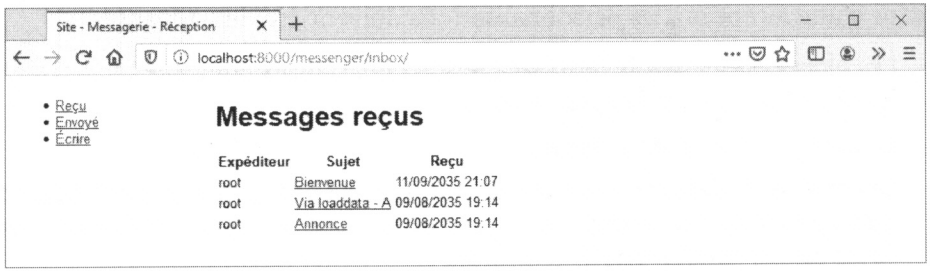
▶Ajoutez le filtre et son paramètre au champ de date :

`mysite\messenger\templates\messenger\base_folder.html`

```
[...]
    <td>{{ message.sent_at|date:"d/m/Y H:i" }}</td>
[...]
```

Il existe toute une panoplie de caractères pour indiquer quelle portion d'une date et heure on veut afficher et sous quelle forme.

Le résultat se présente mieux ainsi :



© Editions ENI - All rights reserved

7.4 Dossier d'envoi

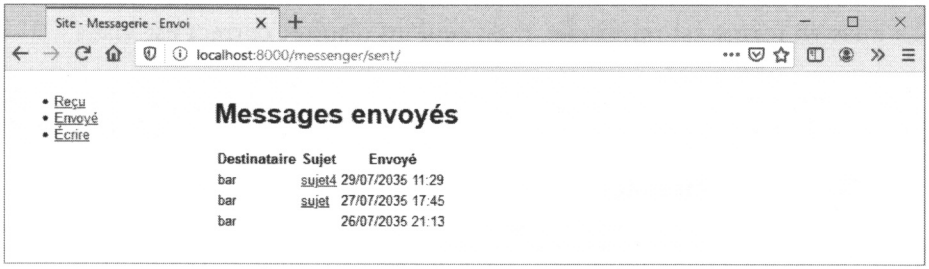
Le gabarit existant se met à niveau de la même façon que celui d'arrivée, simplement en basculant la vision sur l'autre côté de l'échange.

◘Remplacez le contenu du gabarit par cette nouvelle écriture :

`mysite\messenger\templates\messenger\sent.html`

```
{% extends "./base_folder.html" %}
{% block title %}{{ block.super }} - Envoi{% endblock %}
{% block msg_folder_title %}Messages envoyés{% endblock %}
{% block msg_sender_header %}{% endblock %}
{% block msg_date %}Envoyé{% endblock %}
{% block msg_sender_cell %}{% endblock %}
```

Le résultat doit ressembler à cela :



7.5 Lecture de message

L'héritage se fait directement au niveau du gabarit de base de l'application et un contenu doit être fourni.

◘Remplacez le contenu du gabarit par cette nouvelle écriture :

`mysite\messenger\templates\messenger\message_detail.html`

```
{% extends "./base.html" %}
{% block title %}{{ block.super }} - Lecture{% endblock %}
{% block content %}
<h1>Message</h1>
<div>
{{ message.sender }} &raquo; {{ message.recipient }} |
```

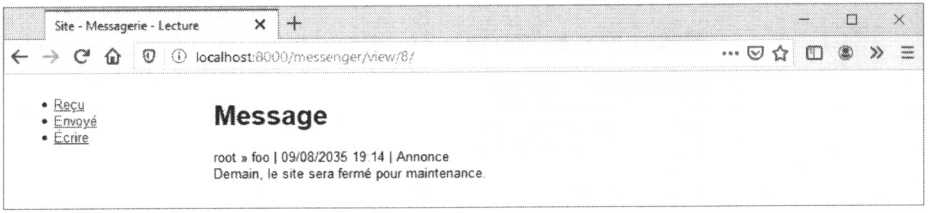
```
{{ message.sent_at|date:"d/m/Y H:i"}} | {{ message.subject }}
</div>
<div>{{ message.body|linebreaksbr }}</div>
{% endblock %}
```

La seule nouveauté introduite est le filtre intégré `linebreaksbr` pour convertir les retours de ligne du texte en balises `<br>`.

Les filtres intégrés générant du code HTML sont rares. On peut citer en plus :

- `linebreaks` pour faire comme `linebreaksbr`, mais en plus produire un ou plusieurs paragraphes avec des balises `<p>`.
- `unordered_list` pour habiller une liste avec des balises `<li>` (mais sans `<ul>`) et éventuellement ses sous-listes avec `<ul>` et `<li>`.
- `urlize` et `urlizetrunc` pour enrober une URL ou une adresse de courriel dans une balise hyperlien `<a>`.

La mise en forme est minimale, mais seul un résultat correct est visé :



7.6 Composition de message

L'héritage se fait de la même manière que pour le gabarit précédent.

▶Remplacez le contenu du gabarit par cette nouvelle écriture :

`mysite\messenger\templates\messenger\message_form.html`

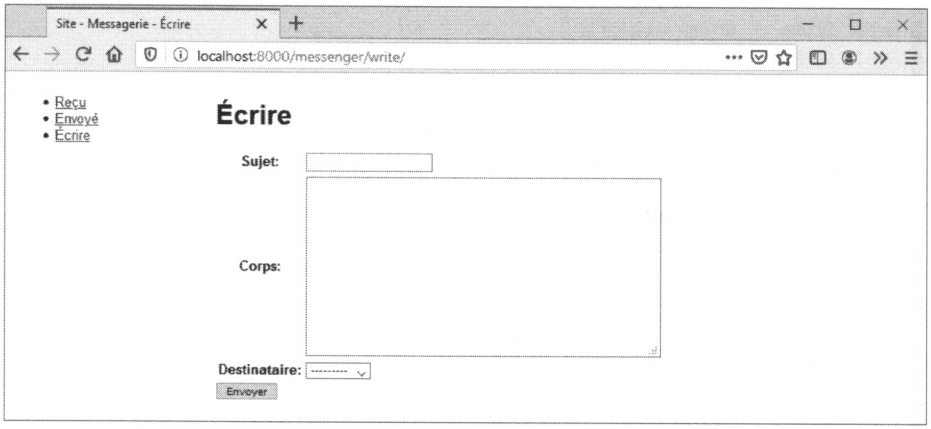
```
{% extends "./base.html" %}
{% block title %}{{ block.super }} - Écrire{% endblock %}
{% block content %}
<h1>Écrire</h1>
<form method="POST">{% csrf_token %}
 <table>{{ form }}</table>
 <button type="submit">Envoyer</button>
```

© Editions ENI - All rights reserved

```
</form>
{% endblock %}
```

La balise `csrf_token` apporte le champ caché de formulaire nécessaire à la sécurité des soumissions, ainsi qu'il a été expliqué dans le chapitre Vues.

La page rendue est peut-être estimée laide, mais elle est fonctionnelle :



Les trois champs de formulaire sont dérivés de la définition du modèle, avec des caractéristiques par défaut :

- Le sujet, modélisé par un `CharField`, est un champ d'entrée de type texte, dont la valeur est requise et la saisie limitée à la taille de l'attribut, soit 120 caractères.
- Le corps, modélisé par un `TextField`, est rendu par une balise `<textarea>` de 40 colonnes et 10 lignes, ce qui explique l'espace occupé.
- Le destinataire, modélisé par un `ForeignKey`, se traduit par une balise `<select>` à choix requis et alimenté par l'ensemble des représentations des objets pointés.

Les libellés sont tirés des noms humains (`verbose_name`) donnés aux champs du modèle, avec la première lettre mise en majuscule.

La mise en page du formulaire est bien sûr réduite à sa plus simple expression dans cet exemple, en laissant la variable donner son rendu par défaut. Des rendus standards sont disponibles sous forme de paragraphe, liste ou tableau.

Pour servir de matière à des manipulations, un formulaire réduit est construit :

```
>>> from mysite.messenger.models import Message
>>> from django.forms import models
>>>
>>> form_class = models.modelform_factory(Message,
... fields=['subject', 'body'])
>>> form = form_class()
>>> form
<MessageForm bound=False, valid=Unknown, fields=(subject;body)>
```

Le rendu par défaut est celui sous forme de lignes de tableau, ce qui justifie dans le gabarit l'enrobage de la variable dans une balise `<table>` :

```
>>> form.as_table()
'<tr><th><label for="id_subject">Sujet:</label></th><td><input
type="text" name="subject" maxlength="120" required
id="id_subject"></td></tr>\n<tr><th><label for="id_body">
Corps:</label></th><td><textarea name="body" cols="40" rows="10"
id="id_body">\n</textarea></td></tr>'
>>>
>>> print(form)
<tr><th><label for="id_subject">Sujet:</label></th><td><input [...]
```

La méthode `as_ul()` donne un ensemble de balises `<li>`, à enrober dans un `<ul>` :

```
>>> from django.template import Context, Template
>>>
>>> c = Context({'form': form})
>>> Template('<ul>{{ form.as_ul }}</ul>').render(c)
'<ul><li><label for="id_subject">Sujet:</label> <input
type="text" name="subject" maxlength="120" required
id="id_subject"></li>\n<li><label for="id_body">Corps:</label>
<textarea name="body" cols="40" rows="10" id="id_body"
>\n</textarea></li></ul>'
```

Enfin, la méthode `as_p()` fait un travail similaire avec des balises `<p>` :

```
>>> Template('{{ form.as_p }}').render(c)
'<p><label for="id_subject">Sujet:</label> <input type="text"
name="subject" maxlength="120" required id="id_subject"></p>\n
<p><label for="id_body">Corps:</label> <textarea name="body"
cols="40" rows="10" id="id_body">\n</textarea></p>'
```

© Editions ENI - All rights reserved

Le fait que les méthodes `as_table()` et `as_ul()` ne rendent pas leur balise mère enveloppante ne doit pas être perçu comme un manque. Au contraire, l'intention voulue est de laisser la main au développeur et lui permettre en particulier de placer tous les attributs souhaités dans le conteneur pour établir son style :

```
<table id="[...]" class="[...]" style="[...]">
 <caption>[...]</caption>
 {{ form }}
</table>
```

L'attribut HTML `name` d'un champ de formulaire a son importance puisqu'il véhicule la valeur lors de la soumission du formulaire et doit être reconnu par le serveur à l'arrivée. Son contenu est directement pris du nom du champ correspondant dans la classe du formulaire, et donc ici du modèle puisqu'il s'agit d'une construction automatique grâce à une méthode `modelform_factory()`.

La syntaxe du code produit est celle du HTML5. On en voit une preuve avec la présence d'un attribut `required`, non existant en HTML 4.01 et devant s'écrire sans minimisation en XHTML, c'est-à-dire `required="required"`. Cet attribut n'est pas produit lorsque le champ du modèle a l'attribut `blank=True`, on peut le constater dans la balise `textarea` du champ `body`.

Ces formats de rendus par défaut, tant pour le formulaire que pour ses champs, ont le mérite d'exister et de présenter des aspects standards. Ils ont notamment un grand intérêt dans des phases exploratoires telles que preuve de faisabilité, expérimentation ou maquettage, où l'aspect présentation est accessoire.

Il est certain par ailleurs qu'on a vite fait d'en vouloir plus et d'être en capacité de décider finement du rendu de chacun des composants. Plusieurs moyens existent pour piloter le rendu, à différents degrés, jusqu'à aller à un formulaire entièrement personnalisé et des composants graphiques également personnalisés.

Pour une première approche du sujet, quelques manipulations sont présentées ci-dessous.

– Itérer les champs d'un formulaire :

```
>>> for f in form: print(f)
...
<input type="text" name="subject" [...]>
<textarea name="body" [...]>
</textarea>
```

– Être capable de répartir les champs dans l'agencement du formulaire :

```
>>> for f in form.hidden_fields(): print(f)
...
>>> for f in form.visible_fields(): print(f)
...
<input type="text" name="subject" [...]>
<textarea name="body" [...]>
</textarea>
```

– Travailler un champ en particulier :

```
>>> subject_fld = form['subject']
>>> vars(subject_fld)
{
'form': <MessageForm bound=False, valid=False, fields=(subject)>,
'field': <django.forms.fields.CharField object at [...]>,
'name': 'subject',
'html_name': 'subject',
'label': 'Sujet',
'help_text': '',
[...]
}
>>> subject_fld.is_hidden
False
```

– Modifier certains attributs du composant graphique :

```
>>> body_fld = form['body']
>>> body_fld.as_widget(attrs={'cols': 12, 'rows': 3})
'<textarea name="body" cols="10" rows="3" id="id_body"
>\n</textarea>'
```

© Editions ENI - All rights reserved

Il est possible d'éviter de passer par la création de classes personnalisées pour simplement apporter quelques retouches à ce qui existe déjà. Pour cela, il faut réussir à surcharger certaines caractéristiques d'un formulaire, d'un champ ou d'un composant graphique.

