

Django avancé

Pour des applications web
puissantes en Python

Yohann Gabory

Préface de Nicolas Ferrari

Avec la contribution de Thomas Petillon



Préface

En quinze ans, les techniques de développement web ont beaucoup évolué. J'ai fait mes premiers pas sur Internet au milieu des années 1990 – après la récente introduction de PHP en 1994. Tout était prétexte à la création d'un site, à la fois pour nourrir la curiosité du développeur, mais aussi pour jouir d'une présence en ligne aux yeux de tous, et par delà les frontières. C'était une période de découvertes : on faisait tout au sein d'un même script, qu'il s'agisse d'interagir avec des bases de données, d'effectuer des traitements ou de générer un affichage HTML. Il n'existait alors pas de bibliothèques pour faciliter le développement, à l'exception de quelques frameworks assez lourds dont la prise en main était complexe si l'on était en phase d'apprentissage.

Ce n'est que plus tard, lors de la « bulle Internet » à partir de la fin des années 1990, qu'est arrivée une première vague de frameworks web formalisant, entre autres, un ensemble de patterns de développement permettant d'organiser le code de façon intéressante. Citons le fameux pattern MVC (Modèle-Vue-Contrôleur) qui propose de séparer clairement les données de leur traitement, et de leur affichage.

Fort de ces différents outils et ressources, le développement web s'est encore enrichi à partir de 2005, avec l'introduction de frameworks modernes construits sur des technologies récentes (Ruby on Rails, développé en Ruby) ou en pleine renaissance (Django, développé en Python). C'est d'ailleurs dans ce contexte que nous avons créé en 2006, avec mon ami Cyril, notre plate-forme d'hébergement alwaysdata.com, faite par et pour les développeurs, avec comme ligne directrice la prise en charge d'un maximum de technologies web.

En quelques années, Django a mûri : non seulement techniquement, puisque après une dizaine de versions antérieures, la version 1.5 devrait enfin voir le jour, mais aussi humainement, puisque sa communauté s'est étoffée avec l'organisation régulière de conférences en France et dans le monde. Django a désormais fait ses preuves dans le secteur du développement web, comme en témoignent de nombreuses applications : Disqus (la plus grosse plate-forme de gestion de commentaires), Instagram (rachetée

par Facebook), Pinterest, Mozilla Foundation, Libération, 20 minutes, Washington Post, Century 21, National Geographic – pour n'en citer que quelques-unes.

Cet ouvrage vient en renfort d'une documentation officielle en ligne déjà fort bien faite, tant pour les novices que pour les utilisateurs chevronnés de Django. Son auteur, Yohann, a également voulu démocratiser plus encore son framework de prédilection. Ainsi Yohann s'est affirmé en véritable « évangéliste » en mettant son expérience et ses compétences au service de ce livre. Passionné par Django depuis plus de six ans, sa participation à l'émancipation du framework l'a amené non seulement à réaliser de nombreux développements, mais aussi à donner des formations, des conférences et écrire des publications.

Son approche pour faire découvrir Django est très didactique : après une introduction sur la technologie et la démarche entreprise, il propose d'emblée d'entrer dans le vif du sujet en parcourant un spectre très large des fonctionnalités offertes par le framework, et ce, au fil du développement d'une application de A à Z. Vous disposez ainsi, avec ce livre, d'un des tutoriels les plus complets existant à ce jour sur Django. À l'issue de votre lecture, nul doute que vous aurez acquis les notions nécessaires au développement de votre prochaine application.

En espérant que cette lecture fasse naître des idées, je vous laisse entre les mains de Yohann et, pourquoi pas, sur un fond sonore de Jazz manouche...

Nicolas Ferrari

Développeur Django et co-fondateur d'alwaysdata.com

Table des matières

Avant-propos	1
Pourquoi ce livre ?	1
À qui s'adresse le livre ?	1
Comment ce livre est-il organisé ?	2
Remerciements	3
CHAPITRE 1	
Bien démarrer avec Django : installation et initialisation d'un projet.....	5
Installer un bon environnement de travail	5
Choisir entre éditeur de texte et EDI	5
Installer Python	6
<i>Installation sous Linux</i>	6
<i>Installation sous Mac OS X</i>	6
<i>Installation sous Windows</i>	7
Installer pip et virtualenv	8
<i>Installation sous Linux</i>	8
<i>Installation sous Mac OS X</i>	9
<i>Installation sous Windows</i>	9
Installer un gestionnaire de versions	9
<i>Les différents systèmes de gestion de versions</i>	10
<i>Installer un gestionnaire de versions : le choix Mercurial</i>	11
Installer et tester Django	12
Initialiser un projet	13
Un projet, des applications	13
Les commandes d'initialisation d'un projet	14
<i>Création d'un projet : startproject</i>	14
<i>Serveur web de développement : runserver</i>	14
Conclusion	16

CHAPITRE 2

Créer sa première application : un tracker..... 19

De la méthode et un framework adaptable	19
Le développement par itérations	20
Un framework transparent et adaptable	21
De l'utilité d'un tracker	21
Organisation du travail	21
Gestion du temps	22
Gestion des jalons	22
Historique	22
Tracker et gestionnaire de versions : les outils d'une méthode « agile »	22
Initialisation du projet	23
Mise en place	23
Création d'une application : startapp	23
Vue d'ensemble du projet	23
Paramétrer les variables d'environnement (settings)	24
<i>Mode de développement (DEBUG)</i>	24
<i>Configuration de la base de données</i>	25
<i>Internationalisation</i>	26
<i>Applications installées par défaut</i>	27
Initialisation de la base de données : syncdb	28
Projet versus application	29
Configuration des applications installées	29
Les URL	30
Une première vue	32
<i>L'objet request</i>	34
<i>L'objet HttpResponse</i>	35
Créer une application dynamique	36
Le modèle de l'objet ticket	39
<i>La classe Task</i>	39
<i>La commande python manage.py shell</i>	41
Première itération : l'interface d'administration	43
Ajouter l'application aux settings	43
Ajouter l'application aux URL	44
Un premier test	44
Ajouter le modèle Task à l'administration	45
Une interface CRUD	46
<i>Un premier aperçu</i>	46
<i>Comment sont créées les URL de l'administration ?</i>	47
<i>Les formulaires de l'administration</i>	47
<i>Modifier l'interface d'administration</i>	47

<i>La fonction __unicode__</i>	48
Aller plus loin dans la configuration de l'administration	49
<i>Afficher les due-dates sur l'interface d'administration</i>	49
<i>Changer la couleur des champs due_date et schedule_date</i>	50
Première conclusion	52
Seconde itération : les bases de l'interface client	53
L'application databrowse (contribs...)	53
Une première vue : liste	55
Introduction aux querysets	55
Présenter l'information : les templates	56
Les fichiers templates	57
Un raccourci : render_to_response	58
Troisième itération : l'interface utilisateur, aspects avancés	60
Mise en place des URL	60
<i>Utiliser le dictionnaire GET</i>	62
Les fichiers statiques	64
<i>staticfiles</i>	65
<i>STATIC_URL et render_to_response</i>	65
Une petite astuce	67
Conclusion	68

CHAPITRE 3

Des bases à la production : créer un agenda partagé 69

Fonctionnalités de l'application	70
Un environnement complet	70
<i>Activer l'environnement virtuel</i>	71
<i>Installer les paquets de base</i>	72
<i>Créer un fichier requirements.txt</i>	72
<i>Configuration du projet</i>	73
<i>Plus d'efficacité avec south et local_settings de settings.py</i>	76
Organisation du projet	79
Conception des modèles	79
<i>La relation many-to-many</i>	80
<i>L'option through</i>	80
Création des tables	80
<i>Première migration avec south</i>	81
<i>Test des modèles et TDD</i>	82
<i>Les tests dans les docstrings</i>	83
Conclusion	90

CHAPITRE 4

Gestion de l'authentification 91

Authentifier un utilisateur sur le site	91
Ajouter les URL login et logout	91
Créer le template login	93
Personnaliser le contenu	95
<i>Une page pour chaque utilisateur</i>	95
<i>Une introduction aux vues génériques : TemplateView</i>	96
<i>Afficher les informations de l'utilisateur connecté</i>	97
Autoriser ou interdire certaines parties du site	97
Découplage de l'application	98
Instancier un formulaire	100
Valider les données du formulaire	101
Enregistrer les données du formulaire	102
La fonction create_account finalisée	103
Les tests unitaires de l'authentification	103
La classe Client	104
Test des URL	104
Les tests des formulaires	106
Les tests d'échec	108
Derniers ajustements	109
Modifier les URL	109
Modifier les templates	112
Conclusion	114

CHAPITRE 5

Création d'un événement 115

Les ModelForm	115
Création du formulaire	116
<i>Présenter l'objet Evenement à l'utilisateur</i>	119
<i>Ajouter des participants</i>	121
Améliorations et finitions	125
Les contraintes	125
<i>L'attribut unique pour la contrainte d'unicité</i>	125
<i>L'attribut unique_together</i>	126
Rendre intelligent le formulaire	127
<i>Présélectionner l'événement</i>	127
<i>Supprimer les choix inutiles</i>	128
<i>Supprimer un participant</i>	129
<i>N'afficher que les champs utiles</i>	131

L'interface CRUD	132
<i>Lister tous les événements</i>	132
<i>Supprimer un événement</i>	134
<i>Modifier un événement</i>	134
Conclusion	135
Les URL : #agenda/personal_calendar/urls.py	135
Les vues : #agenda/personal_calendar/views.py	136
Les formulaires	137
Les modèles (sans les docstrings) :#agenda/models.py	137
Le template liste.html	138
Le template create.html	138
Le template details.html	139
CHAPITRE 6	
Les templates pour l'affichage.....	141
Fonctionnement général : découplage et souplesse pour l'intégration avec CSS/JavaScript	141
Organisation des templates	142
Le fichier base.html	142
La directive extends	144
La directive include	146
Les styles CSS et les images	149
<i>Servir les fichiers statiques durant le développement.</i>	149
Inclure un framework CSS dans Django	150
Intégrer du JavaScript	151
Calendrier JavaScript	151
Pliage et dépliage	153
Les requêtes Ajax	156
Conclusion	161
CHAPITRE 7	
Les vues génériques pour les fonctions d'agenda.....	163
Les vues génériques depuis Django 1.3	163
Les bases des vues génériques	164
Afficher une collection d'événements	164
La pagination	167
La pagination dans les vues	168
Un peu de nettoyage	172
Afficher les détails d'un événement : DetailView	173
Conclusion	174

CHAPITRE 8

Une revue de l'application : factorisation du code..... 175

Utiliser les vues génériques dans l'ensemble de votre application	176
La vue <code>Evenement_Detail</code>	176
<i>Factorisation du formulaire</i>	178
<i>Des URL DRY.</i>	180
Créer, modifier, supprimer des événements	182
<i>La vue <code>CreateView</code> : création d'un nouvel événement.</i>	182
<i>La vue <code>UpdateView</code> : mettre à jour un événement</i>	183
<i>La vue <code>DeleteView</code> : supprimer un événement</i>	183
Conclusion	184

CHAPITRE 9

Les notions de partage 189

Le besoin : gérer sa propre liste d'utilisateurs connectés	189
Spécification des modèles	190
<i>Définition des modèles</i>	190
<i>Les requêtes courantes</i>	191
Gestion des invitations	192
<i>Ajouter le champ email à la création d'un nouvel utilisateur</i>	192
<i>Le modèle <code>Invitation</code></i>	193
<i>Le formulaire d'invitation</i>	194
<i>Envoi des e-mails avec Django.</i>	195
<i>Créer le contact dans le carnet d'adresses de l'utilisateur</i>	197
<i>Rediriger vers la fiche contact et non l'invitation</i>	198
<i>Revue de code</i>	200
Les vues	200
<i>Le formulaire d'ajout d'un cercle</i>	201
<i>Le formulaire de modification d'un contact</i>	201
Relier les applications <code>Calendrier</code> et <code>Agenda</code>	202
Présenter les utilisateurs du carnet d'adresses	202
<i>Ajouter l'utilisateur créateur de l'événement à la liste des participants.</i>	203
<i>La gestion des statuts.</i>	203
<i>Empêcher la suppression de l'hôte</i>	204
<i>L'envoi d'un e-mail à l'ajout d'un participant</i>	204
<i>Laisser l'utilisateur indiquer son statut</i>	205
Conclusion	211

