Pierre Alexis et Hugues Bersini

1 étude de cas inspirée de Facebook !

Apprendre la programmation web avec Python et Django

Principes et bonnes pratiques pour les sites web dynamiques

EYROLLES

Hugues Bersini et Pierre Alexis

Membre de l'Académie Royale de Belgique, Hugues Bersini enseigne l'informatique et la programmation aux facultés polytechnique et Solvay de l'Université Libre de Bruxelles, dont il dirige le laboratoire d'Intelligence Artificielle. Il est l'auteur de très nombreuses publications (systèmes complexes, génie logiciel, sciences cognitives et bioinformatique).

Pierre Alexis est titulaire d'une licence en informatique et travaille au service informatique de la plus grande banque privée belge. Intéressé depuis toujours par l'enseignement et la vulgarisation informatique, il assiste Hugues Bersini à l'Université libre de Bruxelles et dispense un cours de Django/Python à destination d'étudiants en ingénierie commerciale.

Enfin un ouvrage pour étudiants détaillant tous les principes de la programmation web moderne, avec l'un des frameworks de développement web les plus ambitieux : Django, basé sur le langage Python !

Un manuel autonome reprenant tous les fondements de la programmation web, au fil d'une étude de cas inspirée de Facebook

Abondamment illustré d'exemples inspirés de Facebook et rappelant les bonnes pratiques du domaine (modèle MVC, diagrammes UML, patterns), voici un livre de cours magistral et moderne sur la programmation web dynamique, que tous les enseignants en informatique peuvent utiliser. Complet et autonome, il pose solidement les fondamentaux de la conception web, avec ou sans framework : HTML5/CSS3, dynamisme alimenté par bases relationnelles SQL, sessions, JavaScript et Ajax, sans oublier de fournir au lecteur d'essentiels rappels en programmation objet, voire de montrer… ce qu'il ne faut pas faire en CGI !

Le langage Python et le framework Django sont introduits en douceur, et l'utilisation des vues, templates, formulaires et modèles Django, conformément aux principes MVC exposés dans la première partie, est illustrée au fil de l'étude de cas. L'annexe complète le manuel par une description pas à pas de l'installation de l'environnement de développement, tant sous Windows et Mac OS X que sous GNU/Linux : Python, Django, Eclipse, PyDev et les Web Developer Tools.

À qui s'adresse ce livre ?

- Étudiants en informatique (IUT, écoles d'ingénieurs) et leurs enseignants ;
- Développeurs web (PHP, Java, etc.) qui souhaitent passer à Python & Django ;
- Développeurs C, C++ qui souhaitent une introduction systématique à la programmation web.

Au sommaire

Fonctionnement d'un site web • HTTP et notion de serveur web • Statique versus dynamique • Passer des paramètres à une page • **Programmation orientée objet et framework MVC** • Modularité, simplicité et réutilisabilité • Travail d'équipe • Cas d'utilisation (use cases) • Séparation modèle-vue-contrôleur • **Le langage Python** • Déclaration et initialisation des variables • Types et transtypage • Copie de variables • Opérations sur les types simples int, float et string • Listes et dictionnaires • Instructions conditionnelles et boucles while • Fonctions • Variables locales et globales • Classes et objets • Variables de classe (attributs statiques) et d'instance • Héritage et polymorphisme • Accès aux bibliothèques Python avec import et from • **Structurer les pages avec HTML5** • Encodage • Balises et éléments HTML : en-têtes, pieds de page et sections, liens hypertextes, listes, images, mise en évidence du texte, formulaires, etc. • **Mettre en forme avec les feuilles de styles CSS** • Propriétés CSS et principes de sélection d'éléments (sélecteurs) • Dimension et imbrication des éléments • Positionnement par défaut • Sortir des éléments du flux • **Dynamiser les pages à la volée avec JavaScript** • Événements • DHTML, jQuery et frameworks JavaScript • **Une étude de cas inspirée de Facebook** • Cas d'utilisation et maquette du site (wireframes) • Modèle de données et rappel sur les bases relationnelles • Clés primaires et étrangères • Relation 1-*n*, 1-1 et *n*-*n* • Correspondance relationnel/objet (ORM) • Diagramme de classes • **Premier contact avec les bases relationnelles SQL** • Un exemple en CGI : ce qu'il ne faut plus faire • **Les vues Django : orchestration et architecture** • Configurer Django pour écrire du XHTML 5 • **Les templates Django : séparation et réutilisation des rendus HTML** • Langage des templates • Variables, sauts conditionnels et boucles, héritage et réutilisation de templates • **Les formulaires Django** • L'objet request • Formulaire de login • Gestion du message d'erreur • La bibliothèque forms de Django • Valider l'adresse et le mot de passe • **Les modèles Django** • Créer la base de données et le compte administrateur • Regénérer la base • Configurer l'interface d'administration de la base de données • Générer des formulaires à partir de modèles (ModelForms) • **Comprendre et utiliser les sessions** • Manier une variable de session • Authentification • Cookie versus session • Protéger des pages privées • **Des sites encore plus dynamiques grâce à Ajax** • **Annexe : installer l'environnement de développement sous Linux, Mac OS X et Windows** • Python, Django, Eclipse, PyDev et les Web Developer Tools.

Sur le site www.editions-eyrolles.com

Le code source de l'étude de cas est disponible sur le site d'accompagnement du livre.

www.editions-eyrolles.com
Code article : G13499
ISBN : 978-2-212-13499-5

Apprendre la programmation web avec Python et Django

Principes et bonnes pratiques pour les sites web dynamiques

Dans la collection Noire

P. Cegielski. – **Conception de systèmes d'exploitation**. *Le cas Linux*.
N°G11479, 2e édition, 2004, 680 pages.

J. Engels. **HTML5 et CSS3**. *Cours et exercices corrigés*.
N°13400, 2012, 550 pages.

G. Swinnen. **Apprendre à programmer avec Python 3**.
N°13434, 3e édition, 2012, 435 pages.

H. Bersini. **La programmation orientée objet**. *Cours et exercices en UML 2 avec Java 6, C# 4, C++, Python, PHP 5 et LinQ*.
N°12806, 5e édition, 2011, 644 pages.

É. Sarrion. – **jQuery et jQuery UI**.
N°12892, 2011, 132 pages.

A. Brillant. **XML - Cours et exercices**.
N°12691, 2e édition, 2010, 336 pages.

Chez le même éditeur

Développer soi-même son site web avec HTML, CSS, PHP, JavaScript

R. Rimelé. – **HTML5.**
N°12982, 2011, 600 pages.

F. Draillard. – **Premiers pas en CSS et HTML**.
N°13338, 2011, 464 pages.

R. Goetter. – **CSS avancées.** *Vers HTML5 et CSS3*.
N°13405, 2e édition, 2012, 385 pages.

R. Goetter. – **CSS 2 : pratique du design web**.
N°12461, 3e édition, 2009, 340 pages.

R. Rimelé. – **Mémento HTML5.**
N°13420, 2012, 14 pages.

R. Goetter. – **Mémento CSS 3**.
N°13281, 2011, 14 pages.

É. Daspet et C. Pierre De Geyer. **PHP 5 avancé**.
N°13435, 6e édition, 2012, 870 pages.

C. Porteneuve. – **Bien développer pour le Web 2.0**.
N°12391, 2e édition, 2008, 674 pages.

Développer pour le Web mobile

F. Daoust, D. Hazaël-Massieux. – **Bonnes pratiques pour le Web mobile.** *Conception et développement*.
N°12828, 2011, 300 pages.

T. Baillet. **Créer son thème WordPress mobile** *en HTML5 et CSS3*.
N°13441, 2012, 128 pages.

É. Sarrion. – **XHTML/CSS et JavaScript pour le Web mobile**. *Développement iPhone et Android avec et iUI et XUI.*
N°12775, 2010, 274 pages.

É. Sarrion. – **jQuery Mobile**.
N°13388, 2012, 601 pages.

Ressources autour du Web : design, ergonomie, bonnes pratiques

A. Boucher. – **Ergonomie web.** *Pour des sites web efficaces*.
N°13215, 3e édition, 2011, 380 pages.

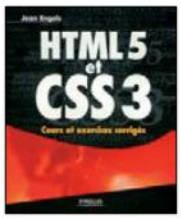
A. Boucher. – **Ergonomie web illustrée**. *60 sites à la loupe*.
N°12695, 2010, 302 pages (Design & Interface).

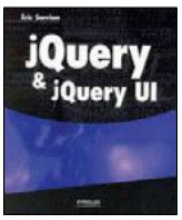
A. Boucher. – **Mémento Ergonomie web.**
N°12698, 2e édition, 2010, 14 pages.

E. Sloïm. – **Mémento Sites web.** *Les bonnes pratiques*.
N°12802, 3e édition, 2010, 18 pages.

O. Andrieu. – **Réussir son référencement web.**
N°13396, 2012, 480 pages.

I. Canivet. – **Bien rédiger pour le Web.** *Stratégie de contenu pour améliorer son référencement*.
N°12883, 2e édition, 2011, 540 pages.





Pierre Alexis et Hugues Bersini

Apprendre la programmation web avec Python et Django

Principes et bonnes pratiques pour les sites web dynamiques

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Remerciements à Anne Bougnoux et Thomas Petillon
pour leurs relectures et contributions

Avant-propos

Cet ouvrage se base sur Django, un outil de création de sites dynamiques écrit en Python, langage aujourd'hui très populaire pour apprendre à programmer. Ce livre enseigne la programmation web de façon aussi moderne que possible et s'adresse aussi bien aux enseignants qu'à leurs élèves. Il vous demande peu de pré-requis, mais reste ambitieux, de sorte que vous n'aurez pas à rougir devant les professionnels.

SOMMAIRE

- Résumer l'ambition de ce livre
- Resituer Django et Python dans l'évolution des technologies logicielles
- À qui s'adresse ce livre
- Plan de l'ouvrage

CULTURE Du Web préhistorique au Web 2.0 moderne et « dynamique »

Il y a vingt ans, on déambulait entre les pages du Web comme entre deux tableaux de maître dans un musée ; seule différence, on était assis face à son écran, souris à la main. Au fil des clics, les sites web s'affichaient : ils étaient déjà jolis et colorés, riches en information et éminemment utiles… mais inaltérables et rétifs à toute autre forme d'interférence que ce vagabondage de page en page.

Il était impossible pour l'internaute d'y écrire son numéro de carte de crédit pour acquérir une toile, de réserver sa place pour une visite du musée, de s'enregistrer comme utilisateur du site et d'épingler sur le sourire de Mona Lisa un commentaire ou des moustaches. Vous reveniez sur la même page quelques minutes plus tard ? Le contenu était inchangé. Tout était placidement statique ; en un mot, le Web fonctionnait pour l'essentiel dans une seule direction : des serveurs d'information vers les clients. Les utilisateurs étaient à la merci de ce que les développeurs côté serveur avaient choisi de leur dévoiler. Le serveur ne percevait que le clic d'entrée dans le site et se bornait à renvoyer en retour la page sollicitée ; il était sourd et indifférent à toute autre requête. Le Web d'alors, certes impressionnant et révolutionnaire, était statique.

James Gosling, le concepteur du langage de programmation Java (dont le succès devra beaucoup au Web), présenta en 1995 un nouveau concept de navigateur lors d'une conférence consacrée au Web. Il fit d'abord apparaître la représentation en 2D d'une molécule dans un navigateur web classique, laissant l'assistance de marbre. Quels ne furent la surprise et l'enthousiasme de ce même public lorsqu'à l'aide de sa souris, il se mit à bouger la molécule ! Elle réagissait aux stimuli de la souris, obéissait aux injonctions de l'internaute. Un programme Java, appelé « applet », s'exécutait dans le navigateur (Mosaic, à l'époque) rendant cette interaction effective. Le Web venait de gagner ses premiers galons de dynamisme et d'interactivité. Il en devenait chatouilleux.

En 2004, prenant conscience que l'utilisateur passait de spectateur passif à acteur essentiel du fonctionnement du Web, sursaturant ce dernier de ses vidéos de vacances, son savoir, ses copains de lycée ou ses prises de position politique, Tim O'Reilly décida de couronner le Web 2.0. Ce nouveau Web, aussi royal qu'il fût, devenait complètement assujetti au bon vouloir de ses sujets qui, au-delà des sempiternels clics de navigation, pouvaient dorénavant commenter, pérorer, refuser, voter, voler, acheter, vendre, négocier, réserver, prêter, jouer, écouter, visionner, projeter, et tant d'autres activités encore l'éloignant de l'écran de télévision pour celui de son PC. La confiscation du Web par cette multiplicité d'ego, choisissant d'en faire cette excroissance narcissique qu'il est devenu, fut considérée à ce point révolutionnaire que le célèbre hebdomadaire américain « Time » choisit d'élire *You* la personne de l'année 2006.

L'affichage d'un site web selon le bon vouloir de son créateur nécessite la maîtrise d'un langage de présentation de contenu appelé HTML (XHTML ou HTML5, vous comprendrez ces nuances par la suite), décrivant et paramétrant l'organisation de la page, la taille et le style des différents éléments de celle-ci. C'est le seul langage compris par les navigateurs, qui affichent la page de la manière précisée et transmise dans les instructions HTML.

Rendre la page « dynamique » requiert, de façon un peu plus exigeante, la maîtrise d'un langage de programmation comme il en existe malheureusement trop : JavaScript, C++, Java, .Net, Python, PHP, Ruby, SmallTalk, Eiffel, OCaml et tant d'autres. Ainsi, la mise en place d'un site web dynamique capable de se comporter comme un programme, réagissant comme il se doit aux sollicitations de son utilisateur, exige de mixer ces deux savoirs dans une proportion dépendant de l'attention accordée soit à son interactivité, soit à son apparence.

Heureusement, depuis l'apport pionnier des applets et des servlets Java, de multiples technologies logicielles sont apparues afin de combiner efficacement programmation et physionomie d'un site. Qui, dans le monde informatique, n'a entendu parler de PHP, JSP (dans la mouvance Java) ou ASP.Net (offre de Microsoft), qui structurent les milliards de sites interactifs disponibles aujourd'hui sur le Web ? Au-dessus de ces langages apparaissent de multiples boîtes à outils logicielles facilitant la conception des plates-formes web : Symfony et Zend pour PHP, Spring pour Java et, sujet de ce livre, Django pour Python.

Elles font partie de ces multiples solutions, relativement équivalentes (même si bien entendu leurs concepteurs s'en défendront avec vigueur), facilitant la prise en charge conjointe de la présentation de la page et de son interactivité.

Bien que ces différentes technologies puissent également s'implémenter côté client, cet ouvrage portera principalement sur celles qui s'exécutent côté serveur, les plus sollicitées. Le code exécuté côté serveur a pour mission de produire à destination du client une page répondant à ses desiderata. Pour ce faire, il est fréquemment nécessaire d'obtenir et d'exploiter des informations stockées dans une base de données relationnelle et de glisser ces dernières de manière judicieuse entre les instructions HTML5 qui se chargent de clairement les présenter.

Pourquoi cet ouvrage ?

Programmation, HTML5/CSS3, bases de données, les technologies se multiplient et semblent exiger de la part des développeurs web un CV épais et noirci d'acronymes incompréhensibles pour la plupart des directeurs du personnel. La tentation est grande de les laisser sur le bas-côté du cursus universitaire, malgré leur omniprésence et une importance stratégique grandissante pour les entreprises.

C'est pour combler ce fossé entre la réalité mouvante des technologies web et le programme plus installé et stable du monde universitaire – d'aucuns diraient un peu poussiéreux et envahi de toiles… d'araignées – que Python et Django sont apparus et ont changé la donne.

Le choix de Python et de Django

Depuis quelques années, grâce à Python et Django, il est devenu possible d'aborder en douceur le cocktail de technologies informatiques à la base de la programmation web. Ils ont pour eux la simplicité d'utilisation. Ainsi, le langage Python est très sou-

vent plébiscité pour sa facilité d'usage, sa rapidité d'acquisition, ce qui en fait un langage de prédilection pour l'apprentissage de la programmation. Son utilisation est élémentaire à démarrer et très interactive. Les instructions étant plus simples et plus intuitives, on parvient plus rapidement au résultat escompté. Bien que l'exécution de ces codes conduise au même message à l'écran, `print ("Hello World")` en Python est incontestablement plus simple à écrire mais aussi à comprendre que le fameux `public static void main (String[] args) {System.out.println "Hello World"}` rédigé avec stupeur par tous les débutants en programmation Java.

Langage de programmation disponible pour plusieurs plates-formes, Python est donc simple d'emploi : pas de typage explicite, instructions concises, structures de données d'un usage élémentaire (listes, dictionnaires et chaînes de caractères). Pour autant, il n'en préserve pas moins les fonctions essentielles de tout langage réputé puissant. Il semble, un peu mieux que les autres, réussir à concilier efficacité et simplicité, à se positionner entre ces deux acteurs exigeants que sont, d'une part, le processeur qui cherche l'efficience et l'économie de mémoire et, d'autre part, le programmeur qui cherche la simplicité et la clarté d'utilisation.

EN PRATIQUE Des goûts et des couleurs en programmation…

Bien sûr, en matière de langage de programmation, ce genre de préférence tient pour beaucoup de l'acte de foi et il en va des guerres de langages comme de celles de religions ou de goûts culinaires. Toutefois, suffisamment d'enseignants ont fait de Python le premier langage à enseigner aux étudiants pour nous conforter dans notre choix.

Django, entièrement développé et basé sur le langage Python, fut à l'origine développé pour faciliter et accélérer la programmation de sites web dans un milieu journalistique (le journal Lawrence Journal-World publié quotidiennement dans la ville de Lawrence au Kansas), donc pas vraiment versé dans les technologies de développement logiciel. Les sites web en question exigeaient d'être développés et mis à jour à un rythme très élevé ; les outils devaient pour cela être faciles à prendre en main et les logiciels produits aisément réutilisables. Il était de surcroît préférable de séparer le métier d'ergonome de sites web, préoccupé pour l'essentiel par l'apparence et la physionomie du site, du métier de développeur, préoccupé par le fonctionnement optimal et sécurisé d'un code dont l'écriture devait rester très abordable et facilement modifiable.

Django est devenu cette boîte à outils logicielle (*framework* dans le jargon professionnel), ce canif suisse du développement web. Aujourd'hui, cette aventure se prolonge selon le modèle collaboratif de l'open source, impliquant des dizaines de milliers d'utilisateurs et de développeurs passionnés tout autour de la planète.

Ainsi, à l'origine de la programmation web, la technologie CGI à exécuter côté serveur donnait naissance à d'horribles codes comme celui ci-dessous (pour afficher une liste d'employés d'un service stockée dans une base de données relationnelle) :

EXEMPLE 0.1 À l'ancienne mode CGI, tout est mélangé !

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

import sqlite3
import cgi

print "Content-Type: application/xhtml+xml; charset=utf-8\n"

print '<?xml version="1.0" encoding="UTF-8" ?>'
print '<!DOCTYPE html>'
print '<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">'
print ' <head>'
print '    <title>eCarnet - Employés d'un service</title>'
print ' </head>'
print ' <body>'

form = cgi.FieldStorage()
service_id = str(form["service"].value)
db_connection = sqlite3.connect('database.sqlite3')
db_connection.row_factory = sqlite3.Row
cursor = db_connection.cursor()
cursor.execute("SELECT nom FROM service WHERE id=" + service_id)
row = cursor.fetchone()
service_nom = str(row['nom'])

print '    <h1>Employés du service « ' + service_nom + ' »</h1>'

cursor.execute("SELECT prenom, nom, tel_fixe FROM employe
               ➥ WHERE id_service=" + service_id)
rows = cursor.fetchall()

print '    <ol>'
for row in rows:
  print '      <li>' + row['prenom'] + ', ' + row['nom'] + ', '
               ➥ + row['tel_fixe'] + '</li>'
print '    </ol>'
print ' </body>'
print '</html>'

db_connection.close()
```

Sans qu'il soit nécessaire d'en saisir toutes les lignes, tout informaticien, même débutant, découvrira un mélange plutôt indigeste de trois réalités informatiques fonda-

mentales pourtant assez faciles à tenir distinctes, trois langages dont l'usage correct évite ce charabia pénible. Ces trois langages sont le SQL pour la gestion et l'interrogation des bases de données relationnelles, le Python pour la programmation et le HTML5 pour l'affichage et l'apparence des pages web.

C'est le pari réussi de Django de maintenir ces trois réalités informatiques suffisamment différentiables pour, par exemple, épargner à l'artiste des pages web les subtilités du stockage relationnel et les joies de la programmation. De surcroît, adopter une technologie alternative d'affichage ou de stockage de données devrait laisser suffisamment inaltérés les autres aspects du développement, contribuant ainsi à stabiliser l'intégralité du développement (souci essentiel d'une informatique dont les technologies ne cessent d'évoluer et de projets logiciels dont le cahier des charges se modifie en continu). Cette séparation découle d'une recette de conception bien connue des informaticiens et dénommée MVC (Modèle Vue Contrôleur), que nous aurons l'occasion de présenter largement par la suite, recette qui facilite à la fois la vie du développeur et celle de l'enseignant de cette pratique de développement.

À qui s'adresse cet ouvrage ?

De par la simplicité de la syntaxe Python, les outils de développement web mis à disposition par Django, ainsi que le souci de ce dernier de maintenir les aspects esthétiques et les besoins du développement assez logiquement séparés, cet ouvrage présentant à la fois Python et Django vise pour l'essentiel trois types de lecteurs.

- Tout d'abord, les **enseignants du monde universitaire et des écoles d'ingénieurs**, tout spécialistes qu'ils soient en programmation, en bases de données relationnelles ou en technologies de présentation de contenu web (HTML5/CSS3, XHTML…) verront un intérêt à cet ouvrage. L'expérience de l'un des auteurs, Hugues Bersini, le montre. Il enseigne la programmation depuis 25 ans dans différentes facultés bruxelloises. Pour la partie plus appliquée et expérimentale de son enseignement, les étudiants sont ravis de mettre en œuvre le langage Python dans le cadre du développement web. Ils se plaisent à faire fonctionner une bibliothèque ou vidéothèque sur le Web, un site bancaire ou de commerce électronique. Ces projets leur donnent l'impression d'être en phase avec la réalité de l'informatique telle qu'elle se pratique aujourd'hui. Nombreux sont ceux qui, ayant découvert Python et Django, leur sont restés fidèles dans la vie professionnelle, dans leur entreprise ou leur start-up.
- Évidemment, les **étudiants** suivant cette même formation trouveront de quoi facilement concevoir une variété infinie d'autres sites web que celui présenté dans cet ouvrage. Au-delà de Django et Python, ils se sensibiliseront au métier de

développeur web, ses recettes, ses exigences, et les multiples solutions logicielles capables de leur mâcher la besogne.

- La troisième cible pouvant tirer profit de la lecture de ce livre comprend les **professionnels impliqués dans le développement web** en quête d'une entrée en douceur dans cet univers technologique somme toute assez exigeant. Même si leur environnement professionnel les contraint à l'une ou l'autre technologie web différente de Django (basée sur ASP.Net, Java/JSP, PHP, Ruby on Rail…), ce qu'ils découvriront à la lecture de ce livre, (les différentes technologies de programmation, stockage relationnel, modèles en HTML ou en CSS, le besoin d'une claire séparation entre elles, la réutilisation de fonctionnalités) leur demeurera d'une grande utilité. Et pourquoi pas, si cette liberté leur est offerte, ils pourront faire le choix de Django pour leur entreprise, plateforme logicielle légère et formidablement réactive ! Pierre Alexis, l'autre auteur de ce livre qui assiste Hugues Bersini dans son enseignement, peut en témoigner dans cet ouvrage commun !

Le plan du cours

> **RESSOURCES EN LIGNE** **Code source du projet Trombinoscoop**
>
> Le code source du site développé dans le livre est librement téléchargeable depuis la fiche du livre sur le site des éditions Eyrolles.
>
> ▸ www.editions-eyrolles.com/livre/9782212134995

Notre propos est découpé en deux grandes parties : nous présentons d'abord l'essentiel des aspects théoriques pour ensuite décrire en détail un projet concret.

La première partie a comme objectif de vous donner toutes les bases indispensables pour comprendre ce qu'est un site web et comment il fonctionne. Nous y présentons à la fois des notions théoriques (web dynamique, MVC, programmation objet, bases de données relationnelles) et les langages nécessaires à leur mise en œuvre (HTML5, CSS, JavaScript, SQL, Python).

- Le **premier chapitre** aborde les rudiments théoriques de la programmation web : HTTP, URL, notions de Web statique et dynamique, de serveurs web et de serveurs de bases de données.
- Le **chapitre 2** insiste sur la nécessité de séparer les tâches de programmation, de présentation et de structure/stockage des informations. Nous y présentons le modèle MVC et la programmation orientée objet. Nous expliquons en outre la notion de *framework* de développement et montrons en quoi l'utilisation d'un framework tel que Django facilite la tâche du développeur.

- Le **chapitre 3** balaie les notions élémentaires de programmation indispensables à l'utilisation de Python : variables, structures de contrôle, fonctions, rudiments de programmation objet. Ce chapitre suffira à comprendre les exemples de code Python présentés dans l'ouvrage et, de manière générale, la plupart des applications développées sous Django.

Aller plus loin

Ce livre n'est pas un manuel approfondi d'utilisation de Python. D'autres ouvrages existent pour cela, dont l'excellent livre de Gérard Swinnen :

📖 *Apprendre à Programmer avec Python*, Gérard Swinnen, Eyrolles 2012

- Le **chapitre 4** s'intéresse aux langages de présentation des pages web, HTML5 et CSS, ainsi qu'à la manière de rendre les pages plus interactives et plus dynamiques (côté client cette fois) par l'utilisation du langage JavaScript.
- Le **chapitre 5** décrit le travail préliminaire à toute programmation web : l'élaboration d'éléments d'analyse qui permettent d'avoir une vue claire et réfléchie sur le projet qu'on désire développer. À cet effet, nous verrons ce que sont cas d'utilisation, wireframes et modèle de données. Pour appuyer notre propos, nous analyserons le projet développé dans la seconde partie de l'ouvrage. Nous fournissons également dans ce chapitre les notions essentielles concernant les bases de données relationnelles.
- Même si Django masque les échanges avec les bases de données, il est nécessaire d'acquérir les rudiments du langage SQL qui sert à dialoguer avec ces dernières. C'est l'objet du **chapitre 6**. Nous nous y appuyons sur les scripts CGI. Les limitations de cet ancêtre des technologies de programmation web démontrent la nécessité d'utiliser le MVC.

La deuxième partie met en application de façon concrète toutes ces notions et langages. Nous y décrivons étape par étape la construction d'un projet web complet inspiré de Facebook en nous appuyant sur le framework Django.

- Le **chapitre 7** s'intéresse à l'architecture et à l'orchestration du site. Il présente ce qu'on appelle les « vues » sous Django, ce qui correspond à l'aspect *Contrôle* du MVC (attention à la confusion dans les termes).
- Le **chapitre 8** sépare ce qui concerne la présentation du site. Les rendus HTML deviennent réutilisables. Cela se fait grâce aux « templates » de Django et correspond à l'aspect *Vue* du MVC.
- Le **chapitre 9** traite séparément le transfert de données entre l'utilisateur et le site, par l'intermédiaire des formulaires.

- Dans le **chapitre 10**, nous voyons comment décrire et implémenter les « modèles » avec Django (ce qui correspond à l'aspect *Modèle* du MVC).
- Élément indispensable aux sites web modernes, la gestion des sessions garantit une bonne authentification des utilisateurs et la persistance momentanée de leurs données. Cela fait l'objet du **chapitre 11**.
- Nous terminons l'élaboration de notre site exemple en construisant dans le **chapitre 12** les pages qui manquent encore.
- Le **chapitre 13** propose d'aller plus loin en ajoutant encore plus de dynamisme aux pages web grâce à la technologie Ajax.

Enfin, l'**annexe** décrit les étapes à suivre afin d'installer les outils de développement, dont Python et Django, quel que soit le système d'exploitation que vous utilisez (Windows, Mac OS X ou Linux). Les exemples d'application étant développés à partir de l'environnement Eclipse, le chapitre en décrit également l'installation, la configuration et la prise en main (intégrant l'emploi de Django).

Remerciements

Pierre et Hugues adressent à Xavier Devos, Gaël Rabier et Jonathan Unikowski leurs remerciements les plus sincères pour leur suivi, leur relecture attentive et les nombreuses corrections et améliorations qui leur sont dues dans les pages qui vont suivre. Un immense merci aussi à l'équipe éditoriale d'Eyrolles : Muriel Shan Sei Fan pour, tout d'abord, y croire dur comme fer, et nous communiquer l'enthousiasme, les repères et les balises essentiels à la bonne conduite d'un tel projet, Laurène Gibaud et Anne Bougnoux pour le formidable travail de remaniement et Gaël Thomas… pour avoir transformé l'essai en une impeccable mise en forme.

Table des matières

Avant-propos V
Pourquoi cet ouvrage ? VII
Le choix de Python et de Django VII
À qui s'adresse cet ouvrage ? X
Le plan du cours XI
Remerciements XIII

PREMIÈRE PARTIE
Les notions essentielles 1

CHAPITRE 1
Comment fonctionne un site web ? 3
Qu'est-ce que le Web ? 4
Fonctionnement d'un site web statique 5
Le protocole HTTP 7
L'URL, adresse d'une page web 8
Le serveur web : à la fois ordinateur et logiciel 9
Des sites web qui se mettent à jour tout seuls 9
Fonctionnement d'un site web dynamique 11
Utilisation d'une base de données 12
Passage de paramètres à une page web 13
Ai-je bien compris ? 14

CHAPITRE 2
Programmation orientée objet et framework MVC 15
Des programmes modulaires 16
Une écriture simplifiée 16
Des modules réutilisables 16
Un travail d'équipe facilité et plus efficace 16
Les langages orientés objet 17
Les cas d'utilisation (use cases) et le MVC 18

Le principe de la séparation modèle-vue-contrôleur (MVC) 18
Le diagramme des cas d'utilisation (use cases) 20
Correspondances entre MVC et cas d'utilisation 21
Django et le MVC 23
Le framework Django 23
Ai-je bien compris ? 24

CHAPITRE 3
Rappels sur le langage Python 25
Qualités et défauts de Python 26
Qualité : la simplicité 26
Défaut : la simplicité ! 27
Les bases : variables et mémoire centrale 28
Déclaration et initialisation des variables 28
Type des variables 29
Modification et transtypage des variables 29
Copie de variables 30
Se renseigner sur une variable 30
Quelques opérations sur les types simples 31
Le type int 31
Le type float 32
Le type string 33
Les types composites : listes et dictionnaires 34
Les listes 35
Les dictionnaires 36
Les instructions de contrôle 37
Les instructions conditionnelles 37
Aiguillage à deux directions : instruction if (...) else 37
Aiguillage à plus de deux directions : instruction if (...) elif (...) else 39
Les boucles 40
La boucle while 40
La boucle for (...) in 41
Les fonctions 43
Les variables locales et les variables globales 44
La programmation objet 45
Les classes et les objets 45
L'association entre classes 47
Héritage et polymorphisme 51
Import et from : accès aux bibliothèques Python 54
Ai-je bien compris ? 55

CHAPITRE 4

Rappels sur HTML5, CSS et JavaScript..... 57

Structurer les pages web avec HTML5 59
Le concept de « balises » 60
Structure d'un document HTML 63
L'encodage de la page 65
Quelques éléments HTML 66
Principaux éléments de structuration HTML 66
Éléments de structuration annexes : en-têtes, pieds de page et sections 68
Les liens hypertextes 70
Les listes 71
Les images 72
Mise en évidence du texte 73
Les formulaires 75
Autres éléments HTML 78
Mettre en forme avec les feuilles de styles CSS 78
Les propriétés CSS 79
Les sélecteurs CSS 79
Sélectionner toutes les balises de même nom 80
Sélectionner un élément particulier : id en HTML et # en CSS 80
Sélectionner quelques éléments de même nature : class en HTML et . en CSS 81
Appliquer une propriété seulement quand l'élément est dans un état donné 81
Combiner les sélecteurs 82
Sélectionner tous les éléments 83
Lier CSS et HTML 83
Placer le code CSS dans les balises HTML 83
Placer le code CSS dans l'en-tête du fichier HTML 84
Placer le code CSS dans un fichier séparé 85
Dimensions des éléments en CSS 86
Imbrication des éléments (boîtes) 87
Positionnement par défaut des éléments en CSS 88
Sortir certains éléments du flux 89
Application à un exemple concret plus complet 91
Dynamiser les pages web « à la volée » avec JavaScript 96
Les événements 96
Langage de programmation de scripts 96
Un premier exemple de DHTML 97
jQuery et les frameworks JavaScript 99
Ai-je bien compris ? 101

CHAPITRE 5
Mise en application : un site web inspiré de Facebook 103
Notre site web : Trombinoscoop 104
Les cas d'utilisation 105
Maquette du site : les wireframes 106
L'écran d'authentification 107
L'écran de création d'un compte 108
L'écran d'accueil 108
L'écran de modification du profil 108
L'écran d'affichage d'un profil 108
L'écran d'ajout d'un ami 108
Scénario complet du site 112
Modèle de données et petit rappel sur les bases de données relationnelles 113
Clé primaire et clé étrangère 114
Relation 1-n 114
Relation 1-1 116
Relation n-n 117
La problématique de la mise en correspondance relationnel/objet 118
Avec Django 119
Retour au modèle de données de Trombinoscoop : son diagramme de classes 119
Des personnes : étudiants et employés 120
... qui travaillent ou étudient à l'université 120
... et qui échangent des messages avec des amis 122
Ai-je bien compris ? 123

CHAPITRE 6
Premier contact avec les bases relationnelles et SQL à partir d'un exemple en CGI..... 125
Analyse du carnet d'adresses : des cas d'utilisation au modèle de données 127
Trois cas d'utilisation simples 127
Maquette des écrans (wireframes) 127
Le modèle de données du carnet 127
Création de la base de données avec SQLite 129
Accès à la base de données via SQL 131
Syntaxe des requêtes SQL les plus courantes 131
Quelques exemples liés à notre base de données 132
Réalisation du carnet d'adresses avec SQL et CGI 133
Lancement d'un serveur web Python 133
L'écran d'accueil de l'ULB 135
La page principale du carnet d'adresses 136

La liste des employés d'un service 140
Ajout d'un employé 143
CGI : ce qu'il ne faut plus faire 145
Ai-je bien compris ? 145

DEUXIÈME PARTIE
Mise en application avec Django 147

CHAPITRE 7
Les vues Django : orchestration et architecture 149
Utilité des vues 151
Le fichier urls.py 152
Le fichier views.py 153
Enfin ! Notre première page web en Django 155
Lancement de l'environnement Eclipse et création du projet 155
Le fichier urls.py 158
Le fichier views.py 158
Importation de la fonction dans urls.py 160
Configuration de Django pour qu'il écrive du XHTML 5 160
Test de notre ébauche de site 161
Bel effort, mais… 163
Ai-je bien compris ? 163

CHAPITRE 8
Les templates Django : séparation et réutilisation des rendus HTML 165
Principe des templates 166
Notre premier template 167
Dynamisons ce premier template 170
Le langage des templates 172
Les variables 172
Formatage des variables 172
Sauts conditionnels et boucles 173
Héritage et réutilisation de templates 174
Et si on avançait dans la réalisation de Trombinoscoop ? 176
Amélioration visuelle de la page de login 180
Ai-je bien compris ? 184

CHAPITRE 9

Les formulaires Django 185

Patron courant de gestion des formulaires 186
L'objet request 187
Traitement du formulaire de login 188
Ajout du formulaire dans la vue 188
Gestion du message d'erreur 189
Présentation du message d'erreur 190
La bibliothèque forms de Django 190
Création d'un formulaire avec la bibliothèque forms 191
Intégration du formulaire dans la vue et le template 192
Validation du formulaire 194
Présentation des messages d'erreur 196
Validation de l'adresse de courriel et du mot de passe 197
Faut-il se contenter d'un seul visiteur autorisé ? 199
Ai-je bien compris ? 199

CHAPITRE 10

Les modèles Django 201

Les modèles Django 202
Création d'un premier modèle 202
Le modèle Personne 203
Configuration 204
Création de la base de données et du compte administrateur (superutilisateur) 205
Création des autres modèles et de leurs liens 207
Le modèle Message : relation 1-n 207
La relation « ami » : relation n-n 208
Les modèles simples Faculté, Campus, Fonction et Cursus 209
Les modèles Employé et Étudiant : héritage 209
Le lien entre Faculté et Personne : relation 1-n 210
Regénération de la base de données 211
Utilisation des modèles 211
Création et modification d'un enregistrement 211
Récupération de plusieurs enregistrements 212
Tri des données 212
Récupération d'un enregistrement unique 212
Suppression d'enregistrements 213
Accès à des objets « liés » 213
Remplissage de la base de données 214
Configuration de l'interface d'administration des bases de données 214

Gestion de la base de données avec l'interface d'administration 216
Authentification utilisant la base de données 219
Les ModelForm 221
Création du formulaire Étudiant dans le fichier forms.py 221
Création de l'URL et de la vue de création de compte 222
Création du template de création de compte 223
Un peu de mise en forme 223
Finalisation de la page de création de compte 225
Création de deux formulaires : un pour les étudiants et un pour les employés . 225
Gestion des deux formulaires dans la vue 226
Gestion des deux formulaires dans le template 227
Un peu de dynamisme 229
Ai-je bien compris ? 230

CHAPITRE 11
Comprendre et utiliser les sessions 231
À quoi servent les sessions 232
Les sessions selon Django 233
Utilisation d'une session 234
Configuration 234
Maniement d'une variable de session 235
Enregistrement de l'utilisateur authentifié 235
Vérification que l'utilisateur est bien authentifié 236
Utilisation des données de la session 237
Que trouve-t-on dans le cookie ? 238
Que trouve-t-on dans la session ? 238
Protection des pages privées 238
Amélioration de notre page d'accueil 239
Personnalisation de la bannière 240
Division du corps de la page 243
Liste des messages 245
Récupération des messages de la liste 245
Présentation de la liste des messages 247
Liste des amis 248
Publication d'un message à destination de ses amis 250
Ai-je bien compris ? 251

CHAPITRE 12
En finir avec Trombinoscoop 253
La page d'ajout d'un ami 254

Ajout d'un formulaire dans forms.py 254
Création de la vue add_friend 255
Création du template add_friend.html 256
Ajout d'une URL dans urls.py 256
Ajout du lien dans la page d'accueil 256
La page de visualisation d'un profil 257
Création du template show_profile.html 258
Création de la vue show_profile 259
Ajout de l'URL dans urls.py 260
Amélioration de la présentation dans style.css 260
Ajout des liens dans la page d'accueil 260
La page de modification d'un profil 262
Création du template modify_profile.html 262
Création de la vue modify_profile 263
Ajout de l'URL dans urls.py 264
Ajout des liens dans la page d'accueil 264
Ai-je bien compris ? 265

CHAPITRE 13
Des sites web encore plus dynamiques avec Ajax..... 267
Exemple de l'interactivité attendue entre client et serveur 268
Validation de la création d'un compte avec Ajax 271
Ajout d'une URL pour la requête Ajax 272
Vue traitant la requête Ajax 272
Ajout du code JavaScript 274
Détecter que l'utilisateur a bien terminé de remplir le champ courriel 274
Validation du courriel saisi 276
Insérer la liste des erreurs au-dessus du champ « fautif » 277
Ajout d'un ami via Ajax 279
Ajout d'un champ texte et d'un lien 279
Ajout de l'URL et de la vue 280
Création du JavaScript d'ajout d'un ami 281
Insertion du HTML dans la page web 283
Conclusions 284
Ai-je bien compris ? 285

ANNEXE A
Installation de l'environnement de développement 287
Que faut-il installer ? 288
Python 288

Django 289
Eclipse 289
En résumé 290
Installation de Python 292
Pour Windows 292
Pour Mac OS X 295
Vérification de l'installation 296
Installation de Django 297
Pour Windows 297
Pour Mac OS X 299
Pour Ubuntu 300
Vérification de l'installation 301
Installation de Java 302
Pour Windows et Mac OS X 302
Pour Ubuntu 302
Installation d'Eclipse 303
Installation du plug-in Eclipse PyDev 305
Installation de Web Developer Tools pour Eclipse 310
Premier projet de test 311
Ai-je bien compris ? 314

Index **315**

PREMIÈRE PARTIE

Les notions essentielles

Cette première partie vise à vous donner toutes les clés pour comprendre ce qu'est un site web, comment il se construit et comment il fonctionne :

- protocole HTTP, URL, notions de web statique et dynamique, serveurs ;
- programmation orientée objet, modèle MVC, notion de framework ;
- éléments de base du langage de programmation Python ;
- structuration des pages web avec HTML5, présentation avec les feuilles de styles CSS, interactivité côté client avec JavaScript ;
- élaboration du modèle de données : cas d'utilisation (use cases), schémas de présentation (wireframes), bases de données relationnelles ;
- éléments de base du langage d'interrogation de bases de données SQL, utilisation par le biais de l'ancienne technologie CGI, discussion des limitations de cette dernière.

1

Comment fonctionne un site web ?

Ce chapitre rappelle les bases du Web, ses origines statiques et son passage graduel vers plus de dynamisme avec, notamment, l'exploitation d'une base de données côté serveur, dont les mises à jour se répercutent côté client. Sont également expliquées les notions d'URL, de protocole HTTP, de serveur web et le passage de paramètres du client vers le serveur.

SOMMAIRE

- Petite histoire du Web
- Du Web statique au Web dynamique
- URL, protocole HTTP et serveur web
- Passage de paramËtres vers le serveur

Qu'est-ce que le Web ?

C'est dans les années 1980, alors qu'il effectue un séjour de six mois comme consultant informatique au CERN (le laboratoire européen de physique des particules), que Tim Berners-Lee développe un programme appelé « Enquire ». À l'aide d'un langage adéquat, ce dernier organise le contenu et la visualisation des documents produits par le laboratoire et, grâce à la technologie hypertexte, permet également à ces pages d'en référer d'autres. Le Web, abréviation du *World Wide Web* (en français la toile [d'araignée] mondiale) était né. Quelque vingt ans plus tard, la reine Elisabeth II anoblira notre talentueux informaticien pour le remercier des nombreuses heures passées sur Facebook en compagnie de ses sujets.

Le Web est un système permettant l'organisation visuelle et la publication de documents, leur consultation via un navigateur web (par exemple Safari, Google Chrome, Firefox ou Internet Explorer.) et leur interconnexion à l'aide de liens. Trois technologies ont été inventées, ou du moins formalisées et adaptées par Tim Berners-Lee et sont à l'origine du Web :

- le concept de **page web**, au centre de tout, les pages renfermant le contenu textuel que l'on désire publier ;
- le concept de **liens hypertextes**, lesquels relient des pages web par l'insertion dans une page de liens vers d'autres. Pour mettre en œuvre ce concept, il a fallu imaginer un mécanisme d'adressage identifiant de manière unique chaque page web : c'est l'URL (*Uniform Resource Locator*), que nous étudierons plus en détail dans ce chapitre.
- un **mécanisme permettant aux navigateurs d'accéder aux pages web**. Il s'agit du protocole HTTP (*HyperText Transfer Protocol*), qui interroge un *serveur HTTP* pour en récupérer le contenu. Ces concepts seront également étudiés et détaillés par la suite.

> **CULTURE Terminologie : Internet vs Web**
>
> Très souvent sont confondus les termes *Internet* et *Web*. Il n'est pas rare d'entendre l'expression erronée « site Internet » pour désigner en réalité un « site web ».
> Internet est un réseau mondial permettant d'interconnecter des petits réseaux d'ordinateurs (d'où l'appellation « Inter Network ») par l'intermédiaire de leur serveur. C'est grâce à Internet qu'un internaute situé en Belgique peut dialoguer avec un autre internaute localisé en Australie.
> Le Web, quant à lui, n'est qu'une application d'Internet parmi d'autres comme le courriel, telnet (connexion à distance sur un ordinateur) ou le peer-to-peer (de préférence légal).
> En fait, il y a là deux réseaux superposés, mais de nature très différente : un réseau d'hyperliens « juste déposés » sur un réseau de connexions physiques. La page de votre voisin de bureau pourrait se trouver à une vingtaine de clics de votre page web ; éloignés sur le Web mais proches sur Internet ou l'inverse, les deux topologies ne se superposent pas.

Après son invention, très vite, le Web connut un succès grandissant et les premiers sites web apparurent. Au départ très basiques, ces sites étaient pour la plupart résolument « statiques ». Un site statique se caractérise par des pages dont le contenu ne change pas ou presque pas au fil du temps. L'image suivante, représentant le site web de l'Université libre de Bruxelles tel qu'il était en 1997, illustre très bien le concept de site statique.

Figure 1–1
Site web de l'ULB dans sa version du 28 juin 1997

On le voit au premier coup d'œil, la page d'accueil est très simple et le contenu ne semble pas destiné à varier au fil du temps. Revenez visiter le site dans un jour, une semaine ou un mois, il est fort probable que la page n'aura pas changé d'un iota.

Fonctionnement d'un site web statique

DÉFINITION Site web statique

Un site web statique est un site qui ne change que si son webmestre en modifie les pages à la main. Il ne possède aucun automatisme de modification de page du côté du serveur.

Imaginons qu'un internaute désire se rendre sur la page web d'accueil du site de l'Université libre de Bruxelles, dont l'adresse est www.ulb.ac.be/homepage (cette adresse est fictive, inutile de l'essayer dans votre navigateur).

Pour ce faire, sur son ordinateur, l'internaute va ouvrir un *navigateur web* (Google Chrome, Safari, Internet Explorer, Firefox ou autre) et saisir www.ulb.ac.be/homepage dans la barre d'adresse.

DÉFINITION Navigateur web

Un navigateur est un logiciel côté client qui permet à l'internaute de dialoguer avec des logiciels serveurs pour en obtenir des informations (pages web). Il affiche les pages demandées et navigue entre elles par le biais des hyperliens.

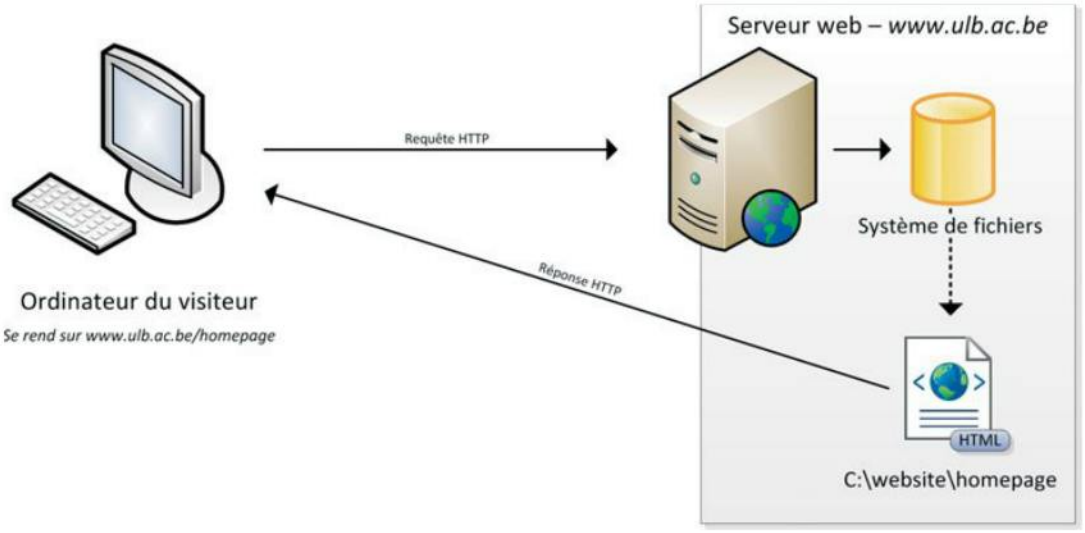


Figure 1–2 Récupération d'une page web statique

Le navigateur côté client va contacter le *serveur web* de l'université. Ce dernier est un ordinateur, pas très différent de ceux que nous utilisons tous les jours, dont le rôle est de *servir* du contenu à la demande. En pratique, les ordinateurs serveurs sont spécialement adaptés à leur fonction : plus robustes, plus compacts, ne possédant aucun gadget (webcam, haut-parleur, manette de jeu, etc.), jusqu'à être dépourvus d'écran. Cela dit, n'importe quel ordinateur peut faire office de serveur web, que ce soit votre ordinateur fixe à la maison ou votre portable.

DÉFINITION Serveur web

Un serveur web est un ordinateur robuste et sécurisé sur lequel un logiciel, également appelé serveur web (voir plus loin), est chargé de répondre aux demandes (ou *requêtes*) des internautes en leur *servant* les pages voulues ou en renvoyant un message d'erreur.

Mais où se cachent ces serveurs web ? Généralement dans des *centres de traitement des données* (*Data center* en anglais), gérés par des entreprises spécialisées et qui peuvent héberger de nombreux serveurs et sites web, souvent au bénéfice de plusieurs clients. Dans le cas de l'Université libre de Bruxelles (qui, comme beaucoup d'universités, possède plus d'espace que de moyens), le serveur web ne se trouve pas dans un centre de données externe, mais en son sein même.

Une fois la connexion établie entre le navigateur client et le serveur web, le premier va envoyer la requête suivante au second : « donne-moi la page d'accueil *homepage* s'il te plaît ». Comme il s'agit d'un site statique, le serveur web va simplement lire sur son disque dur (ou plutôt dans son système de fichiers) le fichier contenant la page web *homepage* et le renvoyer au navigateur. Pour notre exemple, nous avons imaginé un fichier nommé `homepage` se trouvant dans le dossier `C:\website`.

Une fois la page transmise par le serveur au client, le navigateur va en décoder le contenu et l'afficher à l'écran.

On le comprend aisément, à moins de changer le contenu du fichier `homepage` qui se trouve sur le disque dur côté serveur, la page web renvoyée sera toujours la même. Le seul dynamisme sera dû à l'informaticien chargé de mettre à jour les pages web stockées sur le serveur… Beau mais dur métier, même pour un professeur à la retraite ! Autant l'automatiser.

Le protocole HTTP

Afin que le navigateur et le serveur web puissent dialoguer et se comprendre, il a fallu définir ce qu'on appelle un *protocole*, qui détermine le langage et le vocabulaire utilisés pour communiquer.

DÉFINITION Protocole

Un protocole définit la signification des bits échangés entre les deux machines interlocutrices.

Le protocole utilisé pour accéder à un site web est appelé HTTP (*Hypertext Transfer Protocol*). Il définit par exemple comment doit être formatée la commande « donne-moi la page *homepage* » que le navigateur envoie au serveur de l'université lorsqu'on tape l'adresse www.ulb.ac.be/homepage. En réalité, la commande envoyée est transformée en celle-ci :

EXEMPLE 1.1 Traduction de la commande en HTTP

```
GET /homepage HTTP/1.1
Host: www.ulb.ac.be
```

Cette commande est appelée une *requête HTTP*. La réponse donnée par le serveur (dans notre cas, la page web demandée) est appelée une *réponse HTTP*. La réponse aurait pu ne pas être la page web demandée, mais, par exemple, une erreur indiquant que celle-ci n'existe pas (d'où le recours au protocole qui fera parfaitement la différence entre ces deux messages renvoyés par le serveur).

Le protocole utilisé étant HTTP, pour désigner le navigateur on parlera parfois de *client HTTP*, et pour le serveur, sans grande surprise, de *serveur HTTP*.

L'URL, adresse d'une page web

Dans notre exemple de site statique, nous avons supposé que son adresse était www.ulb.ac.be/homepage. Examinons plus en détail cette adresse et la manière dont elle est composée.

En réalité, l'adresse exacte de la page est http://www.ulb.ac.be/homepage. D'ailleurs, si vous saisissez l'adresse dans votre navigateur sans la commencer par http://, ce dernier l'ajoute automatiquement.

Cette adresse est appelée une URL (abréviation de *Uniform Resource Locator*) dont le format a été défini par l'inventeur du Web, Tim Berners-Lee (qui a reconnu, dans un article paru dans le Times en 2009, que les // ne servaient strictement à rien).

Les adresses URL que nous avons utilisées jusqu'à présent sont composées de trois éléments :

- **Le protocole** : une adresse commence toujours par un nom de protocole. Dans notre cas, il s'agit de http. Le nom du protocole est suivi de ://. D'autres protocoles existent (par exemple telnet:// pour l'utilisation d'un terminal à distance), car les adresses URL peuvent être utilisées à d'autres desseins que l'accès à un site web.
- **Un nom de domaine** : le nom de domaine, dans notre cas www.ulb.ac.be, identifie le serveur web sur lequel est hébergé le site.
- **Un emplacement pour la ressource** : dans notre cas, il s'agit simplement de homepage, soit le nom d'un fichier se trouvant dans le dossier `C:\website` sur le serveur. Le chemin vers la ressource aurait pu être plus complexe et se trouver dans un sous-dossier. Par exemple, la page d'adresse http://www.ulb.ac.be/biographies/bio_recteur identifie une ressource dont l'emplacement sur le disque dur est `C:\website\biographies\bio_recteur`.

Le serveur web : à la fois ordinateur et logiciel

Pour terminer notre présentation des sites statiques, il nous reste à clarifier la notion de serveur web. Nous avons vu que cela désigne l'ordinateur dans lequel est stocké le site web et qui se charge de renvoyer les pages demandées par les navigateurs web côté client.

En réalité, l'expression « serveur web » peut désigner deux choses qui méritent qu'on les distingue :

- L'*ordinateur physique* sur lequel est hébergé le site web. C'est ce que nous entendions jusqu'à présent par « serveur web ».
- Le *programme* qui tourne constamment en arrière-plan sur cet ordinateur, avec pour mission d'intercepter les requêtes HTTP et d'y donner suite. Il existe de nombreux logiciels de serveur HTTP sur le marché, les plus connus étant d'une part le logiciel libre *Apache* de l'Apache Software Foundation et d'autre part *Internet Information Services (IIS)* de Microsoft.

Pourquoi cette nuance ? L'ordinateur qui fait office de serveur web peut également, dans le même temps, jouer d'autres rôles et offrir d'autres services : serveur de courriels, serveur de bases de données, etc. Dans ce cas, plusieurs programmes tourneront constamment en arrière-plan, l'un pour intercepter les requêtes HTTP, l'autre pour accueillir les courriels entrants, etc.

On verra également par la suite que le même serveur web peut faire office de client HTTP. En d'autres mots, on peut très bien installer un navigateur sur le serveur web et l'utiliser pour consulter le site web hébergé sur cette même machine.

Quoi qu'il en soit, c'est pourtant bien cette configuration que nous utiliserons par la suite lorsque nous développerons notre propre site web, ceci afin d'éviter d'investir dans de multiples ordinateurs (un pour le navigateur web, un pour le serveur HTTP, un pour la base de données, etc.). Nous travaillons à l'université, ne l'oublions pas.

Des sites web qui se mettent à jour tout seuls

DÉFINITION Site web dynamique

Un site web dit dynamique comprend, du côté du serveur, des mécanismes automatiques de mise à jour des pages servies.

À la différence d'un site statique, le contenu d'un site dynamique évolue constamment. Un exemple typique est le site de Reuters (figure suivante), lequel affiche en

temps réel les cours d'actions cotées en bourse. Il vaut mieux, on le concevra aisément, que l'information sur le cours d'une action puisse se modifier au fil du temps.

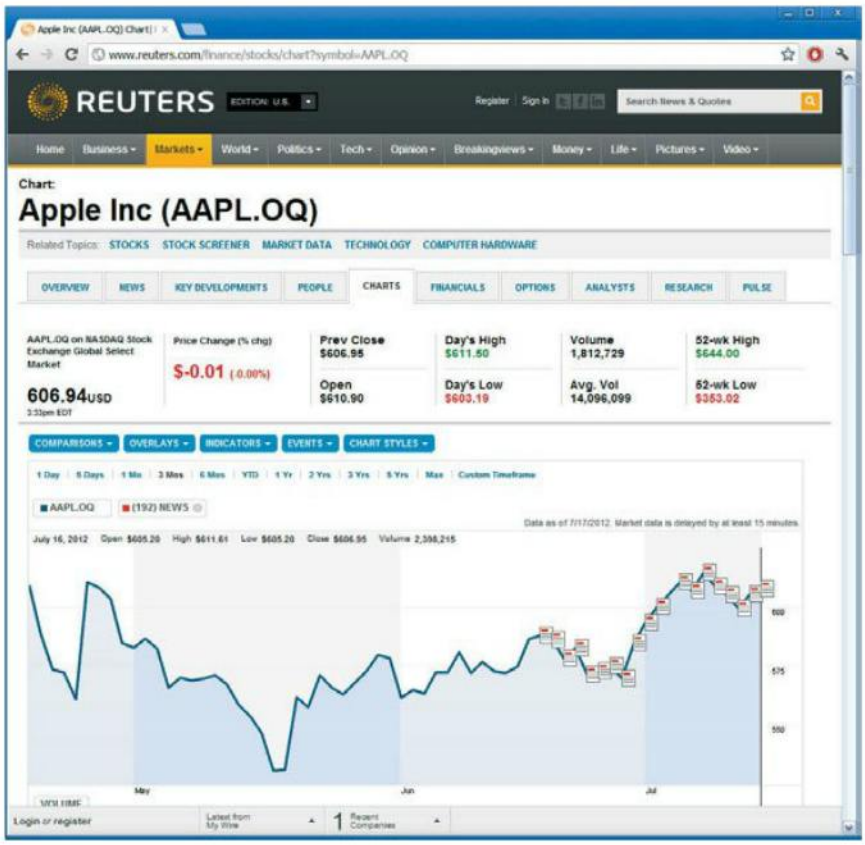


Figure 1–3
Page du site web de Reuters affichant le cours de l'action Apple

Si au début du Web les sites statiques satisfaisaient la plupart des besoins, ce n'est vraiment plus le cas aujourd'hui. Certes, il en existe encore (notamment certains sites non professionnels), mais ils ne forment pas la majorité de ceux présents sur le Web.

Certains sites, que l'on pourrait a priori croire statiques, sont de fait dynamiques ; pensons par exemple à un blog sur lequel l'auteur ne publie que très peu d'articles. En réalité, les qualificatifs « statiques » et « dynamiques » dépendent de la manière dont sont fabriquées les pages web.

DÉFINITION Page web dynamique

Une page est dite dynamique dès que son contenu est calculé *à la volée*.

Fonctionnement d'un site web dynamique

Afin de mieux comprendre ce que l'on entend par « calculé à la volée », reprenons l'exemple du site de Reuters. Imaginons qu'un internaute désire se rendre sur la page affichant le cours de l'action d'InBev (une multinationale brassicole), dont l'adresse est http://www.reuters.com/stocks/abi (cette adresse est fictive, inutile de l'essayer dans votre navigateur.).

Pour ce faire, il va ouvrir un *navigateur web* (Google Chrome, Safari, Internet Explorer, Firefox ou autre) et saisir http://www.reuters.com/stocks/abi dans la barre d'adresse.

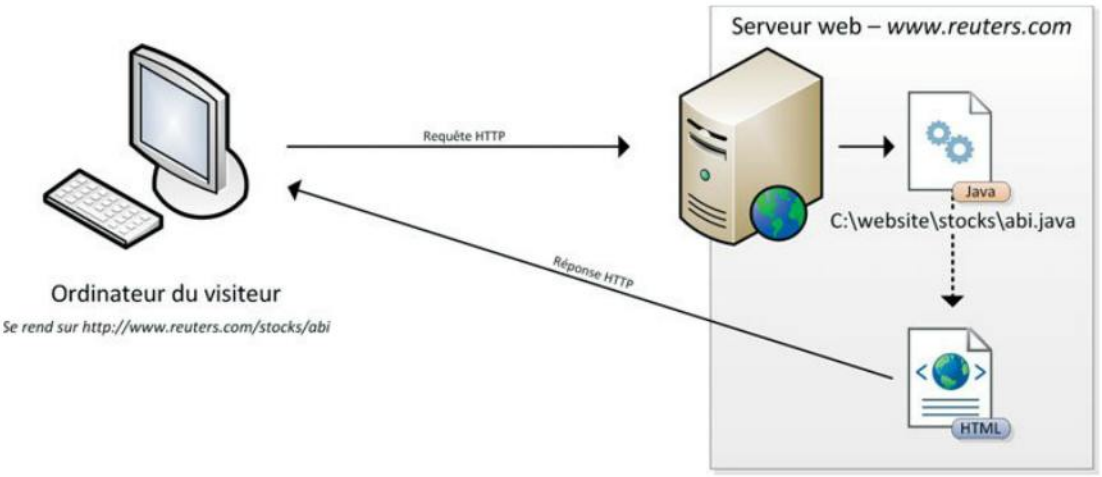


Figure 1–4 Création d'une page dynamique

Tout comme pour un site statique, le navigateur web côté client contacte le *serveur web* de Reuters. Une fois la connexion établie entre le navigateur et le serveur, le premier envoie la requête suivante au deuxième : « donne-moi la page stocks/abi ».

Toutefois, le serveur ne va pas lire sur son disque dur un fichier contenant le contenu de la page web. À la place, il lance un *programme* dont le rôle est de créer le contenu de la page web demandée.

Dans ce livre, nous utiliserons Python, mais ce programme peut être écrit en n'importe quel langage de programmation : Java, C, C#, etc. Sur l'illustration, nous avons imaginé le programme écrit en Java et se trouvant à l'emplacement suivant sur le disque dur : `C:\website\stocks\abi.java`.

Une fois la page produite par le programme, le serveur web l'envoie au client dans un format compris par le navigateur (le même format qu'une page statique). Ce dernier en décode alors le contenu et l'affiche à l'écran.

Utilisation d'une base de données

L'utilisation d'un programme pour calculer la page web offre de nombreux avantages et une souplesse bien plus grande que les pages statiques. Le premier d'entre eux est le possible accès du programme à une base de données afin d'exploiter les données qu'elle recèle. Notre exemple du site Reuters utilise une telle base de données afin d'y enregistrer le cours des actions au fil du temps. C'est elle que Reuters met à jour à la vitesse des transactions, plutôt que d'aviser personnellement tous les traders. La figure suivante illustre l'utilisation d'une base de données par le programme Java en charge de calculer la page web affichant le cours de l'action InBev.

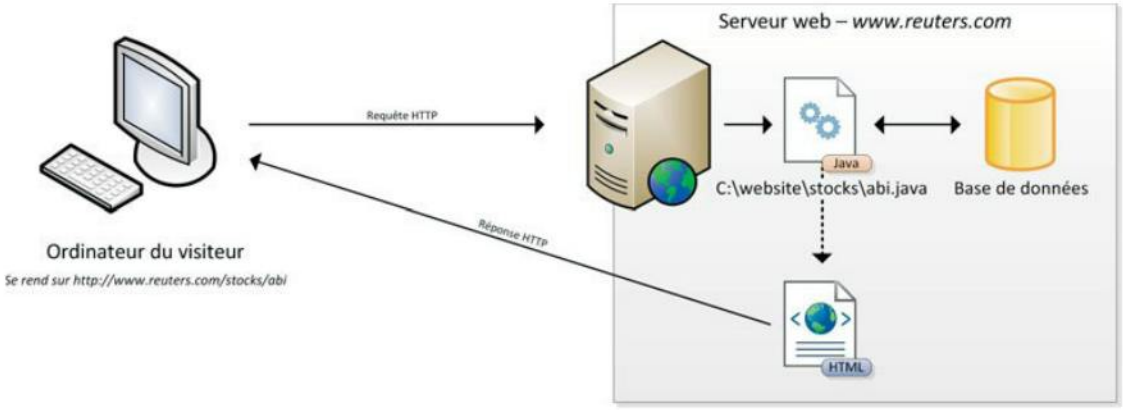


Figure 1–5 Création d'une page dynamique en utilisant une base de données

Sur ce schéma, la base de données se trouve sur le même serveur que le serveur web. Comme les serveurs web, ceux en charge des bases de données sont interrogés à l'aide d'un protocole bien déterminé.

Comme pour le serveur HTTP, le terme « serveur de bases de données » caractérise à la fois :

- l'*ordinateur physique* sur lequel se trouve hébergée la base de données ;
- le *programme* qui tourne constamment en arrière-plan dans l'ordinateur en question et qui a pour mission de répondre aux requêtes des clients désirant consulter ou modifier la base de données. Ce programme est appelé un Système de gestion de bases de données (SGBD). Il en existe de nombreux sur le marché : MySQL, Oracle Database, Microsoft SQL Serveur, PostgreSQL, DB2, etc.

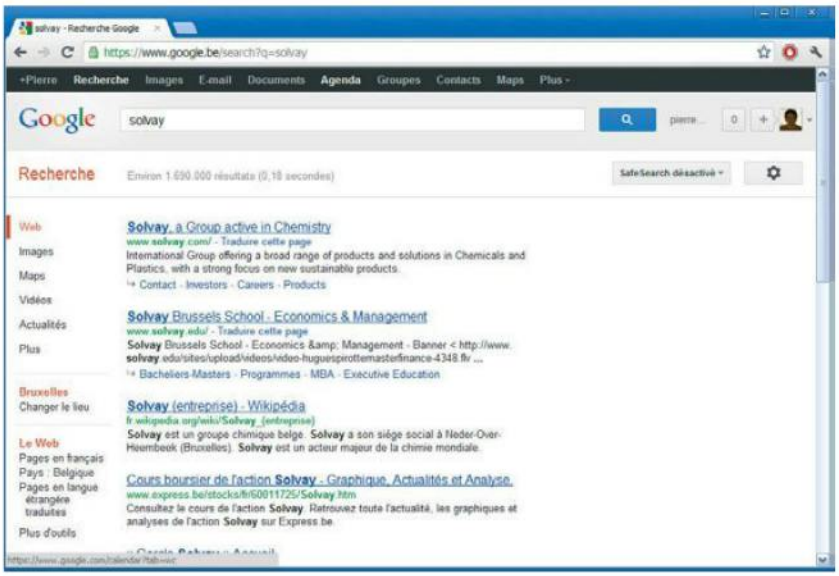
Un logiciel de gestion de bases de données peut se trouver sur une autre machine qu'un serveur de bases de données dédié. On peut aussi installer sur une même machine un serveur web et un système de gestion de bases de données. De multiples répartitions des tâches sont possibles faisant usage de multiples serveurs.

De très nombreux sites web dynamiques ont recours à une base de données.

Passage de paramètres à une page web

Un autre avantage important des pages web dynamiques est qu'il est possible de passer des *paramètres* au programme chargé de composer la page. Un exemple typique est illustré à la figure suivante : le moteur de recherche Google. Sur la page d'accueil, lorsqu'un utilisateur entre l'un ou l'autre mot-clé démarrant la recherche et qu'il clique sur le bouton *Recherche*, la page appelée est http://www.google.be/search (http://www.google.fr/search pour la France). À cette page est joint un paramètre : les mots clés ou l'expression au départ de la recherche.

Figure 1–6
Exemple de page web avec un paramètre



Comment ce paramètre est-il passé à la page ? Pour le savoir, analysons plus en détail l'URL qui est appelée. On voit en haut de la capture d'écran que l'adresse est en réalité http://www.google.be/search?q=Solvay (cette adresse n'est pas fictive, vous pouvez l'essayer dans votre navigateur).

Le paramètre est donc ajouté à la fin de l'URL. Pour séparer le nom de la page des paramètres qui suivent, on fait usage du point d'interrogation. Ensuite, pour chaque paramètre, on spécifie un nom (dans notre exemple le paramètre s'appelle q) et une

valeur (dans notre exemple on recherche l'expression Solvay). Le nom du paramètre et sa valeur sont séparés par un signe égal (nom_param=valeur_param).

Plusieurs paramètres peuvent être transmis à une page, séparés à l'aide de l'esperluette (&). Google permet, par exemple, de préciser la langue dans laquelle la page de résultat doit être affichée, à l'aide du paramètre hl. Par exemple, si on désire chercher l'expression « Solvay » et afficher la page de résultats en néerlandais, on utilisera l'URL suivante :