Pour reprendre l'exemple précédent, le souhait est de donner d'autres dimensions au champ de saisie du corps de message. L'intervention se fait dans la vue `WriteView`, à un endroit où on a la main sur le formulaire. C'est le cas avec la méthode `form_valid()` à l'occasion déjà d'un besoin antérieur d'agir sur le champ expéditeur du formulaire, mais cette méthode entre en jeu trop tard et seulement au moment du traitement de la soumission. Pour intervenir aussi lorsque le formulaire est servi, la méthode `get_form()` est le bon lieu.

▶Ajoutez la méthode :

`mysite\messenger\views.py`

```
[...]
class WriteView(CreateView):
    [...]

    def get_form(self, form_class=None):
        form = super().get_form(form_class)
        form.fields['body'].widget.attrs.update(
            cols='12', rows='3')
        return form
    [...]
```

L'attribut `attrs` étant un dictionnaire, on pouvait aussi écrire :

```
[...].attrs.update({'cols': '12', 'rows': '3'})
```

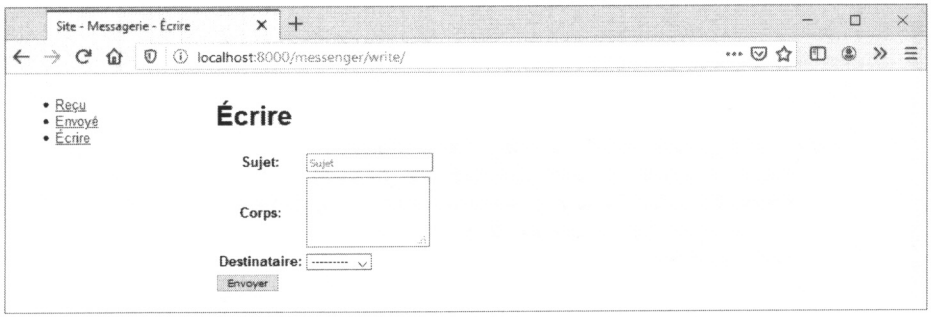
Ce procédé peut être utilisé pour d'autres attributs usuels :

```
        [...]
        field = form.fields['subject']
        field.widget.attrs['placeholder'] = field.label
        return form
```

De façon généralisée, pour une quantité importante de champs ou pour automatiser le procédé et éviter de les nommer individuellement, l'écriture peut évoluer vers :

```
[...]
from django.forms.widgets import TextInput, PasswordInput
[...]
INPUT_WIDGETS = (TextInput, PasswordInput)
[...]
        [...]
        for field in form.fields.values():
            if isinstance(field.widget, INPUT_WIDGETS):
                field.widget.attrs['placeholder'] = field.label
        [...]
```

Le rendu de la page devient :



7.7 Effacement de message

Cette action n'a pas de gabarit dédié. Si on se remémore la vue, il faut faire une soumission, avec les identifiants d'un ou plusieurs messages fournis par un paramètre nommé `pks`. L'action peut être proposée dans deux lieux d'affichage : les dossiers et la consultation d'un message.

Pour commencer par le plus simple, le gabarit de lecture de message est à compléter pour ajouter un formulaire et un bouton.

© Editions ENI - All rights reserved

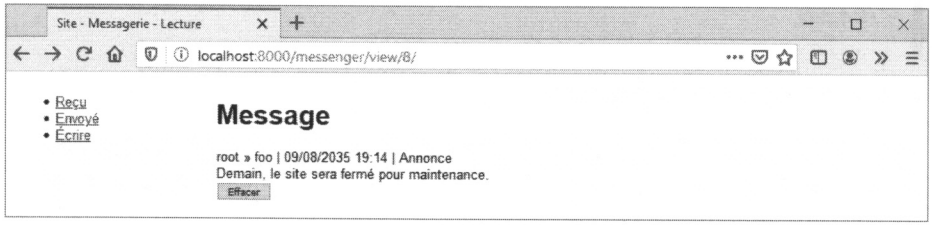
Faites un complément en fin de contenu du gabarit :

`mysite\messenger\templates\messenger\message_detail.html`

```
[...]
<form action="{% url 'messenger:delete' %}?next={% url
'messenger:inbox' %}" method="post">
 {% csrf_token %}
 <input type="hidden" name="pks" value="{{ message.pk }}">
 <button type="submit">Effacer</button>
</form>
{% endblock %}
```

Dans l'URL de destination du formulaire, il est mentionné le paramètre optionnel `next` pour orienter la redirection à venir en cas de succès d'effacement, puisque cette page de départ n'existera plus. Le dossier d'arrivée est un choix raisonnable.

Le nouvel aspect de la page est :



Les dossiers et leur liste de messages sont également à mettre à niveau pour permettre une action d'effacement. Puisqu'on veut pouvoir faire des opérations de masse, une colonne est ajoutée dans le tableau pour placer des cases à cocher.

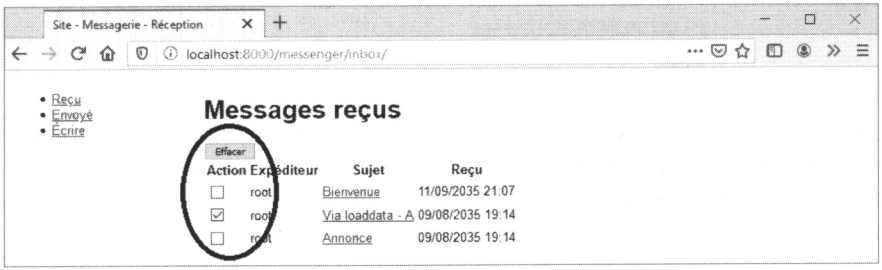
Un bouton est ajouté pour lancer l'action désirée sur les lignes cochées. Dans un esprit plus large, il faut imaginer une barre de boutons avec, selon la nature du dossier, plus ou moins d'autres actions, comme *Undelete, Archive, Mark-As-Read, Mark-As-Unread*. Dans ce cas, chaque bouton devrait être enveloppé dans une balise `block`, de façon à être potentiellement neutralisé par un fils, et l'URL dans l'attribut `action` doit être établie par JavaScript sur l'événement `onclick`.

Modifiez le gabarit pour ces compléments :

`mysite\messenger\templates\messenger\base_folder.html`

```
[...]
{% if msg_messages %}
<form action="{% url 'messenger:delete' %}" method="post">
 {% csrf_token %}
 <button type="submit">Effacer</button>
<table>
 <thead>
  <tr>
   <th>Action</th>
   [...]
 <tbody>{% for message in msg_messages %}
  <tr>
   <td><input type="checkbox" name="pks"
    value="{{ message.pk }}"></td>
   [...]
</table>
</form>
[...]
```

Le nouvel aspect de la page est :



© Editions ENI - All rights reserved

8. Composants de gabarit personnalisés

Une collection intégrée de balises et de filtres pour gabarit est disponible pour couvrir les besoins les plus courants. Une infrastructure logicielle est également proposée pour étendre ce jeu de base et écrire soi-même des composants taillés à son besoin.

L'écriture d'un composant ne s'improvise pas et obéit à des règles bien précises. Pour les respecter, il suffit de suivre deux conseils simples :

– Lire la documentation et l'appliquer.

– S'inspirer des composants existants, lisibles en :

```
django/templates/defaultfilters.py
django/templates/defaulttags.py
```

Le sujet va être exploré avec l'implémentation d'un exemplaire de chacune de ces catégories de composants.

8.1 Balise de gabarit

L'objectif est l'écriture d'une balise de gabarit qui rend le nombre de messages non lus du dossier d'arrivée. Il est choisi de présenter cette pièce d'information dans le menu de l'application, au niveau du lien vers le dossier d'arrivée, là où sa compréhension devrait être implicite.

▶Complétez le menu :

```
mysite\templates\base.html
```

```
{% load static %}
{% load messenger_tags %}
<!DOCTYPE html>
[...]
{% if user.is_authenticated %}<ul>{% messenger_unread as cnt %}
 <li><a href="{% url 'messenger:inbox' %}">Reçu
{% if cnt %} <strong>({{ cnt }})</strong>{% endif %}
 </a></li>
[...]
```

Tout d'abord, la bibliothèque de composants personnalisés doit être chargée. L'opération est demandée avec une balise `{% load %}`. Il n'y a pas d'impératif technique sur l'endroit dans le fichier où placer ce type de balise, à part bien sûr d'être placé avant toute référence à un de ses composants. Dans la situation actuelle, on pouvait la mettre juste avant l'emploi de la balise personnalisée. Attention toutefois à assurer une cohérence entre le lieu de chargement et tous les lieux d'utilisation d'un composant : avec des branchements conditionnels, un chargement non nécessaire peut être épargné, mais le risque est aussi d'introduire un dysfonctionnement s'il vient à manquer. L'usage est plutôt de mettre ces balises en début de fichier, à l'image des instructions `import` dans un fichier Python. La balise de chargement supporte la citation de plusieurs bibliothèques. Il aurait donc été possible de n'écrire qu'une seule balise. Entre les deux extrêmes de style d'écriture, un bon compromis est d'isoler dans un chargement les mentions relatives aux applications intégrées, comme `static` et `i18n`, et d'utiliser une autre balise pour les autres mentions, éventuellement plusieurs si une répartition est justifiée par la quantité ou pour faciliter la compréhension.

Une balise personnalisée nommée `messenger_unread` donne le compte de messages non lus. Ce nombre devant être utilisé plus d'une fois, il est préférable de ne pas solliciter son évaluation à chaque fois.

Cette optimisation est prévue avec la possibilité de diriger le résultat de la balise dans une variable de contexte plutôt qu'en tant qu'affichage, avec une syntaxe choisie à l'identique de celle employée par certaines balises intégrées, par exemple `now` ou `url`, à savoir :

```
as <var>
```

La variable, choisie de nom `cnt` ici, peut ensuite être testée par une balise `{% if %}`, car on choisit de ne montrer le nombre que s'il est utile, c'est-à-dire autre que zéro. L'espace de visibilité d'une variable se limite au bloc dans lequel elle est définie.

Il faut maintenant aménager la bibliothèque et écrire la balise. La disposition dans le système de fichiers obéit à la convention exposée précédemment avec l'exemple de l'application `static`.

© Editions ENI - All rights reserved

▶Créez ce complément de répertoires et fichiers :

```
mysite\messenger\
├ [...]\
├ templatetags\
│  ├ __init__.py
│  └ messenger_tags.py
└ [...]\
```

Si le serveur de développement est en cours de fonctionnement, il ne verra pas ce nouveau module tant qu'il n'aura pas été relancé. Un changement dans un autre fichier quelconque observé suffit à provoquer sa relance et rétablir la synchronisation.

Le fichier `messenger_tags.py` est le conteneur de la balise à écrire et doit correspondre au nom donné dans la balise de chargement.

Pour des cas d'usage simples, le mécanisme d'enregistrement de balise propose deux décorateurs simplifiés :

```
@register.simple_tag
@register.inclusion_tag
```

Ce n'est pas la situation présente et le décorateur complet doit être employé, avec ces deux étapes : une phase de compilation et une phase de rendu.

▶Créez ce fichier avec ce contenu :

`mysite\messenger\templatetags\messenger_tags.py`

```
from django.template import Library, Node, TemplateSyntaxError

from ..models import Message

register = Library()


class InboxCountNode(Node):
   def __init__(self, asvar=None):
       self.asvar = asvar

   def render(self, context):
       try:
           user = context['user']
           count = '' if user.is_anonymous \
               else Message.objects.unread_count(user)
```

```
        except (KeyError, AttributeError):
            count = ''
        if self.asvar:
            context[self.asvar] = count
            return ''
        return count

@register.tag
def messenger_unread(parser, token):
    bits = token.split_contents()
    if len(bits) > 1:
        if len(bits) != 3:
            raise TemplateSyntaxError("'{0}' tag takes no "
            "argument or exactly two arguments".format(bits[0]))
        if bits[1] != 'as':
            raise TemplateSyntaxError("First argument to '{0}'"
            " tag must be 'as'".format(bits[0]))
        return InboxCountNode(bits[2])
    else:
        return InboxCountNode()
```

L'import du modèle utilise une syntaxe relative `..` afin de monter d'un niveau dans la hiérarchie des paquets.