CHAPITRE 10

Django et les bases de données	213
Tour d'horizon des bases de données relationnelles	213
MySQL : SGBD historique	213
PostgreSQL : robustesse	214
SQLite : légèreté	214
Comment choisir sa base de données ?	215
<i>Tour d'horizon des bases de données NoSQL</i>	215
L'ORM de Django : couche d'abstraction entre la BDD et les objets	217
Principe de fonctionnement	217
Une vision objet	217
Les principaux avantages : manipuler les objets de la BDD en Python	218
Les inconvénients : comment les contourner	218
<i>Anatomie d'une queryset</i>	218
<i>Méfiez-vous des templates</i>	220
Les différents types de champs	221
Les champs CharField	221
Les champs IntegerField	222
Les champs TextField	222
Les options remarquables	222
<i>Options relatives aux formulaires</i>	222
<i>Options relatives à la base de données</i>	222
Créez vos propres types de champs	223
Les relations	225
Les relations one-to-one	226
<i>Cas d'utilisation</i>	226
<i>Implémentation</i>	227
<i>Queryset sur une relation one-to-one</i>	227
Les foreign keys (clés étrangères)	228
<i>Implémentation</i>	228
<i>Queryset sur une relation foreign key</i>	228
Les relations many-to-many	230
<i>Implémentation</i>	230
<i>Queryset sur une relation many-to-many</i>	230
<i>related_name</i>	231
<i>Manipulation des relations many-to-many</i>	231
Les héritages de tables	232
Héritage abstrait	232
<i>Les classes abstraites et les applications réutilisables</i>	233
Multitable	234

Proxy	234
Les méthodes des modèles	234
Quelques remarques : gérer les appels à la base de données	235
Les meta	236
<i>Les échanges avec la base de données</i>	236
<i>L'ordre des résultats</i>	236
<i>Les contraintes d'unicité</i>	237
<i>Les permissions</i>	237
Les méthodes prédéfinies	237
Les querysets	239
Description d'une queryset	239
Les relations dans les querysets	240
Le coût des querysets	241
Les managers	242
Le fonctionnement des managers	242
Faire ses propres managers	243
<i>Ajouter de nouvelles méthodes</i>	243
<i>Modifier le fonctionnement du manager</i>	244
<i>Un exemple complet pour comprendre l'intérêt des managers</i>	245
L'objet Q : encapsuler des paramètres pour exécuter des requêtes	247
Description de l'objet Q	247
Quand faut-il utiliser l'objet Q?	248
Conclusion	249

CHAPITRE 11

Le traitement de l'information : les vues..... 251

Tour d'horizon des vues	251
Le workflow standard	252
L'objet request	252
<i>La méthode</i>	253
<i>Les en-têtes</i>	254
<i>Les informations fournies par Django dans la requête</i>	254
L'objet response	255
<i>Les codes HTTP fournis par Django</i>	255
<i>Les autres codes HTTP</i>	256
La compilation du template	256
<i>Le principe de base de la compilation d'un template</i>	256
<i>render_to_response</i>	257
<i>L'objet RequestContext</i>	258
<i>render</i>	259
<i>Les context processors</i>	259

Les vues et les objets de modèles	260
Les vues avancées	261
Organiser les vues d'un projet	261
Cas particulier : Ajax	262
<i>Reconnaître une requête Ajax.</i>	262
<i>Le contexte JSON.</i>	263
<i>serializer.</i>	263
<i>Les relations et JSON.</i>	263
<i>Aller plus loin dans la sérialisation.</i>	264
Cas particulier CSV : format d'échange de données tabulaires	265
<i>Renvoyer du CSV avec le moteur de templates Django.</i>	266
Cas particulier PDF	267
<i>Un exemple simple</i>	267
Avant et après la vue : les middlewares	268
Les vues génériques	269
Un exemple simple : <code>TemplateView</code>	269
<code>ListView</code> et <code>DetailView</code>	271
<i>ListView et queryset</i>	272
<i>Les décorateurs et les vues génériques</i>	273
Les mixins	274
<i>MultipleObjectMixin.</i>	275
<code>CreateView</code> et <code>DeleteView</code>	275
<i>CreateView</i>	275
<i>Le template CreateView.</i>	275
<i>Les paramètres optionnels de CreateView.</i>	276
Les autres vues génériques	276
<i>Les vues génériques basées sur les dates</i>	276
<i>Les vues génériques de dates de Django.</i>	277
Les vues-classes	278
Vues-classes vs vues-fonctions	278
Les verbes REST et les vues-classes	279
Les héritages de vues	279
Du bon usage des vues-classes	280
<i>Utiliser les capacités d'agrégation de Django.</i>	284

CHAPITRE 12

L'affichage de l'information : les templates..... 285

Le langage de templates	285
Le template et le contexte	285
<i>Afficher une variable</i>	286
<i>Afficher une série de variables.</i>	287

<i>Branchements logiques</i>	287
<i>Les filtres</i>	289
Revue de code	289
L'organisation des templates	289
Un dossier template pour le projet	290
Un dossier template par application	290
Les dossiers incluses	291
Créer des template tags et des filtres	291
Les filtres : modifier une variable	292
<i>Structure de fichier</i>	292
<i>Écriture d'un premier filtre</i>	292
<i>Les filtres avec argument</i>	294
Les tags : agir au niveau du contexte	294
<i>Les décorateurs</i>	296
<i>Ajouter des arguments aux tags</i>	296
Conclusion	298

CHAPITRE 13

Dialogue avec l'utilisateur : les formulaires 299

L'objet Form	299
Un premier exemple : formulaire d'enregistrement à une newsletter	300
<i>Traitement du formulaire</i>	300
<i>Affichage du formulaire</i>	301
Les champs de formulaire	302
Les options remarquables d'un champ de formulaire	303
<i>L'option required</i>	303
<i>La validation des champs de formulaire</i>	304
<i>Les options d'affichage</i>	307
<i>Les widgets</i>	308
<i>Modifier l'affichage HTML d'un formulaire</i>	312
Conclusion	314

CHAPITRE 14

Django et le protocole web 315

Les verbes du Web	315
GET et POST	315
Les verbes REST	316
Un exemple concret : mise en place d'une API complète	317
<i>Installation de Mezzanine</i>	317
<i>Description de OAuth2</i>	318
<i>Création de l'authentification OAuth2</i>	318

<i>Création de l'API</i>	319
<i>Sécuriser l'API</i>	320
<i>Authentification</i>	321
<i>Limiter l'accès aux ressources</i>	322
<i>Créer un client</i>	322
<i>slumber un client d'API REST</i>	324
<i>En bref</i>	325
Une API REST en détail	326
<i>Retrouver une ressource unique</i>	326
<i>Créer une nouvelle ressource</i>	327
<i>Modifier une ressource</i>	328
<i>Supprimer une ressource</i>	330
<i>Filtrer les ressources</i>	330
Conclusion	333

CHAPITRE 15

Les contribs Django **335**

Le module auth : authentification des utilisateurs	335
Installation	335
<i>Choix du backend</i>	336
<i>RemoteUserBackend</i>	336
L'objet User	337
<i>Création d'utilisateur</i>	338
<i>Activer un utilisateur</i>	339
<i>Connecter un utilisateur</i>	339
<i>Déconnecter un utilisateur</i>	342
<i>Vérifier qu'un utilisateur est connecté</i>	343
<i>La gestion des utilisateurs connectés dans les templates</i>	344
Les permissions	345
<i>Les permissions d'un utilisateur</i>	345
Les groupes	347
Créer ses propres permissions	349
En bref	349
Le module admin : gérer l'interface d'administration de Django	350
Cas d'utilisation du module admin	350
<i>Le prototypage</i>	350
<i>L'interface d'administration à usage de maintenance</i>	353
<i>L'interface d'administration pour les usagers</i>	354
L'interface d'administration par l'exemple	355
<i>Présenter une liste des objets</i>	355
<i>Rechercher dans la liste d'objets</i>	360

<i>Les formulaires dans l'interface d'administration</i>	362
Aller plus loin avec l'interface d'administration	366
<i>Les méthodes spéciales des ModelAdmin</i>	366
En bref	367
Les autres modules contribs	368
Les commentaires	368
<i>Activer les commentaires pour votre projet</i>	369
<i>Utilisation des commentaires dans les templates</i>	369
<i>Prendre le contrôle des commentaires Django</i>	370
<i>En bref</i>	371
Les content-types	371
<i>Une application de tag utilisant les content-types</i>	371
<i>En bref</i>	374
L'application flatpages : gérer les parties statiques de votre site	374
<i>Installation</i>	374
<i>Fonctionnement</i>	374
Le framework de message	375
<i>Installation</i>	375
<i>Utilisation</i>	375
L'application sitemaps : produire une carte de votre site	376
<i>Installation</i>	377
<i>La classe Sitemap</i>	377
<i>Les raccourcis pour la création de sitemaps</i>	378
<i>En bref</i>	379
Le framework de site : gérer plusieurs sites à la fois	379
<i>Installation</i>	379
<i>Utilisation</i>	379
Le framework de syndication : créer un flux RSS	381
<i>Installation</i>	381
<i>Utilisation</i>	381
<i>Aller plus loin avec les flux RSS</i>	382
En bref	383
Conclusion	383

CHAPITRE 16

Django avancé **385**

Percer les secrets de l'interface d'administration	385
Les templates de l'administration	385
<i>Révision des règles de surclassement de templates</i>	386
<i>Une application thème</i>	387
<i>Surclassement des templates d'administration</i>	387

<i>Surclasser change_list_result</i>	387
En faire plus avec les vues : générer modèles et URL à la volée	388
Une application de prototypage	389
<i>Les URL</i>	389
<i>Les vues</i>	390
<i>Les modèles</i>	391
<i>Les templates</i>	393
Aller plus loin avec les templates	394
Utiliser Django en dehors de Django : déjouer l'anicroche	395
Index	397

Avant-propos

Le Web a pris une place considérable dans notre quotidien et rend d'innombrables services. À mesure que les sites sont devenus plus riches et interactifs, les outils pour les créer ont évolué, faisant émerger des standards, des techniques et des architectures logicielles variés.

Le besoin s'est alors fait sentir de reformuler ce travail au sein d'ensembles cohérents permettant de manipuler des concepts abstraits, et avec des méthodes de travail adaptées aux nouveaux standards. C'est ainsi qu'est né le framework Django, offrant une boîte à outils pour créer des applications riches et modernes... « pour perfectionnistes pressés », comme le disent ses créateurs ! Nous vous invitons au travers de ce livre à en découvrir toute la puissance.

Pourquoi ce livre ?

Nombreux sont les sites français écrits avec le framework Django, et pas des moindres : Libération, 20minutes, Autolib, etc. Et pourtant, point de documentation en français à destination des développeurs web francophones. Ce livre vient combler cette lacune et expose tous les concepts fondamentaux à maîtriser, en les illustrant d'exemples et de cas pratiques.

À qui s'adresse le livre ?

Ce livre vient prolonger l'ouvrage *Apprendre la programmation web avec Python et Django* de Pierre Alexis et Hugues Bersini. Il suppose chez le lecteur, qu'il soit amateur éclairé ou développeur professionnel, une connaissance de Python. Il s'adresse

aussi bien au développeur souhaitant aborder un nouveau framework, qu'à celui ou celle qui ne connaît pas encore le développement web.

Quant au lecteur qui connaît et utilise déjà Django, il verra au fil d'exemples réels comment résoudre certains problèmes complexes auxquels il peut être confronté.