```
http://www.google.be/search?q=Solvay&hl=nl
```

Si l'on résume, les URL contenant des paramètres sont construites selon la forme suivante :

Figure 1–7
Construction d'une URL avec paramètres

Les paramètres ainsi passés peuvent être utilisés directement dans le programme, comme illustré à la figure suivante :

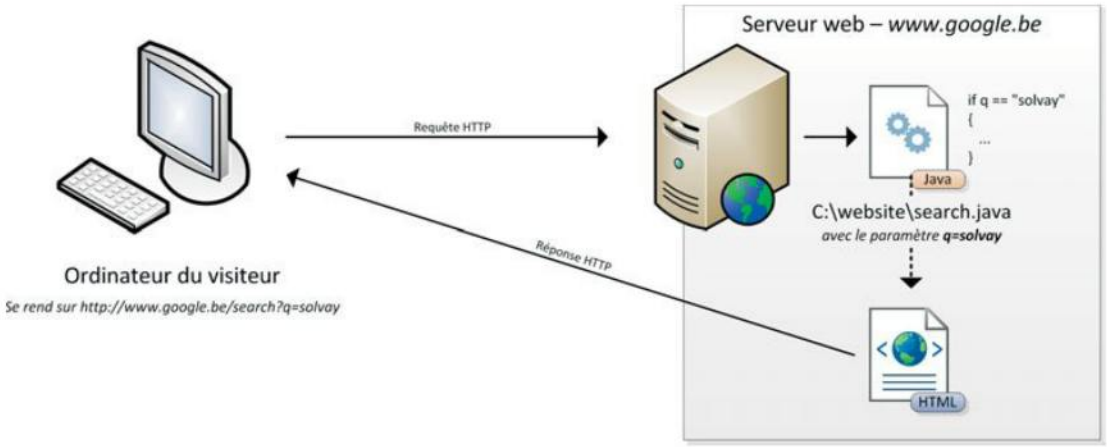


Figure 1–8 Création d'une page dynamique avec paramètres

Ai-je bien compris ?

- Quelles sont les trois technologies qui furent inventées à l'origine du Web ?
- Au niveau des traitements réalisés par un serveur web, quelle est la différence fondamentale qui distingue un site web dynamique d'un site statique ?
- Comment peut-on passer des paramètres à une page web ?

2

Programmation orientée objet et framework MVC

Ce chapitre a pour mission d'expliquer la conception logicielle respectueuse du découpage MVC et sa version sous Django. Nous exposons d'abord les principes de la programmation orientée objet. Ensuite, nous verrons comment la mise à disposition d'un framework s'inscrit dans cette logique.

SOMMAIRE

- Explication de la conception respectant le découpage MVC
- Mise en place du MVC sous Django
- Explication de la notion de framework

Des programmes modulaires

L'écriture d'un logiciel important en taille et en nombre de fonctionnalités, comptant des dizaines de milliers de lignes de code et des centaines d'heures passées devant son écran, exige sans surprise un découpage adéquat.

Une écriture simplifiée

Le premier avantage à découper le programme est d'en simplifier considérablement l'écriture, à condition que les différents modules offrent des contenus fonctionnels relativement indépendants les uns des autres et que la modification de l'un n'entraîne pas systématiquement la réécriture des autres.

Des modules réutilisables

S'ensuit assez logiquement le bénéfice de la réutilisabilité. En effet, si les modules sont relativement indépendants les uns des autres, on conçoit aisément qu'ils puissent être réutilisés dans de multiples et successives applications logicielles. Ainsi, une partie de logiciel chargée d'inscrire un internaute sur un site web quelconque peut se réutiliser d'un site à l'autre. Il en va de même pour toute forme de transaction financière ou tout paiement par carte de crédit : cela fonctionne toujours de la même façon, que vous achetiez un livre de programmation Python ou un billet pour Hawaï. De cette manière, des pans entiers d'un premier développement logiciel sont aisément récupérés dans un second si les traits pointillés qui permettent ce découpage sont tracés avec soin et si chaque découpage fonctionne de et par lui-même.

Un travail d'équipe facilité et plus efficace

Le travail en équipe s'en trouve facilité puisqu'il devient naturel de diviser et répartir les tâches entre les différents spécialistes. Comme tout bon économiste le sait depuis Adam Smith, chaque spécialiste aura dès lors l'occasion de se spécialiser davantage encore, pour un bénéfice collectif plus important – du moins en théorie. La division du travail en informatique est tout aussi avantageuse que dans les chaînes de montage.

L'informatique offre de multiples visages : informatique graphique, ergonomie de sites web, développement et programmation, stockage et exploitation des données, business intelligence... qui exigent des compétences spécifiques.

En pratique

Se prétendre juste informaticien ne veut plus dire grand-chose à l'heure actuelle ; la maîtrise de Photoshop ne présente pas beaucoup de points communs avec celle de Python, de SAP ou de la fouille de données. Parvenir à séparer au mieux les parties logicielles propres à l'une ou l'autre des différentes compétences de l'informatique n'ira certainement pas pour déplaire aux spécialistes de chacune d'entre elles.

Les langages orientés objet

Avant toute chose, les langages de programmation orientée objet (OO), dont Python, mettent leur syntaxe au service de ce découpage. Un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets logiciels collaborant par envoi de messages. Le tout s'opère en respectant un principe de distribution des responsabilités on ne peut plus simple, chaque objet, tel un microprogramme, s'occupant de gérer ses propres données. Lorsqu'un objet exige de s'informer ou de modifier les données d'un autre, il charge cet autre de s'acquitter de cette tâche. Chaque objet expose à ses interlocuteurs un mode d'emploi restreint, une carte de visite limitée aux seuls services qu'il est apte à fournir et continuera à fournir dans le temps, malgré de possibles modifications dans l'implantation concrète de ces services.

EN PRATIQUE

La programmation orientée objet (OO) est fondamentalement distribuée, modularisée et décentralisée. Et, pour autant qu'elle respecte également des principes de confinement et d'accès limité, elle favorise la stabilité des développements, en restreignant au maximum l'effet des modifications apportées au code au cours du temps.

L'orienté objet inscrit la programmation dans une démarche somme toute très classique pour affronter la complexité de quelque problème qui soit : une découpe naturelle et intuitive en des parties plus simples. A fortiori, cette découpe sera d'autant plus intuitive qu'elle s'inspire de notre manière « cognitive » de découper la réalité qui nous entoure. L'héritage, reflet fidèle de notre organisation cognitive, en est le témoignage le plus éclatant.

EN PRATIQUE Approche procédurale

L'approche procédurale qui pour l'essentiel sépare le code en un grand bloc de données et toutes les fonctions chargées de le manipuler, rend cette découpe moins naturelle, plus contraignante, « forcée ». Si de nombreux adeptes de la programmation procédurale sont en effet conscients qu'une manière incontournable de simplifier le développement d'un programme complexe est de le découper physiquement en de petits modules plus lisibles, ils souffrent de l'absence d'une prise en compte naturelle et syntaxique de cette découpe dans les langages de programmation utilisés.
Les langages orientés objet rendent au contraire le découpage du programme beaucoup plus intuitif.
Pour une introduction à la programmation orientée objet, voir :
📖 Hugues Bersini, *La programmation orientée objet*, Eyrolles, 2012

La programmation objet permet ce découpage suivant deux dimensions orthogonales :

- Horizontalement, les classes se délèguent mutuellement des services.
- Verticalement, les classes *héritent* entre elles d'attributs et de méthodes installés à différents niveaux d'une hiérarchie taxonomique.

Pour chacune de ces dimensions, reproduisant fidèlement nos mécanismes cognitifs de conceptualisation, en plus de simplifier l'écriture des codes, la récupération de ces parties dans de nouveaux contextes en est facilitée. S'en trouve également mieux garantie la robustesse de ces parties aux changements survenus dans d'autres. Un code orienté objet, idéalement, sera aussi simple à créer qu'à maintenir, réutiliser et faire évoluer.

En programmation objet, les fonctions et les données ne sont plus d'un seul tenant, mais éclatées en un ensemble de modules (dénommés « classes ») reprenant, chacun à son compte, une sous-partie de ces données et les seules méthodes qui les manipulent. Il faut apprendre à programmer en s'essayant au développement d'une succession de microprogrammes et au couplage soigné et réduit au minimum de ces microprogrammes. En découpant 1 000 lignes de code en 10 modules de 100 lignes, le gain, bien plus que linéaire, réside dans la simplicité extraordinairement accrue consistant à programmer 100 lignes plutôt que 1 000. En substance, la programmation objet pourrait reprendre à son compte ce slogan devenu très célèbre parmi les adeptes des courants altermondialistes : « agir localement, penser globalement ».

Les cas d'utilisation (use cases) et le MVC

Le principe de la séparation modèle-vue-contrôleur (MVC)

Imaginez la programmation en Python du petit menu de l'application visible sur la figure suivante. Lorsqu'on clique sur un élément du menu, une fenêtre apparaît contenant juste un bouton *Do nothing button*. Totalement inutile, certes, mais c'est pour l'exemple.

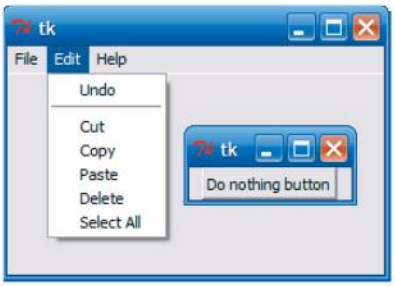


Figure 2–1
Une application avec juste un menu

Le code Python dont l'exécution affiche ce menu est repris ci-après, mais il est inutile à ce stade-ci d'en comprendre le fonctionnement en détail.

EXEMPLE 2.1 Code Python permettant d'afficher un menu à l'écran

```
from Tkinter import *

def donothing():
  filewin = Toplevel(root)
  button = Button(filewin, text="Do nothing button")
  button.pack()

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)
filemenu.add_command(label="New", command=donothing)
filemenu.add_command(label="Open", command=donothing)
filemenu.add_command(label="Save", command=donothing)
filemenu.add_command(label="Save as...", command=donothing)
filemenu.add_command(label="Close", command=donothing)

filemenu.add_separator()

filemenu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="File", menu=filemenu)
editmenu = Menu(menubar, tearoff=0)
editmenu.add_command(label="Undo", command=donothing)

editmenu.add_separator()

editmenu.add_command(label="Cut", command=donothing)
editmenu.add_command(label="Copy", command=donothing)
editmenu.add_command(label="Paste", command=donothing)
editmenu.add_command(label="Delete", command=donothing)
editmenu.add_command(label="Select All", command=donothing)

menubar.add_cascade(label="Edit", menu=editmenu)
helpmenu = Menu(menubar, tearoff=0)
helpmenu.add_command(label="Help Index", command=donothing)
helpmenu.add_command(label="About...", command=donothing)
menubar.add_cascade(label="Help", menu=helpmenu)

root.config(menu=menubar)
root.mainloop()
```

L'affichage et l'utilisation d'un tel menu exigent la prise en compte de trois fonctionnalités logicielles assez distinctes (mais pourtant en partie mélangées dans le code) :

- le **graphisme** du menu (par exemple sa couleur, le nombre d'items qui apparaissent suite à l'activation du menu, le type de caractères utilisé pour les items du menu) ;
- les **items** qui composent le menu ;
- le **fonctionnement** du menu (qu'advient-il lorsqu'on choisit un des items ?).

On conçoit aisément que, même si le code affiché ne le fait délibérément pas, il est préférable de tenir séparés ces trois aspects. Ainsi, on pourrait décider de modifier la liste des items sans pour autant qu'il soit nécessaire de modifier la représentation du menu ou l'exécution associée à certains des items. On devrait de même pouvoir aisément changer la couleur du menu sans affecter en rien son fonctionnement. On pourrait même récupérer la liste des items pour un tout autre composant graphique, un ensemble de « checkbox » par exemple.

Il en découle qu'il est plus logique et bien plus aéré que le haut du code se découpe comme suit : d'abord la liste contenant tous les items, ensuite une boucle permettant de créer les éléments graphiques associés à ces items.

EXEMPLE 2.2 Version améliorée du code précédent

```
menuItems = ["New", "Open", "Save", "Save as...", "Close"]

root = Tk()
menubar = Menu(root)
filemenu = Menu(menubar, tearoff=0)

for x in menuItems:
   filemenu.add_command(label=x, command=donothing)
```

Cet exemple, quoique modeste, montre bien la séparation des trois fonctionnalités logicielles selon la recette de conception MVC. Ici, le *modèle* serait la liste des items (que l'on pourrait installer dans un fichier séparé), la *vue* serait la représentation graphique de ce menu (type de composant graphique pour afficher la liste, disposition géométrique du menu, couleur et taille des items) et son *contrôleur* le mode de fonctionnement du menu et des items qui lui sont associés (dans le code, la déclaration séparée des fonctions `donothing()`).

Autre avantage, un développeur tiers pourrait récupérer votre formidable menu, mais pour une toute autre liste d'items. Imaginez le cas d'un restaurateur chinois qui voudrait récupérer le menu d'un collègue italien.

Le diagramme des cas d'utilisation (use cases)

Depuis l'avènement d'UML (*Unified Modeling Language*) comme langage de modélisation graphique des développements logiciels, tout projet informatique débute généralement par la mise au point du diagramme *use cases* (dit « cas d'utilisation » en français). Ce diagramme a pour mission d'identifier les différents usages de ce logiciel vu du côté utilisateur ; il compose son « cahier des charges ». La figure qui suit illustre un tel diagramme pour un distributeur automatique d'argent.

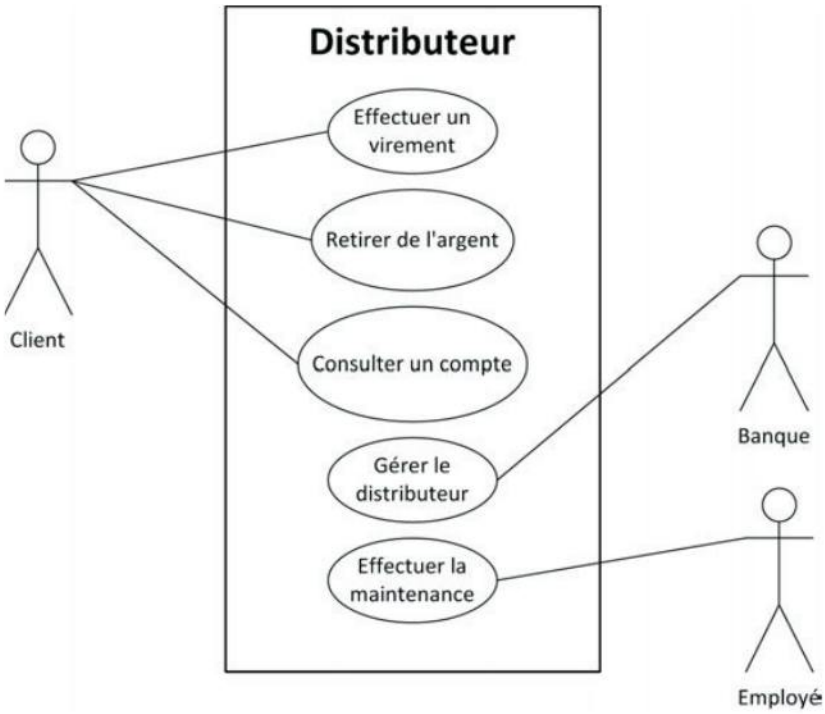


Figure 2–1
Diagramme « use case » d'un distributeur automatique d'argent

Décrivons la figure. On y voit trois types d'acteurs : `le client`, `la banque` et `l'employé de maintenance`. Chacun de ces acteurs, et c'est ce qui les différencie, en fera un ensemble d'usages (appelé les *use cases*) qui lui sont propres. Ainsi, le client pourra `retirer de l'argent` ou `effectuer un virement` mais pas `effectuer la maintenance`.

Correspondances entre MVC et cas d'utilisation

Comme l'indique la figure qui précède, les use cases vont donner naissance à trois types de fonctionnalités logicielles : *contrôles*, *entités* et *interfaces*, qui reprennent parfaitement la recette de conception MVC (il suffit d'effectuer la substitution Modèle = Entités et Vue = Interfaces).

Chaque use case sera pris en charge par les éléments contrôles. Ainsi, effectuer un virement sera la responsabilité d'un ensemble d'acteurs logiciels particuliers qui n'aura à se préoccuper ni de l'interface avec le client ni de celles avec les bases de données ou les éléments entités. Le contrôle se devrait de fonctionner de façon identique si l'on change la manière dont le client voit sa banque sur son navigateur et la manière dont les comptes en banque sont finalement stockés sur le disque dur. À chaque intersection entre les acteurs et le use case doit correspondre un ensemble d'éléments interfaces (ou vue) dont le rôle sera en effet de gérer l'interface GUI (*Graphic User Interface*) avec l'utilisateur.

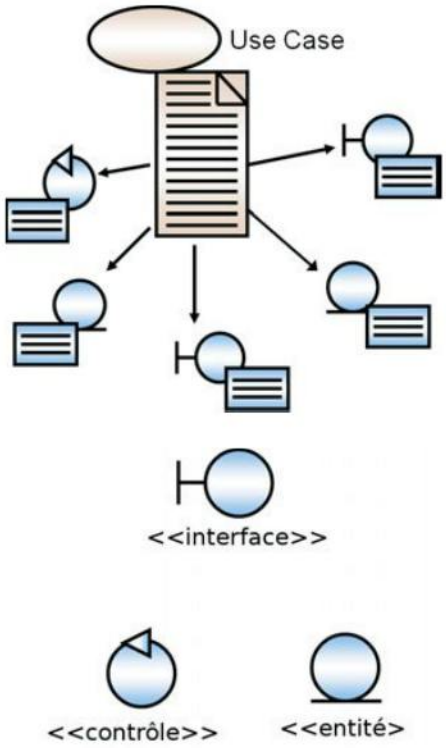


Figure 2–2
Le modèle MVC et Use Case

Enfin, les objets fondamentaux utilisés par l'application (qu'on appelle parfois objets « métier »), tels que `CompteEnBanque`, `Virement` ou `Client`, seront représentés chacun par une classe ou des éléments logiciels qui, in fine, s'associeront à des versions permanentes stockées sur disque dur (par exemple des tables `clients` et `compte en banque` dans une base de données relationnelle).

Comme la figure ci-après l'indique, les éléments contrôles demeurent au centre de tout et sont les éléments pivots de l'application. Les interfaces, reçoivent les sollicitations des utilisateurs et s'adressent aux contrôles pour relayer ces dernières. À leur tour, les contrôles parlent aux éléments entités (par exemple, ils vont chercher le statut des comptes en banque ou créer de nouveaux virements) et sollicitent en retour les éléments interfaces pour recevoir de la part des utilisateurs d'autres informations utiles à la continuation de l'interaction (combien retirer du compte, sur quel autre compte effectuer le virement…).

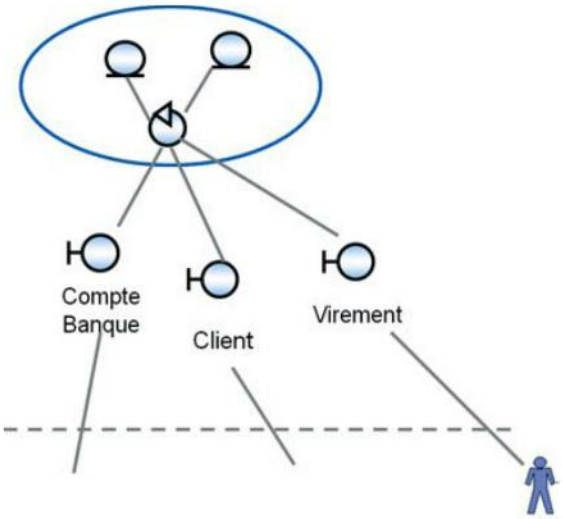


Figure 2–3
MVC et distributeur d'argent automatique

Django et le MVC

Dans Django, les trois rôles modèle-vue-contrôleur sont joués respectivement par :

- les classes `Modèles`, qui s'occuperont de la mise en correspondance avec la base de données relationnelle ;
- les `templates HTML`, qui permettront de coder en XHTML et CSS la visualisation des pages web ;
- les fonctionnalités `view` qui, par l'intermédiaire de méthodes Python, s'occuperont de toute la logique de fonctionnement de l'application.

> EN PRATIQUE **Django : ambiguïté sémantique entre contrôle et view**
>
> Attention : dans Django, le contrôle est donc plutôt joué par les `view`.

Le framework Django

Un framework est une boîte à outils logicielle, un ensemble de solutions qui sont à votre disposition pour répondre le plus simplement possible à la plupart de vos besoins.

> ANALOGIE **Une maison ouverte**
>
> La maison est à votre disposition, clé sur la porte ; il vous reste à décider du papier peint et des posters de plage ou de voiture à punaiser au mur.

Par exemple, supposez que votre application gère des livres. Afin de créer une entité `livre` avec son titre et sa date de publication dans la base de données relationnelle, il suffira d'utiliser les modèles Django et d'écrire :

EXEMPLE 2.3 Modèle Django pour créer une classe et une table livre

```
class Livre (models.Model)
  titre = models.CharField(max_length=50)
  pub_date = models.DateField()
```

La création de cette classe `Livre` exige d'hériter (l'héritage en orienté objet sera expliqué dans le chapitre suivant) de la classe `models.Model`, qui fait partie en effet du framework Django et qui se chargera de la mise en correspondance avec la table `livre` associée dans la base de données relationnelle. Ce faisant, toute la gestion de la base de données relationnelle vous est entièrement épargnée. Les amateurs de SQL, le langage d'interrogation des bases de données relationnelles, seront déçus sinon frustrés, car les bibliothèques Django s'occupent de cette partie de l'application à leur place. De même, pour obtenir les dix derniers livres parus, il vous suffit d'écrire une méthode `view` telle que :

EXEMPLE 2.4 View pour extraire les derniers livres de la base de données relationnelle

```
def dernier_livres(request)
  liste_livres = Livre.objects.order_by('-pub_date')[ :10]
  return render_to_response('dernier_livre.html',
         {'liste_livres' :liste_livres})
```

À nouveau, on se rend compte de la simplicité d'écriture de cette requête et de l'exploitation de fonctionnalités déjà pré-codées comme `objects.order_by(…)` pour ordonner les ouvrages compris dans la liste. Tout cela est en place et à votre disposition dans le framework Django. Pourquoi réinventer la roue et ne pas aller au plus vite à l'essentiel ? N'oubliez pas que Django fut conçu à l'origine pour des journalistes, pressés comme il se doit de publier leur scoop sans devoir rédiger de lourdes requêtes SQL pour remplir la base de données.

Ai-je bien compris ?

- Quelle découpe propose le modèle de développement MVC ?
- Quelles sont les similitudes et les différences entre la découpe proposée par l'approche orientée objet et la découpe MVC ?
- À quoi sert un framework ?

3

Rappels sur le langage Python

Ce chapitre a pour mission de présenter le langage de programmation Python, sur lequel repose le projet Django. Les règles syntaxiques brièvement décrites ici devraient vous suffire pour comprendre les différentes applications de Django que nous verrons par la suite, ainsi que pour maîtriser les aspects « contrôle » de votre projet.

SOMMAIRE

- Introduction au langage de programmation Python
- Les aspects procéduraux du langage
- Python et l'orienté objet

Qualités et défauts de Python

Il n'est pas question dans ce chapitre de faire de vous, lecteurs, des experts en programmation Python : le but est de vous permettre de comprendre tous les développements et utilisations de Django qui vont suivre. Le B.A.-BA de la programmation, c'est-à-dire les variables, les typages de base, les instructions d'affectation et de contrôle, le découpage des programmes par l'utilisation de fonctions et une rapide introduction aux aspects orientés objet devrait suffire.

CULTURE Python et son papa, Guido Van Rossum

Guido Van Rossum, inventeur et « superviseur » du langage Python, est hollandais d'origine, détenteur de maîtrises en mathématiques et en informatique de l'université libre d'Amsterdam. Il fait partie, à la fin des années 1980, d'un groupe de développeurs dont le but est de mettre au point un langage de programmation abordable par des non-experts, d'où son nom : « ABC ». Dès 1991, il s'attaque à Python, écrit en C et exécutable sur toutes les plates-formes. Guido Van Rossum travaille alors au CWI (*Centrum voor Wiskunde en Informatica*). En 1995, il prend la direction des États-Unis et travaille pour le CNRI (*Corporation for National Research Initiatives*) jusqu'en 2000. À cette époque, il publie *Computer Programming for Everybody*, sa profession de foi pour l'enseignement de la programmation. C'est également à cette période qu'il est nommé directeur des Python Labs de Zope Corporation, qui comprennent aujourd'hui une demi-douzaine de développeurs.
Depuis 2005, il travaille pour Google, qui semble investir beaucoup dans le langage Python. Guido y divise son temps de travail entre le projet Open Source Python et les développements de Google utilisant ce langage. Il se targue de porter le titre très ambigu de Dictateur Bénévole à Vie. Ainsi, toute proposition de modification du langage est débattue par le noyau et soumise à la communauté Python, mais la prise en compte de celle-ci dans le langage reste la prérogative de Guido, qui conserve le dernier mot (d'où son titre : gentil et à l'écoute… mais dictateur tout de même, ces modifications n'étant pas soumises à un vote majoritaire).

Python en est aujourd'hui à sa troisième version, mais nous abordons ici la version 2.7, la seule que Django prenne en charge à ce jour. Néanmoins, la présentation que nous en faisons ne devrait pas souffrir de l'éventuelle migration que vous seriez amenés à faire lorsque Django s'adaptera aux nouvelles versions de Python.

Qualité : la simplicité

En tant que langage de programmation, Python présente énormément de qualités liées pour l'essentiel à sa facilité d'accès, de téléchargement et de mise en œuvre, à la clarté de sa syntaxe, à son côté « open source », au fait qu'il soit interprété plutôt que compilé puis exécuté, rendant possible de voir directement les résultats des instructions s'afficher sans étape préalable de compilation. Python a visé dès son origine une grande simplicité d'écriture, tout en conservant tous les mécanismes de programmation objet de haut niveau. Il cherche à soulager au maximum le programmeur de problèmes syn-

taxiques non essentiels aux fonctionnalités clés du programme. Les informaticiens disent souvent de lui qu'il est un excellent langage de prototypage, à remplacer par un langage plus « solide », plus « robuste » (Java, .Net ou C++) lorsqu'on arrive aux termes de l'application. Autre avantage certain, tout ce qui est nécessaire à l'utilisation du langage et à sa compréhension se trouve réuni sur un site unique.

CULTURE Communauté : un site unique

La communauté Python reste très structurée autour d'un seul site web.

▸ http://www.python.org

C'est un atout considérable, surtout lors de l'apprentissage et de la découverte du langage : toutes les sources restent accessibles et disponibles. Monsieur-tout-le-monde peut participer à l'évolution du produit, mais la prise en charge officielle des évolutions reste sous la responsabilité d'un seul. À quelques subtilités près, Python est dans le prolongement de l'aventure Open Source, dont le représentant le plus emblématique reste Linux. Il pourrait devenir le langage de programmation phare de l'Open Source, tout comme Linux est celui du système d'exploitation. Il semble que les éditeurs informatiques, tant Sun que Microsoft, aient décidé d'évoluer pour leur propre bébé, vers ce même modèle de développement. Java est en effet devenu Open Source et le projet Mono poursuit en Open Source l'aventure .Net.

Pour un retour aux sources du logiciel libre et de l'open source :

📖 *Richard Stallman et la révolution du logiciel libre - Une biographie autorisée*, Eyrolles 2010

Par sa simplicité d'usage, Python s'impose assez naturellement lorsqu'on démarre la programmation. De nombreuses universités le préfèrent aujourd'hui à Java et C++. Sa popularité n'a cessé de croître et de nombreuses solutions logicielles dans le monde Linux ou Microsoft y ont recours. Un Python.Net est même sur le point d'éclore.

CULTURE D'où vient le nom « Python » ?

Python ne doit rien au reptile. Il se réfère aux humoristes anglais que sont les Monty Python, lesquels ont révolutionné dans les années 1970 et 1980 à la fois l'humour et la télévision ; d'où la nécessité de les remplacer par un jumeau ouvert sur le Web et la programmation !

Défaut : la simplicité !

Revers de la médaille, la simplicité de sa syntaxe se retourne contre Python ; l'absence de typage explicite et de compilation deviennent des défauts. Pour les critiques, il est difficile d'aborder des projets complexes dans un idiome aussi primitif. Peut-on écrire Hamlet en Esperanto ? C'est le Python se mangeant la queue !

Tout langage de programmation est toujours à la recherche d'un Graal au carrefour de la simplicité d'usage et d'écriture, de l'efficience machine et de la robustesse. Au vu du nombre et de la diversité des programmeurs, on peut douter qu'il soit possible de le trouver un jour. Il est plutôt difficile de rationaliser ses préférences, et il en va –

presque ! – des guerres de langages comme de celles de religions. Nous ne nous positionnerons pas dans ce débat, Python nous étant de toute façon imposé par Django, comme Java est imposé par la technologie JSP.

EN PRATIQUE Simplicité ou puissance ?

Python est très souvent plébiscité pour la simplicité de son fonctionnement, ce qui en fait un langage de choix pour l'apprentissage. La programmation est, de fait, très interactive. Vous tapez 1+1 à l'écran et 2 apparaît comme par magie. On arrive au résultat escompté par des écritures plus simples et plus intuitives, donc plus rapidement.

Nous restons pourtant un peu sceptiques quant à l'extension de cette même simplicité à la totalité de la syntaxe. Python est un langage puissant car il est à la fois orienté objet et procédural. Pour l'essentiel, il n'a rien à envier à des langages comme Java ou C++ et exige donc de maîtriser, comme pour ceux-ci, toutes les bases logiques de la programmation afin de parvenir à des codes d'une certaine sophistication.

Ce langage de programmation veut préserver la simplicité qui caractérise les langages de script interprétés plutôt que compilés :

- exécutions traduites dans un bytecode intermédiaire et s'exécutant au fur et à mesure qu'elles sont rencontrées ;
- disponible pour toutes les plates-formes ;
- pas de typage statique ;
- structures de données très simples d'emploi (listes, dictionnaires et chaînes de caractères).

Il souhaite en outre préserver toutes les fonctionnalités qui caractérisent les langages puissants (orienté objet, ramasse-miettes, héritage multiple et bibliothèques d'utilitaires très fournies, comme nous le verrons dans les chapitres qui suivent). Une telle hybridation, pour autant qu'elle soit réussie, en ferait un langage de tout premier choix pour le prototypage de projet, quitte à revenir par la suite à des langages plus robustes pour la phase finale du projet (comme C++ et Java, avec lesquels, d'ailleurs, Python se couple parfaitement : il peut hériter ou spécialiser des classes Java ou C++).

Les bases : variables et mémoire centrale

Déclaration et initialisation des variables

Pendant l'exécution d'un programme, les données que ce dernier manipule sont stockées en mémoire centrale. Les variables permettent de manipuler ces données sans avoir à se préoccuper de l'adresse explicite qu'elles occuperont effectivement en mémoire. Pour cela, il suffit de leur choisir un nom. Bien entendu, ceci n'est possible que parce qu'il existe un programme de traduction (l'interpréteur) du programme qui, à l'insu du programmeur mais pour son plus grand confort, s'occupe d'attribuer une adresse à chaque variable et de superviser tous les éventuels futurs changements d'adresse.

DÉFINITION Variable

Une variable est un nom servant à repérer l'emplacement précis de la mémoire centrale (l'adresse de cet emplacement) où le programme stocke une donnée qu'il manipule.
Une variable peut être assimilée à une boîte aux lettres portant un nom. Par exemple, `a=5` signifie : « à l'adresse mémoire référencée par `a` – autrement dit, dans la boîte aux lettres dénommée `a` –, se trouve la valeur `5` ». Nous utilisons plus fréquemment l'abus de langage « la variable `a` vaut `5` ».

Étant donné la liberté autorisée dans le choix du nom des variables, il est préférable, pour de simples raisons mnémotechniques mais également pour améliorer la lisibilité des programmes, de choisir des noms de variables en fonction de ce qu'elles représentent réellement. Préférerez des dénominations parlantes telles que `couleur`, `taille` ou `prix` à `x`, `y` ou `z`.

En Python, la simple déclaration d'une variable ne suffit pas à la créer. Après avoir choisi son nom, il est nécessaire de lui affecter une valeur initiale.

EXEMPLE 3.1 Déclaration et initialisation de variables

```
monChiffre=5
petitePhrase= " Quoi de neuf ? "
pi=3.14159
```

Type des variables

Comme le montre l'exemple précédent, différents types d'information (nombres, chaînes de caractères…) peuvent être placés dans une variable et il est capital de connaître comment la valeur qui s'y trouve a été codée. Cette distinction correspond à la notion de *type*, fondamentale en informatique car elle débouche in fine sur le codage binaire de la variable (par exemple, un entier numérique sur 32 bits et un caractère sur 16 bits) ainsi que sur les usages qui peuvent en être faits (on ne peut pas multiplier un caractère par un nombre). Ainsi :

- Une variable recevant une valeur numérique entière est du type `int`.
- Une variable recevant un réel sera du type `float`.
- Une variable recevant au moins un caractère sera du type `string`.

Il s'agit là des trois types dits simples, qui nous occuperont pour l'essentiel dans la suite.

Modification et transtypage des variables

L'instruction d'affectation réalise donc la création et la mémorisation de la variable, l'attribution d'un type à la variable, la création et la mémorisation d'une valeur et

finalement l'établissement d'un lien entre le nom de la variable et l'emplacement mémoire. Ainsi, les deux instructions suivantes :

EXEMPLE 3.2 Modifier une variable

```
maVariable = 5
maVariable = 2.567
```

modifient simplement le contenu de la « boîte aux lettres » `maVariable`. À la première ligne, on indique qu'à l'adresse `maVariable` est stockée la valeur `5` et, à la seconde, on remplace cette valeur par `2,567`. Après exécution des deux lignes, la valeur sauvegardée en mémoire à l'adresse `maVariable` est donc `2,567`.

Notons au passage que le type de `maVariable` a changé : il est passé de `int` à `float`. Cette spécificité est une facilité essentielle de plus permise par le langage Python. À la différence d'autres langages de programmation plus contraignants, Python est dit *typé dynamiquement* et non *statiquement*.

DÉFINITION Typage dynamique et typage statique

Avec le typage dynamique, il n'est plus nécessaire de préciser à l'avance le type des informations exploitées dans le programme, celui-ci devenant « implicite ». Python devine le type par la valeur que nous installons dans la « boîte aux lettres » – et ce type peut changer au fil des valeurs que la boîte va contenir. Le choix entre typage statique et dynamique est de ces controverses qui ont amusé, distrait, même déchiré, la communauté informatique depuis la nuit des temps (les années 1960 en informatique).

Copie de variables

Nous pouvons également prendre le contenu de `maVariable` et le copier dans `taVariable` par la simple instruction :

Syntaxe

```
taVariable=maVariable
```

Le contenu de `maVariable` reste inchangé car il s'agit de recopier une valeur et non pas d'un transport physique à proprement parler (ici la métaphore de la boîte aux lettres montre ses limites). En fait, l'information est dupliquée d'un lieu de la mémoire vers un autre.

Se renseigner sur une variable

Si l'on comprend que la valeur affectée à une variable peut évoluer dans le temps, on comprend tout autant qu'il est possible de réaliser toutes sortes d'opérations sur ces variables, autorisées par leur type. Nous détaillerons pratiquement certaines de ces opérations par la suite. Nous allons voir qu'outre l'abstraction offerte par les variables, le

langage nous offre toute une bibliothèque d'utilitaires. Citons d'ores et déjà deux fonctions prédéfinies : `type (nomDeLaVariable)` qui affiche le type de la variable mise entre parenthèses et s'avère très utile pour vérifier ce qui y a été installé jusqu'ici, et `print nomDeLaVariable` qui affiche à l'écran la valeur stockée à l'adresse désignée par la variable `nomDeLaVariable`.

Quelques opérations sur les types simples

Le type int

Le code suivant illustre des opérations réalisées dans la console interactive de Python, sur des variables de type entier (`int`).

SYNTAXE Commentaires

Les lignes débutant par # sont des commentaires en Python – donc non exécutés – et seront utilisées ici pour éclairer le comportement des instructions.

Les `>>>` que vous verrez apparaître par la suite sont produits par l'environnement de développement et sont une « invite » à taper vos instructions Python. Les informations retournées par Python sont présentées en italique.

EXEMPLE 3.3 Opérations sur des variables de type int

```
>>> maVariable = 3
>>> print maVariable
3
>>> type(maVariable)
<type 'int'>
>>> taVariable=4
>>> maVariable=taVariable
>>> print maVariable
4
>>> maVariable = saVariable 3
#Une erreur apparaît car saVariable n'existe pas
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    maVariable = saVariable
NameError: name 'saVariable' is not defined
>>> maVariable = maVariable + 3 1
>>> maVariable #Dans la console interactive de Python, le seul appel de
#la variable suffit à donner sa valeur sans qu'il soit nécessaire
#d'utiliser print explicitement.
```

```
7
>>> maVariable = maVariable/2 (5)
>>> maVariable
3
#Python ne conserve que la valeur entière au vu du type de maVariable
>>> maVariable + 3 = 5 (4)
SyntaxError: can't assign to operator
#On ne peut pas écrire n'importe quoi et il faut respecter un minimum de
#syntaxe python.
>>> maVariable+=3 (2)#Raccourci d'écriture !!!
>>> maVariable
6
>>>
```

Outre l'affectation, les opérations mathématiques sur les valeurs contenues dans les variables font partie des premières instructions élémentaires comprises par tout processeur. Ainsi, `maVariable=taVariable+1` signifie que la valeur contenue dans `maVariable` va prendre celle contenue dans `taVariable` augmentée de 1 (ici, on est plutôt en présence de deux classes d'instructions élémentaires : une addition et un déplacement de valeur d'une variable à l'autre). Nous pouvons aussi constater que l'expression `maVariable=maVariable+3` signifie que la nouvelle valeur de `maVariable` est égale à l'ancienne augmentée de 3 (1). On peut d'ailleurs en raccourcir l'écriture : `maVariable+=3` (2).

Nous pouvons d'ores et déjà répertorier deux formes d'erreurs. La première consiste à effectuer des opérations sur une variable qui n'a pas été initialisée au préalable (3). Dans l'exemple, on ne peut en effet assigner la valeur de `saVariable` à `maVariable` car aucune valeur n'a préalablement été assignée à `saVariable`. La ligne (4), quant à elle, montre que la liberté d'écriture n'est pas totale : il y a certaines règles de syntaxe et conventions à respecter et nous en découvrirons d'autres par la suite.

EN PRATIQUE Compilé/interprété : détection des erreurs de syntaxe

Pour les langages de programmation de type « compilés », c'est le compilateur qui s'occupe en général de repérer ces erreurs de syntaxe, alors que, dans Python (« interprété »), tout se passe à l'exécution. De nombreux informaticiens jugent cela un peu tard pour s'apercevoir d'une erreur, d'où la préférence qu'ils expriment pour les autres langages requérant l'étape de compilation (et le typage statique que cette compilation requiert). C'est en cela qu'ils jugent ces concurrents plus robustes : de nombreuses « idioties » sont détectées dès la compilation, vous évitant l'humiliation en public d'un plantage lors de l'exécution de votre programme.

Le type float

Le code suivant illustre des opérations élémentaires sur des réels.

EXEMPLE 3.4 Opérations sur des variables de type float

```
>>> taVariable=3.0
>>> maVariable=taVariable/2 (2)
>>> maVariable
1.5
>>> type(maVariable)
<type 'float'>
>>> print maVariable
1.5
>>> print mavariable (1)

Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    print mavariable
NameError: name 'mavariable' is not defined
```

Côté erreur, on peut ajouter que Python est sensible à la casse, c'est-à-dire différencie majuscules et minuscules (1) : `maVariable` et `mavariable` sont deux références différentes. On peut également remarquer qu'une opération de division non entière sur un entier naturel donne lieu à une troncature du résultat (ligne (5) dans l'exemple de la section précédente), alors que la même opération réalisée sur un `float` (un réel) donne lieu au résultat correct (2). Retenons donc que `3` est `int` alors que `3.0` est `float`.

Le type string

Pour en terminer avec ces types simples, le code suivant illustre des exemples concernant des variables de type `string` (des chaînes de caractères).

EXEMPLE 3.5 Opérations sur des variables de type string

```
>>> phrase1="le framework"
>>> phrase2=" Django"
>>> phrase1+=phrase2 (1)
>>> phrase1
'le framework Django'
>>> phrase2*=3 (2)
>>> phrase2
' Django Django Django'
>>> print phrase2[1] (3)#On saute le caractère blanc du début de ligne
D
>>> phrase2 += 5 (4)

Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    phrase2 += 5
```

```
TypeError: cannot concatenate 'str' and 'int' objects
>>> phrase2="phrase1" (5)
>>> phrase2
'phrase1'
```

On constate ici que des opérations sur les littéraux sont possibles : le `+` en présence de deux `string` sert à les concaténer (1), alors que le `*` sert à répéter un littéral (2). Remarquons également que la chaîne de caractères se présente comme une collection de caractères indexés, le premier élément de la chaîne étant repéré par la valeur 0, le second la valeur 1 (3), etc.

EN PRATIQUE Indexation des collections : partir de 0 et non de 1

Dans la plupart des langages de programmation, le premier élément d'une collection est toujours indexé par 0. Le dernier se trouve donc indexé par la longueur de la collection moins un. Commencer un tableau à « un » plutôt qu'à « zéro » est une source d'erreur classique en programmation.

Enfin, remarquons qu'il est interdit de concaténer deux éléments de types différents, par exemple un `string` et un nombre naturel (4). À nouveau, l'erreur se produira à l'exécution vu l'absence de typage statique et d'une étape préalable de compilation.

Une chaîne de caractères se distingue par la présence des apostrophes. En l'absence de celles-ci, la signification de l'expression peut être toute autre. Dans la ligne (1), c'est la variable `phrase1` (sans apostrophes) qui est prise en compte, alors que dans la ligne (5), c'est la chaîne `« phrase1 »` (entre apostrophes doubles) qui est affectée à la variable `phrase2`.

SYNTAXE Python et les apostrophes simples et doubles

Contrairement à d'autres langages, Python ne fait pas de différence entre les apostrophes simples et les doubles.

Les types composites : listes et dictionnaires

Nous avons jusqu'ici vu des données de types simples à savoir les `int`, les `float` et les `string`. Il existe également des types composites, qui regroupent en leur sein plusieurs entités de types simples.

Nous en avons déjà vu un cas particulier. En effet, les chaînes de caractères sont des collections de caractères, chacun de type `string`. Ici, toutes les entités sont du même type.

Les autres données composites qui nous intéresseront par la suite sont les *listes* et les *dictionnaires*. L'utilité de ces collections est de répéter un même traitement sur chacun des éléments composant la collection. Par exemple, si l'on souhaite mettre toutes les lettres d'un mot en majuscules, il serait stupide de réécrire cette opération autant de fois qu'il y a de lettres dans le mot ; autant ne l'écrire qu'une fois et placer cette opération dans une boucle qui balaiera tout le mot. Collections et *instructions de boucle* vont souvent de pair.

Les listes

Nous avons vu que, dans une chaîne de caractères, chaque élément est indexé par une valeur entière numérotée à partir de 0. Les listes ne sont qu'une généralisation de ce principe aux autres types : entiers, réels, caractères, chaînes. Le code suivant clarifie leur utilisation.

EXEMPLE 3.6 Opérations sur des listes

```
>>> liste=["journal",9,2.7134,"pi"] (1)
>>> liste[0]
'journal'
>>> print liste[3]
pi
>>> type(liste[1])
<type 'int'>
>>> liste[4] (2)

Traceback (most recent call last):
  File "<pyshell#59>", line 1, in <module>
    liste[4]
IndexError: list index out of range
#Erreur car on a dépassé la capacité de la liste !!!
>>> liste.append("bazar") (3)
>>> liste
['journal', 9, 2.7134, 'pi', 'bazar']
>>> liste.insert(4,"truc") (4)
>>> liste
['journal', 9, 2.7134, 'pi', 'truc', 'bazar']
>>> len(liste) (5) #donne la longueur de la liste
6
```

On voit clairement (1) qu'une liste est une série d'entités de types simples, chacune étant accessible par l'intermédiaire de son index : 0,1,2… Attention à ne pas dépasser la longueur initiale de la liste (2), déterminée lors de son initialisation. Pour ajouter un élément en fin de liste (3), il suffit d'utiliser l'instruction `append`, alors que pour insérer un nouvel élément dans la liste en une position précise, il faut utiliser l'instruction `insert` (4). L'instruction `len` retourne quant à elle le nombre d'éléments de la liste (5).

Les indexations des éléments de la liste peuvent également s'effectuer en se référant à l'exemple suivant :

EXEMPLE 3.7 Opérations sur des listes

```
>>> monTexte = "hello" + " world"
>>> monTexte[2]
'l'
>>> monTexte[0:2] #a:b signifie « de a inclus à b exclu »
'he'
>>> monTexte[:4]
'hell'
>>> monTexte[6:]
'world'
>>> monTexte[-1]
'd'
```

Les dictionnaires

Les dictionnaires, derniers types composites, sont quant à eux une généralisation des listes. Chaque élément est indexé non plus par sa position dans la liste, mais par un élément choisi parmi les trois types simples. Il s'agit donc d'une sorte de matrice 2 × n d'éléments simples dont ceux de la première colonne réalisent l'index ou la clé (`key`) d'accès au second.

L'exemple traité dans le code clarifiera les idées.

EXEMPLE 3.8 Opérations sur des dictionnaires

```
>>> dico={"computer":"ordinateur"}
>>> dico["mouse"]="souris" (1)
>>> dico["mouse"]
'souris'
>>> dico[2]="two" (1)
>>> dico
{2: 'two', 'computer': 'ordinateur', 'mouse': 'souris'}
>>> dico.keys()
[2, 'computer', 'mouse']
>>> del dico[2]
>>> dico
{'computer': 'ordinateur', 'mouse': 'souris'}
```

Pareillement aux listes, les dictionnaires sont un type dynamique : on peut ajouter *dynamiquement*, c'est-à-dire même après une première initialisation, des éléments au dictionnaire (1). Ainsi, après la création de celui-ci, nous l'avons enrichi de `souris` et `two` indexés respectivement par `mouse` et `2`. Les éléments de la première colonne ser-

vent à accéder à leur pendant dans la seconde. L'ordre d'apparition des éléments n'a plus aucune importance dans les dictionnaires car l'indexation de chaque élément ne se fait plus par sa position.

Les instructions de contrôle

Jusqu'ici, les exemples ont montré comment l'interpréteur Python parcourait le code de manière séquentielle. Ainsi, lorsqu'une séquence d'instructions est lue, l'interpréteur les exécute les unes à la suite des autres dans l'ordre où elles sont écrites et où il les découvre. Cette exécution systématique présente des limitations dans de nombreux cas. Des instructions de rupture de séquence apparaissent donc nécessaires. Certaines instructions élémentaires des processeurs ont la charge de modifier la séquence naturelle d'exécution du programme en bifurquant vers une nouvelle instruction qui n'est plus la suivante. En substance, il existe deux types d'instructions qui permettent de rompre avec la séquence en cours : la *forme conditionnelle* et la *forme répétitive*.

Les instructions conditionnelles

Ce type d'instruction permet au code de suivre différents chemins suivant les circonstances. Il s'agit, en quelque sorte, d'un aiguillage dans le programme.

Aiguillage à deux directions : instruction if (...) else

L'instruction conditionnelle `if (...) else` teste une condition et, en fonction du résultat, décide du chemin à suivre dans le programme. La syntaxe est la suivante :

SYNTAXE. Instruction if (...) else

```
if condition:(4)
  bloc1 (1)
else: (4)
  bloc2 (2)
bloc3 (3)
```

L'évaluation de la condition renvoie un résultat « vrai » ou « faux » ; on dit de cette expression qu'elle possède une *valeur booléenne*. Le `bloc1` (1) est un ensemble d'instructions qui ne seront exécutées que si la condition est vérifiée. Si la condition est fausse, c'est le `bloc2` (2) qui sera appliqué. Dans un cas comme dans l'autre, le programme continuera en exécutant le `bloc3` (3).

Remarquons la présence essentielle des deux-points (4) et du retrait devant `bloc1` (1) et `bloc2` (2). Les deux-points marquent la fin de la condition. Toutes les instructions

à exécuter « comme un seul bloc » doivent impérativement être identiquement décalées légèrement vers la droite. En l'absence de cette indentation, l'interpréteur retournerait une erreur.

DÉFINITION Indentation

On dit du code décalé vers la droite qu'il est *indenté*.
En Python, l'indentation est fondamentale. Elle fait intégralement partie de la syntaxe du langage. Ce qui était une bonne pratique d'écriture de code, ayant pour finalité d'en aérer la présentation (et généralement suivie par la majorité des programmeurs, tous langages confondus), s'est trouvée transformée en une règle d'écriture stricte.

Selon l'indentation choisie pour une instruction quelconque, le résultat à l'exécution s'avère très différent, comme l'illustre l'exemple de code suivant. En (1), l'instruction `print "c'est faux"` est indentée comme `a=6` ; elle fait donc partie du bloc conditionné par `a>10` et ne s'exécute pas puisque la condition n'est pas vérifiée. En (2), l'instruction `print "c'est faux"` est indentée comme `print "c'est vrai"` ; elle fait donc partie du bloc conditionné par `a<10` et s'exécute.

EXEMPLE 3.9 Indentation dans l'instruction if

```
>>> a=5
>>> if a<10:
      print "c'est vrai"
  if a>10:
    a=6
    print "c'est faux" (1)

c'est vrai
>>> if a<10:
  print "c'est vrai"
  if a>10:
    a=6
  print "c'est faux" (2)

c'est vrai
c'est faux
```

Toutes les instructions à exécuter lorsque la condition n'est pas satisfaite doivent être regroupées sous le mot-clé `else`.

DÉFINITION Else

Else signifie « autre » en anglais ; `else` représente tous les cas non couverts par la condition du `if`.

EXEMPLE 3.10 Instruction if (...) else

```
>>> a=10
>>> if a<10:
  print "a inferieur a 10"
else:
  print "a superieur ou egal a 10"

a superieur ou egal a 10
```

Notez, là encore, la présence des deux-points terminant le `else` et l'indentation. En théorie, et en l'absence des invites `>>>`, le `else` se trouve comme il se doit en-dessous du `if`.

Aiguillage à plus de deux directions : instruction if (...) elif (...) else

Imaginons maintenant un aiguillage à plus de deux directions. Les conditions doivent être disjointes. En effet, l'ordinateur ne s'accommodant pas d'incertitude, son comportement face à plusieurs conditions validées en même temps serait très ambigu. Il y a dans l'exemple qui suit deux situations particulières, correspondant aux conditions 1 et 2, et un bloc `else` reprenant toutes les autres situations.

SYNTAXE. Aiguillage à plus de deux directions

```
if condition1:
  bloc1
elif condition2:
  bloc2
else:
  bloc3
```

DÉFINITION elif

`elif` est une concaténation de `else` et `if` et pourrait se traduire par « sinon si ».

On peut définir autant de situations disjointes qu'on le souhaite en ajoutant des blocs `elif`.

SYNTAXE. Aiguillage à plus de trois directions

```
if condition1:
  bloc1
elif condition2:
  bloc2
elif condition3:
  bloc3
else:
  bloc4
```

Un des avantages du `elif` est de pouvoir aligner toutes les conditions disjointes les unes en-dessous des autres, sans être contraint de décaler de plus en plus vers la droite à chaque condition (la largeur du papier n'y suffirait pas).

Les boucles

Ce type d'instruction permet au programme de répéter, de compter ou d'accumuler, avec une économie d'écriture considérable.

La boucle while

La syntaxe de cette instruction est :

SYNTAXE. Boucle while

```
while condition:
  bloc
```

Comme pour la condition `if`, les deux-points (qui terminent la condition) et l'indentation du bloc à exécuter sont indispensables. Tant que la condition est vérifiée, c'est l'ensemble du bloc, c'est-à-dire la suite d'instructions indentées de la même manière, qui va être exécuté. Dès que la condition cesse d'être vérifiée, le bloc est ignoré et le programme passe à la suite. On peut d'ores et déjà voir apparaître deux problèmes :

- l'*initialisation des variables* composant la condition avant de procéder à l'évaluation ;
- le *risque d'une boucle infinie* en présence d'une condition toujours vérifiée.

Il faut donc veiller à faire évoluer dans le bloc la valeur d'au moins une des variables intervenant dans la condition, ce qui permettra à la boucle de s'interrompre.

La boucle est un mode de programmation évitant de réécrire un bloc d'instructions autant de fois qu'on souhaite l'exécuter.

EXEMPLE 3.11 Boucle while

```
compteur=1 (1)
while compteur <= 2: (2)
  bloc1
  compteur =compteur + 1
bloc2
```

Lors de l'exécution du code qui précède, le `bloc1` va être répété deux fois : en effet, au démarrage, le compteur vaut 1 (1). La condition (2) étant vérifiée, on va exécuter le `bloc1` et incrémenter le compteur : la valeur de ce dernier est donc 2 à présent. À la fin de l'exécution du bloc, la condition du `while` est à nouveau testée. Comme elle est toujours vérifiée, le `bloc1` va s'exécuter encore une fois et le compteur passera à 3. À

ce stade-ci, la condition n'est plus validée et les instructions subordonnées au `while` seront ignorées. Le `bloc2` sera exécuté et le code ira de l'avant.

Évidemment, même si la répétition est ici limitée (seulement deux fois), on imagine aisément l'économie d'écriture dans le cas où nous souhaiterions exécuter 350 fois le `bloc1`. Remarquons que si le compteur n'était pas incrémenté de 1 à chaque tour, la condition serait toujours vérifiée et le `bloc1` s'exécuterait à l'infini.

EXEMPLE 3.12 Boucle while

```
>>> compteur = 0
>>> while compteur < 4:
  print compteur
      compteur += 1

0
1
2
3
```

La boucle for (...) in

L'instruction `for (...) in` est une autre forme de boucle, qui permet cette fois d'itérer sur une collection de données, telle une liste ou un dictionnaire.

Le petit exemple qui suit l'illustre dans le cas d'une liste. L'avantage est double : il n'est pas nécessaire de connaître la taille de la collection et la variable `x` incluse dans le `for (...) in` prend successivement la valeur de chaque élément de la collection.

EXEMPLE 3.13 Boucle for (...) in

```
>>> liste=["Bob","Jean","Pierre","Alain","Yves"]
>>> for x in liste:
  print (x)

Bob
Jean
Pierre
Alain
Yves
```

Muni de ces différents éléments, il est aisé de comprendre le code suivant, impliquant un dictionnaire cette fois. Soit une collection d'étudiants, chacun ayant obtenu une note à un examen. Le programme qui suit, dans lequel de nombreuses lignes ne sont pas montrées, établit la moyenne des notes ainsi que le nombre d'étudiants ayant raté l'examen. Sachez juste que `for x,y in listeEtudiant.items ()` permet de boucler à la fois sur la clé (`x`) et sur les éléments indexés par celle-ci (`y`).

EXEMPLE 3.14 Boucle for

```
>>> Moyenne = 0
>>> #On crée un dictionnaire des notes indexé par les étudiants
>>> listeEtudiant ["Pierre"] = 10
>>> ……..
>>> NombreEtudiants=0
>>> NombreRates=0
>>> for x,y in listeEtudiant.items():
  NombreEtudiants+=1
  Moyenne+=y
  if y < 10:
    NombreRates += 1

>>> Moyenne/=NombreEtudiants
>>> Moyenne
10.5
>>> NombreRates
3
```

Le petit code qui suit est inspiré d'un « jeu à boire » comme bien des étudiants en ont connu, notamment à l'Université Libre de Bruxelles, où on l'appelle le « ding ding bottle ». Ce jeu simple consiste à énumérer les nombres à tour de rôle et, si le nombre est un multiple de 5 ou de 7, le remplacer respectivement par « ding ding » ou « bottle ». (On imagine que si chaque erreur est punie par l'ingurgitation cul sec d'un verre, les erreurs s'enchaînent de plus en plus vite.) Voici une version Python d'une sobriété sans égale de ce petit jeu décapant.

EXEMPLE 3.15 Ding ding bottle en Python

```
>>> while compteur < 20:
  if (compteur % 5 == 0) and (compteur % 7 == 0):
    print ("ding ding bottle")
  elif (compteur % 5 == 0):
    print ("ding ding")
  elif (compteur % 7 == 0):
    print ("bottle")
  else:
    print (compteur)
  compteur += 1

ding ding bottle
1
2
3
4
ding ding
```

```
6
bottle
8
9
ding ding
11
12
13
bottle
ding ding
16
17
18
19
```

Les fonctions

Dans Python comme dans tous les langages de programmation, il est possible de découper le programme en blocs fonctionnels, appelés « fonctions » ou « procédures », qui peuvent être appelés partout dans le code. Ces fonctions rendent le programme plus modulaire et plus clair en évitant des répétitions de blocs d'instructions parfois longs. Le comportement et le résultat des fonctions dépendront des arguments reçus en entrée. L'exemple simple illustré ci-après est la définition de la fonction `cube(x)` qui renvoie le cube de l'argument `x`. Après avoir défini la fonction, on l'appelle 4 fois à l'intérieur d'une boucle.

EXEMPLE 3.16 Fonction

```
>>> def cube(x):
  return x*x*x (1)

>>> compteur = 1
>>> while compteur < 5:
  cube(compteur)
  compteur+=1

1
8
27
64
```

L'exécution de la fonction se termine en renvoyant le contenu du `return` (1) au code appelant. On pourra bien sûr appeler la fonction partout où cela s'avère nécessaire.

L'appel de la fonction à l'intérieur d'un code interrompt la séquence en cours (afin d'exécuter les instructions de la fonction). Ce que le code faisait jusque-là est sauvegardé dans une zone mémoire dédiée, de sorte qu'une fois la fonction achevée, le code puisse reprendre son cours là où il l'avait laissé.

Comme autre exemple, la fonction `inverse` décrite ci-après inverse une chaîne de caractères passée comme argument. Elle est ensuite exécutée sur tous les éléments d'une liste.

EXEMPLE 3.17 Exemple de fonction

```
>>> def inverse(x):
  i=""
  n=0
  for a in x:
    n+=1
  compteur = n
  while compteur > 0:
    i+=x[compteur-1]
    compteur-=1
  return i

>>> inverse("Bersini")
'inisreB'

>>> liste=["Jean","Paul","Jacques"]
>>> n=0
>>> for x in liste:
  liste[n]=inverse(x)
  n+=1

>>> liste
['naeJ', 'luaP', 'seuqcaJ']
```

Les variables locales et les variables globales

Lorsqu'on passe un argument dans une fonction, il est important de comprendre qu'il s'agit là d'une *variable locale* (à la différence de *globale*) qui n'aura d'existence que le temps d'exécution de la fonction. Lorsque cette dernière est terminée, toutes les variables passées en arguments disparaissent de la zone de mémoire locale associée à la seule fonction. Le petit programme qui suit devrait vous permettre de mieux saisir ce principe.

EXEMPLE 3.18 Exemple de fonction

```
>>> def test(x): 2
  x+=1
  print (x)
```

```
>>> x=5 (1)
>>> test(x)
6
>>> x
5
```

Alors que la variable x prend la valeur 5 au départ de l'exécution de la fonction (1), le fait qu'elle soit incrémentée de 1 à l'intérieur de la fonction ne modifie en rien sa valeur originale, car cette incrémentation n'affecte qu'une copie locale de la variable (2) qui disparaîtra de la mémoire aussitôt l'exécution de la fonction terminée. La variable originale, elle, sera préservée dans une zone mémoire dédiée et sera récupérée intacte à l'issue de l'appel de la fonction. Bien que les deux variables portent le même nom, elles ne sont en rien confondues par le programme : en (1), x est une variable globale (valable pour le programme principal), alors qu'en (2), c'est une variable locale (limitée à la fonction).

La programmation objet

Les classes et les objets

En programmation objet, on répartit les instructions dans des *classes* représentant les acteurs de la réalité que le programme se doit d'affronter. Ainsi, une application bancaire reprendra comme classes « les clients », « les comptes en banque », « les opérations », « les emprunts », etc. Chaque classe est à la source d'un ensemble d'instances appelées *objets* et qui ne se différencient entre elles que par la valeur prise par leurs attributs.

Une classe se définit par ses *attributs*, qui décrivent ce qu'elle *est*, et ses *méthodes*, qui décrivent ce qu'elle *fait*. Chaque objet possède à un moment donné pour ses attributs des valeurs qui lui sont propres et sur lesquels les méthodes agissent. Les méthodes sont déclarées exactement comme des fonctions mais elles ne portent que sur les attributs de la classe. L'état de l'objet évolue dans le temps au fur et à mesure de l'exécution des méthodes sur cet objet. Décortiquons le code suivant.

EXEMPLE 3.19 Programmation orientée objet en Python

```
class Client:
#Le constructeur définit et initialise les attributs de l'objet
  def __init__(self,nomInit,prenomInit,adresseInit,ageInit):
    self.__nom=nomInit #par la présence des « __ » les attributs
                       #deviennent privés.
    self.__prenom=prenomInit
```

```
        self.__age=ageInit
        self.__adresse=adresseInit

    def donneNom(self):
        return self.__nom

    def donneAdresse(self):
        return self.__adresse

    def demenage(self,newAdresse): 6
        self.__adresse=newAdresse

    def donneAge(self):
        return "age : " + str(self.__age)

    def vieillit(self): 7
        self.__age+=1

    def __str__(self): 4
        return "client : " + self.__prenom + " " + self.__nom


#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20) 3

#creation d'un deuxieme client
client2 = Client("Lenders","Pascale","rue Juliette",30) 3

#exécution des méthodes sur les objets clients
#d'abord la méthode __str__ appelée de manière implicite.
print (client1) 5
print (client2)
client1.__nom="Dupont" 2
#on ne peut pas accéder à l'attribut privé __nom
print (client1)
print (client1.donneAdresse())
#la seule manière de modifier l'adresse
client1.demenage("rue Juliette")
print (client1.donneAdresse())
#la seule manière de modifier l'âge
client2.vieillit()
print (client2.donneAge())
```

Résultat de l'exécution

```
>>>
client: Hugues Bersini
client: Pascale Lenders
client: Hugues Bersini
rue Louise
rue Juliette
age: 31
>>>
```

La classe `Client` possède quatre attributs *privés* (1), c'est-à-dire inaccessibles de l'extérieur de la classe. C'est la présence de __, précédant le nom de l'attribut, qui leur donne ce caractère « privé ». Ainsi, lorsque plus bas dans le code on essaie de modifier le nom de `client1` (2), le changement ne s'effectue pas, où plutôt s'effectue sur une autre variable créée à la volée, mais pas sur l'attribut qui nous intéresse ici. L'orienté objet, pour des raisons de stabilisation de code, favorise l'*encapsulation* de la plupart des caractéristiques de la classe (les attributs comme la plupart des méthodes). Tout ce qui est encapsulé ou privé peut se trouver modifié sans affecter en rien les autres parties du code qui accèderont à la classe client (accès restreint à ses seules parties publiques) ; c'est pourquoi l'encapsulation est garante d'une stabilité dans le temps.

C'est le *constructeur* `__init__` qui se charge d'initialiser les attributs privés. Il est appelé lors de la création de l'objet. Dans le code, deux objets `client1` et `client2` sont ainsi créés (3).

La méthode `__str__` est *héritée* de la super-classe `object` et est *redéfinie* dans la classe `Client` (4) (nous expliquerons l'héritage et la « redéfinition » plus tard) ; elle est appelée de manière implicite à chaque fois que l'on affiche l'objet (5) (en fait à chaque fois que l'on veut associer une chaîne de caractère à l'objet).

L'adresse et l'âge étant deux attributs privés, la seule manière d'y accéder est d'appeler les méthodes publiques (aucun __ ne précède leur déclaration). Ainsi, seule la méthode `demenage` (6) pourra modifier l'adresse et la méthode `vieillit` (7) l'âge du client.

> **DÉFINITION Mot-clé self**
>
> Le mot-clé `self` est indispensable dès qu'il s'agit de faire référence à l'objet lui-même. De fait, toutes les méthodes portant sur l'**objet** se doivent de recevoir ce dernier, c'est-à-dire `self`, en paramètre.

L'association entre classes

La pratique de l'orienté objet consiste d'abord et avant tout en l'éclatement de l'application logicielle entre les classes constitutives du projet. Ainsi, si dans un logi-

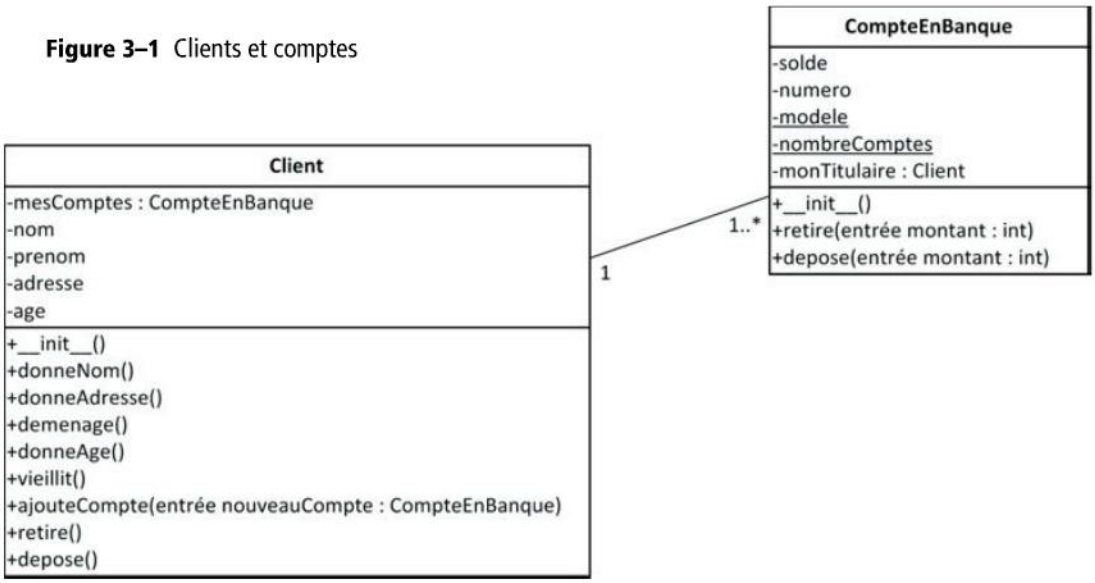
ciel bancaire, le client veut retirer 1 000 euros de son compte en banque, c'est l'objet `client` qui sollicitera la méthode `retrait(1000)` exécutée sur le compte en question. Ce déclenchement de la méthode `retrait` sur le compte, provoqué par le client, est appelé dans le jargon objet un « envoi de message ». Tous les ingrédients d'un message s'y retrouvent : l'expéditeur (objet `client`), le destinataire (objet `compte en banque`) et le contenu du message (appel à la méthode `retrait(1000)`).

Dans le diagramme de classes UML représenté ci-après, chaque objet `client` est associé à un ensemble d'objets `compte en banque` et, en retour, chaque objet `compte en banque` est associé à un et un seul `client`. Cette association « 1-n » est bidirectionnelle. Le code qui suit réalise cette association entre les deux classes. Son exécution se limite à créer un client et un compte en banque, puis à permettre au client de déposer un certain montant sur son compte. Pour ce faire, le client va envoyer le message `depose(1000)` sur le compte qu'il aura sélectionné. Notez également l'apparition d'attributs *statiques* dans la classe `compte en banque` : le modèle utile au numéro de compte et le nombre de comptes créés (qui seront tout deux utilisés pour attribuer un numéro de compte au client).

DÉFINITION Attributs statiques (ou variables de classe)

Les attributs statiques sont associés à la **classe** plutôt qu'aux objets (ils sont soulignés dans le diagramme de classe). Ils n'ont besoin que de la classe pour exister et être utilisés. Ils le sont d'ailleurs en faisant toujours allusion à leur classe (comme dans `CompteEnBanque.nombreComptes`). Leur utilisation ne nécessite donc pas de recourir au mot-clé `self`. Tout ce qui est statique peut exister sans objet.

Figure 3–1 Clients et comptes



EXEMPLE 3.20 Implémentation du diagramme en Python

```
class CompteEnBanque:
    __modele="210-" #attribut statique
    __nombreComptes=0 #attribut statique

    def __init__ (self):
        self.__solde=0
        self.__titulaire=""
        self.__numero=CompteEnBanque.__modele + \ #a la ligne
            str(CompteEnBanque.__nombreComptes)
        CompteEnBanque.__nombreComptes+=1

    def assigneTitulaire(self,titulaire):
        self.__monTitulaire = titulaire

    def donneNumero(self):
        return self.__numero

    def retire(self,montant):
        self.__solde-=montant

    def depose(self,montant):
        self.__solde+=montant

    def __str____(self):
        return "le solde du compte : " + self.__numero + \
               " est : " + str(self.__solde)



class Client:

    #Le constructeur initialise les attributs de l'objet
    def __init____(self,nomInit,prenomInit,adresseInit,ageInit):
        self.__nom=nomInit
        self.__prenom=prenomInit
        self.__age=ageInit
        self.__adresse=adresseInit
            self.__mesComptes=[]

    def donneNom(self):
        return self.__nom
    def donneAdresse(self):
        return self.__adresse

    def demenage(self,newAdresse):
        self.__adresse=newAdresse
```

```
    def donneAge(self):
        return "age: " + str(self.__age)

    def vieillit(self):
        self.__age+=1

    def __str__(self):
        return "client : " + self.__prenom + \
            " " + self.__nom

    #cette methode ajoute un compte dans la liste du client
    #elle s'assure egalement que le compte ait bien son client

    def ajouteCompte(self,compte):
        self.__mesComptes.append(compte)
        compte.assigneTitulaire(self)

    def identifieCompte(self):
        print ("Sur quel compte ?")
        x=input("?") #cette instruction permet de lire un string
                     #a l'ecran en affichant un "?"
        for c in self.__mesComptes:
            if c.donneNumero() == x:
                res=c
                break
        return c

    def retire(self):
        c=self.identifieCompte()
        print ("quel montant ?")
        x=int(input("?"))
        c.retire(x)

    def depose(self):
        c=self.identifieCompte()
        print ("quel montant ?")
        x=int(input("?"))
        #le string lu a l'ecran devra etre modifie en int
        c.depose(x)

#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20)
#creation d'un premier compte
compte1= CompteEnBanque()
print (compte1)
#ajout du compte dans la liste des comptes du client
client1.ajouteCompte(compte1)
```

```
#le client depose de l'argent sur son compte
client1.depose()
print (compte1)
```

Résultat de l'exécution

```
>>>
le solde du compte : 210-0 est : 0
Sur quel compte ?
?210-0
quel montant ?
?1000
le solde du compte : 210-0 est : 1000
>>>
```

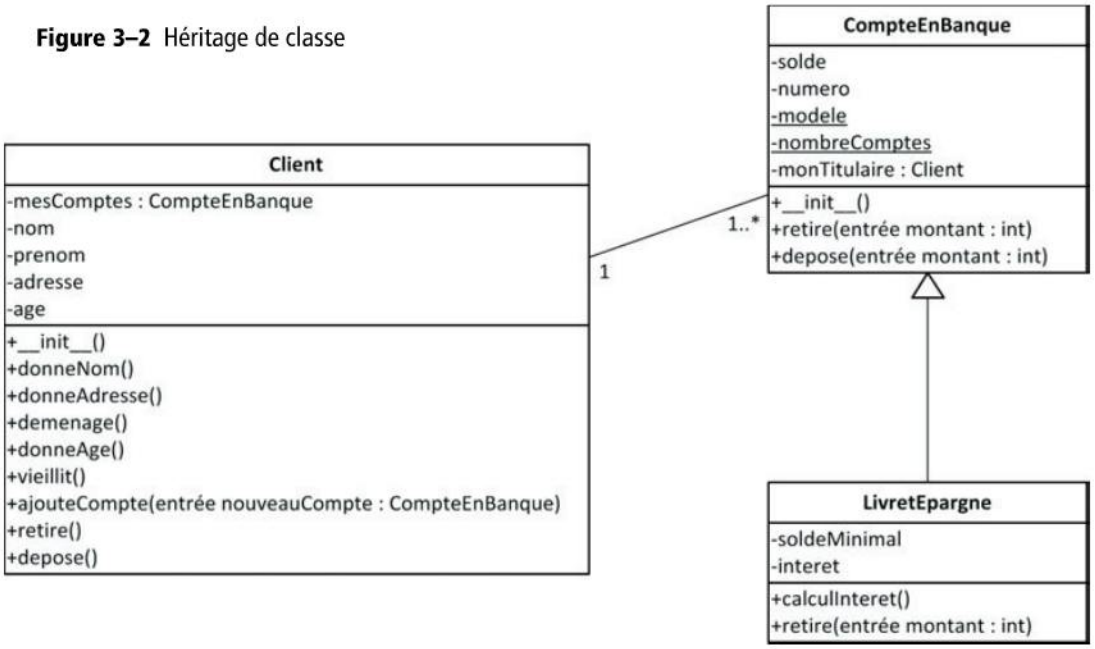
Héritage et polymorphisme

En orienté objet, la deuxième manière de découper l'application en classes est de recourir au mécanisme d'*héritage* qui permet de considérer les classes à différents niveaux de généralité. On parlera ici de classes parents (ou super-classes) pour les plus génériques et de classes filles (ou sous-classes) pour celles qui en héritent. Les classes s'installent ainsi dans une taxonomie, des plus génériques aux plus spécifiques. Les classes filles héritent des attributs et des méthodes de leurs parents et peuvent se distinguer de trois manières :

- en ajoutant de **nouveaux attributs** qui leur seront propres,
- en ajoutant de **nouvelles méthodes** qui leur seront propres,
- en redéfinissant des méthodes déjà présentes dans les classes parents. La **redéfinition** signifie qu'on reprend exactement la même signature de la méthode (nom et argument), mais qu'on lui associe un code différent.

Dans le code et le diagramme de classes qui suivent, la classe `LivretEpargne` hérite de la classe `CompteEnBanque`, via les parenthèses. Elle ne possède en propre que deux attributs statiques : le `soldeMinimal` (1), en deçà duquel le client ne peut débiter son solde, et un taux intérêt (2). Elle ajoute une méthode particulière pour calculer son intérêt (3) et elle redéfinit la méthode `retire` (4) en prenant en considération la contrainte du solde minimal. Si nécessaire, elle peut rappeler la méthode prévue dans la classe mère (en la référant par `CompteEnBanque.retire()`). Par ailleurs, le solde étant privé dans la classe mère, il n'est accessible que par l'entremise de la méthode `donneSolde()` (5). Ici, il n'est pas nécessaire de lever une ambiguïté sur l'appel en faisant explicitement allusion à la classe mère puisque la méthode n'est pas redéfinie.

Figure 3–2 Héritage de classe



EXEMPLE 3.21 Implémentation du diagramme en Python

```
class LivretEpargne(CompteEnBanque):
    __soldeMinimal=0 (1) #attribut statique
    __interet = 0.1 (2) #attribut statique

    def calculInteret(self): (3)
        depose(donneSolde()*__interet)

    def retire(self,montant): (4)
        if (self.donneSolde() (5) - montant) >= \
        LivretEpargne.__soldeMinimal:
            CompteEnBanque.retire(montant)
            #appel de la methode de la classe mère
        else:
            print ("pas assez d'argent sur le compte")
```

Dans les instructions qui suivent, on voit que le client dépose d'abord 1 000 € sur son compte puis souhaite en retirer 2 000. Comme le compte en question s'avère un livret d'épargne et non un compte en banque quelconque, c'est bien la version redéfinie de la méthode `retire` du livret d'épargne qui sera appelée.

EXEMPLE 3.22 Utilisation des classes définies

```
#creation d'un premier client
client1 = Client("Bersini","Hugues","rue Louise",20)

#creation d'un premier compte
compte1= LivretEpargne()
print (compte1)
#ajout du compte dans la liste des comptes du client
client1.ajouteCompte(compte1)
#le client depose de l'argent sur son compte
client1.depose()
print (compte1)
#le client retire de l'argent du compte
client1.retire()
```

Résultat de l'exécution

```
>>>
le solde du compte : 210-0 est : 0
Sur quel compte ?
?210-0
quel montant ?
?1000
le solde du compte : 210-0 est : 1000
Sur quel compte ?
?210-0
quel montant ?
?2000
pas assez d'argent sur le compte
>>>
```

DÉFINITION Polymorphisme

L'appel de la méthode redéfinie en fonction de la nature de la sous-classe sur laquelle cette méthode s'exerce est appelé *polymorphisme*.

On conçoit aisément l'avantage du polymorphisme en présence d'un ensemble d'objets tous issus de classes filles différentes et toutes ayant leur propre implantation de la méthode `retire`. Envoyer le même message sur tous ces objets aura des effets différents selon le type de classe fille dont est issu l'objet. Cela permettra à la classe expéditrice d'envoyer un même message à plusieurs sous-classes de la classe `CompteEnBanque` sans se préoccuper plus que nécessaire de leur nature ultime. Chacune exécutera le message à sa manière et c'est tout ce qu'on lui demande.

Nous avons déjà rencontré un exemple de redéfinition et de polymorphisme avec la redéfinition de la méthode `__str__` qui associe un `string` à un objet. Dans le même ordre d'idée, plus loin dans le livre, Django fera usage de la redéfinition de la méthode `__unicode__`, qui permet également une représentation des objets sous forme de caractère.

Import et from : accès aux bibliothèques Python

Comme dans tous les langages, le programmeur Python fait un très large usage de fonctionnalités déjà codées et installées dans des modules couramment appelés bibliothèques (*library* en anglais). Toutefois, avant de pouvoir les utiliser, il faut évidemment les localiser et donc indiquer à l'exécution où elles se cachent (ou, plus précisément, quel nom elles portent).

En Python, cela se fait par l'entremise des instructions `import` (1) (on écrira alors, séparés par un point, le nom de la bibliothèque suivi de celui de la fonction à utiliser (2)) et/ou `from` (3). L'exemple qui suit illustre ces instructions.

EXEMPLE 3.23 Instructions import et from

```
>>> #Je souhaite obtenir un nombre aléatoire entre 1 et 10
>>> print (random.randint(1,10))

Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print (random.randint(1,10))
NameError: name 'random' is not defined
>>> import random (1) # j'importe la bibliotheque random
#un nombre au hasard entre 1 et 10
>>> print (random.randint(1,10)) (2)
7 #Le résultat sera peut-être différent pour vous !