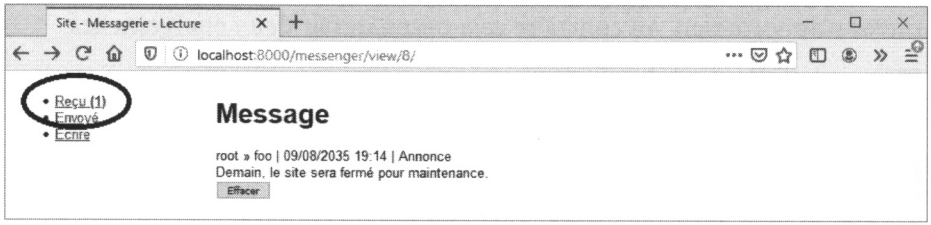
La fonction `messenger_unread()` implémente la phase de compilation. Par défaut, la fonction donne son nom à la balise de gabarit. Son rôle consiste à analyser les portions de la balise, faire des contrôles de validité, puis construire et rendre une instance de classe dérivée de `Node`.

La classe `InboxCountNode` est une sous-classe d'une base `Node` et implémente le rendu à travers sa méthode `render()`. Le retour de cette méthode doit toujours être une chaîne de caractères, éventuellement vide si rien n'est à délivrer. La méthode `__init__()` permet de récupérer un éventuel nom de variable de contexte où le résultat doit être logé.

La connaissance de l'utilisateur est obtenue de l'objet `context` passé à la fonction, grâce au processeur de contexte de `auth`. À condition que l'utilisateur soit connu, l'accès au compte de ses messages non lus est repris d'une méthode du gestionnaire du modèle, introduite dans le chapitre Vues et annoncée par avance comme devant être à nouveau mise à profit pour les gabarits.

© Editions ENI - All rights reserved

Le nouvel aspect de la page est :



Cette balise de gabarit est une alternative au processeur de contexte `inbox` présenté dans le chapitre Vues. Le choix entre les deux procédés peut s'évaluer en fonction de plusieurs critères : goûts personnels, intensité de la diffusion de l'information, préférence pour une mise à disposition implicite ou explicite, etc.

8.2 Filtre

La mise en forme d'un horodatage est l'exemple pris pour écrire un filtre. Le but est d'afficher l'information de façon compacte, c'est-à-dire plus ou moins simplifiée en fonction de l'ancienneté par rapport à l'instant présent. Un format d'affichage différent est admis pour chacune de ces trois situations : le même jour, le même mois, et le reste.

▶Complétez et modifiez le gabarit :

`mysite\messenger\templates\messenger\base_folder.html`

```
{% extends "./base.html" %}
{% load messenger_tags %}
{% block content %}
[...]
    <td>{{ message.sent_at|compact_date:"G:i,j b,j/n/y" }}</td>
[...]
```

Le filtre introduit est appelé `compact_date` et nécessite un paramètre. Étant donné qu'un seul argument est supporté, les trois motifs sont passés dans une chaîne de caractères avec la virgule comme séparateur.

L'implémentation du filtre est logée dans le même module `messenger_tags.py`, qui contient déjà les balises. Séparer les balises et les filtres n'apporte rien, au contraire cela nécessiterait deux chargements.

▶Complétez le module :

`mysite\messenger\templatetags\messenger_tags.py`

```
import datetime
[...]
from django.template.defaultfilters import date
[...]


@register.filter(expects_localtime=True)
def compact_date(value, arg):
    bits = arg.split(',')
    if len(bits) < 3:
        return value  # Invalid arg.
    today = datetime.date.today()
    return date(value,
        bits[0] if value.date() == today
        else bits[1] if value.year == today.year
        else bits[2])
[...]
```

Un filtre s'écrit sous forme de fonction et s'enregistre avec un décorateur. Le principe est simple : accepter une valeur d'entrée `value`, avec optionnellement un paramètre `arg`, et rendre une nouvelle valeur.

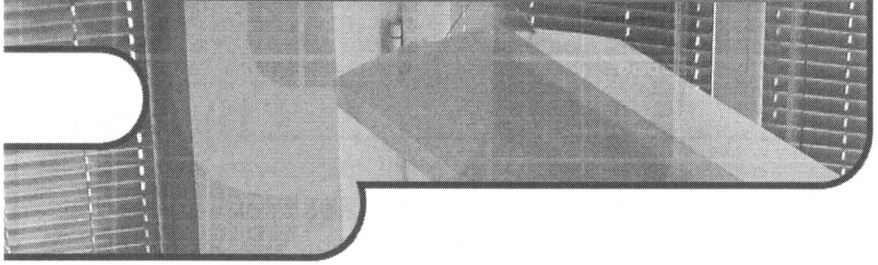
Par défaut, la fonction donne son nom au filtre. L'option `expects_localtime` posée à l'enregistrement précise qu'il est souhaité que l'horodatage passé au filtre soit au préalable converti en heure locale. C'est généralement le cas lorsqu'on manipule des heures liées à des fuseaux horaires.

Ici, la charge du filtre est modeste puisqu'elle consiste à propager le traitement vers le filtre intégré `date`, avec l'un des trois motifs de mise en forme.

© Editions ENI - All rights reserved

Quelques expérimentations manuelles mettent en évidence le fonctionnement du filtre et son rendu (instant présent fixé fictivement à 2035) :

```
>>> import datetime
>>> from django.template import Context, Template
>>>
>>> now = datetime.datetime.now()
>>> t = '{% load messenger_tags %}'
>>> t += '{{ dt|compact_date:"G:i,j b,j/n/y" }}'
>>>
>>> c = Context({'dt': now})  # même jour
>>> Template(t).render(c)
'9:41'
>>>
>>> c = Context({'dt': now + datetime.timedelta(-1)})  # même an
>>> Template(t).render(c)
'8 dec'
>>>
>>> c = Context({'dt': now + datetime.timedelta(-365)})  # -1 an
>>> Template(t).render(c)
'9/12/34'
```



Chapitre 9
Alternatives

1. Propos

Ce chapitre est consacré à la présentation d'alternatives aux choix retenus pour mener la construction du projet exposé tout au long de cet ouvrage. Ils ne sont pas moins valables, mais soit il fallait bien arrêter un choix pour ne pas se disperser, soit il était plus confortable dans un premier temps d'éviter des solutions plus élaborées.

2. Alternance entre plusieurs versions de Django

Django vise un rythme de publication d'une nouvelle version environ tous les huit mois pour apporter des fonctionnalités. On peut toujours se dire que son projet n'en a pas besoin et décider de conserver sa version actuelle puisqu'elle donne satisfaction. Mais cette cadence élevée de publication a comme conséquence l'obsolescence tout aussi rapide des anciennes versions, ce qui signifie un terme à la maintenance corrective des dysfonctionnements et plus sensiblement des failles de sécurité.

Si on ne fait rien de particulier, une installation ordinaire d'une nouvelle version de Django va remplacer la version actuelle. Cette bascule sans retour peut convenir dans des situations simples et sans enjeu, un environnement de démonstration par exemple.

Dans la plupart des cas, il est nécessaire de prévoir une phase de transition, où les projets sont évalués au sein de la nouvelle version de l'infrastructure logicielle et éventuellement ajustés, notamment face aux pratiques dépréciées qui font apparaître des messages d'avertissement, voire d'erreur si on n'a pas anticipé les changements annoncés. Dans tous les cas, il est courant que le développement du projet suive son cours et que, pour des raisons d'engagements de maintenance corrective, il faille s'assurer du bon fonctionnement du projet sur la version précédente, ou même sur un certain nombre de versions anciennes.

C'est encore plus vrai pour un développeur de paquets redistribuables publics, qui par nature ne maîtrise aucunement la progression des montées de version des plateformes utilisatrices de ses paquets.

L'objectif va être d'avoir un moyen efficace pour alterner la version de Django. Ce n'est évidemment pas la méthode brutale de réinstallation de l'une sur l'autre, même si l'opération n'est pas longue.

2.1 Par configuration propre au site

Remarque

Mise en garde :

Cette méthode est un contournement et n'est pas officiellement supportée par l'infrastructure logicielle, mais elle a l'avantage d'être simple et ne nécessite l'emploi d'aucun outil additionnel.

Il a été montré, dans le chapitre Installation, que l'installation de Django dépose ses composants dans deux emplacements sous la racine de Python : `Lib\site-packages\django\` *et* `Scripts\`*. Dans le premier cas, les fichiers appartiennent à un répertoire dédié, il est donc envisageable d'agir par une manipulation de répertoires pour activer un choix parmi plusieurs. Dans le second cas, l'emplacement n'est pas dédié à Django seul, ses fichiers sont là parmi ceux d'autres produits installés.*

© Editions ENI - All rights reserved

L'éventuelle faiblesse de la méthode tient au fait que celle-ci ne prend pas en compte une alternance de ces scripts. Malgré tout, ces scripts sont en très faible nombre : un `.exe`, construit par l'installateur, dont l'effet se veut équivalent à celui du `.py` ; un `.py` servant de point d'entrée pour les commandes en mode ligne. Ces scripts sont très stables, pour ne pas dire invariants. Il y a donc peu à craindre de cette approximation puisqu'il s'agirait d'alterner des exemplaires identiques. Bien sûr, on n'est pas à l'abri d'un contexte différent dans une version future. C'est pourquoi, par prudence, il faut considérer cette méthode admissible, mais pas parfaite.

La technique est basée sur l'emploi de fichiers de configuration de chemins, offert par le module `site` du Python. Il n'y a rien de particulier à mettre en jeu, ce module étant automatiquement importé à l'initialisation de l'interpréteur, car il s'agit d'un comportement par défaut (au contraire, il faut employer une option en paramètre pour le désactiver).

Un fichier de configuration de chemin porte un nom quelconque avec une extension `.pth`. La présence de cette nature de fichier est explorée en deux endroits :

```
D:\Python37\
D:\Python37\Lib\site-packages\
```

La valeur principale du contenu du fichier consiste en des lignes mentionnant des chemins, absolus ou relatifs, qui viendront enrichir la configuration `sys.path`, ce qui permet de donner des orientations au chargeur de paquets.

Supposons que l'arborescence de fichiers de départ soit celle de l'installation telle qu'elle a été réalisée dans le chapitre Installation :

```
Python37\Lib\site-packages\
├─ django\
└─ Django-2.1.5.dist-info\
```

La version actuelle de Django est :

```
D:\>\Python37\Scripts\django-admin --version
2.1.5
```

2.1.1 Apport d'une ancienne version

L'objectif est de mettre en place une version supplémentaire de Django, antérieure à celle actuellement en cours sur le poste.

L'arborescence est modifiée, afin d'aboutir à cette nouvelle disposition :

```
Python37\Lib\site-packages\
├─ Django-2.1.5.dist-info\
├─ Django-2.1.5.manuel\
│  └─ django\
├─ Django-2.0.10.manuel\
│  └─ django\
└─ django.pth
```

- Créez les répertoires `Django-2.1.5.manuel` et `Django-2.0.10.manuel`.

Le choix du nom est libre. Cette écriture est retenue, par convention, pour rester proche du schéma utilisé par `pip`, en remplaçant `dist-info` par `manuel` pour bien souligner le fait qu'il s'agit d'une installation manuelle.

- Déplacez le dossier existant `django` vers `Django-2.1.5.manuel`.

Il est nécessaire de ne pas laisser de paquet `django` immédiatement sous le répertoire `site-packages`, car celui-ci est dans les premières places de la liste de recherche. Sachant que la règle est « le premier trouvé est retenu », cela réduirait à néant les efforts pour orienter la recherche vers d'autres emplacements.

- Téléchargez le fichier `Django-2.0.10.tar.gz`.

Le fichier est à disposition dans la section `download` du site de Django.

Dans cette manière de faire, manuelle et visant une version inférieure à l'actuelle, l'outil `pip` ne doit pas être utilisé, même s'il est possible de préciser le numéro de version dans l'argument (`Django==2.0.10`). La raison est qu'il n'est pas voulu que les composants dans le répertoire `Python37\Scripts` soient impactés, pour ne pas s'exposer à une improbable mais éventuelle régression. À l'inverse, une autre manière, décrite en section suivante, l'exploite.