Comment ce livre est-il organisé ?

Ce livre est composé de trois parties qui suivent la progression du lecteur dans sa maîtrise de Django.

La **première partie** est composée de deux exemples d'apprentissage. Le premier explique les fondement de Django, son installation, sa prise en main et la création d'une première application, à savoir un **outil de suivi de bogues**. Le second permet très tôt de manipuler les concepts plus avancés de Django en créant un **agenda partagé**.

Dans la **deuxième partie** de ce livre, vous apprendrez à utiliser les différentes facettes de Django avec des exemples permettant de reproduire facilement les concepts que nous aborderons : gestion de la base de données, traitement de l'information et manipulation de formulaires seront au programme.

Enfin, dans la **troisième partie** de ce livre, vous découvrirez des méthodes bien plus avancées qui vous permettront de tirer parti de Django de manière parfois inattendue : après une revue des applications contribs vous manipulerez et créerez des API REST avec une authentification OAuth2. Nous dévoilerons ensuite quelques secrets du framework Django tirés de notre expérience professionnelle, avec entre autres un générateur de classes pour le prototypage d'applications.

1

Bien démarrer avec Django : installation et initialisation d'un projet

Après l'installation des outils qui nous serviront tout au long de l'ouvrage, nous initialiserons un premier projet.

Installer un bon environnement de travail

L'environnement de travail dans lequel vous allez évoluer et les outils utilisés sont très importants pour la qualité future de votre développement.

Choisir entre éditeur de texte et EDI

En tant que développeur, vous avez sans doute un outil de prédilection que vous ne changeriez pour rien au monde. Certains travaillent avec un « simple » éditeur de texte, comme Notepad++ (sous Windows), Gedit ou Geany (sous Linux), ou encore TexMate (sous Mac OS X). D'autres optent pour un environnement de développement intégré (EDI) comme Eclipse, surtout s'ils viennent du monde Java. Mais peut-être préférez-vous ces éditeurs d'un autre âge que sont Emacs ou Vim, certes antédiluviens mais terriblement puissants ? Dans tous les cas, Django s'intégrera parfaitement à la solution que vous aurez retenue.

Installer Python

Python est le langage de programmation sur lequel est basé Django, son installation est nécessaire (et nous conseillons son apprentissage).

RÉFÉRENCES

- 📖 G. Swinnen, *Apprendre à programmer avec Python*, Eyrolles.
- 📖 P. Alexis et H. Bersini, *Apprendre la programmation web avec Python et Django*, Eyrolles.

VERSION Pourquoi Python 2.7 ?

C'est en décembre 2008 que la version 3.0 du langage Python a vu le jour. Elle est incompatible avec les précédentes versions, numérotées 2.x. Or c'est la version 2.7 du langage qui peut être vue comme celle stable à ce jour.

L'équipe de développement de Django est en train de mener un énorme travail de refonte afin que le framework devienne compatible avec la série 3.x. Ce travail est aujourd'hui bien avancé, et un support bêta de Python 3 a vu le jour dans la toute récente version 1.5 de Django. Il faudra attendre la version 1.6 du framework pour que Django soit pleinement compatible avec Python 3.

Installation sous Linux

L'installation de Django sur un système GNU/Linux est très simple. Qu'il soit au format RPM (CentOS, RedHat, Fedora...) ou sous forme de paquet `.deb` (Ubuntu, Debian...), votre système fournit tout le nécessaire pour une installation simplifiée. Vous n'avez donc rien à faire pour le moment. En effet, c'est votre gestionnaire de paquets qui se chargera d'installer pour vous Django et ses dépendances, notamment Python, s'il n'est pas déjà installé sur votre machine.

Installation sous Mac OS X

Sous ses dehors de système d'exploitation du XXI^e siècle, Mac OS X est en réalité basé sur un très vieux système : Unix – plus précisément BSD, un Unix libre. C'est d'ailleurs la raison du « X » de Mac OS X. Inventé dans les années 1970, ce système est particulièrement bien adapté à un environnement de développement. Ainsi, vous n'aurez pas de difficultés majeures à installer tout ce dont vous aurez besoin pour faire fonctionner Python et Django.

Xcode

Afin de bénéficier du meilleur environnement qui soit, nous vous conseillons fortement d'installer Xcode. C'est un ensemble de logiciels gratuits, à télécharger sur l'App Store, qui permet de transformer votre ordinateur en plate-forme de développement.

Si votre choix n'est pas encore fixé sur un éditeur de texte en particulier, vous pourrez utiliser celui livré avec Xcode.

Python

Python étant installé par défaut sous Mac OS X, vous n'avez rien de plus à faire pour cette première partie.

Installation sous Windows

Windows n'étant pas le plus accueillant des systèmes d'exploitation pour le développement informatique, quelques étapes de préparation sont nécessaires avant d'avoir un environnement fonctionnel.

Cygwin

Tout d'abord, installez Cygwin (<http://www.cygwin.com>) qui va créer une base accueillante pour Python et Django, notamment par le biais d'une console. Il vous sera demandé les outils que vous souhaitez installer. Voici ceux dont vous aurez l'utilité :

- Python ;
- Mercurial (un gestionnaire de versions, voir plus loin) ;
- GCC et GCC+ (pour compiler en langage machine certains modules qui seront utiles tout au long de ce livre) ;
- Wget (petit utilitaire fort pratique qui installe les outils de base de Python).

À la fin de l'installation, n'oubliez pas de créer une icône sur le Bureau, comme cela vous est proposé, afin de retrouver plus rapidement la console ou le terminal dont vous allez faire usage régulièrement. Double-cliquez sur cette nouvelle icône. Un écran noir apparaît : il s'agit de la console.

Les outils setuptools

Depuis cette console, vous allez installer l'ensemble d'outils appelé `setuptools`, qui sert à installer des programmes Python sur votre système.

Rendez-vous à l'adresse <http://pypi.python.org/pypi/setuptools>. Dans la liste de téléchargement, copiez le lien qui correspond à votre installation de Python. À l'heure où nous écrivons ces lignes, il s'agit de la version 2.7.

Collez ce lien dans votre console, précédé de `wget` – cela va télécharger sur votre ordinateur une copie des `setuptools` qui vous seront utiles.

```
$ wget http://pypi.python.org/packages/2.7/s/setuptools/setuptools-0.6c11-py2.7.egg#md5=fe1f997bc722265116870bc7919059ea
```

Ensuite, toujours dans la console, saisissez le nom du fichier téléchargé, ici `setuptools-0.6c11-py2.7.egg`, précédé de `sh` :

```
sh setuptools-0.6c11-py2.7.egg
```

Vous avez désormais à votre disposition une nouvelle commande nommée `easy_install`, qui facilite l'installation des paquets Python (ou *packages*). Ces derniers sont l'équivalent de programmes ou de bibliothèques tierces qui vous seront utiles tout au long de ce livre, et probablement par la suite.

Installer pip et virtualenv

Au cours de ce livre et dans votre parcours de développeur Django, vous allez réaliser régulièrement de nouveaux projets. Seulement, ces derniers n'ont pas vocation à rester sur votre ordinateur personnel mais à être placés sur des ordinateurs distants : les serveurs. Il est probable que votre machine de développement ne soit pas identique à la machine de production, et qu'elles n'auront pas le même système d'exploitation. Il vous faudra donc un système qui permette d'isoler vos projets afin de les dupliquer facilement d'une machine à l'autre. Deux outils vous seront nécessaires.

- `pip`, programme qui permet d'installer facilement des paquets Python, quel que soit votre système. Il fournit la liste des paquets déjà installés, les met à jour et les supprime éventuellement. Il sait également installer automatiquement une liste prédéfinie de paquets.
- `virtualenv`, un gestionnaire et créateur d'environnements virtuels. Il crée des espaces de travail isolés du reste de votre machine. Ainsi, lorsque vous installez un logiciel Python dans un environnement virtuel, il n'est disponible que dans cet environnement et non sur l'ensemble de votre système.

Installation sous Linux

L'installation des deux outils se fait très simplement via votre gestionnaire de paquets. Sur un système à base de paquets `deb` :

```
$ sudo apt-get install python-pip python-virtualenv
```

et sur un système à base de RPM :

```
$ sudo yum install python-pip python-virtualenv
```

Il est très probable que Django soit déjà dans les dépôts de votre distribution. Tapez simplement, sur un système à base de [deb](#) :

```
$ sudo apt-get install python-django
```

ou sur un système à base de RPM :

```
$ sudo yum install python-django
```

Installation sous Mac OS X

Ouvrez simplement un terminal – ce programme est déjà installé dans le répertoire [Application/Accessoires](#) – et tapez :

```
$ sudo easy_install pip
```

puis :

```
$ sudo pip install virtualenv
```

Puisque vous avez utilisé [sudo](#), vous serez invité à entrer votre mot de passe de super-utilisateur. Ainsi, vous pourrez installer ces deux programmes sur l'ensemble de votre système, vous permettant d'en bénéficier à tout moment.

Installation sous Windows

Vous devez commencer par installer [pip](#). Dans votre console Cygwin, tapez :

```
$ easy_install pip
```

Puis [virtualenv](#) s'installe directement avec votre nouveau logiciel :

```
pip install virtualenv
```

Installer un gestionnaire de versions

Maintenant, il vous faut installer un outil de suivi de code : un gestionnaire de versions.

MÉTHODE Pourquoi utiliser un gestionnaire de versions ?

Si vous avez déjà fait du développement informatique, vous connaissez l'intérêt d'un gestionnaire de versions. Si ce n'est pas le cas, nous vous proposons de suivre un petit exemple. Imaginez que vous souhaitiez élaborer une nouvelle recette de cuisine. Vous voulez qu'elle soit parfaite car vous participez au concours de la meilleure recette d'omelette du monde. Vous allez donc devoir la tester et la retester afin de l'améliorer par étapes successives. Pour commencer, vous partez sur une recette assez simple. Vous notez cette recette sur une fiche de cuisine et la testez sur vos convives. Même si cette omelette est réussie, vous trouvez que les lardons manquent de cuisson. Vous allez donc modifier votre mise en œuvre et ajouter les lardons à la préparation, au même moment que les œufs.

Ingrédients : 4 œufs, 150 grammes de lardons, beurre et fromage râpé.

Mise en œuvre : Battre les œufs longuement dans une jatte. Beurrer une poêle à fond plat. Faire chauffer le beurre sans le laisser brunir et ajouter les œufs. Lorsque ces derniers commencent à prendre, verser les lardons. En fin de cuisson, ajouter le fromage râpé. Une fois celui-ci fondu, plier l'omelette et servir avec une salade de mesclun.

Vous faites donc une seconde fiche. Et vous testez de nouveau la recette. Le résultat vous plaît, mais vous aimeriez tester avec des lamelles de pommes de terre. Vous ajoutez donc les pommes de terre à vos ingrédients, modifiez la mise en œuvre et retestez la recette.

Vous allez donc rapidement devoir classer vos fiches. Peut-être faut-il les numéroter ? Mais dans quel ordre ? Chronologique ? Et que faire alors des essais ? Des fiches d'une autre couleur ? Mais comment s'y retrouver quand on réalise plusieurs essais en modifiant la même recette de base ? Il vous faut analyser posément vos besoins.

- Conserver toutes les **versions** de votre recette.
- Garder en mémoire les **différences** entre les recettes.
- Pouvoir **annuler** certains changements apportés.
- Marquer différentes « **branches** » de la même recette de départ.

C'est exactement ce que fait un gestionnaire de versions. Ce logiciel va « suivre » l'évolution de certains fichiers présents sur votre ordinateur, en mémoriser tous les changements pour vous permettre de revenir quand vous le souhaitez à une version antérieure. Vous pourrez également ôter les recettes finalisées, créer de nouvelles versions de la même recette et les fusionner quand les tests sont concluants.

Si cela vous semble un peu perturbant, rassurez-vous ! Dès que vous commencerez à vous en servir, vous ne pourrez plus vous en passer.