>>> liste=["a","b","c","d","e","f"]
>>> print liste
['a', 'b', 'c', 'd', 'e', 'f']
#J'utilise une autre fonction du module random
>>> random.shuffle(liste) (2)
>>> liste
['c', 'b', 'e', 'a', 'f', 'd']
```

```
#Je veux avoir l'heure à ce moment précis
>>> clock()
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    clock()
NameError: name 'clock' is not defined
#j'importe la seule fonctionnalité qui me le permet: clock()
>>> from time import clock (3)
>>> clock()
6.634921477450347e-06
#clock() me donne l'heure dans un format particulier.
```

Ai-je bien compris ?

- Comment peut-on changer le type d'une variable ?
- Quelle est la différence entre une liste et un dictionnaire ?
- Que permet l'héritage entre classes ? Qu'est-ce que le polymorphisme ?

4

Rappels sur HTML5, CSS et JavaScript

Ce chapitre va s'intéresser aux langages de présentation des pages web, HTML5 et CSS, ainsi qu'à la manière de rendre les pages plus interactives et plus dynamiques côté client, par l'utilisation du JavaScript. Ces trois technologies sont devenues incontournables, quelle que soit la plate-forme de développement web choisie.

SOMMAIRE

- Synthèse rapide des technologies HTML5, CSS et JavaScript
- HTML5, CSS, JavaScript : pièces maîtresses du Web communes à tous les environnements de développement web

L'objectif de ce chapitre est de présenter brièvement HTML, CSS et JavaScript, pas d'en étudier tous les aspects. Le but est de vous donner les bases élémentaires pour réaliser vos premiers sites. Si le besoin s'en fait sentir, n'hésitez pas à approfondir ces trois sujets par le biais d'autres livres ou sources d'informations. De nombreux ouvrages abordent ces technologies à différents degrés de profondeur.

À LIRE HTML, CSS et JavaScript

- Jean Engels, *HTML5 et CSS3*, Eyrolles, 2012
- Rodolphe Rimelé, *HTML5*, Eyrolles, 2011
- Raphaël Goetter, *CSS avancées*, Eyrolles, 2012
- Christophe Porteneuve, *Bien développer pour le Web 2.0*, Eyrolles, 2008
- Thierry Templier et Arnaud Gougeon, *JavaScript pour le Web 2.0*, Eyrolles, 2007 (ebook)

Créer un site web, qu'il soit statique ou dynamique, c'est avant tout créer des *pages web*. Ces dernières contiennent l'information que l'on désire partager et afficher. Dans ce chapitre, nous allons étudier les langages et technologies qui permettent de les construire.

Le premier de ces langages est HTML (*HyperText Markup Language*) ; il décrit *le contenu et la structure* d'une page web. Nous nous intéresserons ensuite aux CSS (*Cascaded Style Sheets*), qui décrivent *l'aspect* d'une page web (couleurs, polices, position des éléments, etc.). Nous terminerons par un aperçu de JavaScript, qui ajoute de l'interactivité et du dynamisme aux pages web.

On retrouve à un autre niveau la fameuse trinité modèle-contrôle-apparence évoquée au chapitre 2. Ici, c'est HTML qui s'occupe du modèle et de l'information à présenter, les CSS de l'aspect et JavaScript du contrôle. La recette gagnante est de séparer au mieux ces trois aspects du site web, de les confier à des spécialistes différents et de jouer au mieux la division du travail.

Ces trois langages sont exécutés et interprétés *côté client* par le navigateur web, *a contrario* de Python ou Java qui sont des langages exécutés *côté serveur*. Reprenons le schéma vu au chapitre 1 pour mieux comprendre cette distinction.

Sur ce schéma, un internaute désire consulter le site de Reuters. Le serveur web de Reuters est contacté par le navigateur de l'internaute. Java (il pourrait s'agir de Python, ASP.Net ou PHP) entre alors en action pour créer la page à renvoyer ; plus précisément, Java écrit le code HTML (accompagné de ses CSS et JavaScript) à renvoyer au navigateur. Le code Java est donc exécuté *côté serveur*. Lorsque la page web parvient au navigateur, ce dernier en interprète le HTML, les CSS et le JavaScript, afin d'en construire son rendu graphique et de l'afficher à l'écran. Ces trois langages sont donc exécutés *côté client* par le navigateur.

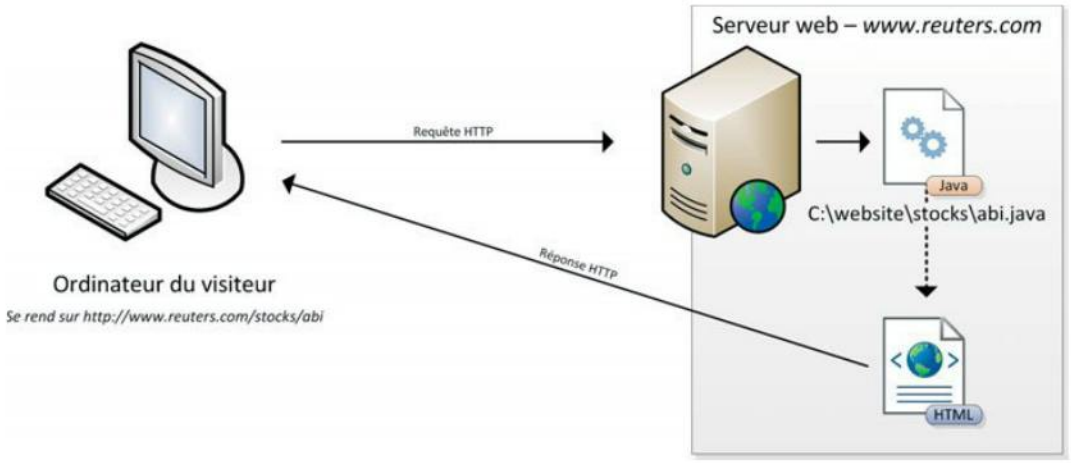


Figure 4–1 Création d'une page dynamique

EN PRATIQUE Partage du travail sur le Web

Le développement web exige de répartir les rôles entre ce qui doit s'exécuter côté serveur et ce qui doit s'exécuter côté client. Ainsi, l'accès à la base de données s'exécute côté serveur alors que la saisie d'un formulaire s'exécute côté client.

Il n'est donc pas nécessaire d'avoir un serveur web pour construire une page web et visualiser son rendu : on peut très bien composer la page à l'aide d'un éditeur HTML et la visualiser avec un navigateur, sans passer par un serveur web. C'est ce nous ferons tout au long de ce chapitre pour tester nos pages HTML, CSS et JavaScript. Laissons notre serveur se reposer un peu, nous ne tarderons pas à le solliciter de nouveau.

Structurer les pages web avec HTML5

Nous l'avons évoqué au chapitre 1, HTML constitue l'un des trois fondements du Web. Ce langage imaginé par Tim Berners-Lee sert à construire des pages et à les lier entre elles par des liens hypertextes.

HTML est un sigle qui signifie *HyperText Markup Language* (et non « Hyper Trop Moche Langage » comme les mauvais esprits le pensent). Que signifie exactement ce sigle ?

- **Language** : HTML est effectivement un langage (vous le saviez déjà !). Il obéit à des règles d'écriture et à une sémantique bien précises. Des erreurs de syntaxe sont aisément détectables.

- **Markup** : la traduction française est « balise ». Ce langage est construit à l'aide de balises (qu'on retrouve parfois sous les yeux des webmestres qui travaillent tard dans la nuit). Nous reviendrons abondamment sur cette notion par la suite.
- **HyperText** : la navigation dans le site se fait interactive par l'insertion dans chaque page de liens vers d'autres pages : les *liens hypertextes*.

> **CULTURE HTML, plus de vingt ans déjà**
>
> HTML est un langage relativement ancien (inventé à la fin des années 1980) et a beaucoup évolué au fil de ses versions successives. Les premières années, il évolua sous l'impulsion des développeurs de navigateurs. Par conséquent, aucune norme réellement formelle ne s'imposait et il fallait parfois compter avec des implémentations différentes du langage d'un navigateur à l'autre. Vers le milieu des années 1990, un effort de normalisation fut accompli par la création d'un consortium, le *World Wide Web Consortium* (abrégé en *W3C*) dont la mission était, et est toujours, de normaliser les technologies du Web.

Au départ, HTML était destiné à représenter à la fois le contenu d'une page web et son aspect graphique. À partir de la quatrième version du langage, dans l'esprit de séparation des fonctionnalités, son rôle a évolué afin de se concentrer uniquement sur le contenu et la structure de la page. L'aspect graphique n'était plus du ressort de HTML, mais des CSS.

Récemment, le langage a encore subi une évolution majeure par l'introduction d'une cinquième version. Elle standardise de nombreuses nouvelles fonctionnalités, comme l'ajout de vidéos ou de sons dans une page. Nous étudierons cette version, même si elle est toujours en cours de développement et de normalisation, car ses bases sont déjà largement stabilisées. HTML5 est de plus en plus utilisé par les développeurs, que ce soit pour les sites web ou pour les applications mobiles.

Depuis sa version 4, HTML propose deux standards de syntaxe : la syntaxe d'origine basée sur SGML, et le XHTML basé sur le langage XML. Les différences entre les deux sont assez succinctes. XHTML s'avère plus strict dans sa syntaxe ; certains raccourcis ne sont pas admis. Les auteurs de ce livre sont friands de précision, voire un peu maniaques : c'est donc ce standard d'HTML que nous allons décrire et utiliser tout au long de ce livre.

Rentrons maintenant dans le vif du sujet par l'introduction du concept de balise qui nous amènera très rapidement à écrire notre première page web HTML et à ressentir les premiers frissons des webmestres.

Le concept de « balises »

Imaginons une page web nommée « Ma collection de chats siamois ». Sans indication, le navigateur ne peut pas savoir ce que ces mots représentent. On va donc uti-

DÉFINITION Balise

Les documents HTML sont construits à l'aide de balises, que l'on nomme *tags* ou *markups* en anglais. Ces éléments se distinguent facilement du contenu réel d'une page. Elles permettent de l'annoter et de lui donner une sémantique.

liser des balises pour entourer le texte et signifier au navigateur qu'il s'agit du titre de la page. En HTML, on écrira : `<title>Ma collection de chats siamois</title>`. Facile, non ?

Cet exemple nous montre que les balises HTML sont construites à l'aide de chevrons `<` et `>` entourant leur nom (dans notre exemple, le nom de la balise est `title`). Deuxième constat, notre texte est entouré par une balise ouvrante `<title>` et une balise fermante `</title>` qui délimitent où commence et où finit le titre, la balise fermante se distinguant par la barre oblique ajoutée après le premier chevron. Les balises vont toujours par deux : une ouvrante et une fermante.

Les balises HTML peuvent être *imbriquées* pour indiquer qu'un élément se trouve dans un autre. Par exemple, si notre page contient un *article*, lui-même composé de plusieurs *sections*, on écrira en HTML :

EXEMPLE 4.1 Balises imbriquées

```
<article>
    La Bretagne, un pays à voir.
    <section>
      Introduction
      Bla bla
    </section>
    <section>
      Historique
      Bla bla
    </section>
</article>
```

En revanche, les éléments ne peuvent jamais se chevaucher. On ne pourrait pas écrire le code suivant :

EXEMPLE 4.2 Mauvais exemple : des balises qui se chevauchent

```
<article>
    La Bretagne, un pays à voir.
    <section>
      Introduction
      Bla bla
```

```
</article>
<!-- Erreur ! Il faut refermer la balise section avant de refermer
article. -->
    </section>
```

Le schéma suivant représente ce concept d'imbrication d'éléments en HTML ; à gauche ce qu'il faut faire, à droite ce qu'on ne peut pas faire.

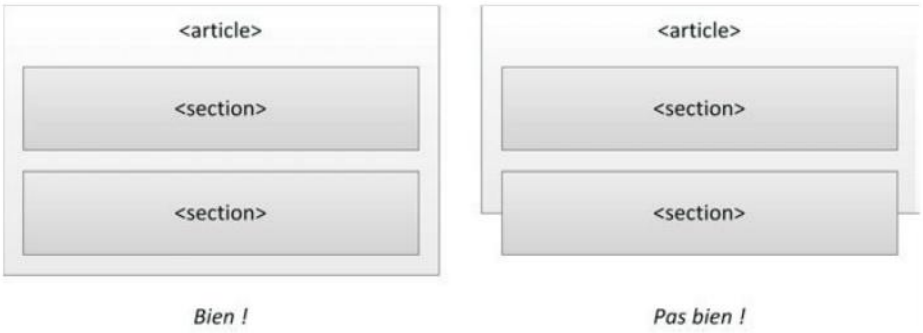


Figure 4–2
Imbrication d'éléments en HTML

DÉFINITION Attribut

Les balises peuvent être enrichies d'informations diverses par l'ajout d'*attributs*.

Imaginons que dans notre page, nous aimerions préciser qu'un article est en français et qu'un autre est en anglais. L'attribut `lang` permet cet enrichissement. On écrirait alors :

EXEMPLE 4.3 Attribut lang

```
<article lang="fr">
    La Bretagne, un pays à voir.
</article>
<article lang="en">
    Brittany, a country to be seen.
</article>
```

SYNTAXE Attribut

Les attributs se placent *dans* les balises. La valeur de l'attribut, entourée de guillemets, est séparée du nom de l'attribut par un signe = (égal).

Tous les attributs possibles sont définis dans le standard HTML et vous ne pouvez pas en inventer d'autres. Nous utiliserons abondamment les attributs par la suite ; ils se révèlent souvent comme un complément indispensable de certaines balises.

Structure d'un document HTML

Maintenant que nous avons présenté le concept de balise, nous pouvons nous intéresser à la construction d'un document HTML et à sa structure générale. Cela nous permettra de créer notre première page XHTML.

Un document XHTML commence toujours par les deux lignes suivantes, qui servent à préciser au navigateur qu'il a affaire à du XHTML dans sa version 5 :

SYNTAXE. Les deux première lignes d'un document XHTML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
```

Ensuite se construit le document HTML proprement dit, qui commence toujours par une balise racine `html` englobant tout le reste du document. Vous remarquerez que nous avons défini deux attributs dans la balise `html` (1). Le premier, `xmlns`, est propre à XML et est obligatoire. Le deuxième, facultatif, définit simplement la langue du document : ici, le français (nos amis belges mettront `fr-be`).

À l'intérieur de la balise `html`, notre document est scindé en deux grandes parties :

- La balise `head` (2) représente l'en-tête du document, qui contient toutes les informations annexes au document (par exemple son titre, son auteur, sa date de création).
- La balise `body` (3) définit le corps du document et contient le contenu de la page à afficher par le navigateur.

SYNTAXE. Forme globale d'un document XHTML5

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr"> (1)
  <head> (2)
    Contenu de l'en-tête…
  </head>
  <body> (3)
    Contenu du document…
  </body>
</html>
```

Ajoutons maintenant un titre à la page et un peu de chair au `body`.

EXEMPLE 4.4 Ajout de contenu et d'un titre

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Histoire de la France</title>
  </head>
  <body>
    L'histoire de la France commence il y a très longtemps…
  </body>
</html>
```

Nous avons maintenant notre première page HTML complète que l'on peut afficher dans un navigateur web. Pour ce faire, recopiez ce code dans un éditeur de texte (par exemple Notepad sous Windows). Enregistrez le fichier en lui donnant une extension `.html`. Vous pouvez maintenant l'ouvrir avec votre navigateur. Il affichera le renversant résultat de la figure suivante. Vous êtes presque prêt pour postuler chez Google !

Figure 4–3
Notre première page web !

Le titre de la page a été récupéré par le navigateur pour nommer l'onglet. Le contenu de la balise `body` est affiché dans la zone principale du navigateur.

Avouons-le, la page que nous venons de créer est plutôt élémentaire. Nous allons donc voir dans la section suivante d'autres balises HTML qui nous permettront de rendre cette page un peu plus attrayante.

L'encodage de la page

Il y a un attribut que nous avons volontairement omis d'expliquer jusqu'à présent et qui revêt pourtant une importance cruciale. C'est l'attribut `encoding`.

Retirons-le pour voir… Tous les accents sont fichus ! En atteste la capture d'écran suivante.

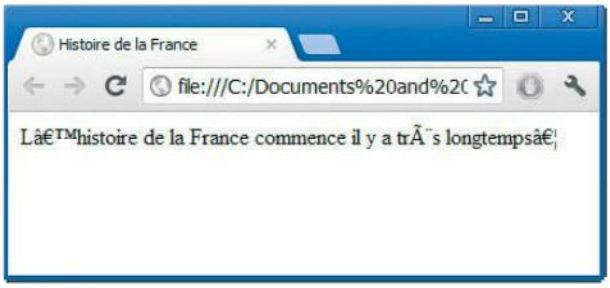


Figure 4–4
Notre première page web dans un triste état !

Ce triste résultat résulte du retrait de l'attribut `encoding` : le navigateur ne sait pas quel est l'encodage utilisé par la page et, pas de chance, en présume un mauvais.

Mais que diable entend-on par encodage ? Un peu d'histoire… Aux prémices de l'informatique, la mémoire coûtait très cher. Octet était presque synonyme de carat. En effet, on ne parlait pas en termes de téra, giga ou méga, mais plutôt en termes de bits, d'octets ou de kilooctets dans le meilleur des cas.

On évitait donc à tout prix de consommer la mémoire. On décida de représenter les caractères manuscrits sur un octet (huit bits). Cela nous donne donc $2^8 = 256$ possibilités. Nos amis américains, inventeurs de l'informatique moderne, étaient satisfaits : 256 possibilités suffisent amplement à représenter tous les caractères qu'ils utilisent (chiffres, minuscules, majuscules, ponctuations, etc.). C'était oublier que le reste du monde ne fonctionne pas comme eux ! La langue française, avec les accents, la cédille, les ligatures, etc., est plus riche dans sa palette de caractères. Et que dire des Suédois, Danois ou Norvégiens qui possèdent en plus des voyelles bizarres (å, ö, etc.), et des Chinois dont les symboles se chiffrent en milliers. 256 possibilités n'était clairement pas assez pour représenter tous les caractères de toutes les langues.

On a alors imaginé un système palliatif pour que chaque langue puisse représenter ses caractères tout en gardant la taille d'un octet. On a décidé que les caractères communs (les 26 lettres, les 10 chiffres et les ponctuations les plus courantes), seraient codés en utilisant les 128 premières possibilités, les 128 restantes étant au libre choix de chaque pays/langue/culture. Par exemple, pour les francophones, le nombre 199 allait représenter la cédille majuscule. En revanche, en ouzbek, ce nombre représentera un signe dont on n'ose reproduire la graphie. Et les Chinois ? Malheureusement pour eux, avec 256 possibilités, il était impossible de représenter tous leurs symboles.

C'est pourquoi tout document textuel (comme HTML) doit être accompagné d'une mention qui indique quel encodage est utilisé. Ainsi, le navigateur sait s'il doit afficher une cédille ou un « o » avec *umlaut* s'il rencontre un octet qui vaut « 199 ». L'encodage sur un octet le plus souvent utilisé par les francophones est le « ISO 8859-1 » (on inventa par la suite l'ISO 8859-15 pour ajouter, par exemple, le symbole euro).

Le prix de la mémoire a chuté, et se limiter à un octet n'a plus de raison d'être. On a imaginé d'autres encodages qui utilisent 2, 4 voire 8 octets, permettant de représenter les symboles qui foisonnent aux quatre coins du monde. Aujourd'hui, l'encodage « multi-octet » le plus populaire est l'UTF-8, car il est astucieux : les 128 premiers caractères sont toujours codés sur un octet, tandis que les caractères plus exotiques sont codés sur deux octets. Si le document ne contient que des caractères élémentaires (pas d'accents), il sera affiché correctement même si le navigateur n'est pas au courant qu'on travaille en UTF-8.

C'est ce qui donne parfois des résultats surprenants, comme dans notre exemple : notre document est codé en UTF-8, mais le navigateur pense qu'il a affaire à de l'ISO 8859-1. Dès lors, tous les caractères accentués sont représentés par deux caractères.

Pour conclure, lorsque vous créez un fichier HTML, il faut toujours savoir dans quel encodage votre éditeur travaille. Nous vous conseillons évidemment de toujours travailler en UTF-8. Cela se fait via l'attribut `encoding`.

Quelques éléments HTML

HTML sert avant tout à décrire et à structurer le contenu des pages web. Les éléments que nous allons décrire dans les sections suivantes ajouteront une sémantique à nos textes (ce texte est un titre, ce texte est un lien, ce texte est une liste à puces, etc.) et structureront notre document.

Principaux éléments de structuration HTML

Dans notre premier exemple de page web, nous avons placé du texte directement dans la balise `body`. Ce n'est pas interdit, mais lorsque nous aurons beaucoup de texte, mieux vaudra le structurer. Plusieurs balises servent à clarifier le contenu :

- `<p>` (1) représente un paragraphe de texte.
- `<article>` (2) représente un article, c'est-à-dire un morceau du contenu de la page web qui peut-être lu séparément du reste (exactement comme un article de journal). Cette balise contiendra généralement plusieurs paragraphes.
- `<section>` (3) permet de regrouper des articles de forme ou de contenu similaire. Cette balise sert également à diviser un article en plusieurs morceaux (introduction, conclusion, etc.).

- <h1> 4, <h2> 5, <h3> 6, etc. représentent des titres de niveaux différents. Un document aura généralement un seul titre de niveau un, ensuite plusieurs titres de niveaux deux, chacun comportant des titres de niveaux trois, et ainsi de suite. À ce stade, rien n'est précisé sur la taille graphique de ces différents niveaux de titres (c'est la responsabilité de CSS) : HTML se limite à constater que plusieurs niveaux de titres peuvent coexister.

À l'aide de ces premiers éléments, nous pouvons déjà réaliser un premier document nettement plus fourni. Le code suivant s'inspire d'une page d'accueil d'un site de presse écrite. Cette page possède deux grandes sections : actualités nationales et actualités internationales. Dans chacune des sections, on retrouve plusieurs articles.

EXEMPLE 4.5 Page web au contenu structuré

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Dernières actualités</h1> 4
    <section> 3
      <h2>Actualités nationales</h2> 5
      <article> 2
        <h3>65% des Français sont contre les sondages</h3> 6
       <p>D'après un sondage TNS Sofres Ipsos CSA Ifop Médiamétrie,
        65% des Français sont contre les sondages contre 42% qui
        sont pour. Blah blah blah.</p> 1
      </article>
      <article>
        <h3>Incendie dans une usine d'extincteurs</h3>
        <p>En Mayenne, le feu a pris ce matin dans une usine blah blah
         blah.</p>
      </article>
    </section>
    <section>
      <h2>Actualités internationales</h2>
      <article>
        <h3>Obama favorable à une intervention en Belgique</h3>
        <p>Suite aux problèmes communautaires, blah blah blah.</p>
      </article>
    </section>
  </body>
</html>
```

Voici le résultat dans un navigateur :

Figure 4–5
Une page web structurée

Pour la plupart, les éléments de structuration n'ont pas de rendu visuel. Les titres sont mis en évidence (police plus grande et grasse), mais les articles et les sections n'ont rien pour les identifier du regard. Du moins est-ce ainsi par défaut, car à l'aide des CSS, on pourra modifier l'apparence de tous ces éléments à notre guise.

Éléments de structuration annexes : en-têtes, pieds de page et sections

Dans la plupart des sites web, les pages présentent de nombreux éléments annexes au contenu principal : des bannières, des menus, des pieds de page, des formulaires de recherche, etc. HTML5 prévoit une série d'éléments permettant d'englober tout ce contenu annexe :

- `<footer>` (1) représente un pied de page ou de section.
- `<header>` (2) représente un en-tête de page ou de section.
- `<aside>` (3) représente du contenu annexe, par exemple un menu ou un champ de recherche.

Passons immédiatement à un exemple illustrant l'usage de ces balises.

EXEMPLE 4.6 Page web au contenu structuré avec contenu annexe

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <header> 2
      <p>La Libre France, un journal où il fait bon lire.</p>
    </header>
    <aside> 3
      <p>Nous sommes le 29 février 2012</p>
    </aside>
    <h1>Dernières actualités</h1>
    …
    <footer> 1
      <p>© La Libre France 2012</p>
    </footer>
  </body>
</html>
```

La figure suivante illustre le résultat de ce code dans un navigateur.

Figure 4–6
Une page web structurée avec du contenu annexe

Certes, le rendu n'en demeure pas moins laid (du moins d'une sobriété à la limite de l'austérité). Le positionnement du texte reste basique : les lignes se suivent simplement alors qu'on pourrait s'attendre à ce que le contenu de la balise `aside` apparaisse ailleurs. Patience, la section sur CSS nous apprendra comment procéder.

Les liens hypertextes

Pour définir un lien dans une page (par défaut un texte bleu souligné qui, lorsqu'on clique dessus, nous amène sur une autre page web), il faut utiliser la balise `<a>`. Pour définir la cible du lien, on utilise un attribut nommé `href` (voyez comme les attributs sont vite indispensables !). Le contenu de la balise est le texte du lien.

SYNTAXE. La balise <a>

```
<a href="adresse Internet">Texte du lien</a>
```

EXEMPLE 4.7 Page avec lien hypertexte

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Sites partenaires</h1>
    <p><a href="http://www.lemonde.fr">Visiter le site du Monde !</a></p>
  </body>
</html>
```

Le résultat de ce code dans un navigateur est le suivant.

Figure 4–7
Exemple de rendu d'un lien hypertexte

Les listes

HTML permet également de définir des listes. Elles peuvent être de deux types :

- soit numérotées, auquel cas on utilisera la balise `<ol>` et les éléments seront numérotés automatiquement ;
- soit à puces, auquel cas on utilisera la balise `<ul>`.

Les éléments de la liste sont définis à l'aide de la balise `<li>` que l'on répétera autant de fois qu'il y a d'éléments.

SYNTAXE. Liste

```
<ul>     # ou <ol>
  <li>item 1</li>
  <li>item 2</li>
  <li>...</li>
</ul>    # ou </ol>
```

EXEMPLE 4.8 Page contenant une liste à puces

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Pays de diffusion du journal</h1>
    <ul>
      <li>France</li>
      <li>Belgique</li>
      <li>Allemagne</li>
    </ul>
  </body>
</html>
```

Le résultat de ce code dans un navigateur est le suivant.

Figure 4–8
Exemple de rendu d'une liste HTML

Les images

Pour insérer une image dans une page web, on passe par trois balises différentes :

- `<figure>` représente une « figure » au sens large. Cela peut être une image, du code, un diagramme, etc.
- `<img>` représente l'image proprement dite. C'est à l'aide de son attribut `src` que l'on va définir où se trouve le fichier contenant l'image. La balise `<img>` sera placée dans la balise `<figure>`.
  Un second attribut, obligatoire, est à spécifier pour la balise : `alt`. Il ajoute un texte alternatif (invisible) décrivant l'image au cas où elle ne s'afficherait pas. Ce texte alternatif est également très important pour renseigner les internautes malvoyants (dont les navigateurs sont capables de « lire » ces textes alternatifs).
  Deux autres attributs sont intéressants : `width` et `height` : ils précisent la hauteur et la largeur de l'image en pixels. C'est très utile lorsque l'image est en cours de chargement ; le navigateur sait déjà quelle place elle va prendre et peut en tenir compte pour afficher correctement la page web.
- `<figcaption>` précise une légende pour la figure. Cette balise, optionnelle, sera placée dans la balise `<figure>`.

SYNTAXE. Les balises <figure>, <img> et <figcaption>

```
<figure>
  <img src="adresse du fichier image"
  alt="Texte court décrivant l'image" width="largeur en pixels"
  height="hauteur en pixels" /> (1)
  <figcaption>légende de l'image</figcaption>
</figure>
```

On remarque que la balise `<img>` n'est pas fermée comme toutes celles rencontrées jusqu'à présent (1). Il s'agit d'un raccourci très utilisé : comme la balise n'a pas de contenu, plutôt que d'écrire `<img src="..."></img>`, on écrira `<img src="..." />`. La balise est dans ce cas à la fois ouvrante et fermante.

EXEMPLE 4.9 Page contenant une image

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Partenaires</h1>
```

```
    <figure>
      <img src="https://www.google.be/images/srpr/logo3w.png"
      alt="The Google Logo" width="275" height="95" />
      <figcaption>Fig1. - Le Logo de Google.</figcaption>
    </figure>
  </body>
</html>
```

Le résultat de ce code dans un navigateur est le suivant.

Figure 4–9
Exemple de rendu d'une insertion d'image dans une page web

Mise en évidence du texte

HTML permet de définir des portions de textes à mettre en évidence. Deux balises sont utiles à cet effet :

- `<strong>` : le texte entouré par cette balise est défini comme « important ».
- `<em>` : le texte entouré par cette balise est défini comme « avec emphase ».

**SYNTAXE. Les balises et **

```
<p>Voici du <strong>texte important</strong> et du <em>texte mis en
emphase</em>.</p>
```

EXEMPLE 4.10 Page avec mise en évidence de texte

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Mise en évidence de texte</h1>
    <p>Voici du <strong>texte important</strong> et du <em>texte
    mis en emphase</em>.</p>
  </body>
</html>
```

Voici le résultat de ce code dans un navigateur.

Figure 4–10
Exemple de rendu de texte mis en évidence

EN PRATIQUE Erreur courante : et ne sont pas gras et italique

Par défaut, les navigateurs mettent en gras les textes entourés de la balise `<strong>` et en italique ceux entourés de la balise `<em>`. Le raccourci (erroné) est vite pris de dire : « pour mettre un texte en gras, il suffit d'utiliser la balise `<strong>` ». C'est tout bonnement incorrect.
HTML ne sert qu'à définir et structurer le contenu d'une page et à lui donner une sémantique, certainement pas à s'occuper du formatage et de l'aspect du texte. HTML nous permet de dire simplement qu'un texte est *important* ou *a de l'emphase*. Les goûts et les couleurs, c'est pour CSS : c'est à ce niveau qu'on définira comment doit apparaître un texte « important » ou « à emphase ». Certains le laisseront en gras, d'autres le mettront plutôt en rouge. Il se pourrait très bien qu'un jour, un navigateur décide que, par défaut, les textes importants ne soient plus en gras, mais en bleu fluo.

Les formulaires

L'objectif de ce livre est de vous apprendre à réaliser des sites web dynamiques, en d'autres mots, des sites web dont les pages sont créées dynamiquement par un serveur web.

Une des grandes forces du Web dynamique est la possibilité de passer à une page web des paramètres, souvent renseignés par l'utilisateur lui-même.

SYNTAXE Insertion de paramètres dans les requêtes HTTP

Par exemple, sur la page de recherche de Google, nous avons un champ de recherche. Lorsque l'utilisateur entre la requête « Solvay » dans ce champ et clique sur le bouton *Recherche Google*, l'URL suivante est appelée :

```
http://www.google.be/search?q=Solvay
```

Cette adresse n'est pas fictive, vous pouvez l'essayer dans votre navigateur. Plus précisément, c'est la page web `search` qui est appelée avec un paramètre nommé `q` et dont la valeur est `Solvay`.

Nous allons étudier comment insérer dans une page web des zones de texte et des boutons pour communiquer ce type d'information.

En termes HTML, on parle de « formulaires » lorsqu'il s'agit de récolter des informations auprès de l'utilisateur pour les transmettre en paramètre. Un formulaire peut contenir des zones de saisie de texte, des cases à cocher, des listes de choix, des boutons radio, etc. Chaque formulaire doit contenir un *bouton de soumission*.

DÉFINITION Bouton de soumission

Il s'agit d'un bouton qui, une fois pressé, appelle une URL, lui transmettant en paramètres tous les champs du formulaire.

Servons-nous d'un exemple pour mieux illustrer le concept de formulaires. Imaginons que l'on veuille refaire la page d'accueil de Google. Cette page contient deux champs : une zone de texte et un bouton de soumission. Pour réaliser ce formulaire en HTML, nous allons utiliser deux balises :

- `<form>` représente le formulaire et entourera tous les champs du formulaire.
- `<input>` représente un champ. Son attribut `type` permet de définir la forme que prendra le champ. Pour une zone de saisie, on utilisera le type `text`, pour le bouton de soumission, on utilisera le type `submit`.

Voici le code HTML de notre formulaire.

SYNTAXE. Formulaire de recherche type Google

```
<form action(1)="http://www.google.be/search" method(2)="get">
  Rechercher :(3)
  <input type="text(4)" name(5)="q" />
  <input type="submit(6)" value(7)="Rechercher !" />
</form>
```

L'attribut `action` (1) de la balise `form` sert à soumettre le formulaire et à appeler la page http://www.google.be/search.

L'attribut `method` (2) (prenons la valeur `get` pour l'instant, nous discuterons plus loin d'une alternative) dit au navigateur que lorsqu'on clique sur le bouton de soumission du formulaire, les champs et leur valeur doivent être ajoutés à la fin de l'URL définie dans l'attribut `action`.

Ensuite, nous avons un petit texte (3) suivi de nos deux champs. Le premier est de type `text` (4). Il s'agit donc de la zone de saisie. L'attribut `name` (5) sert à préciser le nom de ce champ et sera utilisé par le navigateur lorsqu'il devra construire l'URL. Le nom du champ sera le nom du paramètre ajouté à l'URL appelée.

Le second champ est de type `submit` (6). Il s'agit du bouton de soumission. L'attribut `value` (7) définit l'intitulé qui doit apparaître sur le bouton.

Voyons maintenant comment faire apparaître ce formulaire par un exemple de code HTML complet.

EXEMPLE 4.11 Page proposant un formulaire

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Google</title>
  </head>
  <body>
    <h1>Bienvenue sur ce faux Google !</h1>
    <form action="http://www.google.be/search" method="get">
      Rechercher :
      <input type="text" name="q" />
      <input type="submit" value="Rechercher !" />
    </form>
  </body>
</html>
```

Ce code produit le résultat suivant dans un navigateur.

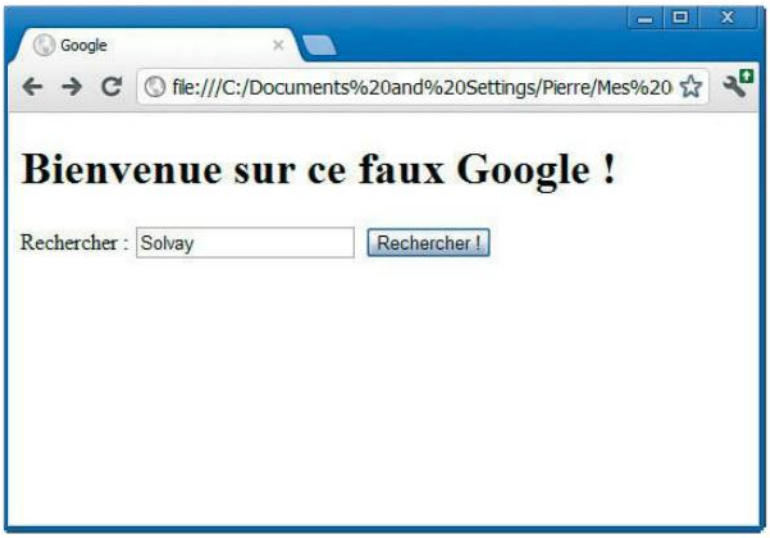


Figure 4–11
Exemple de rendu d'un formulaire dans une page web

Lorsqu'on clique sur le bouton *Rechercher*, la page suivante est appelée. On constate que l'URL construite est bien celle attendue.

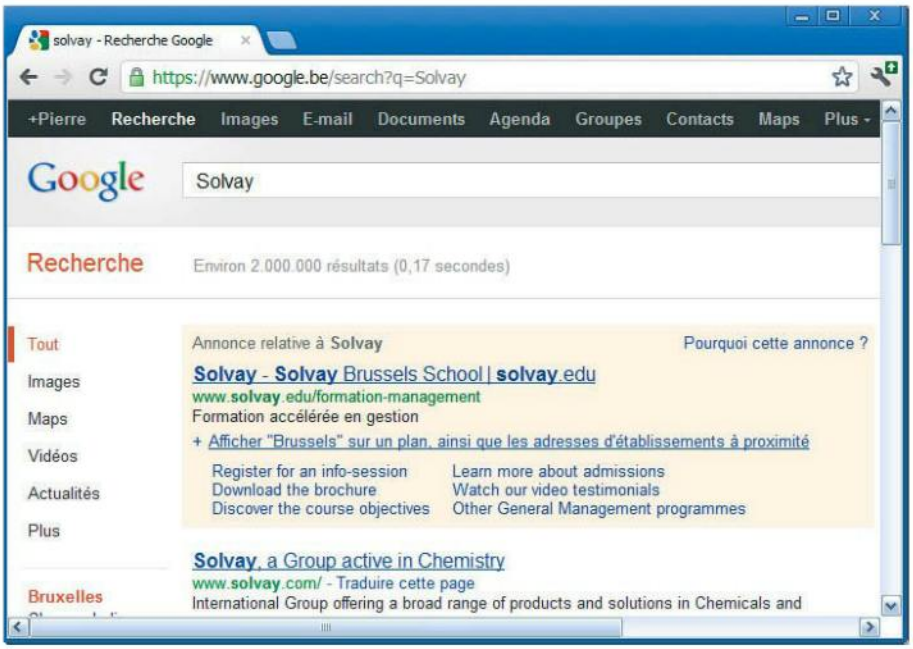


Figure 4–12
Page appelée lors de la soumission du formulaire

Autres éléments HTML

Il existe beaucoup d'autres éléments HTML. On pense aux éléments pour ajouter des tableaux, des vidéos, des sons, ou pour enrichir vos formulaires avec d'autres types champs ou affiner la sémantique de vos textes. Prenez cela comme une mise en bouche pour aller consulter d'autres ouvrages ou sites web.

> **OUTIL Écrire des pages web sans manipuler directement HTML5**
>
> De nombreux logiciels de développement existent, tels Dreamweaver, qui vous permettent jusqu'à un certain point de vous passer de ce langage en obtenant directement à l'écran le résultat de vos manipulations (WYSIWYG – *What You See Is What You Get*).

Maintenant que nous avons appris les bases de HTML et que nous sommes capables de créer une page web, il est temps de passer à l'étape suivante, à savoir la mise en forme de la page à l'aide des CSS.

> **À LIRE**
>
> 📖 Rodolphe Rimelé, *HTML5 - Une référence pour le développeur web*, Eyrolles 2011

Mettre en forme avec les feuilles de styles CSS

Lorsque HTML est apparu, sa vocation première était de traiter tous les aspects d'une page web : à la fois son contenu et son apparence. Se trouvaient donc mélangées dans le code HTML des balises s'occupant de la sémantique du contenu (tel texte est un paragraphe, tel autre texte est un titre) et des balises de formatage (tel texte doit être bleu et souligné).

Avec le temps, et particulièrement à partir de la version 4 d'HTML, la définition du contenu proprement dit d'une page web et son aspect ont été tenus séparés. Les avantages sont les suivants :

- **Centralisation des instructions relatives à l'apparence d'un site web**. Ainsi, si on veut des titres en gras de taille « 16 », on ne le définira pas au niveau de chaque titre de chaque page du site, mais à un seul endroit, où il sera facile d'en modifier au besoin les caractéristiques.
- **Séparation des différentes tâches de développement**. La division du travail permet au développeur de ne s'occuper que du contenu brut des pages HTML tandis qu'un graphiste web se charge de définir l'apparence que doit prendre le site. L'un ne doit pas s'aventurer dans le code de l'autre, sous risque de représailles !

CSS signifie *Cascading Style Sheets* (en français, « feuilles de styles en cascade »). Analysons plus en détail ce sigle :

- « Feuilles de styles » désigne simplement le fait que CSS permet de « styler » une page web et que son code, bien séparé du HTML, est placé dans des « feuilles ».
- « En cascade » désigne la possibilité de définir à plusieurs endroits des styles s'appliquant à un même élément. Des règles de priorité se chargent de départager ces instructions de styles afin de savoir lesquelles il faut réellement appliquer à l'élément.

Analysons maintenant deux principes de base des CSS : les propriétés et les sélections d'éléments.

Les propriétés CSS

Changer l'apparence d'un objet se fait via des *propriétés*. Elles sont nombreuses et variées. Citons quelques exemples :

- `background-color` définit la couleur d'arrière-plan d'un élément HTML.
- `border-width` définit la taille de la bordure d'un élément.
- `height` définit la hauteur d'un élément.
- `font-family` définit la police à utiliser pour les textes (Times, Arial, etc.).

À chaque propriété est associée une *valeur*. On peut par exemple associer la valeur `red` à la propriété `background-color`. En CSS, on écrira :

EXEMPLE 4.12 Affectation d'une valeur à une propriété CSS

```
background-color: red;
```

La propriété et sa valeur sont séparées par un signe deux-points ; le tout est terminé par un point-virgule.

Les sélecteurs CSS

Les propriétés CSS servent à définir l'apparence d'une page web mais encore faut-il préciser à quel(s) élément(s) elles s'appliquent. C'est le rôle des sélecteurs. Ces derniers sélectionnent un élément HTML ou plusieurs, au(x)quel(s) on va appliquer une propriété ou plusieurs.

La syntaxe générale est la suivante :

SYNTAXE. Syntaxe générale CSS

```
selecteur {
  propriete1: valeur1;
  propriete2: valeur2;
  ...
}
```

On définit d'abord un sélecteur. Ensuite, entres accolades, on insère toutes les propriétés CSS que l'on désire appliquer aux objets sélectionnés.

Sélectionner toutes les balises de même nom

Il y a de nombreuses manières de définir un sélecteur. On peut décider de sélectionner toutes les balises portant un même nom. Par exemple, si on veut que tous les paragraphes HTML aient un fond rouge, on écrira :

EXEMPLE 4.13 Fond rouge pour tous les paragraphes

```
p {
  background-color: red;
}
```

Le sélecteur est simplement `p`, le nom de la balise choisie.

Sélectionner un élément particulier : id en HTML et # en CSS

Si on désire mettre en rouge un paragraphe en particulier, la première étape sera de donner un identifiant au paragraphe dans le code HTML. Pour ce faire, on utilise l'attribut `id` comme suit :

EXEMPLE 4.14 Identifier un élément HTML avec l'attribut id

```
<ul id="listeDesPays">
  <li>Belgique</li>
  <li>France</li>
  <li>Allemagne</li>
</ul>
```

ATTENTION ! L'identifiant doit être unique

Un `id` ne doit évidemment être attribué qu'à un seul élément, sous peine d'ambiguïté.
Par ailleurs, il ne doit pas être confondu avec l'attribut `name` que l'on retrouve dans les formulaires et que nous découvrirons par la suite.

Ensuite, en CSS, pour sélectionner précisement cet élément :

EXEMPLE 4.15 Fond rouge pour la liste « listeDesPays »

```
ul#listeDesPays {
  background-color: red;
}
```

Le sélecteur est alors agrémenté du signe dièse suivi de l'`id` de l'élément.

Sélectionner quelques éléments de même nature : class en HTML et . en CSS

Si l'on désire cibler plusieurs éléments de même nature, mais pas tous, c'est également possible. Imaginons que dans notre liste de pays, nous aimerions mettre en bleu tous les pays francophones. Pour cela, il faut d'abord en HTML, marquer ces éléments comme étant similaires. Cela se fait via l'attribut `class` :

EXEMPLE 4.16 Marquer des éléments similaires

```
<ul id="listeDesPays">
  <li class="francophone">Belgique</li>
  <li class="francophone">France</li>
  <li>Allemagne</li>
</ul>
```

Un élément HTML peut avoir plusieurs classes. Par exemple :

EXEMPLE 4.17 Plusieurs classes pour un élément

```
<ul id="listeDesPays">
  <li class="francophone neerlandophone">Belgique</li>
  <li class="francophone">France</li>
  <li>Allemagne</li>
</ul>
```

Ensuite, en CSS, on cible tous les éléments de même classe :

EXEMPLE 4.18 Fond bleu pour les pays francophones

```
li.francophone {
  background-color: blue;
}
```

Le sélecteur est alors agrémenté d'un point suivi de la classe à cibler. Dans notre exemple, Belgique et France auront un fond bleu car tous deux sont « au moins » de classe « francophone ».

Appliquer une propriété seulement quand l'élément est dans un état donné

On peut également appliquer un ensemble de propriétés à un élément lorsqu'il est dans un certain état. Le sélecteur est alors agrémenté de deux-points suivi de l'état à cibler. Exemple courant, imaginons que l'on veuille colorer en rouge les liens hypertextes au survol de la souris. On écrira en CSS :

EXEMPLE 4.19 Liens en rouge au survol de la souris

```
a:hover {
  color: red;
}
```

Ce sélecteur signifie « tous les éléments `a` lorsqu'ils sont survolés ».

Voici quelques états possibles pour les liens hypertextes :

- `link` : tous les liens non visités.
- `visited` : tous les liens visités.
- `hover` : tous les liens survolés.
- `active` : tous les liens « actifs » (qui ont le focus).

Combiner les sélecteurs

On peut également « imbriquer » des sélecteurs pour plus de raffinement. Imaginons les listes suivantes :

EXEMPLE 4.20 Deux listes différentes, mais avec mêmes classes

```
<ul id="listeDesPays">
  <li class="francophone neerlandophone">Belgique</li>
  <li class="francophone">France</li>
  <li class="germanophone">Allemagne</li>
</ul>
<ul id="listeDesPresidents">
  <li class="francophone">François Hollande</li>
  <li class="anglophone">Barak Obama</li>
  <li class="francophone">Joseph Kabila</li>
</ul>
```

Ici, les mêmes classes ont été appliquées à deux listes différentes. Si l'on veut mettre en bleu les pays francophones mais pas les présidents francophones, on écrira en CSS :

EXEMPLE 4.21 Fond bleu pour les pays francophones

```
ul#listeDesPays li.francophone {
  background-color: blue;
}
```

Le sélecteur dans ce cas signifie « tous les éléments `li` qui ont la classe `francophone` et qui se trouvent dans un élément `ul` d'`idlisteDesPays` ».

Si l'on veut appliquer un même ensemble de propriétés à plusieurs sélecteurs, il faut séparer ces derniers par des virgules. Le code suivant :

EXEMPLE 4.22 Fond bleu pour les francophones et les anglophones

```
li.francophone, li.anglophone {
  background-color: blue;
}
```

est équivalent à :

EXEMPLE 4.23 Exemple. Fond bleu pour les francophones et les anglophones

```
li.francophone {
  background-color: blue;
}

li.anglophone {
  background-color: blue;
}
```

On peut très bien combiner la sélection d'un élément et un état particulier :

EXEMPLE 4.24 Liens vers Google en rouge au survol de la souris

```
a#lienVersGoogle:hover {
  color: red;
}
```

Ce sélecteur signifie « lorsque l'élément `a` dont l'`id` est `lienVersGoogle` est survolé ».

Sélectionner tous les éléments

Enfin, sachez que l'étoile permet de sélectionner tous les éléments.

EXEMPLE 4.25 Fond rouge pour tous (très laid, on vous l'accorde)

```
* {
  background-color: red;
}
```

D'autres sélecteurs existent encore, mais ils sont moins utilisés, car ils ne sont pas pris en charge par tous les navigateurs.

Lier CSS et HTML

Nous venons de voir comment cibler des éléments et modifier leur apparence grâce à des instructions CSS. Reste à savoir où placer ce code. Plusieurs possibilités s'offrent à vous.

Placer le code CSS dans les balises HTML

Une première (mauvaise) possibilité consiste à placer directement des règles CSS au niveau des éléments HTML, à l'aide de l'attribut `style`. C'est ce qu'on appelle du CSS *inline*.

EXEMPLE 4.26 Document HTML contenant des règles CSS

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
  </head>
  <body>
    <h1>Liste de pays</h1>
    <ul>
      <li style="background-color: red;">Belgique</li>
      <li style="background-color: red;">France</li>
      <li>Allemagne</li>
     </ul>
  </body>
</html>
```

Cette méthode est fortement déconseillée, car on perd tout l'avantage de la séparation du contenu et de la forme.

Placer le code CSS dans l'en-tête du fichier HTML

Deuxième solution, vous pouvez placer le code CSS dans l'en-tête du fichier HTML. Il faudra l'insérer dans une balise nommée `<style>`, elle-même placée dans la balise `<head>`.

EXEMPLE 4.27 Règles CSS placées dans le code HTML

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
    <style>
      ul#listeDesPays li.francophone {
        background-color: blue;
      }
    </style>
  </head>
  <body>
    <h1>Liste de pays</h1>
    <ul id="listeDesPays">
      <li class="francophone neerlandophone">Belgique</li>
      <li class="francophone">France</li>
      <li>Allemagne</li>
     </ul>
  </body>
</html>
```

Cet exemple nous donne le rendu suivant :

Figure 4–13
Notre premier CSS !

C'est votre première CSS, mais par la suite vous n'aurez de CeSSe de recourir à cette pratique !

Placer le code CSS dans un fichier séparé

La troisième solution, préférable, consiste à placer le code dans un fichier séparé auquel on donne l'extension `.css`. Ensuite, dans le HTML, on insère dans l'en-tête un lien vers ce fichier CSS.

EXEMPLE 4.28 Document HTML avec lien vers un fichier CSS

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>La Libre France - Actualités</title>
    <link rel="stylesheet" type="text/css" href="styleAffreux.css" />
  </head>
  <body>
    <h1>Liste de pays</h1>
    <ul id="listeDesPays">
      <li class="francophone neerlandophone">Belgique</li>
      <li class="francophone">France</li>
      <li>Allemagne</li>
     </ul>
  </body>
</html>
```

EXEMPLE 4.29 Contenu de styleAffreux.css

```
ul#listeDesPays li.francophone {
  background-color: blue;
}
```

La meilleure solution reste de placer son CSS dans un fichier séparé. Ainsi, graphistes et développeurs peuvent travailler en totale indépendance les uns des autres, et le fichier CSS peut être utilisé pour plusieurs pages différentes.

Dans les sections suivantes, nous allons nous attarder sur quelques propriétés et concepts CSS bien utiles et pas toujours évidents à comprendre.

Dimensions des éléments en CSS

CSS considère tout élément HTML comme une *boîte*, ou, en d'autres mots, un rectangle possédant une largeur et une hauteur. Il convient d'être plus précis sur la notion de boîte. En effet, un élément peut posséder des marges extérieures, intérieures et des bordures, comme illustré à la figure suivante.

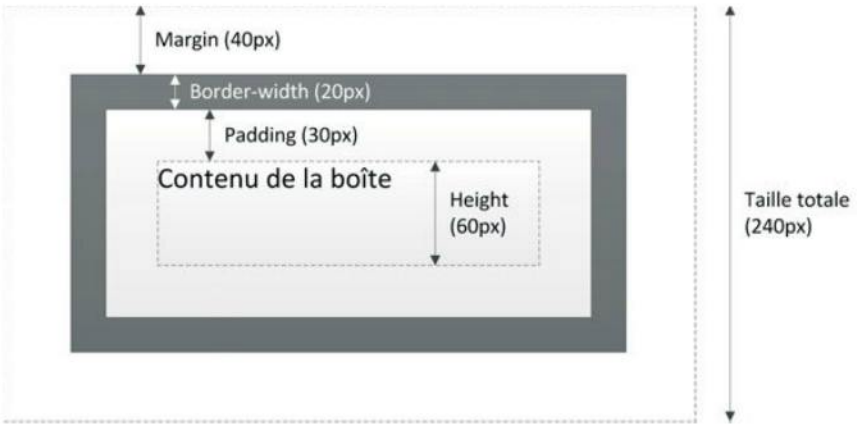


Figure 4–14
Le modèle de « boîte » CSS

Marges extérieures (appelées `margin` en anglais), marges intérieures (`padding`), bordures (`border`), hauteurs (`height`) et largeurs (`width`) peuvent être modifiées à notre guise via des propriétés CSS ad hoc. Pour l'élément représenté sur la figure, on aura le CSS suivant (on imagine que l'élément est un paragraphe) :

EXEMPLE 4.30 Personnalisation des dimensions d'un élément

```
p {
  margin: 40px;
  padding: 30px ;
  height: 60px;
  border: 20px solid black;
  width: 100%; (1)
}
```

À la lecture de ce code, on constate qu'on a défini une hauteur fixe en pixels. La largeur, en revanche, a été définie comme étant égale à 100 % (1), ce qui signifie que la boîte va prendre toute la place disponible en largeur.

Chaque élément possède des valeurs par défaut pour ces différentes propriétés CSS. Par exemple, les paragraphes ont par défaut une marge verticale permettant de les espacer les uns des autres.

Modifier ces valeurs donne toute liberté pour modifier l'aspect des pages web. Nous verrons par la suite un exemple plus complet illustrant un usage judicieux de ces propriétés.

Imbrication des éléments (boîtes)

En CSS (et en HTML), les éléments sont tous imbriqués. L'élément principal, qui englobe tous les autres est le `html`. Ensuite, chaque élément peut lui-même englober d'autres éléments qui eux-mêmes peuvent en englober d'autres, et ainsi de suite. La figure suivante illustre ce principe d'imbrication.

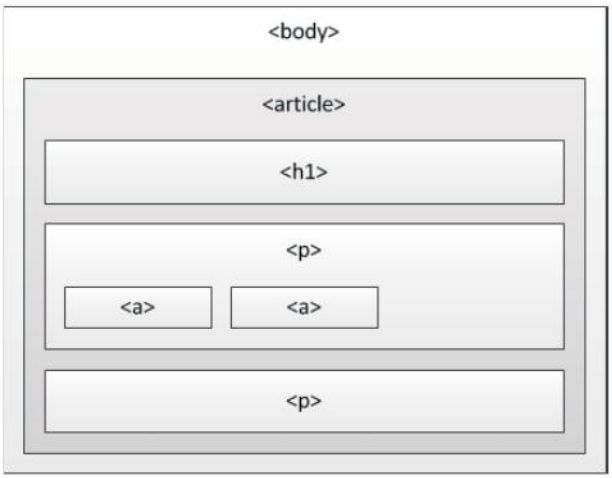


Figure 4–15
Imbrication de « boîtes » en CSS

Ce concept d'imbrication amène les notions suivantes :

- Lorsqu'un élément en contient un autre, il est nommé « parent ».
- Un élément contenu dans un autre est nommé « enfant ».
- Tous les éléments enfants d'un même parent sont des « frères ».

Souvent, le comportement d'un enfant sera dépendant de celui de son parent et de ses frères. Lorsqu'on spécifie par exemple que la largeur d'un élément doit être égale à 100 %, c'est toujours par rapport au parent. L'élément prendra la totalité de la largeur héritée du parent et non la totalité de la largeur de la fenêtre.

Positionnement par défaut des éléments en CSS

Positionner les éléments peut se révéler ardu si l'on ne comprend pas comment CSS procède.

En CSS, tous les éléments *frères* se suivent dans leur *parent*. En d'autres mots, ils sont placés « l'un après l'autre ». Cette expression diffère cependant en fonction du type d'élément en présence. Il existe fondamentalement deux types d'éléments :

- Les **éléments `block`** prennent (par défaut) la totalité de la largeur disponible. « Se suivre » signifie donc pour eux « être placés l'un en dessous de l'autre ». Les éléments de type `block` sont, par exemple, les paragraphes, les articles, les sections, les titres et les listes.
- Les **éléments `inline`** prennent la largeur de leur contenu, comme les balises représentant des liens hypertextes. « Se suivre » signifie donc pour ces éléments « être placé l'un à côté de l'autre, avec éventuellement un retour à la ligne s'il n'y a plus de place horizontalement ». Les éléments de type `inline` sont, par exemple (et assez naturellement), les images, les liens hypertextes, les balises de mise en emphase et les boutons.

La figure suivante illustre ces deux concepts.

Figure 4–16
Positionnement en CSS

On voit clairement que les paragraphes prennent toute la largeur de la fenêtre (si on exclut les marges) et que les liens hypertextes prennent juste la largeur de leur texte.

Lorsque des éléments frères sont positionnés en se suivant « l'un après l'autre », on dira qu'ils suivent le « flux courant ».

Il est important de noter que seuls les éléments de type `block` peuvent voir leurs dimensions modifiées à l'aide des propriétés `margin`, `width` et `height`.

Sortir certains éléments du flux

CSS permet de *sortir* des éléments du flux courant d'affichage. Ce mécanisme est un des plus complexes à comprendre, c'est pourquoi nous nous y attardons.

Lorsqu'on sort un élément du flux courant, il n'est plus placé par rapport à ses frères en se suivant « l'un après l'autre ». Un exemple est donné à la figure suivante.

Figure 4–17 Le paragraphe 2 est sorti du flux courant.

Dans cet exemple, nous avons sorti le paragraphe 2 du flux. C'est triste pour ses frères, mais c'est comme s'il n'existait plus : le paragraphe 3 vient se placer après le paragraphe 1. Il est intéressant de noter également qu'après être sorti du flux, le paragraphe 2 a pris, par défaut, la largeur de son contenu.

La sortie du flux se fait via la propriété CSS `position`, qui peut prendre plusieurs valeurs, en fonction de ce que l'on désire :

- `static` : valeur par défaut de la propriété. Indique que l'élément doit rester dans le flux.
- `relative` : ne sort pas l'élément du flux, mais permet de décaler sa position par rapport à celle qu'il aurait dans le flux. Pour effectuer un décalage, on utilise les propriétés `top`, `left`, `right` et `bottom`.

- `absolute` : sort l'élément du flux. On peut également utiliser les propriétés CSS `top`, `left`, `right` et `bottom` pour définir le positionnement de l'élément, lequel se fera par rapport au premier élément parent qui a une position autre que statique. S'il n'y en a aucun, ce sera par rapport à l'élément racine `html` (en d'autres mots, par rapport au début de la page web).

La figure suivante illustre les positions relatives et absolues.

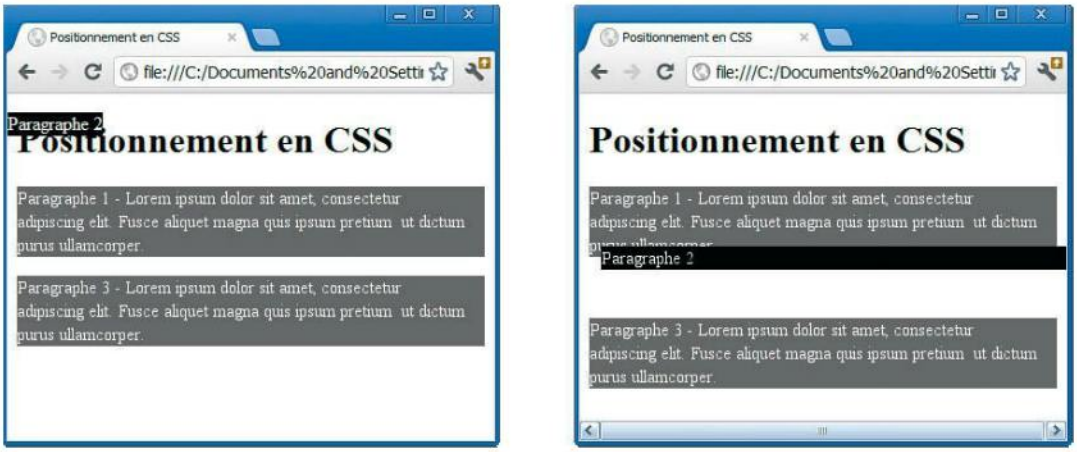


Figure 4–18 Exemple de positionnement absolu et relatif

Dans l'exemple de gauche, le paragraphe 2 a été mis en position `absolute` avec `top` et `left` à 0 à l'aide du code suivant :

EXEMPLE 4.31 Positionnement absolu du paragraphe 2

```
p#p2 {
  position: absolute;
  top: 0;
  left: 0;
}
```

Résultat, le paragraphe a été positionné en haut à gauche du premier élément parent qui n'est pas en position *statique*. Dans notre cas, le paragraphe a été placé par rapport au début de la page. Le paragraphe 3, quant à lui, est « remonté » pour prendre la place du paragraphe 2 qui a quitté le flux.

Dans l'exemple de droite, le paragraphe 2 a été mis en position `relative` avec `top` à -25 px et `left` à +10 px à l'aide du code suivant :

EXEMPLE 4.32 Positionnement relatif du paragraphe 2

```
p#p2 {
  position: relative;
  top: -25px;
  left: 10px;
}
```

Résultat, le paragraphe a été décalé par rapport à sa position initiale dans le flux : de 25 pixels vers le haut et de 10 pixels vers la droite. Ce décalage ne modifie en rien la position du paragraphe frère suivant (le paragraphe 3) qui agit comme si le paragraphe 2 était à sa position initiale dans le flux.

Application à un exemple concret plus complet

On vous l'accorde, les concepts de dimension et de positionnement introduits dans les sections précédentes restent assez abstraits. Pour y voir plus clair, intéressons-nous maintenant à un exemple concret d'application de ces concepts.

Imaginons que l'on veuille créer une page web contenant les éléments suivants : un en-tête pour y placer une bannière, un menu à gauche, un menu à droite et, au centre, le contenu réel de la page. On aimerait que la largeur des menus soit fixe, de même que la hauteur de la bannière. Sous forme de maquette, cela ressemblerait à la figure suivante.