- Extrayez de l'archive le dossier `django`, le seul nécessaire, vers le répertoire `Django-2.0.10.manuel`.

© Editions ENI - All rights reserved

Créez un fichier `django.pth`, avec ce contenu :

```
Django-2.0.10.manuel
```

Le choix du nom du fichier est libre (à l'exception de quelques noms que des installateurs Python emploient pour leur propre usage). Il n'y a aucune condition à respecter entre le nom du fichier et l'intention exprimée par son contenu. Utiliser le terme `django` ici facilite la compréhension de la cible concernée.

La version visée est désormais en place :

```
D:\>\Python37\Scripts\django-admin --version
2.0.10
```

Le retour à la version d'actualité est aussi piloté par le fichier `.pth`.

Complétez le fichier `django.pth`, pour détenir ce contenu :

```
#Django-2.0.10.manuel
Django-2.1.5.manuel
```

Le caractère # en début de ligne, significatif de commentaires, permet de désactiver la directive. La ligne suivante prend tout son effet.

Ce n'est pas grave si plusieurs lignes sont laissées actives, dans ce cas la première sera effective.

Il suffit désormais de jouer sur la mise en commentaire pour basculer aisément d'une version de Django à une autre.

2.1.2 Apport d'une nouvelle version

Le mécanisme est proche de celui de la section précédente et certaines explications qui y sont données s'appliquent aussi ici.

Cette fois-ci, l'objectif est de mettre en place une version supplémentaire de Django, supérieure à celle actuellement en cours sur le poste.

Supposons qu'une version 3.0.1 vienne d'être mise en diffusion et que nous voulions en disposer.

L'arborescence est modifiée, afin d'aboutir au final à cette disposition :

```
Python37\Lib\site-packages\
├─ Django-3.0.1.dist-info\
├─ Django-2.1.5.manuel\
│  └─ django\
├─ Django-3.0.1.manuel\
│  └─ django\
└─ django.pth
```

- Créez les répertoires `Django-2.1.5.manuel` et `Django-3.0.1.manuel`.
- Déplacez le dossier existant `django` vers `Django-2.1.5.manuel`.
- Installez Django avec `pip`, comme dans le chapitre Installation.

Dans cette manière de faire, manuelle et visant une version supérieure à l'actuelle, l'outil `pip` est employé contrairement à la section précédente, car il est préférable par principe que les composants dans le répertoire `Python37\Scripts\` soient mis à niveau. Il est néanmoins probable qu'ils n'aient pas évolué, et la mise en œuvre d'un `Django-3.0.1.tar.gz`, telle que décrite en section précédente, est acceptable.

L'outil va commencer par conduire une désinstallation de la version 2.1.5 estimée présente du fait de la détection de la présence d'un répertoire `.dist-info` correspondant :

```
Found existing installation: Django 2.1.5
  Uninstalling Django-2.1.5:
    Successfully uninstalled Django-2.1.5
```

La désinstallation va faire son effet sur la partie `Scripts`, mais ne sera pas troublée de ne plus trouver à enlever les éléments de la partie `django` qui a été soustraite par déplacement.

- Déplacez le nouveau dossier `django` vers `Django-3.0.1.manuel`.
- Créez ou complétez le fichier `django.pth`, pour détenir ce contenu :

```
#Django-2.1.5.manuel
Django-3.0.1.manuel
```

Il ne reste plus qu'à jouer sur la mise en commentaire pour basculer aisément d'une version de Django à une autre.

© Editions ENI - All rights reserved

2.1.3 Emploi d'une instance déjà présente

Rien n'interdit de citer dans un fichier de configuration des chemins plus élaborés, en notation absolue ou relative.

Cette faculté peut être mise à profit pour continuer à utiliser des paquets déjà présents sous une installation précédente de Python et s'épargner la charge de les installer à nouveau.

Supposons que la machine fonctionnait jusqu'à présent en Python 3.6 avec un choix de versions de Django, selon cet extrait de l'arbre :

```
Python36\Lib\site-packages\
├─ Django-2.1.5.manuel\
│  └─ django\
└─ Django-2.2.2.manuel\
   └─ django\
```

La volonté est de passer en Python 3.7, et par la même occasion en profiter pour se mettre à niveau en passant à Django 3.0.1. Pour autant, il faudra continuer à vérifier le fonctionnement du projet sous les versions anciennes.

L'arborescence s'enrichit et devient :

```
Python36\Lib\site-packages\
├─ Django-2.1.5.manuel\
│  └─ django\
└─ Django-2.2.2.manuel\
   └─ django\
Python37\Lib\site-packages\
├─ Django-3.0.1.manuel\
│  └─ django\
└─ django.pth
```

Le fichier `django.pth` peut pointer vers les versions de la branche jumelle :

```
#../../../Python36/Lib/site-packages/Django-2.1.5.manuel
../../../Python36/Lib/site-packages/Django-2.2.2.manuel
#Django-3.0.1.manuel
```

Les fichiers `.pyc`, issus d'une compilation et présents en cache, peuvent avoir été générés par une version de Python différente de celle qu'on emploie à ce moment. Il est légitime de s'inquiéter de leur compatibilité et de se demander s'il ne faudrait pas éliminer tous ces fichiers au moment d'alterner. En pratique, les fichiers compilés contiennent un marqueur indicatif de leur format et l'interpréteur en déduit s'il est capable de le lire. Si ce n'est pas le cas, le fichier compilé actuel sera écarté et remplacé par une nouvelle occurrence. Le marqueur changeant si souvent, les chances de conservation du fichier compilé dans cette alternance peuvent être estimées nulles.

2.2 Par environnement virtuel

Le but d'un environnement virtuel est d'être placé dans un espace qui a l'air de se présenter comme le système réel, mais qui pourtant en est isolé, ce qui permet de le personnaliser pour un usage dédié et évite d'affecter le système dans son entier et donc d'interférer avec le fonctionnement pour d'autres usages. Il ne s'agit pas d'une simple duplication massive de fichiers, car un environnement virtuel reste lié à une base vers laquelle il pointe à l'aide de lien symbolique (lorsque la plateforme le supporte).

Python dispose du module `venv` dans sa bibliothèque de base pour créer de tels environnements virtuels.

Les environnements virtuels peuvent être situés n'importe où, mais pour éviter la dispersion une bonne pratique est de les garder groupés sous un même répertoire centralisateur, souvent situé sous son répertoire personnel. Pour faciliter les écritures, un répertoire à la racine d'un disque va être employé. Le choix du nom étant libre, un nom court est adopté, conventionnel et expressif :

```
D:\venvs\
```

Premier environnement

Créez un premier environnement virtuel :

```
D:\>py -m venv --system-site-packages \venvs\py37dj21
```

Il n'est pas requis que le répertoire racine existe au préalable, il sera créé si nécessaire par la commande.

© Editions ENI - All rights reserved

La façon de nommer les environnements mérite une attention particulière, pour s'y retrouver plus tard lorsque les instances d'environnement deviennent nombreuses. En l'occurrence, par le nom `py37dj21`, il est signifié qu'il s'agit d'un environnement basé sur du Python 37 et dans lequel un Django de la série 2.1.x est employé.

En cas de changement d'avis sur le nom d'un environnement existant, imaginer qu'il suffit de renommer le répertoire est une illusion, car ce nom a été enfoui dans certains des fichiers générés. C'est le cas parmi les scripts `.bat`, ce qui pourrait éventuellement se résoudre, car leur format est lisible, mais c'est aussi le cas de fichiers `.exe`, ce qui devient dissuasif. Réfléchir en amont à un plan de nommage stable est une sage précaution pour éviter de gâcher son temps par la suite dans des migrations d'environnements.

L'option `--system-site-packages` donne à l'environnement virtuel l'accès au répertoire `site-packages` du système. De cette façon, il n'est pas nécessaire d'installer la totalité des paquets dans chaque environnement virtuel, si cette règle de répartition est suivie : installer une seule fois au niveau système les paquets suffisamment généraux pour être utiles à une majorité de projets, comme le paquet `pytz` ou les pilotes de bases de données, et installer au niveau virtuel les paquets spécifiques au projet.

L'environnement virtuel est ainsi constitué :

```
venvs\py37dj21\
├─ Include\
├─ Lib\site-packages\
│   ├─ pip...\, pkg_resources\, setuptools...\
│   └─ easy_install.py
├─ Scripts\
│   └─ activate, activate.bat, Activate.ps1, deactivate.bat
│      easy_install..., pip..., python...
└─ pyenv.cfg
```

Le contenu du fichier `pyenv.cfg` révèle de quel Python est dépendant l'environnement :

```
home = D:\Python37
include-system-site-packages = true
version = 3.7.2
```

Même après création de l'environnement, il reste encore possible de changer d'avis quant à l'accès ou pas aux paquets globaux du système, en jouant sur l'entrée `include-system-site-packages`. L'absence de l'entrée équivaut à la valeur Vrai. Toute valeur autre que le mot `true`, indifféremment en lettres minuscules ou majuscules, revient à une valeur Faux. Le format du fichier ne prévoit pas de syntaxe dédiée aux commentaires : seules les lignes au motif mot-clé=valeur, espaces indifférents, sont prises en considération et seulement pour des mots-clés ciblés, le reste est donc simplement ignoré.

Avant de pouvoir utiliser un environnement virtuel, il faut le rendre actif en passant une commande.

▶ Activez l'environnement :

```
D:\>\venvs\py37dj21\Scripts\activate
(py37dj21) D:\>
```

L'invite de commandes informe si on est dans un contexte d'environnement virtuel et, si oui, lequel. L'essentiel de la valeur de l'activation est d'ajouter le chemin vers le répertoire `Scripts` en début de la variable d'environnement `PATH`, ce qui le rend prioritaire dans le parcours de recherche des exécutables, notamment celui de l'interpréteur `python`.

```
(py37dj21) D:\>path
PATH=D:\venvs\py37dj21\Scripts;C:\WINDOWS\system32;C:\WINDOWS;D:\
bin\gettext-utils;
(py37dj21) D:\>where py python
C:\Windows\py.exe
D:\venvs\py37dj21\Scripts\python.exe
(py37dj21) D:\>set
[...]
PROMPT=(py37dj21) $P$G
VIRTUAL_ENV=D:\venvs\py37dj21
_OLD_VIRTUAL_PATH=[...]
_OLD_VIRTUAL_PROMPT=$P$G
```

© Editions ENI - All rights reserved

Dans ce contexte, il est préférable de désigner explicitement la commande `python` plutôt que le lanceur `py` pour éviter une potentielle confusion sur l'interpréteur sollicité, même si la situation apparaît correctement gérée :

```
(py37dj21) D:\dj>py -c "import sys; print(sys.executable)"
D:\venvs\py37dj21\Scripts\python.exe
(py37dj21) D:\dj>py -3.7 -c "import sys; print(sys.executable)"
D:\Python37\python.exe
```

Une autre conséquence de la nouvelle priorité de chemin est que les scripts des paquets installés par la suite peuvent être lancés sans avoir à préciser leur emplacement absolu.

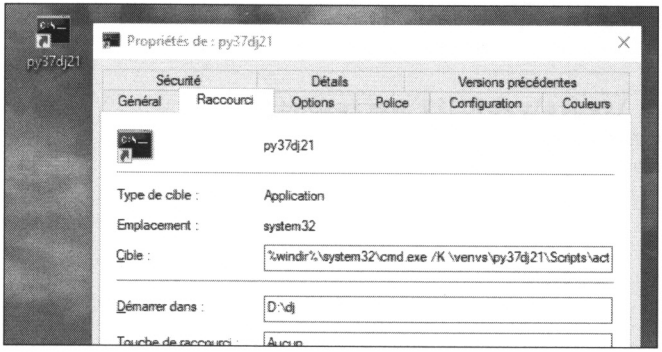
Pour sortir de l'environnement, il convient de le désactiver par la commande symétrique, sans nécessité ici de fournir le chemin :

```
(py37dj21) D:\>deactivate
D:\>
```

Le `PATH` original est restitué :

```
D:\>path
PATH=C:\WINDOWS\system32;C:\WINDOWS;D:\bin\gettext-utils;
D:\>where py python
C:\Windows\py.exe
Information : impossible de trouver "python".
D:\>py -c "import sys; print(sys.executable)"
D:\Python37\python.exe
```

L'activation peut avantageusement être incluse dans des scripts d'ouverture de session de console pour automatiser la procédure :



Sous l'environnement virtuel actif, on peut procéder à l'installation de la version de Django visée.