Les différents systèmes de gestion de versions

Les gestionnaires de versions existent depuis la nuit des temps informatiques – le premier fut SCCS en 1972. Depuis, leur nombre a augmenté de façon exponentielle. Aujourd'hui, on peut les séparer en deux groupes distincts : les gestionnaires centralisés (CVS ou SVN) et les gestionnaires distribués (Mercurial ou Git).

Les premiers offrent une architecture où le dépôt est unique et installé sur une seule machine. A contrario, les seconds proposent une architecture où chaque participant possède intégralement le dépôt et tout son historique. Dans la pratique, ces différentes architectures n'ont d'importance que lors d'un travail à plusieurs personnes. Dans le cas d'un seul développeur, vous ne verrez pas de différences notables entre ces deux architectures.

Installer un gestionnaire de versions : le choix Mercurial

Tout d'abord, vous devez choisir entre gestionnaire de versions distribué ou centralisé. Comme aujourd'hui la tendance va plutôt vers une architecture décentralisée, nous avons choisi de vous proposer un outil présentant cette architecture. Les deux gestionnaires de versions distribués disponibles actuellement sont Mercurial et Git. Même s'ils sont d'aussi bonne qualité l'un que l'autre, nous avons opté pour Mercurial, que nous trouvons un peu plus simple d'utilisation. Tous les exemples présentés dans ce livre utiliseront donc ce gestionnaire de versions. Sachez que ses commandes sont très proches de celles de Git ; si un jour vous devez changer d'outil, vous n'aurez ainsi pas de difficultés majeures à vous adapter.

POUR EN SAVOIR PLUS **Git**

📖 R. Hertzog, P. Habouzit, *Mémento Git à 100 %*, Eyrolles, 2012.

L'installation de Mercurial dépend bien entendu de votre système d'exploitation.

- Sous Windows, vous l'avez déjà installé lors de la configuration de Cygwin.
- Sous Mac OS X, rendez-vous sur le site de Mercurial (<http://mercurial.selenic.com>). Téléchargez, puis installez le binaire.
- Sous Linux, Mercurial est déjà présent dans vos dépôts. Vous pouvez donc l'installer simplement en suivant la méthode habituelle. Par exemple, sous Debian :

```
apt-get install mercurial
```

Pour vérifier que l'installation s'est correctement déroulée, ouvrez un terminal et tapez :

```
$ hg
```

Vous devriez voir apparaître ce message, ce qui vous indique que l'installation s'est bien déroulée :

```
Mercurial Distributed SCM
```

```
basic commands:
```

```
add          add the specified files on the next commit
annotate    show changeset information by line for each file
clone       make a copy of an existing repository
commit     commit the specified files or all outstanding changes
diff       diff repository (or selected files)
```

```

export      dump the header and diffs for one or more changesets
forget     forget the specified files on the next commit
init       create a new repository in the given directory
log        show revision history of entire repository or files
merge     merge working directory with another revision
pull      pull changes from the specified source
push      push changes to the specified destination
remove    remove the specified files on the next commit
serve     start stand-alone webserver
status    show changed files in the working directory
summary   summarize working directory state
update    update working directory (or switch revisions)

```

use "hg help" for the full list of commands or "hg -v" for details

Installer et tester Django

Vous venez d'installer un ensemble d'outils fort intéressants mais, pour le moment, il n'y a pas de Django à l'horizon. Il est donc temps de tester votre installation.

Commencez par sélectionner un espace de travail dans lequel vous rangerez vos différents projets. C'est à vous de décider ce qui vous convient le mieux : sur votre Bureau, dans un répertoire personnel... Pour nos exemples, nous avons choisi de créer notre dossier de travail à la racine de notre répertoire personnel et de le nommer `DEV`.

Via le terminal, rendez-vous donc dans votre dossier de développement. Créez-y votre premier environnement virtuel :

```

$ virtualenv environnement_de_test
New python executable in environnement_de_test/bin/python
Installing setuptools.....done.
Installing pip.....done.

```

Activez ensuite votre nouvel environnement virtuel :

```

$ source environnement_de_test/bin/activate
(environnement_de_test)$

```

Entre parenthèses, avant votre invite de commande, vous voyez le nom de l'environnement virtuel actuellement actif ; cela vous évitera de vous tromper d'environnement au cours de votre travail.

Quand vous souhaitez quitter cet environnement virtuel ou en changer, tapez simplement :

```

(environnement_de_test)$ deactivate
$

```

Ne le faites pas pour le moment car vous allez enfin installer Django avec la commande suivante :

```
(environnement_de_test)$ pip install django
Downloading/unpacking django
  Downloading Django-1.4.3.tar.gz (7.7Mb): 7.7Mb downloaded
  Running setup.py egg_info for package django

Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.7/django-admin.py from 644 to 755

    changing mode of /Users/user/Dev/environnement_de_test/bin/django-admin.py
to 755
Successfully installed django
Cleaning up...
(environnement_de_test)$
```

Django est maintenant installé dans votre environnement virtuel. Pour vérifier votre installation, tapez :

```
$ django-admin.py --version
1.4.3
```

`django-admin.py` est un exécutable fourni avec Django qui va vous servir à créer de nouveaux projets.

Initialiser un projet

Maintenant que vous possédez l'outillage du parfait développeur Django, vous pouvez créer votre premier projet.

Un projet, des applications

Django est construit sur une approche très modulaire ; il implémente avec efficacité les notions de découplage et de réutilisabilité. Pour y parvenir, il découpe chaque projet en un ensemble de plus petits paquets nommés « apps ».

Lorsque vous souhaitez créer une application web, vous réalisez donc un projet dans lequel résideront différentes applications. Ces dernières pourront aisément être déplacées d'un projet à l'autre, vous permettant, dans l'organisation même de vos répertoires, d'être modulaire et DRY (*Don't Repeat Yourself* ou « on ne duplique pas de code »).

Django fournit un ensemble de commandes qui faciliteront la gestion de votre projet tout au long de sa vie. Nous verrons même, dans les chapitres suivants, comment créer soi-même des commandes de ce type. Pour le moment, voyons d'abord celles qui vous seront utiles immédiatement.

Les commandes d'initialisation d'un projet

Pour ce premier exemple, utilisez l'environnement virtuel que vous venez de réaliser. Pour mémoire :

```
$ source environnement_de_test/bin/activate
(envnement_de_test)$
```

Avant de tester les différents extraits de code, vérifiez que vous êtes bien dans le bon environnement de test.

Création d'un projet : startproject

La première commande que vous allez utiliser est :

```
[user@local]$ django-admin startproject premiertest
```

Cette fonction réalise l'arborescence nécessaire à votre projet. Vous pouvez constater qu'un dossier `premierest` vient d'être créé.

Serveur web de développement : runserver

Pour le moment, nous pouvons d'ores et déjà vérifier que tout est fonctionnel, en lançant le serveur de développement.

Rendez-vous tout d'abord dans le répertoire de votre projet :

```
[user@local]$ cd premiertest
[user@local]$ python manage.py runserver
Validating models...

0 errors found
Django version 1.4.3, using settings 'premierest.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Le retour du terminal vous invite à vous rendre à l'adresse <http://127.0.0.1:8000>, qui correspond à votre propre machine sur le port 8000. En effet, Django vient de lancer pour vous un serveur web rudimentaire, particulièrement bien adapté pour le déve-

lancement, grâce auquel vous testerez facilement votre application sur votre machine. Entre autres fonctionnalités, ce serveur de développement « ausculte » en permanence votre projet et, dès qu'un fichier est modifié, il se recharge automatiquement. Ainsi, vous pourrez apprécier immédiatement un changement dans votre code, simplement en actualisant votre navigateur.

ATTENTION Pas d'utilisation en production

Ce serveur web ne devra jamais être employé en production : il n'est pas conçu pour accepter plus d'une connexion à la fois, prohibant son utilisation dans quelques autres cas que celui des tests ou du développement.

Sachez également que vous pouvez vous rendre à l'adresse <http://localhost:8000>, qui est un alias de <http://127.0.0.1:8000>. Si, pour une raison ou pour une autre, le port 8000 n'est pas disponible, voici le message d'erreur qui doit s'afficher :

```
Validating models...
0 errors found
Django version 1.3, using settings 'premiertest.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
Error: That port is already in use.
```

Il vous suffit alors de lancer la commande, suivie du numéro de port que vous souhaitez utiliser, par exemple 5000 :

```
$ python manage.py runserver 5000
```

Il existe d'autres options disponibles pour le serveur de développement de Django. Pour en avoir un aperçu, tapez :

```
python manage.py help runserver
```

Une fois votre navigateur web favori ouvert sur la page <http://localhost:8000>, vous verrez s'afficher l'écran suivant. Félicitations, l'installation de Django s'est parfaitement déroulée !



Figure 1-1 Écran d'accueil de Django

Cette page vous invite à indiquer les paramètres de votre base de données dans le fichier `settings.py` et à créer votre première application.

Nous aimerions attirer votre attention sur un point intéressant. Revenez sur le terminal d'où vous avez lancé la commande `python manage.py runserver`. Vous devriez voir s'afficher une ligne de ce type :

```
[27/Ju]/2011 12:44:10] "GET / HTTP/1.1" 200 2053
```

Il s'agit tout simplement de la requête que Django vient de vous servir, décomposée comme suit :

- la date et l'heure ;
- le verbe utilisé ;
- l'adresse demandée (ici `/`, la racine de votre site) ;
- le protocole utilisé ;
- la taille (en octets) de la réponse.

Cette fonctionnalité va vous être très précieuse dès que vous souhaiterez déboguer votre application. C'est un élément qui vous sera utile tout au long de ce livre, et même après !

Conclusion

Vous avez donc maintenant une installation fonctionnelle, qui est reproductible très simplement. Dès que vous souhaiterez réaliser un nouveau projet, vous n'aurez qu'à suivre ces quelques étapes :

- 1 Créer un nouvel environnement virtuel et l'activer :

```
$ virtualenv votre_nouvel_environnement  
$ source votre_nouvel_environnement/bin/activate
```

2 Installer Django :

```
(votre_nouvel_environnement)$ pip install django
```

3 Initialiser un nouveau projet :