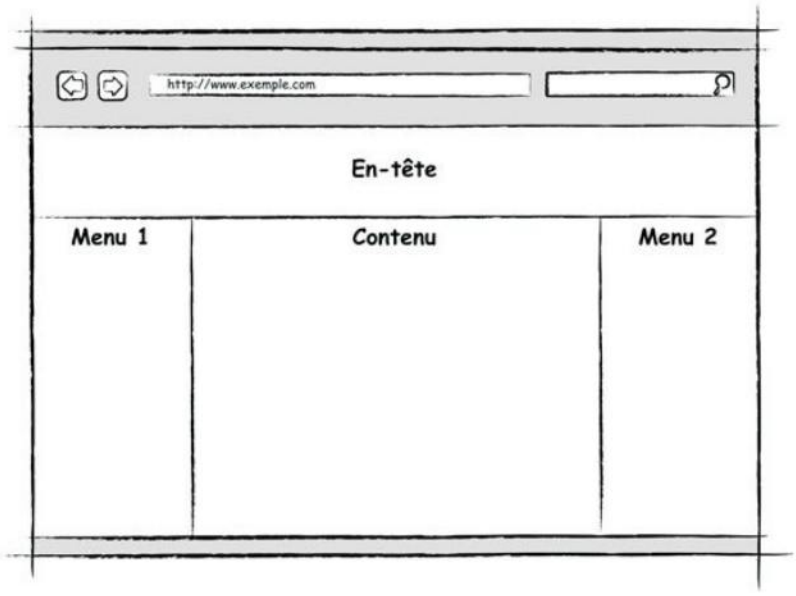


Figure 4–19
Maquette de notre page web d'exemple

Avant d'attaquer la CSS, écrivons d'abord le code HTML correspondant :

EXEMPLE 4.33 Code HTML de la page

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Ma page web</title>
  </head>
  <body>
    <header id="enTete">
      <p>En-tête de la page</p>
    </header>
    <aside id="menu1">
      <p>Ceci est le menu 1</p>
    </aside>
    <aside id="menu2">
      <p>Ceci est le menu 2</p>
    </aside>
    <section id="contenu">
      <h1>Contenu</h1>
      <p>Ceci est le contenu</p>
    </section>
  </body>
</html>
```

Pour mieux comprendre, nous avons coloré chacun des éléments à l'aide de cette CSS, dont les instructions sont placées dans l'en-tête de la page :

EXEMPLE 4.34 Mise en couleur des éléments

```
<head>
  <title>Ma page web</title>
  <style>
    header#enTete {
      background-color: red;
    }
    aside#menu1 {
      background-color: green;
    }
    aside#menu2 {
      background-color: yellow;
    }
    section#contenu {
      background-color: blue;
    }
  </style>
</head>
```

Le rendu dans un navigateur est le suivant.

Figure 4–20 Première ébauche de notre page web

On constate, et c'est normal, que tous les éléments se suivent les uns après les autres puisque, par défaut, ils font tous partie du flux courant.

Essayons maintenant de placer le premier menu à gauche et le deuxième menu à droite. Pour ce faire, on va les sortir du flux en leur attribuant une position absolue. Profitons-en pour leur donner leur largeur fixe. Nous ajoutons les instructions CSS suivantes :

EXEMPLE 4.35 Positionnement absolu des menus

```
aside#menu1, aside#menu2 {
  top: 0;
  width: 0;
  position: absolute;
}

aside#menu1 {
  background-color: green;
  left: 0;
}

aside#menu2 {
  background-color: yellow;
  right: 0;
}
```

Le résultat est maintenant le suivant.

Figure 4–21 Placement des menus

On aimerait maintenant que les menus se trouvent sous la bannière ; il suffit de les déplacer vers le bas à l'aide de la propriété `top`. De même, on aimerait que la bannière ait une hauteur fixe de 50 pixels et qu'elle soit « collée » aux bords de la page ; nous allons pour cela passer cet élément en position absolue. Lorsqu'il s'agit de toucher à des propriétés telles que hauteur et position, il est souvent plus facile de travailler en position absolue, afin de ne pas subir des effets de bords dus au positionnement automatique dans le flux courant. Nous allons donc écrire le code suivant :

EXEMPLE 4.36 Positionnement absolu de l'en-tête

```
header#enTete {
  background-color: red;
  height: 50px;
  position: absolute; (1)
  top: 0; (2)
  left: 0; (3)
  width: 100%; (4)
}
```

```
aside#menu1, aside#menu2 {
  top: 50px;
  width: 80px;
  position: absolute;
}
```

Grâce à ce code, nous sortons le `header` du flux courant (1), nous le plaçons tout en haut (2) à gauche (3) et lui demandons de prendre la totalité de la largeur (4). Le rendu est visible sur la figure suivante.

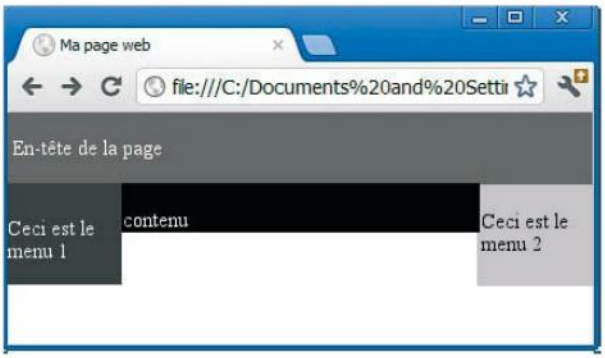


Figure 4–22
Positionnement de l'en-tête

Il reste un tout dernier problème à régler. Le contenu se trouve derrière les boîtes que nous avons placées en position absolue. Un moyen simple de résoudre ce problème consiste à ajouter des marges extérieures à la boîte de contenu : plus précisément, une marge en haut de taille égale à la hauteur de l'en-tête, et des marges à droite et à gauche de taille égale à la largeur des menus. Afin de ne pas « coller » la boîte de contenu aux boîtes qui l'entourent, nous allons augmenter ces trois marges de 10 pixels.

Nous allons donc écrire le code suivant :

EXEMPLE 4.37 Décalage du contenu

```
section#contenu {
  background-color: blue;
  margin-top: 60px;
  margin-left: 90px;
  margin-right: 90px;
}
```

La figure qui suit en présente le rendu.

Figure 4–23 Décalage du contenu

Chose étrange : la boîte de contenu n'est certes pas collée à ses boîtes adjacentes, mais l'espace à droite et à gauche est plus important que celui d'en haut ! Pourtant, nous avons ajouté exactement 10 pixels à chacune des trois marges… L'explication est simple : le body possède lui aussi des marges qu'il va falloir annuler (si vous reprenez les premiers exemples, vous verrez que les boîtes du flux ne sont pas collées au bord de la page). Pour annuler les marges du body, on va écrire le code suivant :

EXEMPLE 4.38 Ajustement du décalage du contenu

```
body {
  margin: 0;
  padding: 0;
}
```

Le rendu est cette fois celui espéré.

Figure 4–24
Ajustement du décalage du contenu

Nous terminons ici notre tour d'horizon des CSS. Nous avons volontairement omis de vous parler en détail de toutes les propriétés possibles, car elles sont pour la plupart triviales à utiliser. Il était préférable de se focaliser sur les quelques concepts difficiles à comprendre et à maîtriser.

À LIRE Ressources sur les CSS

Afin d'avoir un aperçu exhaustif de toutes les propriétés possibles et de leur utilisation, nous ne pouvons que vous conseiller de vous référer au site web suivant, qui contient une référence complète sur les CSS, agrémentée de nombreux exemples et de tutoriels :

- http://www.w3schools.com/

ainsi qu'à l'ouvrage de Raphaël Goetter :

📖 Raphaël Goetter, *CSS avancées*, Eyrolles, 2012

Dynamiser les pages web « à la volée » avec JavaScript

Les pages web que nous avons créées jusqu'à présent sont dépourvues de dynamisme et d'interactivité avec l'internaute. Une fois affichées par le navigateur, elles ne bougent plus. Leur aspect reste identique, tous leurs éléments restent à la même place et conservent leur apparence.

Or, dans certains cas, il serait intéressant de pouvoir modifier l'aspect de la page en fonction d'*événements*. Par exemple :

- faire apparaître un menu lorsque la souris survole un texte ou une image ;
- avoir des liens « lire la suite » qui, lorsqu'on clique dessus, affichent la suite du texte que l'on est en train de lire *dans la même page* ;
- valider un champ d'un formulaire lorsque l'utilisateur saisit sa valeur et afficher, le cas échéant, un message d'erreur dans le cas d'une saisie incorrecte, avant même que le formulaire ne soit soumis au serveur.

Le DHTML (*Dynamic HTML*) ajoute ce genre de dynamisme et d'interactivité à une page web. Il ne s'agit pas d'un langage à proprement parler, mais d'une combinaison de technologies.

Les événements

Le DHTML se base sur la notion d'*événements*. HTML prévoit en effet une série d'événements associés à ses éléments. Par exemple :

- `click` : se produit lorsqu'on clique sur un élément.
- `dblclick` : se produit lorsqu'on double-clique sur un élément.
- `mouseover` : se produit lorsqu'on passe la souris sur un élément.
- `change` : se produit lorsque la valeur d'un champ de formulaire change.

C'est à ces événements que l'on va réagir via du code qui changera l'aspect et le contenu de la page.

Langage de programmation de scripts

La deuxième technologie nécessaire pour dynamiser le HTML est le langage de programmation de scripts. Celui-ci est chargé d'adapter l'aspect et le contenu d'une page web, en réaction à des événements. On pourra à la fois modifier le HTML de la page (ajouter des éléments, en retirer, changer leur contenu) et modifier les CSS du document.

Les deux principaux langages qui peuvent être utilisés sont le JavaScript et le VBScript. Ce dernier n'étant utilisable que sous Internet Explorer, c'est tout naturellement Java-

Script qui a rencontré les faveurs d'un plus grand nombre de développeurs, notamment car il est compatible avec tous les navigateurs.

Pour l'essentiel, ce livre se concentre sur un dynamisme qui se déploie côté serveur, mais nous retrouverons JavaScript en appoint de Django dans le dernier chapitre, lorsqu'il sera question de répartir un peu d'interactivité aussi du côté client. Il est en effet capital d'ores et déjà de noter que ces langages de scripts s'exécutent *côté client*, c'est-à-dire dans le navigateur et non dans le serveur.

La syntaxe de JavaScript est proche de celle de Java, mais plus simple (n'allez jamais dire cela à un programmeur Java susceptible !). Nous n'allons pas étudier en détail cette syntaxe, les exemples que nous vous proposerons seront suffisants pour comprendre.

Un premier exemple de DHTML

Imaginons que nous ayons dans notre page HTML un article composé de deux paragraphes. Le premier représente un petit texte « d'accroche », le second est l'article au complet. Par défaut, on veut que seul le premier paragraphe soit affiché et que l'autre ne soit visible que lorsqu'on clique sur un lien « lire l'article complet ».

EXEMPLE 4.39 Code HTML d'un article de deux paragraphes

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Ma page web</title>
  </head>
  <body>
    <article id="contenu">
      <p id="accroche">Le Président est mort (<a href="#" 1>lire l'article
      complet</a>)</p>
      <p id="articleComplet">Hier soir, après avoir visionné la dernière
      comédie d'Alain Chabat, le Président a déclaré
      « Je suis mort de rire ! »</p>
    </article>
  </body>
</html>
```

Le rendu est, sans surprise, le suivant :

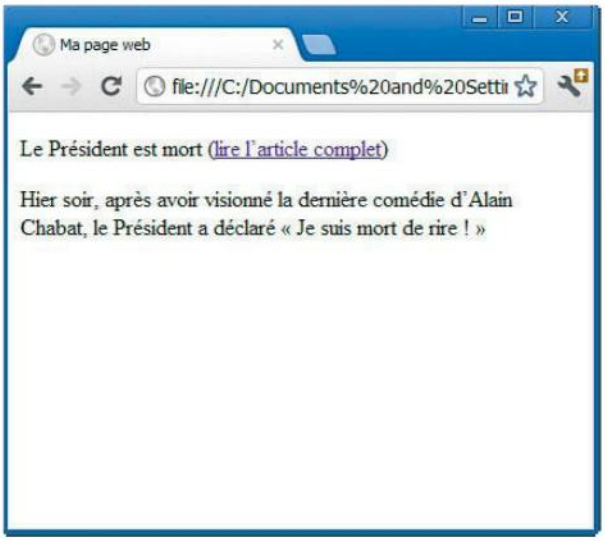


Figure 4–25
Un article composé de deux paragraphes et un lien

Notez que, comme cible pour le lien, nous avons mis `#` (1). Cela permet de rester sur la page lorsqu'on clique sur le lien.

Afin que le deuxième paragraphe soit par défaut invisible, nous allons ajouter le code CSS suivant :

EXEMPLE 4.40 Masquage du paragraphe

```
p#articleComplet {
  display: none;
}
```

Passons maintenant à l'écriture du code JavaScript. Au clic de la souris sur le lien, nous allons simplement changer le code CSS du paragraphe (et sa propriété `display`) pour le rendre à nouveau visible. Le code sera le suivant :

EXEMPLE 4.41 Code JavaScript pour réafficher le paragraphe

```
document.getElementById('articleComplet').style.display='block'
```

Ce code va chercher l'élément dont l'`id` est `articleComplet` et modifie sa propriété CSS `display`. Nous assignons à cette propriété la valeur `block`, car un paragraphe est un élément de ce type.

Où placer ce code ? Une solution est de le placer directement au niveau du lien :

EXEMPLE 4.42 Masquage du paragraphe

```
<a href="#" onclick=
"document.getElementById('articleComplet').style.display='block'">lire
l'article complet</a>
```

Le code est placé dans l'attribut `onclick` de l'élément `a`. Il se déclenchera lors du clic sur le lien.

Le code peut aussi être placé dans une balise `<script>` située dans l'en-tête du document. Dans ce cas, il devra être écrit dans une *fonction*, qui sera appelée dans l'événement `onclick` :

EXEMPLE 4.43 Le code est mis dans l'en-tête

```
<script type="text/javascript">
  function afficherArticleComplet()
  {
    document.getElementById('articleComplet').style.display='block'
  }
</script>
```

EXEMPLE 4.44 Appel de la fonction au niveau du lien

```
<a href="#" onclick="afficherArticleComplet()">lire l'article complet</
a>
```

Une autre possibilité est de placer le code dans un fichier séparé d'extension `.js`, que l'on liera avec notre HTML à l'aide de la balise suivante placée dans l'en-tête :

SYNTAXE. Inclusion d'un fichier .js dans le HTML

```
<head>
  <script type="text/javascript" src="monJavaScript.js"></script>
</head>
```

jQuery et les frameworks JavaScript

Aux prémisses du DHTML, un casse-tête venait entraver le travail des développeurs : d'un navigateur à l'autre, le code JavaScript n'était pas tout à fait le même pour arriver à un résultat identique. Par exemple, pour récupérer un élément à partir de son `id`, il existait plusieurs méthodes :

EXEMPLE 4.45 Récupération d'un élément par son id

```
/* Méthode standard */
document.getElementById(ID)

/* Internet Explorer */
document.all(ID)

/* Netscape */
eval('document.' + ID)
```

Il n'était donc pas rare de voir du code ressemblant à « If Internet Explorer Alors, If Firefox Alors, etc. ».

Afin de faciliter la vie des développeurs, sont apparus des *frameworks* JavaScript dont l'objectif était de cacher cette complexité. Le framework offrait alors ses propres fonctions, lesquelles se chargeaient d'écrire le bon code pour le bon navigateur.

Le rôle de ces frameworks s'est très vite étendu pour compiler et offrir des fonctions nouvelles souvent utilisées par les développeurs.

Aujourd'hui, un des frameworks les plus connus et utilisés est jQuery. Il permet entre autres de :

- simplifier et enrichir l'accès aux éléments de la page ;
- simplifier et enrichir la gestion des événements ;
- appliquer des effets et des animations sur les éléments d'une page web ;
- simplifier l'application de styles à des éléments.

Écrit en JavaScript, jQuery est donc devenu *le* framework de DHTML par excellence.

Concrètement, jQuery se présente sous la forme d'un fichier JavaScript comprenant une série de fonctions. Ce fichier est téléchargeable sur le site officiel du framework. Une fois téléchargé, pour l'utiliser, il suffit de l'inclure dans l'en-tête du fichier HTML :

SYNTAXE. Inclusion du framework jQuery

```
<head>
  <script type="text/javascript" src="jquery-1.7.2.min.js"></script>
</head>
```

Une fois le framework inclus, celui-ci s'utilise notamment via la fonction `$`, qui sélectionne des éléments de la page web :

SYNTAXE. Sélection d'éléments en jQuery

```
$('*')      /* Sélectionne tous les éléments */
$('#bla')   /* Sélectionne l'élément portant l'id « bla » */
$('.fr')    /* Sélectionne tous les éléments de classe « fr » */
```

On le voit, la sélection d'élément s'opère de la même manière que dans les CSS.

Une fois les éléments sélectionnés, on peut les manipuler :

SYNTAXE. Manipulation d'éléments en jQuery

```
$('#bla').hide() /* Cache l'élément */
$('#bla').fadeIn() /* Affiche l'élément avec un effet de fondu */
$('#bla').addClass() /* Ajoute à l'élément une classe CSS */
$('#bla').outerHeight() /* Récupère la hauteur totale d'un élément */
$('#bla').css("background-color","yellow") /* Change la couleur de
                                              fond de l'élément */
```

De nombreuses autres possibilités sont offertes par jQuery. Nous ne les relaterons pas ici, l'objectif étant de vous offrir juste un petit aperçu de cette technologie et de son rôle dans le cadre du développement web.

À LIRE jQuery

Si vous désirez en apprendre plus sur jQuery :

📖 Éric Sarrion, *jQuery 1.7 et jQuery UI*, 2e édition, Eyrolles, 2012

Ai-je bien compris ?

- Quel rapport peut-on faire entre le modèle MVC et l'ensemble HTML-CSS-JavaScript ?
- Quelles différences y a-t-il entre jQuery, DHTML et JavaScript ?
- En CSS, quelle est la différence entre les positionnements « absolu » et « relatif » ?

5

Mise en application : un site web inspiré de Facebook

*Ce chapitre présente le projet de développement qui servira de base pour tous les autres chapitres : Trombinoscoop. Tout projet web démarre par la rédaction de ses cas d'utilisation (*use cases*) et la conception de ses* wireframes*, et d'une base de données relationnelle côté serveur. Quelques rappels théoriques sur le modèle relationnel se révèlent indispensables. Django exige que le modèle de données soit réalisé via un diagramme de classes, donc dans une perspective purement orientée objet. Par conséquent, quelques lignes sont consacrées à l'épineux problème de la mise en correspondance entre le modèle relationnel et le modèle objet.*

SOMMAIRE

- Trombinoscoop : le projet auquel ce livre va se consacrer
- Cas d'utilisation et maquettage (*wireframes*)
- Rappels théoriques sur les bases de données relationnelles et différence avec la modélisation orientée objet
- Diagramme de classes de Trombinoscoop

Ce chapitre décrit le projet de site web que nous réaliserons à partir de Django. Son développement sera exposé sous forme de tutoriel qui servira de fil conducteur à l'apprentissage du framework Django. Il sera construit progressivement, à mesure que de nouvelles notions seront introduites. Il permettra de comprendre par la pratique et concrètement chaque aspect du framework.

Comme pour tout projet de développement informatique, nous allons également passer par une phase préalable d'*analyse*. Décrire brièvement les fonctionnalités que l'on désire voir sur le site web ne suffit pas. Avant d'écrire la moindre ligne de code, il est important de se poser un instant, prendre un peu de recul, réfléchir à ce que l'on va réellement développer et à certains éléments de structure du code.

C'est pourquoi ce chapitre se propose également de vous exposer brièvement quelques méthodes d'analyse. Rassurez-vous, cet exposé sera plus que sommaire, à mille lieues de la complexité que l'on trouve parfois dans certains projets réels. Nous allons nous limiter à décrire et exploiter au plus pressé trois éléments d'analyse :

- Les *use cases* du site web (ou en français « cas d'utilisation ») entrevus dans le chapitre deux. Ils décrivent sommairement les fonctionnalités principales du site et délimitent l'étendue du projet. À quoi va bien pouvoir servir ce site ?
- Les *wireframes* du site web, plus en rapport avec la conception graphique. En d'autres mots, on va décrire les différents écrans du site web et leur enchaînement (« lorsque je clique sur tel bouton, j'arrive sur tel écran »).
- Le *modèle objet* de la base de données. Il est en effet déjà utile à ce stade de réfléchir sur les données que l'on voudra sauvegarder et traiter dans notre site web.

Les *use cases*, les *wireframes* et le *modèle objet* ne doivent pas être figés : ils peuvent être modifiés et évoluer au gré des développements.

Notre site web : Trombinoscoop

Notre projet trouve ses origines (fictives) à l'Université libre de Bruxelles, qui avait lancé il y a quelques années un carnet d'adresses électronique. Ce carnet reprenait les coordonnées de l'ensemble de la communauté universitaire et permettait d'y effectuer des recherches.

En découvrant ce carnet web un peu statique, Marc Van Saltberg, étudiant en informatique de l'Université, eut une idée géniale pour le rendre plus personnel et plus vivant : pourquoi ne pas créer un site web où chaque personne gèrerait elle-même son profil, pourrait indiquer quels sont ses amis et publier des messages (à destination de tous ou uniquement de ses amis).

Une fois développé, le site de Marc connut un tel succès qu'il décida de le rendre disponible à d'autres universités, puis également au monde non universitaire. On connaît la suite de cette fameuse histoire belge : le succès du site fut planétaire et, quelques années après, il comptait des millions d'utilisateurs (ce qui permit à Marc de créer une entreprise florissante, qu'il décida même de coter en Bourse avec une valorisation de plusieurs milliards d'euros).

> **PRÉCISION Toute ressemblance...**
>
> Oui, c'est bien l'histoire de Facebook que nous venons de vous conter. Nous avons juste changé les noms de personnes et de lieux...

C'est ce site web, dans sa première version basique et élémentaire, que nous allons réaliser dans les chapitres suivants. Avant toute chose, Marc, tout geek qu'il est, a dû procéder à un peu d'analyse.

Les cas d'utilisation

Il est important de bien définir les premières fonctionnalités que l'on veut offrir. Par la suite, d'autres fonctionnalités pourront enrichir le site, mais la première analyse par cas d'utilisation ne doit pas couvrir tous les cas possibles au départ du développement.

> **DÉFINITION Diagrammes de cas d'utilisation**
>
> Il s'agit de diagrammes très simples, reprenant simplement un intitulé court pour chacune des fonctionnalités et une ligne vers un utilisateur pour indiquer qui va faire usage de cette fonction.

La figure 5.1, page suivante, expose les différents objectifs de notre projet Trombinoscoop.

Nous avons identifié neuf *use cases*, tous utilisés exclusivement par un même acteur que l'on dénommera simplement *l'internaute*.

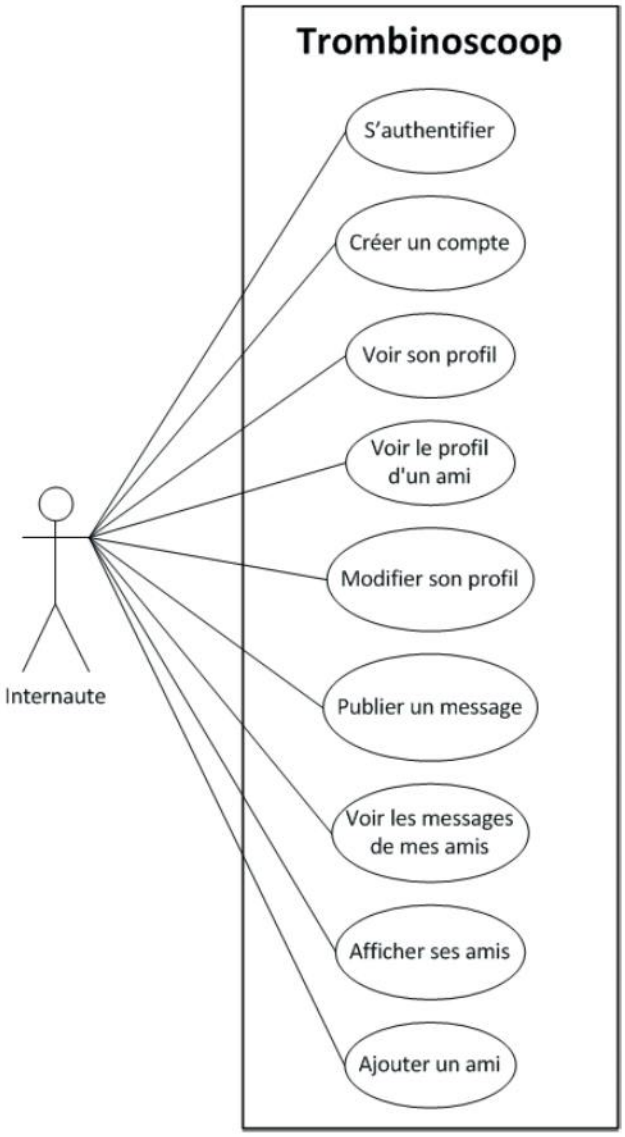


Figure 5–1
Cas d'utilisation de Trombinoscoop

Maquette du site : les wireframes

L'aspect du site et sa navigabilité peuvent s'élaborer assez tôt en identifiant l'organisation visuelle et le mode d'emploi de chaque page. Il faut s'interroger sur les informations accessibles sur les pages, ainsi que sur les fonctionnalités rendues disponibles pour chacune d'entre elles. Des priorités tant entre ces informations qu'entre ces fonctionnalités devront être établies. Pour cela, il est souhaitable d'envisager certains scénarios d'utilisation pour concevoir et tester au mieux l'aspect et le mode d'utilisation de la page.

Ces étapes nécessiteront la réalisation d'une succession de maquettes visuelles appelées *wireframes* en anglais.

> **DÉFINITION Wireframes**
>
> Il s'agit de dessins *grossiers* représentant sommairement chacun des écrans du futur site web. Les wireframes doivent permettre au premier coup d'œil de se faire une idée du contenu et des fonctionnalités de chaque page, ainsi que des liens qui les unissent.

Pour imaginer les écrans de Trombinoscoop, il faut partir des cas d'utilisation que nous venons de définir. Tous nos cas doivent être couverts. Attention : plusieurs écrans seront parfois nécessaires pour un seul cas d'utilisation et, inversement, un écran peut très bien proposer plusieurs cas d'utilisation.

Pour imaginer les écrans, il n'est malheureusement pas de recette miracle. Seules l'imagination, l'inspiration et l'expérience vous amèneront à un résultat probant.

Pour Trombinoscoop, nous sommes arrivés à la définition de six écrans :

- L'écran d'authentification est le premier écran à apparaître lorsqu'on arrive sur le site.
- Un écran sert à se créer un compte si l'on n'en possède pas encore.
- L'écran d'accueil est l'écran principal du site. Il affiche un résumé du profil de l'utilisateur connecté, les messages publiés par ses amis et la liste de ses amis. C'est à partir de cet écran que l'utilisateur pourra publier de nouveaux messages. Cet écran offre également des liens permettant d'ajouter un ami, consulter ou modifier son profil.
- Un écran sert à l'utilisateur pour modifier son profil.
- Un écran affiche le profil de l'utilisateur.
- Un écran composé d'un seul champ permet d'ajouter quelqu'un à sa liste d'amis sur la base de son courriel.

Une fois ces écrans imaginés, il reste à les coucher sur papier sous forme de wireframes. Un crayon et un papier suffisent amplement. Il n'est nul besoin d'utiliser des logiciels graphiques complexes ou d'être un graphiste hors pair : un assemblage primitif de carrés et de rectangles devrait faire l'affaire.

L'écran d'authentification

Cet écran est le premier qui s'affiche lorsqu'on arrive sur le site web Trombinoscoop. Il contient simplement un formulaire de login et un lien vers la page de création de compte pour les visiteurs qui ne possèderaient pas encore de compte (voir figure 5-2).

L'écran de création d'un compte

Cet écran collecte diverses informations personnelles, différentes selon que l'on crée un compte « étudiant » ou « employé ». Du code JavaScript devra être écrit pour afficher les bons champs à compléter (voir figure 5-3).

L'écran d'accueil

Il s'agit de l'écran principal du site. Il reprend dans l'en-tête un bref résumé du profil de l'utilisateur, au centre l'ensemble des messages des amis, et dans la colonne de droite la liste des amis.

On voit clairement que cet écran *réalise* plusieurs cas d'utilisation à la fois ; il s'agit d'un exemple parfait de *non-bijectivité* entre « cas d'utilisation » et « écran » (voir figure 5-4).

L'écran de modification du profil

Cet écran, très proche de celui de création d'un profil, permet simplement de modifier les informations personnelles sur l'internaute (voir figure 5-5).

L'écran d'affichage d'un profil

Cet écran affiche soit le profil de l'utilisateur connecté, soit celui de l'un de ses amis (voir figure 5-6).

L'écran d'ajout d'un ami

Cet écran très simple sert à ajouter un ami à partir de son adresse de courriel (voir figure 5-7).

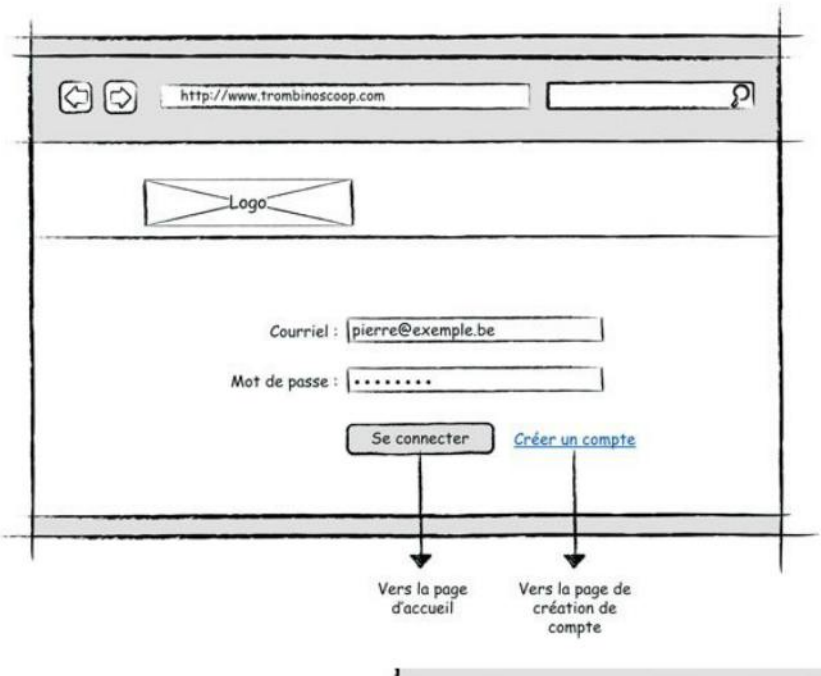


Figure 5–2
Wireframe de l'écran d'authentification

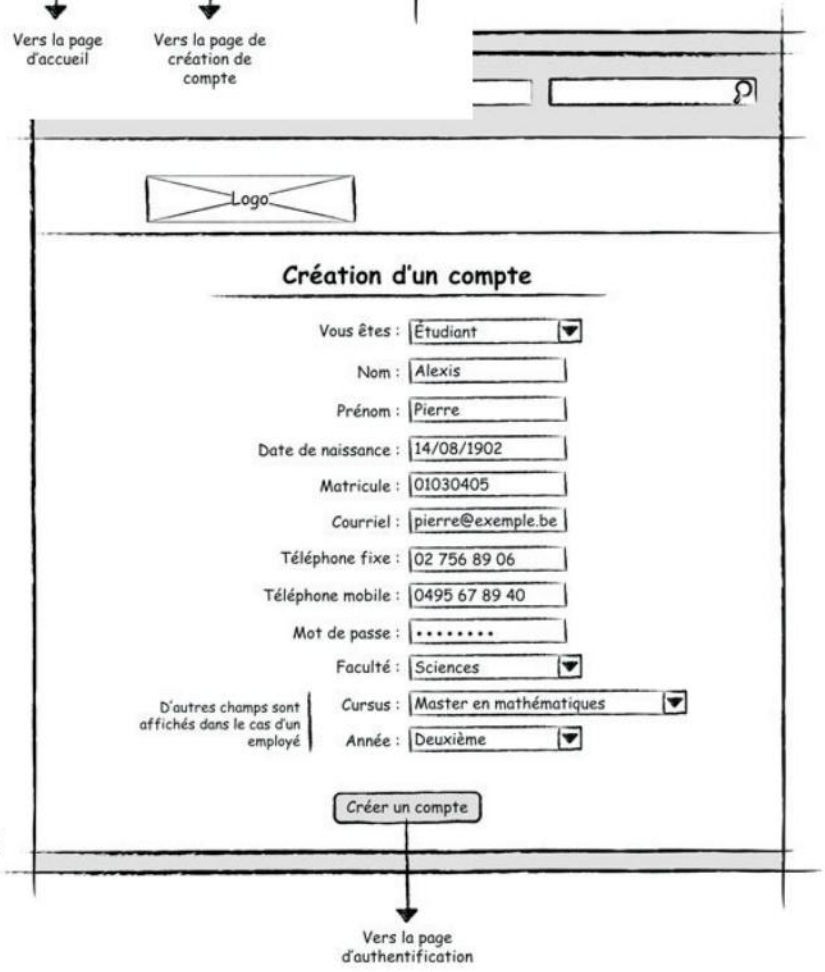


Figure 5–3
Wireframe de l'écran de création d'un compte

Figure 5–3
Wireframe de l'écran d'accueil

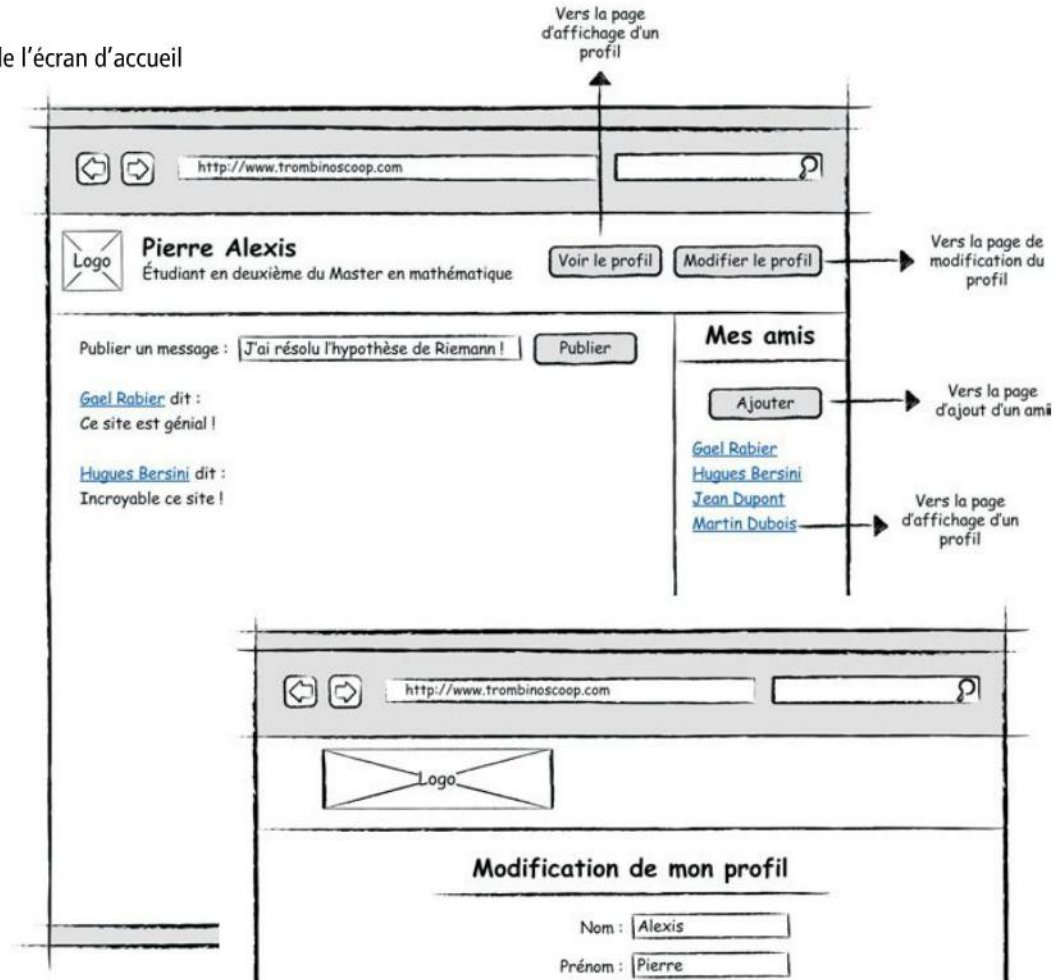
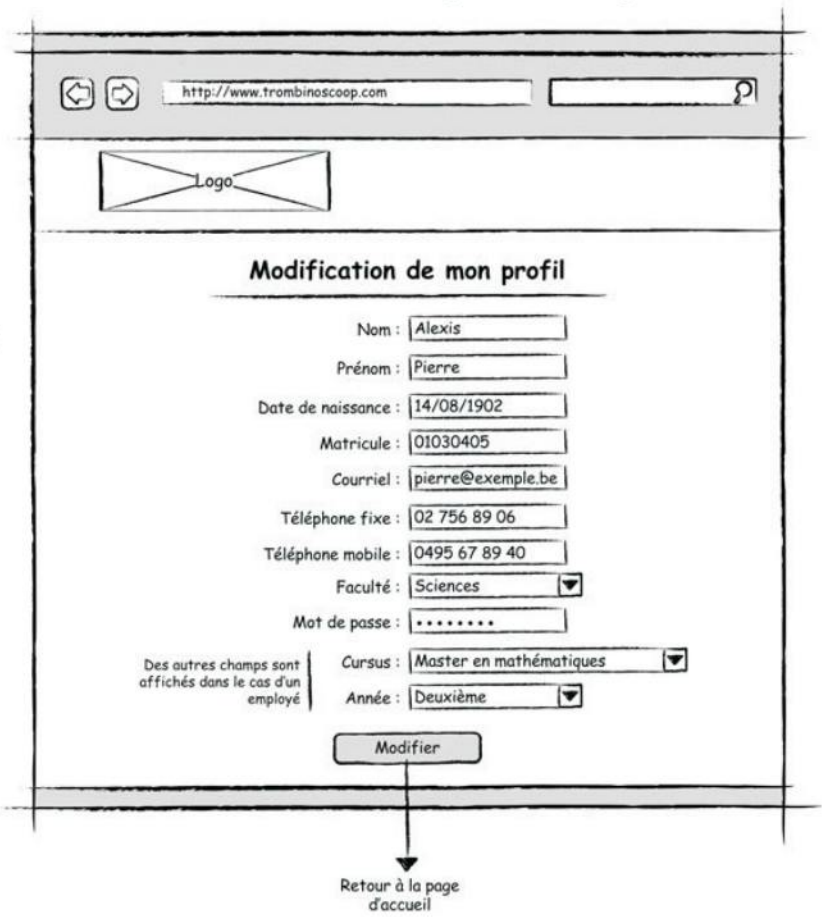


Figure 5–5
Wireframe de l'écran de modification du profil



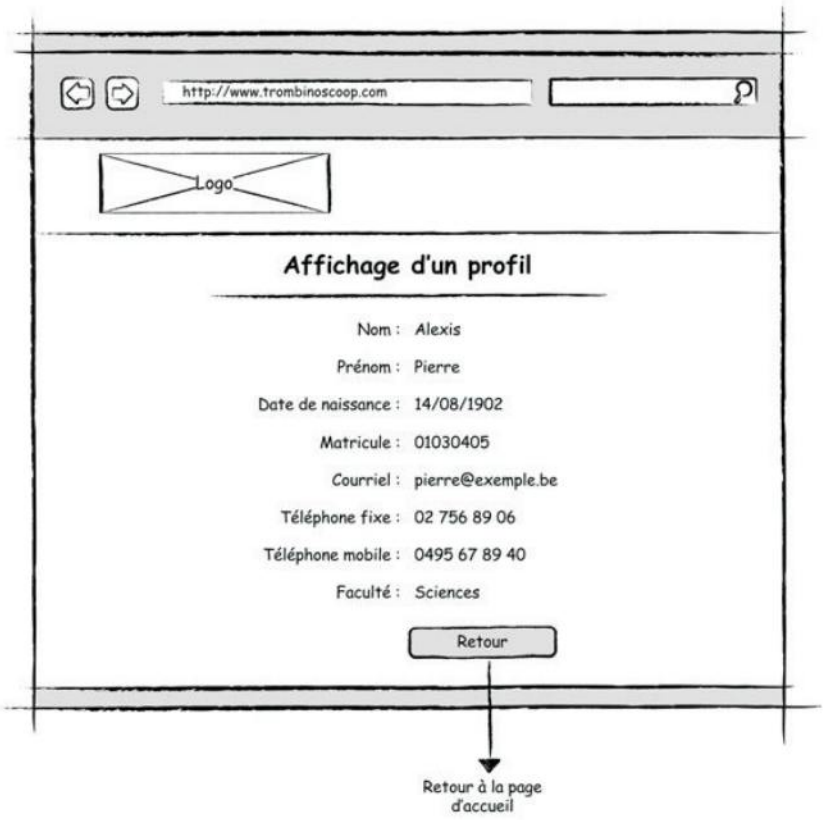


Figure 5–4
Wireframe de l'écran d'affichage d'un profil

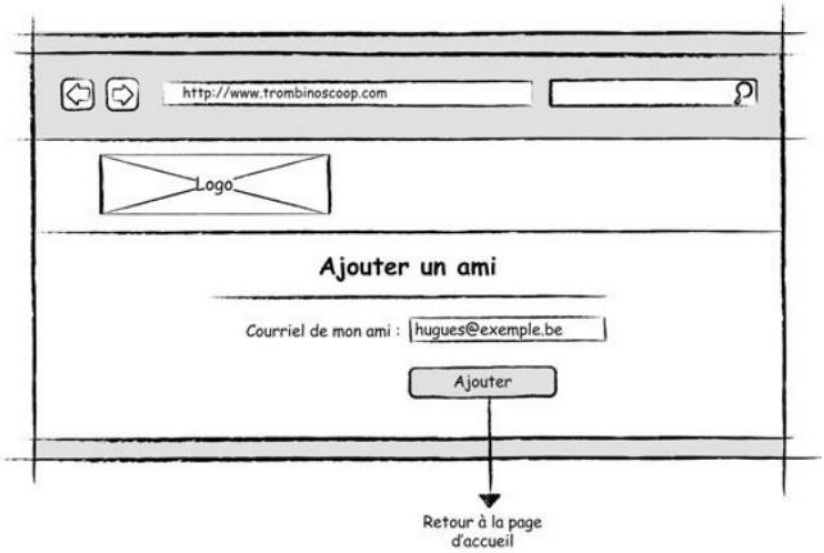


Figure 5–7
Wireframe de l'écran d'ajout d'un ami

Scénario complet du site

Les wireframes permettent de comprendre en un coup d'œil les enchaînements entre les écrans. C'est pourquoi nous avons réalisé un dessin reprenant tous les écrans du site et leurs liens, avec, pour chaque lien, l'événement qui est à l'origine du passage d'un écran à l'autre.

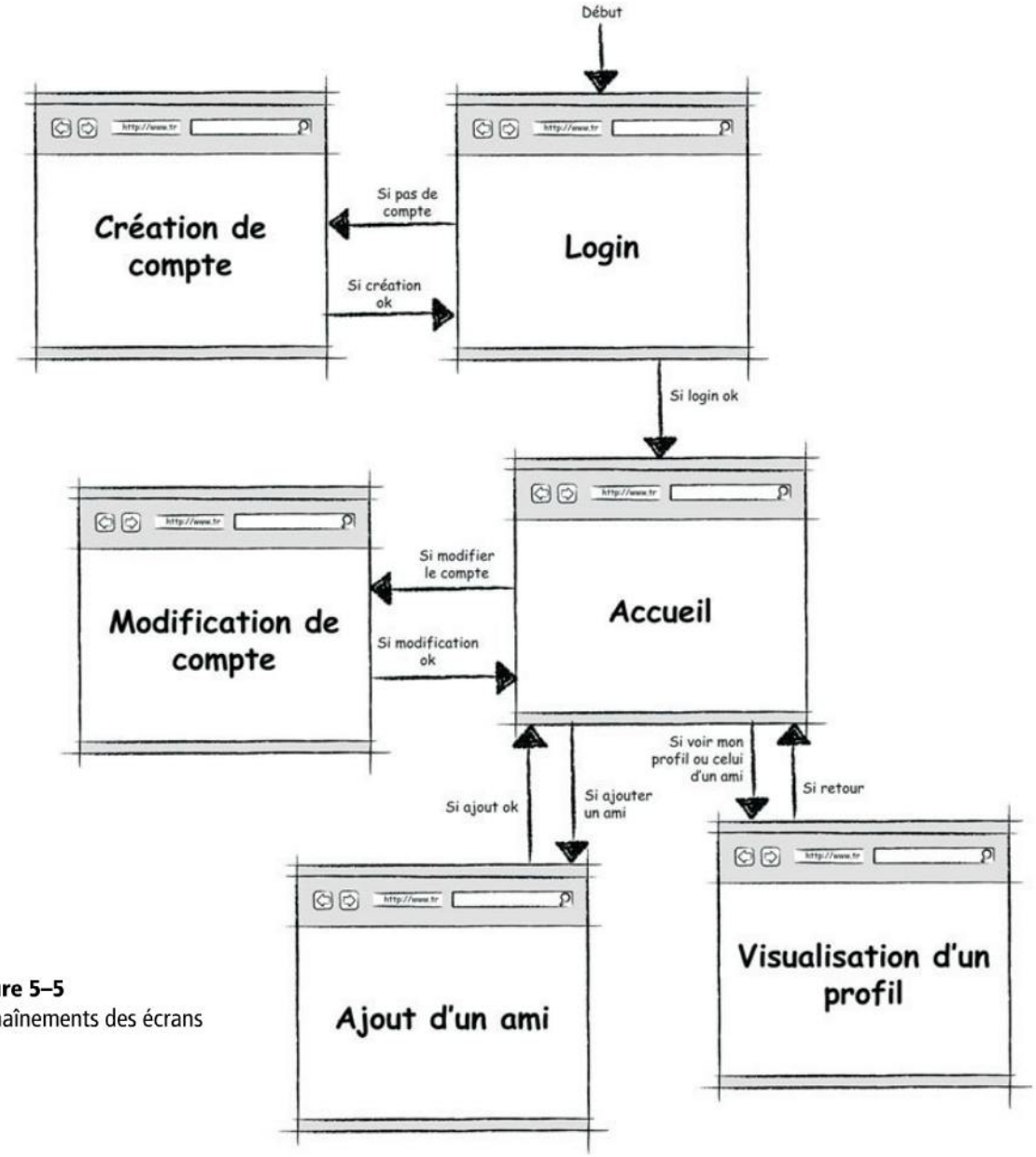


Figure 5–5
Enchaînements des écrans

EN PRATIQUE Et l'aspect réel du site ? Un aperçu du design web...

Les wireframes sont plutôt sommaires et austères. C'est normal : leur vocation est de poser les premières briques du site et de son architecture. Elles serviront essentiellement à la production des pages HTML. Elles ne sont pas là pour représenter au pixel près l'aspect final du site web.
L'objectif de ce livre n'étant pas de former des futurs Andy Warhol, les aspects graphiques seront un peu laissés pour compte ; sur Trombinoscoop, le graphisme se résume à sa plus simple expression. Néanmoins, les wireframes ne sont pas suffisantes pour définir l'intégralité du *look and feel* du site web. Vous pouvez forger une identité visuelle au site web, ce petit « je ne sais quoi » qui va rendre votre Trombinoscoop unique et incontournable. Couleurs, polices, dispositions précises, logo de l'entreprise, etc. devront être réfléchis et définis. Cette tâche sera généralement dévolue à un graphiste qui réalisera des maquettes beaucoup plus élaborées du site web, représentant le rendu exact que doivent avoir les pages. Cette tâche parfois négligée est en réalité assez ardue. Il faut parvenir à l'équilibre ô combien délicat entre sobriété, esthétique, facilité d'emploi et exhaustivité. L'identité visuelle du site doit se maintenir pendant la navigation.
À cette fin, il est capital que le site soit évalué par les futurs utilisateurs dès ses premières ébauches, de manière à corriger ou réorienter au plus vite sa conception. Toutes les nouvelles méthodologies de développement informatique (telle la programmation agile) exigent de fonctionner de manière très itérative et très réactive, en gardant toujours le futur utilisateur dans la boucle. Il n'est plus question de rémunérer des analystes pendant deux ans, travaillant seuls dans leur coin, sans que le client ne soit tenu informé des étapes successives et nécessairement testables du logiciel. Le site devra être en évaluation permanente et très rapidement expérimenté sur tous les principaux navigateurs et systèmes d'exploitation ainsi que pour différentes tailles d'écran. Le site doit être très facile d'emploi et certaines règles de conception (comme les trois clics maximum pour dénicher une information) gagnent à être suivies. De nombreux ouvrages existent, entièrement dédiés à cette matière de plus en plus exigeante.

▸ http://www.editions-eyrolles.com/Collection/12270/design-web

Modèle de données et petit rappel sur les bases de données relationnelles

Une des premières missions, très sensible, qui échoit aux informaticiens est de concevoir la base de données qui stockera les données du site web.

Les bases de données relationnelles sont le moyen le plus répandu pour stocker l'information de manière permanente. Les programmes, quel que soit leur langage, doivent s'interfacer avec ce mode de stockage à un moment ou un autre de leur exécution pour accéder aux informations ou en entreposer de nouvelles.

Les raisons de ce quasi-monopole sont nombreuses.

- Tout d'abord, à l'instar de l'approche orientée objet, les données sont concentrées dans des structures de type « objet ». Les informations à stocker le sont en tant que valeurs d'attributs, regroupées dans des structures appelées *tables*, très proches en l'esprit des « classes » de la programmation objet. Chaque instance de table, obtenue

en fixant les valeurs d'attributs pour un cas donné, et qui donnerait naissance à un objet dans le cas de la classe, donne ici naissance à un *enregistrement*. Nous parlerons des différences essentielles que présentent tables et classes dans la suite du chapitre.

- Une deuxième raison est l'existence d'un mode d'organisation de ces tables, dit *relationnel*, dont la qualité première est d'éviter d'avoir à reproduire la même information plusieurs fois. On imagine aisément tous les problèmes liés à la duplication des informations : mises à jour plus pénibles et erreurs d'encodage plus probables. Chaque table se doit de coder un et un seul concept donné, et il est nécessaire de séparer l'encodage des informations dans des tables distinctes mais conservant toutefois des relations qui permettent, à partir de l'une d'entre elles, de retrouver les informations de l'autre.

EN PRATIQUE Différence entre liens entre tables et liens entre classes

Nous verrons que l'existence de ces relations entre les tables, conceptuellement proches des liens d'association entre les classes, intensifie davantage la ressemblance avec l'approche objet dont Python est un digne représentant. C'est pourtant dans le mécanisme qui concrétise ces relations que réside toute la différence entre les bases de données relationnelles et la manière dont les objets existent et se réfèrent entre eux dans la mémoire de l'ordinateur. Aujourd'hui encore, la façon différente dont s'associent les tables dans une base de données relationnelle et les classes dans un diagramme UML donne des cheveux gris à des légions d'informaticiens.

- Une troisième raison est la quantité importante de solutions techniques disponibles aujourd'hui, pour gérer de façon automatique des problèmes aussi critiques que les sauvegardes automatisées, les accès concurrentiels (quand plusieurs accès doivent s'opérer simultanément), les accès sécurisés et fiabilisés, les accès de type transactionnel (quand une étape défaillante dans une succession d'opérations rend caduques toutes les opérations effectuées jusque-là), le stockage réparti sur plusieurs ordinateurs, etc.

Nous allons par la suite décrire en détail la base de données sur laquelle reposera le site Trombinoscoop : les tables et les différentes relations qui les lient. Nous en profiterons pour glisser quelques rappels essentiels à la bonne pratique des bases de données relationnelles.

Clé primaire et clé étrangère

Relation 1-n

Débutons par la table `Employé` que nous voyons illustrée ci-après.

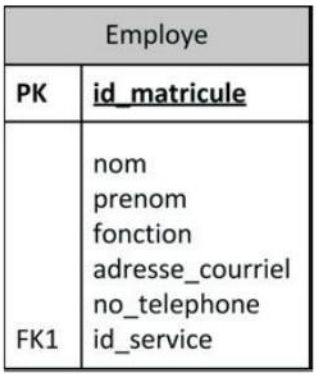


Figure 5–6
Table « Employé »

Le premier attribut de la table, et le plus important, est appelé la *clé primaire*. Tout employé de l'Université libre de Bruxelles possède un matricule qui permet de l'identifier. On conçoit l'existence d'un tel attribut dans une base de données. Ni le nom, ni le prénom, ni l'adresse, ni l'âge ne permettrait l'accès unique à un employé donné. C'est par l'attribut clé primaire que l'on parvient à tous les enregistrements désirés et eux seuls. Le système de gestion de la base de données sait, à partir d'une valeur donnée unique de cet attribut, retrouver l'enregistrement unique qui lui correspond. Aucun membre de notre université ne partage son matricule avec un autre.

DÉFINITION La clé primaire

La clé primaire d'une table est l'attribut qui permet d'accéder, de manière unique et sans aucune équivoque, à un des enregistrements de la table. La valeur de cet attribut doit être distincte pour chaque enregistrement, une valeur entière étant ce qu'il y a de plus courant. Parfois, il n'existe pas d'attribut « naturel » qui peut faire office de clé primaire. Dans ce cas, on en ajoute un, artificiel, que l'on nomme généralement `id`. Il s'agit alors d'un champ « technique » qui ne correspond à rien dans la réalité.

Chaque employé de l'université fait partie d'un service. Chaque service est composé d'un nombre important d'employés et se caractérise lui-même par un ensemble d'attributs qui lui sont propres : la dénomination du service, le nombre de ses employés, sa location géographique (son campus), la faculté à laquelle il appartient, etc.

EN PRATIQUE Pas de duplication d'information

Bien sûr, il serait théoriquement nécessaire d'ajouter toutes les informations du service qui l'héberge pour chaque employé, mais on conçoit bien le ridicule qu'il y a à répéter ces mêmes informations pour tous les employés d'un même service. Et c'est sans compter les multiples sources d'erreurs que la répétition de ces mêmes informations risque de causer. Rappelez-vous la doctrine première de tout informaticien qui se respecte : séparer les informations, de manière à ce que l'on puisse affecter une partie de celles-ci sans en affecter une autre. Il serait stupide, si le service venait à déménager ou à changer de nom, qu'il faille répercuter ce changement sur tous les employés de ce service.

La solution, illustrée dans la figure qui suit, est de recourir à une autre table pour y stocker ce qui ne concerne que les services et d'établir une relation « 1-n » avec la table `Employe` (un employé appartient à 1 service et chaque service englobe n employés). Dans le modèle dit « relationnel », les informations à stocker sont regroupées selon leur nature dans des tables mises en relation par un mécanisme de jointure de clés : de la clé primaire (PK sur la figure, ou *Primary Key*) à la clé étrangère (FK ou *Foreign Key*).

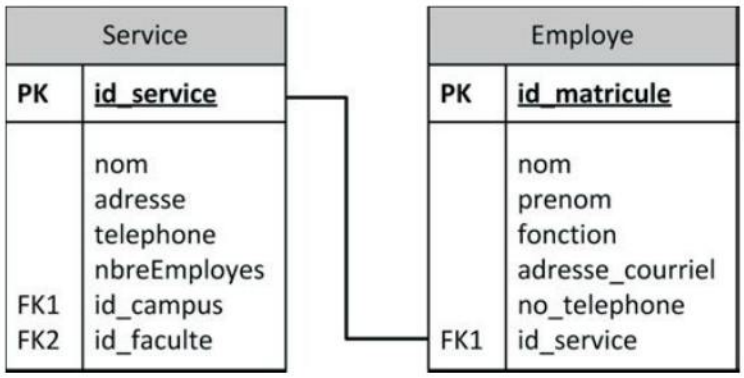


Figure 5–7
Tables « Employé » et « Service »

Ainsi, un des attributs de l'employé est la clé étrangère du service qui ne peut prendre que les seules valeurs prises par la clé primaire de ces mêmes services. Pour obtenir les informations du service à partir de l'employé, il suffit de connaître la valeur de sa clé étrangère et de se référer à la table `Service`. De la sorte, chaque service n'apparaît plus qu'une seule fois même s'il est partagé par de nombreux employés. C'est le lien (qui apparaît dans le graphique) entre la clé primaire d'une table et la clé étrangère de l'autre qui indique, pour chaque enregistrement de la deuxième table (chaque employé), quel est son correspondant unique dans la première (son unique service).

Relation 1-1

Même si cette « division du travail » coule de source dans le cas d'une relation « 1-n » entre les deux tables, une semblable division peut également résulter d'une relation « 1-1 » entre deux tables. Ainsi imaginons que chaque employé occupe un seul bureau et qu'un bureau, lorsqu'il est occupé, ne l'est que par un seul employé. Bien qu'il s'agisse d'une relation « 1-1 », garder séparées les deux tables et les informations qu'elles contiennent témoigne d'une bonne pratique de conception. En effet, les notions de bureau et d'employé restent bien distinctes et il est logique de séparer les informations relatives à l'un et à l'autre.

A priori, la clé étrangère pourrait cette fois se trouver dans l'une ou l'autre des deux tables. C'est la nature du problème qui choisira l'emplacement à privilégier. On peut imaginer que, s'il est impossible pour un employé de se trouver sans bureau, l'inverse n'est pas vrai ; un bureau pourrait dès lors rester inoccupé. Dans ce cas, on conçoit qu'il est préférable de mettre la clé étrangère du bureau dans la table `Employe` plutôt que l'inverse.

Relation n-n

La chose se corse un peu avec l'introduction de la table `Campus` et de son lien avec la table `Faculté` (voir figure suivante). En effet, une faculté peut se trouver sur plusieurs campus et un campus héberger plusieurs facultés. Si nous nous limitons à ces deux tables, comment résoudre l'emplacement de la clé étrangère ? Elle devrait théoriquement figurer dans les deux tables.

Plaçons-nous du côté `Faculté`. La présence de plusieurs facultés sur le même campus « 1 » se fait par les enregistrements suivants : 1-1, 2-1, 3-1, 4-2, 5-2 (voir tableau 5-1). Malheureusement, cette manière de faire ne permet pas de dire que la faculté « 1 » se trouve aussi sur le campus « 2 ». Le même problème se poserait du côté de la table `Campus`.

Tableau 5–1 Relation Faculté-Campus

Id_primaire_faculté	Id_étrangère_campus
1	1
2	1
3	1
4	2
5	2

La solution au problème consiste en l'addition d'une table dite de « jonction » (voir figure suivante). Elle joint les facultés aux campus (et réciproquement) par l'entremise de deux relations « 1-n », l'une entre les facultés et la table de jonction et l'autre entre les campus et la table de jonction. Cette table contient les enregistrements repris dans le tableau 5-2 qui situent les facultés « 1 » et « 2 » sur le campus « 1 », mais aussi sur le campus « 2 ». Comme le tableau et le schéma relationnel le montrent, une clé primaire est également requise dans la table de jonction car une autre table, par exemple `Services` dans notre schéma, pourrait s'y rattacher.

Tableau 5–2 La table de jonction « Faculté-Campus »

Id_primaire-campus/faculté	Id_étrangère_faculté	Id_étrangère_campus
1	1	1
2	2	1
3	1	2
4	2	2

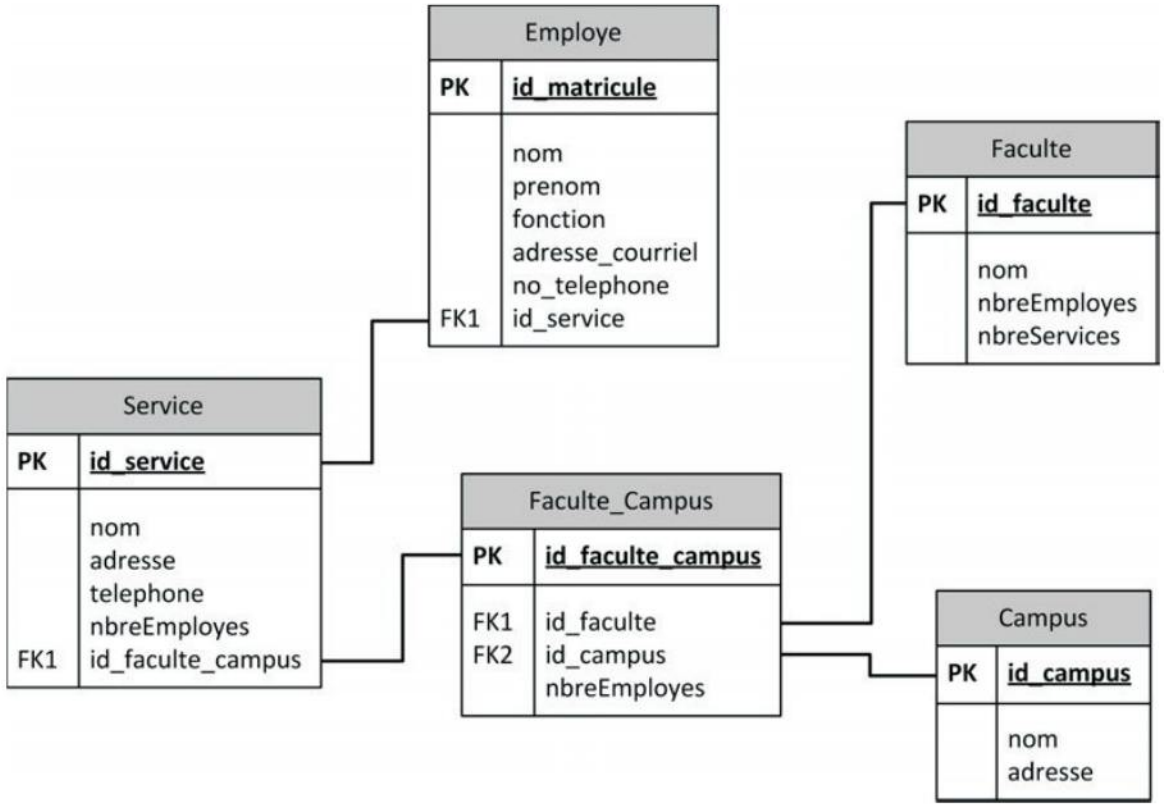


Figure 5–8 Diagramme relationnel complet

La problématique de la mise en correspondance relationnel/objet

Peut-être vous trouvez-vous un peu dérouté par ces deux méthodes représentant la même information : le relationnel d'un côté et l'orienté objet de l'autre. En fait, c'est leur histoire respective qui justifie ces deux méthodes et leurs différences, et plus d'un informaticien y a perdu son latin (euh, « son assembleur »).

Le relationnel se préoccupe du stockage efficace des données. Il se doit de fonctionner quel que soit le mode final de stockage de données, que ces données soient

d'ailleurs stockées ou pas. Les enregistrements doivent être accessibles où qu'ils se cachent. C'est la valeur de la clé primaire qui rend cet accès possible et unique.

L'orienté objet, lui, trouve sa source dans un mode de programmation plus proche de notre mode de pensée. En orienté objet, il s'agit de programmation donc de traitement de données ; par conséquent, les informations à traiter sont installées d'office dans une mémoire centrale informatique. Cette mémoire et son accès font partie intégrante de la programmation. L'adresse de son emplacement suffit à singulariser un objet. Pour accéder à cet objet, il suffit de connaître son adresse mémoire ; un objet ne pourra jamais se trouver qu'à une seule adresse.

En substance, **l'adresse de l'orienté objet fait office de clé primaire du relationnel**. Très logiquement, un objet en réfère un autre simplement en connaissant son adresse. Pour un enregistrement, cette même référence nécessite de joindre la clé primaire d'un côté à la clé étrangère de l'autre.

Une relation *n-n* ne pose pas de problème particulier en orienté objet. Chaque objet possède un tableau d'adresses permettant d'accéder à ceux auxquels il se trouve associé. Le relationnel exige une table intermédiaire de jonction.

Avec Django

Django vous épargne de saisir parfaitement cette mise en correspondance et vous fait grâce de ces différences en ne travaillant que dans un esprit orienté objet (bien que les données se trouvent finalement stockées dans un mode relationnel). En Django, le modèle de données est réalisé sous la forme d'un diagramme de classes (voir chapitre 3) en utilisant tous les modes possibles de mise en relation entre objets (association et héritage). Travailler sous Django vous permet de le faire dans un « esprit objet », jusqu'à ne rien connaître du fonctionnement des bases de données relationnelles. Cette indifférence vis-à-vis du relationnel et du langage SQL est une des qualités parmi les plus appréciées de Django.

Néanmoins, cela ne dispense pas d'une compréhension, fût-elle sommaire, du fonctionnement des bases de données relationnelles (nous décrirons un peu de SQL au prochain chapitre), qui restent à la base de tout site dynamique.

Retour au modèle de données de Trombinoscoop : son diagramme de classes

Après cette parenthèse rappelant les principes de base du modèle relationnel, reprenons sans plus attendre notre cahier des charges du début du chapitre et réfléchissons aux données qu'il sera nécessaire de stocker dans une base pour réaliser Trombinoscoop. Cette étude se fera dans une perspective pleinement objet et fera appel au dia-

gramme de classes, même si les données sont finalement stockées et gérées dans une base de données relationnelle.

Des personnes : étudiants et employés

Le premier objet qui nous vient naturellement à l'esprit est le concept « d'utilisateur ». Il va en effet falloir stocker quelque part les comptes utilisateur qui sont créés sur notre site, avec toutes leurs données : ce sera l'objet `Personne`.

Comme il existe deux types d'utilisateurs (étudiants et employés), nous allons dériver cet objet `Personne` en deux sous-objets. Nous nous reposons pour cela sur le concept d'*héritage*. Tous les attributs communs (nom, prénom, etc.) composeront l'objet `Personne`, tandis que les champs spécifiques seront placés dans les sous-objets respectifs (par exemple l'année d'étude pour un étudiant, qui n'a aucun sens pour un employé). Le diagramme de classes résultant est présenté sur la figure suivante.

EN PRATIQUE Django : héritage et relationnel

Bien que la notion d'héritage n'existe pas dans le monde relationnel, Django va s'occuper de la mise en correspondance et de créer les tables relationnelles nécessaires à cela.

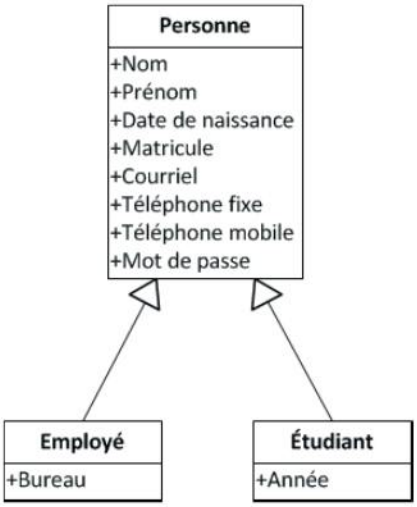


Figure 5–9
« Personne », « Employé » et « Étudiant » dans notre modèle de données

... qui travaillent ou étudient à l'université

Le lecteur attentif aura remarqué qu'il manque dans ce modèle certains attributs pourtant présents dans nos maquettes graphiques. Par exemple, la faculté des utilisateurs ne se retrouve nulle part. De même, aucun attribut ne permet de stocker le

cursus de l'étudiant. À ce stade, on peut juste enregistrer que l'étudiant est en deuxième année, mais non en Sciences mathémathiques dans la faculté des Sciences.

Nous allons ajouter des concepts séparés pour ces informations, plutôt que d'en faire des attributs des objets `Personne` ou `Étudiant`. Sous forme de diagramme de classes, notre nouveau modèle de données se présente comme suit.

EN PRATIQUE Séparer les concepts

Nous retrouvons bien évidemment les raisons fondamentales motivant l'établissement de relations entre les tables plutôt que tout « entasser » dans une gigantesque table unique : économiser l'encodage, éviter les erreurs et faciliter la mise à jour.

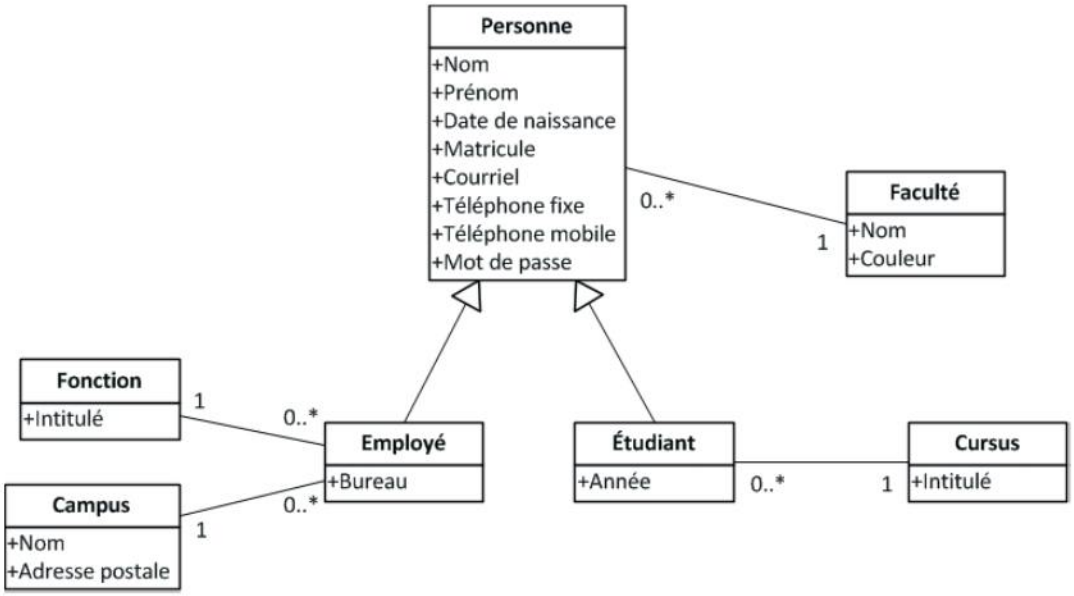


Figure 5–10 « Personne », « Employé » et « Étudiant » agrémentés de leurs objets annexes

Nous avons ajouté le concept de `Faculté` à `Personne` ; en effet, qu'on soit étudiant ou employé, on appartient à une faculté. Les employés ont une `Fonction` (secrétaire, professeur, technicien, etc.) et sont affectés à un `Campus` (sur lequel se trouve leur bureau). Les étudiants suivent un `Cursus` (par exemple « Master en Sciences physiques »).