▶Passez la commande d'installation :

```
(py37dj21) D:\>pip install --no-compile django==2.1.*
Collecting django==2.1.*
  Downloading https://files.pythonhosted.org/packages/c7/87/[...]
  [...]/Django-2.1.7-py3-none-any.whl (7.3MB)
Requirement already satisfied: pytz in
 d:\python37\lib\site-packages (from django==2.1.*) (2018.9)
Installing collected packages: django
Successfully installed django-2.1.7
```

Étant donné qu'il a été demandé la visibilité des paquets système au moment de la création de l'environnement, on constate qu'en effet le paquet `pytz` y est trouvé et il n'est pas nécessaire d'en installer une copie.

Le succès de l'installation se confirme par des interrogations de version.

Exemples

```
(py37dj21) D:\>python -c "import django;[...]
 [...]print(django.get_version())"
2.1.7

(py37dj21) D:\>django-admin --version
2.1.7
```

Les apports sont bien situés dans les répertoires de l'environnement et non ceux du système :

```
(py37dj21) D:\>dir \venvs\py37dj21\Lib\site-packages\d*
 <DIR>          django
 <DIR>          Django-2.1.7.dist-info

(py37dj21) D:\>dir \venvs\py37dj21\Scripts\dj*
 102 808 django-admin.exe
     146 django-admin.py
```

© Editions ENI - All rights reserved

Si on examine le contenu de `django-admin.py`, on pourra constater que la première ligne, dite *shebang line*, désigne l'interpréteur en charge de l'exécution du code et qu'il s'agit bien de celui de l'environnement virtuel. Cet indice étant supporté par le lanceur Python `py` sous Windows, il est tout à fait possible de lancer aussi l'exécution du script par un double clic, sans mettre en jeu le mécanisme de surcharge du `PATH` et d'activation. Ceci est valable pour les scripts installés par `pip`, mais de façon générale les scripts et binaires placés sous environnement virtuel se doivent d'être exécutés par l'interpréteur de cet environnement sans qu'il soit obligatoire de l'activer.

Exemples

```
D:\>py \venvs\py37dj21\Scripts\django-admin.py version
2.1.7
D:\>py \Python37\Scripts\django-admin.py version
2.0.10
```

Deuxième environnement

Pour compléter la démonstration de l'intérêt du procédé de la virtualisation, un exemplaire supplémentaire de Django va être mis en place, en version de série 1.11.x.

- Créez un nouvel environnement virtuel :

```
D:\>py -m venv --system-site-packages \venvs\py37dj111
```

- Installez Django dans cet environnement :

```
D:\>\venvs\py37dj111\Scripts\activate
(py37dj111) D:\>pip install --no-compile django==1.11.*
Collecting django==1.11.*
  Downloading https://files.pythonhosted.org/packages/8e/1f/[...]
  [...]/Django-1.11.20-py2.py3-none-any.whl (6.9MB)
Requirement already satisfied: pytz in
 d:\python37\lib\site-packages (from django==1.11.*) (2018.9)
Installing collected packages: django
Successfully installed django-1.11.20
```

En alternant les activations et désactivations d'environnement, on peut ainsi faire tourner son site ou son paquet sur un nombre quelconque de versions de Django et vérifier son bon fonctionnement dans toutes les situations.

Éliminer un environnement

L'élimination définitive d'un environnement virtuel consiste simplement à le désactiver si nécessaire, puis à supprimer son répertoire avec tout son contenu.

3. Variations de configurations

Le chapitre Création de site a donné plusieurs techniques élémentaires pour ajuster des paramètres de configuration en fonction de l'environnement d'exécution : développement, tests, production, etc. Le revers de leur simplicité était la possible introduction de redondances dans les définitions. Cette section va exposer deux autres techniques, plus complètes, pour perfectionner ce point.

3.1 Superposition en profondeur

Le point de départ est la dernière base de code donnée dans la sous-section «Les techniques par superposition» du chapitre Création de sites, rappelée ci-dessous.

settings.py

```
# Configuration par défaut (production)
# ...

try:
 from mysite import settings_local
except ImportError:
 print("""
   ------------------------------------------------------
    Vous devez créer un fichier settings_local.py
    et y mettre vos paramètres de configuration locale.
   ------------------------------------------------------
   """)
 import sys
 sys.exit(1)
else:
 import re
 for attr in dir(settings_local):
   if not attr[0].isupper(): continue
   value = getattr(settings_local, attr)
```

© Editions ENI - All rights reserved

```
    match = re.search('^EXTRA_(\w+)', attr)
    if match:
      globals()[match.group(1)] += value
    else:
      globals()[attr] = value
```

Dans la configuration par défaut, supposons cette définition ordinaire :

```
TEMPLATES = [
 {
   'BACKEND': 'django.template.backends.django.DjangoTemplates',
   'DIRS': [os.path.join(_SITE_ROOT, 'templates')],
   'APP_DIRS': True,
   'OPTIONS': {
     'context_processors': [
       'django.contrib.auth.context_processors.auth',
       'django.contrib.messages.context_processors.messages',
     ],
     # 'string_if_invalid': , # defaults to an empty string
   },
 },
]
```

Supposons maintenant qu'en contexte de développement il est souhaité, d'une part, de bénéficier d'une aide à la mise au point des gabarits en repérant des variables non valides, et, d'autre part, de disposer d'un processeur de contexte supplémentaire. Pour cela, il faut alimenter l'option `string_if_invalid` et compléter la liste `context_processors`.

Avec le code actuel, toute la définition du paramètre `TEMPLATES` doit être répétée dans le fichier de configuration locale :

`settings_local.py`

```
TEMPLATES = [
 {
   'BACKEND': 'django.template.backends.django.DjangoTemplates',
   'DIRS': [os.path.join(_SITE_ROOT, 'templates')],
   'APP_DIRS': True,
   'OPTIONS': {
     'context_processors': [
       'django.contrib.auth.context_processors.auth',
       'django.contrib.messages.context_processors.messages',
       'django.template.context_processors.debug',
```

```
        ],
        'string_if_invalid': '## variable inconnue : %s ##',
      },
    },
]
```

L'objectif va être de ne définir que le strict nécessaire, c'est-à-dire de désigner ces chemins :

```
TEMPLATES
└ élément d'indice 0
   └ entrée OPTIONS
      ├ entrée context_processors pour ajout
      └ entrée string_if_invalid pour positionnement
```

Formaliser ces chemins, en restant dans le jeu de caractères permis pour des variables, peut se réaliser avec ces formes :

```
EXTRA__TEMPLATES__0__OPTIONS__context_processors
TEMPLATES__0__OPTIONS__string_if_invalid
```

Le motif double-soulignement sert de séparateur pour exprimer le chaînage des nœuds. Le terme EXTRA, en première position, signifie par convention que les valeurs viendront en addition aux valeurs déjà existantes plutôt qu'en écrasement.

De cette façon, le fichier de configuration locale se réduit désormais à :

settings_local.py

```
EXTRA__TEMPLATES__0__OPTIONS__context_processors =
                    ['django.template.context_processors.debug']
TEMPLATES__0__OPTIONS__string_if_invalid =
                                    '## variable inconnue : %s ##'
```

Le code d'interprétation doit être adapté en conséquence, à partir de :

settings.py

```
# ...
else:
  for attr in dir(settings_local):
    if not attr[0].isupper(): continue
    value = getattr(settings_local, attr)
    # ... suite 1
```

© Editions ENI - All rights reserved

On devine le besoin d'une boucle, alors des états de départ sont établis :

Suite 1

```
# suite 1
tokens = attr.split('__')  # (a)
base   = globals()         # (b)
extra = False              # (c)
name = tokens.pop(0)       # (d)
if name == "EXTRA":        # (e)
  extra = True
  name = tokens.pop(0)
# ... suite 2
```

(a) : Le nom de l'attribut est éclaté en une liste de jetons.

(b) : La variable `base` pointe sur l'espace de réception de la définition à apporter. Elle sera amenée à descendre en profondeur au fil des tours de boucle.

(c) : La variable `extra` mémorise le fait d'être en addition ou en écrasement.

(d) : La variable `name` désigne le jeton courant. Le parcours de la liste des jetons se fait en vidant progressivement cette liste par sa tête.

(e) : La détection de présence du mot-clé `EXTRA` en première position est un cas particulier. On mémorise ce constat et on passe au jeton suivant.

Passons à la boucle de parcours des jetons :

Suite 2

```
# suite 2
while tokens:              # (a)
  base = base[name]        # (b)
  name = tokens.pop(0)     # (c)
  if name.isnumeric():     # (d)
    name = int(name)
# ... suite 3
```

(a) : Le parcours se fait jusqu'à épuisement de la liste des jetons.

(b) : On progresse dans la profondeur des structures de stockage.

(c) : Le jeton suivant est acquis.

(d) : Si le jeton, actuellement sous forme de chaîne de caractères, est représentatif d'un nombre, il faut le transformer en véritable numérique pour être considéré comme un indice à l'étape (b) suivante.

Sachant maintenant quel est son destinataire, il reste à injecter la valeur :

Suite 3 et fin

```
# suite 3
if extra:                # (a)
  try:
    base[name] += value
  except KeyError:       # (b)
    base[name] = value
else:                    # (c)
  base[name] = value
```

(a) : Dans la situation d'ajout, on vise l'accumulation de la valeur à la destination.

(b) : La destination d'un ajout est censée être existante. Mais si pour une raison quelconque ce n'était pas le cas, il suffit de se protéger d'un échec en se repliant sur une simple affectation de valeur.

(c) : La situation ordinaire de positionnement de la valeur.

3.2 Exécution de code

Cette technique repose sur l'exécution dynamique de code Python. En Python 2, il était possible d'employer soit l'instruction `exec`, soit la fonction `execfile()`. Ces possibilités n'existent plus en Python 3, elles doivent être remplacées par l'emploi de la fonction `exec()`.

Il n'y pas de nécessité que le fichier de configuration locale ait l'extension `.py` puisqu'il sert simplement de contenant. Lui donner une autre extension semble d'ailleurs préférable pour bien marquer le fait qu'il n'a pas vocation à être considéré comme un module habituel. Prenons par exemple l'extension `.conf` pour rester dans le thème de la configuration.

Le contenu doit être du code Python, ni plus ni moins. Par exemple :

`settings_local.conf`

© Editions ENI - All rights reserved

```
DEBUG = True
INSTALLED_APPS += [
    'tools',
]
MIDDLEWARE += [
    'tools.middleware.DebugConnectionsMiddleware',
]
TEMPLATES[0]['OPTIONS']['string_if_invalid'] =
                                    '## variable inconnue : %s ##'
```

L'incorporation est implémentée en fin du fichier des paramètres de configuration :

`settings.py`

```
# Configuration par défaut (production)
# ...

with open('mysite/settings_local.conf', encoding='utf-8') as f:
  exec(f.read())
```

Le contenu du fichier est lu et soumis à exécution. Il vaut mieux préciser l'encodage à l'ouverture pour ne pas se laisser imposer celui par défaut, qui est dépendant de la plateforme, tel que `cp1252` sous Windows. L'emploi de l'instruction `with` procure une écriture plus propre en évitant de laisser inutilement ouvert le fichier.

L'analyseur de la fonction `exec()` s'attend à des marques de fin de ligne du style Unix. Mais ce n'est pas un souci si on laisse à la fonction `open()` son paramètre `newline` par défaut qui assure ainsi une conversion depuis d'autres styles (Windows, Mac).

Leur importance est subjective, mais les points défavorables de cette technique sont :

- Le passage par une étape de compilation à chaque usage, alors qu'un code binaire déjà présent dans un fichier `.pyc` apporte un gain de performances.
- Le fichier de configuration locale n'étant pas géré comme un module, la sauvegarde d'une modification de son contenu ne provoque pas un rechargement automatique du serveur de développement `runserver`.

Variante

Si un besoin d'une logique combinatoire se fait sentir, plutôt qu'un seul fichier de configuration locale, on peut répartir les paramètres dans une collection de fichiers de façon à faire des compositions à volonté, plus finement. Pour régler la configuration, il n'est ainsi plus nécessaire de modifier du contenu, mais de mettre en service ou hors service certains fichiers.