```
(votre_nouvel_environnement)$ django-admin nouveau-projet
```

Le projet est la pièce indispensable au bon fonctionnement d'une application web réalisée à l'aide du framework Django. Les différentes fonctionnalités résident pourtant dans des composants différents : les applications. Dans le prochain chapitre, en même temps que vous créerez votre premier outil, vous apprendrez à configurer un projet et à lui ajouter de nouvelles fonctionnalités à l'aide de ces nouveaux composants.

2

Créer sa première application : un tracker

Dans ce chapitre, vous allez créer votre première application. Ce faisant, vous découvrirez progressivement certaines des fonctionnalités majeures de Django. Loin des « wiki en vingt minutes » et autres « mon premier blog en Django », ce premier projet vous accompagnera tout au long du livre et, nous l'espérons, bien au-delà. Vous allez créer un tracker, outil qui tracera l'avancement de votre projet en vous donnant une vision claire de votre travail et du temps que vous y consacrez.

De la méthode et un framework adaptable

Nous n'allons pas créer d'emblée toutes les fonctionnalités de notre projet, de la base de données vers l'interface graphique, comme cela se faisait il y a encore quelques années avec le mode de développement par cahier des charges. Cette manière de procéder est à éviter à bien des égards. Entre autres, vous risquez les deux écueils suivants.

- **Oublier une fonctionnalité majeure.** Au beau milieu de votre projet, vous vous apercevez que vous avez oublié un élément primordial. Seulement, vous avez déjà créé la base de données et les fonctions principales du cœur de votre application ; modifier toutes ces fonctions est un tel travail que vous n'avez plus qu'à tout reprendre depuis le début ou, tout simplement, abandonner le projet.

- **Coder une fonctionnalité inutile.** Bien que ce cas paraisse moins grave que le précédent, il se produit plus souvent que l'on ne le pense. En imaginant l'application, vous croyez que telle ou telle fonctionnalité sera indispensable ou très utile. Vous codez donc votre application autour de cette pièce maîtresse. En réalisant les premiers tests fonctionnels, vous vous apercevez que cette fonction n'est finalement pas utile, voire qu'elle gêne le reste du projet. C'est bien souvent dans les dernières étapes, lorsque l'on dessine l'interface web de l'application, que les problématiques de ce genre se font jour. Il faut alors repenser entièrement le programme, ce qui peut bien souvent mener à l'abandon du projet

Une autre raison de rejeter le développement par « cahier des charges » est sans doute d'ordre plus personnel. En effet, lorsque l'on est seul sur son projet, avec pour seul appui sa motivation, devoir s'atteler à une tâche parfois de plusieurs mois avant de récolter les fruits de son travail est très vite décourageant.

Pour éviter ces écueils (et bien d'autres), nous vous proposons de découvrir dès votre premier projet une autre méthode : **le développement par itérations.**

Le développement par itérations

Le développement par itérations consiste à créer un programme par petites parties indépendantes les unes des autres. Si, par exemple, vous devez gérer une liste d'utilisateurs, vous commencerez par créer un objet utilisateur, puis l'interface nécessaire pour le manipuler. Démuni de fonctionnalités, cet embryon d'application n'en sera pas moins pleinement fonctionnel ; vous pourrez le manipuler, « jouer avec », le tester dans différentes situations et détecter très rapidement les éventuelles erreurs de conception. À ce niveau de la vie de l'application, il n'est évidemment pas question de régler tous les détails graphiques, tous les cas particuliers. Il s'agit seulement d'en dessiner les grandes lignes et de pouvoir manipuler l'objet pour découvrir très tôt ses failles et ses qualités. Une fois que ce bloc de fonctionnalités vous semblera satisfaisant, vous ajouterez alors de nouvelles briques à l'ensemble en procédant de la même manière. Ainsi, brique après brique, vous verrez votre application grandir, mûrir, évoluer avec, à chaque fois, la possibilité de modifier, transformer et adapter votre application à de nouveaux cas.

L'un des avantages principaux de développer une application web à l'aide d'un framework, et c'est particulièrement vrai avec Django, est d'obtenir rapidement des résultats tangibles, ce qui est évidemment bienvenu dans le cadre d'un développement par itérations. Dès cette première application, vous allez vous rendre compte à quel point Django vous offre la possibilité de modéliser rapidement les premières briques en quelques commandes seulement.

Un framework transparent et adaptable

Très souvent, cette rapidité de développement souffre d'un revers de taille. En effet, lorsque vous souhaitez sortir de ce qui est prévu par le framework, les choses commencent à se gâter. Mais, après plusieurs années de développement avec Django, nous vous assurons qu'il n'en est rien.

Cela tient en définitive à deux choses.

- Tout d'abord, Django repose sur le langage de programmation Python. La nature interprétée et dynamique de ce dernier rend son utilisation très souple et polymorphique : il est par conséquent très simple de modifier la façon dont fonctionne ici une fonction, là une classe.
- La seconde raison tient à une volonté précise des créateurs de Django qui, le 1^{er} mai 2006, ont pris la décision de supprimer la « magie » de ce framework. Cela a principalement consisté à enlever tous les éléments qui cachaient aux développeurs le fonctionnement interne de Django. Désormais, ils peuvent modifier entièrement son comportement pour un besoin précis et l'adapter à volonté. À la fin de ce livre, dans la partie consacrée aux techniques avancées avec Django, nous aborderons plusieurs exemples de modifications de ce type.

Nous allons vous guider pas à pas dans le fonctionnement interne de Django. L'idée n'est pas de vous assommer dès le début avec des concepts complexes, mais de reprendre son principe de fonctionnement grâce à un exercice didactique et d'en retirer toute la magie (ou au moins une partie).

De l'utilité d'un tracker

Comme premier exercice, nous vous proposons de créer votre premier outil : un *tracker*.

Organisation du travail

Dans sa forme la plus basique, un tracker est une sorte de liste de choses à faire qui aide à maintenir un cap dans l'évolution d'un projet. Élément indispensable du travail en équipe, il en assure la cohésion et indique à tous qui fait quoi et sur quelle partie du code.

Même lors d'un travail solitaire, un tracker vous rendra de fiers services. Lorsque vous reviendrez à votre projet après quelques jours d'inactivité, vous saurez, grâce à lui, quelles tâches il vous reste encore à accomplir. Vous en garderez ainsi une vision claire et maintiendrez son évolution avec sérénité.

Gestion du temps

Un tracker vous aidera également à gérer votre charge de travail en fonction de vos disponibilités, à planifier les futures évolutions de votre projet et à estimer le temps passé sur telle ou telle fonctionnalité. Cette capacité à gérer votre projet dans le temps garantit une évolution stable et vous fournit les atouts indispensables à sa réussite.

Gestion des jalons

Un autre point fort d'un tracker réside aussi dans l'utilisation intelligente de jalons qui marqueront les différentes étapes de votre projet. Un jalon se définit comme une liste des choses qu'il reste à accomplir avant qu'une partie ou une fonctionnalité du programme soit opérationnelle. C'est particulièrement utile dans un mode de développement par itérations, où chaque jalon correspond à une itération précise. Un tracker vous permettra donc de mesurer et de suivre l'évolution de votre projet, jalon après jalon.

Historique

Enfin, vous serez en mesure de garder une trace de votre travail et du temps nécessaire à son accomplissement. Comme vous enregistrez le temps passé sur chacune de vos tâches, et donc le temps global passé sur un jalon, vous pourrez y revenir pour estimer la somme des efforts qu'il vous aura fallu fournir pour mener à bien votre tâche.

Avec le temps, vous estimerez de plus en plus finement le temps nécessaire à la réalisation d'une tâche. Vous pourrez la planifier, l'organiser et donc rester maître du temps, notion essentielle en programmation (mais pas seulement !).

Tracker et gestionnaire de versions : les outils d'une méthode « agile »

Comme nous l'avons vu dans le chapitre précédent, le gestionnaire de versions est lui aussi indispensable à la bonne conduite d'un projet. Nous ne pouvons que vous en conseiller très fortement l'usage, quelle que soit la taille de votre projet, même s'il ne s'agit que d'un petit script dans un coin de votre disque dur. Un gestionnaire de versions est très simple à mettre en place, et tout aussi simple à utiliser. En revanche, il vous rendra des services immenses lors de la moindre évolution de votre projet.

Dans les chapitres suivants, nous verrons donc comment intégrer un gestionnaire de versions à votre tracker. Vous aurez ainsi, dès votre entrée dans l'univers de Django, un ensemble d'outils efficaces et performants qui garantiront la réussite de vos futurs projets. En somme, vous avez en main les outils indispensables à l'implémentation d'une méthode « agile » et vous allez dès maintenant expérimenter cette méthode : travail itératif, gestion du temps et gestionnaire de versions.

GESTION DE PROJET Les méthodes agiles

Les méthodes agiles sont un ensemble de techniques qui visent à être plus pragmatiques que les anciennes. Au cours du développement, elles impliquent beaucoup le demandeur (souvent le client) et se caractérisent par des cycles de livraison courts.

📖 V. Messenger, *Gestion de projet agile*, Eyrolles, 2013.

Initialisation du projet

Il est désormais temps de réutiliser ce que vous connaissez déjà pour mettre en place votre environnement de travail.

Mise en place

Dans le chapitre précédent, vous avez appris à réaliser un environnement de travail, et à y installer Django et un nouveau projet. Lancez donc les commandes que vous connaissez déjà afin de créer l'environnement adéquat pour votre tracker :

```
$ virtualenv tracker_env
(tracker_env)$ source tracker_env/bin/activate
(tracker_env)$ pip install django
(tracker_env)$ django-admin.py startproject tracker
```

Création d'une application : startapp

Maintenant, créez une nouvelle application au sein de votre projet :

```
(tracker_env)$ cd tracker
(tracker_env)$ python manage.py startapp ticket
```

Comme vous le voyez, un nouveau dossier vient d'être créé dans le répertoire de votre projet, portant le nom `ticket`.

Avant d'aller plus loin, observons comment Django a organisé l'espace de travail.

Vue d'ensemble du projet

Dans le dossier `tracker`, vous trouverez :

- `tracker/__init__.py`. Si vous avez déjà des notions de Python, vous savez que ce fichier permet à ce dossier d'être traité comme un paquet Python ;

- `manage.py`. C'est le point d'entrée vers toutes les commandes `manage` que vous utiliserez tout au long du développement de votre application ;
- `tracker/urls.py`. C'est la table de routage de votre application. Dans ce fichier, vous allez « brancher » les différentes applications de votre projet à vos URL ;
- `ticket`. Vous retrouvez bien entendu le dossier de l'application que vous venez de créer – nous verrons bientôt comment ce fichier s'organise ;
- `tracker/settings.py`. Il s'agit d'un fichier dans lequel vous retrouvez tous les paramètres nécessaires à votre application. Pour le moment, ce fichier contient déjà un ensemble de variables d'environnement qui rendent votre installation fonctionnelle, mais vous allez dès maintenant modifier ce fichier selon vos besoins spécifiques.

MÉTHODE La démarche Django

Avec Django, quelques lignes de code suffisent pour mettre en place très rapidement une application, de façon claire et découpée :

- ajouter votre application aux `settings` ;
- modifier le fichier `urls.py` à la racine de votre projet pour le faire pointer vers votre application ;
- créer un fichier `urls.py` au sein de votre application ;
- indiquer une première URL ;
- créer la vue correspondante.

Paramétrer les variables d'environnement (settings)

Les `settings` sont les variables d'environnement de votre projet, qui définissent les informations de votre base de données, les paramètres de langue, les applications que votre projet utilisera, l'emplacement de vos fichiers statiques et bien d'autres choses que nous verrons en détail.

Mode de développement (DEBUG)

`/tracker/settings.py`

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

La variable `DEBUG` lance Django en mode de développement. De nombreux outils y sont mis à votre disposition pour faciliter le développement. Par exemple, pour chaque erreur que vous rencontrerez, vous aurez accès au `backtrace` complet, qui vous aidera à trouver l'origine du problème. Il est bien entendu fortement conseillé de laisser le serveur en mode `DEBUG` pendant tout la phase de développement de votre application.

Backtrace

Lorsqu'un programme Python rencontre une erreur, il suit un cheminement au travers des différentes fonctions qui ont exécuté le code erroné. Il affiche l'ensemble de ce cheminement, ce qui vous permet de retracer à l'envers ce qui a conduit à l'erreur.

`TEMPLATE_DEBUG` ne peut être utile que dans le cas où la variable `DEBUG` est positionnée à `True`. En effet, cette dernière affichera une page détaillée pour chaque erreur, directement dans le navigateur (voir figure 2-1).

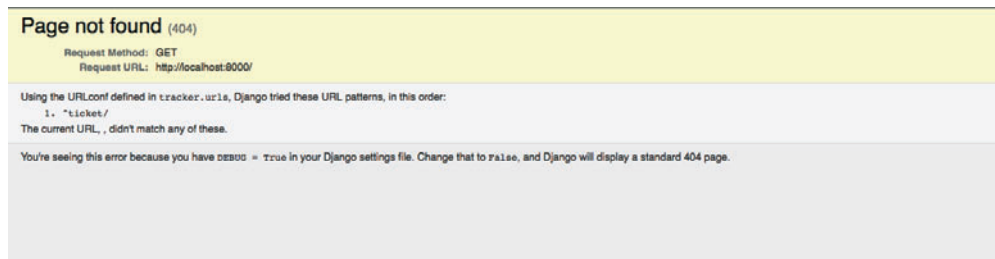


Figure 2-1 Une page d'erreur

Configuration de la base de données

Il est temps de configurer la base de données de votre projet. Il suffit pour cela d'éditer la variable `DATABASE`. Voici comment se présente cette partie des variables (settings) avant modification.

Paramètre de base de données avant modification

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.',
        # Add 'postgresql_psycopg2', 'postgresql', 'mysql', 'sqlite3' or 'oracle'.
        'NAME': '',
        # Or path to database file if using sqlite3.
        'USER': '',
        # Not used with sqlite3.
        'PASSWORD': '',
        # Not used with sqlite3.
        'HOST': '',
        # Set to empty string for localhost. Not used with sqlite3.
        'PORT': '',
        # Set to empty string for default. Not used with sqlite3.
    }
}
```

Puisque vous serez le seul à utiliser ce tracker, au moins dans un premier temps, nous opterons pour une base de données SQLite. Django étant très tolérant en termes de base de données, je vous conseille, lors des premiers pas de votre application, d'utiliser SQLite, un SGBD performant, très simple à mettre en place et qui permet, entre autres avantages, de maintenir votre concentration sur votre code et non sur la gestion de votre base. Cette base de données sous forme de fichier se configure très simplement.