Entre ces concepts, nous avons représenté des liens, dotés de *cardinalités*. Ces dernières sont faciles à comprendre. Prenons le lien qui relie `Personne` à `Faculté`. Le `1` signifie qu'à chaque personne correspond une seule faculté. Le `0..*` signifie qu'à chaque faculté peut correspondre entre zéro et une infinité de personnes.

EN PRATIQUE Attention : soyez attentifs à la position des informations de cardinalité !

Un objet `Personne` sera toujours associé à une seule faculté alors qu'une faculté, elle, se trouve associée à un nombre indéterminé de personnes.

EN PRATIQUE Django : les aspects relationnels sont masqués

Vous pouvez constater que les attributs « techniques » (les `id`) qui servent à lier les objets entre eux n'ont pas été représentés. Comme nous l'avons vu, c'est le framework Django qui se charge de les créer sur la seule base des relations définies. Django vous dissimule entièrement le modèle relationnel qui s'occupe du stockage et du traitement des données.

... et qui échangent des messages avec des amis

À ce stade, nous avons toutes nos données pour représenter les profils de nos utilisateurs. Il reste néanmoins deux données à sauvegarder : les amis d'un utilisateur et les messages qu'il publie (et ce, même si l'immense majorité des messages transmis sur les sites sociaux ne mériterait pas un tel gaspillage de bits à sauvegarder).

Pour les messages, rien n'est plus simple. Il suffit de créer un objet `Message` lié à `Personne`. On indiquera à l'aide des cardinalités qu'une personne peut publier autant de messages qu'elle veut, tandis qu'un message est écrit par une seule personne.

Pour représenter les amis, c'est également assez simple. Qu'est-ce qu'un ami au fond, sinon une personne ? On va donc simplement relier l'objet `Personne` avec lui-même, en précisant à l'aide des cardinalités qu'une personne peut être liée à une infinité d'autres personnes (c'est maintenant le nombre d'amis sur Facebook qui supplante le plus costaud des papas).

Notre modèle de données final ressemblera à la figure ci-contre.

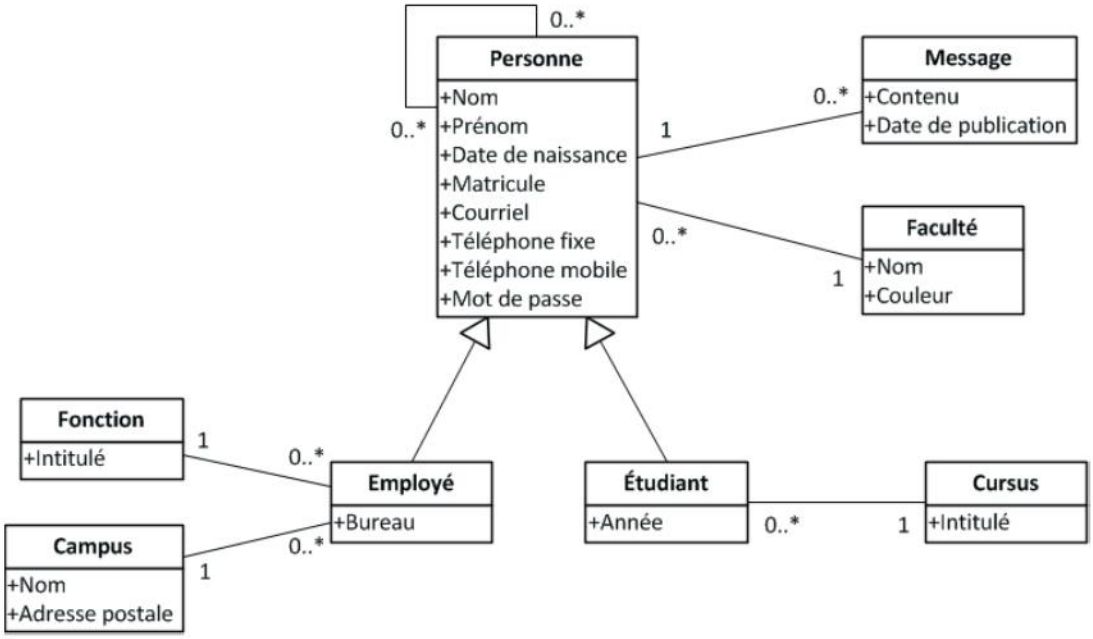


Figure 5–11 Modèle de données final de Trombinoscoop

Ai-je bien compris ?

- Quelles différences distinguent le modèle objet du modèle relationnel ?
- À quoi servent cas d'utilisation et wireframes ?
- Comment représente-t-on une relation « *n*-*n* » en modèle objet et en modèle relationnel ?

6

Premier contact avec les bases relationnelles et SQL à partir d'un exemple en CGI

Le site web doit pouvoir interroger et modifier la base relationnelle où sont stockées toutes les données du modèle. Il lui faut pour cela un langage particulier : le SQL. Les notions de SQL dont nous aurons besoin par la suite, mais qui sont le plus souvent cachées par le framework Django, seront exposées dans un contexte web réalisé en CGI. Cette technologie n'est plus de mise aujourd'hui pour le développement web côté serveur, mais à titre pédagogique, ce chapitre nous permettra de l'évoquer comme un parfait exemple d'un mélange de genres à éviter.

L'installation des outils de développement est expliquée en annexe A

SOMMAIRE

- Aperçu des requêtes SQL côté serveur
- Introduction à CGI : un condensé des mauvaises pratiques du Web

Dans le chapitre précédent, nous avons élaboré notre modèle de données et décidé de stocker les informations de notre projet dans une base de données relationnelle. Il nous faut maintenant étudier comment faire communiquer notre site avec ces bases. C'est le rôle du langage SQL (*Structured Query Language*).

Il y a quelques années, la mise en place de l'interactivité et de la dynamique côté serveur recourait à la technologie CGI (*Common Gateway Interface*). Cette technologie s'accommode de petits scripts côté serveur, écrits en langage interprété (comme Perl ou Python) ou en langage compilé (très souvent C++), qui créent sur le fil des pages HTML pour les renvoyer et les afficher côté client.

Rien ne vous interdit de continuer à la pratiquer, mais vous risquez de sérieux maux de tête ! Nous vous le déconseillons très vivement, sinon en désespoir de cause ou pour de très petits projets bien ciblés.

Malgré tout, nous vous la présentons très succinctement dans deux optiques :

- d'une part pour **mettre rapidement en pratique les notions de SQL** qui vous seront nécessaires pour comprendre les chapitres suivants, bien que souvent cachées par les objets de Django ;
- d'autre part pour vous **montrer**, à titre pédagogique, **ce qu'il ne faut pas faire** : un code fourre-tout dans lequel se mélangent indigestement langage de programmation, HTML et SQL.

Pour conduire notre propos, nous allons « construire » un simple carnet d'adresses web. Trois fonctionnalités seront développées : « afficher tous les employés de l'université », « les afficher par service » et « ajouter de nouveaux employés ».

> **EN PRATIQUE Et si on faisait le site Trombinoscoop en CGI ?**
>
> En guise d'exemple, nous pourrions développer en CGI notre projet Trombinoscoop. Ce serait de fait une tâche titanesque et un si gros projet n'est pas utile pour vous donner un aperçu de la technologie CGI. C'est pourquoi nous allons nous atteler à un projet beaucoup plus modeste : un simple carnet d'adresses électronique.

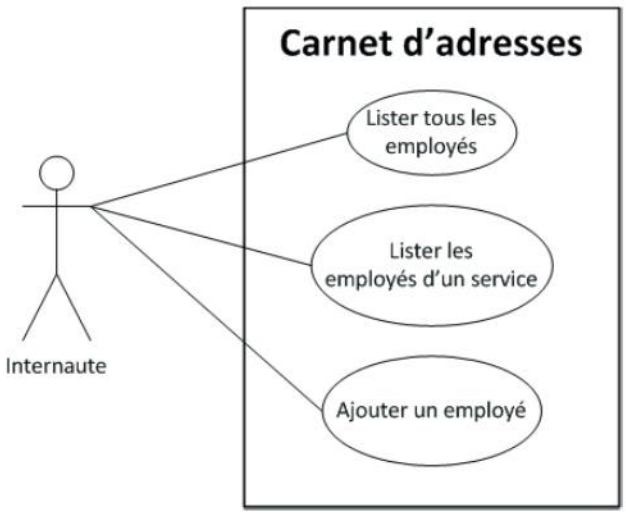
Aussi modeste soit ce carnet d'adresses, nous n'allons pas nous épargner la phase d'analyse, afin de déterminer exactement ce que l'on veut développer et d'esquisser le résultat final attendu.

Analyse du carnet d'adresses : des cas d'utilisation au modèle de données

Trois cas d'utilisation simples

Les cas d'utilisation sont très simples et découlent des trois fonctionnalités que nous avons énumérées à la section précédente.

Figure 6–1
Cas d'utilisation du carnet d'adresses



Maquette des écrans (wireframes)

Les wireframes sont également assez simples. Nous avons groupé tous les écrans et leurs liens en un seul dessin (voir figure 6-2, page suivante).

Le modèle de données du carnet

Le modèle de données de notre carnet d'adresses électronique sera similaire à celui que nous avons produit pour Trombinoscoop au chapitre précédent.

Seuls deux objets sont définis dans notre modèle : `Employé` et `Service`, avec un lien `0..* - 1` les unissant : un service par employé et de multiples employés par service.

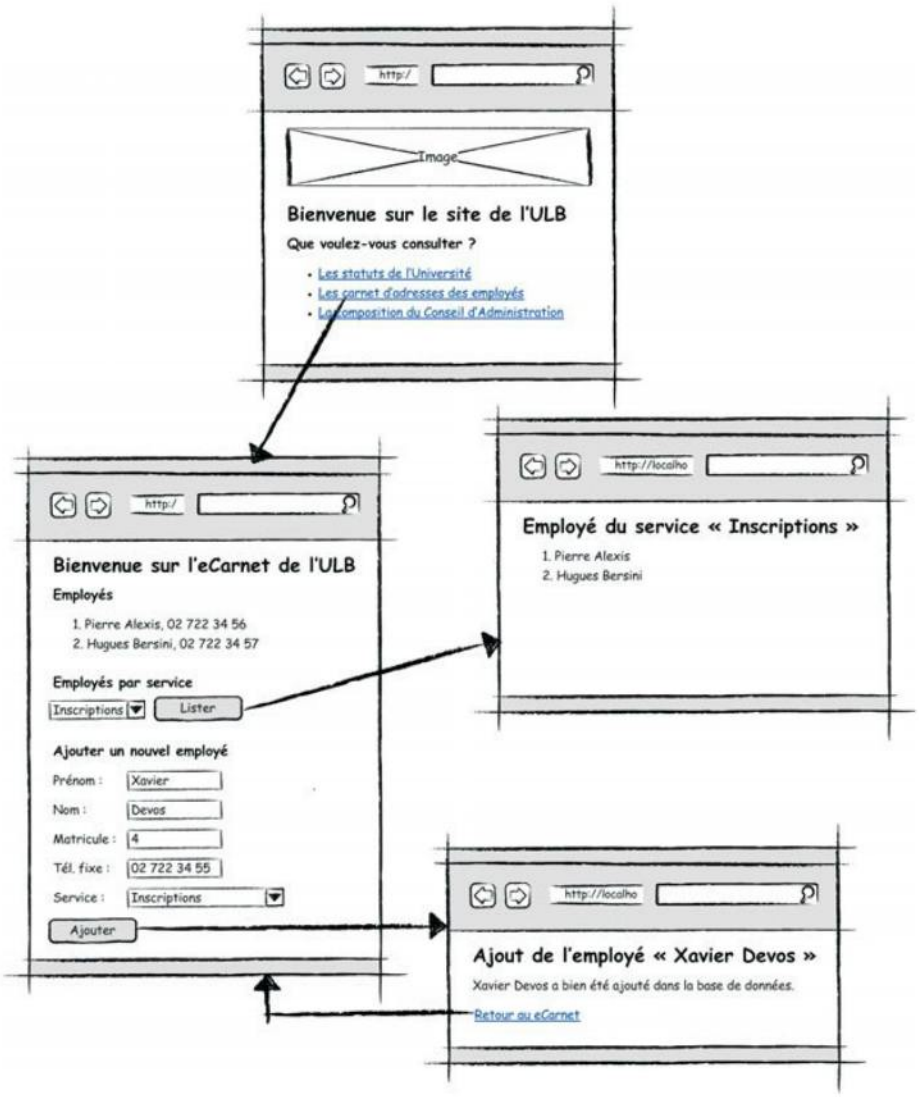


Figure 6–2
Maquette des écrans du carnet d'adresses

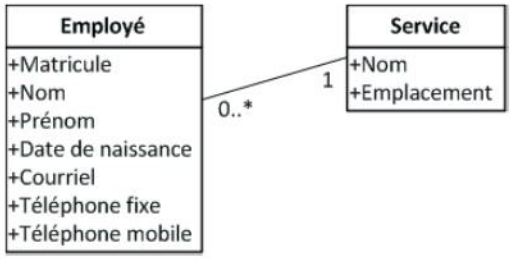


Figure 6–3
Modèle de données conceptuel du carnet d'adresses

Ce modèle est dit « conceptuel », car il ne représente pas exactement les tables que nous allons implémenter dans la base de données. Plusieurs modifications d'ordre technique doivent être apportées au modèle.

- La plus importante est l'ajout de clés dans les deux tables. Nous avons vu que toute table devait posséder un attribut identifiant chacun de ses enregistrements de manière unique.

  Pour la table `Employé`, le matricule constituera parfaitement la clé primaire. En revanche, aucun attribut de la table `Service` ne peut jouer ce rôle. `Nom`, par exemple, n'est pas un bon candidat de clé, car on peut très bien imaginer plusieurs services qui portent le même nom au sein de l'Université (par exemple « Secrétariat »). On va donc ajouter un attribut nommé `Id`.

EN PRATIQUE Comparaison : clé primaire en CGI et avec Django

La technologie CGI a recours explicitement aux bases de données relationnelles, mais Django, que nous découvrirons par la suite, pourrait réaliser cet ajout à notre place et de manière transparente.

  Cet `Id` de service sera également la clé étrangère de la table `Employé`, sous le nom `ID-Service`. L'ajout de ces deux attributs d'identification dans les tables `Employés` et `Service` n'est rien d'autre que l'expression concrète de la relation `1-n` qui unit les deux tables.
- Les majuscules, espaces et accents doivent être supprimés, car les bases de données n'en sont pas friandes.

Notre modèle va donc devenir le suivant :

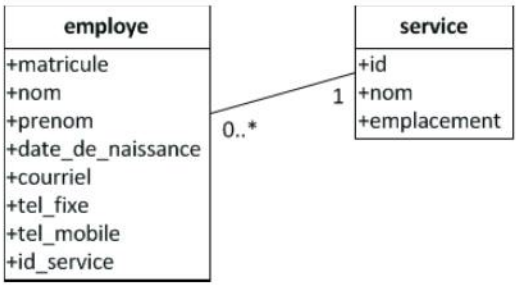


Figure 6–4
Modèle de données technique du carnet d'adresses

Création de la base de données avec SQLite

Pour notre base de données, nous allons utiliser le logiciel libre SQLite. Il s'agit d'un système de gestion de bases de données (SGBD) très léger et très facile d'emploi, qui n'utilise qu'un unique fichier pour encoder les bases de données. Ces dernières sont

très facilement exploitables sous Python, nativement et sans à avoir à installer la moindre bibliothèque supplémentaire.

EN PRATIQUE Outil : créer une base de données SQLite

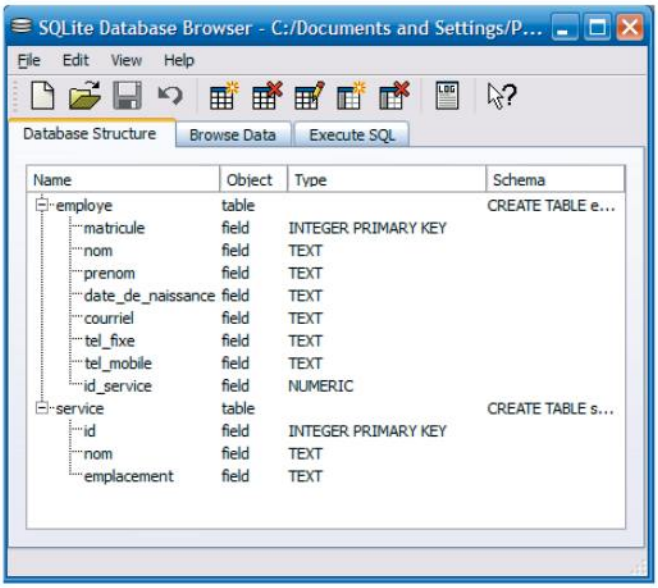
Pour créer une base de données SQLite, il existe de nombreux petits utilitaires libres et gratuits. Nous avons utilisé *SQLite Database Browser* disponible librement à l'adresse suivante :

- http://sqlitebrowser.sourceforge.net/

Cet utilitaire fonctionne sur Windows, Linux et Mac OS X.

Créer notre base de données à l'aide de SQLite Database Browser, sur la base du modèle que nous nous sommes défini, est assez élémentaire. Vous devriez rapidement obtenir le résultat de la figure suivante.

Figure 6–5
Création de la base de données SQLite



Notez que chaque attribut doit être typé et que trois types de données ont été utilisés : `TEXT`, `NUMERIC` et `INTEGER PRIMARY KEY`. Nos champs sont pour la plupart textuels. Un nom, c'est du texte. Un téléphone en est aussi, car il peut contenir des caractères non numériques (par exemple le numéro +32 (0)789 56 78 90). Seuls nos identifiants et notre matricule sont numériques ou entiers. Quant à `INTEGER PRIMARY KEY`, c'est tout simplement la clé primaire.

Une fois la base de données créée, n'hésitez pas à vous familiariser avec ce logiciel élémentaire et à y encoder quelques données d'exemple.

Accès à la base de données via SQL

Notre site web nécessite d'accéder à la base de données. En Django, cet accès se fera via des objets Python. En CGI en revanche, nous n'avons aucun framework pour nous aider à accéder à la base de données. Nous allons donc devoir utiliser un langage nommé SQL, que tous les informaticiens pratiquement allègrement.

DÉFINITION SQL

Le sigle de ce langage signifie *Structured Query Language*, ou « Langage de requêtes structurées ». C'est devenu le standard incontournable du « monde relationnel » et toute application informatique devant s'interfacer avec une base de données relationnelle, ne peut le faire qu'en « s'exprimant » dans ce langage.
SQL est un standard bien installé aujourd'hui, que l'on retrouve à l'œuvre chez tous les acteurs importants du monde des bases de données relationnelles, tels MySQL, PostgreSQL, Oracle, Microsoft Access, Sybase ou Informix. Il est devenu partie intégrante du matériel de base que tout informaticien se doit d'inclure dans ses bagages pour monter à l'assaut de son premier sommet web.

EN PRATIQUE Usage du SQL dans Django

Nous l'avons vu, en Django, l'usage de SQL sera « caché » dans la grande majorité des cas. Le framework se charge de traduire lui-même les accès à la base de données. C'est une de ses qualités majeures, qui permet d'interfacer une même application Django avec n'importe quel SGBD.

Syntaxe des requêtes SQL les plus courantes

Comme son nom l'indique, SQL est basé sur un concept de « requêtes », qui permettent la création des tables et des relations, la consultation, l'insertion, l'effacement et la modification des enregistrements, ainsi que la définition des permissions au niveau des utilisateurs. Voici un très bref aperçu de la construction des requêtes SQL les plus utilisées.

SYNTAXE. SELECT pour lire les enregistrements

```
SELECT [ALL] | [DISTINCT]
<liste des noms de colonnes> | *
FROM <Liste des tables>
[WHERE <condition logique>]
```

Notez que la barre verticale sépare des options. Par exemple, soit on liste des noms de colonnes, soit on les choisit toutes avec l'étoile.

Les crochets, quant à eux, désignent des éléments optionnels. Il n'est donc pas obligé d'avoir une clause `WHERE`.

SYNTAXE. INSERT pour insérer de nouveaux enregistrements

```
INSERT INTO Nom_de_la_table
(colonne1,colonne2,colonne3,...)
VALUES (Valeur1, Valeur2, Valeur3, ...)
```

SYNTAXE. UPDATE pour mettre à jour des enregistrements

```
UPDATE Nom_de_la_table
SET Colonne = Valeur
[WHERE qualification]
```

SYNTAXE. DELETE pour effacer des enregistrements

```
DELETE FROM Nom_de_la_table
WHERE qualification
```

Quelques exemples liés à notre base de données

Nous verrons par la suite des exemples concrets de requêtes SQL. D'ores et déjà, pour obtenir les noms et prénoms de tous les employés classés par ordre alphabétique du nom, il faudrait écrire en SQL :

EXEMPLE 6.1 Notre première requête SQL

```
SELECT prenom, nom FROM employe ORDER BY nom
```

Si la requête porte sur plusieurs tables mises en relation (par exemple obtenir tous les employés travaillant dans le deuxième service), il faudrait écrire :

EXEMPLE 6.2 Requête sur des tables liées

```
SELECT employe.prenom, employe.nom, service.nom
FROM employe, service
WHERE employe.id_service = service.id AND service.id = 2
ORDER BY employe.nom
```

Ce que l'on trouve dans le `WHERE` est une *condition de jointure*. C'est grâce à elle que l'on va préciser comment chaque enregistrement d'une table peut être lié à un ou plusieurs enregistrement(s) d'une autre table. C'est ici que se met en place le lien reliant les deux tables et qui apparaît dans le graphique de la base de données : en égalisant ces deux clés, on peut passer d'une table à l'autre.

Voici maintenant une requête qui se limite à corriger le nom d'un des employés :

EXEMPLE 6.3 Corriger le nom d'un employé

```
UPDATE employe SET nom = "Bersini" WHERE matricule = 2
```

Réalisation du carnet d'adresses avec SQL et CGI

Lancement d'un serveur web Python

La mise en place d'un site web en CGI requiert l'exécution d'un programme côté serveur que nous appellerons le « serveur web », dont une version très simple vous est fournie ci-après. Il s'agit d'un petit programme Python qu'il vous faut toujours exécuter avant toute chose. Il suffit de lancer la console généralement fournie lors de l'installation de Python.

SYNTAXE. Démarrage du serveur web Python pour Windows

```
from BaseHTTPServer import HTTPServer
from CGIHTTPServer import CGIHTTPRequestHandler

httpd = HTTPServer(('', 8080), CGIHTTPRequestHandler)
print("Demarrage du serveur web...")
httpd.serve_forever()
```

Ce script démarre l'exécution d'un serveur web à même votre ordinateur sur le port local 8080. Pour accéder à notre site web, nous devrons donc introduire l'adresse http://localhost:8080 dans notre navigateur.

DÉFINITION Port

Un port est un canal de communication que votre ordinateur met à disposition pour communiquer avec d'autres.

Nous venons de créer un serveur CGI basique. Il permet de *servir* deux types de pages :

- des pages statiques : toutes les pages placées dans le répertoire qui contient notre script seront servies de manière statique. C'est là qu'on va placer nos pages écrites en pur HTML et qui ne comportent aucun code Python.
- des pages dynamiques : toutes les pages placées dans le sous-dossier `cgi-bin` seront d'abord envoyées à l'interpréteur Python.

Les figures suivantes illustrent ces deux cas de figure.

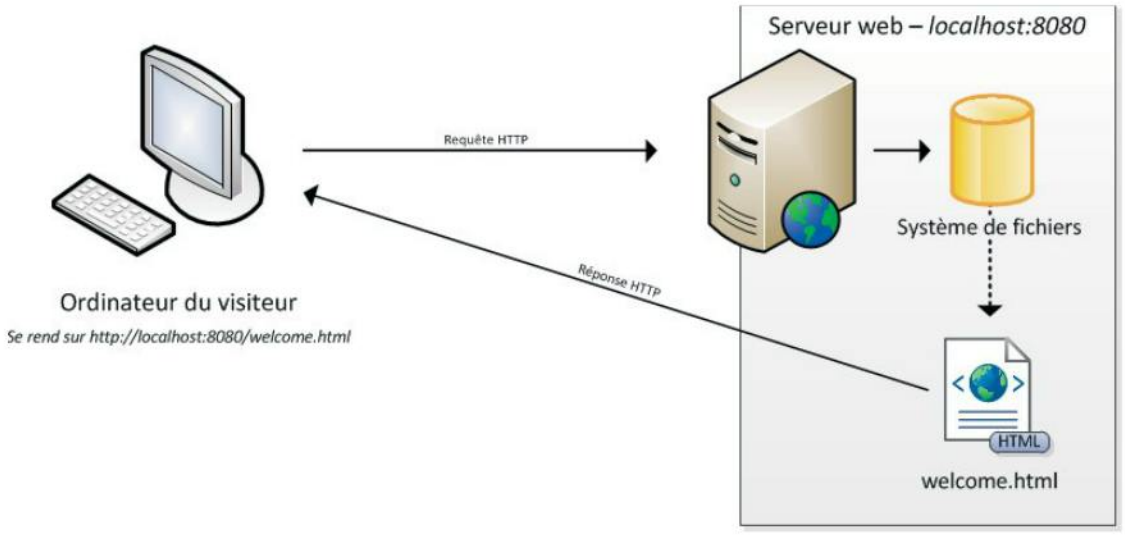


Figure 6–6 Comportement du serveur web lorsqu'on visite la page « welcome.html »

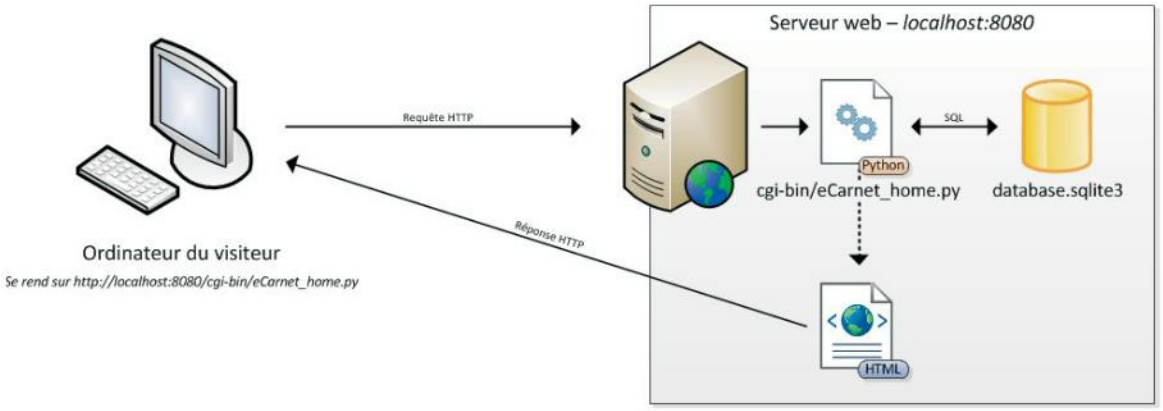


Figure 6–7 Comportement du serveur web lorsqu'on visite la page « cgi-bin/eCarnet_home.py »

Dans le premier cas, on visite une page statique. Le serveur web va donc simplement la chercher dans le système de fichiers et renvoie son contenu.

Dans le second cas, dynamique, le serveur web exécute préalablement le script Python correspondant. Ce script a besoin de se connecter à la base de données `database.sqlite3` pour créer la page HTML.

Passons maintenant aux développements de nos différentes pages.

L'écran d'accueil de l'ULB

L'écran d'accueil est une page entièrement statique composée uniquement en HTML, qui doit donc être placée dans le même dossier que le script Python exécutant le serveur web.

EXEMPLE 6.4 Carnet d'adresses. Le fichier welcome.html

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Université libre de Bruxelles</title>
  </head>
  <body>
    <img src="logo_ulb.png"
    alt="Logo de l'ULB" width="656" height="80" />
    <h1>Bienvenue sur le site de l'ULB</h1>
    <section>
      <h2>Que voulez-vous consulter ?</h2>
      <ul>
        <li><a href="statuts.html">Les statuts de
        l'Université</a></li>
        <li><a href="cgi-bin/eCarnet_home.py">Le carnet
        d'adresses des employés</a>(1)</li>
        <li><a href="ca.html">La composition du Conseil
        d'Administration</a></li>
      </ul>
    </section>
  </body>
</html>
```

Le deuxième lien (1) propose d'accéder à notre carnet d'adresses. Il pointe vers le script `cgi-bin/eCarnet_home.py`, qui va se charger de créer notre deuxième écran.

Visuellement, voici à quoi ressemble notre première page statique.

Figure 6–8
Notre page d'accueil

La page principale du carnet d'adresses

Cette page est la plus compliquée de ce mini-site ; elle sert de point d'entrée à un certain nombre de fonctionnalités.

Voici le code de notre deuxième page qui, cette fois-ci, est dynamique et produite par l'exécution d'un code Python.

EXEMPLE 6.5 Carnet d'adresses. Le fichier cgi-bin/eCarnet_home.py

```
#!/usr/bin/env python (1)
# -*- coding: utf-8 -*- (2)

import sqlite3 (3)

print "Content-Type: application/xhtml+xml; charset=utf-8\n" (4)

# Partie statique de la page HTML
print '<?xml version="1.0" encoding="UTF-8" ?>'
print '<!DOCTYPE html>'
print '<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">'
print ' <head>'
print '    <title>eCarnet - Home</title>'
print ' </head>'
print ' <body>'
print '    <h1>Bienvenue sur l'eCarnet de l'ULB</h1>'
print '    <h2>Employés</h2>'
```

```
# Connexion à la base de données
db_connection = sqlite3.connect('database.sqlite3') (5)
db_connection.row_factory = sqlite3.Row
cursor = db_connection.cursor() (6)

#Sélection des enregistrements
cursor.execute("SELECT prenom, nom, tel_fixe FROM employe") (7)

#Création de la liste des employés
rows = cursor.fetchall() (8)
print '    <ol>'
for row in rows: (9)
  print '      <li>' + row['prenom'] + ' ' + row['nom'] + ', '
                    ➥ + row['tel_fixe'] + '</li>' (10)
print '    </ol>'

#Formulaire de recherche des employés d'un service
print '    <h2>Employés par service</h2>' (11)
print '    <form action="eCarnet_service.py" method="get">' (14)
print '      <p><select name="service">'
cursor.execute ("SELECT id, nom FROM service") (15)
rows = cursor.fetchall()
for row in rows: (16)
  print '        <option value="' + str(row['id']) + '">'
                    ➥ + row['nom'] + '</option>' (12)
print '      </select>'
print '      <input type="submit (13)" value="Lister" /></p>'
print '    </form>'

# Formulaire d'ajout d'un nouvel employé
print '    <h2>Ajouter un nouvel employé</h2>'
print '    <form action="eCarnet_add_employee.py" method="get">'
print '      <p>Prénom : <input type="text" name="prenom" /></p>'
print '      <p>Nom : <input type="text" name="nom" /></p>'
print '      <p>Matricule : <input type="text" name="matricule" /></p>'
print '      <p>Tél. fixe : <input type="text" name="tel_fixe" /></p>'
print '      <p>Service : <select name="service">'
for row in rows:
  print '        <option value="' + str(row['id']) + '">' + row['nom']
                    ➥ + '</option>'
print '      </select></p>'
print '      <p><input type="submit" value="Ajouter" /></p>'
print '    </form>'
print ' </body>'
print '</html>'

db_connection.close() (17)
```

Le rendu de cette page est présenté sur la figure suivante.

Figure 6–9
Page principale de notre carnet d'adresses

Ne nous attardons pas sur l'apparence de la page et passons plutôt en revue ce mélange indigeste de Python et de HTML.

La première ligne indique, pour les systèmes Unix, que l'interpréteur Python doit être utilisé (1). La deuxième indique à l'interpréteur Python que le fichier est encodé en UTF-8 (2) (voir le chapitre 4, page 65). Ensuite, nous importons la bibliothèque `sqlite3` (3) pour nous interfacer à la base de données. Il faut également indiquer au serveur web quel type de contenu nous allons renvoyer dans la requête HTTP (4), en l'occurrence une page XHTML encodée en UTF-8.

Ensuite vient du HTML. Remarquez que nous devons à chaque fois faire usage de la fonction Python `print` dès qu'il s'agit d'écrire du HTML « à la volée ». Passons outre ces lignes faciles à comprendre et sautons directement au point (5).

Le but de cet extrait de code est de construire la liste HTML qui énumère tous nos employés. On se connecte à la base de données à l'aide de la méthode `connect` de

SQLite, laquelle crée un objet que nous stockons dans la variable `db_connection`. C'est sur cet objet que nous enverrons nos requêtes SQL.

> **EN PRATIQUE Attention à l'emplacement des fichiers**
>
> La base de données doit se trouver dans le même dossier que le script Python qui a servi à lancer notre site web, donc *pas* dans le dossier `cgi-bin`.

Ensuite, nous allons récupérer les listes des employés via SQL. On procède d'abord à l'obtention d'un objet `cursor` (6) qui envoie la requête SQL `SELECT` à la base de données et en récupère les résultats : noms, prénoms et numéros de téléphone (7).

Les résultats renvoyés par la base de données sont extraits via la méthode `fetchall` du curseur. Ils sont placés dans une variable liste que nous nommons `rows` (8). Nous pouvons maintenant parcourir cette liste à l'aide d'une boucle (9) pour passer en revue tous ses éléments. À mesure que l'on parcourt l'ensemble des résultats, on crée les éléments HTML `<li>` de notre liste (10). Une fois la liste construite, nous pouvons passer au formulaire de recherche des employés d'un service. La première ligne écrit le titre en HTML (11). Ensuite, nous créons un formulaire HTML qui va contenir deux champs :

- une liste de choix reprenant tous les services disponibles (12) ;
- un bouton de soumission (13).

À la soumission du formulaire, nous aimerions être redirigés vers le script `eCarnet_service.py`. C'est ce que mentionne l'attribut `action` de l'élément `form` (14). Pour établir notre liste de choix avec tous les services, nous devons également faire une requête sur la base de données (15) et boucler sur les résultats (16) afin de créer toutes les « options » possibles.

Nous créons de manière similaire le formulaire d'ajout d'un employé, dont nous vous laissons le plaisir d'analyser le code.

Il est important de toujours « clôturer » toute interaction avec la base de données, de manière à libérer celle-ci pour de nouveaux usages. Cette obligation est satisfaite par l'entremise de l'instruction `db.close()` (17).

Une fois le script terminé, le HTML résultant est envoyé en réponse à la requête HTTP d'origine.

EXEMPLE 6.6 Carnet d'adresses. Le HTML créé par le script

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>eCarnet - Home</title>
  </head>
  <body>
    <h1>Bienvenue sur l'eCarnet de l'ULB</h1>
    <h2>Employés</h2>
    <ol>
      <li>Pierre Alexis, 02 722 34 56</li>
      <li>Hugues Bersini, 02 722 34 57</li>
      <li>Gael Rabier, 02 722 34 52</li>
    </ol>
    <h2>Employés par service</h2>
    <form action="eCarnet_service.py" method="get">
      <p><select name="service">
        <option value="1">Recherche opérationnelle</option>
        <option value="2">Inscriptions</option>
        <option value="3">Comptabilité</option>
        <option value="4">Informatique</option>
      </select>
      <input type="submit" value="Lister" /></p>
    </form>
    <h2>Ajouter un nouvel employé</h2>
    <form action="eCarnet_add_employee.py" method="get">
      <p>Prénom : <input type="text" name="prenom" /></p>
      <p>Nom : <input type="text" name="nom" /></p>
      <p>Matricule : <input type="text" name="matricule" /></p>
      <p>Tél. fixe : <input type="text" name="tel_fixe" /></p>
      <p>Service : <select name="service">
        <option value="1">Recherche opérationnelle</option>
        <option value="2">Inscriptions</option>
        <option value="3">Comptabilité</option>
        <option value="4">Informatique</option>
      </select></p>
      <p><input type="submit" value="Ajouter" /></p>
    </form>
  </body>
</html>
```

La liste des employés d'un service

Le code de la page listant les employés d'un service n'est pas vraiment plus compliqué. Le voici :

EXEMPLE 6.7 Carnet d'adresses. Le fichier cgi-bin/eCarnet_service.py

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

import sqlite3
import cgi

print "Content-Type: application/xhtml+xml; charset=utf-8\n"

print '<?xml version="1.0" encoding="UTF-8" ?>'
print '<!DOCTYPE html>'
print '<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">'
print ' <head>'
print '    <title>eCarnet - Employés d'un service</title>'
print ' </head>'
print ' <body>'

form = cgi.FieldStorage()
service_id = str(form["service"].value) (1)
db_connection = sqlite3.connect('database.sqlite3')
db_connection.row_factory = sqlite3.Row
cursor = db_connection.cursor()
cursor.execute("SELECT nom FROM service WHERE id=" + service_id)
row = cursor.fetchone()
service_nom = str(row['nom']) (2)

print '    <h1>Employés du service « ' + service_nom + ' »</h1>' (3)

cursor.execute("SELECT prenom, nom, tel_fixe FROM employe WHERE id_service="
               ➥ + service_id) (4)
rows = cursor.fetchall()

print '    <ol>'
for row in rows:
  print '      <li>' + row['prenom'] + ', ' + row['nom'] + ', '
        ➥ + row['tel_fixe'] + '</li>' (5)
print '    </ol>'
print ' </body>'
print '</html>'

db_connection.close()
```

La subtilité dans ce script, par rapport au précédent, est le passage d'un paramètre (1). En effet, ce code permet de lister les employés d'un service en particulier. Mais lequel ? Cette information doit bien lui être transmise…

Au niveau HTML, le paramètre a été défini dans le formulaire de notre écran décrit précédemment, dont revoici le code tel que créé par Python :

EXEMPLE 6.8 Carnet d'adresses. Formulaire de sélection d'un service

```
<form action="eCarnet_service.py" method="get">
  <p><select name="service">
    <option value="1">Recherche opérationnelle</option>
    <option value="2">Inscriptions</option>
    <option value="3">Comptabilité</option>
    <option value="4">Informatique</option>
  </select></p>
  <p><input type="submit" value="Lister" /></p>
</form>
```

Un attribut HTML est important ici : l'attribut `name` du `select`, dont la valeur est `service`. Le service sélectionné par l'utilisateur sera bien transmis via un paramètre nommé `service`. La valeur qui sera donnée à ce paramètre sera son `id`, puisque c'est ce que nous avons mis dans les attributs `value` de chacune des « options ». Ainsi donc, si l'on choisit le service « Inscriptions » et si l'on clique sur le bouton, l'URL appelée sera la suivante : http://localhost:8080/cgi-bin/eCarnet_service.py?service=2.

Il reste à récupérer la valeur de ce paramètre dans notre script `eCarnet_service.py`. Cela se fait via les lignes suivantes que nous rappelons ici :

EXEMPLE 6.9 Carnet d'adresses. Récupération d'un paramètre dans le script

```
import cgi
...
form = cgi.FieldStorage()
service_id = str(form["service"].value)
```

Nous devons utiliser la bibliothèque `cgi`. Ensuite, la méthode `FieldStorage` récupère, sous forme d'un dictionnaire, l'ensemble des paramètres passés au script (1). On en extrait alors l'élément nommé `service`, avant de le convertir en un `string` car il servira à construire les requêtes SQL, qui sont des requêtes textuelles.

Le reste du code de notre page liste les employés d'un service. On récupère d'abord dans la base de données le nom du service (2) (à ce stade, on ne connaît que son `id`) afin de pouvoir personnaliser le titre de la page (3). Ensuite, on récupère tous les employés du service en question à l'aide d'une requête SQL plus ciblée (4) et on les affiche (5).

Visuellement, le résultat est le suivant :

Figure 6–10
Page affichant les employés d'un service

Ajout d'un employé

La dernière fonctionnalité à implémenter est l'ajout d'un nouvel employé en spécifiant son matricule, son prénom, son nom, son téléphone fixe et son service. L'obtention de son service se fait par la même petite liste déroulante que précédemment.

Cette fonctionnalité requiert l'exécution d'un nouveau script Python `eCarnet_add_employee`, dont voici le code.

EXEMPLE 6.10 Carnet d'adresses. Le fichier eCarnet_add_employee.py

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

import sqlite3
import cgi

print "Content-Type: application/xhtml+xml; charset=utf-8\n"

print '<?xml version="1.0" encoding="UTF-8" ?>'
print '<!DOCTYPE html>'
print '<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">'
print ' <head>'
print '    <title>eCarnet - Ajout d'un employé</title>'
print ' </head>'
print ' <body>'

form = cgi.FieldStorage()
id_service = str(form["service"].value)
nom = str(form["nom"].value)
prenom = str(form["prenom"].value)
matricule = str(form["matricule"].value)
tel_fixe = str(form["tel_fixe"].value)
```

```
db_connection = sqlite3.connect('database.sqlite3')
db_connection.row_factory = sqlite3.Row
cursor = db_connection.cursor()
(1) cursor.execute("INSERT INTO employe(matricule, prenom, nom, tel_fixe,
                    ➥ id_service) VALUES (?, ?, ?, ?, ?)",
                (matricule, prenom, nom, tel_fixe, id_service))
db_connection.commit() (2)
db_connection.close()

print '    <h1>Ajout de l'employé « ' + prenom + ' ' + nom + ' »</h1>'
print '    <p>' + prenom + ' ' + nom + ' a bien été ajouté dans la base
          ➥ de données.</p>'
print '    <p><a href="eCarnet_home.py">Retour au eCarnet.</a></p>'
print ' </body>'
print '</html>'
```

L'insertion d'un nouvel enregistrement dans la table des employés se fait par l'exécution sur l'objet `cursor` d'une requête `INSERT` dont les valeurs nécessaires (`VALUES`) sont obtenues via le formulaire `form` (1). Notez les points d'interrogation de la requête qui sont remplacés automatiquement par les valeurs spécifiées dans le deuxième paramètre de la fonction `execute`. C'est un raccourci d'écriture non négligeable car cela évite de devoir construire le `string` de la requête par concaténation.

L'exécution seule de la requête ne suffit pas. Il faut également penser à faire un `commit` (2) sur le curseur afin que les données soient effectivement enregistrées dans la base de données. Une fois le nouvel employé ajouté, on affiche un message pour expliquer que « tout va bien » et on propose un lien permettant de revenir à la page principale de notre eCarnet. Visuellement, le code donne ceci :

Figure 6–11
Ajout d'un employé dans la base de données

CGI : ce qu'il ne faut plus faire

Vous l'aurez constaté, nous ne portons pas la technologie CGI dans nos cœurs d'informaticiens (de ceux qui battent au gigahertz). Avouons-le, notre funeste tableau est quelque peu exagéré. Quand nous prétendons que « rien de bon ne peut sortir de CGI », ce n'est pas tout à fait vrai. Cela nous aura au moins permis de vous expliquer un des langages phares de l'informatique, SQL. De surcroît, on peut très bien réaliser des sites web complexes au rendu agréable et ergonomique à partir de CGI. Le problème se situe en réalité au niveau du code et de son organisation : CGI amène très rapidement d'infâmes codes « spaghetti » impossibles à maintenir.

Plus concrètement, en vous référant à notre exemple de site en CGI, vous aurez pu constater que les fichiers réalisés souffrent d'un terrible mélange de genres : HTML, Python et SQL se retrouvent mêlés les uns aux autres dans un même fichier. Un concepteur web ne pourra donc pas travailler sur son HTML pendant qu'un programmeur avancera sur le code Python. Pire, ce mélange de genres impose au concepteur web de comprendre le code Python pour savoir où il devra modifier son HTML et comment.

Vous remarquerez également que de nombreuses lignes de code se répètent et que les « copier-coller » foisonnent. C'est le cas par exemple de l'en-tête de nos HTML : dans tous les scripts, il est plus ou moins le même, alors qu'il doit être possible de ne le définir qu'en un seul endroit.

Certes, nous pourrions utiliser toute la puissance de Python et de l'orienté objet pour améliorer la structure du code. Dans ce cas, en réalité, cela reviendrait à entamer la construction de notre propre framework en lieu et place de Django, recommencer ce projet logiciel dont un jour des journalistes du lointain Kansas ont saisi toute l'utilité et le temps que cela permettrait d'épargner.

En matière de facilité de conception et de stabilité de code, il y a donc beaucoup d'améliorations possibles. Et ça tombe plutôt bien, car c'est là que Django va faire toute la différence.

Ai-je bien compris ?

- À quoi sert le langage SQL ?
- Comment, en CGI, un serveur peut-il servir à la fois du contenu statique et dynamique ?

Deuxième partie

Mise en application avec Django

Dans cette deuxième partie, nous allons construire notre site Trombinoscoop en nous appuyant sur le framework Django. Nous détaillerons chacune des étapes d'élaboration d'un site web :

- « vues » Django (correspondant à l'aspect Contrôle du MVC) ;
- « templates » Django (correspondant à l'aspect Vue du MVC) ;
- formulaires ;
- « modèles » Django (correspondant à l'aspect Modèle du MVC) ;
- gestion des sessions ;
- suppléments en Ajax.

7

Les vues Django : orchestration et architecture

Ce chapitre va nous lancer dans l'utilisation du framework Django et nous permettre d'en découvrir les premiers aspects : URL et « Views ». MVC, la recette de conception fondamentale de tout développement web mise en place par Django, commencera à être mieux comprise. Un premier projet Django sera réalisé se bornant à renvoyer une page HTML très élémentaire, un peu comme on le ferait pour une application statique toute basique.

SOMMAIRE

- Le modèle MVC selon Django
- Utilité des vues
- Définition des URL
- Définition des méthodes « vues »
- Création d'un projet Django dans Eclipse
- Réalisation de notre première page web
- Démarrage et test d'un projet dans Eclipse

Django, le framework que nous allons utiliser pour développer notre site web Trombinoscoop, obéit à la philosophie de développement MVC. Comme nous l'avons vu dans les chapitres précédents, cette architecture se propose de découper le développement et le code en trois grandes parties :

Django et le MVC

Bien antérieur à Django, le modèle MVC ne lui est pas spécifique. Il est également proposé par de nombreux autres frameworks tels que Symfony pour PHP.

- Le modèle (*Model*) représente la partie « données » de l'application : leur représentation, leur sauvegarde, leur récupération et leur traitement.
- La vue (*View*) représente l'interface (dans notre cas « graphique ») de l'application. C'est dans la vue que l'on va gérer et définir les écrans HTML de notre applicatif.
- Le contrôleur (*Controller*) agit comme le chef d'orchestre de l'applicatif. Il sert de lien entre le modèle et la vue. Plus concrètement, le contrôleur va recevoir les requêtes HTTP, appeler les méthodes ad hoc du modèle et de la vue, puis retourner la page HTML produite.

Django utilise d'autres termes pour désigner les trois composantes de l'architecture MVC :

- À la place des *vues*, on parlera de *templates*.
- À la place des *contrôleurs*, on utilisera *vues* (*views*). Attention à la confusion…
- Pour le *modèle*, coup de chance ! Django garde le même terme.

EN PRATIQUE Attention : terminologie MTV

Par la suite, nous allons utiliser la terminologie propre à Django : MTV (*Model Templates Views*).

L'objectif de ce chapitre est d'étudier les vues Django. On peut se passer des deux autres composantes (le modèle et les templates) pour réaliser un site web fonctionnel, mais on ne peut en aucun cas se passer des vues.

Ce chapitre sera également l'occasion de commencer le développement de notre site « Trombinoscoop ». Le développement sera très progressif afin de vous permettre d'appréhender pas à pas chacune des composantes de Django. Le premier pas sera une page d'accueil affichant un message de bienvenue. Attendez un peu pour des pages web plus riches en fonctionnalités et en contenu !

Utilité des vues

Pour comprendre précisément l'utilité des vues Django, reprenons le schéma modélisant la visite d'une page web quelconque par un internaute.

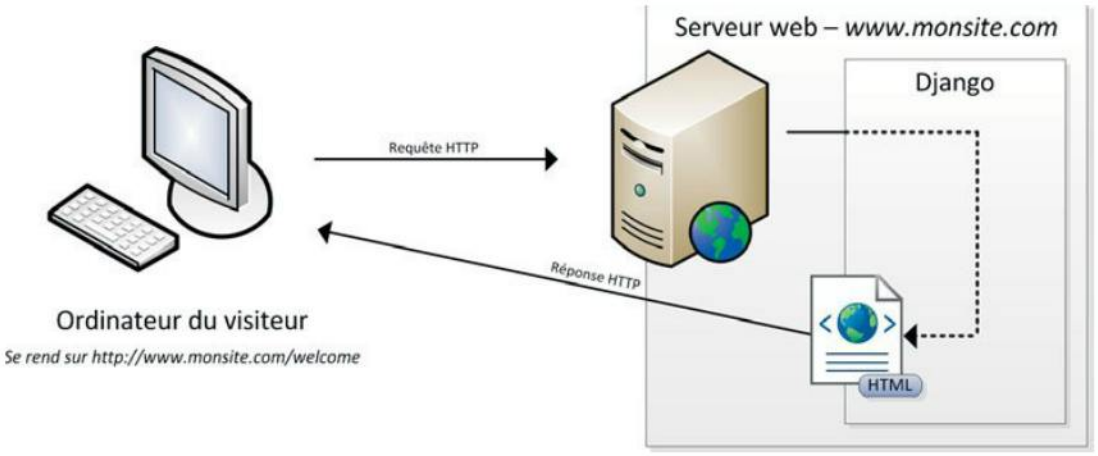


Figure 7–1 Élaboration d'une page dynamique par Django

Imaginons qu'un internaute désire se rendre sur notre site web, et plus précisément sur la page d'accueil de ce site dont le nom est `welcome`. Il entre l'URL http://www.monsite.com/welcome. Nous verrons par la suite par quoi il faut remplacer www.monsite.com dans cette URL, car il est évident que vous ne possédez pas ce nom de domaine. Le navigateur contacte notre serveur web et lui envoie une requête HTTP, dans laquelle on retrouve :

- le nom de la page, dans notre cas `welcome` ;
- les éventuels paramètres passés à la page (aucun dans notre cas, car nous allons pour l'instant laisser cet aspect de côté) ;
- diverses informations au sujet de l'internaute : la version de son navigateur, sa langue, etc.

À la réception de la requête, le serveur web la transmet telle quelle à Django, charge à ce dernier de la décoder et d'agir en conséquence.

Le **premier rôle des vues Django** est alors de lister toutes les pages possibles ou, en d'autres mots, toutes les URL possibles. Dans notre exemple, il va falloir préciser que l'URL welcome existe et qu'il y aura du traitement à faire dessus. Pour chacune des URL possibles, on précise quelle fonction Python doit être appelée pour traiter la requête et, au final, écrire le HTML à renvoyer au navigateur. Les traitements réalisés par ces fonctions constituent le **deuxième rôle des vues Django**.

Le fichier urls.py

La définition des URL possibles se fait dans un fichier nommé `urls.py`. Il contient essentiellement une liste de correspondance entre des URL et des fonctions Python :

EXEMPLE 7.1 Fichier urls.py

```
urlpatterns = patterns('',
  ('^welcome$', myproject.views.welcome),
  ('^login$', myproject.views.login),
  ('^logout$', myproject.views.logout),
  ('^$', myproject.views.welcome)
)
```

`urlpatterns` est une liste Python qui contient toutes les correspondances URL/fonction à appeler. Il est important que cette liste porte ce nom-là : Django attend cette dénomination précise. Par ailleurs, cette liste doit respecter un certain format, d'où l'usage de la fonction `patterns`, qui construit une liste de correspondances dans le format attendu par Django.

Le premier paramètre de la fonction `patterns` est un *prefix*. Laissons-le de côté car nous n'allons pas l'utiliser et, comme vous pouvez le constater, nous le laissons vide. Ensuite, on ajoute à la fonction autant de paramètres qu'il y a d'URL dans notre site (pour notre exemple, nous avons défini quatre URL, donc la fonction prend quatre paramètres en plus du *prefix*), comme ceci :

SYNTAXE. La fonction patterns

```
urlpatterns = patterns(prefix, param1, param2, param3, param4...)
```

Tous ces paramètres supplémentaires sont des *tuples* qui mettent en correspondance chaque URL et la fonction Python à appeler. Ils sont construits sous la forme suivante :

SYNTAXE. Tuples passés à la fonction patterns

```
(URL, fonction à appeler)
```

Que contiennent exactement ces tuples ?

- « URL » est une *expression régulière*. Dans nos exemples, nous utiliserons exclusivement des expressions régulières triviales qui font correspondre une fonction Python à une et une seule chaîne de caractères. Retenez simplement que, dans ce cas, l'expression régulière se résume au nom de la page entouré par les caractères `^` et `$`.
- « Fonction à appeler » est simplement la fonction Python à appeler en réponse à l'appel de cette URL.

DÉFINITION Expression régulière

Il s'agit d'un motif qui permet de décrire une ou *plusieurs* chaîne(s) de caractères possible(s). À l'aide des expressions régulières, on peut, par exemple, exprimer la correspondance suivante « pour toutes les URL qui commencent par `article-` on appelle la fonction Python `afficherUnArticle` ».

À partir de ces explications, on comprend désormais mieux le fichier `urls.py` que nous vous avons donné en exemple. Quatre URL y ont été définies, à l'aide de quatre tuples : le premier pour la page welcome, le second pour la page login, le troisième pour la page logout et le dernier, un peu spécial, pour la page « sans nom ». Cette dernière page est appelée lorsqu'on entre l'adresse suivante dans le navigateur : http://www.monsite.com/ (aucun nom de page n'est précisé). On parle également de page par défaut.

Nous sommes persuadés que vous brûlez d'envie de déjà lancer Eclipse, créer un projet et taper les lignes de code que nous venons de voir dans un fichier `urls.py`. N'en faites rien ! Nous verrons bientôt comment construire pas à pas notre premier projet Django dans Eclipse. Avant tout, attardons-nous sur un autre fichier d'une très grande importance : `views.py`.

Le fichier views.py

Le rôle du fichier `views.py` est de définir toutes les fonctions Python déclarées dans les tuples. Ces fonctions sont appelées des « vues ».

Chacune de ces vues reçoit toujours en entrée un objet représentant la requête HTTP et dont les informations transmises pourront être exploitées. Pour fournir une réponse HTTP à la requête qu'elle a reçue, chaque vue doit construire une page HTML à renvoyer vers l'utilisateur. D'un appel à l'autre de cette même fonction, le HTML renvoyé pourra être différent. Là réside toute la saveur et la subtilité de la mise en place de sites web dynamiques : leur contenu évolue au fil du temps.

Sans plus attendre, passons à un exemple de fichier `views.py`.

EXEMPLE 7.2 Un fichier views.py

```
from django.http import HttpResponse (2)

def welcome(request): (1)
    return HttpResponse('<html><body><p>Bienvenue !</p></body></html>')
```

Dans ce fichier, nous avons défini une fonction nommée `welcome` (1), laquelle sera appelée lorsqu'on entrera l'URL http://www.monsite.com/welcome (comme défini dans `urls.py`).

Cette fonction accepte un paramètre : `request`. Nous ne l'utiliserons pas pour le moment, mais sachez qu'il contient toutes les données de la requête HTTP.

Ensuite, cette fonction retourne un objet de type `HttpResponse` (type que nous avons dû importer d'une bibliothèque Django, d'où la ligne 2). Dans le constructeur de cet objet, nous passons simplement le code HTML à renvoyer sous forme de texte. Nous avons volontairement écrit du HTML simplifié (même invalide) afin de ne pas compliquer l'exemple. C'est aussi simple que cela ; ces trois lignes suffisent à produire notre première page web.

Un dessin vaut souvent mieux qu'un long discours ; résumons donc ce que nous venons de voir sur un schéma.

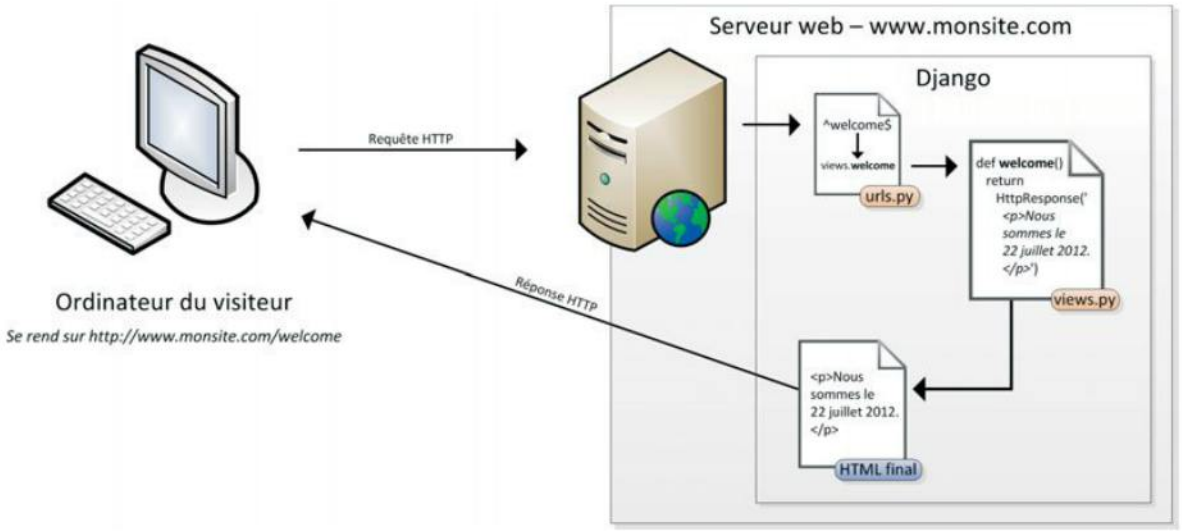


Figure 7–2 urls.py et views.py au sein de Django

On constate qu'à la réception de la requête, Django va avant tout trouver une correspondance entre la page demandée et une vue Python. La vue trouvée, Django va l'appeler, charge à celle-ci de produire en retour le HTML qui sera renvoyé tel quel en réponse.

Jusqu'à présent, l'exemple reste théorique et nous ne l'avons pas encore codé réellement dans notre éditeur Eclipse. De même, le résultat n'est pas encore apparu dans un navigateur. Remédions-y en découvrant comment, très pratiquement, réaliser cette petite application dans Eclipse.

Installation d'Eclipse

Nous expliquons comment installer notre environnement de développement dans l'annexe A.

Enfin ! Notre première page web en Django

Lancement de l'environnement Eclipse et création du projet

Tout d'abord, il nous faut lancer Eclipse. Nous arrivons sur l'écran d'accueil.

Pour démarrer un nouveau projet, cliquez sur le menu *File > New > Project...* Dans la boîte de dialogue qui apparaît, choisissez *PyDev Django Project*.

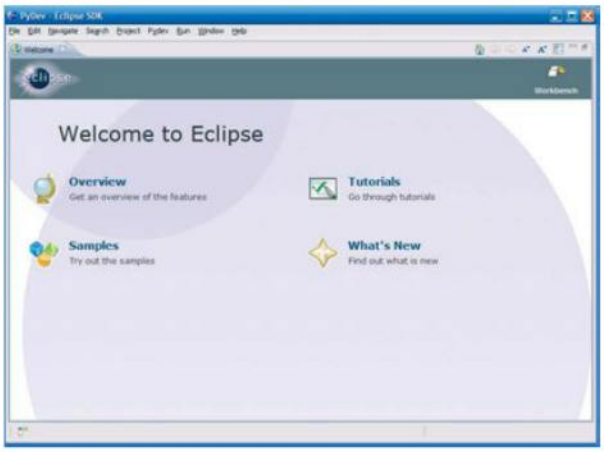


Figure 7–3 Écran d'accueil d'Eclipse

Figure 7–4 Création d'un projet Django

Après avoir cliqué sur *Next*, un assistant de création de projet se lance. À la première étape, il suffit de donner un nom au projet.

Cliquez alors sur *Next*. L'écran suivant demande quelle version de Django utiliser ainsi que quelques paramètres relatifs à la base de données. Les valeurs par défaut peuvent être laissées.

Cliquez sur *Finish*. Le projet est créé et apparaît dans l'écran principal d'Eclipse.

Remarquez que plusieurs fichiers sont déjà créés, dont `urls.py`. C'est normal, car nous avons créé un projet de type « Django » et Eclipse sait quels fichiers doivent obligatoirement figurer dans un tel projet.

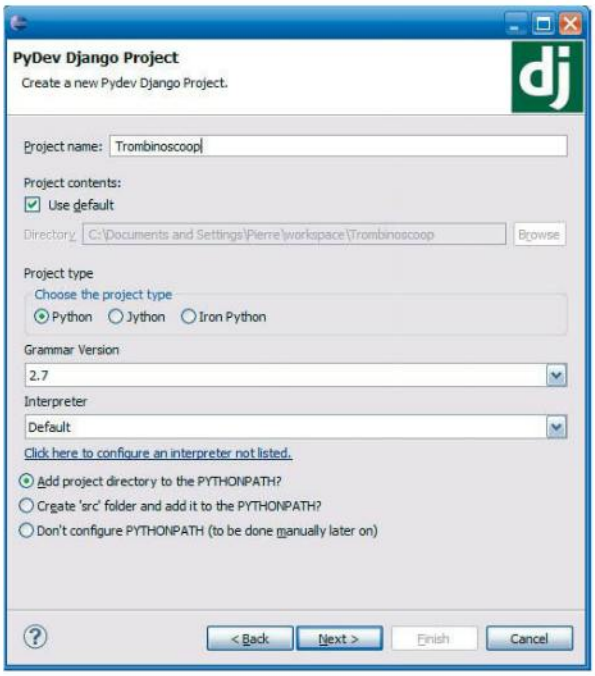


Figure 7–5 Deuxième étape de la création d'un projet

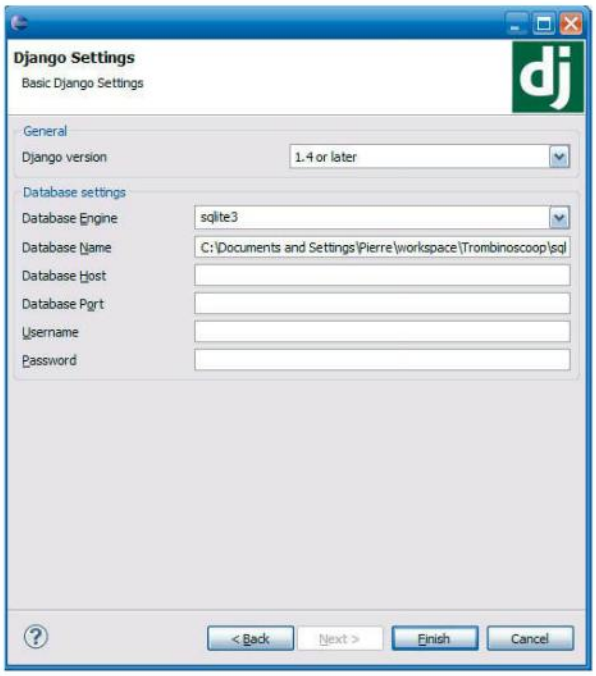


Figure 7–6 Troisième étape de la création d'un projet

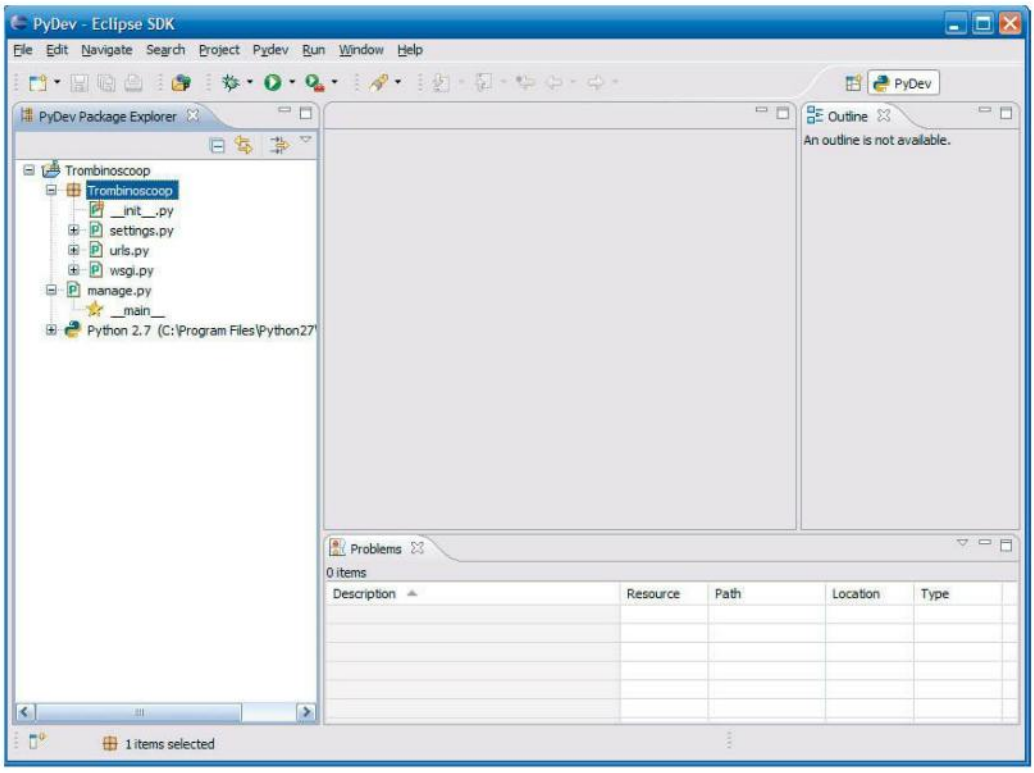


Figure 7–7 Le projet fraîchement créé

EN PRATIQUE Au fait, quid de l'encodage ?

Nous avons vu au chapitre 4 qu'on devait toujours préciser l'encodage de nos fichiers, au risque d'avoir des problèmes avec accents et autres caractères exotiques.
Plus précisément, il faut :

- s'assurer que l'encodage dans lequel l'éditeur (ici Eclipse) travaille est le bon (nous aimerions de l'UTF-8) ;
- s'assurer que le navigateur est bien prévenu de l'encodage utilisé pour nos pages.

Par défaut, Eclipse ne travaille pas en UTF-8. Changeons donc ça !
Rendez-vous dans le menu *Windows > Preferences*. Dans la boîte de dialogue qui apparaît, cliquez sur la catégorie *Web*. Au sein de cette catégorie, sélectionnez successivement les sous-catégories *CSS Files* et *HTML Files* et, pour chacune d'elles, changez le paramètre *Encoding* en *UTF-8*.

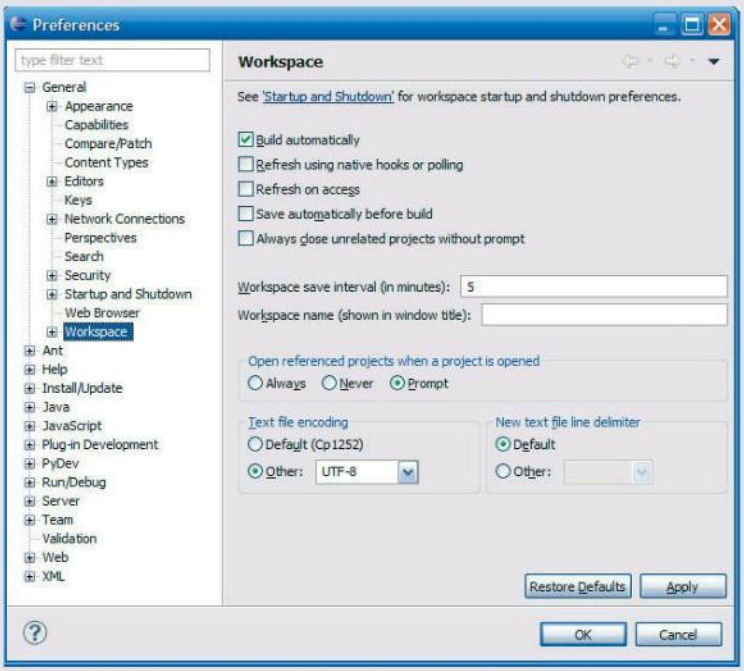


Figure 7–8 Changement de l'encodage par défaut d'Eclipse

Très important, comme nous venons de changer l'encodage par défaut d'Eclipse, tous les fichiers Python créés le seront en UTF-8. Si par malheur vous utilisez dans votre code un caractère accentué ou exotique (donc codé sur deux octets), le compilateur Python ne va rien comprendre et va râler… Pour éviter cela, nous vous conseillons d'ajouter en début de tous vos fichiers Python la ligne suivante :

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

Cela informera le compilateur qu'il a affaire à de l'UTF-8.
Ensuite, il reste à indiquer au navigateur quel est l'encodage des fichiers HTML qu'on va lui renvoyer. Cela se passe dans la configuration de Django ou, en d'autres mots, dans le fichier `settings.py`. Il suffit d'ajouter la ligne suivante :

```
DEFAULT_CHARSET = 'utf-8'
```

Grâce à cette ligne, Django va préciser, dans toutes les réponses HTTP envoyées au navigateur, que l'encodage est de l'UTF-8.

Le fichier urls.py

Nous pouvons maintenant directement éditer le fichier `urls.py` en double-cliquant dessus. Remplacez tout le code préexistant par celui-ci :

EXEMPLE 7.3 Contenu de notre premier fichier urls.py

```
from django.conf.urls import patterns

urlpatterns = patterns('',
  ('^welcome$', welcome)
)
```

À l'aide de ce code, nous avons prévu une seule page « fonction », nommée `welcome`. Un indicateur rouge dans la marge indique qu'Eclipse ne trouve pas la méthode `myproject.views.welcome` :

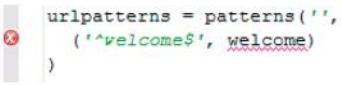


Figure 7–9
Eclipse ne trouve pas la méthode welcome !