Un cas d'application peut être de dérouler des tests automatiques dans différentes situations de configuration.

Il semble préférable de garder la main sur l'ordre d'entrée des éléments dans l'assemblage puisqu'il peut y avoir des additions ou des remplacements. C'est pourquoi les fichiers vont être préfixés par un nombre pour permettre leur classement.

Supposons par exemple la présence d'un répertoire pour héberger la collection de fichiers élémentaires de configuration :

```
mysite\
├─ __init__.py
├─ settings.py
└─ confs\
   ├─ 10-base.conf
   ├─ 20-db-dev.conf
   └─ 50-opt-mailer-app.conf
```

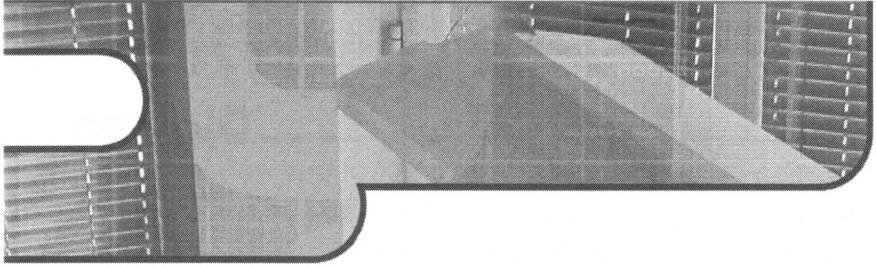
L'incorporation des fichiers de configuration est implémentée ainsi :

settings.py

```
# Éventuellement, configuration par défaut (production)
# ...

import os.path
from pathlib import Path
paths = Path(os.path.dirname(__file__), 'confs').glob('*.conf')
for p in sorted(paths):
  exec(p.read_text(encoding='utf-8'))
```

La méthode `read_text()` (Python 3.5) se charge d'ouvrir, lire et fermer le fichier.

© Editions ENI - All rights reserved



Chapitre 10
Internationalisation

1. Propos

Au tout début de la création d'un site, avec l'idée qu'il est destiné à des personnes francophones, on peut naturellement avoir tendance à faire l'impasse sur l'aspect linguistique. Cette attitude est effectivement valable si une véritable réflexion a été menée en amont pour aboutir à la conclusion que le site n'a pas et n'aura jamais vocation à s'adresser à des personnes de langue étrangère. Elle l'est beaucoup moins dans deux autres postures, à l'opposé l'une de l'autre :

- 1. Se douter que ce sera à faire, mais repousser l'implémentation pour une version future.
- 2. Dans le doute, le faire par principe.

Dans le cas du « On fera plus tard », la raison usuelle invoquée est : « Ce n'est pas la priorité, démarrons d'abord le site en français et on s'occupera du marché international après ». Le raisonnement n'est pas absurde en soi, mais le risque est de constater après coup que certains choix, par exemple d'architecture, du modèle de la base de données, de répartition des ressources graphiques, de composants tiers, sont difficilement voire pas du tout compatibles avec une déclinaison multilingue. Rien n'est irrémédiable, mais le surcoût d'adaptation peut être sensible en comparaison de la prise en compte du sujet dès le départ.

Pour le cas du « Faisons par principe », voici quelques motifs courants : a) si la Direction décide plus tard de proposer une autre langue, on est irréprochable et fier de pouvoir clamer « C'est prévu ! » ; b) le développement est réalisé par une équipe off-shore, ils font tout en anglais et c'est à nous de composer les textes en français ; c) de façon conventionnelle dans le métier, le développeur code déjà en anglais et donc il a décidé de tout faire de façon internationale, car il travaille toujours comme ça. À nouveau, rien de fondamentalement mauvais à cela, sauf que, sachant que tout effort représente un coût, une part de budget sera mobilisée pour une fonctionnalité qui a toutes les chances de ne jamais être exploitée.

Ainsi, pour éviter de se faire piéger par du « pas assez » ou du « trop », il est préférable de mener une réflexion le plus tôt possible sur le sujet du multilinguisme.

Dans ce chapitre dédié à la personnalisation en fonction du public visé, il sera aussi abordé la problématique du temps, car non seulement tout le monde ne parle pas la même langue, mais en plus il n'a pas la même heure. La découpe du globe en tranches, dits fuseaux horaires, est bien utile pour que chacun ait son heure dite locale.

Point de vocabulaire

Que ce soit dans un contexte rédactionnel ou technique, les termes anglais *i18n* et *l10n* sont fréquemment rencontrés. Il s'agit simplement d'une notation conventionnelle pour abréger des mots encombrants car comportant un nombre élevé de signes.

La règle de compression est basique : retenir le premier caractère du mot, donner le nombre de caractères omis du mot, finir par le dernier caractère.

Les significations sont ainsi :

– *i18n* <-> *internationalization*

– *l10n* <-> *localization*

Au terme anglais *internationalization* est associé le mot français « internationalisation » et correspond globalement au fait de traduire un contenu.

© Editions ENI - All rights reserved

Au terme anglais *localization* est souvent associé le mot « localisation ». En français ordinaire, ce mot a avant tout le sens de situer quelque chose dans l'espace ou le temps, ou de limiter géographiquement son envergure. Il serait plus juste de le voir comme appartenant à la catégorie des faux-amis, puisqu'un mot anglais comme *locate* est plus adéquat pour cette signification. Mais avec le temps, l'usage en a été admis et les dictionnaires mentionnent aussi la définition d'adapter quelque chose à une zone géographique. Il serait pourtant plus pur d'employer le mot « régionalisation ».

Un autre abus du mot « localisation » est de l'utiliser à la place du mot « internationalisation », pour un sens global de traduction. Or, son rôle est plus limité, car il consiste à adapter un contenu aux usages régionaux du l'utilisateur. Il s'agit donc plutôt d'une mise en forme pour se conformer à la représentation de l'information à laquelle s'attend son lecteur. Ici, la notion de région doit se comprendre de façon élargie : elle ne se restreint pas à une zone géographique, mais peut signifier une communauté définie en fonction de facteurs culturels.

Prenons l'exemple typique d'une date qu'on veut montrer sous une forme compacte. On ne va pas employer des mots, mais seulement des chiffres, donc il ne s'agit pas vraiment d'une traduction. Par contre, la représentation d'une date prend des formes différentes selon à qui on s'adresse, car celui-ci s'attend à interpréter les chiffres d'après leur positionnement.

Pour la même date du 3 décembre 2018, un Français ou un Anglais s'attend à voir « 03/12/2018 », un Suisse « 03.12.2018 », un Américain « 12/03/2018 » ou « 2018-12-03 ». On comprend bien la confusion potentielle avec la date du 12 mars.

2. Configuration

Abordons le sujet par un examen des paramètres de configuration qui ont une influence incontournable en rapport avec le thème.

settings.py

```
MIDDLEWARE = [
    # ...
    'django.middleware.locale.LocaleMiddleware',
    # ...
```

Il est nécessaire d'ajouter ce composant pour qu'il soit tenu compte des préférences de langues communiquées par le navigateur de l'utilisateur. En son absence, le positionnement des paramètres suivants paraîtront sans effet, sauf cas particulier où la langue est positionnée par programmation.

```
USE_I18N = True
```

Ce paramètre a par défaut la valeur `True`, considérant qu'il est plus probable de vouloir construire un site à visée internationale. Cependant, si rien n'est fait dans le restant du travail pour gérer effectivement le contenu en plusieurs langues, toute la mécanique mobilisée dans la détermination de la langue à servir est inutile. Certains diront que le site fonctionne tout autant, alors pourquoi s'en préoccuper ? Il est vrai que les pages seront servies apparemment sans différence.

Le bénéfice de la présence de ce paramètre est de permettre un gain en performances en désactivant les traitements dont on peut se passer. Les pages seront ainsi servies plus rapidement.

Ceux qui ne sont pas sensibles à l'aspect des performances, au sens consommation de temps, ne devraient pas pour autant ignorer un autre aspect sensible : celui de la consommation de ressources, notamment énergétiques.

© Editions ENI - All rights reserved

À la création du projet, ce paramètre avait été basculé à `False`, à juste raison. Désormais, l'internationalisation va être mise en œuvre. Supprimer la ligne ou la mettre en commentaire sont deux façons de faire valides, mais il est préférable de positionner explicitement le paramètre pour des raisons de lisibilité et d'homogénéité. C'est d'ailleurs la situation dans le fichier de configuration établi lors de la phase de création du projet. Ainsi, on n'est pas obligé de mémoriser cette valeur par défaut, d'autant plus que d'autres paramètres apparentés peuvent avoir une valeur par défaut différente.

```
USE_L10N = True
```

Ce paramètre a par défaut la valeur `False`.

À la création du projet, ce paramètre avait été basculé à `False`, puisque cette fonctionnalité n'était pas utile à ce moment. Désormais, la régionalisation va être rendue effective.

Les paramètres `USE_L10N` et `USE_I18N` restent indépendants l'un de l'autre, même si dans la plupart des situations ils sont positionnés ensemble en cohérence, comme le démontrent les expériences suivantes.

Dans un gabarit, la présentation du moment présent est voulue.

Exemple :

```
{% now "DATETIME_FORMAT" %}
```

Dans le cas où `USE_I18N` est à `False`, il n'y a pas de traduction, le contenu servi est tel qu'il est dans les gabarits de pages.

Avec `USE_L10N` à `False`, les mises en forme des dates et nombres sont régies par d'autres paramètres unitaires, tels que `DATETIME_FORMAT`, qui n'ont pas été positionnés dans le fichier de configuration. Bien sûr, ces paramètres ont des valeurs par défaut et celles-ci sont prévues pour la culture américaine. Ainsi, tout se passe comme si le site était établi en langue `en-us`.

Tous les utilisateurs verront donc cette représentation identique :

```
April 25, 2018, 4:46 p.m.
```

Avec `USE_L10N` à `True`, les mises en forme sont cette fois régies selon le paramètre `LANGUAGE_CODE`, qui a été positionné à `fr`.

Tous les utilisateurs verront donc cette autre représentation :

```
25 April 2018 16:46
```

Il s'agit bien de la convention française d'affichage d'une date et d'une heure, mais sans traduction les mots restent en anglais.

Pour mettre encore plus en évidence, s'il en est besoin, un besoin de cohérence entre ces deux paramètres, malgré leur dite indépendance, prenons la situation : `USE_I18N` est à `True`.

Avec `USE_L10N` à `False`, un Français dont le navigateur est paramétré avec `fr` ou `fr-fr` en tête de la liste des langues d'affichage des pages, et un Grec avec son `el` verront respectivement ces représentations :

```
avr. 25, 2018, 4:46 après-midi
Απρίλ. 25, 2018, 4:46 μμ.
```

Cette fois, on a bien les mots en langue locale, mais la mise en forme est américaine. C'est une combinaison qui donne un résultat vraiment étrange et qu'il vaut mieux éviter.

Avec `USE_L10N` à `True`, l'affichage redevient conforme aux habitudes de chacun :

```
25 avril 2018 16:46
25/04/2018 4:46 μμ.
```

Le tableau suivant résume les combinaisons :

		USE_I18N	
		False	True
USE_L10N	False	April 25, 2018, 4:46 p.m.	avr. 25, 2018, 4:46 après-midi
	True	25 April 2018 16:46	25/04/18 16:46

En pratique, il faut donc estimer qu'il sera dans la plupart des cas nécessaire à la fois d'internationaliser et de régionaliser son site, même par exemple pour un site dont tout le contenu rédactionnel est en anglais, mais qui va présenter une date de validité d'une clé API en retour d'un formulaire de vérification.

© Editions ENI - All rights reserved

Il est possible de positionner soi-même, dans le fichier de configuration, un ou plusieurs des paramètres répertoriés ci-dessous, qui ne seront pris en considération que si la régionalisation n'est pas mise en fonctionnement. Mais à quoi bon puisque, justement, la régionalisation est là pour faire tout ce travail ?