Configuration de la base de données

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        # Indique quel type de base de données utiliser.
        'NAME': 'tracker.db',
        # Le fichier de base de données sera créé dans le dossier tracker.
    }
}
```

Pour communiquer avec la base de données, Django a besoin que le connecteur Python vers ce type de base soit installé sur votre machine. Il se présente sous la forme d'un module qu'il faut installer spécifiquement pour chaque type de base de données. En ce qui concerne SQLite3, le connecteur est livré en standard avec Python. Si ce dernier est installé, vous avez déjà tout ce qu'il faut pour continuer.

Internationalisation

Django propose de puissants outils d'internationalisation. Il est donc très simple de le configurer pour le faire fonctionner en français.

Tout d'abord, configurez le fuseau horaire, en utilisant les codes standards de Django dont vous trouverez une description complète, par exemple, dans la documentation en ligne de PostgreSQL.

► <http://www.postgresql.org/docs/8.1/static/datetime-keywords.html#DATETIME-TIMEZONE-SET-TABLE>

Remplacez donc :

```
TIME_ZONE = 'America/Chicago'
```

par

```
TIME_ZONE = 'Europe/Paris'
```

RESSOURCES Paramètres de langue

Pour modifier les paramètres de langue, Django s'appuie sur les codes de la norme ISO 639-1, dont vous pourrez trouver la liste ici :

▸ http://fr.wikipedia.org/wiki/Liste_des_codes_ISO_639-1

Dans le cas présent, vous pouvez modifier le paramètre :

```
LANGUAGE_CODE = 'en-us'
```

en

```
LANGUAGE_CODE = 'fr-fr'
```

Applications installées par défaut

Regardez le paramètre `INSTALLED_APPS` :

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

Django est livré par défaut avec quelques applications. La première, `django.contrib.auth`, va gérer toute la partie authentification de votre application. Nous aborderons plus tard les autres applications.

Il y a bien d'autres choses que vous pourrez faire avec les settings mais, pour le moment, les changements que nous venons d'appliquer sont suffisants pour bien démarrer.

Initialisation de la base de données : syncdb

Maintenant que vous avez indiqué à Django quelle base de données vous souhaitez utiliser, il faut l'initialiser avec la commande suivante :

```
[user@local]$ python manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table auth_message
Creating table django_content_type
Creating table django_session
Creating table django_site
```

```
You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no):
```

Plusieurs tables sont créées par cette commande, toutes nécessaires au fonctionnement des applications qui ont été définies par Django dans la section [INSTALLED_APPS](#). À la question que Django vous pose, répondez **yes** pour qu'il crée un super-utilisateur qui vous sera utile dans un moment. Répondez ensuite à toutes les questions (nom d'utilisateur, adresse électronique, mot de passe) :

```
Would you like to create one now? (yes/no): yes
Username (Leave blank to use 'user'):
E-mail address: user@example.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
No fixtures found.
```

Si vous observez le contenu de votre répertoire `tracker`, vous vous apercevrez que Django a créé pour vous le fichier `tracker.db` qui accueille votre base de données.

Projet versus application

Comme nous l'expliquions plus haut, Django découpe votre projet en plusieurs applications. Avec la commande `python manage.py startapp ticket`, vous avez déjà créé la première. Observons maintenant ce que Django a créé pour vous :

```
[user@local]$ ls ticket
__init__.py models.py tests.py views.py
```

Voyons à quoi ces éléments correspondent.

- `__init__.py` : comme son homologue présent dans le dossier `tracker`, ce fichier permet à Python de considérer ce dossier comme un paquet.
- `models.py` : ce fichier fait le lien entre votre application et la base de données. Il correspond tout simplement au « M » du sigle MVC (*Model-View-Controller*, ou modèle-vue-contrôleur). Nous en reparlerons en détail au chapitre 10 « Django et les bases de données ».
- `test.py` : c'est un fichier que vous utiliserez pour lancer vos batteries de tests, dans le cas très conseillé où vous mettiez en pratique le principe de TDD (*Test Driven Development*, ou développement dirigé par les tests).
- `views.py` : il s'agit du cœur de votre application. Les vues (*views*) vont accueillir l'essentiel de votre code. Nous en reparlerons longuement au chapitre 11 « Le traitement de l'information : les vues ».

Configuration des applications installées

Il faut indiquer à Django que vous souhaitez utiliser cette application pour votre projet `tracker`. Ouvrez tout simplement le fichier `settings.py` et ajoutez votre application à la liste des `INSTALLED_APPS` :

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

Les URL

Il faut ensuite indiquer à Django, dans le fichier `urls.py`, l'URL racine qui pointera vers votre application.

Fichier `urls.py`

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

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

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

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

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

Décommentez la ligne :

```
# url(r'^tracker/', include('tracker.foo.urls')),
```

et modifiez-la comme suit :

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

Elle va simplement indiquer à Django que toutes les URL commençant par `/ticket/` doivent pointer vers les URL définies dans le module `urls`, fichier `urls.py` du paquet `ticket`. Cette façon de procéder est très pratique car elle permet de bénéficier de l'héritage dans les URL. Elle vous aide à découper ces dernières de façon claire et modulaire, puisque chaque application viendra compléter dans son propre fichier `urls.py` le point d'entrée défini dans le fichier `urls.py` à la racine du projet.

MÉTHODE Développement dirigé par les erreurs

Les erreurs font partie intégrante du développement d'une application web. L'une des clés vers la maîtrise du framework Django est de savoir reconnaître celles que vous allez rencontrer et comment les corriger. C'est pourquoi, dans les exemples qui suivent, vous allez être confronté à quelques cas d'erreurs et apprendrez à les corriger.

Pour le moment, le fichier `ticket/urls.py` n'existe pas. Il faut donc le créer.

Création du fichier `ticket/urls.py`

```
from django.conf.urls.defaults import patterns, url
from views import home

urlpatterns = patterns('',
    url(r'^home/$', home, name="home")
)
```

Tout d'abord, importez les fonctions dont vous avez besoin depuis Django (ici, `patterns` et `url`). Ensuite, importez la fonction `home` du fichier `views` (cette fonction n'existe pas encore, mais vous allez la créer dans un instant). Enfin, créez un nouveau `urlpatterns` qui ne va contenir qu'une URL :

```
url(r'^home/$',home, name = "home")
```

Cette ligne indique à Django que l'URL correspondant à `/ticket/home/` doit pointer vers la fonction `home` que vous venez d'importer du fichier `views`.

Testez ces quelques modifications. Lancez votre serveur de développement :

```
python manage.py runserver
```

et rendez-vous sur la page <http://localhost:8000> (voir figure 2-2).

```
ImportError at /
cannot import name home

Request Method: GET
Request URL: http://localhost:8000/
Django Version: 1.4.3
Exception Type: ImportError
Exception Value: cannot import name home
Exception Location: /Users/yohann/Dev/LIVRE/tracker/ticket/urls.py in <module>, line 2
Python Executable: /Users/yohann/Dev/LIVRE/tracker/env/bin/python
Python Version: 2.7.1
Python Path:
['/Users/yohann/Dev/LIVRE/tracker',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages/setuptools-0.6c11-py2.7.egg',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages/pip-1.0.2-py2.7.egg',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/plat-darwin',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/plat-mac',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-dk',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-old',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/lib-dynload',
'/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7',
'/system/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-darwin',
'/system/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-ik',
'/system/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac',
'/system/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/plat-mac/lib-scriptpackages',
'/Users/yohann/Dev/LIVRE/tracker_env/lib/python2.7/site-packages']

Server time: mer, 19 Déc 2012 10:53:23 +0100
```

Figure 2-2 Message d'erreur

Une première vue

Django signale l'absence de la fonction `home`. Vous devez donc la créer dans le fichier `ticket/views.py`.

Fichier `ticket/views.py`

```
def home(request):  
    print "hello world"
```

Voici une fonction du fichier `views.py` dans sa forme la plus simple (dans le reste du livre, nous utiliserons le terme « vue » pour parler des fonctions du fichier `views.py`). Elle prend en paramètre l'objet `request` créé par Django, dans lequel vous trouverez de nombreuses informations – nous y reviendrons plus loin.

Testez de nouveau votre projet en vous rendant sur la page <http://localhost:8000>.

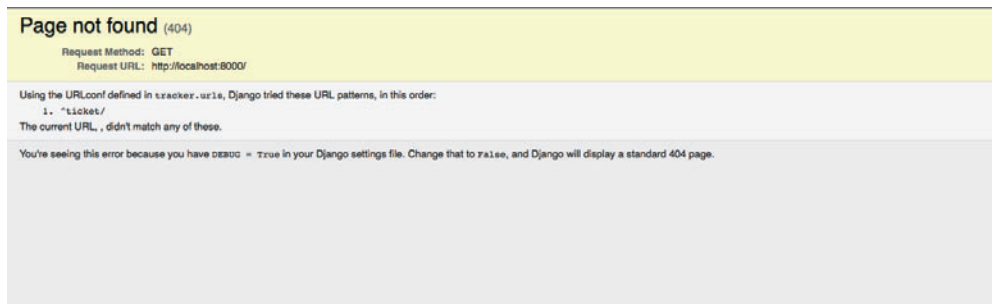


Figure 2-3 La page n'est pas trouvée.

Comme vous le remarquez, vous n'obtenez plus l'écran de la figure 2-2, mais un message qui vous indique que la page recherchée n'existe pas. Un coup d'œil dans votre terminal vous confirmera d'ailleurs cette erreur.