Pas de panique ! C'est tout à fait normal : nous ne l'avons pas encore définie et Eclipse ne peut pas la sortir de nulle part.

Le fichier views.py

Le fichier `views.py` n'est pas créé par défaut ; il faut le faire manuellement. Pour cela, cliquez-droit sur le dossier *Trombinoscoop* qui contient nos autres fichiers et choisissez *New > PyDev Module*.

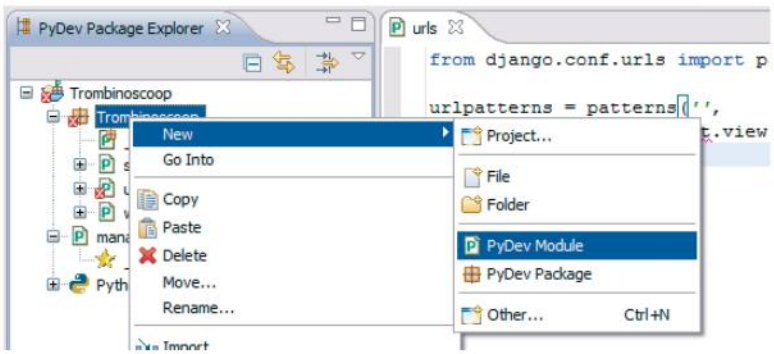


Figure 7–10
Ajout d'un module

Dans la fenêtre qui apparaît, précisez le nom du fichier (`views`) sans l'extension `py`.

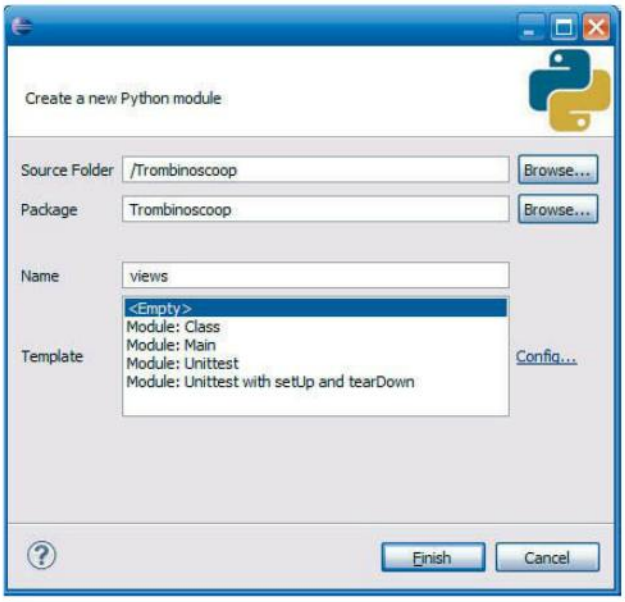


Figure 7–11
Deuxième étape de l'ajout d'un module

Cliquez sur *Finish* et le fichier est créé. Vous pouvez maintenant y ajouter le code suivant :

EXEMPLE 7.4 Contenu de notre premier fichier views.py

```
from django.http import HttpResponse

def welcome(request):
    return HttpResponse('<?xml version="1.0" encoding="UTF-8" ?> \
<!DOCTYPE html> \
<html xmlns="http://www.w3.org/1999/xhtml"> \
  <head> \
    <title>Trombinoscoop</title> \
  </head> \
  <body> \
    <p>Bienvenue sur Trombinoscoop !</p> \
  </body> \
</html>')
```

Nous avons cette fois-ci écrit un HTML plus complet et correct. Pour gagner en lisibilité en ne tassant pas tout sur une seule ligne, on utilise des barres obliques inverses (\) pour signaler que la chaîne de caractères continue à la ligne suivante.

Importation de la fonction dans urls.py

Si vous retournez dans le fichier `urls.py`, vous constatez qu'Eclipse ne trouve toujours pas la fonction `welcome`. En effet, il faut tout d'abord l'importer (ce qu'on fait avec l'instruction 1). Nous obtenons finalement le fichier `urls.py` complet suivant :

EXEMPLE 7.5 Contenu final de notre premier fichier urls.py

```
from django.conf.urls import patterns
from views import welcome 1

urlpatterns = patterns('',
  ('^welcome$', welcome)
)
```

Et voilà ! Tout le code de notre première page a été écrit. Il reste un petit détail à régler avant de tester le résultat.

Configuration de Django pour qu'il écrive du XHTML 5

Notre site, nous le faisons en XHTML 5 et non en HTML 5. Or, le standard XHTML impose que la réponse HTTP construite par le serveur web fasse mention de son contenu écrit en XHTML 5. Nous devons donc retrouver dans l'en-tête de la réponse HTTP la ligne suivante : `Content-Type: application/xhtml+xml`. Comment ajouter cette ligne dans la réponse HTTP ? C'est très simple : il suffit d'ajouter une ligne dans le fichier de configuration de Django. Ce fichier se nomme `settings.py` et a été créé automatiquement à la création de notre projet. Voici la ligne à ajouter ; elle peut être placée n'importe où dans le fichier (mais nous vous conseillons toutefois de la placer au début) :

CONFIGURATION. Le contenu produit par Django est de l'XHTML

```
DEFAULT_CONTENT_TYPE = 'application/xhtml+xml'
```

EN PRATIQUE XHTML5 ou HTML5 ?

XHTML5 existe bel et bien et n'est pas une version mutante et bricolée de XHTML 1.1. Il s'agit simplement d'une syntaxe plus stricte d'HTML5, dont la description est faite dans le standard HTML5. Nous avons choisi à dessein cette syntaxe plus stricte (en XML) de l'HTML5, pour des raisons pédagogiques : pas question pour l'instant d'inventer des balises, ni d'oublier d'en fermer, etc. Nous encourageons une rigueur à la virgule près ! Cela étant dit, dans la pratique, certains font le choix d'écrire leurs pages web en HTML5, tout en s'imposant les rigueurs de l'XHTML5. On appelle cela du « Polyglot markup ». Cela évite de devoir changer la configuration du serveur web pour qu'il annonce au navigateur qu'on travaille en XHTML5 et d'écrire les balises <?xml> en début de document ou d'ajouter l'attribut xmlns dans la balise html.

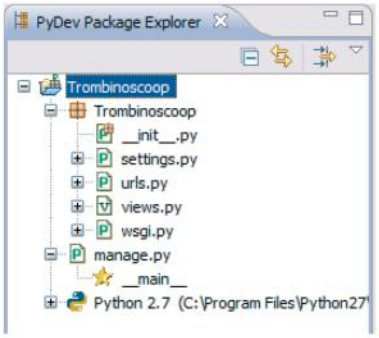
Test de notre ébauche de site

Nous pouvons maintenant tester notre site. Python et Django sont livrés par défaut avec un serveur web léger qui permet de tester très simplement nos merveilleux sites web. Pour lancer ce serveur web, rien n'est plus simple ; cela se passe au niveau de l'icône suivante dans la barre d'outils :

Figure 7–12
Bouton permettant le démarrage du serveur web

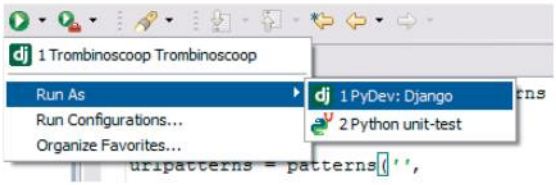
Ne cliquez pas encore dessus, car il faut d'abord se positionner au niveau du projet en le sélectionnant dans l'explorateur de fichiers.

Figure 7–13
Sélection du projet avant de démarrer le serveur web



Ensuite, on peut cliquer sur l'icône « play » verte, ou plutôt sur sa petite flèche annexe qui affiche un menu déroulant. Choisissez *Run As > 1 PyDev: Django*.

Figure 7–14
Démarrage du projet



À ce stade, Eclipse lance une série de validations du code et démarre le serveur web. Si tout s'est bien passé, dans la console on doit lire le texte suivant :

EXEMPLE 7.6 Résultat : validation du code et démarrage du serveur

```
Validating models...

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

EN PRATIQUE En cas d'erreur...

0 erreur, parfait ! Sinon revenez au début du chapitre et assurez-vous que vous nous avez suivis au doigt et à l'œil.

Un message nous indique également par quelle adresse est accessible le serveur web : `http://127.0.0.1:8000/`. Il s'agit en réalité de l'adresse IP locale de *votre* machine. Vous comprenez donc maintenant pourquoi et comment remplacer le http://www.mon-site.com/ que nous avons utilisé de manière fictive tout au long de ce chapitre.

Notez également que l'adresse proposée dans la console contient *un numéro de port* (8000). D'habitude, lorsqu'on consulte un site web, on ne spécifie pas ce numéro de port. Par défaut en effet, les navigateurs utilisent le port 80 et les serveurs web sont configurés pour tourner sur ce dernier. Cependant, comme ici le serveur web exploite un autre port, il faut le spécifier.

Donc, pour tester notre premier site web, il suffit de lancer un navigateur et de se rendre à l'adresse suivante : `http://127.0.0.1:8000/welcome`. Le rendu sera le suivant :

Figure 7–15
Résultat de notre première page web !

Bel effort, mais…

Ce premier site web, ou plutôt cette première page web, est certes un bel effort mais deux remarques s'imposent :

- La page web est entièrement statique, il n'y a aucun élément variable. On aurait très bien pu obtenir exactement le même résultat en écrivant du code HTML dans un fichier `.html`. Or, rappelons-le, notre objectif est de réaliser des sites *dynamiques*, pas du HTML amorphe.
- Le code HTML de la page se trouve au milieu du code Python, ce qui veut dire qu'il faut modifier le programme Python si on veut modifier des éléments purement esthétiques. De plus, cela ne permet pas d'éditer le code HTML de manière conviviale. Enfin, si deux personnes différentes doivent s'occuper l'une du Python et l'autre du HTML, elles vont très probablement finir par s'étriper sur ce fichier.

Les *templates* Django permettent de pallier ces problèmes et de séparer le code HTML du code Python, tout en laissant la possibilité d'insérer dans le code HTML des éléments variables. C'est l'objectif du prochain chapitre.

Ai-je bien compris ?

- Quel changement apporte Django au sigle « MVC » ?
- À quoi servent les fichiers `urls.py` et `views.py` ?
- Dans ce chapitre, quelles sont les composantes du modèle MVC que nous avons implémentées ? Quelles sont celles que nous n'avons pas implémentées ?

8

Les templates Django : séparation et réutilisation des rendus HTML

Tout en continuant le développement de notre projet « Trombinoscoop », ce chapitre aide à comprendre comment la mise en place des « templates » Django tient les instructions HTML (dédiées à la présentation du site web) séparées des instructions Python (plus centrées sur son fonctionnement). Le rôle et l'utilisation des templates constituent l'essentiel de ce qui va suivre.

SOMMAIRE

- Introduction aux templates de Django qui séparent au mieux le HTML du code Python
- Description de tout ce que l'on peut faire et produire à partir de ces templates
- La suite de notre projet Trombinoscoop

Au chapitre précédent, nous avons étudié une des premières composantes du modèle MTV (*Model Templates Views*) de Django, à savoir les vues. Cela nous a permis de réaliser une première page, certes plutôt basique, de notre site web. Cependant, notre code est entaché de l'affreux mélange de HTML et de Python. Heureusement, les templates Django sont là pour nous aider à tenir séparés les deux types de codes. Ils conduisent à réutiliser facilement les lignes de HTML et à éviter des duplications intempestives de code.

L'objectif de ce chapitre est d'étudier les templates Django et de les appliquer à la page de login de notre site web Trombinoscoop.

Principe des templates

Les templates Django sont avant tout des fichiers HTML. Toutefois, nous serions face à un problème de taille s'ils ne se réduisaient qu'à du HTML stupidement statique ! C'est pourquoi les templates introduisent une notion « d'éléments variables » qui ajoutent un certain dynamisme au HTML produit.

Imaginons une page de bienvenue dans laquelle apparaîtrait le message « Bienvenue Pierre », Pierre étant le prénom de l'internaute connecté. Créons le template de notre page de bienvenue :

EXEMPLE 8.1 Template d'une page de bienvenue personnalisée

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="fr">
  <head>
    <title>Bienvenue sur notre site</title>
  </head>
  <body>
    <p>Bienvenue {{ logged_user_name }}</p>
  </body>
</html>
```

Ce template ressemble à s'y méprendre à un simple fichier HTML. Néanmoins, ouvrez l'œil et regardez plus attentivement : l'élément `{{ logged_user_name }}` devrait retenir votre attention. Les doubles accolades indiquent qu'il s'agit d'une variable que Django va interpréter et remplacer par sa vraie valeur au moment de l'utilisation du template.

L'utilisation d'un template s'avère très simple. L'idée est de nettoyer les views de ce HTML inopportun. Nous n'utiliserons pas le code suivant présenté au chapitre précédent :

EXEMPLE 8.2 Production de HTML directement dans la vue

```
def welcome(request):
    return HttpResponse('<html><body><p>Bienvenue' + logged_user_name
                        ➥ + '</p></body></html>')
```

À la place, nous allons plutôt tirer profit de la fonction `render_to_response` (1), à laquelle on passe en paramètre un nom de template, suivi de la valeur de tous les paramètres qui jalonnent le template. Le code HTML a disparu comme par magie de l'utilisation des views.

EXEMPLE 8.3 Appel à un template

```
def welcome(request):
    return render_to_response (1)('my_template.html',
                                  ➥ {'logged_user_name': 'Pierre'})
```

Réalisons maintenant notre premier template.

Notre premier template

Nous allons repartir du code que nous avons laissé au chapitre précédent et faire migrer tout le HTML dans un template afin d'illustrer le rôle et l'utilisation de ce dernier.

EN PRATIQUE Installation supplémentaire : Web Developer Tools

Si vous n'avez pas encore installé les « Web Developper Tools », c'est le moment ! Voyez la manipulation en annexe.

EN PRATIQUE Configuration : encore une histoire d'encodage !

Deux éditeurs viennent d'être ajoutés : un pour le HTML, un pour les CSS. Il va falloir les configurer pour qu'ils produisent du code en UTF-8.
Cela se fait dans le menu *Windows > Preferences*. Dans la boîte de dialogue qui apparaît, rendez-vous dans la catégorie *General > Workspace* et changez le paramètre *Text file encoding* en *UTF-8*.

Élaborons notre premier template. Commençons par créer un nouveau dossier à la racine de notre projet, qui nous servira à stocker tous nos templates. Cliquez-droit sur le paquet *Trombinoscoop* qui se trouve dans le projet et, dans le menu déroulant, choisissez *New > Folder*.

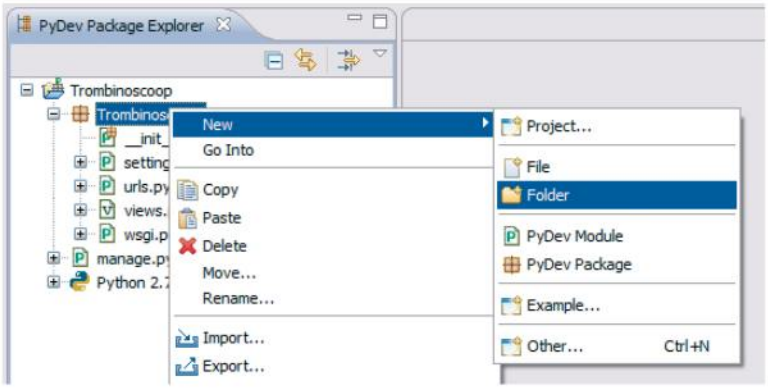


Figure 8–1
Création du dossier qui contiendra nos templates

Nommons ce dossier *templates*. Il faut ensuite expliquer à Django où chercher les templates : cela se définit dans le fichier `settings.py` qui se trouve déjà dans le projet :

CONFIGURATION. Spécification du dossier dans lequel se trouvent les templates

```
import os
PROJECT_ROOT = os.path.abspath(os.path.dirname(__file__)) 1

TEMPLATE_DIRS = (
  os.path.join(PROJECT_ROOT, 'templates'),
) 2
```

Les deux premières lignes 1 sont à placer au tout début du fichier `settings.py`. La définition que nous vous proposons de la variable `TEMPLATE_DIRS` 2 doit remplacer celle qui existe déjà dans le fichier.

Nous pouvons maintenant créer notre premier template : cliquez-droit sur le dossier que nous venons de créer et choisissez *New > File*. Donnez-lui le nom `welcome.html`. Dans ce fichier, recopiez tout le code HTML que nous avions placé dans `views.py`, à savoir :

EXEMPLE 8.4 Code HTML du template welcome.html

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Trombinoscoop</title>
```

```
  </head>
  <body>
    <p>Bienvenue sur Trombinoscoop !</p>
  </body>
</html>
```

Il ne nous reste plus maintenant qu'à modifier la vue afin que celle-ci fasse appel à notre template. Pour ce faire, remplacez le code de `views.py` par le suivant :

EXEMPLE 8.5 Appel du template welcome.html dans la vue

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

from django.shortcuts import render_to_response

def welcome(request):
    return render_to_response('welcome.html')
```

Et voilà ! Nous pouvons maintenant redémarrer notre projet pour voir ce que cela donne, en cliquant sur l'icône .

EN PRATIQUE Stopper et redémarrer l'application web

Si votre application était encore démarrée, il faut d'abord l'arrêter en cliquant sur l'icône .

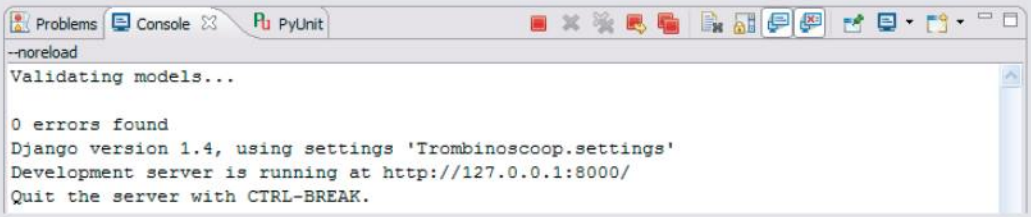


Figure 8–2 Barre d'outils permettant notamment de stopper et redémarrer un serveur

Une fois l'application stoppée, vous pourrez la démarrer à nouveau à l'aide de l'icône verte.
Pour aller plus vite, vous pouvez stopper et redémarrer votre application en un coup en cliquant sur l'icône .

Testons l'URL dans un navigateur. Nous obtenons exactement le même résultat qu'au chapitre précédent… si ce n'est que maintenant, notre code HTML n'est plus mélangé à celui de Python ! Cela fait une énorme différence !

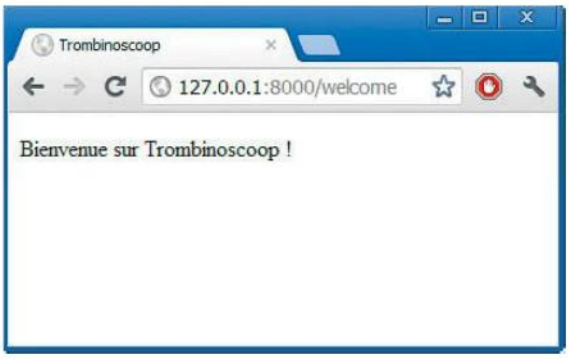


Figure 8–3
Le rendu est le même qu'au chapitre précédent.

Dynamisons ce premier template

Pour l'instant, notre template n'est autre qu'un fichier HTML aussi statique qu'un escalier en marbre blanc. Voyons maintenant comment y inclure du contenu dynamique : par exemple, la date et l'heure courante. Dans notre template, cette information temporelle va se retrouver en variable (1).

EXEMPLE 8.6 Ajout d'une variable dans welcome.html

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Trombinoscoop</title>
  </head>
  <body>
    <p>Bienvenue sur Trombinoscoop ! Nous somme le {{ current_date_time }}. (1)</p>
  </body>
</html>
```

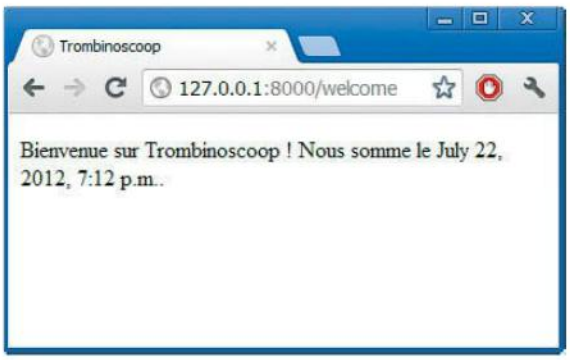
Au niveau de notre vue, nous allons transmettre la date et l'heure courante au template sous forme de paramètre (2).

EXEMPLE 8.7 Transmission d'un paramètre au template welcome.html

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

from django.shortcuts import render_to_response
from datetime import datetime (3)

def welcome(request):
    return render_to_response('welcome.html',
                              {'current_date_time': datetime.now} (2))
```

N'oubliez pas le nouvel import (3) pour que Python puisse jouer avec dates et heures. Voici le résultat dans un navigateur :

Figure 8–4
La date est affichée de manière dynamique.



La date et l'heure sont affichées de façon dynamique : la page change maintenant à chaque visite.

Ce premier exemple de template dynamique est l'occasion de résumer le fonctionnement de Django, tel que nous l'avons vu jusqu'à présent. Schématiquement, voilà ce qui se passe lorsqu'on nous visitons la page `welcome` :

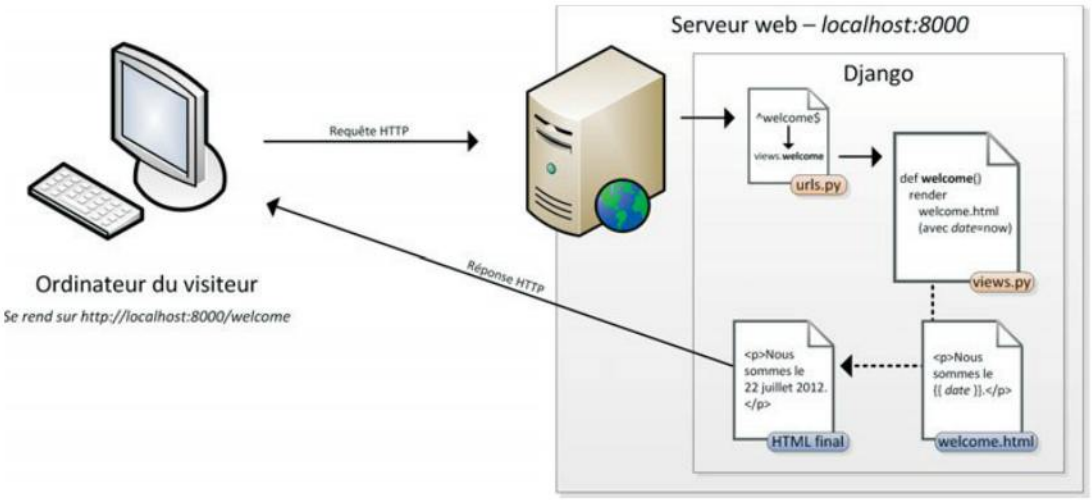


Figure 8–5 La date est affichée de manière dynamique.

Lorsqu'on visite l'URL http://localhost:8000/welcome, le navigateur envoie une requête HTTP au serveur web `localhost` pour lui demander la page `welcome`. Immédiatement après avoir reçu la requête, le serveur web passe la main à Django. Le traitement commence dans le fichier `urls.py` où l'on va tenter de trouver une correspon-

dance entre l'URL demandée et une fonction de `views.py` : dans notre cas, `views.welcome` doit être appelée.

La fonction `views.welcome` appelle alors le template `welcome.html` en lui passant en paramètre la date du jour. Le moteur Django remplace dans le template la variable `date` (ainsi nommée dans le schéma, mais appelée `current_date_time` dans le code) par cette valeur transmise. Le HTML final est enfin prêt à être renvoyé en réponse au navigateur client. Superbe division du travail : personne n'empiète sur les plates-bandes des autres.

Le langage des templates

Nous avons expliqué comment insérer une variable dans un template, afin qu'elle puisse être dynamiquement remplacée par une valeur. Ce n'est pas la seule construction possible, il existe d'autres instructions pour dynamiser les templates. Les concepteurs de Django appellent cela le « langage des templates ». En voici un petit tour d'horizon.

Les variables

Nous avons déjà vu que les variables étaient entourées de doubles accolades :

SYNTAXE EXEMPLE. Variable dans un template

```
<p>Bonjour {{ nom }} !</p>
```

Notons que les variables peuvent être des attributs d'objets Python, par exemple :

SYNTAXE EXEMPLE. Variable utilisant un attribut d'un objet dans un template

```
<p>Bonjour {{ personne.nom }} !</p>
```

On peut également cibler le $x^{\text{-ième}}$ élément d'une liste :

SYNTAXE EXEMPLE. Impression du deuxième élément d'une liste

```
<p>Bonjour {{ ma_liste.2 }} !</p>
```

Formatage des variables

Les variables, avant d'être incluses dans le HTML final, peuvent être formatées à l'aide de filtres. Par exemple, si l'on désire convertir une variable en minuscules, on écrira :

SYNTAXE EXEMPLE. Le nom est imprimé en minuscule

```
<p>Bonjour {{ nom|lower }} !</p>
```

Ainsi, si le nom transmis au template est `ALEXIS`, le HTML retourné sera :

EXEMPLE 8.8 HTML résultant

```
<p>Bonjour alexis !</p>
```

On peut également convertir le premier élément d'une liste en minuscules, à l'aide de ce filtre :

SYNTAXE EXEMPLE. Convertit le premier élément de la liste en minuscules

```
<p>Bonjour {{ ma_liste|first|lower }} !</p>
```

Dernier exemple de filtre, on peut afficher la longueur d'une variable, comme ceci :

SYNTAXE EXEMPLE. Donne la longueur de la variable

```
<p>Votre nom fait {{ nom|length }} caractères !</p>
```

Il existe de nombreux autres filtres. Vous en trouverez la liste exhaustive dans la documentation (très bien faite !) de Django disponible sur le site officiel.

Sauts conditionnels et boucles

Il est également possible de prévoir des sauts conditionnels et des boucles dans des templates Django. C'est très pratique si l'on désire, par exemple, afficher tous les éléments d'une liste : on va boucler sur chacun d'eux. Au niveau du code, un `if` donnera ceci :

SYNTAXE EXEMPLE. Un saut conditionnel dans un template

```
{% if personne.sexe == "F" %}
  <p>Chère Madame {{ nom }},</p>
{% else %}
  <p>Cher Monsieur {{ nom }},</p>
{% endif %}
```

Remarquez en passant les `{% ... %}` pour entourer les instructions alors que nous avions des `{{ ... }}` pour évaluer des variables.

On peut également utiliser les opérateurs booléens `and`, `or` et `not` pour construire des conditions plus complexes.

Pour écrire une boucle, on écrira :

SYNTAXE EXEMPLE. Une boucle dans un template

```
<ul>
{% for livre in livres %}
<li>{{ livre.nom }}</li>
{% endfor %}
</ul>
```

Ce code est assez simple à comprendre : nous itérons sur chaque élément d'une liste de livres. Et pour chacun, nous imprimons un élément `<li>` HTML.

Nous insistons sur le caractère élémentaire des instructions des templates : c'est entièrement voulu par Django et la philosophie MTV. Seuls des traitements simples doivent être réalisés dans les templates. Les traitements Python compliqués sont à reporter dans les vues et les modèles. On ne le répètera jamais assez : template = aspect, control ou views = intelligence.

Héritage et réutilisation de templates

La puissance des templates Django réside dans la possibilité de créer des templates génériques qui contiennent tout le HTML commun à un ensemble de pages. En effet, en-têtes, pieds de page, menus sont souvent identiques d'une page à l'autre dans un même site web. Afin de s'épargner de la besogne, le code HTML de ces parties-là sera placé dans une page de base. Les pages de contenu, quant à elles, dériveront ou hériteront chacune de cette page de base et ne contiendront que le code HTML du contenu qui leur est propre.

Ici, un exemple vaut mieux qu'un long discours. Imaginons que dans notre site web figurent les deux pages suivantes.

EXEMPLE 8.9 Une première page

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Bienvenue</title>
  </head>
  <body>
    <p>Bienvenue sur Trombinoscoop !</p>
  </body>
</html>
```

EXEMPLE 8.10 Une deuxième page

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Copyright</title>
  </head>
  <body>
    <p>Ce site est soumis aux lois du Copyright.</p>
  </body>
</html>
```

Les seules différences entre ces deux pages sont mises en exergue. Nous allons créer un template `base.html` contenant le code commun.

EXEMPLE 8.11 Template base.html constitué du code commun

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title></title>
  </head>
  <body>
    <p></p>
  </body>
</html>
```

Dans ce template de base, nous allons définir des « blocs » qui pourront être redéfinis par les templates « enfants » qui en dérivent. Utilisons pour cela deux instructions `block` : l'une pour accueillir le titre de la page (1) et l'autre pour accueillir le contenu (2).

EXEMPLE 8.12 Template base.html avec des « blocs » à personnaliser par les templates enfants

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>{% block title %}{% endblock %} (1)</title>
  </head>
  <body>
    <p>{% block content %}{% endblock %} (2)</p>
  </body>
</html>
```

Nos deux pages seront ensuite redéfinies comme suit.

EXEMPLE 8.13 La première page redéfinie

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

{% block title %}Bienvenue{% endblock %}

{% block content %}Bienvenue sur Trombinoscoop !{% endblock %}
```

EXEMPLE 8.14 La deuxième page redéfinie

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

{% block title %}Copyright{% endblock %}

{% block content %}Ce site est soumis aux lois du Copyright.{% endblock %}
```

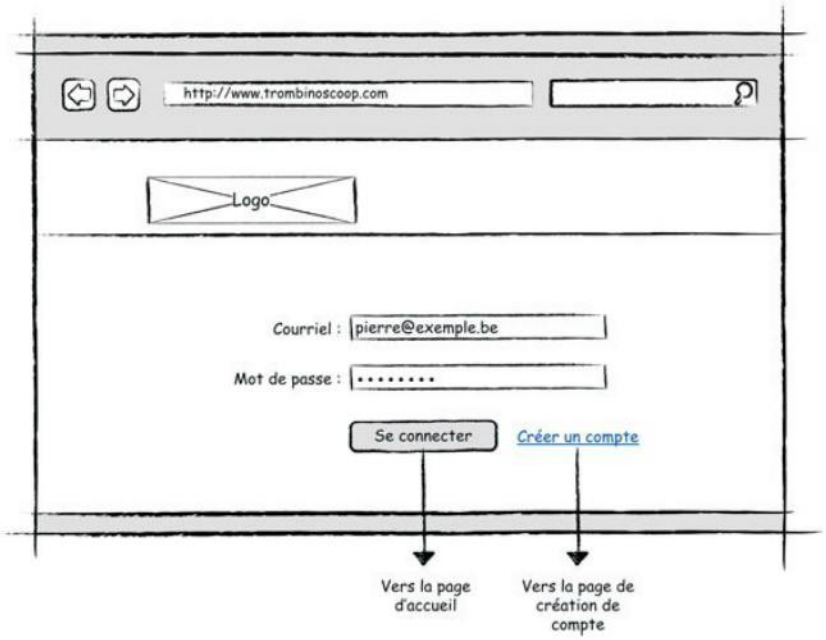
Comment être insensible à cette économie !

Et si on avançait dans la réalisation de Trombinoscoop ?

Pour l'instant, à l'exception d'une maigre page de bienvenue, les visiteurs de notre site n'ont pas beaucoup à se mettre sous la dent. Mettons donc en pratique ce que nous avons appris pour réaliser complètement la page de login.

Pour rappel, voici le *wireframe* de la page de login que nous devons implémenter :

Figure 8–6
Wireframe de la page de login



Commençons par nous occuper du HTML de cette page. Comme nous l'avons signalé plus haut, nous allons utiliser les fonctionnalités d'héritage de Django afin de placer le code HTML commun à toutes les pages dans un template séparé. Allons-y.

Dans le dossier `templates` du projet, créez un nouveau fichier nommé `base.html`. Copiez-y le code suivant :

EXEMPLE 8.15 Trombinoscoop. Contenu du fichier base.html

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Trombinoscoop - {% block title %}Bienvenue{% endblock %}</title> 6
  </head>
  <body 1 id="{% block bodyId %}genericPage{% endblock %}"> 5
    <header> 2
    {% block headerContent %}{% endblock %} 4
    </header>
    <section id="content"> 3
    {% block content %}{% endblock %} 4
    </section>
  </body>
</html>
```

Quelques explications s'imposent. Analysons tout d'abord la structure du contenu du `body` 1. Elle est assez simple : nous avons simplement un en-tête 2 suivi d'une section nommée `content` 3. Le contenu des balises `header` et `section` sera précisé par les templates enfants, d'où la présence d'instructions `block` 4. Par défaut, ces blocs sont vides.

À notre balise `body`, nous ajoutons un `id` égal à `genericPage` 5, mais qui peut être modifié par un template enfant, de nouveau grâce à l'instruction `block`. Nous insistons sur la *possibilité* et non l'*obligation* qui est offerte à l'enfant de redéfinir le bloc. Si l'enfant ne redéfinit pas le bloc `bodyId`, alors c'est la valeur `genericPage` qui sera utilisée : on peut donc la voir comme une valeur par défaut.

Pourquoi avoir ajouté un tel `id` au `body` ? En fait, cela nous sera très utile dans nos instructions CSS, afin d'y écrire des instructions qui ne s'appliquent qu'à une page et pas à une autre.

Enfin, un dernier élément mérite une explication : le titre de la page. Chaque page aura un titre qui commence par « Trombinoscoop – » 6. Les templates enfants pourront compléter ce titre à leur guise. Par défaut, le titre est complété par « Bienvenue ».

Nous pouvons maintenant passer à l'écriture du HTML de la page de login. Créez un fichier nommé `login.html` dans le dossier `templates` et ajoutez-y le code suivant :

EXEMPLE 8.16 Trombinoscoop. Contenu du fichier login.html

```
{% extends "base.html" %} (1)

{% block title %}Connexion{% endblock %} (2)

{% block bodyId %}loginPage{% endblock %} (3)

{% block content %}
<form action="." method="get"> (5)
  <p>
    <label for="email">Courriel :</label> (4)
    <input name="email" id="email" size="30" type="email" />
  </p>
  <p>
    <label for="password">Mot de passe :</label> (4)
    <input name="password" id="password" size="30" type="password" />
  </p>
  <p>
    <input type="submit" value="Se connecter"/>
    <a href=" ">Créer un compte</a> (6)
  </p>
</form>
{% endblock %}
```

Première instruction, `extends` signifie à Django que l'on va dériver ce nouveau template de `base.html`, plus générique (1). Ensuite, nous donnons le titre « Connexion » à notre page (2) et spécifions un `id` pour la balise `body` (`loginPage`) (3).

Arrive le contenu réel de notre page de login, avec essentiellement un formulaire et des champs de connexion. Le formulaire contient trois paragraphes : un pour le champ `Courriel`, un autre pour le champ `Mot de passe` et un dernier pour le bouton de connexion et le lien vers la page de création de compte. Les deux champs textuels sont précédés par une balise `label`, qui entoure un texte représentant l'intitulé du champ (4). Par exemple, l'intitulé du champ dont l'`id` est `email` est « Courriel : ». Enfin, notez que nous n'avons pas encore précisé l'attribut `action` (5) du formulaire, ni la destination du lien hypertexte permettant de se créer un compte (6). Nous nous en occuperons plus tard.

Nos HTML sont maintenant fin prêts. Il reste juste à ajouter une URL dans le fichier `urls.py` et une vue. Dans le fichier `urls.py`, nous proposons d'ajouter en réalité deux URL (1) :

EXEMPLE 8.17 Trombinoscoop. Contenu du fichier urls.py

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

from django.conf.urls import patterns
from views import welcome, login (2)

urlpatterns = patterns('',
  ('^$', login), (1)
  ('^login$', login), (1)
  ('^welcome$', welcome)
)
```

Pourquoi deux URL pointant vers la même vue ? La première représente en réalité l'URL « vide » (http://localhost:8000/), celle qui est appelée sans aucun nom de page. La deuxième est celle qui nomme explicitement la page login.

Notez que nous avons également importé la fonction `login` du module `views` (2) et qu'il nous faut maintenant l'implémenter. Voici donc les lignes que nous ajoutons dans le fichier `views.py`.

EXEMPLE 8.18 Trombinoscoop. Lignes à ajouter dans le fichier views.py

```
def login(request):
  return render_to_response('login.html')
```

Tout est maintenant mis en place pour obtenir une première ébauche de notre page de login. Voyons à quoi elle ressemble dans un navigateur.

Figure 8–7
Première ébauche de la page de login

C'est simple, c'est efficace, mais c'est laid ! Pour améliorer la page et la rendre plus fidèle au wireframe, il va falloir composer avec des CSS.

Amélioration visuelle de la page de login

Un site web, c'est une identité visuelle. Il nous faut un logo, dont nous avions prévu l'emplacement dans le wireframe, là où se trouve la bannière.

Sortons crayons, gomme et papier, ou lançons notre logiciel graphique favori, Gimp ou Photoshop. Après quelques clics, des litres de café noir, l'angoisse de la page blanche et tutti quanti… voici le résultat. Il va de soi que toute ressemblance avec un site existant ou ayant existé serait purement fortuite.

Figure 8–8
Le logo de Trombinoscoop

Doté de ce logo choc, nous pouvons commencer à écrire nos feuilles de styles. Créons tout d'abord un fichier `style.css` qui contiendra toutes nos instructions CSS. Mais au fait, où créer ce fichier ? Le placer dans l'arborescence du projet ne suffit pas, car le navigateur web ne saura pas à quelle URL le récupérer.

Les fichiers CSS, tout comme les images ou les fichiers JavaScript, sont des fichiers statiques. Pour ces fichiers-là, Django propose un mécanisme simple : il suffit de préciser dans la configuration du framework le dossier où nous placerons tous les fichiers statiques.

Rendez-vous donc dans le fichier de configuration de Django (`settings.py`) et complétez les sections suivantes :

CONFIGURATION. Emplacement des fichiers statiques

```
STATIC_URL = '/static/'

STATICFILES_DIRS = (
    os.path.join(PROJECT_ROOT, 'static'),
)
```

Ces lignes signifient deux choses :

- D'une part, les fichiers statiques seront accessibles via les URL commençant par http://localhost:8000/static/.
- D'autre part, ils seront placés dans un dossier nommé `static` que nous mettrons à la racine du projet.

Créons le dossier `static` à la racine du projet, au même niveau que le dossier `templates`. Créons également un sous-dossier par type de fichier statique : `img`, `css` et `js`. Nous obtenons donc l'arborescence suivante, dans laquelle nous avons ajouté notre logo et le fichier `style.css` :

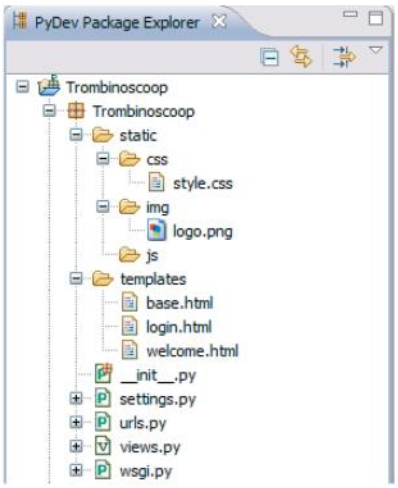


Figure 8–9
Arborescence des fichiers et dossiers du projet

Attaquons maintenant le CSS et commençons par styliser notre bannière. Pour l'instant, celle-ci n'apparaît pas, le contenu de la balise `header` étant vide. Donnons donc à cette balise une taille artificielle, colorons son fond en bleu et ajoutons notre logo :

EXEMPLE 8.19 Trombinoscoop. Description de la bannière dans style.css

```
header {
  position: absolute;
  left: 0;
  top: 0;
  height: 80px;
  width: 100%;
  background: #3b5998 url('../img/logo.png') 1 no-repeat 50px 25px;
  border-bottom: 1px solid #133783;
}
```

Vous pouvez constater que l'URL de notre image de fond est un chemin relatif par rapport à l'emplacement du fichier CSS 1 (les deux points permettent de « remonter » d'un répertoire). Il en est toujours ainsi pour les chemins relatifs mis dans des CSS.

Profitons également de ce premier travail dans `style.css` pour supprimer les marges du `body` et changer la police de notre document, ainsi que sa couleur :

EXEMPLE 8.20 Trombinoscoop. Marge du body et police du document

```
body {
  margin: 0;
  padding: 0;
  font-family: "Lucida Grande", Tahoma, Verdana, Arial, sans-serif;
  font-size: 11px;
  color: #000333;
}
```

Il reste à référencer le fichier CSS dans le HTML de nos pages. Dans le template `base.html`, ajoutez la ligne suivante au sein de la balise `<head>` :

EXEMPLE 8.21 Trombinoscoop. Référencement du fichier CSS dans base.html

```
<link rel="stylesheet" type="text/css" href="/static/css/style.css" />
```

EN PRATIQUE

Bien entendu, on pourrait remplacer `href="/static/css/style.css"` par un lien vers une variable : `href="{{STATIC-URL}}"`.

Nous pouvons maintenant tester ce que tout cela donne.

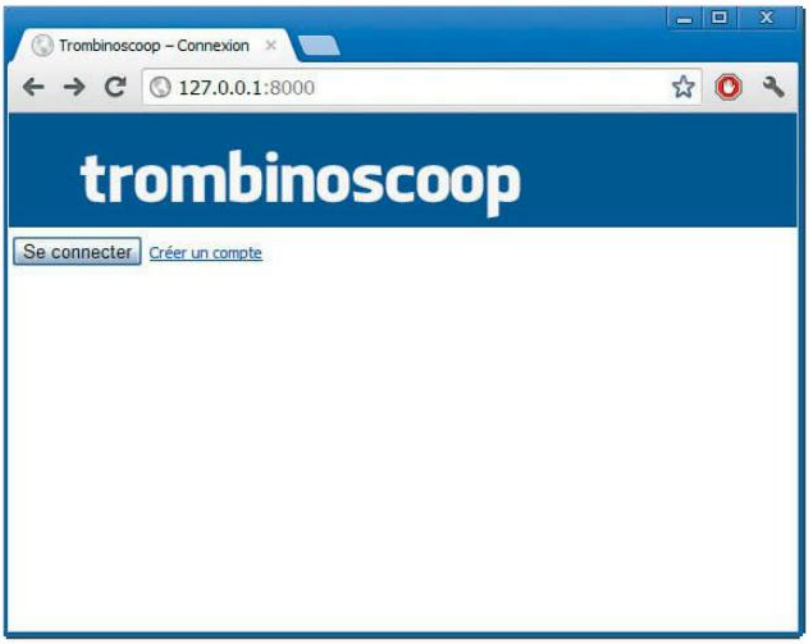


Figure 8–10
Rendu de notre bannière

C'est déjà beaucoup plus sympathique. Malheureusement, notre formulaire se trouve caché par la bannière. C'est logique, puisque cette dernière a été placée en position absolue. Le contenu de la page, qui lui est resté dans le flux courant de positionnement, se retrouve derrière.

Remédions à cela et profitons de l'occasion pour améliorer l'aspect du formulaire selon notre goût, à l'aide du code suivant.

EXEMPLE 8.22 Trombinoscoop. Positionnement et embellissement du formulaire dans style.css

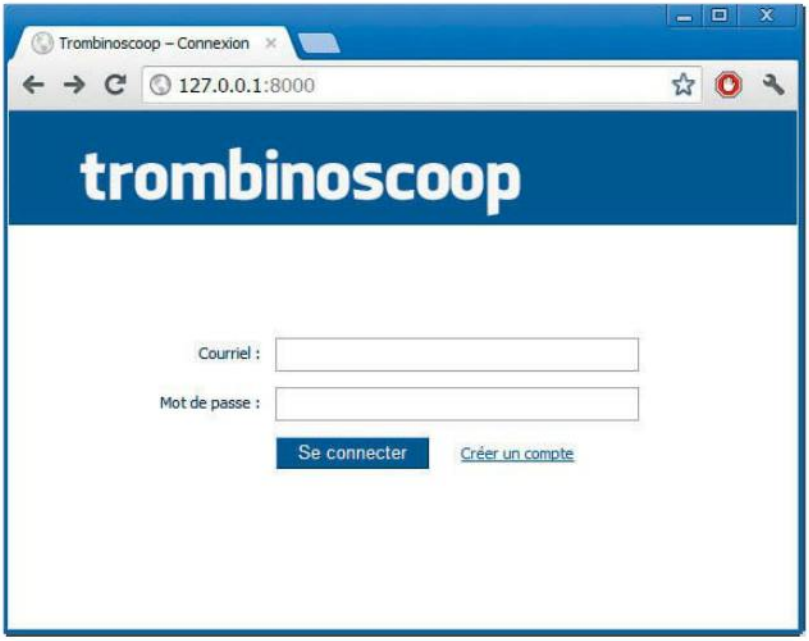
```
body#loginPage section#content { (1)
  position: absolute; (2)
  top: 150px; (3)
  left: 50%; (5)
  width: 500px; (4)
  margin: 0 0 0 -250px; (6)
  padding: 0;
}

form label { (7)
  display: block;
  width: 150px;
  float: left; (8)
  text-align: right;
  padding: 4px 10px 0 0;
}

form input[type="email"], form input[type="text"],
form input[type="password"]{
  display: block;
  width: 250px; (9)
  padding: 2px;
}

form input[type="submit"] { (10)
  margin: 0 20px 0 160px;
  border-style: solid;
  border-width: 1px;
  border-color: #d9dfea #0e1f5b #0e1f5b #d9dfea;
  background-color: #3b5998;
  color: #FFFFFF;
  padding: 2px 15px 3px 15px;
  text-align: center;
}

a { (11)
  color: #3b5998;
}
```

On s'occupe tout d'abord de la `section` d'`id content` de la page `loginPage` (1). On la positionne en absolu (2), décalée vers le bas (3) pour laisser la place à la bannière. On lui donne une largeur de 500 pixels (4) (la boîte de login ne devant pas être très large). Petite astuce, on positionne cette boîte avec un `left` à 50 % (5) et on lui donne une marge négative de 250 pixels (6). Cela permet de centrer la boîte horizontalement : le 50 % positionne le bord gauche de la boîte juste au milieu de la page et la marge négative décale la boîte vers la droite de la moitié de sa largeur.

On s'occupe ensuite des `label` (7) du formulaire. On les fait flotter à gauche, à l'aide de la propriété `float` (8), qui retire le `label` du flux de positionnement courant et invite les éléments successifs restant dans le flux (en l'occurrence ici les `input`) à se positionner autour de la boîte flottante, et non pas à rester cachés derrière.

Ensuite on modifie légèrement la taille des `input` de type `text` (9). Pour finir, on habille le bouton de soumission du formulaire (10) et on colore les liens hypertextes (11). Le résultat est visible sur la figure suivante.

Figure 8–11
Rendu de la page de login



Notre page est certes bien plus jolie, mais pas très fonctionnelle. Si par mégarde on clique sur le bouton *Se connecter* (avouez que c'est tentant !), c'est la Bérézina. En effet, nous avons laissé des points d'interrogation dans l'attribut `action`. Dès le chapitre suivant, nous allons étudier comment gérer les formulaires.

Ai-je bien compris ?

- Citez deux avantages des templates.
- Un template désire réutiliser du HTML déjà écrit dans un autre. Comment peut-il réutiliser ce code ?
- Pouvez-vous décrire le cheminement dans le code Python qui est effectué lorsqu'un internaute visite une page de votre site ?

9

Les formulaires Django

L'objectif de ce chapitre est d'étudier comment gérer les formulaires HTML avec Django. Nous nous poserons les questions suivantes :

- *Qu'allons-nous mettre comme URL dans l'attribut* `action` *?*
- *Comment allons-nous gérer dans Django la réception des données du formulaire ?*
- *Comment gérer des données erronées du formulaire ?*
- *Comment réagir lorsque toutes les données du formulaire sont correctes ?*

SOMMAIRE

- Comment réaliser les formulaires en Django ?
- Installation de nos premiers formulaires dans Trombinoscoop

Au chapitre précédent, nous avons vu ce qu'étaient les templates et les avons utilisés dans la page de login de Trombinoscoop, avec un résultat au final assez sympathique. Malheureusement, la page n'est pas vraiment opérationnelle. Si on clique sur le bouton *Se connecter*, rien ne se passe. Nous allons maintenant ajouter le code pour gérer ce formulaire et notamment préciser le contenu de l'attribut `action`.

Patron courant de gestion des formulaires

La gestion des formulaires suit souvent un même patron (pas le chef, mais le modèle ou le canevas). Lorsqu'on clique sur le bouton de soumission d'un formulaire, on rappelle la *même* page. Dans le cas de notre formulaire de login, on s'attend, lorsqu'on clique sur le bouton *Se connecter*, à arriver sur la page principale du site. Pourtant, nous n'allons pas mettre cette page dans l'attribut `action` mais l'URL même de la page de login.

Au niveau de Django, la même vue sera appelée, mais cette fois-ci recevant une liste de paramètres. On prévoit généralement dans la vue de tester si les valeurs introduites sont valides. Si elles ne le sont pas, on affiche un message d'erreur dans la même page. Si elles le sont, on redirige l'utilisateur vers une autre page. Ainsi, on s'assure de toujours arriver sur la vraie page de destination du formulaire et en présence de données validées.

Reprenons l'exemple de la page de login. Si les données d'authentification introduites sont incorrectes à la soumission du formulaire, on réaffiche la page de login avec un message d'erreur. Sinon, si tout est bien, on redirige l'utilisateur vers la page principale du site.

Cette redirection s'opère grâce aux réponses HTTP de redirection. Le protocole HTTP permet de renvoyer autre chose que du HTML en réponse à une requête. On peut par exemple renvoyer une erreur (comme la fameuse erreur 404 lorsqu'une page n'existe pas), ou une instruction de redirection qui va demander au navigateur à l'origine de la requête d'aller voir ailleurs. En Django, lorsqu'on désire envoyer une telle réponse, on écrit :

EXEMPLE 9.1 Réponse de redirection

```
return HttpResponseRedirect('/welcome')
```

L'instruction `HttpResponseRedirect` remplace le traditionnel `render_to_response`, qui permet de renvoyer du HTML. En recevant la réponse `redirect` de notre exemple, le navigateur sait qu'il doit se rendre sur la page web `welcome` et ne vous désobéira pas !

Au niveau de la vue responsable de la page contenant un formulaire, la logique s'en tiendra à cette même philosophie. Prenons l'exemple de notre page de login. La logique sera la suivante (écrite en pseudo-code) :

EXEMPLE 9.2 Traitement du formulaire de login (pseudo-code)

```
def login(request):
  si formulaire envoyé
    si paramètres pas ok
      on affiche la page de login avec une erreur
    sinon
      on redirige vers la page d'accueil
  sinon
    on affiche la page de login vierge de toute erreur
```

Il reste maintenant à savoir comment traduire ce pseudo-code en code effectif Python/Django. L'élément essentiel de cette démarche est le paramètre `request` de la méthode `login`.

L'objet request

L'objet `request`, premier paramètre de toutes nos méthodes `views` revêt une importance capitale. À vous de parfaitement le saisir ! Cet objet, de type `HttpRequest`, nous permet d'accéder à toutes les informations encapsulées dans la requête HTTP, dont l'ensemble des paramètres passés à la page web.

Pour accéder à ces paramètres, il suffit d'utiliser les objets `GET` et `POST` de l'objet `request`, qui se comportent comme des dictionnaires. Par exemple, si l'on désire connaître la valeur du paramètre `email` de notre formulaire de login, on écrira :

SYNTAXE EXEMPLE. Récupération d'un paramètre

```
email = request.GET['email']
```

C'est aussi simple que ça !

À l'aide des objets `GET` et `POST`, on peut également vérifier qu'un paramètre a bel et bien été transmis.

SYNTAXE EXEMPLE. Vérification qu'un paramètre a bien été transmis

```
if 'email' not in request.GET:
  # Paramètre pas transmis
```

Il est d'ailleurs de bonne pratique de toujours vérifier qu'un paramètre a bien été transmis avant d'essayer de le récupérer. Sinon, c'est le plantage assuré.

EN PRATIQUE Quelle différence entre GET et POST ?

Il existe deux méthodes, GET et POST, pour passer des paramètres à une page web.
Avec la méthode GET, les paramètres et leurs valeurs sont ajoutés à la fin de l'URL, séparés du nom de la page web par un point d'interrogation. Lorsqu'on clique sur le bouton de soumission d'un formulaire, le navigateur construit l'URL en ajoutant tous les paramètres. Dans le cas de notre formulaire de login, cela donnerait : http://127.0.0.1:8000/login?email=test&password=test.
Avec la méthode POST, la transmission des paramètres est un peu différente. Ceux-ci ne sont pas ajoutés à la fin de l'URL, mais inclus directement dans la requête HTTP. En conséquence, on ne les voit pas. Alors que l'effet d'une URL construite avec des paramètres est de les voir dans la barre d'adresse du navigateur, les requêtes HTTP du navigateur ne les montrent pas.
Cela explique pourquoi il existe en Django deux « dictionnaires » permettant de récupérer les paramètres : un pour les paramètres GET, un autre pour les paramètres POST. Il est d'ailleurs possible d'avoir à la fois des paramètres GET et POST.
Cette distinction explique pourquoi, dans les formulaires HTML, il existe un attribut `method` qui précise au navigateur selon quelle méthode envoyer les paramètres. Pour notre part, nous allons toujours utiliser la méthode GET, qui laisse voir ce qui est transmis. Bien entendu dans un formulaire de login réel, on n'utilisera jamais GET mais plutot POST.

Traitement du formulaire de login

Ajout du formulaire dans la vue

Nous allons tout d'abord modifier l'attribut `action` du formulaire dans `login.html`, puis ajouter le code de traitement du formulaire dans la vue.

EXEMPLE 9.3 Trombinoscoop. Modification de l'attribut action du formulaire dans login.html

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

EXEMPLE 9.4 Trombinoscoop. Traitement du formulaire dans views.py

```
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect (1)

def login(request):
  # Teste si formulaire a été envoyé
  if len(request.POST) > 0: (2)
    # Teste si les paramètres attendus ont été transmis
    if 'email' not in request.POST or 'password' not in request.POST: (3)
      error = "Veuillez entrer une adresse de courriel et un mot de passe."
      return render_to_response('login.html', {'error': error}) (4)
    else:
      email = request.POST['email'] (5)
```

```
    password = request.POST['password'] (5)
    # Teste si le mot de passe est le bon
    if password != 'sesame' or email != 'pierre@lxs.be': (6)
      error = "Adresse de courriel ou mot de passe erroné."
      return render_to_response('login.html', {'error': error}) (7)
    # Tout est bon, on va à la page d'accueil
    else:
      return HttpResponseRedirect('/welcome') (8)
# Le formulaire n'a pas été envoyé
else:
  return render_to_response ('login.html') (9)
```

Nous avons ajouté un import (1), afin de pouvoir utiliser la fonction renvoyant une redirection HTTP.

Pour savoir si le formulaire a été envoyé, on regarde dans l'objet `request` si la liste des paramètres reçus est vide ou pas (2). Si elle est vide, c'est qu'aucun paramètre n'a été transmis, donc la page n'est pas appelée suite à la soumission de son formulaire. A contrario, si la liste n'est pas vide, c'est que le formulaire a été soumis.

Dans le cas où le formulaire a été soumis, on vérifie que c'est bien les deux paramètres `email` et `password` qui ont été transmis. Ceux-ci doivent se trouver dans le dictionnaire `GET` de l'objet `request` (3). Si ce n'est pas le cas, on réaffiche la page login, mais en passant une erreur au template (4).

Ensuite, on récupère la valeur des deux paramètres (5) et on teste que le mot de passe et le login sont bien ceux attendus (6). Si ce n'est pas le cas, on réaffiche la page de login, avec une erreur (7). Si tout va bien, on peut rediriger l'internaute vers la page principale du site (8).

Il reste le cas où le formulaire n'a pas été soumis. Dans ce cas, on affiche la page de login vierge de tout message d'erreur (9).

Gestion du message d'erreur

La gestion du message d'erreur dans le template est triviale : s'il y en a un, on l'affiche, sinon on ne l'affiche pas. Il suffit d'ajouter le code suivant :

EXEMPLE 9.5 Trombinoscoop. Affichage du message d'erreur éventuel

```
<form action="login" method="POST">
  {% if error %}
  <p class="error">{{ error }}</p>
  {% endif %}
Suite du code du formulaire…
</form>
```

Présentation du message d'erreur

Nous avons spécifié une classe au paragraphe contenant le message d'erreur. Cela sert à personnaliser son apparence à l'aide de directives CSS. Vous pouvez par exemple mettre le message en rouge (cela fait plus grave).

EXEMPLE 9.6 Trombinoscoop. Personnalisation de l'aspect du message d'erreur dans style.css.

```
form p.error {
  margin-left: 160px;
  color: #d23d24;
}
```

Au final, lorsqu'on soumet le formulaire avec un mauvais identifiant ou un mauvais mot de passe, le résultat est le suivant.

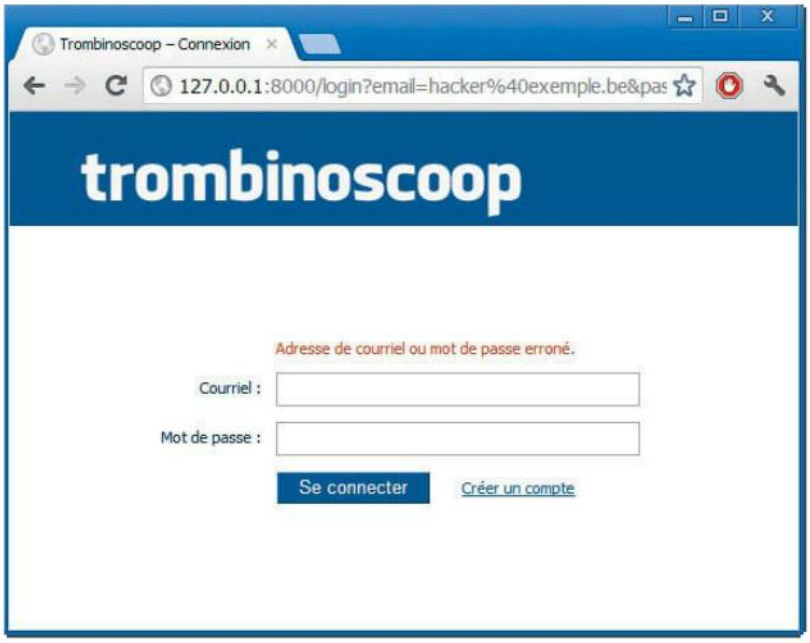


Figure 9–1
Rendu de la page de login avec erreur

Voilà qui n'est déjà pas si mal ! Cependant, les sections suivantes vont vous montrer qu'on peut faire ça plus proprement.

La bibliothèque forms de Django

Le but de Django est de vous simplifier la vie et d'automatiser les tâches courantes afin que vous n'ayez pas à réinventer la roue.

Or, lorsqu'il s'agit de produire et gérer des formulaires, on est souvent amené à réécrire des lignes de code identiques d'une fois sur l'autre. Django simplifie tout cela à l'aide de sa bibliothèque `forms`, qui permet entre autres de :

- ne plus devoir écrire soi-même le HTML d'un formulaire. Tous les `label` et `input` sont ajoutés par Django.
- avoir une validation réalisée par Django, sans avoir à tout vérifier soi-même. Django est, par exemple, capable de valider si une donnée encodée par un utilisateur est bien une adresse courriel.
- réafficher le formulaire complété dans le cas où des erreurs auraient été détectées. Pour l'instant, dans notre version de la page de login, lorsque les données sont erronées, la page est réaffichée, mais toutes les données encodées par l'utilisateur sont effacées. Quand il n'y a que deux champs, ce n'est pas grave, mais dans le cas d'un formulaire de plusieurs dizaines de champs, cela peut passablement énerver le visiteur de votre site.
- convertir dans des types Python les données soumises à partir d'un formulaire. Tous les paramètres qui accompagnent une requête HTTP sont textuels ; en conséquence, si un des champs d'un formulaire est numérique à la base (par exemple l'âge du capitaine), pour être utilisable en Python, il va falloir convertir la donnée reçue en un type numérique. La bibliothèque `forms` de Django prend cela en charge.

Création d'un formulaire avec la bibliothèque forms

Si on désire utiliser la bibliothèque `forms`, la première étape consiste à définir le formulaire et ses champs. Pour ce faire, on crée une classe héritant de la classe `forms.Form`. Pour notre formulaire de login, cela donne le code suivant, que nous plaçons dans un nouveau fichier nommé `forms.py`, situé au même niveau que `views.py` :

EXEMPLE 9.7 Trombinoscoop. Création d'un formulaire de login dans forms.py

```
from django import forms (1)

class LoginForm(forms.Form): (2)
  email = forms.EmailField(label='Courriel')
  password = forms.CharField(label='Mot de passe',
                             widget = forms.PasswordInput) (3)
```

Tout d'abord, on importe la bibliothèque `forms` (1). Ensuite, on définit notre formulaire de login en créant une classe `LoginForm` qui hérite de la classe `Form` (2). Il reste à définir les deux champs du formulaire : le courriel et le mot de passe. On définit simplement deux attributs qui sont de type champ (`Field`) : un de type `EmailField`,

l'autre de type `CharField`. Pour les deux, nous avons également spécifié le label du champ, qui servira pour l'affichage.

SYNTAXE Pourquoi CharField et pas PasswordField ?

Car ce dernier n'existe tout simplement pas… Pour définir un champ de type « mot de passe », il faut utiliser un `CharField` et changer son rendu à l'aide de `widget = forms.PasswordInput` (3).

EN PRATIQUE D'autres types de champs...

Il existe de nombreux autres types de champs prévus par Django : `BooleanField`, `ChoiceField`, `ComboField`, `DateField`, `IntegerField`, etc.