Affichage de dates et heures :

- `DATE_FORMAT`, `TIME_FORMAT`, `DATETIME_FORMAT`
- `SHORT_DATE_FORMAT`, `SHORT_DATETIME_FORMAT`
- `YEAR_MONTH_FORMAT`, `MONTH_DAY_FORMAT`
- `FIRST_DAY_OF_WEEK`

Saisie de dates et heures :

- `DATE_INPUT_FORMATS`, `TIME_INPUT_FORMATS`
- `DATETIME_INPUT_FORMATS`

Gestion des nombres :

- `DECIMAL_SEPARATOR`, `THOUSAND_SEPARATOR`
- `NUMBER_GROUPING`

3. Détermination de la langue de l'utilisateur

Lorsque le paramètre `USE_I18N` est à `True`, on cherche à servir le contenu dans la langue de l'utilisateur. Le serveur détermine la langue préférentielle en déroulant un algorithme qui va balayer plusieurs sources potentielles, dans un ordre de priorité. La première source qui donne l'information établit la langue pour le traitement de la requête. Si vraiment aucune source ne convient, il y a repli sur le paramètre global `LANGUAGE_CODE`.

Dans le cas ordinaire, lorsqu'on ne fait rien de particulier dans la configuration ou le code, la source d'information est l'en-tête HTTP `Accept-Language`.

Exemple

```
Accept-Language: fr-FR,fr;q=0.8,en-US;q=0.5,en;q=0.3
```

Si on considère que la valeur numérique donnée à l'élément `q` exprime un poids de préférence, cette syntaxe exprime ceci de la part de l'utilisateur : « Je préfère avant tout le français de France, mais je veux bien les autres formes de français (Canada, Suisse, Belgique, etc.), sinon de l'anglais américain avant tout autre anglais ».

Il est possible d'employer d'autres techniques, plus impératives, pour fixer la langue sans tenir compte de la liste préférentielle du navigateur.

Elles sont brièvement exposées ci-dessous, dans leur ordre de priorité.

Technique n°1

La technique consiste à véhiculer un code de langue dans l'URL même. Il est fait usage de la fonction `i18n_patterns()` dans la construction des `urlpatterns`, ce qui provoquera l'ajout automatique d'un préfixe avec le code de la langue active au moment du rendu de l'URL.

Ainsi, ce qui est rendu d'ordinaire comme `/television/` sera rendu par exemple sous la forme `/es/television/` lorsque la langue active sera l'espagnol. Le mécanisme est dit automatique puisqu'il n'y a rien à adapter dans les gabarits ou dans le code existant.

Bien sûr, cela suppose que cette langue dite active soit établie une première fois pour qu'elle soit maintenue au cours de la navigation et des sauts d'hyperliens.

Il reste possible de mettre sous préfixe une partie des URL du site et de garder telles quelles d'autres URL génériques. Attention dans ce cas à ne pas s'exposer à un potentiel conflit entre les deux espaces de nommage. Par exemple, une agence de voyage prévoit la page particulière `/en/transit/` à usage des Français, mais a aussi une part de son site sous racine `/transit/` pour la clientèle étrangère. Un visiteur anglais (avec son préfixe `en`) risque fort d'être déconcerté.

© Editions ENI - All rights reserved

Technique n°2

Il s'agit d'exploiter dans la session de l'utilisateur la clé nommée par :

```
django.utils.translation.LANGUAGE_SESSION_KEY
```

Bien sûr, cela suppose qu'on n'a pas écarté la gestion de session, proposée par défaut.

Le positionnement de la valeur de la clé peut être fait dans le code de l'application.

Exemple

```
from django.utils import translation
request.session[translation.LANGUAGE_SESSION_KEY] = 'fr'
```

Cet usage peut notamment être mis à profit dans les cas où l'utilisateur est un inscrit, avec un profil mémorisé dont la langue est un des paramètres de son choix. On peut aussi utiliser cette manière lorsque l'information est tirée d'un système tiers externe, un annuaire par exemple.

Technique n°3

Un cookie nommé par `LANGUAGE_COOKIE_NAME` est employé pour transmettre et mémoriser la langue adoptée.

Le positionnement de la valeur du cookie est à la charge de l'application.

Exemple

```
from django.conf import settings
response.set_cookie(settings.LANGUAGE_COOKIE_NAME, 'fr')
```

De la même façon que son nom, d'autres paramètres dont le nom commence par `LANGUAGE_COOKIE_` permettent d'affiner au besoin les caractéristiques du cookie : durée de validité, chemin, domaine.

Technique complémentaire

Lorsqu'on travaille ainsi le positionnement de la langue de façon personnalisée et qu'on veut permettre à l'utilisateur de basculer de langue à volonté, il est intéressant de regarder aussi les capacités offertes par la vue `django.views.i18n.set_language()`. Une URL associée toute prête, de la forme `setlang/`, est aussi proposée.

Elle est conçue pour faciliter le choix de la langue en relation avec l'interface graphique à proposer à l'utilisateur. Une réalisation courante est une collection de drapeaux si la quantité est limitée ou sinon une liste de sélection.

La vue a un avantage non négligeable : elle implémente elle-même le code mentionné dans les techniques n°2 et n°3, c'est-à-dire la mémorisation du choix dans la session si le fonctionnement en session est en mis en œuvre, et dans le cookie (avant la version 2.1, c'était l'un ou l'autre).

4. Limitation de la quantité de langues supportées

En interne, la plateforme supporte plus de 90 langues et variantes. Par ailleurs, dans l'élaboration d'un site, il est souvent intégré des modules applicatifs tiers qui supportent eux-mêmes un certain nombre de langues, selon le choix de leurs concepteurs. Vous allez aussi décliner vos contenus dans une collection limitée de langues.

Tout ceci peut mener à une cacophonie si on n'y prend pas garde. Imaginons un site construit pour être bilingue, en français et en anglais. Un module applicatif tiers y est intégré pour gérer une section d'actualités. Ce module est capable de présenter des libellés de titrage dans une dizaine de langues.

Que se passe-t-il pour un visiteur espagnol, par exemple ? Il peut lui être présenté un mélange à la fois d'espagnol et d'anglais, ce qui risque d'engendrer un inconfort de lecture. Il vaut souvent mieux faire un repli complet sur la langue par défaut, en anglais dans ce cas.

La déclaration des langues supportées se réalise par l'intermédiaire du paramètre de configuration `LANGUAGES`. Par défaut, il cite toutes les langues de la plateforme. Il est alors utile de le spécifier explicitement de façon à restreindre les langues supportées.

© Editions ENI - All rights reserved

Exemple

```
LANGUAGES = [
    ('en', _('English')),
    ('fr', _('French')),
]
```

Les éléments de cette liste sont des 2-tuples pour : code et nom. Le code est une information technique destinée à être manipulée uniquement par du code. Par contre, le nom a toutes les chances d'être présenté à l'écran pour effectuer un choix et doit le plus souvent l'être en étant traduit. C'est pourquoi il est propre de s'assurer que la chaîne est marquée pour traduction, c'est-à-dire enveloppée dans un appel à une fonction ayant un nom répondant au motif `[*]gettext[_*]`, puisque c'est ainsi que la commande d'administration `makemessages` est capable de repérer les besoins de traduction.

S'agissant d'une langue supportée par la plateforme, on peut se reposer sur le fait que la traduction y est déjà présente. On pourrait donc omettre le marquage. D'ailleurs, même si le marquage est fait, on pourra laisser une chaîne vide dans le fichier `po` créé pour l'application. Mais dans l'hypothèse où on emploie une langue qui ne le serait pas, il vaut mieux appliquer cette écriture dans tous les cas, par principe.

Concernant la fonction de marquage, il n'est pas possible d'importer `gettext` ou `gettext_noop` du module `django.utils.translation`, car cela constituerait une importation circulaire du fait que le module lui-même importe `django.conf.settings` à l'occasion de l'exécution de ces fonctions.

La fonction `gettext_lazy` reste cependant utilisable :

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

Cette implémentation implique que, pour toute utilisation, le nom sera traduit, mais cela ne devrait pas constituer une gêne, car c'est le cas d'usage usuel.

Il s'agit de l'implémentation suggérée dans la documentation actuelle, mais dans des versions plus anciennes, elle mentionnait une implémentation différente :

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

Le nom est ainsi marqué pour faire partie de la collection des chaînes à traduire, mais sa traduction n'est pas utilisée pour la constitution du paramètre `LANGUAGES`.

Cette implémentation reste encore tout à fait valable. C'est d'ailleurs une implémentation équivalente qui est employée pour la définition par défaut des langues. Dans ces écritures sans traduction immédiate, l'important est de ne pas oublier de procéder à une traduction réelle au moment de l'usage du paramètre, si c'est le rendu final voulu.

Exemple

```
{% load i18n %}
<select name="language">
{% for lang in LANGUAGES %}
  <option value="{{ lang.0 }}">{% trans lang.1 %}</option>
{% endfor %}
</select>
```

Dans cette situation, la variable `LANGUAGES` est mise à disposition par la présence de ce processeur de contexte :

```
django.template.context_processors.i18n
```

La balise de gabarit `trans` est apportée par le chargement de la bibliothèque `i18n` cité en première ligne.

© Editions ENI - All rights reserved

Il faudra rester vigilant à cette nuance pour rester cohérent entre la configuration et les gabarits : si on emploie le paramètre `LANGUAGES` par défaut, les noms ne sont pas traduits d'office, mais si on définit le paramètre dans sa configuration et qu'on applique la fonction de marquage donnée en exemple dans la documentation, alors les noms sont nativement traduits et le gabarit ne doit pas faire de traduction.

Si des modules applicatifs tiers sont intégrés au site, il faut aussi penser à vérifier que chacun d'eux supporte les langues retenues, en allant constater la fourniture des traductions sous leur répertoire `locale/`.



.pth, 337
.pyc, 19, 342, 353
@csrf_exempt, 255
@csrf_protect, 255
32 bits, 13
64 bits, 13
_default_manager, 243



Accept-Language, 361
activate, 344
add, 282
add_arguments, 147
ADMINS, 177
Adresse
    configuration, 75
__all__, 253
ALLOWED_HOSTS, 52, 56
APP_DIRS, 54
AppDirectoriesFinder, 292
APPEND_SLASH, 77
app_label, 141
Application
    création, 45
    emplacement, 46
application instance, 79
application namespace, 79
app_name, 80

as_p(), 320
as_table(), 320
as_ul(), 320
as_view(), 192
as_widget, 322
attrs, 323
AuthenticationMiddleware, 202
Authentification, 221
AUTH_PASSWORD_VALIDATORS, 59, 95
AUTH_USER_MODEL, 99
AutoField, 120
auto_now, 102
auto_now_add, 102

B

BACKEND, 53
backends, 207
Balayage séquentiel, 155
Balisage, 281
Base de données
    création, 86
    instanciation, 86
    référencement, 88
    requêtes, 153
BaseCommand, 146
BASE_DIR, 55
_base_manager, 117
base_manager_name, 117
bind_template(), 204
Bitmap Index Scan, 157

© Editions ENI - All rights reserved

blank, 98, 122, 321
block, 304
block.super, 311
Boucle de dépendance, 46
BrokenLinkEmailsMiddleware, 57
bulk_create, 133

C

Cache, 240
cache_control, 241
CBV, 192
Champ, 95
Change Code Page
    Voir chcp
CharField, 96, 319
chcp, 89, 176
check, 40, 88
class-based views
    Voir CBV
Classe, 191
Classement
    aléatoire, 104
    application, 53
    descendant, 104
    intergiciels, 57
    objet, 103
    processeur, 201
    secondaire, 105
--clear, 297
CMS, 283
collectstatic, 297, 302

commands, 146
comment, 307
commit, 259
CommonMiddleware, 77
connection, 159, 163
console, 177
console_debug, 167, 177
Content Management System
    Voir CMS
context, 197
contextmanager, 204
ContextMixin, 196
context_object_name, 244
context_processors, 54, 58
    auth, 54, 205
    csrf, 205
    debug, 58, 165
    i18n, 366
    messages, 58
    request, 58
    static, 293
context_processors.py, 200
Cookie, 223
count(), 130
create database, 142
createsuperuser, 94, 119
create_superuser(), 94
create_user, 118
CreateView, 252, 261
Cross Site Request Forgery
    Voir csrf_protect
csrf_protect, 254

© Editions ENI - All rights reserved

csrf_token, 205, 255, 319
CSRF_TRUSTED_ORIGINS, 271
CSRF_USE_SESSIONS, 256
CsrfViewMiddleware, 255
current_app, 83