```
[28/Jul/2011 23:32:03] "GET / HTTP/1.1" 404 1983
```

Le code `404` est renvoyé lorsque le serveur ne trouve pas la page demandée.

Les codes HTTP

Le protocole HTTP, entre autres choses, définit un ensemble de *status codes*. Il s'agit de codes renseignant sur le résultat global d'une requête. Les plus courants sont `200` (succès), `404` (ressource non trouvée) et `500` (erreur interne du serveur).

Fort heureusement, sur la page d'erreur, Django indique qu'il a dans ses URL un pattern commençant par `^ticket/`. Essayez de charger la page `http://localhost:8000/ticket/` (voir figure 2-4).

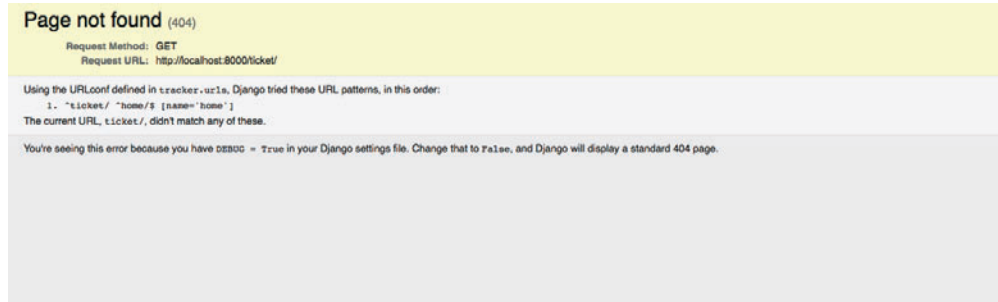


Figure 2-4 Ce n'est toujours pas la bonne URL.

Encore une fois, Django vous donne de précieuses informations. S'il ne trouve toujours pas la page que vous lui demandez, il vous indique en revanche qu'il existe un chemin `http://localhost:8000/ticket/home/`. Rendez-vous donc à cette adresse (voir figure 2-5).

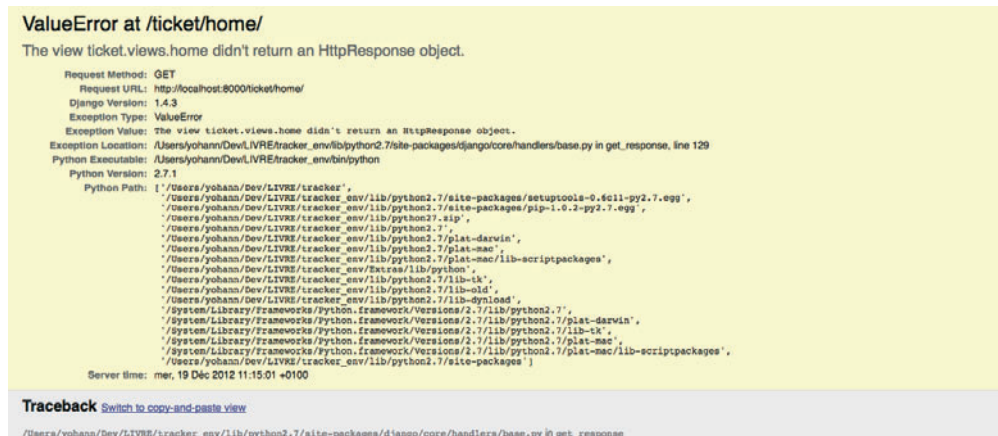


Figure 2-5 Il manque un objet de type `HttpResponse`.

Encore une erreur ! Cette fois, Django précise que la vue `tracker.ticket.views.home` n'a pas renvoyé d'objet de type `HttpResponse`. Nous verrons plus loin comment créer ce type d'objet mais, pour le moment, tournez-vous vers votre terminal :

```
Hello World!  
[29/Jul/2011 00:05:15] "GET /ticket/home/ HTTP/1.1" 500 55420
```

Vous constatez que votre fonction a eu, en réalité, le résultat escompté : bien que Django ne puisse pas afficher une page dans le navigateur en raison de l'objet `HttpResponse` manquant, il a tout de même affiché le message `Hello World!` dans le terminal. Cette fonctionnalité vous permettra tout au long du développement de votre application de tester, de déboguer et de profiter des capacités d'introspection qu'offrent Python et Django.

L'objet request

Revenons à l'objet `request` que vous avez découvert dans la fonction `home`. Pour en savoir un peu plus sur lui, modifiez cette dernière.

```
# Create your views here  
def home(request):  
    print request
```

Rechargez la page <http://localhost:8000/ticket/home/>. La page renvoyée par Django affiche encore le même message d'erreur, ce qui est parfaitement normal puisque vous ne lui fournissez toujours pas d'objet `HttpResponse`. Cependant, le terminal vous indique de nouveaux éléments :

```
<WSGIRequest  
GET:<QueryDict: {}>,  
POST:<QueryDict: {}>,  
COOKIES:{},  
...<truncated>  
'HTTP_ACCEPT': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;  
q=0.8',  
'HTTP_ACCEPT_CHARSET': 'ISO-8859-1,utf-8;q=0.7,*;q=0.7',  
'HTTP_ACCEPT_ENCODING': 'gzip, deflate',  
'HTTP_ACCEPT_LANGUAGE': 'fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3',  
'HTTP_CONNECTION': 'keep-alive',  
'HTTP_HOST': 'localhost:8000',  
'HTTP_USER_AGENT': 'Mozilla/5.0 (X11; Linux i686; rv:5.0) Gecko/20100101  
Firefox/5.0',  
...<truncated>
```

Vous avez dans l'objet `request` toute la requête qui a été transmise au serveur : les paramètres `GET` et `POST`, les en-têtes HTTP, etc. C'est grâce à cet objet que vous serez en mesure de dialoguer avec l'utilisateur tout au long de la vie de l'application.

L'objet `HttpResponse`


Comme toute requête appelle une réponse, voyons maintenant comment renvoyer au navigateur l'objet `HttpResponse` réclamé par Django. Ouvrez votre fichier `views.py` et modifiez-le comme suit :

```
from django.http import HttpResponse
def home(request):
    return HttpResponse("Hello world!")
```

La seule différence par rapport à l'exemple précédent est d'avoir importé la classe `HttpResponse` et de l'avoir instanciée avec un message.

Rendez-vous encore une fois à l'adresse <http://localhost:8000/ticket/home/> (voir figure 2-6). Comme vous le constatez, votre page affiche désormais votre message.

Figure 2-6
Une première page fonctionnelle



Hello world!

RESSOURCE Sur le gestionnaire de versions

Pour vous simplifier la vie et vous aider à détecter vos erreurs, dans le cas où vous en rencontreriez encore à cette étape de la vie de votre application, vous pouvez récupérer une copie fonctionnelle de ce code sur Bitbucket (<https://www.bitbucket.org>) en utilisant la commande `hg clone https://bitbucket.org/boblefrag/livre-django-tracker`. Nous indiquerons à chaque étape la révision que vous devrez utiliser pour que les exemples de code correspondent au livre.

RAPPEL La démarche Django

Récapitulons la démarche à adopter avec Django pour mettre en place très rapidement une application, de façon claire et découplée : ajouter l'application aux settings, modifier le fichier `urls.py` à la racine du projet pour le faire pointer vers votre application, créer un fichier `urls.py` au sein de l'application, indiquer une première URL et créer la vue correspondante.

Créer une application dynamique

L'exemple que vous venez de créer est pleinement fonctionnel, mais il lui manque ce qui fait qu'une application web est plus qu'un simple site statique. Heureusement, tout est en place pour qu'avec quelques modifications, votre application statique devienne dynamique.

Imaginons cette fois qu'au lieu d'`Hello World!` nous souhaitons afficher le nom d'une personne, par exemple `Hello Pierre!`.

Commencez par modifier le fichier `urls.py` de votre application `ticket` :

```
from django.conf.urls.defaults import patterns, url
from views import home
urlpatterns = patterns('',
    url(r'^home/(\w+)$', home, name="home")
)
```

Vous venez d'ajouter `(\w+)`, qui est une expression régulière indiquant à Django de capturer toutes les lettres après le `/` et de transmettre à la vue la chaîne de caractères ainsi constituée.

Pour l'heure, intéressez-vous au pattern que vous venez de définir : `w+`.

- `w` indique un caractère alphabétique. C'est une écriture raccourcie de `[Aa-Zz]`.
- `+` signifie aucune, une fois ou plusieurs fois le type de caractère précédent.

`w+` correspondra donc à `a`, `aa`, `ba`, mais pas à `1`, `a1`, `1e`, `aertry-jhz`, etc.

Dans notre exemple, les URL qui seront routées vers notre vue pourront être :

- `http://localhost:8000/ticket/home/pierre`
- `http://localhost:8000/ticket/home/Untest`

mais pas :

- `http://localhost:8000/ticket/home/Un-test`
- `http://localhost:8000/ticket/home/un_test`

Testez maintenant votre application en saisissant l'une des deux premières URL. Vous devriez rencontrer l'erreur ci-contre (voir figure 2-7).

Django a bien fait passer l'argument que vous avez défini dans votre URL vers la vue `home`. Seulement, celle-ci prend `request` comme seul argument. Or, avec la modification que vous venez d'effectuer dans vos URL, vous lui avez fourni un nouvel argument dont elle ne sait que faire.

Et voilà, votre application est désormais pleinement dynamique. Seulement, l'adresse <http://localhost:8000/home/> ne fonctionne plus.

La première chose qui pourrait vous venir à l'idée serait de créer deux URL différentes et deux vues pour chacune des URL :

```
from django.conf.urls.defaults import patterns, url
from views import home, home_name

urlpatterns = patterns('',
    url(r'^home/$', home, name="home"),
    url(r'^home/(\w+)/$', home_name, name="home_name")
)
```

et dans le fichier `views.py` :

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello World!")

def home_name(request, name):
    return HttpResponse("Hello %s " % name)
```

Voyons si nous ne pouvons pas factoriser un peu plus :

```
from django.conf.urls.defaults import patterns, url
from views import home

urlpatterns = patterns('',
    url(r'^home/$', home, name="home"),
    url(r'^home/(\w+)$', home, name="home")
)
```

Vous avez ainsi deux URL pointant vers la même fonction `home`.

Spécifiez dans votre vue un paramètre optionnel :

```
from django.http import HttpResponse

def home(request, name=None):
    # Vérifions l'existence de l'attribut name.
    if name:
        return HttpResponse("Hello %s " % name)
    else:
        return HttpResponse("Hello World!")
```

De cette façon, suivant l'URL que vous indiquerez, le serveur vous renverra l'une ou l'autre des réponses. Vous verrez que cette méthode très simple est extrêmement puissante, et qu'elle vous sera très utile lorsque vous aurez besoin de traiter des requêtes Ajax ou de renvoyer du PDF par exemple.

Gestionnaire de versions : révision 1

Pour retrouver le code comme il devrait être à cette étape, récupérez le code présent sur Bitbucket et mettez-le à jour à la révision 1 avec la commande `hg up -r 1`.

Le modèle de l'objet ticket

Vous venez de mettre en place une première vue dynamique. Si vous connaissez déjà bien le principe du MVC, vous constaterez que nous n'avons fait qu'effleurer le « C » (contrôleur) avec les vues et que nous avons introduit le principe de routage avec les URL. Maintenant, il est temps de rentrer dans le vif du sujet en définissant notre premier modèle.

Pour votre application `ticket`, vous allez avoir besoin d'un objet tâche qui représentera les choses à faire. Vous pouvez le définir comme ceci :

- un nom pour la retrouver facilement ;
- une description de la tâche à accomplir ;
- la date de création ;
- la date à laquelle cette tâche doit être terminée (*due-date*) ;
- la date à laquelle vous aimeriez commencer à travailler sur cette tâche (*schedule-date*) ;
- l'état du ticket : ouvert ou fermé – un ticket fermé étant une tâche réalisée.

Commencez donc à réfléchir aux types dont vous allez avoir besoin :

- un nom : une chaîne de caractères ;
- une description : un texte ;
- la date de création du ticket : une date ;
- la due-date : une date ;
- la schedule-date : une date ;
- fermé ou ouvert : un booléen.

La classe Task

Il suffit d'ouvrir le fichier `models.py` et de créer une classe `Task` comme suit :

Création de la classe Task

```
from django.db import models

class Task(models.Model):
    name = models.CharField(max_length=250)
    description = models.TextField()
    created_date = models.DateField(auto_now_add=True)
    due_date = models.DateField()
    schedule_date = models.DateField()
    closed = models.BooleanField(default=False)
```

La classe que vous venez de créer hérite de la classe `Model`, et chaque attribut de votre classe est affecté à une classe du module `models` qui définit son type. Dans le cas présent, vous utilisez les champs de types suivants.

- `CharField` sera interprété au niveau de la base de données comme une colonne de type `string` (c'est-à-dire une chaîne de caractères). Nous lui ajoutons le paramètre `max_length` qui adjoindra la contrainte `MAX_LENGTH` (longueur maximale) à 250 caractères dans la base de données.
- `TextField` (un texte sur plusieurs lignes) sera interprété au niveau de la base de données comme une colonne de type `text`.
- `DateField` (un objet de type date) sera traduit, au niveau de la base de données, comme un objet de type `date` et, au niveau de Django, comme un objet de type `datetime.datetime`. Le paramètre optionnel `auto_now_add` permet l'autocomplétion de l'attribut à la date du jour lors de sa création.
- `BooleanField` (un objet de type booléen) sera traduit, au niveau de la base de données, comme un objet de type `Boolean`. Le paramètre `default` place l'état fermé de la tâche à « Faux » ; la tâche est donc ouverte.