Intégration du formulaire dans la vue et le template

La classe formulaire définie, nous pouvons l'utiliser dans nos templates, après en avoir créé une instance dans la vue. Nous allons donc d'abord modifier notre vue comme suit (n'oubliez pas le nouvel import (1)) :

EXEMPLE 9.8 Trombinoscoop. Utilisation du LoginForm dans la vue

```
from forms import LoginForm (1)

def login(request):
  form = LoginForm()
  return render_to_response('login.html', {'form': form})
```

Nous avons pour l'instant laissé de côté tout ce qui concernait la validation du formulaire. Le but ici est d'observer la création automatique du formulaire dans le template. Remplaçons le code du template par le suivant :

EXEMPLE 9.9 Trombinoscoop. Utilisation du LoginForm dans le template

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

{% block title %}Connexion{% endblock %}

{% block bodyId %}loginPage{% endblock %}

{% block content %}
<form action="login" method="post">
  {{ form.as_p }} (1)
  <p>
```

```
    <input type="submit" value="Se connecter" />
    <a href="????">Créer un compte</a>
  </p>
</form>
{% endblock %}
```

Pour « imprimer » le formulaire, on utilise la méthode `as_p` (1) de l'objet `form` passé en paramètre depuis la vue. Cette méthode a pour effet de mettre chaque champ dans un paragraphe, exactement comme nous l'avions fait manuellement précédemment. Il existe également la méthode `as_table` qui produit le formulaire sous forme de tableau (en plaçant chaque champ dans une ligne du tableau) et `as_ul` qui produit des éléments de liste.

Remarquez que la fonction `as_p` ne fait que produire les champs du formulaire, charge à nous de définir le formulaire en lui-même (la balise `form`) et d'ajouter le bouton de soumission.

Si on teste ce code, voilà le rendu qu'on obtient dans le navigateur :

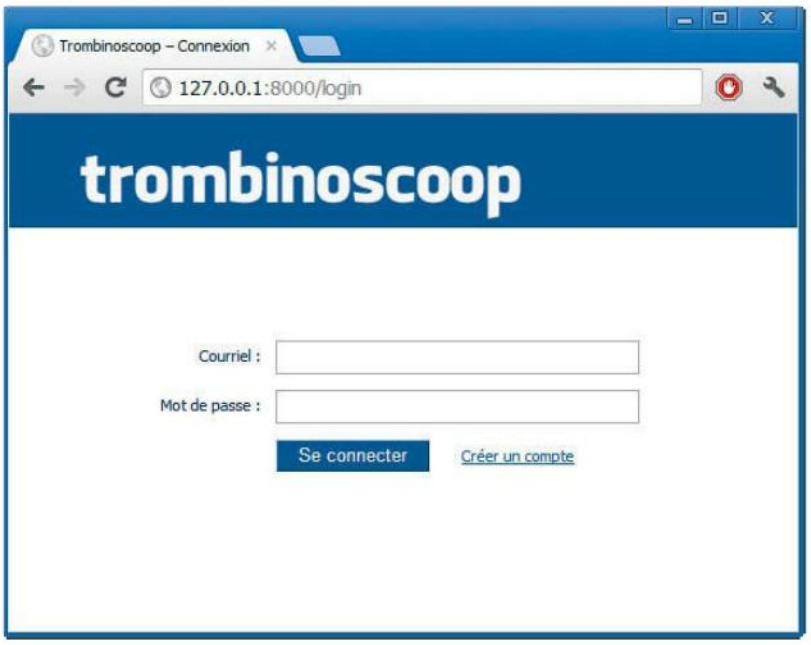


Figure 9–2
Rendu de la page de login en utilisant la bibliothèque « forms »

Le résultat est identique au précédent ! Et pour cause, Django a écrit exactement le même HTML que nous (à quelques détails près) :

EXEMPLE 9.10 Trombinoscoop. HTML créé pour le formulaire

```
<form action="login" method="post">
  <p>
    <label for="id_email">Courriel :</label> (1)
    <input type="text" name="email" id="id_email" /> (2)
  </p>
  <p>
    <label for="id_password">Mot de passe :</label> (1)
    <input type="password" name="password" id="id_password" /> (2)
  </p>
  <p>
    <input type="submit" value="Se connecter" />
    <a href="????">Créer un compte</a>
  </p>
</form>
```

Les différences sont les suivantes :

- Les `id` choisis pour les champs ne sont pas les mêmes. Peu importe. D'une part, nous ne les utilisons pas dans les CSS ; d'autre part, Django reste cohérent en utilisant la même valeur d'`id` pour l'attribut `for` de l'élément `label` (1) et l'attribut `id` de l'élément `input` (2).
- Le type du champ de courriel n'est pas `email` mais `text`. En fait, le type `email` est assez récent : il est apparu avec HTML 5. Or, Django n'ose pas produire du HTML 5, par peur d'être incompatible avec les navigateurs les plus anciens. Il produit donc du HTML 4 – sauf à utiliser des bibliothèques complémentaires comme floppyforms.
- Il n'y a pas d'attribut `size` pour donner une longueur au champ.

Toutes ces différences peuvent facilement être corrigées. Django permet de personnaliser très finement la manière dont sont rendus les formulaires : on peut changer la manière dont les `id` sont produits ou encore demander qu'un champ de courriel ait un `input` de type `email`. Vous n'aurez aucun mal à trouver des explications complémentaires sur Internet.

Validation du formulaire

Voyons maintenant comment valider de notre formulaire, afin de retrouver le comportement que nous avions auparavant.

Les formulaires Django possèdent une méthode intitulée `is_valid`. Elle autorise des validations de base : vérifier que les champs obligatoires sont fournis, vérifier que le courriel entré est bien une adresse valide, etc. La validation du mot de passe est en revanche un peu plus complexe ; il faudra la coder nous-mêmes. Heureusement, Django permet d'ajouter des validations propres.

Avant de s'atteler à la validation du mot de passe, voyons déjà à quoi va ressembler notre vue si on veut qu'elle reproduise le comportement qu'on avait précédemment, à savoir tester la validité des champs, et, le cas échéant, rediriger l'utilisateur sur la page principale du site ou réafficher la page de login avec des erreurs.

Le code de la vue sera le suivant, beaucoup plus simple et facile à comprendre.

EXEMPLE 9.11 Trombinoscoop. Traitement du formulaire dans views.py

```
def login(request):
  # Teste si le formulaire a été envoyé
  if request.method=="POST")
    form = LoginForm(request.POST) 2
    if form.is_valid(): 3
      return HttpResponseRedirect('/welcome')
  else:
    form = LoginForm() 1
    return render_to_response ('login.html', {'form': form})
```

Le début est le même : on teste si le formulaire a été soumis. Si ce n'est pas le cas, on se crée un objet de type `LoginForm` vierge qu'on passe au template 1. Si, au contraire, le formulaire a été soumis, on crée un objet de type `LoginForm` en lui passant cette fois-ci les données `GET` de la requête HTTP. On peut ensuite appliquer la méthode `is_valid` 3 qui va vérifier la validité de tous les champs du formulaire : si le formulaire est valide, on redirige l'utilisateur, sinon on réaffiche la même page.

Voyons maintenant ce que cela donne dans un navigateur, après avoir entré « pasbon » comme adresse de courriel et avoir omis de préciser un mot de passe.

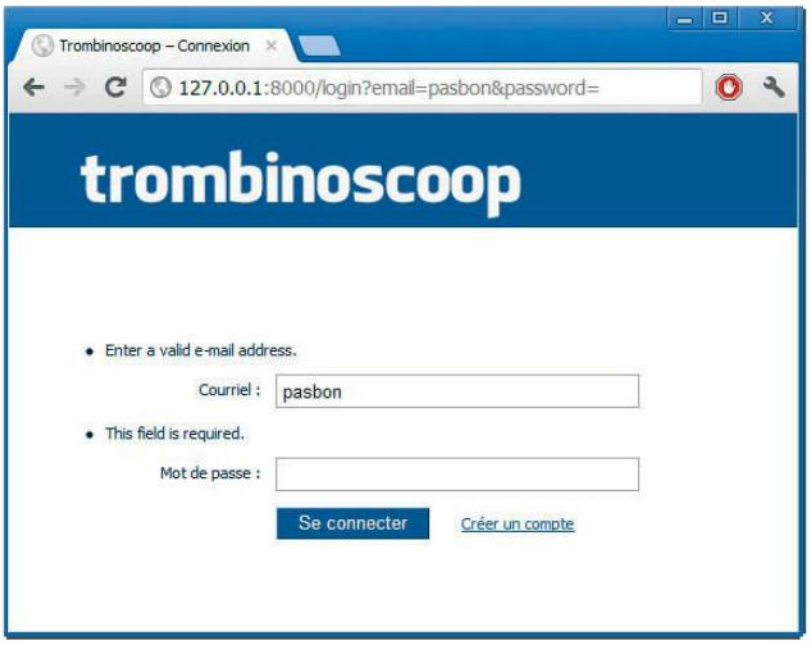


Figure 9–3
Gestion des erreurs par la bibliothèque forms

On le voit, la gestion des erreurs est un peu différente de ce que nous avions dans notre version manuelle du formulaire. En réalité, il faut l'avouer, elle est plus complète : pour chaque champ erroné, un message d'erreur s'affiche (voire plusieurs, le cas échéant). En outre, l'adresse de courriel encodée par l'utilisateur a été reprise, alors que notre formulaire manuel remettait tous les champs à zéro.

EN PRATIQUE Gestion des erreurs dans notre formulaire manuel

Dans notre formulaire manuel, nous ne pouvions afficher qu'une seule erreur, en haut du formulaire. Et si deux champs étaient erronés, nous affichions d'abord la première erreur, puis une fois celle-ci corrigée et le formulaire resoumis, nous affichions la deuxième.

Cependant, les messages d'erreur sont en anglais. Rien de surprenant : la langue de Shakespeare est l'esperanto de la programmation ! Nous allons les laisser en l'état, mais sachez qu'il est possible de les traduire en plusieurs langues.

Présentation des messages d'erreur

Il serait bien de modifier nos CSS pour mieux positionner les messages d'erreur et les afficher dans une autre couleur. Si on analyse le code HTML produit par Django, on constate que les messages d'erreur sont en fait des listes `ul` de classe `errorList` (1).

EXEMPLE 9.12 Trombinoscoop. Code HTML produit par Django pour les messages d'erreur

```
<ul class="errorlist"> (1)
  <li>Enter a valid e-mail address.</li>
</ul>
<p>
  <label for="id_email">Courriel :</label>
  <input type="text" name="email" value="pasbon" id="id_email" />
</p>
```

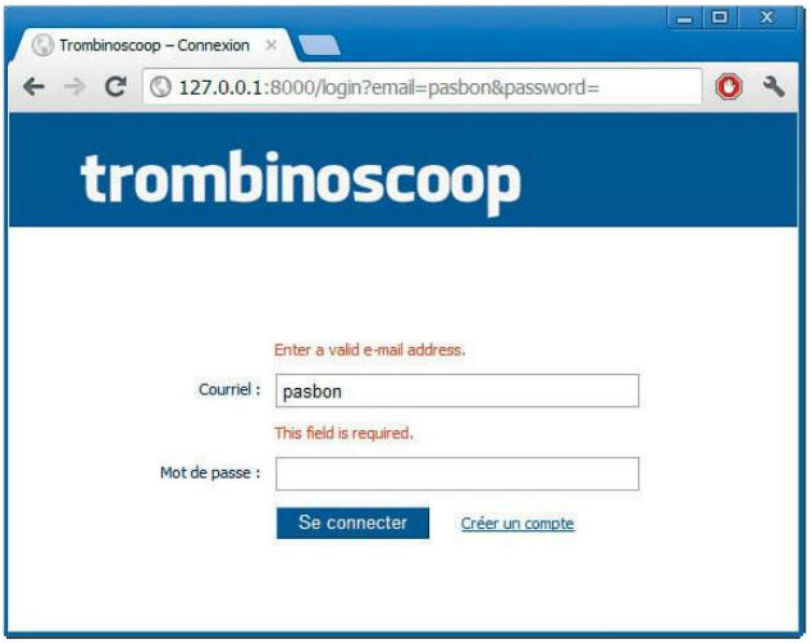
Donc, dans nos CSS, nous pouvons écrire le code suivant, en remplacement des instructions CSS que nous avions destinées au sélecteur `form p.error` :

EXEMPLE 9.13 Trombinoscoop. Présentation des messages d'erreur

```
ul.errorlist {
  margin-left: 160px;
  padding: 0;
  list-style-type: none;
  color: #d23d24;
}
```

Le résultat obtenu est maintenant le suivant.

Figure 9–4
Nouvel aspect des messages d'erreur



Validation de l'adresse de courriel et du mot de passe

Pour l'instant, si on entre un courriel et un mot de passe tous deux choisis au hasard, le formulaire nous laisse rentrer sur le site. Ce n'est pas très sécurisé, tout cela !

Nous allons ajouter une validation qui concerne à la fois le champ de courriel et le champ de mot de passe ; cette validation interviendra donc au niveau du formulaire, et non au niveau d'un des champs. En d'autres mots, c'est dans la classe `LoginForm` que tout va se passer, plus précisément dans la méthode `clean` de la classe, qu'il faut ajouter ce genre de validation. Cette méthode sert, à la base, à retourner les champs « nettoyés », c'est-à-dire les champs valides et convertis dans un format de donnée Python.

Voici le code de la classe `LoginForm` dans son intégralité.

EXEMPLE 9.14 Trombinoscoop. Ajout de la validation du courriel et du mot de passe

```
class LoginForm(forms.Form):
  email = forms.EmailField(label='Courriel :')
  password = forms.CharField(label='Mot de passe :',
                             widget = forms.PasswordInput)
  def clean(self): (1)
    cleaned_data = super (LoginForm, self).clean() (2)
    email = cleaned_data.get("email") (3)
    password = cleaned_data.get("password") (3)
```

```
        # Vérifie que les deux champs sont valides
        if email and password: (4)
          if password != 'sesame' or email != 'pierre@lxs.be': (5)
            raise forms.ValidationError("Adresse de courriel
                                        ou mot de passe erroné.") (6)

        return cleaned_data (7)
```

Nous redéfinissons donc la méthode `clean` (1). Tout d'abord, on appelle cette même fonction au niveau de la classe parente (c'est le `super` qui s'en charge (2)), car cette dernière effectue des traitements qu'on aimerait ne pas devoir ré-implémenter. Ensuite, on récupère la valeur « nettoyée » des champs (3). Si les champs n'étaient pas valides, les lignes `cleaned_data.get` ne retourneraient rien.

On vérifie ensuite que les deux champs sont valides, en s'assurant que les lignes `cleaned_data.get` ont retourné quelque chose (4). En effet, il ne sert à rien de vérifier que le courriel ou le mot de passe sont les bons, si les données ne satisfont déjà pas les contrôles basiques.

Ensuite, on peut vérifier que courriel et mot de passe sont bien ceux attendus (5). Si ce n'est pas le cas, on retourne une erreur à l'aide de la méthode Python `raise` (6). C'est ainsi que Django exige que l'on procède ; cela lui permet d'ajouter notre erreur à la liste des erreurs du formulaire.

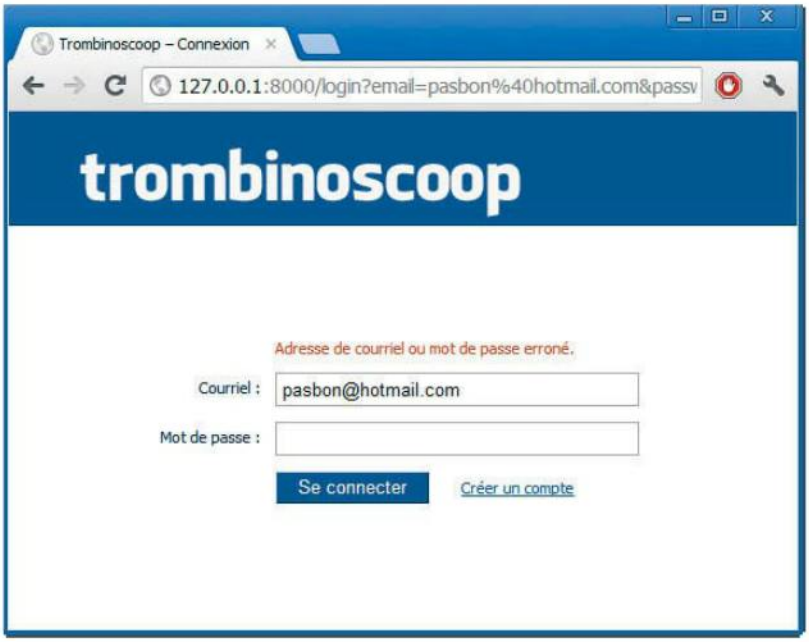


Figure 9–5
Ajout de la vérification du courriel et du mot de passe

Enfin, pour terminer, on retourne les données nettoyées, car, ne l'oublions pas, cela reste après tout le but premier de cette méthode (7).

Ce code une fois ajouté, si on entre un courriel et un mot de passe qui, pris séparément, sont valides, mais ne correspondent pas à ce qu'on attend, on obtient le résultat présenté à la figure 9-5.

C'est exactement ce que l'on recherchait ! Et on peut constater que Django a signalé l'erreur au tout début du formulaire, avant le premier champ.

Faut-il se contenter d'un seul visiteur autorisé ?

Et voilà, notre page de login est terminée ! Enfin… presque. Pour l'instant, un seul utilisateur peut se connecter au site : celui dont le courriel et le mot de passe sont codés en dur dans le code Python. Pourtant, nous aimerions évidemment avoir plus d'un utilisateur pour notre site.

Il va donc falloir stocker nos utilisateurs dans une base de données et la procédure d'authentification devra vérifier dans cette dernière si le courriel et le mot de passe entrés sont bien ceux attendus.

C'est l'objectif du chapitre suivant : créer notre base de données et apprendre comment y accéder.

Ai-je bien compris ?

- Comment sont habituellement traités les formulaires dans une vue ? Pourquoi y a-t-il une instruction de redirection ?
- À quoi sert l'objet `request` ?
- Quels sont les avantages apportés par la bibliothèque Django `forms` ?

10

Les modèles Django

L'objectif de ce chapitre est d'étudier comment mettre en place une base de données à l'aide des modèles Django et comment interagir avec elle. Plus précisément, nous allons apprendre à :

- *décrire les données à enregistrer dans la base de données ;*
- *créer la base de données ;*
- *insérer des données dans la base de données ;*
- *accéder aux données pour ensuite les utiliser dans notre code.*

SOMMAIRE

- Réalisation de la base de données à la manière de Django
- Introduction aux modèles Django, version objet de la base de données sous-jacente
- Suite du projet Trombinoscoop

Au chapitre précédent, nous avons appris à récupérer et utiliser les données d'un formulaire, essentiellement pour la mise en place de notre page d'authentification.

Cette page souffre d'un grand défaut : la vérification de l'adresse de courriel et du mot de passe est entièrement statique, ces deux données d'authentification étant inscrites directement dans le code Python. Il est donc très difficile d'ajouter de nouveaux utilisateurs. En réalité, un élément essentiel manque à notre application web : une base de données qui permettrait d'enregistrer plusieurs utilisateurs avec toutes leurs caractéristiques, et en particulier leur adresse courriel et leur mot de passe.

Les modèles Django

Django, comme tout bon framework, offre une couche d'abstraction presque totale pour accéder aux bases de données et les gérer. La vision orientée objet du monde Python éclipse totalement l'aspect relationnel des données. En d'autres mots, on ne doit pas soi-même créer et définir la base de données, on ne doit pas écrire la moindre ligne de SQL pour y accéder. Tout se fait de manière invisible. C'est à peine si l'on soupçonne l'existence d'une base de données pour stocker nos données !

Cette abstraction est réalisée à l'aide des *modèles*, qui définissent une fois pour toutes les données que l'on désire stocker, en respectant entièrement l'approche orientée objet ébauchée au chapitre 3. Toutes les données que l'on veut sauvegarder sont modélisées sous forme de classes qui possèdent des attributs et des méthodes. À partir de cette définition de classes, Django sera capable de :

- créer automatiquement une base de données contenant les tables et les champs nécessaires pour stocker les objets ;
- ajouter des méthodes permettant de créer, modifier et supprimer les objets depuis la base de données.

Afin de mieux comprendre ce principe, tentons de mettre en pratique le modèle de données que nous avons défini au chapitre 5.

Création d'un premier modèle

Pour rappel, le diagramme de classes du modèle à implémenter était le suivant :

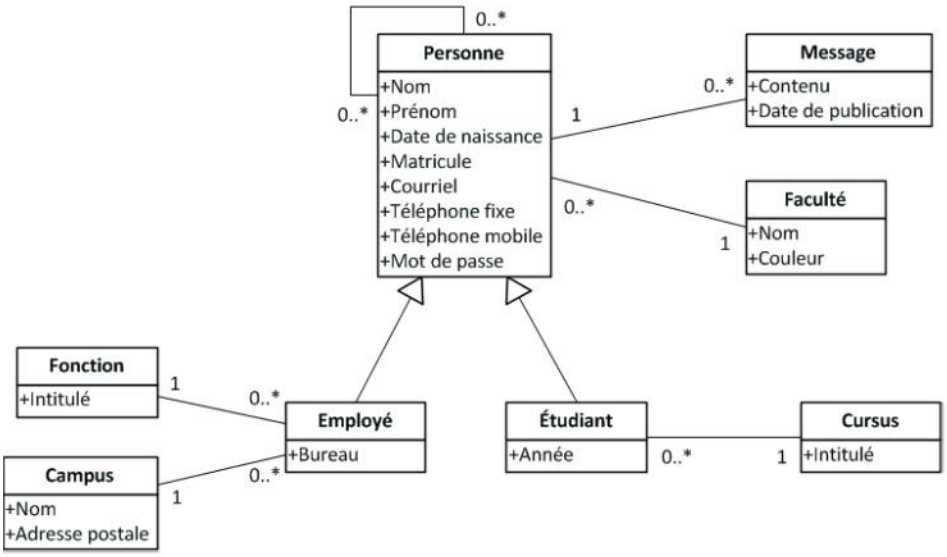


Figure 10–1
Modèle de donnée de Trombinoscoop

Le modèle Personne

Démarrons progressivement et voyons d'abord comment s'occuper de l'objet `Personne` à l'aide des modèles, sans se soucier des autres objets pour l'instant.

En Django, un *modèle* est une classe qui hérite de la classe `Model` fournie par Django. On a généralement une classe de type `Model` par objet que l'on désire sauvegarder (ou par table de la base de données). Dans notre cas, nous aurons donc un modèle dénommé `Personne`. Ensuite, définir le modèle consiste essentiellement à lister les champs qui le composent, avec leur type.

On regroupe généralement les modèles dans un fichier nommé `models.py`. Créons donc ce fichier dans Eclipse, au même niveau que `views.py` et ajoutons le code suivant :

EXEMPLE 10.1 Trombinoscoop. Définition d'un premier modèle pour l'objet Personne

```
from django.db import models (1)

class Personne(models.Model): (2)
  matricule = models.CharField(max_length=10)
  nom = models.CharField(max_length=30)
  prenom = models.CharField(max_length=30)
  date_de_naissance = models.DateField()
  courriel = models.EmailField()
  tel_fixe = models.CharField(max_length=20)
  tel_mobile = models.CharField(max_length=20)
  mot_de_passe = models.CharField(max_length=32)
```

C'est aussi simple que cela ! On importe la bibliothèque `models` (1), on crée une classe `Personne` héritant de `models.Model` (2) et, pour terminer, on définit tous nos champs en leur donnant un type.

EN PRATIQUE Type des numéros de téléphone

Notez que nous utilisons des types textuels et non des types numériques pour les numéros de téléphone, car ces derniers peuvent contenir des caractères qui ne sont pas numériques : dièses, barres obliques, espaces, etc. (par exemple +32 486/95 67 89).

Avec ces informations, Django est capable de créer la base de données adéquate.

Configuration

Auparavant, il faut toutefois déclarer l'application Trombinoscoop dans `settings.py`. C'est obligatoire dès que l'on utilise des modèles. Il faut donc ajouter la ligne suivante :

CONFIGURATION. Déclaration de Trombinoscoop dans settings.py

```
INSTALLED_APPS = (
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.sites',
'django.contrib.messages',
'django.contrib.staticfiles',
'Trombinoscoop',
# Notez qu'il serait préférable de ne pas mettre de majuscule a un nom d'application
)
```

Vérifions également les paramètres relatifs à la base de données. Ceux-ci se trouvent généralement par défaut dans `settings.py` (vous devriez changer le `Name`) :

CONFIGURATION. Paramètres de la base de données

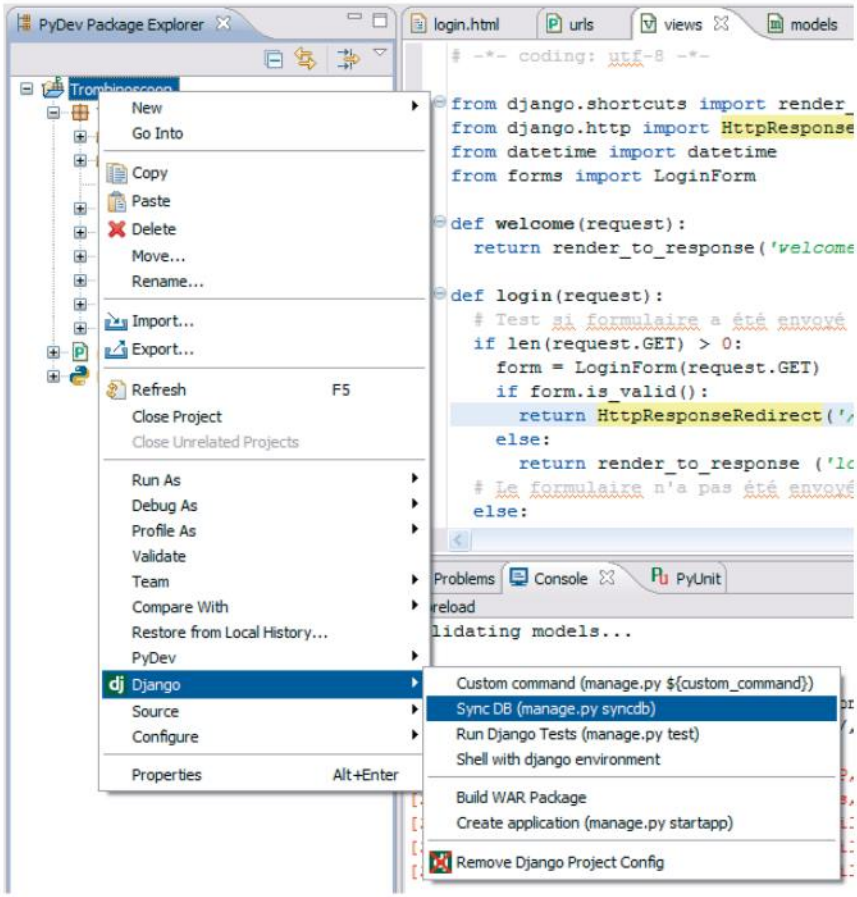
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(PROJECT_ROOT, 'sqlite.db'),
        'USER': '',
        'PASSWORD': '',
        'HOST': '',
        'PORT': '',
    }
}
```

Comme vous pouvez le constater, le moteur de bases de données utilisé par Django est SQLite3, cela tombe bien. Quant à l'emplacement, la partie que nous avons modifiée demande à Django de créer la base de données dans le dossier du projet.

Création de la base de données et du compte administrateur (superutilisateur)

Lançons maintenant la création de la base de données. Cela se fait via la commande `Sync DB`, propre à Django. Celle-ci est disponible dans les menus Eclipse, comme le montre la figure suivante (cliquez-droit sur le projet *Trombinoscoop*, ensuite sur le menu *Django* et enfin sur *Sync DB (manage.py syncdb)*).

Figure 10–2
Création de la base de données



Au lancement de cette commande, Django part à la recherche de toutes les classes de type `Model` et crée les tables correspondantes. À la console, on constate que Django est bien en train de créer la base de données :

EXEMPLE 10.2 Résultat. Création de la base de données par Django

```
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 django_content_type
Creating table django_session
Creating table django_site
Creating table Trombinoscoop_personne
```

Django vous demande ensuite si vous souhaitez créer un *superuser* dans le système. Faites-lui confiance et répondez : cliquez dans la console et tapez au clavier le mot `yes` avant d'appuyer sur la touche *Entrée*. Il faut ensuite introduire un nom d'utilisateur, une adresse de courriel et un mot de passe (deux fois). Cet utilisateur pourra introduire des données de test dans les tables. Notez bien ces informations, elles vous seront utiles par la suite.

Si tout se passe bien, la console indique ceci :

EXEMPLE 10.3 Résultat. Création de l'administrateur (superuser) de la base de données

```
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): yes
Username (leave blank to use 'pierre'): pierre
E-mail address: pierre@lxs.be
Password: azerty
Password (again): azerty
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
Finished "C:\Documents and Settings\Pierre\Mes documents\Mes
projets\Trombinoscoop\manage.py syncdb" execution.
```

Et voilà, la base de données est créée. En atteste le nouveau fichier `sqlite.db` qui se trouve dans notre projet. Empressons-nous de l'ouvrir à l'aide de l'outil SQLite Database Browser (que nous avons présenté au chapitre 6).

EN PRATIQUE Si vous ne voyez pas sqlite.db dans Eclipse...

... un petit rafraîchissement s'impose : appuyez sur *F5* et il apparaîtra.

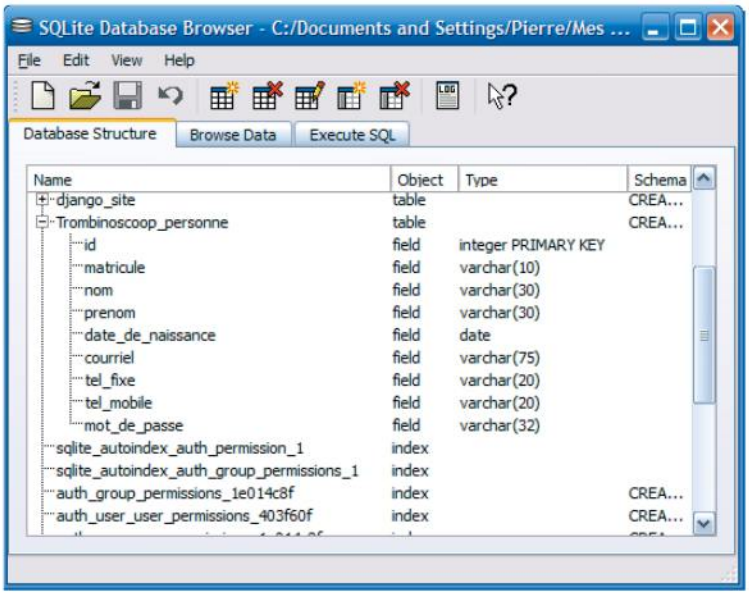


Figure 10–3
Ouverture de la base de données avec SQLite Database Browser

De nombreuses tables ont été créées alors qu'on s'attendait à en trouver une seule. C'est normal, Django ajoute à la base de données des tables de travail dont il aura besoin pour d'autres fonctionnalités du framework (nous en reparlerons dans les chapitres suivants). Laissons donc de côté ces tables et concentrons-nous sur `Trombinoscoop_personne` en particulier. Tous les champs ont été créés conformément à nos desiderata. Un champ supplémentaire a cependant été ajouté : `id`. C'est la clé primaire identifiant de manière unique chaque enregistrement. Comme nous n'avons pas dit que le champ `matricule` pouvait faire office de clé primaire, Django a pris l'initiative d'en créer une. Finalement, pourquoi ne pas la laisser ?

Création des autres modèles et de leurs liens

Le modèle Message : relation 1-n

Créer les autres modèles de notre projet implique de définir les liens les unissant. En Django, cela s'exprime à l'aide de champs un peu particuliers :

- Pour les liens « 1-n », on utilise le champ `models.ForeignKey` (clé étrangère… témoignage du modèle relationnel sous-jacent).
- Pour les liens « n-n », on utilise le champ `models.ManyToManyField`.
- Pour les liens « 1-1 », on utilise le champ `models.OneToOneField`.

EN PRATIQUE Liens entre les tables

- Les liens « 1-n ». Exemple : une personne peut publier plusieurs messages, mais un message est rédigé par une seule personne.
- Les liens « n-n ». Ils sont interdits dans le modèle relationnel : il faut le casser en deux liens « 1-n ». Exemple : une faculté peut avoir des bâtiments sur plusieurs campus, tandis que chaque campus peut accueillir plusieurs facultés. La relation « ami » est un cas particulier de lien « n-n » à l'intérieur d'une même table.
- Les liens « 1-1 ». Nous n'en avons pas dans notre modèle de données, mais ce serait le cas si on avait décidé de « couper » un objet en deux objets distincts, par exemple pour séparer des données confidentielles (les données d'authentification) de données publiques (le profil).

Commençons par ajouter l'objet `Message` :

EXEMPLE 10.4 Trombinoscoop. Ajout du modèle Message

```
from django.db import models

class Personne(models.Model):
  nom = models.CharField(max_length=30)
  prenom = models.CharField(max_length=30)
  date_de_naissance = models.DateField()
  matricule = models.CharField(max_length=10)
  courriel = models.EmailField()
  tel_fixe = models.CharField(max_length=20)
  tel_mobile = models.CharField(max_length=20)
  mot_de_passe = models.CharField(max_length=32)

class Message(models.Model):
  auteur = models.ForeignKey(Personne) (1)
  contenu = models.TextField()
  date_de_publication = models.DateField()
```

Comme un message correspond à une et une seule personne, nous avons ajouté un champ `ForeignKey` que nous avons nommé `auteur` et qui pointe vers la classe `Personne` (1). Du côté de la classe `Personne`, il n'y a rien à ajouter. La relation devient implicite.

La relation « ami » : relation n-n

Ajoutons maintenant la relation « ami » qui lie des personnes entre elles et qui est de type « n-n ». Pour exprimer cette relation, il suffit d'ajouter un champ `ManyToManyField` qui pointe vers la classe elle-même (self) (2). Ce champ, nous le nommons `amis` (au pluriel). Notez que nous avons ajouté le paramètre `null=True`

afin de rendre ce champ optionnel. On n'est pas obligé d'avoir au moins un ami : les asociaux ne sont pas interdits sur notre site.

EXEMPLE 10.5 Trombinoscoop. Ajout de la relation « ami »

```
from django.db import models

class Personne(models.Model):
  nom = models.CharField(max_length=30)
  prenom = models.CharField(max_length=30)
  date_de_naissance = models.DateField()
  matricule = models.CharField(max_length=10)
  courriel = models.EmailField()
  tel_fixe = models.CharField(max_length=20)
  tel_mobile = models.CharField(max_length=20)
  mot_de_passe = models.CharField(max_length=32)
  # Dans un cas réel, on ne stockera pas le mot de passe en clair.
  amis = models.ManyToManyField("self", null=True) (2)
```

Les modèles simples Faculté, Campus, Fonction et Cursus

Définissons ensuite les objets `Faculte`, `Campus`, `Fonction` et `Cursus`. Leur définition est élémentaire et ne nécessite aucune explication :

EXEMPLE 10.6 Trombinoscoop. Ajout des modèles Faculte, Campus, Fonction et Cursus

```
class Faculte(models.Model):
  nom = models.CharField(max_length=30)
  couleur = models.CharField(max_length=6)

class Campus(models.Model):
  nom = models.CharField(max_length=30)
  adresse_postale = models.CharField(max_length=60)

class Fonction(models.Model):
  intitule = models.CharField(max_length=30)

class Cursus(models.Model):
  intitule = models.CharField(max_length=30)
```

Les modèles Employé et Étudiant : héritage

Il reste à définir les objets `Employe` et `Etudiant`. C'est très simple, il suffit de créer deux nouveaux modèles qui héritent de la classe `Personne`, exactement comme dans notre modèle conceptuel de données.

Django sait convertir cette hiérarchie de classes en tables dans la base de données, en ajoutant les liens nécessaires. En fait, il traduit le mécanisme d'héritage en une liaison « 1-1 » au sens du relationnel : une clé étrangère reliée à la table `Personne` a été ajoutée dans les tables `Employé` et `Étudiant`. Quand bien même le mécanisme d'héritage existe entre les classes dans le « monde objet » et non entre les tables dans le « monde relationnel », Django réussit à gérer la base de données relationnelle pour qu'elle tienne compte de cet héritage. Chapeau bas ! Voici le code de nos deux modèles :

EXEMPLE 10.7 Trombinoscoop. Modèles Employé et Étudiant

```
class Employe(Personne):
  bureau = models.CharField(max_length=30)
  campus = models.ForeignKey(Campus)
  fonction = models.ForeignKey(Fonction)

class Etudiant(Personne):
  cursus = models.ForeignKey(Cursus)
  annee = models.IntegerField()
```

Le lien entre Faculté et Personne : relation 1-n

Comme tout dernier détail, il nous reste à définir la relation « 1-n » qui lie `Faculté` et `Personne`.

EXEMPLE 10.8 Trombinoscoop. Ajout de la Faculté à Personne

```
from django.db import models

class Personne(models.Model):
  nom = models.CharField(max_length=30)
  prenom = models.CharField(max_length=30)
  date_de_naissance = models.DateField()
  matricule = models.CharField(max_length=10)
  courriel = models.EmailField()
  tel_fixe = models.CharField(max_length=20)
  tel_mobile = models.CharField(max_length=20)
  mot_de_passe = models.CharField(max_length=32)
  amis = models.ManyToManyField("self")
  faculte = models.ForeignKey(Faculte)
```

EN PRATIQUE L'ordre de définition des classes est très important

Définissez les classes dans le bon ordre. Ici, comme Python s'exécute en séquence, il faut que `Faculté` soit écrite avant `Personne`. Autrement Python ne saura pas ce qu'on entend par `Faculte`, n'ayant pas encore trouvé cette classe sur son chemin.

Regénération de la base de données

Il ne reste plus qu'à refaire un `Sync DB` pour regénérer notre base de données avec tous ces nouveaux modèles.

> **EN PRATIQUE La commande Sync DB ne supprime pas les bases de données existantes**
>
> Lorsqu'on ré-exécute un Sync DB, la base de données existante n'est pas écrasée et recréée à zéro. Elle est simplement mise à jour par l'ajout des tables et des champs manquants. Si entre-temps, un champ ou une classe a été supprimé(e) du modèle, Sync DB ne les supprime pas de la base de données.
> Pour avoir une base de données lavée de tout soupçon, il est donc toujours conseillé de la détruire (en supprimant le fichier `sqlite.db`) préalablement à l'exécution de la commande Sync DB. Attention, dans ce cas on perd toutes les données déjà saisies dans la base !

Utilisation des modèles

Maintenant que notre base de données est créée et que nos modèles sont définis, comment les utiliser ? Comment créer un nouvel enregistrement ? Comment récupérer un enregistrement existant ? Comment en supprimer un ?

Création et modification d'un enregistrement

Le code suivant illustre la création et la modification d'un enregistrement. Nous utiliserons plus tard ce genre de code dans notre projet.

SYNTAXE EXEMPLE. Création et modification d'un enregistrement

```
xavier = Etudiant(nom='Devos', prenom='Xavier', etc.)
# Ici, l'objet a juste été créé en mémoire
xavier.save()
# Ici, l'objet est sauvé dans la base de données
xavier.matricule = '0123456789'
# Ici, l'objet a juste été modifié en mémoire
xavier.save()
# Ici, l'objet est modifié dans la base de données
```

Le code est assez explicite. Remarquez cependant que, lorsqu'on crée l'objet `Etudiant`, il faut fournir en paramètre tous les champs qui n'ont pas été déclarés optionnels (à l'aide de `null=True`).

Récupération de plusieurs enregistrements

Le code suivant illustre la récupération de plusieurs enregistrements.

SYNTAXE EXEMPLE. Récupération de plusieurs enregistrements

```
# Récupération de tous les étudiants
etudiants = Etudiant.objects.all()

# Récupération de tous les étudiants de première année
etudiants = Etudiant.objects.filter(annee=1)

# Récupération de tous les étudiants dont le nom commence par A
etudiants = Etudiant.objects.filter(nom__startswith='A')
```

À nouveau, les exemples sont assez explicites. La documentation Django vous apprendra quels filtres exactement on peut utiliser.

Notez qu'en retour de toutes ces fonctions, on reçoit une liste d'objets de type `Etudiant`.

Tri des données

On peut également trier les enregistrements. Exemple :

SYNTAXE EXEMPLE. Tri d'enregistrements

```
# Récupération de tous les étudiants de première année et tri
etudiants = Etudiant.objects.filter(annee=1).order_by('nom')
```

Attention ! Ne confondez pas `filter` et `order_by`. Le premier récupère un sous-ensemble de données, le second trie un ensemble de données.

Récupération d'un enregistrement unique

Le code suivant illustre la récupération d'un enregistrement unique (que ce soit sur la base de la clé primaire ou d'un autre champ).

SYNTAXE EXEMPLE. Récupération d'un enregistrement unique

```
# Récupération d'un enregistrement
xavier = Etudiant.objects.get(id=1)
```

Attention, si zéro ou, au contraire, plus d'un enregistrement est récupéré, une exception est levée. Il vaut donc mieux être sûr de vous !

Suppression d'enregistrements

Le code suivant illustre la suppression d'un enregistrement (1) ou plusieurs (2).

SYNTAXE EXEMPLE. Suppression d'enregistrements

```
# Suppression de l'étudiant Xavier Devos
xavier = Etudiant.objects.get(id=1)
xavier.delete() (1)

# Suppression de tous les étudiants
etudiants = Etudiant.objects.all()
etudiants.delete() (2)
```

Accès à des objets « liés »

Imaginons que l'on veuille accéder à l'auteur d'un message. `Auteur` et `Message` sont deux modèles différents, unis par un lien « 1-n ». Ce n'est, en fin de compte, pas plus compliqué que pour accéder à un attribut élémentaire, puisque dans notre classe `Message` il existe une `ForeignKey` nommée `auteur`.

SYNTAXE EXEMPLE. Récupération de l'auteur d'un message

```
auteur_du_message = Message.objects.get(id=156).auteur
```

En revanche, si on veut récupérer tous les messages d'une personne, c'est un peu plus subtil car dans la classe `Personne`, il n'y a aucun champ pour cela.

Heureusement, Django est là pour nous assister. Il crée automatiquement des attributs permettant de parcourir les relations en sens inverse : ces attributs sont toujours nommés en ajoutant `_set` au nom de la classe visée (écrit en minuscules). On peut donc facilement récupérer tous les messages d'une `Personne`.

SYNTAXE EXEMPLE. Récupération des messages de Xavier Devos

```
xavier = Etudiant.objects.get(matricule='0123456789')
messages_de_xavier = xavier.message_set.all()
```

Remplissage de la base de données

Une fois la base de données créée, se pose la question de son remplissage initial. Tant que nous n'avons aucune page web créant de nouveaux utilisateurs, il nous est impossible d'injecter des données dans la base via le site web de Trombinoscoop.

Une solution possible est de passer directement par l'utilitaire SQLite Database Browser qui donne un accès direct à la base de données, sans passer par Django ou par du code Python. Cette solution est peu pratique : il faut comprendre comment Django a créé les tables et à quoi servent les champs techniques qu'il a ajoutés. Par exemple, pour représenter la relation « n-n » qui lie des personnes à d'autres, Django a créé une table de liaison intermédiaire. S'il n'est pas très compliqué de comprendre comment elle fonctionne, il faut tout de même faire attention à ne pas introduire de données erronées.

Configuration de l'interface d'administration des bases de données

Django offre une autre solution, très simple et élégante, pour administrer la base de données durant les phases de développement (et même après). Il s'agit d'une interface d'administration activable en quelques lignes de configuration. Tous les objets du modèle peuvent être alors gérés grâce à cette interface. La figure suivante l'illustre :

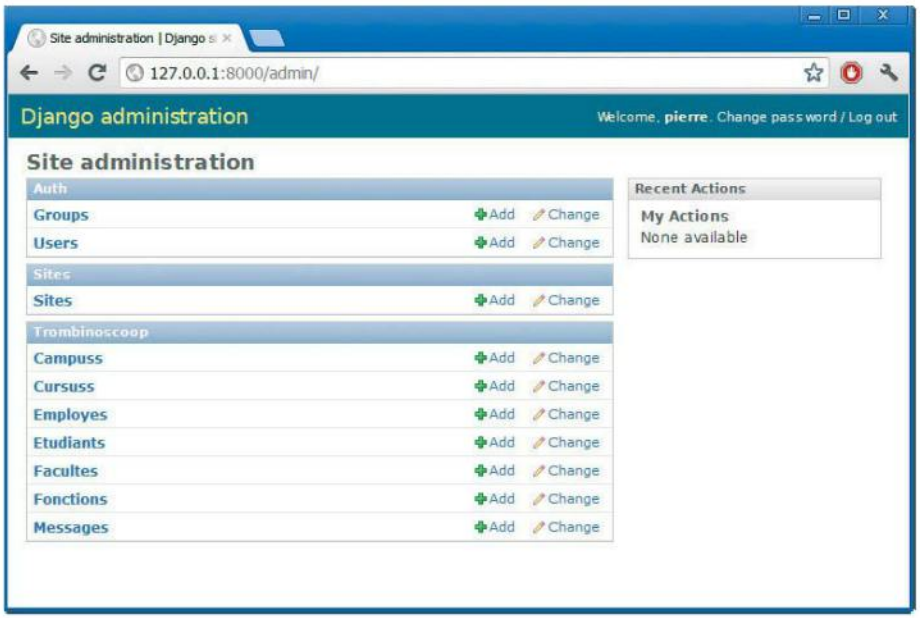


Figure 10–4
Page principale du site d'administration des bases de données

Pour l'obtenir, on doit créer un fichier `admin.py`, au même niveau que nos autres fichiers Python. Dedans, nous allons lister tous les modèles qui doivent être gérés par le site d'administration.

EXEMPLE 10.9 Trombinoscoop. Contenu de admin.py

```
from django.contrib import admin
from Trombinoscoop.models import Faculte, Campus, Fonction,
                                 Cursus, Employe, Etudiant, Message

admin.site.register(Faculte)
admin.site.register(Campus)
admin.site.register(Fonction)
admin.site.register(Cursus)
admin.site.register(Employe)
admin.site.register(Etudiant)
admin.site.register(Message)
```

Nous avons volontairement omis le modèle `Personne` (qui est une classe parent), car nous allons toujours créer soit des étudiants, soit des employés (les classes enfants qui seules donneront naissance à de véritables objets).

Ensuite, on configure `urls.py` afin qu'à une certaine URL corresponde le site d'administration. Cela se fait en ajoutant ces quelques lignes :

EXEMPLE 10.10 Trombinoscoop. Contenu de urls.py

```
from django.conf.urls import patterns, include
from views import welcome, login
from django.contrib import admin

admin.autodiscover()

urlpatterns = patterns('',
  ('^$', login),
  ('^login$', login),
  ('^welcome$', welcome),
  ('^admin/', include(admin.site.urls))
)
```

Si elle ne s'y trouve pas, il faut également ajouter l'application `admin` dans les `INSTALLED_APPS` du fichier de configuration de Django `settings.py` :

CONFIGURATION. **Ajout de l'application admin**

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

Tout dernier détail, le site d'administration de Django est en HTML 5 depuis la version 1.4. On ne peut donc avoir comme type de contenu par défaut `application/xhtml+xml` tel que nous l'avons défini dans `settings.py`. Il faut changer ce paramètre pour revenir à un contenu par défaut qui est en HTML :

CONFIGURATION. **Modification de settings.py**

```
DEFAULT_CONTENT_TYPE = 'text/html'
```

EN PRATIQUE **Quid de nos pages en XHTML 5 ?**

Rassurez-vous, ce n'est pas trop grave si on renvoie un `content-type` qui n'est pas tout à fait exact ; les navigateurs savent s'en accommoder. Espérons quand même que les prochaines versions de Django rendront compatible la partie administration avec XHTML 5 !

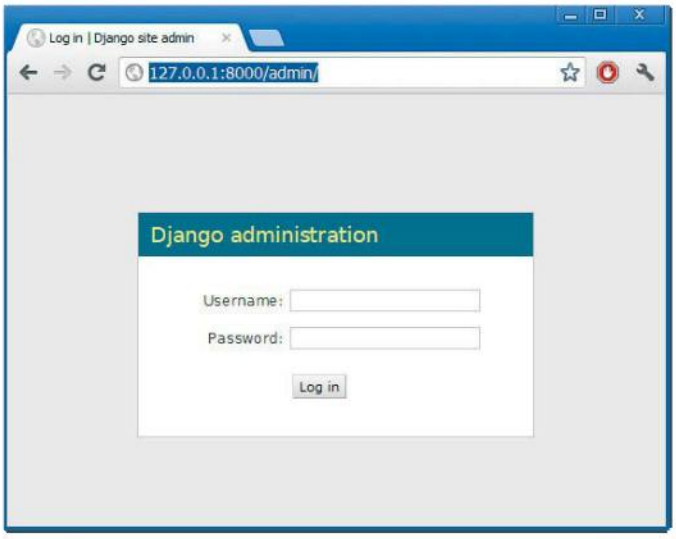
Gestion de la base de données avec l'interface d'administration

Nous pouvons maintenant relancer le projet. En tapant l'URL http://localhost:8000/admin/, nous obtenons la page ci-contre.

Le nom d'utilisateur et le mot de passe attendus sont tout simplement ceux que vous avez choisis lors de la création de la base de données.

Ajoutons quelques données d'exemple. Commençons par l'ajout d'un cursus. Pour ce faire, cliquez sur *Add* en regard de *Cursuss* (deux *s*, car Django essaye de mettre lui-même au pluriel le mot *Cursus*). À l'écran suivant, il suffit d'entrer un intitulé, par exemple « Master en Histoire ». Cliquez ensuite sur *Save and add another* pour sauver le cursus et en ajouter un deuxième, par exemple « Master en Géographie ». Cette fois-ci, cliquez sur *save*. Django affiche alors la liste des cursus encodés dans notre base de données.

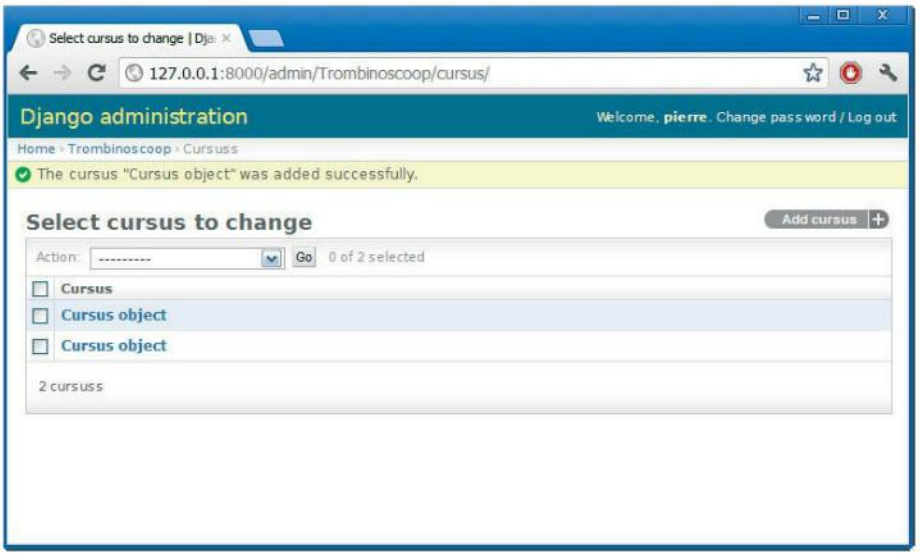
Figure 10–5
Page d'authentification du site d'administration



EN PRATIQUE J'ai oublié mes identifiants

Nous vous avions prévenus, mais si vous ne vous les avez pas retenus, pas de panique : il suffit de détruire la base de données (en supprimant le fichier `sqlite.db`) et de refaire une synchronisation.

Figure 10–6
Liste des cursus encodés



Nous avons bien deux cursus, cependant ils s'affichent tous les deux avec le même intitulé « Cursus object ». Django ne sait pas quel attribut de l'objet `cursus` utiliser lorsqu'il doit l'afficher.

Heureusement, on peut remédier facilement à cela en modifiant légèrement toutes les classes modèles et en ajoutant un peu de code. En fait, pour chacune de ces classes, il suffit de redéfinir la fonction `__unicode__`, comme ceci (surtout, ne touchez pas au code existant !) :

EXEMPLE 10.11 Trombinoscoop. Redéfinition de la fonction __unicode__

```
class Faculte(models.Model):
  # Définition des champs
  def __unicode__(self):
      return self.nom

class Personne(models.Model):
  # Définition des champs
  def __unicode__(self):
      return self.prenom + " " + self.nom

class Campus(models.Model):
  # Définition des champs
  def __unicode__(self):
      return self.nom

class Fonction(models.Model):
  # Définition des champs
  def __unicode__(self):
      return self.intitule

class Cursus(models.Model):
  # Définition des champs
  def __unicode__(self):
      return self.intitule

class Message(models.Model):
  # Définition des champs
  def __unicode__(self):
      if len(self.contenu) > 20:
          return self.contenu[:19] + "..."
      else:
          return self.contenu
```

Nous avons découvert le principe de la redéfinition de méthodes dans le chapitre trois consacré à l'orienté objet en Python. `__unicode__` tout comme `__str__`associe une chaîne de caractères à l'objet. Django sait maintenant que les objets de type `Cursus` doivent s'afficher en utilisant leur intitulé (pour autant que vous n'ayez pas oublié de redémarrer le votre projet Django !) :

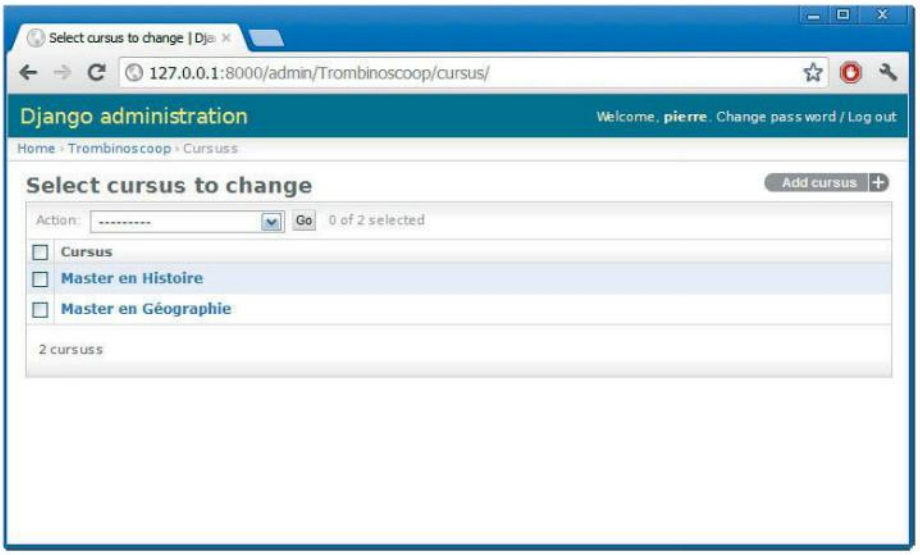


Figure 10–7
Liste des cursus encodés avec leur initulé

Remarquez que pour la classe `Message`, notre fonction `__unicode__` est un peu plus complexe. Elle retourne le contenu du message, tronqué et suivi de points de suspension s'il est trop long (supérieur à 20 caractères).

Vous pouvez ensuite créer des cursus, des facultés, des étudiants, etc.

> **EN PRATIQUE De l'ordre, toujours de l'ordre !**
>
> Notez que les champs d'un étudiant sont pour la plupart obligatoires, y compris `cursus` et `faculté`. Il faut donc préalablement créer sa faculté et son cursus, au risque d'être bloqué à la création de l'étudiant, car on ne pourra pas fournir de valeur pour ces deux champs obligatoires.

Maintenant que nous sommes en mesure d'ajouter des données dans la base, nous pouvons modifier notre procédure de login afin qu'elle fasse coïncider les données encodées par l'utilisateur avec celles de la base de données.

Authentification utilisant la base de données

Afin que notre authentification puisse s'effectuer à partir de la base de données, il faut modifier le code et le charger de vérifier que l'adresse de courriel et le mot de passe introduits sont corrects.

EN PRATIQUE Classe chargée des vérifications

Dans notre projet, c'est la classe `LoginForm` qui se charge de toutes les validations du formulaire d'authentification.

EXEMPLE 10.12 Trombinoscoop. Validation du mot de passe et du courriel utilisant la base de données

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

from django import forms
from Trombinoscoop.models import Personne (1)

class LoginForm(forms.Form):
  email = forms.EmailField(label='Courriel :')
  password = forms.CharField(label='Mot de passe :',
                                    ➥ widget = forms.PasswordInput)
  def clean(self):
    cleaned_data = super(LoginForm, self).clean()
    email = cleaned_data.get("email")
    password = cleaned_data.get("password")

    # Vérifie que les deux champs sont valides
    if email and password:
      result = Personne.objects.filter(mot_de_passe=password,
                                    ➥ courriel=email) (2)
      if len(result) != 1: (3)
        raise forms.ValidationError("Adresse de courriel ou mot de passe
                                    ➥ erroné(e).")

    return cleaned_data
```

Avec Django, deux lignes suffisent pour vérifier le mot de passe et le courriel en utilisant la base de données (trois lignes avec l'import (1)). La première cherche toute personne dont le mot de passe et le courriel correspondent à ce qui a été entré dans le formulaire (2). La deuxième vérifie qu'il n'y a qu'un seul résultat (3).

EN PRATIQUE Cohérence des données

Si plus d'une personne a été trouvée, nous faisons face à un gros problème d'incohérence de données… Il faudra impérativement s'assurer que deux enregistrements de personnes ne pourront avoir le même mot de passe et la même adresse (par exemple lors de la création d'un compte sur le site et surtout en imposant une contrainte sur le modèle (unique=True). Dans un cas réel, on utiliserait django.contrib.auth (modèle:User)).

Cette fois-ci, notre page d'authentification est véritablement terminée. Nous allons maintenant nous intéresser à la page de création de compte, pour laquelle nous allons faire usage des `ModelForm`.

Les ModelForm

Nous avons vu au chapitre précédent que la classe `Form` de Django automatisait la gestion des formulaires et, en particulier, la création du code HTML correspondant.

Django permet d'aller encore plus loin dans l'automatisation des formulaires, en créant un formulaire sur la base d'un modèle. Cette fonctionnalité tire son existence d'un constat pratique : dans les sites web, il existe souvent des formulaires qui servent à créer un objet de la base de données et qui reprennent tous les champs de cet objet. C'est typiquement le cas dans Trombinoscoop : la page de création de compte possède pratiquement tous les champs du modèle `Etudiant` ou `Employe`.

Les `ModelForm` s'utilisent pratiquement de la même manière que les `Form`. Ils ajoutent simplement une méthode `save` qui, une fois les données du formulaire validées, les sauve dans la base de données.

Passons sans plus attendre à la réalisation de notre page de création de compte. Dans un premier temps, nous n'allons permettre que la création d'un profil étudiant.

Création du formulaire Étudiant dans le fichier forms.py

EXEMPLE 10.13 Trombinoscoop. Création du ModelForm pour le modèle Etudiant

```
from django import forms (1)
from Trombinoscoop.models import Etudiant (2)

class StudentProfileForm(forms.ModelForm): (3)
  class Meta: (4)
      model = Etudiant
      exclude = ('amis',)
```

On importe d'abord la bibliothèque `forms` (1) et le modèle `Etudiant` (2). On crée ensuite la classe `StudentProfileForm` qui hérite de `ModelForm` (3). À l'intérieur, on définit une autre classe, `Meta` (4), qui sert à configurer notre formulaire, et en particulier à préciser sur quel modèle il doit se baser. Cela se fait à l'aide de la ligne `model = Etudiant`. Enfin, on termine en excluant l'attribut `amis` du modèle (ce n'est pas à la création du profil que l'utilisateur va préciser qui sont ses amis).

À l'aide de ces quelques lignes, notre formulaire est défini. Comme pour toutes les autres pages du site, il va falloir créer un template, une URL et une vue.

Création de l'URL et de la vue de création de compte

Nommons register la page de création de compte et ajoutons-la à `urls.py` :

EXEMPLE 10.14 Trombinoscoop. Ajout de l'URL register

```
from django.conf.urls import patterns, include
from views import welcome, login, register
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
  ('^$', login),
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^admin/', include(admin.site.urls))
)
```

Cette URL appelle la vue `register` que nous définissons comme suit :

EXEMPLE 10.15 Trombinoscoop. Définition de la vue register

```
from django.shortcuts import render_to_response
from django.http import HttpResponseRedirect
from forms import StudentProfileForm

def register(request):
  if len(request.GET) > 0:
    form = StudentProfileForm(request.GET)
    if form.is_valid():
      form.save(commit=True)
      return HttpResponseRedirect('/login')
    else:
      return render_to_response ('login.html', {'form': form})
  else:
    form = StudentProfileForm()
    return render_to_response ('user_profile.html', {'form': form})
```

La logique de gestion du formulaire est exactement la même que pour le formulaire de login vu au chapitre précédent. La seule différence tient dans l'ajout de la ligne `form.save(commit=True)` dans le cas où le formulaire est valide ; cette instruction a pour effet d'enregistrer dans la base de données les informations du formulaire et donc de créer un nouvel étudiant. Une fois le nouvel utilisateur créé, on le redirige vers la page de login où il pourra s'authentifier.

Création du template de création de compte

Appelons-le `user_profile.html` ; le nom est volontairement générique, car nous allons probablement récupérer ce template pour la page de modification de compte.

EXEMPLE 10.16 Trombinoscoop. Contenu de user_profile.html

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

{% block title %}Création d'un profil{% endblock %} (1)

{% block bodyId %}userProfilePage{% endblock %} (2)

{% block content %}
<h1>Création d'un compte</h1> (5)
<form action="register" method="get"> (3)
  {{ form.as_p }}
  <p>
    <input type="submit" value="Créer un compte" /> (4)
  </p>
</form>
{% endblock %}
```

On observe quelques petits changements par rapport à notre page de login : le titre de la page (1), l'`id` de la balise `body` (2), la page de destination de l'attribut `action` (3), l'intitulé du bouton de soumission (4) et l'ajout d'un titre `<h1>` (5).

Un peu de mise en forme

Il reste un petit détail à régler : les CSS. Souvenez-vous, nous avions le code suivant pour positionner l'élément `section` :

EXEMPLE 10.17 Trombinoscoop. Contenu actuel des CSS

```
body#loginPage section#content {
  position: absolute;
  top: 150px;
  left: 50%;
  width: 500px;
  margin: 0 0 0 -250px;
  padding: 0;
}
```

Ce code ne s'applique qu'aux éléments `section` dont l'`id` est `content` qui se trouvent dans une balise `body` dont l'`id` est `loginPage`. Or, nous venons de changer l'`id` de la balise `body` pour notre nouveau template. Comme le positionnement tel qu'écrit con-

vient aussi pour la page de création de compte, et sans doute pour les futures pages que nous allons créer, nous allons appliquer ce même style à toutes les pages, à l'exception de la position `top` que l'on va remonter un peu pour toutes les pages autres que celle du login. Par la même occasion, nous allons également améliorer l'aspect du titre qu'on a ajouté.

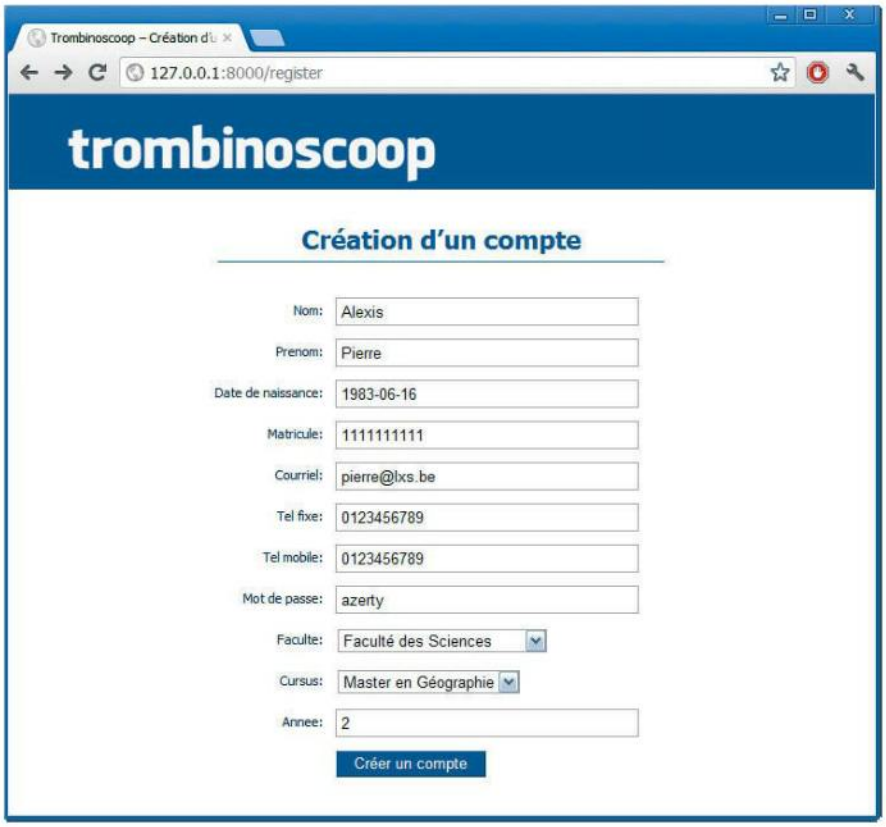
EXEMPLE 10.18 Trombinoscoop. Modification des CSS

```
body section#content {
  position: absolute;
  top: 110px;
  left: 50%;
  width: 500px;
  margin: 0 0 0 -250px;
  padding: 0;
}

body#loginPage section#content {
  top: 150px;
}

h1 {
  text-align: center;
  font-size: 22px;
  margin: 0px 60px 30px 60px;
  padding: 0 0 5px 0;
  color: #3b5998;
  border-bottom: 1px solid #3b5998;
}
```

Et voilà ! Maintenant nous allons pouvoir tester notre nouvelle page ! Relancez le projet ; le résultat est le suivant lorsqu'on tape l'adresse http://localhost:8000/register.

Rentrez les informations comme Django vous force à le faire (rappelez-vous que c'est lui qui valide le formulaire à sa manière). Avouons-le, le peu de code qu'il a fallu écrire pour obtenir cette superbe page est impressionnant !

Figure 10–8
Création d'un compte



Finalisation de la page de création de compte

Notre page nous permet pour l'instant de créer des comptes étudiants, mais pas des comptes employés. Dans nos wireframes, une option permettait de choisir la création de l'un ou l'autre profil, avec les champs qui apparaissaient ou disparaissaient selon l'option choisie.

Nous voulons donc en réalité un formulaire complexe, avec une partie commune et des champs propres à chaque type de profil. Comment réaliser cela en Django ? La réponse n'est pas si évidente.

Création de deux formulaires : un pour les étudiants et un pour les employés

Le plus simple est d'inclure dans notre HTML deux formulaires : l'un reprenant tous les champs d'un étudiant et l'autre tous ceux d'un employé. Peu importe si certains champs se trouvent deux fois dans la page. Ajoutons au début de la page une

liste déroulante pour choisir le type de profil à créer ; en fonction du choix de l'utilisateur, nous afficherons l'un ou l'autre formulaire.

Commençons par créer le formulaire basé sur le modèle `Employe`.

EXEMPLE 10.19 Trombinoscoop. Création du ModelForm pour le modèle Employe

```
from django import forms
from Trombinoscoop.models import Employe

class EmployeeProfileForm(forms.ModelForm):
  class Meta:
      model = Employe
      exclude = ('amis',)
```

Gestion des deux formulaires dans la vue

EXEMPLE 10.20 Trombinoscoop. Gestion des deux formulaires dans la vue

```
from forms import StudentProfileForm, EmployeeProfileForm

def register(request):
  if len(request.GET) > 0 and 'profileType' in request.GET: (1)
    studentForm = StudentProfileForm(prefix="st")
    employeeForm = EmployeeProfileForm(prefix="em")
    if request.GET['profileType'] == 'student': (3)
      studentForm = StudentProfileForm(request.GET, prefix="st")
      if studentForm.is_valid(): (4)
        studentForm.save(commit=True)
        return HttpResponseRedirect('/login')
    elif request.GET['profileType'] == 'employee':
      employeeForm = EmployeeProfileForm(request.GET, prefix="em")
      if employeeForm.is_valid():
        employeeForm.save(commit=True)
        return HttpResponseRedirect('/login')
    # Le formulaire envoyé n'est pas valide
    return render_to_response('user_profile.html',
                              {'studentForm': studentForm,
                               'employeeForm': employeeForm}) (5)
  else: (2)
    studentForm = StudentProfileForm(prefix="st")
    employeeForm = EmployeeProfileForm(prefix="em")
    return render_to_response('user_profile.html',
                              {'studentForm': studentForm,
                               'employeeForm': employeeForm})
```

Ce code est un peu plus subtil et il comprend plusieurs petites astuces. La première, c'est que nous utilisons un paramètre supplémentaire qui nous permet de savoir quel formulaire a été envoyé. Ce paramètre s'appelle `profileType` et nous devrons l'ajouter dans notre template. Il peut prendre deux valeurs :

- `student` s'il s'agit du formulaire de création d'un profil étudiant ;
- `employee` s'il s'agit du formulaire de création d'un profil employé.

À la première ligne, nous vérifions si au moins un formulaire a été soumis, en évaluant le nombre de paramètres reçus (1). On s'assure également qu'on a bien reçu le paramètre `profileType`.

Si aucun formulaire n'a été soumis (2), c'est simple. On crée deux `ModelForm` vierges qu'on transmet à notre template. Jusque-là, c'est presque comme avant, si ce n'est qu'on a deux formulaires et qu'on les préfixe. Comme les champs sont pour la plupart communs aux deux types de personnes, le préfixage évite à Django de créer des `id` qui portent le même nom, alors que ceux-ci doivent être uniques dans notre HTML.

Si un formulaire a été soumis, on regarde lequel. Si c'est le formulaire de création d'étudiant (3), on crée un `ModelForm` étudiant qu'on initialise avec les données reçues et un `ModelForm` employé vierge. Ensuite, on vérifie si le formulaire étudiant est valide (4). S'il l'est, tout va bien, on enregistre les données et on redirige l'utilisateur vers la page de login. Sinon, on réaffiche la page de création de profil en passant en paramètre les deux `ModelForm` (5).

On réalise exactement l'inverse si c'est le formulaire de création d'employé qui est soumis.