D

DATABASES, 55, 59, 86, 159
DatagramHandler, 179
DateTimeField, 101
DATETIME_FORMAT, 359
dbshell, 89
db_table, 109
deactivate, 345
default_manager_name, 117
defaultfilters.py, 327
DEFAULT_LOGGING, 166
defaulttags.py, 327
delete, 128
DeleteView, 266
DeletionMixin, 266
Désinstallation, 340
DetailView, 243
disable_existing_loggers, 167
dispatch(), 193
Django
    installation, 34
django.contrib
    admin, 53
    auth, 53, 78, 93, 205

contenttypes, 53
flatpages, 282
messages, 56, 58, 69
sessions, 56, 221
sites, 283
staticfiles, 44, 56, 291
django.db.backends, 167, 236
django-admin.py, 347
django_migrations, 93, 109, 142, 149
DJANGO_SETTINGS_MODULE, 59, 64
Django Template Language
Voir DTL
DoesNotExist, 127
Données de contexte, 197
drop database, 142
--dry-run, 107, 152
DTL, 276
dumpdata, 117, 135, 143, 144, 145

E

elif, 309
else, 309
--empty, 139
empty, 314
EmptyResultSet, 270
enable_seqscan, 156
Encodage, 87, 89, 176, 353
encoding, 176
Enregistreur, 172
Espace de noms, 78

© Editions ENI - All rights reserved

_execute_query(), 159
exists(), 131
expects_localtime, 332
explain, 157
extends, 304, 308, 311
extra_context, 197

F

FBV, 192
fields, 252
file_appending, 177
FileHandler, 173
FileSystemFinder, 292
filters, 167
findstatic, 296
first(), 128
FIXTURE_DIRS, 141
fixtures, 141, 143
    Voir Instantané
FlatpageFallbackMiddleware, 285
flush, 142
Fonction, 191
force_insert, 125
force_update, 125
ForeignKey, 97, 99, 121, 155, 319
forloop, 314
form_class, 252
form_valid(), 259, 323
FormView, 266

function-based views
Voir FBV
Fuseaux horaires, 33, 332, 356
Voir aussi USE_TZ



Gabarit, 275
balise, 327
écriture, 305
personnalisé, 327
Gestionnaire, 114
get, 127
get_absolute_url(), 263
get_context_data(), 196, 225, 227
get_current_site(), 284
get_field, 110
get_fields, 110
get_flatpages, 290
get_form(), 261, 323
get_form_class(), 261
get_form_kwargs(), 261
getlist(), 267
get_object(), 243, 245, 249
get_or_create, 128
getpreferredencoding(), 176
get_queryset(), 243, 246, 249
get_redirect_url(), 83
get_static_prefix, 293
get_template, 204

© Editions ENI - All rights reserved

gettext, 31, 365
    gettext_lazy, 365
    gettext_noop, 365
    gettext-utils, 32
get_user_model(), 101
get_version(), 35
Greffon, 195
GZipMiddleware, 57

H

handle, 148
handler404, 74
--help, 18, 147
help_text, 113
Heure locale
    Voir Fuseaux horaires
hidden_fields(), 322
HTTP_REFERER, 268
HttpRequest, 192
HttpResponse, 192
HttpResponseForbidden, 215
HttpResponseRedirect, 271

I

i18n, 356
i18n_patterns(), 362
id, 127
IDE, 16
IDLE, 16, 22, 234

if, 309
Importation circulaire, 46, 365
in_bulk, 134
include, 304
include(), 51, 77
inclusion_tag, 329
--indent, 135
IndexError, 128
initial, 107
initial_data, 139
--insecure, 45
INSERT, 123
INSTALLED_APPS, 53, 56
instance namespace, 79
Instantané, 134
Integrated Development and Learning Environment
Voir IDLE
Intergiciel, 183
INTERNAL_IPS, 165
Internationalisation, 355
Introspection, 109, 113
is_active, 99
is_ajax(), 215

J

Jinja2, 276
Journalisation, 153
JsonResponse, 213

© Editions ENI - All rights reserved

L

l10n, 356
LANGUAGE_CODE, 59, 359, 361
LANGUAGE_COOKIE_NAME, 363
LANGUAGES, 364
LANGUAGE_SESSION_KEY, 363
last(), 128
lazy, 210
len(), 130
Library, 330
linebreaks, 318
linebreaksbr, 318
--link, 300
load, 306
loaddata, 135, 142, 144, 145
LocaleMiddleware, 57, 358
LOGGING, 166
login_required, 217
Long Term Support
    Voir LTS
LTS, 12

M

mail_admins, 177
make_context(), 203
makemessages, 365
makemigrations, 106
manage.py, 39, 64

management
voir commands
Manager, 115
max_length, 113
Message, 251
composition, 318
effacement, 265, 324
lecture, 317
META, 268
Meta, 102, 109, 117, 125
_meta, 109
Métadonnée, 102, 109
method_decorator, 218, 241
Method Resolution Order
Voir MRO
MIDDLEWARE, 57, 186, 202, 353, 358
middleware
Voir intergiciel
middleware.py, 185
MiddlewareMixin, 185
MiddlewareNotUsed, 189
migrate, 92, 108, 139, 142
Migration, 90, 105, 139
mixin, 195
Modèle, 85, 219
modelform_factory(), 321
ModelFormMixin, 259
models.py, 96
Moteur, 276
intégré, 276
personnalisé, 278
Moteur de base de données, 23

© Editions ENI - All rights reserved

MRO, 195
MultipleObjectMixin, 266
MultipleObjectsReturned, 127
MultipleObjectTemplateResponseMixin, 266
mutable, 289
MySQL, 23, 88, 96, 98

N

never_cache, 241
--no-compile, 30
Node, 330
--noinput, 300
--noreload, 41
--nostatic, 44
NOTSET, 174
null, 121

O

objects, 114, 117
Object Relation Mapping
    Voir ORM
Objet, 117
    création, 118
    lecture, 126
    mise à jour, 123
    optimisation, 130
    suppression, 128
on_delete, 98, 129
Oracle, 23

ordering, 102
ORM, 105

P

Page, 275
    non dynamique, 282
    structuration, 304
Paramètres de configuration, 52
PATH, 20, 30, 32, 89, 344, 345, 347
PermissionDenied, 215, 246
permission_required, 217
pgAdmin, 24, 154, 157
    pgAdmin4, 86
pickle, 182
Pilote, 29, 343
pk, 127
placeholder, 323
Planificateur, 156, 160
po, 365
postgres, 87
PostgreSQL, 23, 24
post_process(), 302
primary_key, 120
primary key
    Voir pk
Processeurs de contexte, 200
propagate, 167
psycopg2, 29
    psycopg2-binary, 30
py launcher, 17

© Editions ENI - All rights reserved

pyenv.cfg, 343
Python
    installation, 12
    Python 2, 12
    Python 3, 12
PYTHONPATH, 48
Python TimeZone
    Voir pytz
pytz, 33, 343, 346



Q, 247
queries, 163, 165
query, 160
QUERY PLAN, 155
QueryDict, 267, 289
QuerySet, 115, 126, 157, 233, 237
queryset, 243, 245, 248



raw(), 158
RawQuerySet, 158
redirect(), 271
RedirectView, 76, 83, 193
Referer, 271, 273
Régionalisation, 357
related_name, 100, 113
render_to_response(), 196
re_path, 74, 302

RequestContext, 197
RequestSite, 284
Requête AJAX, 212
require_debug_true, 167
Résolution inversée, 78
resolve(), 198
ResolverMatch, 198
reverse(), 83, 264
reverse_lazy(), 264
ROOT_URLCONF, 53, 73
RotatingFileHandler, 178
Routage, 73
RunPython, 140
runserver, 44, 143, 291

S

SECRET_KEY, 52
select_on_save, 125
select_related(), 237, 247
sensitive_post_parameters, 257
Seq Scan, 155
serializers, 148
SessionMiddleware, 202
set_language(), 364
--settings, 64
settings.py, 39, 61, 52
setUp(), 143
setUpTestData(), 143
shell, 22, 234

© Editions ENI - All rights reserved

showmigrations, 91, 106
SimpleLazyObject, 209
simple_tag, 329
Site
    création, 37
    premier lancement, 40
_SITE_ROOT, 55
slug, 244
SocketHandler, 179
socketserver, 180
SQL
    journalisation des requêtes, 163
SQLite, 11, 23
    sqlite3, 42, 55, 86
sqlmigrate, 92, 106
sql_queries, 165
sqlsequencereset, 125, 137
sql_with_params(), 162
startapp, 49, 69, 184
startproject, 38
static, 292
STATICFILES_DIRS, 292, 294, 296
STATICFILES_FINDERS, 292
STATIC_ROOT, 297, 302
STATIC_URL, 60, 291, 292, 294
string_if_invalid, 349
sub, 282
success_url, 263
Super-utilisateur, 87, 93
syncdb, 92, 139
sys.path, 48, 337

--system-site-packages, 343

T

TCP, 179
template tags, 281
template_engine, 209
template_name, 195, 227, 289
template_name_field, 244
template_name_suffix, 244
TemplateResponseMixin, 196
TEMPLATES, 53, 279, 349
TemplateSyntaxError, 330
templatetags, 306
TemplateView, 193
TestCase, 143
testserver, 135, 143
TextField, 96, 319
TimedRotatingFileHandler, 178
TIME_ZONE, 59
Trace, 153, 171
Traduction, 31, 97, 357
trans, 366
TransactionTestCase, 143
Transmission Control Protect
Voir TCP
truncatewords, 314

© Editions ENI - All rights reserved

U

UDP, 179
unordered_list, 318
UPDATE, 123, 126
update(), 131, 270
update_fields, 126, 250
update_or_create, 128
UpdateView, 261
urlconf, 198
urlize, 318
urlizetrunc, 318
urllib.request, 234
urlpatterns, 51, 74, 78, 362
urls.py, 39, 51, 73
USE_I18N, 60, 358
USE_L10N, 60, 359
User, 93, 114
User Datagram Protocol
    Voir UDP
USE_TZ, 60
using, 208, 279

V

validate_password, 119
venv, 342
verbose_name, 97, 102, 319
verbose_name_plural, 102
--verbosity, 107, 136, 296, 300
--version, 21, 32, 35, 337, 339

version, 167
View, 192
Virualisation, 347
visible_fields(), 322
Vue, 191
    écriture, 224
    générique, 194
    intégrée, 193
vue404(), 74

W

wheel, 30
wsgi.py, 39, 59
WSGI_APPLICATION, 59

© Editions ENI - All rights reserved

INFORMATIQUE
TECHNIQUE

Django

Développez vos applications web en Python

(fonctionnalités essentielles et bonnes pratiques)

Ce livre sur **Django** s'adresse aux développeurs qui souhaitent **découvrir ce framework Python sous un angle résolument pratique** avec la mise en place complète et effective d'un environnement de développement et la **conception d'une application web**.

Tout au long du livre, l'auteur utilise comme fil rouge l'exemple du développement d'une application de messagerie interne pour les utilisateurs d'un site, suffisamment représentatif pour que le lecteur étudie les **fonctionnalités incontournables** de Django et les **bonnes pratiques** à mettre en œuvre.

Le lecteur est d'abord accompagné pour l'**installation et la configuration** des outils nécessaires permettant de disposer d'un **environnement de développement** sur son poste avant de découvrir la structure attendue de l'application. Il explore ensuite en détail les notions de **routage** et de **modèles d'objets**. Des chapitres relatifs à la **pose de traces** ou aux **intergiciels** donnent la possibilité de diagnostiquer et de comprendre des traitements internes, en particulier les échanges avec la base de données. Le vaste sujet des **vues**, **pages** et **gabarits** permettant de rendre l'**application dynamique** est également détaillé dans des chapitres dédiés.

Pour finir, l'auteur présente des **alternatives à certaines techniques** présentées ainsi que la mise en œuvre de l'**internationalisation d'une application**.

Bien qu'invité à écrire progressivement les contenus successifs des fichiers du projet développé en exemple, le lecteur pourra en télécharger une copie finale sur le site www.editions-eni.fr.

Patrick SAMSON

Enthousiasmé depuis toujours par l'informatique et la puissance du numérique, **Patrick SAMSON** mène une carrière de développeur indépendant depuis plus de 25 ans. Tout au long de son parcours, il a eu l'occasion de pratiquer plusieurs dizaines de langages de programmation et d'outils. La découverte des qualités du langage Python et du framework Django a été une telle révélation dans son métier qu'il a souhaité partager à travers cet ouvrage son plaisir de concilier les capacités d'un framework et la créativité du développeur.

Sur **www.editions-eni.fr** :
→ Le code source de l'application développée en exemple.

Pour plus d'informations :

39 €

ISBN : 978-2-409-02417-7

eni

www.editions-eni.fr