Pour le moment, ces quelques informations sont suffisantes pour commencer à utiliser votre application. Enregistrez votre fichier `models.py` et indiquez à Django qu'il lui faut créer la table `task`.

Création de la table task

```
[user@local]$ python manage.py syncdb
Creating tables ...
Creating table ticket_task
Installing custom SQL ...
Installing indexes ...
No fixtures found.
```

Avec cette simple commande, Django vient de créer pour vous la table `task` (tâche) et toutes les colonnes que vous avez définies dans le fichier `models.py`.

La commande python manage.py shell

Pour le vérifier, utilisez la commande `python manage.py shell`, très pratique.

```
[user@local]$ python manage.py shell
Python 2.7.1 (r271:86832, Apr 12 2011, 16:16:18)
[GCC 4.6.0 20110331 (Red Hat 4.6.0-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from ticket.models import Task# Importez le modèle Task dans la console
interactive.
>>> Task.objects.all()# Listez tous les objets Task présents dans la base de
données.
[]
```

La réponse que vous renvoie Django est une liste vide, car vous n'avez ouvert aucun ticket pour le moment. Vous allez maintenant essayer d'enregistrer une nouvelle tâche. Comme vous le remarquerez, Django veille sur vous et ne vous permet pas de créer des objets si certaines données sont manquantes.

Création d'une tâche

```
>>> tache = Task()# On instancie un objet vide de la classe Task.
>>> tache.name = "Ma première tâche" # On lui donne un nom
>>> tache.description = "La première tâche d'une longue série" # une description
>>> tache.save() # Essayons de l'enregistrer.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/usr/lib/python2.7/site-packages/django/db/models/base.py", line 460,
in save
    self.save_base(using=using, force_insert=force_insert,
force_update=force_update)
  File "/usr/lib/python2.7/site-packages/django/db/models/base.py", line 553,
in save_base
    result = manager._insert(values, return_id=update_pk, using=using)
  File "/usr/lib/python2.7/site-packages/django/db/models/manager.py", line
195, in _insert
    return insert_query(self.model, values, **kwargs)
  File "/usr/lib/python2.7/site-packages/django/db/models/query.py", line 1436,
in insert_query
    return query.get_compiler(using=using).execute_sql(return_id)
  File "/usr/lib/python2.7/site-packages/django/db/models/sql/compiler.py",
line 791, in execute_sql
    cursor = super(SQLInsertCompiler, self).execute_sql(None)
  File "/usr/lib/python2.7/site-packages/django/db/models/sql/compiler.py",
line 735, in execute_sql
```

```
    cursor.execute(sql, params)
File "/usr/lib/python2.7/site-packages/django/db/backends/util.py", line 34,
in execute
    return self.cursor.execute(sql, params)
File "/usr/lib/python2.7/site-packages/django/db/backends/sqlite3/base.py",
line 234, in execute
    return Database.Cursor.execute(self, query, params)
IntegrityError: ticket_task.due_date may not be NULL
```

Encore une fois, la réponse de Django est on ne peut plus claire :

```
IntegrityError: ticket_task.due_date may not be NULL
```

Il refuse d'enregistrer le ticket que vous venez de créer, car vous n'avez pas précisé de `due_date`. C'est une fonctionnalité très puissante de Django, puisqu'il s'occupe à votre place de toute la partie validation de données ; elle vous sera particulièrement utile, par exemple, lorsque vous devrez créer des formulaires.

Renseignez tous les champs nécessaires à l'enregistrement de votre objet `tache`.

Enregistrement de l'objet task

```
>>> import datetime
# Comme la due-date est de type date, nous importons le module standard datetime
de python.
>>> tache.due_date = datetime.datetime.now() # Une due_date à aujourd'hui
>>> tache.schedule_date = datetime.datetime.now() # Une schedule_date à
aujourd'hui également.
>>> tache.save() # Tous les champs sont maintenant remplis, vous pouvez
enregistrer votre ticket.
>>> Task.objects.all() # Listez tous les objets Task.
[<Task: Task object>]
>>> tache # Vérifiez que les champs autoremplis par Django sont bien exacts.
<Task: Task object> # Hum, ce n'est pas très instructif !
>>> tache.name
'Ma première tâche'
>>> tache.created_date
datetime.date(2011, 7, 29)
>>> tache.closed
False
```

Vous venez de créer le modèle `Task` et de voir comment le manipuler au travers de la console interactive fournie par Django. Vous utiliserez régulièrement cette console dès que vous aurez besoin d'en savoir plus sur un objet, ou tout simplement pour réaliser quelques tests avant d'inclure un modèle dans le flux de votre application. Nous allons maintenant vous montrer à quel point Django peut vous simplifier la vie avec l'une de ses applications les plus puissantes : l'interface d'administration.

Première itération : l'interface d'administration

L'interface d'administration de Django est une application au même titre que l'application `ticket` que vous êtes en train de réaliser. Elle vous permet de créer, de modifier, de supprimer et de lister tous les objets de vos modèles, dans le navigateur.

Nous montrerons plus loin dans ce livre comment la configurer très finement.

Pour activer l'interface d'administration, vous devez reprendre le cheminement que vous connaissez maintenant bien car vous l'avez déjà expérimenté avec l'application `ticket` :

- 1 Ajouter l'application aux settings.
- 2 Créer un lien dans les URL à la racine de votre projet vers votre application.

Tout le reste se fait de façon automatique, bien entendu.

Ajouter l'application aux settings

En réalité, Django vous simplifie le travail au maximum (c'est le principe d'un framework) ; il a déjà balisé le chemin pour vous. Il vous suffit donc, pour activer l'interface d'administration dans les settings, de décommenter la ligne :

```
# 'django.contrib.admin'
```

La partie `INSTALLED_APPS` des settings doit donc ressembler à cela :

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


Ajouter l'application aux URL

Si vous avez bien suivi le processus de mise en place d'une nouvelle application Django dans votre projet, vous connaissez déjà la prochaine étape : elle consiste à ajouter un chemin d'URL depuis votre projet vers le fichier URL de votre nouvelle application. Là encore, Django a balisé le chemin pour vous. Décommentez les lignes suivantes.

Fichier `tracker/urls.py`

```
# from django.contrib import admin
# admin.autodiscover()
# url(r'^admin/', include(admin.site.urls))
```

Un premier test

Vous avez maintenant tout ce qu'il vous faut pour tester l'interface d'administration. Lancez votre serveur de test :

```
python manage.py runserver
```

Puis rendez-vous à l'adresse <http://localhost:8000/admin/> (voir figure 2-9). Le nom d'utilisateur et le mot de passe sont ceux que vous avez définis lorsque vous avez lancé la commande `python manage.py syncdb`.

Figure 2-9

Authentification lors de la connexion à l'interface d'administration



Une fois connecté, voici l'écran que vous allez découvrir (voir figure 2-10).



Figure 2-10 Écran d'accueil de l'interface d'administration

Visiblement, si l'interface d'administration est bien fonctionnelle pour les applications `auth` et `site`, il n'y a aucune trace de l'application que vous venez de créer. Là encore, ce n'est pas une erreur mais une fonctionnalité très utile de Django. En effet, il vous laisse la possibilité de choisir très simplement les applications que vous souhaitez rendre disponibles dans l'interface d'administration. Nous verrons bientôt que cette finesse de configuration va bien au-delà du niveau « application » puisqu'il est également possible de sélectionner les modèles que vous souhaitez rendre disponibles sur l'interface d'administration.

Ajouter le modèle `Task` à l'administration

Pour ajouter le modèle `Task` à l'interface d'administration, il suffit de le signifier à Django via le fichier `admin.py` de chaque application. Voici une version très simple, mais fonctionnelle, de ce fichier.

Fichier `admin.py`

```
# Importez tout d'abord l'application admin dans ce fichier.
from django.contrib import admin
# Importez le modèle que vous souhaitez rendre disponible.
from models import Task

# Indiquez à Django que vous souhaitez le rendre disponible dans l'interface
d'administration.
admin.site.register(Task)
```

Rechargez votre navigateur pour y voir apparaître les changements (voir figure 2-11).

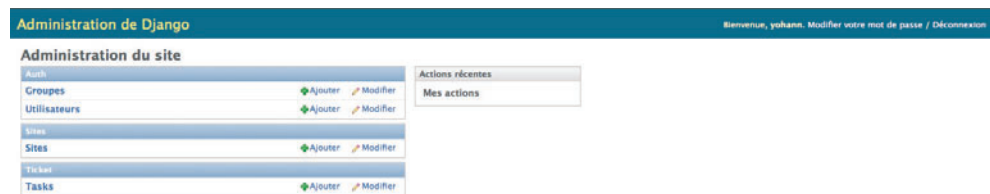


Figure 2-11 L'application ticket apparaît désormais.

L'application `ticket` apparaît désormais sur l'écran d'accueil de Django et le modèle `Task` peut être géré entièrement depuis l'interface d'administration.

Gestionnaire de versions : révision 2

Retrouvez cette étape à la révision 2 du projet.

Une interface CRUD

L'interface d'administration est entièrement CRUD (*Create, Read, Update, Delete*, soit « créer, lire, modifier, supprimer »).

Un premier aperçu

Rendez-vous à l'adresse <http://localhost:8000/admin/ticket/> (voir figure 2-12). Sur cette page sont listés tous les modèles de l'application `ticket`. Pour le moment, seul un modèle est disponible, le modèle `Task`.

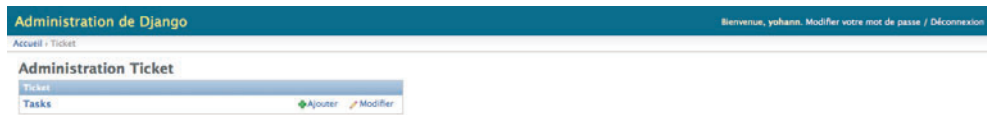


Figure 2-12 Administration de l'application ticket

À l'adresse <http://localhost:8000/admin/ticket/task> sont listés tous les objets du modèle `Task`. Pour le moment, seul un objet est présent : celui que vous avez créé via `python manage.py shell`. En cliquant sur son nom, *Task object*, vous accédez à un formulaire qui vous permet de le modifier (voir figure 2-13).

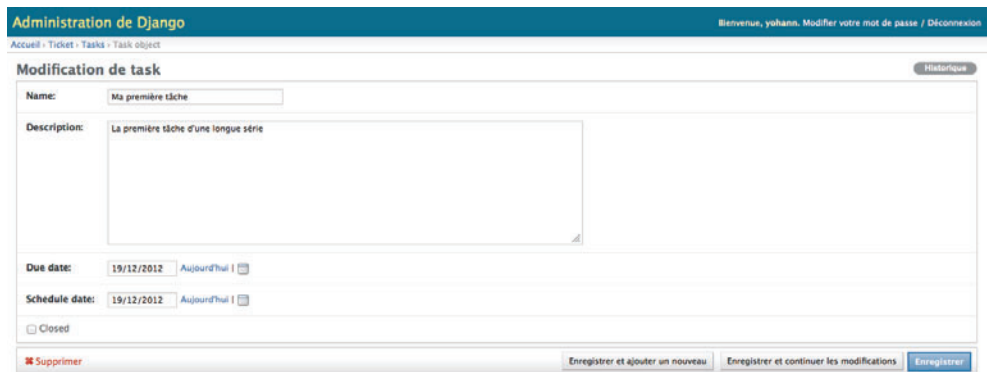


Figure 2-13 Le formulaire d'édition

Le lien *Supprimer*, qui pointe vers <http://localhost:8000/admin/ticket/task/1/delete/>, efface le ticket que vous aviez créé. Enfin, le lien <http://localhost:8000/admin/ticket/task/add/> propose, au contraire, un formulaire d'édition pour créer de nouveaux tickets.