Gestion des deux formulaires dans le template

Voyons maintenant ce que nous avons ajouté dans le template `userprofile.html`. Les choses se corsent petit à petit :

EXEMPLE 10.21 Trombinoscoop. Gestion des deux formulaires dans le template

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

{% block title %}Création d'un profil{% endblock %}

{% block bodyId %}userProfilePage{% endblock %}

{% block content %}
<form> (1)
  <p>
    <label for="profileType">Vous êtes :</label>
```

```
    <select id="profileType">
      <option value="student" {% if studentForm.is_bound %} (2)
              selected="selected" {% endif %}>Étudiant</option>
      <option value="employee" {% if employeeForm.is_bound %}
              selected="selected" {% endif %}>Employé</option>
    </select>
  </p>
</form>

<form action="register" method="get" id="studentForm">
  {{ studentForm.as_p }}
  <p>
    <input type="hidden" name="profileType" value="student" /> (3)
    <input type="submit" value="Créer un compte" />
  </p>
</form>

<form action="register " method="get" id="employeeForm">
  {{ employeeForm.as_p }}
  <p>
    <input type="hidden" name="profileType" value="employee" /> (3)
    <input type="submit" value="Créer un compte" />
  </p>
</form>
{% endblock %}
```

Premier constat : nous avons trois formulaires, alors que nous nous attendions à n'en définir que deux.

Le premier formulaire est là pour accueillir la liste déroulante (1). Ce n'est pas obligatoire, mais c'est plus facile du point de vue CSS. En plaçant la liste déroulante dans un formulaire, les styles que nous avons déjà définis s'appliqueront. De toute façon, ce formulaire ne possède ni bouton de soumission, ni attribut `action`.

Notre liste déroulante possède deux options, étudiant ou employé ; c'est logique. Le code `{% if studentForm.is_bound %} selected="selected" {% endif %}` (2) mérite en revanche une explication. Il permet de traiter le cas où un formulaire soumis n'est pas valide : lorsque la page est réaffichée, la valeur précédemment choisie par l'utilisateur est pré-sélectionnée dans la liste déroulante.

EN PRATIQUE L'attribut is_bound du formulaire

L'attribut `is_bound` du formulaire indique si on a affaire à un formulaire vierge ou un formulaire qui a été rempli à l'aide de paramètres reçus. Si le formulaire étudiant est vierge, c'est que ce n'est pas lui qui a été soumis. En revanche, s'il n'est pas vierge, c'est que c'est lui qui a été soumis. Dans ce cas, on ajoute l'attribut `selected="selected"` qui sélectionne l'élément `Étudiant` dans la liste.

Ensuite, nous avons la définition de nos deux formulaires. Le code est similaire à ce que nous avons vu précédemment, si ce n'est qu'un nouveau champ a été ajouté (3) : le champ `profileType` qui est de type `hidden`. C'est lui qui nous indique lequel des deux formulaires a été soumis. Et comme il s'agit d'un champ technique que l'utilisateur de notre site n'a pas à voir, on le cache.

Un peu de dynamisme

Enfin, nous aimerions que, dynamiquement, lorsque l'utilisateur choisit un élément dans la liste, le bon formulaire soit affiché et l'autre masqué.

Nous allons utiliser du jQuery pour cela, comme nous l'avons découvert au chapitre 4. La première chose à réaliser consiste à ajouter une référence à la bibliothèque dans l'en-tête de la page HTML. Cela se fait dans le template `base.html`.

EXEMPLE 10.22 Trombinoscoop. Ajout de la bibliothèque jQuery

```
<head>
  <title>Trombinoscoop - {% block title %}Bienvenue{% endblock %}</title>
  <link rel="stylesheet" type="text/css" href="/static/css/style.css" />
  <script type="text/javascript" src="http://code.jquery.com/jquery-1.7.2.min.js">
  </script>
</head>
```

Dans le template `user_profile.html`, insérons le code suivant, qui apporte le dynamisme voulu. Ce code est à ajouter dans le bloc `content`, par exemple au tout début, juste avant la balise `<h1>`.

EXEMPLE 10.23 Trombinoscoop. Affichage dynamique du bon formulaire

```
<script type="text/javascript">
  function displayRightForm() (1) {
    if ($('#profileType').val() == 'student') {
      $('#employeeForm').hide();
      $('#studentForm').show();
    }
    else {
      $('#studentForm').hide();
      $('#employeeForm').show();
    }
  }
  $(document).ready(displayRightForm); (3)
  $('#profileType').change(displayRightForm); (2)
</script>
```

On définit d'abord une fonction qui affiche et masque l'un ou l'autre des deux formulaires en fonction de la valeur de la liste déroulante. (1)

Cette fonction doit ensuite être appelée après deux événements :

- chaque fois que l'utilisateur change la valeur de la liste déroulante (2) ;
- une première fois, au chargement de la page (via l'événement `ready` du document) (3).

Tout ce code nous permet finalement d'obtenir le magnifique résultat suivant :

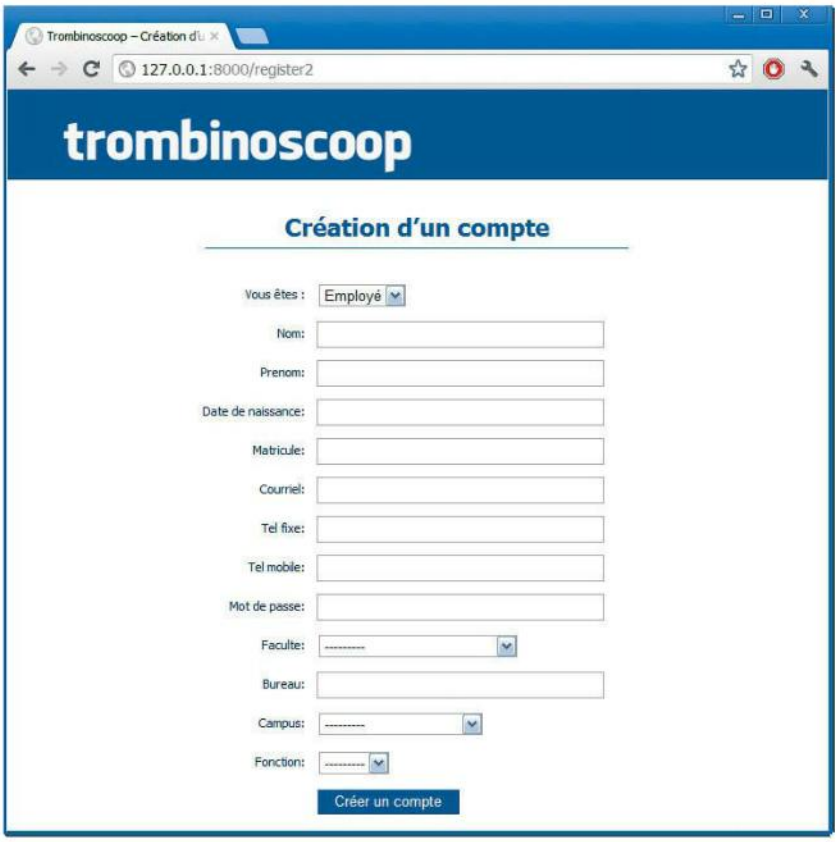


Figure 10–9
On peut maintenant choisir le type de compte que l'on désire créer.

Ai-je bien compris ?

- Pourquoi dans ce chapitre ne voit-on pas une seule ligne de SQL ? Ce langage est-il utilisé par notre application web ?
- Comment exprime-t-on les relations « 1-n », « n-n » et « 1-1 » dans les modèles ?
- À quoi servent les `ModelForms` ?

11

Comprendre et utiliser les sessions

Ce chapitre va nous familiariser avec les sessions, qui servent à sauvegarder un contexte global d'interaction entre deux visites sur le serveur web. Les sessions fonctionnent grâce aux célèbres et tant décriés cookies qui mémorisent les accès au serveur. Django a sa propre manière de répartir cette mémoire entre le client et la base de données côté serveur. Le projet Trombinoscoop sera enrichi par exemple par la mémorisation de l'identité du visiteur du site.

EN PRATIQUE **Les applications Django travaillent pour vous**

Notez que l'application django.contrib.auth peut faire pour vous la grande partie de ce que nous expliquons dans ce chapitre, mais nous choisissons bien sûr de nous en passer, pour l'intérêt de l'exercice.

SOMMAIRE

- Explication de la notion de session
- Explication et utilisation des cookies avec Django
- Prise en charge des sessions dans Trombinoscoop

Au chapitre précédent, nous vous annoncions, triomphants, que la page d'authentification était totalement terminée. En réalité, plusieurs problèmes de taille continuent d'entacher notre mécanisme d'authentification :

- Lorsque l'authentification réussit, l'utilisateur est redirigé vers la page d'accueil (`welcome`). Or, sur cette page on ne vérifie plus du tout l'identité du visiteur. Il suffit donc de connaître l'URL de la page d'accueil pour facilement contourner la page d'authentification. Voilà une aberrante faille de sécurité !
- Sur la page d'accueil, on ne sait plus qui s'y est connecté ; tous les paramètres transmis à la page de login sont perdus pour la page suivante. Chaque page possède sa propre « mémoire », hermétique au contexte des autres. Aucun contexte global ne nous permet, par exemple, de stocker l'identité de la personne connectée.

On pourrait résoudre ces deux problèmes avec les techniques que nous avons déjà vues. Pour s'assurer qu'une page est visitée par un utilisateur authentifié et autorisé, il suffirait de transmettre le login et le mot de passe de ce dernier en paramètre de chaque page web. Dans chaque vue, le premier traitement que l'on réaliserait serait de vérifier si le login et le mot de passe transmis sont corrects. Ce mécanisme nous permettrait par la même occasion de savoir sur chaque page qui en est le visiteur. Cependant, vous en conviendrez, ce n'est pas très pratique et, surtout, c'est on ne peut plus répétitif.

À quoi servent les sessions

Grâce aux sessions, nos deux problèmes vont être résolus :

- Nous saurons à tout moment quel utilisateur est à l'origine d'une requête HTTP. Dès que celui-ci s'est authentifié sur la page de login, il suffit de sauvegarder son identité dans un cookie. À chaque requête vers toute page de notre site, le navigateur enverra l'identité enregistrée dans le cookie.
- Nous protégerons facilement toutes nos pages d'un accès indésirable par une personne qui n'est pas authentifiée. Il suffira de vérifier dans la session si une identité est présente. Si oui, c'est que le visiteur est passé avec succès par la page de login. Si non, c'est que l'auteur de la requête n'est pas identifié et qu'il essaie de se faufiler dans notre site sans s'y être invité ; dans ce cas, on le redirige vers la page de login.

DÉFINITION Sessions

Les sessions servent à sauvegarder un contexte, c'est-à-dire un ensemble de données liées à chaque visite sur notre site web.

Les sessions fonctionnent grâce aux célèbres cookies, pour lesquels nous nous refusons le moindre jeu de mot !

DÉFINITION Cookies

Les cookies sont des fichiers stockés par le navigateur dans le disque dur de l'internaute, lors de la visite d'un site web. Ils sont créés à la demande du site visité et contiennent toute donnée que ce dernier juge utile de sauvegarder. À chaque requête HTTP, le contenu du cookie est inclus dans l'en-tête de la requête et peut être exploité par le serveur web pour adapter sa réponse en fonction de vos visites précédentes. C'est grâce à ce mécanisme que l'on va attacher un contexte à chaque « session » de visite de notre site web et passer de précieuses informations de pages en pages.

Voyons maintenant comment Django implémente ce mécanisme de session et appliquons-le à Trombinoscoop.

Les sessions selon Django

Par défaut, Django implémente le principe des sessions un peu différemment. Plutôt que d'enregistrer toutes les données de la session dans un cookie, Django les sauvegarde dans la base de données et attribue un « identifiant de session » aux données enregistrées dans la base ; c'est uniquement cet identifiant qui est sauvegardé dans le cookie côté client.

L'idée étant toujours de « tracer » un internaute d'une visite à l'autre sur le site, les sessions viennent résoudre certains défauts inhérents aux cookies :

- plus grosse capacité de stockage, les données de session étant stockées côté serveur, à la différence des cookies stockés côté client ;
- sécurité des données, car il est impossible d'accéder aux variables de session, via JavaScript par exemple ;
- moins de trafic réseau, intégrité et sécurité des données mieux garanties, car, à la différence des cookies, les « informations de session » ne transitent pas sans arrêt entre le navigateur et le serveur.

EN PRATIQUE Sécurité des cookies

Sauvegarder des données personnelles dans des cookies est rarement une bonne idée. Ces petits fichiers sont facilement accessibles, lisibles et modifiables. Imaginons qu'on stocke dans le cookie le matricule de l'utilisateur connecté. Un bidouilleur aura vite fait de créer un faux cookie contenant le matricule de son voisin. Notre site web n'y verra alors que du feu et pensera que l'utilisateur connecté est le malheureux voisin. À l'inverse, déduire un identifiant de session, calculé aléatoirement par Django, est pratiquement impossible.

Cette technique comporte néanmoins un désavantage de taille : si la base de données n'est pas nettoyée régulièrement des sessions inactives, sa taille risque très vite d'exploser. Django propose deux solutions pour remédier à cela :

- Offrir une page de déconnexion au visiteur. Cependant, les internautes qui n'utilisent jamais les liens *Se déconnecter* et qui ferment simplement la fenêtre de leur navigateur sont nombreux. Pour ceux-là, comme on ne sait jamais quand leur session se termine, Django préconise la deuxième solution.
- Lancer régulièrement la commande `django-admin.py cleanup` qui se charge de nettoyer la base de données. Comme nous sommes pour l'instant en phase de développement et que personne d'autre que nous ne va visiter notre site, il ne sera pas utile de nettoyer la base de données. Cependant, le jour où vous ouvrirez votre site au public, il faudra penser à écrire un script de nettoyage qu'il faudra exécuter à une certaine fréquence. La documentation de Django explique précisément comment faire.

Utilisation d'une session

Configuration

Par défaut, les sessions sont activées dans Django. En effet, dans le fichier `settings.py`, on trouve déjà les lignes suivantes (1) :

CONFIGURATION. Activation des sessions dans settings.py

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware', (1)
    'django.middleware.csrf.CsrfViewMiddleware', # pour éviter des posts
                                                 # tiers malicieux
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
)

INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions', (1)
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'Trombinoscoop',
    'django.contrib.admin',
)
```

Souvenez-vous : lorsque nous avons créé la base avec `Sync DB`, nous avons constaté que Django créait de nombreuses tables supplémentaires. Il s'agit de celles qui sont nécessaires à la sauvegarde des données de session.

Maniement d'une variable de session

Une fois activées, les sessions sont simplissimes à utiliser. Il suffit d'utiliser l'objet `session`, disponible dans l'objet `request` qu'on reçoit automatiquement en paramètre de chacune de nos vues.

SYNTAXE EXEMPLE. Utilisation des sessions

```
def login(request):

    # Lecture de la valeur d'une variable de session
    langue_choisie = request.session['langue']

    # Changement de la valeur d'une variable de session
    request.session['langue'] = 'fr'

    # Suppression d'une variable de session
    del request.session['langue']

    # Verification de la presence d'une variable de session
    if 'langue' in request.session:
```

EN PRATIQUE Durée de vie des cookies de session

Il est possible de définir la durée de vie des cookies de session. Elle peut être éternelle, limitée à une date bien précise, ou exprimée en jours. Plus périssable encore, on peut faire en sorte que le cookie soit détruit lorsque le visiteur ferme son navigateur. Ces durées de vie différentes expliquent pourquoi, sur certains sites, il faut s'authentifier de nouveau à chaque fois qu'on relance son navigateur, alors que sur d'autres, on reste authentifié pendant plusieurs jours.
La durée de vie par défaut des cookies se définit dans le fichier `settings.py` à l'aide des paramètres `SESSION_EXPIRE_AT_BROWSER_CLOSE` et `SESSION_COOKIE_AGE`. Par défaut, les cookies de Django n'expirent pas à la fermeture du navigateur et leur durée de vie est établie à deux semaines.
Dans une optique de test, nous allons mettre la variable `SESSION_EXPIRE_AT_BROWSER_CLOSE` à `True`, afin de pouvoir facilement supprimer le cookie de session à la fermeture du navigateur.

Enregistrement de l'utilisateur authentifié

Reprenons le code de notre vue `login` qui gère l'authentification d'un utilisateur, et adaptons-le afin de sauvegarder l'identité du visiteur authentifié.

EXEMPLE 11.1 Trombinoscoop. Enregistrement de l'utilisateur authentifié dans la session

```
from Trombinoscoop.models import Personne

def login(request):
  # Teste si le formulaire a été envoyé
  if request.method=="POST":
    form = LoginForm(request.POST)
    if form.is_valid():
      user_email = form.cleaned_data['email'] (1)
      logged_user = Personne.objects.get(courriel=user_email) (2)
      request.session['logged_user_id'] = logged_user.id (3)
      return HttpResponseRedirect('/welcome')
  else:
    form = LoginForm()
    return render_to_response ('login.html', {'form': form})
```

Seules trois lignes de code ont été ajoutées (mis à part l'import). La première récupère à partir du formulaire l'adresse de courriel que l'utilisateur a introduite (1). On utilise ensuite cette adresse de courriel pour récupérer dans la base de données l'objet `Personne` qui possède cette adresse (2). On sauvegarde ensuite dans la session l'`id` de cette personne. Ensuite seulement, on fait la redirection vers la page d'accueil.

Vérification que l'utilisateur est bien authentifié

Modifions la page d'accueil pour vérifier que le visiteur est bien authentifié, c'est-à-dire qu'un `id` d'utilisateur est bien présent dans la session. Si ce n'est pas le cas, on redirige l'utilisateur vers la page de login.

EXEMPLE 11.2 Trombinoscoop. Vérification que l'utilisateur est authentifié

```
from Trombinoscoop.models import Personne

def welcome(request):
  if 'logged_user_id' in request.session: (1)
    logged_user_id = request.session['logged_user_id'] (2)
    logged_user = Personne.objects.get(id=logged_user_id) (3)
    return render_to_response('welcome.html',
                              ➥ {'logged_user': logged_user}) (4)
  else:
    return HttpResponseRedirect('/login') (5)
```

On vérifie d'abord qu'on a bien un `id` d'utilisateur dans la session (1). Si oui, c'est que le visiteur de notre site est authentifié (2) ; on le récupère alors dans la base de données (3), puis on appelle le template `welcome.html` en lui passant en paramètre

l'utilisateur authentifié ❹. L'affichage de la page d'accueil sera personnalisé avec des données de l'utilisateur. En revanche, si on ne trouve pas d'`id` dans la session, c'est que le visiteur n'est pas authentifié ; on le redirige donc vers la page de login ❺.

Utilisation des données de la session

Dans notre template, écrivons le code suivant afin d'utiliser les données de l'utilisateur authentifié ❶. Par la même occasion, nous allons modifier l'aspect de la page d'accueil en utilisant le template de base ❷.

EXEMPLE 11.3 Trombinoscoop. Utilisation des données de l'utilisateur dans welcome.html

```
{% extends "base.html" %} ❷

{% block title %}Accueil{% endblock %}

{% block bodyId %}welcomePage{% endblock %}

{% block content %}
<p>{{ logged_user.prenom }} {{ logged_user.nom }}, ❶
bienvenue sur Trombinoscoop !</p>
{% endblock %}
```

Testons maintenant notre nouvelle page d'accueil. Lorsqu'on n'est pas authentifié, on est redirigé vers la page de login. Si l'authentification se passe correctement, on obtient le résultat suivant :

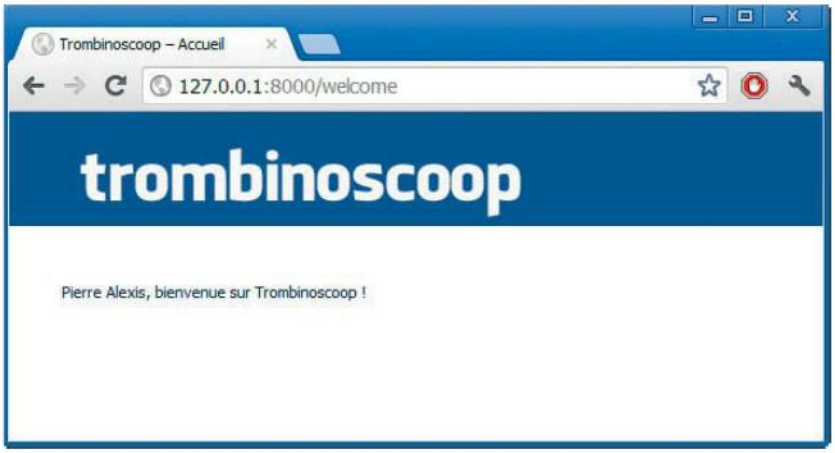


Figure 11–1
La nouvelle page d'accueil personnalisée

Que trouve-t-on dans le cookie ?

Voyons ce qui se trouve sauvegardé dans le cookie.

> EN PRATIQUE **Outil : éditeur de cookie**
>
> Les navigateurs récents possèdent généralement des outils pour visionner le contenu des cookies. C'est le cas par exemple de Google Chrome : *Outils > Outils de développement > Cookies*.

EXEMPLE 11.4 Trombinoscoop. Cookie de notre site

```
Nom du cookie: sessionid
Valeur: f3055087679de66ed70b26fbffb8199e
```

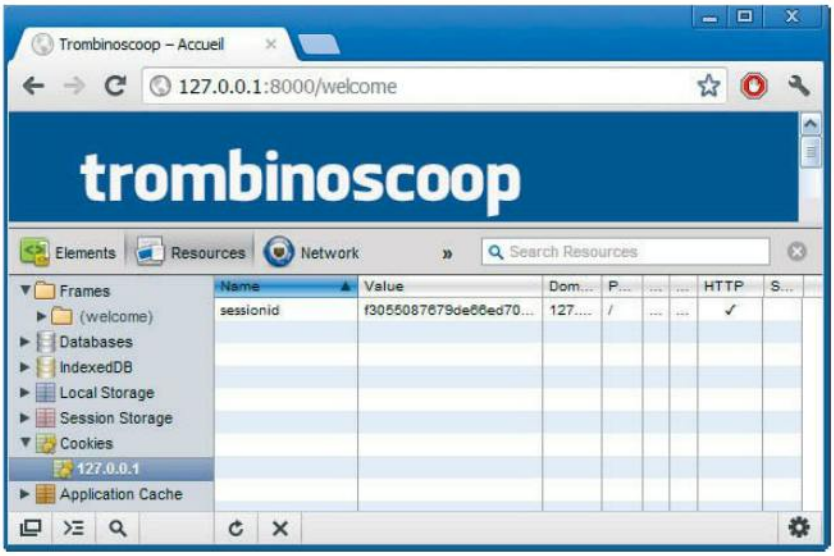


Figure 11–2
Contenu du cookie de Trombinoscoop (dans Google Chrome)

Que trouve-t-on dans la session ?

Malheureusement, l'outil SQLite Database Browser ne nous aidera guère à le savoir : les données sont encryptées par Django. Sécurité oblige !

Protection des pages privées

Maintenant que nous savons vérifier si un utilisateur est authentifié, nous allons pouvoir protéger toutes nos pages privées. Plutôt que de dupliquer le code de la vue `welcome`, plaçons-le dans une fonction dont le rôle sera également de récupérer, le cas échéant, l'utilisateur authentifié de la base de données. Nommons la fonction `get_logged_user_from_request` et insérons-la dans le fichier `views.py`.

EXEMPLE 11.5 Trombinoscoop. Fonction get_logged_user_from_request

```
from Trombinoscoop.models import Etudiant, Employe

def get_logged_user_from_request(request):
  if 'logged_user_id' in request.session: (1)
    logged_user_id = request.session['logged_user_id']
    # On cherche un etudiant
    if len(Etudiant.objects.filter(id=logged_user_id)) == 1: (3)
      return Etudiant.objects.get(id=logged_user_id)
    # On cherche un Employe
    elif len(Employe.objects.filter(id=logged_user_id)) == 1: (4)
      return Employe.objects.get(id=logged_user_id)
    # Si on n'a rien trouve
    else:
      return None (5)
  else:
    return None (2)
```

On regarde si on trouve un `id` d'utilisateur dans la session (1). Si non, on renvoie directement `None` (2). Si un utilisateur est authentifié, alors on recherche d'abord s'il s'agit d'un étudiant (3), puis sinon un employé (4) et si rien n'est trouvé, alors on retourne `None` (5).

Notre vue `welcome` devient alors :

EXEMPLE 11.6 Trombinoscoop. Vue gérant la page d'accueil

```
def welcome(request):
  logged_user = get_logged_user_from_request(request)
  if logged_user:
    return render_to_response('welcome.html',
                              {'logged_user': logged_user})
  else:
    return HttpResponseRedirect('/login')
```

Le code est on ne peut plus simple ! On appelle la fonction `get_logged_user_from_request` et on vérifie sa valeur de retour. Si elle n'est pas `None`, c'est qu'un utilisateur est authentifié.

Amélioration de notre page d'accueil

Pour l'instant, notre page d'accueil ne ressemble pas vraiment au wireframe que nous avions conçu au départ.

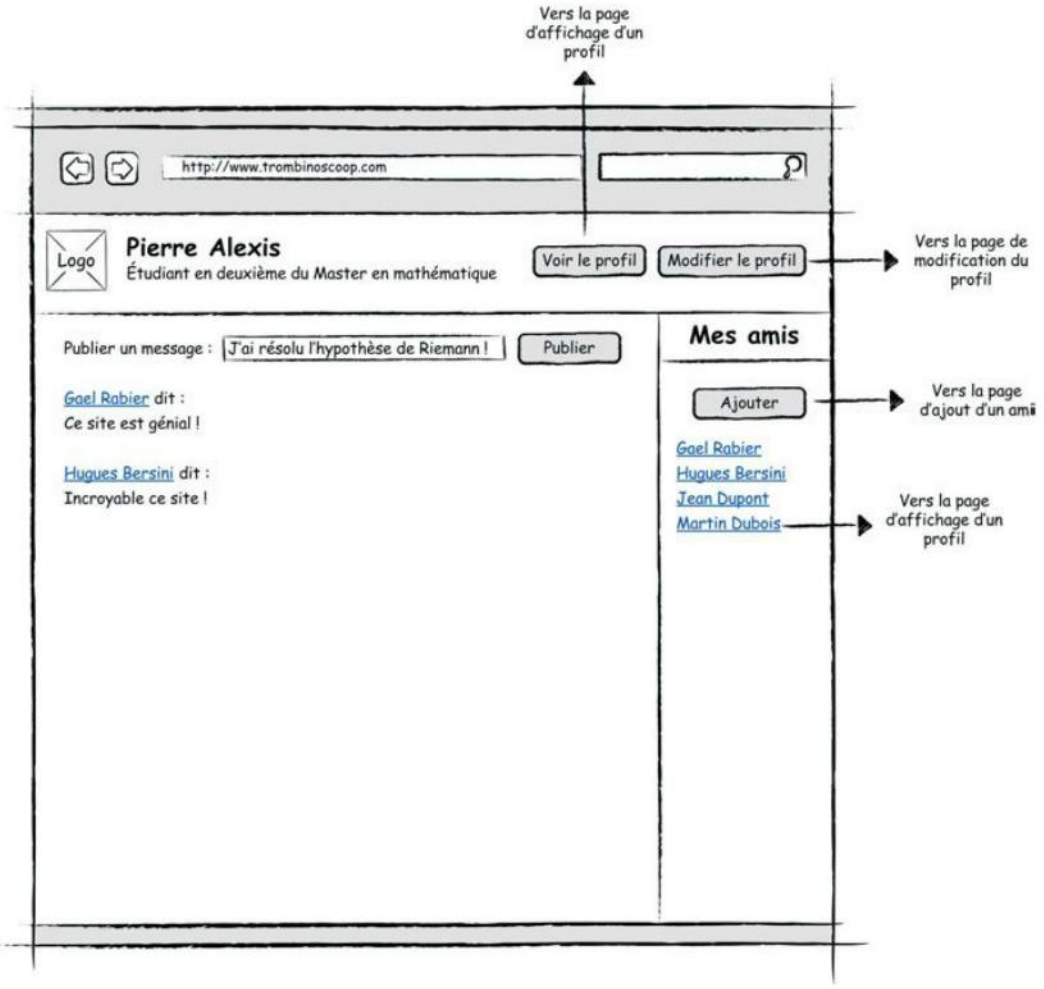


Figure 11–3 Wireframe de la page d'accueil

Personnalisation de la bannière

Commençons par nous occuper de la bannière qui doit contenir les données de l'utilisateur connecté. Nous aurons besoin de savoir ce qu'est l'utilisateur connecté : un étudiant ou un employé. Nous pourrions vérifier le type de l'objet `logged_user` passé au template. Malheureusement, ce n'est pas possible, les templates n'offrant pas d'instruction pour connaître le type exact d'un objet Python.

Le plus propre est donc d'ajouter un attribut à la classe `Personne`, qu'on surcharge dans les classes `Étudiant` et `Employé` pour retourner le bon type.

EXEMPLE 11.7 Trombinoscoop. Ajout d'une méthode permettant de connaître le type

```
class Personne(models.Model):
  # Autres champs
  type_de_personne = 'generic'

class Employe(Personne):
  # Autres champs
  type_de_personne = 'employee'

class Etudiant(Personne):
  # Autres champs
  type_de_personne = 'student'
```

EN PRATIQUE Pensez à synchroniser la base...

Comme nous avons ajouté un champ, il ne faut pas oublier de refaire un `sync-db`.

Nous pouvons maintenant remplir notre bannière. Celle-ci n'étant plus juste une simple et triste bannière vide comme pour la page de login, nous allons pour la première fois redéfinir le bloc `headerContent`. Voici le code que nous avons placé dans le template `welcome.html` :

EXEMPLE 11.8 Trombinoscoop. HTML de l'en-tête de la page d'accueil

```
{% block headerContent %}
<p id="name">{{ logged_user.prenom }} {{ logged_user.nom }}</p>
<p id="function">
  {% if logged_user.type_de_personne == 'student' %}
  Étudiant en
    {% if logged_user.annee == 1 %}
      {{ logged_user.annee }}ère
    {% else %}
      {{ logged_user.annee }}ème
    {% endif %}
    {{ logged_user.cursus.intitule }}
  {% else %}
    {{ logged_user.fonction.intitule|capfirst }}
    dans la faculté {{ logged_user.faculte }}
  {% endif %}
</p>
<p id="profileLinks">
  <a href="???" class="buttonLink">Voir le profil</a>
  <a href="???" class="buttonLink">Modifier le profil</a>
</p>
{% endblock %}
```

Le code est plutôt simple. La complexité apparente vient de l'affichage du statut de l'utilisateur, en-dessous de son nom. En effet, on n'affiche pas la même chose pour les étudiants et les employés.

Nos instructions CSS restent à modifier, afin que le contenu de l'en-tête soit bien positionné et formaté :

EXEMPLE 11.9 Trombinoscoop. Modification de style.css

```
body#welcomePage header {
  background: #3b5998 url('../img/shortLogo.png') no-repeat 20px 25px;
  color: #FFFFFF;
}

body#welcomePage header p#name {
  font-size: 22px;
  margin: 20px 0 0 70px;
}

body#welcomePage header p#function {
  margin: 0 0 0 70px;
}

body#welcomePage header p#profileLinks {
  position: absolute;
  right: 40px;
  top: 25px;
}

a.buttonLink {
  border-style: solid;
  border-width: 1px;
  border-color: #d9dfea #0e1f5b #0e1f5b #d9dfea;
  background-color: #3b5998;
  color: #FFFFFF;
  padding: 2px 15px 3px 15px;
  text-align: center;
  text-decoration: none;
}
```

Notez que nous avons changé le logo pour un plus petit, `shortLogo.png`, qui prend moins de place et reprend uniquement la première lettre de Trombinoscoop.

Sauvegardons et relançons le serveur web. Le résultat est le suivant :

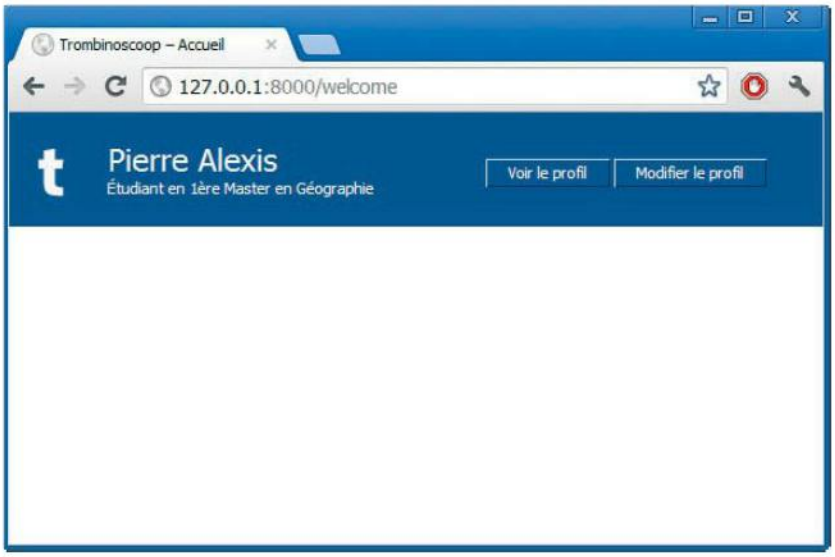


Figure 11–4
La page d'accueil et sa bannière

Division du corps de la page

Le contenu de la page situé en-dessous de la bannière est divisé en deux grandes parties : à gauche la liste des messages publiés par les amis, à droite la liste des amis.

Commençons par le code HTML de la partie à placer dans le bloc `content` du template `welcome.html`. Ajoutons deux éléments `section` : l'un pour la liste des messages, l'autre pour la liste des amis.

EXEMPLE 11.10 Trombinoscoop. Divison du contenu en deux sections

```
{% block content %}
<section id="messageList">
  test
</section>
<section id="friendList">
  test
</section>
{% endblock %}
```

Au niveau des CSS, nous allons placer les deux sections l'une à côté de l'autre, en leur donnant une largeur bien précise. Au passage, nous allons, pour cette page d'accueil, agrandir la largeur de l'élément accueillant le contenu. Nous avons besoin de plus de place que sur les autres pages. Le code CSS que nous ajoutons est le suivant :

EXEMPLE 11.11 Trombinoscoop. Positionnement des sections de contenu

```
body#welcomePage section#content {
  width: 800px;
  margin: 0 0 0 -400px;
}

body#welcomePage section#messageList {
  position: absolute;
  width: 600px;
  background-color: red;
}

body#welcomePage section#friendList {
  position: absolute;
  width: 200px;
  right: 0;
  background-color: yellow;
}
```

EN PRATIQUE Des couleurs pour mieux se repérer

Notez que nous avons ajouté une couleur de fond très voyante à nos deux sections. Rassurez-vous, ce n'est que temporaire ! Cela aidera à repérer facilement les sections et à voir où elles se positionnent exactement.

Le rendu graphique est le suivant.

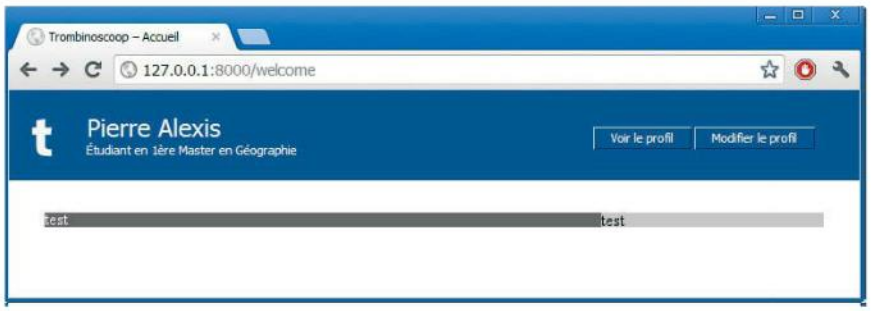


Figure 11–5
Positionnement de la liste des messages et de la liste des amis

Liste des messages

Récupération des messages de la liste

Nous pouvons maintenant remplir la section contenant la liste des messages. Celle-ci commence par un formulaire qui permet d'envoyer un nouveau message. À la suite de celui-ci, on retrouve la liste des messages de l'utilisateur connecté. Nous ajoutons donc le code HTML suivant :

EXEMPLE 11.12 Trombinoscoop. Liste des messages

```
<section id="messageList">
  <form action="welcome" method="get" class="inlineForm">
    <label for="newMessageInput">Publier un message :</label>
    <input type="text" name="newMessage" id="newMessageInput" />
    <input type="submit" value="Publier" />
  </form>
  <ul>
    {% for message in friendMessages %}
    <li>
      <p>
        <a href="???">{{ message.auteur.prenom }}
        {{ message.auteur.nom }}</a> dit :
      </p>
      <p>{{ message.contenu }}</p>
    </li>
    {% endfor %}
  </ul>
</section>
```

Le formulaire est assez standard : un champ texte et un bouton de soumission. Nous avons ajouté une classe au formulaire. Cela nous sera utile pour le style et la syntaxe propre aux CSS.

Pour imprimer la liste des messages des amis, on boucle sur une variable `friendMessages`. Il est clair que cette variable devra être initialisée dans la vue, contenir tous les messages des amis de l'utilisateur connecté et être transmise au template.

EN PRATIQUE Pourquoi ne pas utiliser logged_user déjà présente dans le template ?

Tout simplement car récupérer l'ensemble des messages des amis de l'utilisateur authentifié demande une requête trop compliquée qu'il est impossible d'exprimer avec Django seul. Ce n'est par ailleurs pas du tout le rôle des templates d'effectuer des traitements aussi complexes. On va donc effectuer cette requête dans la vue, en présence des méthodes de filtrage du modèle.

La vue sera modifiée comme suit de sorte à initialiser la liste des messages :

EXEMPLE 11.13 Trombinoscoop. Récupération des messages dans la vue

```
from Trombinoscoop.models import Message

def welcome(request):
  logged_user = get_logged_user_from_request(request)
  if not logged_user is None:

    friendMessages = Message.objects.filter
       (auteur__amis=logged_user).order_by('-date_de_publication')

    return render_to_response('welcome.html',
                              {'logged_user': logged_user,
                               'friendMessages': friendMessages})
  else:
    return HttpResponseRedirect('/login')
```

Une ligne de code nous suffit à récupérer les messages voulus. Pour récupérer l'ensemble des messages des amis de l'utilisateur authentifié, on part de l'objet `Message`. On y applique un filtre assez simple qui signifie « tous les messages dont un des amis de l'auteur du message est l'utilisateur authentifié ». Le résultat retourné par le filtre est ensuite trié par ordre chronologique inverse (le message le plus récent doit apparaître en premier) et envoyé au template.

Ce filtre sera transformé par Django en une requête SQL plutôt complexe et efficace, que voici :

EXEMPLE 11.14 Trombinoscoop. SQL produit par Django

```
'SELECT "Trombinoscoop_message"."id", "Trombinoscoop_message"."auteur_id",
"Trombinoscoop_message"."contenu", "Trombinoscoop_message"."date_de_publication"
FROM "Trombinoscoop_message" INNER JOIN "Trombinoscoop_personne"
ON ("Trombinoscoop_message"."auteur_id" = "Trombinoscoop_personne"."id")
INNER JOIN "Trombinoscoop_personne_amis" ON ("Trombinoscoop_personne"."id" =
"Trombinoscoop_personne_amis"."from_personne_id")
WHERE "Trombinoscoop_personne_amis"."to_personne_id" = 2
ORDER BY "Trombinoscoop_message"."date_de_publication" DESC LIMIT 21
```

Si nous avions dû écrire cette requête nous-mêmes, il est plus que probable qu'après une très longue réflexion, nous serions arrivés exactement au même résultat ; c'est la plus simple et la plus efficace pour récupérer les messages désirés.

On voit ici encore toute la puissance du framework Django (on ne s'en lasse pas !), capable de transformer du code Python simple en instructions SQL complexes et qui ne perdent rien en performance.

La vue étant mise à jour, nous pouvons lancer l'application. Le résultat est le suivant :

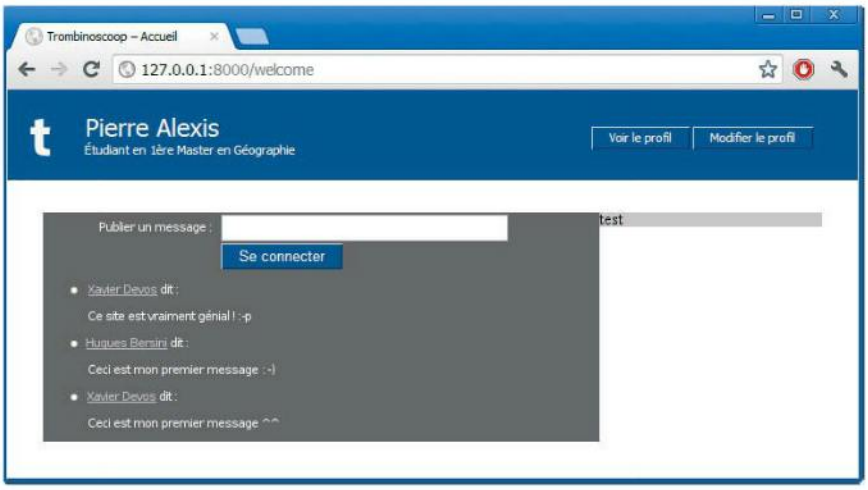


Figure 11–6
Première version de la liste des messages

Présentation de la liste des messages

L'aspect de notre formulaire et de la liste n'est pas *exactement* celui attendu. Un petit peu de CSS va régler tout cela :

EXEMPLE 11.15 Trombinoscoop. Présentation de la liste des messages

```
form.inlineForm label {
  display: inline;
  float: none;
  padding: 0;
}

form.inlineForm input[type="text"] {
  display: inline-block;
  width: 150px;
}

form.inlineForm input[type="submit"] {
  margin: 0;
}

section#messageList ul {
  margin: 20px 0 0 0;
  list-style: none;
  padding: 0;
}
```

```
section#messageList ul li {
  margin-bottom: 10px;
}

section#messageList ul li p {
  margin: 8px 0 0 0;
}
```

Nous avons créé une nouvelle classe de formulaire, `inlineForm`, pour laquelle tous les champs doivent être placés sur une seule ligne. Cette classe annule les styles génériques que nous avions définis pour tous les formulaires basés sur un affichage de type bloc.

Les règles CSS qui suivent modifient légèrement l'aspect de la liste, notamment en supprimant la puce. Notre page d'accueil ressemble maintenant à ceci :

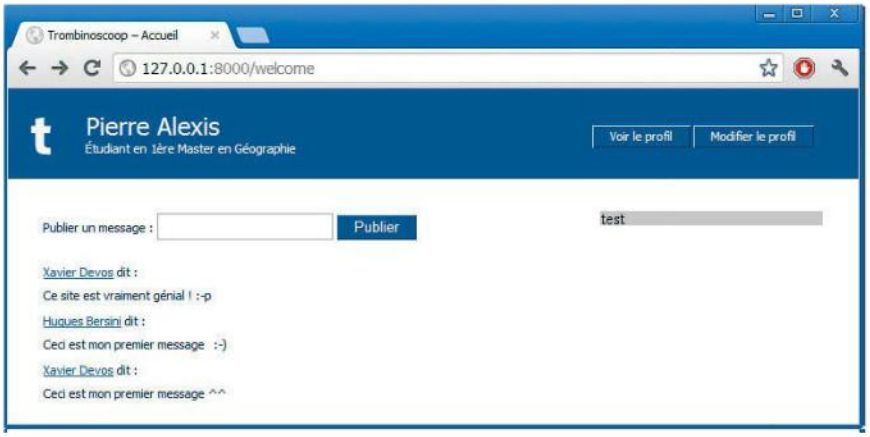


Figure 11–7
Version définitive de la liste des messages

Liste des amis

Il nous reste à imprimer la liste des amis et à la formater. Dans le template, nous ajoutons le code suivant (le mot-clé `all` appliqué à l'attribut `amis` sert à récupérer tous les amis de l'utilisateur authentifié) :

EXEMPLE 11.16 Trombinoscoop. Ajout de la liste des amis

```
<section id="friendList">
  <p class="title">Mes amis</p>
  <p><a href="???" class="buttonLink">Ajouter</a></p>
  <ul>
  {% for ami in logged_user.amis.all %}
    <li><a href="???">{{ ami.prenom }} {{ ami.nom }}</a></li>
  {% endfor %}
  </ul>
</section>
```

Nous modifions également nos CSS pour améliorer le rendu de cette liste :

EXEMPLE 11.17 Trombinoscoop. Formatage de la liste d'amis

```
body#welcomePage section#friendList {
  position: absolute;
  width: 180px;
  right: 0;
  padding: 10px;
  background-color: #DCE0ED;
}

section#friendList p.title {
  font-size: 22px;
  color: #3b5998;
  border-bottom: 1px solid #3b5998;
  margin: 0 0 20px 0;
}

section#friendList ul {
  margin: 20px 0 0 0;
  list-style: none;
  padding: 0;
}

section#friendList ul li {
  margin: 0 0 5px 0;
}
```

Et voilà, notre liste est terminée ! Et avec elle, c'est tout l'aspect de la page d'accueil qui est achevé et en de bonnes mains.

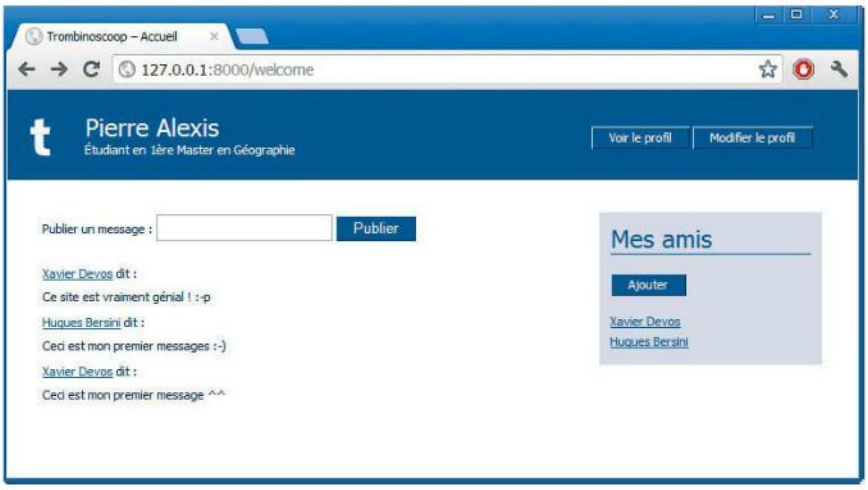


Figure 11–8
Rendu final de la page d'accueil

Pas mal hein ! Si l'apparence est définitive, il nous reste cependant une dernière fonctionnalité à implémenter.

Publication d'un message à destination de ses amis

Le formulaire a déjà été créé en HTML. À sa soumission, il est configuré pour « auto-appeler » la page `welcome`. Un seul champ a été défini, nommé `newMessage`.

Le code à ajouter, du côté de la vue, est donc très simple. Nous le mettons en évidence :

EXEMPLE 11.18 Trombinoscoop. Gestion de la publication d'un nouveau message

```
def welcome(request):
  logged_user = get_logged_user_from_request(request)
  if not logged_user is None:
    if 'newMessage' in request.GET and request.GET['newMessage'] != '':
      newMessage = Message(auteur=logged_user,
                           contenu=request.GET['newMessage'],
                           date_de_publication = datetime.date.today())
      newMessage.save()

    friendMessages = Message.objects.filter(auteur__amis=logged_user)
                            .order_by('-date_de_publication')

    return render_to_response('welcome.html',
                              {'logged_user': logged_user,
                               'friendMessages': friendMessages})
  else:
    return HttpResponseRedirect('/login')
```

On vérifie tout simplement si un nouveau message est passé en paramètre. Si tel est le cas, on crée un nouvel objet `Message` en initialisant chacun de ses champs pour ensuite le sauver dans la base de données.

Cette fois, notre page d'accueil est terminée ! Nous pouvons la définir comme page par défaut de notre site (en d'autres mots, celle qui s'affiche lorsque l'utilisateur n'entre pas de nom de page dans l'URL), en modifiant le fichier `urls.py` comme suit :

EN PRATIQUE Page par défaut de Trombinoscoop

Jusqu'à maintenant, c'est la page de login qui bénéficiait de ce privilège, mais ce n'est plus utile. Mettre la page d'accueil par défaut présente l'avantage que l'utilisateur déjà authentifié arrive directement sur cette page ; s'il n'est pas authentifié, il sera de toute façon redirigé vers la page de login.

EXEMPLE 11.19 Trombinoscoop. La page d'accueil devient page par défaut

```
urlpatterns = patterns('',
  ('^$', welcome), # au lieu de login
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^admin/', include(admin.site.urls))
)
```

Ai-je bien compris ?

- À quoi servent les sessions et les cookies ?
- Comment un internaute malveillant pourrait-il « voler » la session d'un autre ?
- Où Django sauve-t-il les données de la session ?

12

En finir avec Trombinoscoop

Finalisons notre projet de développement web en réalisant les derniers wireframes. Nous appliquerons pour cela tout ce que nous avons appris dans les précédents chapitres.

SOMMAIRE

- Finition de Trombinoscoop

Aux chapitres précédents, nous avons étudié les principales composantes du framework et leur étude a été l'occasion de réaliser les premières pages du site Trombinoscoop. Ici, nous allons réaliser les pages manquantes. Rassurez-vous, vous avez déjà vu tous les outils dont nous avons besoin.

Voici un petit récapitulatif de ce que nous avons déjà terminé :

- la page d'authentification ;
- la page de création de compte ;
- la page d'accueil.

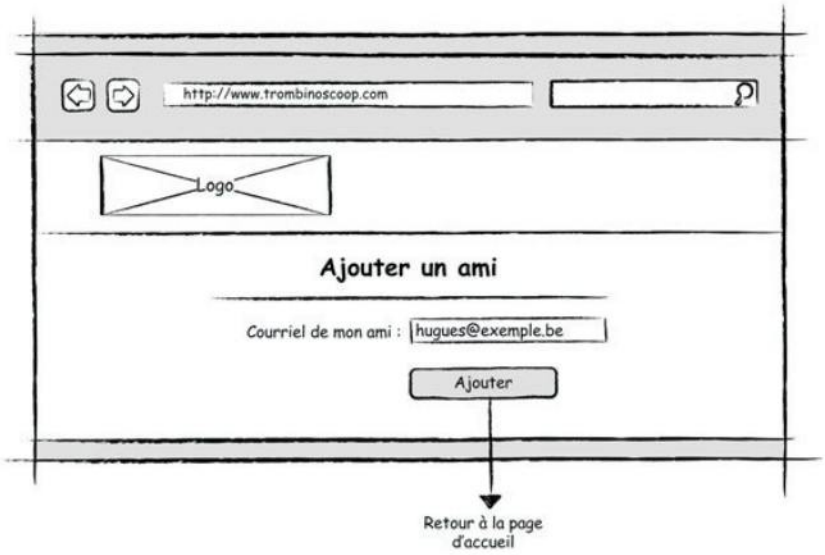
Il nous reste donc à réaliser :

- la page d'ajout d'un ami ;
- la page de visualisation d'un profil ;
- la page de modification d'un compte.

La page d'ajout d'un ami

Le wireframe de cette page est le suivant :

Figure 12–1
Page d'ajout d'un ami



Ajout d'un formulaire dans forms.py

Nous allons utiliser un formulaire Django issu de la bibliothèque `forms` (voir chapitre 10). C'est dans le fichier `forms.py` que nous ajoutons le code suivant :

EXEMPLE 12.1 Trombinoscoop. Définition du formulaire d'ajout de contact

```
class AddFriendForm(forms.Form):
  email = forms.EmailField(label='Courriel :')
  def clean(self):
    cleaned_data = super(AddFriendForm, self).clean()
    email = cleaned_data.get("email")

    # Vérifie que le champ est valide
    if email:
      result = Personne.objects.filter(courriel=email)
      if len(result) != 1:
        raise forms.ValidationError("Adresse de courriel erronée.")

    return cleaned_data
```

Création de la vue add_friend

EXEMPLE 12.2 Trombinoscoop. Ajout du formulaire dans la vue

```
from forms import AddFriendForm

def add_friend(request):
  logged_user = get_logged_user_from_request(request)
  if logged_user:
    # Teste si le formulaire a été envoyé
    if len(request.GET) > 0:
      form = AddFriendForm(request.GET)
      if form.is_valid():
        new_friend_email = form.cleaned_data['email']
        newFriend = Personne.objects.get(courriel=new_friend_email)
        logged_user.amis.add(newFriend)
        logged_user.save()
        return HttpResponseRedirect('/welcome')
      else:
        return render_to_response ('addFriend.html', {'form': form})
    # Le formulaire n'a pas été envoyé
    else:
      form = AddFriendForm()
      return render_to_response ('addFriend.html', {'form': form})
  else:
    return HttpResponseRedirect('/login')
```

Création du template add_friend.html

EXEMPLE 12.3 Trombinoscoop. Contenu du template add_friend.html

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

{% block title %}Ajout d'un ami{% endblock %}

{% block bodyId %}addUserPage{% endblock %}

{% block content %}
<h1>Ajout d'un ami</h1>
<form action="addFriend" method="get">
  {{ form.as_p }}
  <p>
    <input type="submit" value="Ajouter" />
  </p>
</form>
{% endblock %}
```

Ajout d'une URL dans urls.py

EXEMPLE 12.4 Trombinoscoop. Ajout d'une URL pour la page d'ajout d'un ami

```
from views import add_friend

urlpatterns = patterns('',
  ('^$', welcome), # au lieu de login
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^addFriend$', add_friend),
  ('^admin/', include(admin.site.urls))
)
```

Ajout du lien dans la page d'accueil

Pour terminer, dans le template de la page d'accueil, on remplace les points d'interrogation du lien vers la page d'ajout par l'URL de la page :

EXEMPLE 12.5 Trombinoscoop. Correction du lien d'ajout d'un ami

```
<p><a href="addFriend" class="buttonLink">Ajouter</a></p>
```

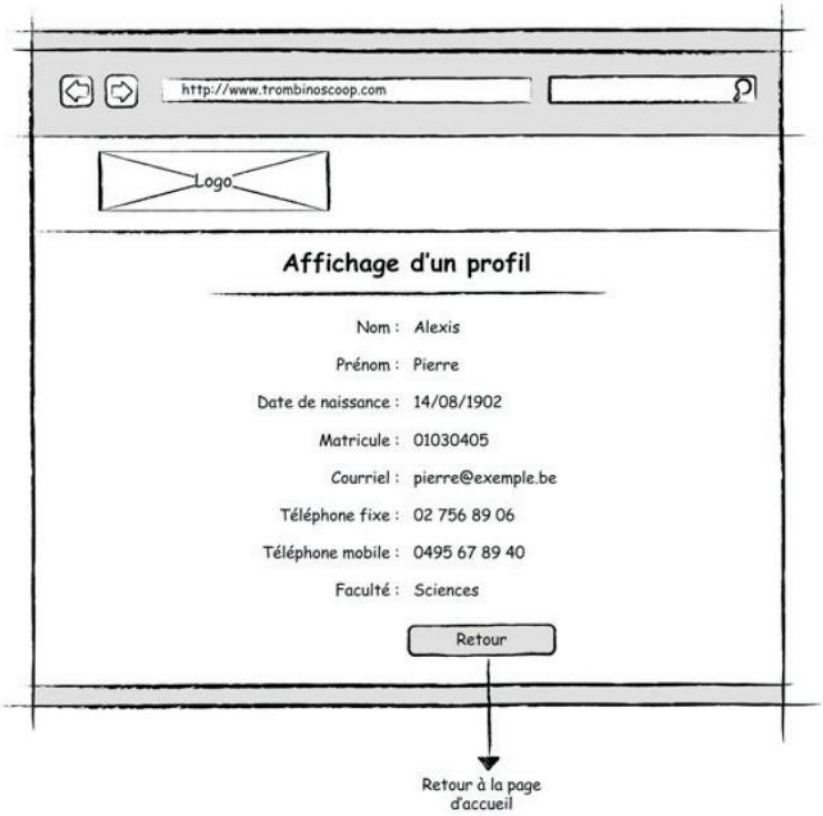
Et voilà ! La page d'ajout d'un ami est terminée et voici ce que cela donne :

Figure 12–2
Page d'ajout d'un ami

La page de visualisation d'un profil

Souvenez-vous, le wireframe que nous avions créé pour cette page est le suivant :

Figure 12–3
Page d'affichage d'un profil



Cette page est assez triviale : il suffit d'ajouter un template qui reçoit en paramètre l'employé ou l'étudiant à afficher et d'afficher l'ensemble des champs du modèle.

Création du template show_profile.html

EXEMPLE 12.6 Trombinoscoop. Contenu du template show_profile.html

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

{% block title %}Affichage d'un profil{% endblock %}

{% block bodyId %}showProfilePage{% endblock %}

{% block content %}
<h1>Affichage d'un profil</h1>
<dl class="fieldList">
  <dt>Nom :</dt>
    <dd>{{ user_to_show.nom }}</dd>
  <dt>Prénom :</dt>
    <dd>{{ user_to_show.prenom }}</dd>
  <dt>Date de naissance :</dt>
    <dd>{{ user_to_show.date_de_naissance }}</dd>
  <dt>Matricule :</dt>
    <dd>{{ user_to_show.matricule }}</dd>
  <dt>Courriel :</dt>
    <dd>{{ user_to_show.courriel }}</dd>
  <dt>Tél. fixe :</dt>
    <dd>{{ user_to_show.tel_fixe }}</dd>
  <dt>Tél. mobile :</dt>
    <dd>{{ user_to_show.tel_mobile }}</dd>
  <dt>Faculté :</dt>
    <dd>{{ user_to_show.faculte.nom }}</dd>
{% if user_to_show.type_de_personne == "student" %}
  <dt>Cursus :</dt>
    <dd>{{ user_to_show.cursus.intitule }}</dd>
  <dt>Année :</dt>
    <dd>{{ user_to_show.annee }}</dd>
{% elif user_to_show.type_de_personne == "employee" %}
  <dt>Bureau :</dt>
    <dd>{{ user_to_show.bureau }}</dd>
  <dt>Campus :</dt>
    <dd>{{ user_to_show.campus.nom }}</dd>
  <dt>Fonction :</dt>
    <dd>{{ user_to_show.fonction.intitule }}</dd>
{% endif %}
</dl>
<p id="showProfileNavigationButtons">
  <a href="welcome" class="buttonLink">Retour</a>
</p>
{% endblock %}
```

Le contenu est un peu verbeux, mais très simple à comprendre. L'affichage de toutes les informations de l'utilisateur se base sur l'instruction HTML « liste de définitions » (balise `<dl>`). Celle-ci fait correspondre à un élément sa définition, ce qui correspond tout à fait à l'usage que l'on veut en faire. L'aspect de cette liste sera amélioré dans les CSS.

Création de la vue show_profile

EXEMPLE 12.7 Trombinoscoop. Ajout de la vue show_profile

```
def show_profile(request):
  logged_user = get_logged_user_from_request(request)
  if logged_user: (1)
    # Teste si le paramètre attendu est bien passé
    if 'userToShow' in request.GET and request.GET['userToShow'] != '': (2)
      results = Personne.objects.filter(id=request.GET['userToShow']) (3)
      if len(results) == 1:
        if Etudiant.objects.filter(id=request.GET['userToShow']):
          user_to_show = Etudiant.objects.get(id=request.GET['userToShow']) (5)
        else:
          user_to_show = Employe.objects.get(id=request.GET['userToShow']) (5)
        return render_to_response ('show_profile.html',
                                   ➥ {'user_to_show': user_to_show}) (6)
      else:
        return render_to_response ('show_profile.html',
                                   ➥ {'user_to_show': logged_user}) (4)
    # Le paramètre n'a pas été trouvé
    else:
      return render_to_response ('show_profile.html',
                                   ➥ {'user_to_show': logged_user})
  else:
    return HttpResponseRedirect('/login')
```

On vérifie d'abord qu'un utilisateur est authentifié (1), comme sur toutes nos pages « privées ». Ensuite, on vérifie que l'`id` de la personne dont on veut voir le profil est bien passé en paramètre (2) et on recherche l'enregistrement correspondant dans la base de données (3). En cas d'échec, on affiche le profil de l'utilisateur authentifié à la place (4). Ainsi, il n'y a pas de message d'erreur à gérer dans le template.

En revanche, si tout se passe pour le mieux, on cherche si la personne à afficher est un étudiant ou un employé (5) et on extrait de la base de données le bon objet. On appelle ensuite le template en passant en paramètre l'étudiant ou l'employé (6).

Ajout de l'URL dans urls.py

EXEMPLE 12.8 Trombinoscoop. Ajout de l'URL

```
from views import show_profile

urlpatterns = patterns('',
  ('^$', welcome), # au lieu de login
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^addFriend$', add_friend),
  ('^showProfile$', show_profile),
  ('^admin/', include(admin.site.urls))
)
```

Amélioration de la présentation dans style.css

EXEMPLE 12.9 Trombinoscoop. Modification de l'aspect de certains éléments

```
dl.fieldList dt {
  display: block;
  width: 250px;
  float: left;
  text-align: right;
  padding: 0 10px 0 0;
  font-weight: bold;
}

dl.fieldList dd {
  display: block;
  width: 450px;
  padding: 0px;
  margin: 0 0 10px 0;
}

p#showProfileNavigationButtons {
  margin: 20px 0 0 260px;
}
```

Ajout des liens dans la page d'accueil

Dernier point, mais non des moindres, n'oubliez pas de corriger sur la page d'accueil tous les liens qui devraient pointer vers la nouvelle page d'affichage d'un profil. Il s'agit de tous ceux qui entourent un nom (dans la liste des amis et dans la liste des messages) et qui permettent à l'utilisateur, dès qu'il voit le nom d'une personne, de cliquer dessus et de voir son profil.

Il s'agit donc de modifier les lignes suivantes dans `welcome.html` :

EXEMPLE 12.10 Trombinoscoop. Lignes modifiées dans welcome.html

```
<p id="profileLinks">
  <a href="showProfile?userToShow={{ logged_user.id }}" class="buttonLink">Voir
le profil</a>
  <a href="???" class="buttonLink">Modifier le profil</a>
</p>

<li>
  <p><a href="showProfile?userToShow={{ message.auteur.id }}">{{
message.auteur.prenom }} {{ message.auteur.nom }}</a> dit :</p>
  <p>{{ message.contenu }}</p>
</li>

<ul>
{% for ami in logged_user.amis.all %}
  <li><a href="showProfile?userToShow={{ ami.id }}">{{ ami.prenom }} {{ ami.nom
}}</a></li>
{% endfor %}
</ul>
```

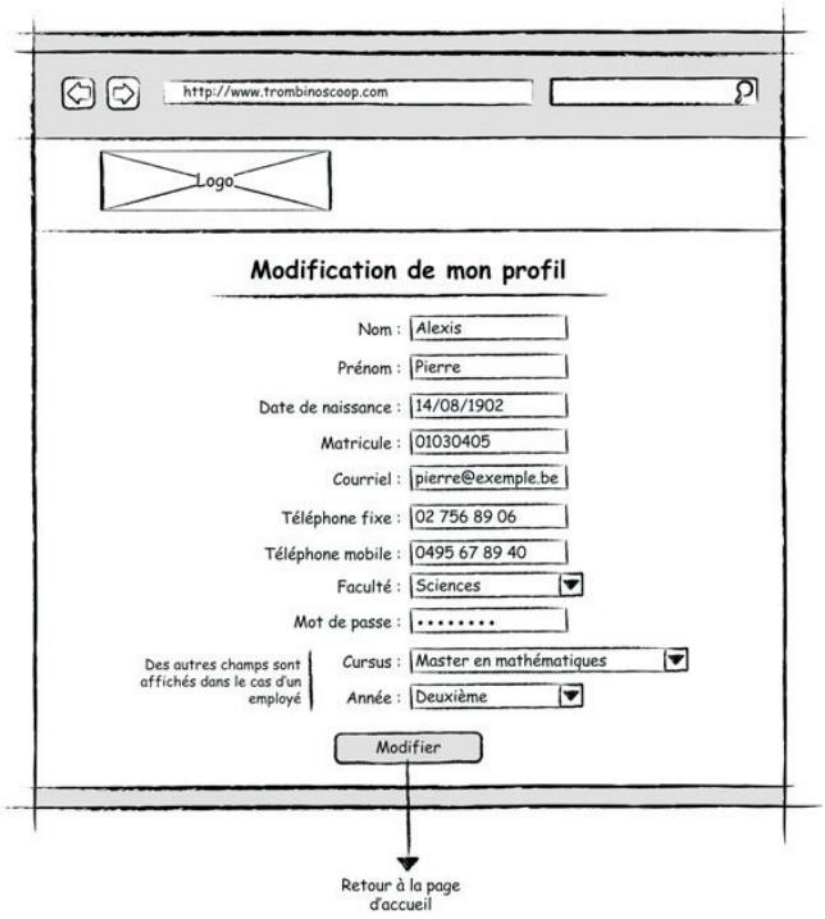
Et voilà, notre page exposant le profil d'un utilisateur de notre site est prête. Le rendu en est le suivant :

Figure 12–4
Page d'affichage d'un profil

La page de modification d'un profil

Il ne nous reste plus que la page de modification d'un profil, dont voici le wireframe :

Figure 12–5
Wireframe de la page de modification du profil



Nous ne pourrons malheureusement pas récupérer tel quel le template de création de compte car, dans sa version la plus aboutie, celui-ci contenait deux formulaires : le premier pour créer un étudiant, le second pour créer un employé. Or, un seul nous suffit pour modifier un profil. Dans notre projet, nous allons supposer qu'on ne peut pas changer de type de profil : un étudiant restera un étudiant à vie, et un employé restera un employé à vie…

Création du template modify_profile.html

Ce template fort simple ressemble à ce que nous avons déjà fait pour la page de création d'un compte.

EXEMPLE 12.11 Trombinoscoop. Contenu de modify_profile.html

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

{% block title %}Modification de mon profil{% endblock %}

{% block bodyId %}modifyProfilePage{% endblock %}

{% block content %}
<h1>Modification de mon profil</h1>
<form action="modifyProfile" method="get">
  {{ form.as_p }}
  <p>
    <input type="submit" value="Modifier" />
  </p>
</form>
{% endblock %}
```

Création de la vue modify_profile

EXEMPLE 12.12 Trombinoscoop. Création de la vue modify_profile

```
def modify_profile(request):
  logged_user = get_logged_user_from_request(request)
  if logged_user:
    if len(request.GET) > 0:
      if type(logged_user) == Etudiant:
        form = StudentProfileForm(request.GET, instance=logged_user)
      else:
        form = EmployeeProfileForm(request.GET, instance=logged_user)
      if form.is_valid():
        form.save(commit=True)
        return HttpResponseRedirect('/welcome')
      else:
        return render_to_response ('modify_profile.html', {'form': form})
    else:
      if type(logged_user) == Etudiant:
        form = StudentProfileForm(instance=logged_user)
      else:
        form = EmployeeProfileForm(instance=logged_user)
      return render_to_response ('modify_profile.html', {'form': form})
  else:
    return HttpResponseRedirect('/login')
```

La structure de cette vue est assez standard. Petite nouveauté cependant : toutes les initialisations des `ModelForm` se font en passant en paramètre la personne dont on désire modifier le profil (en l'occurrence, l'utilisateur authentifié). Si on omet de pré-

ciser ce paramètre, les `ModelForm` ne sauront pas qu'il faut modifier un objet existant de la base de données et vont en ajouter un nouveau, créant ainsi un doublon altéré.

Ajout de l'URL dans urls.py

EXEMPLE 12.13 Trombinoscoop. Ajout de l'URL

```
from views import modify_profile

urlpatterns = patterns('',
  ('^$', welcome), # au lieu de login
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^addFriend$', add_friend),
  ('^showProfile$', show_profile),
  ('^modifyProfile$', modify_profile),
  ('^admin/', include(admin.site.urls))
)
```

Ajout des liens dans la page d'accueil

EXEMPLE 12.14 Trombinoscoop. Modification du lien pointant vers la page

```
<p id="profileLinks">
  <a href="showProfile?userToShow={{ logged_user.id }}" class="buttonLink">Voir
le profil</a>
  <a href="modifyProfile" class="buttonLink">Modifier le profil</a>
</p>
```

Et voilà, la dernière page de notre site est terminée !

Figure 12–6
Page de modification du profil

Ai-je bien compris ?

- Dans le fichier `urls.py`, quelle est la différence entre `addFriend` et `add_friend` ?
- Comment procède-t-on pour que le même template puisse être utilisé pour l'affichage des profils étudiants et des profils employés ?
- Comment effectue-t-on cette même différenciation dans la page de modification de profil ?

13

Des sites web encore plus dynamiques avec Ajax

Ce chapitre va se pencher sur la technologie Ajax qui permet au client de solliciter le serveur sans quitter la page actuellement active de son côté. Le résultat du serveur viendra juste s'insérer dans la page client courante sans qu'il faille entièrement la remplacer par une autre. C'est du JavaScript s'exécutant côté client qui va se charger de réaliser la connexion avec le serveur, la récupération des informations et leur insertion dans la page client courante. L'interaction sera dite asynchrone car, lorsque le client envoie la requête, il n'est pas contraint d'attendre la réponse du serveur pour poursuivre sa tâche.

SOMMAIRE

- Présentation de la technologie Ajax
- Discussion sur la répartition de la partie « programme » du site web entre le client et le serveur
- Mise en place de solutions Ajax pour enrichir le dynamisme de Trombinoscoop
- Point final de Trombisnoscoop

Trombinoscoop est un site web dynamique : le contenu de la plupart des pages est construit par les utilisateurs du site. Messages publiés, amis ajoutés, profils modifiés, les pages se construisent à mesure que le site se trouve exploité par les internautes.

Nous ne sommes cependant pas encore parvenus à un niveau de dynamisme équivalant à celui des applications dites « lourdes » (par opposition, sans surprise, à celles dites « légères »). En effet, d'une requête HTTP à l'autre, le contenu des pages reste totalement inchangé tant que l'utilisateur ne soumet pas de formulaire ou ne clique sur un lien hypertexte.

Prenons l'exemple de la validation d'un formulaire. Dans Trombinoscoop, la validation des champs se fait côté serveur, une fois le formulaire complètement rempli et soumis. Or, il serait intéressant de pouvoir avertir l'utilisateur beaucoup plus tôt qu'il a introduit une donnée erronée, par exemple lorsqu'il a fini d'encoder un champ et qu'il se prépare à encoder le suivant.

Exemple de l'interactivité attendue entre client et serveur

Supposons que vous souhaitiez réserver une chambre d'hôtel sur un site web. Vous découvrez sur une page dédiée à cette réservation un formulaire, sur lequel vous pouvez saisir le nombre de chambres souhaitées, pour combien de personnes, et les dates d'arrivée et de départ. Dans la zone prévue pour les deux dates, un calendrier est à votre disposition et il suffit de sélectionner et cocher les deux jours. Cette page en question n'a rien de statique car elle permet à l'internaute de dérouler ce calendrier et de sélectionner les dates en question. Seule l'exécution d'un programme « inséré » dans la page rend cette interaction possible. Le nombre de chambres qu'il est possible de réserver est restreint par un menu déroulant (fonction des chambres qui restent libres dans l'hôtel). Lorsqu'il faudra sortir la carte bancaire pour payer, de nombreux raccourcis vous seront proposés afin d'aller le plus rapidement possible à l'essentiel. Si le numéro rentré est farfelu ou si vous n'en indiquez aucun, ce même programme en arrière-plan vous le fera très vite remarquer.

On atteint très vite les limites de ce dont le client peut s'acquitter seul. En effet, on conçoit aisément que les informations concernant la disponibilité des chambres se trouvent dans un serveur affilié à l'hôtel et, la plupart du temps, stockées dans une base de données relationnelles. Sur sollicitation du client, c'est ce serveur qui prend le relais, consulte la base de données pour vérifier la disponibilité des chambres en question. Il renvoie ensuite côté client, dans un fichier HTML qu'il engendre sur le fil, les chambres disponibles, leurs prix et, pourquoi pas, des photos. Il joint aussi à côté de chacune des chambres, une case à cocher (ainsi que les instructions JavaScript qui réagiront à cette action) pour effectuer la réservation. C'est à nouveau au client

de réagir s'il souhaite effectuer cette réservation ; il ne verra ce souhait finalement exaucé par le serveur qu'à la suite d'une requête SQL de mise à jour de la base de données, en y ajoutant une réservation et en rendant la chambre réservée indisponible pendant cette période. Ainsi, un autre internaute désirant réserver cette même chambre au même moment verrait l'information d'indisponibilité s'intercaler dans sa page sans avoir à la rafraîchir.

EN PRATIQUE Place d'Ajax parmi les acteurs des interactions client-serveur

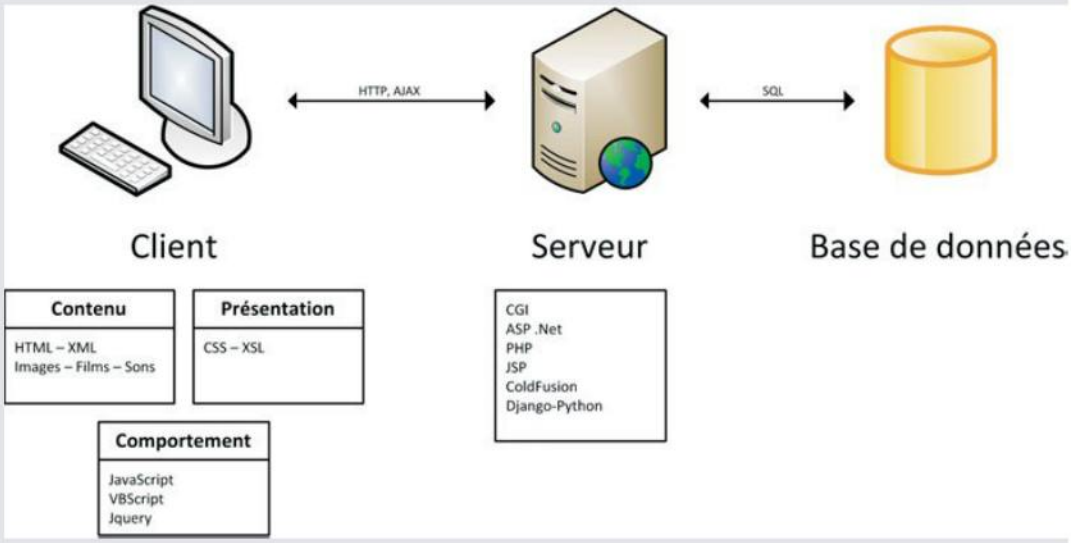


Figure 13–1 Positionnement des différentes technologies web

La figure, non exhaustive, présente quelques techniques parmi les plus répandues et populaires auprès des informaticiens. N'importe quel site web est le résultat d'une interaction client-serveur. Le client fait la demande de la page via son navigateur ; il précise l'URL du site et la page souhaitée, qu'elle soit statique (HTML) ou dynamique (PHP, ASP, JSP…), et le serveur la lui envoie.

En général, du code s'exécutant côté client sera à même de produire une petite animation, faciliter la prise d'informations, les recevoir et vérifier leur cohérence et leur complétude.

Si programme il y a, c'est qu'il y a langage de programmation. JavaScript est le langage le plus usité pour la conception de codes exécutables par le navigateur **côté client**. En effet, tous les navigateurs sont dotés d'un « interpréteur » d'instructions JavaScript en plus d'afficher la page selon les instructions HTML, exécutera quelques commandes portant sur la manipulation de ces pages par l'internaute.

Si programme il y a, c'est qu'il y a langage de programmation… mais cette fois du **côté serveur**. Bien évidemment, c'est souvent de ce côté que les choses sérieuses se passent, car c'est vraiment lui qui se trouve au cœur de l'information. C'est pour cela que les acteurs importants du monde logiciel proposent pour la plupart des solutions innovantes : PHP (un grand classique de la programmation web), Java (technologies JSP), ASP.Net, Python (par l'entremise du framework Django). Ainsi, les programmeurs peuvent aisément et efficacement mêler le HTML, la programmation et l'interaction avec une base de données à l'aide ou non du langage SQL.

La plupart du temps, la sollicitation du serveur se conclut par la mise à disposition d'une nouvelle page HTML qui vient se substituer à celle d'où provient cette sollicitation. Or, souvent, ce n'est pas toute la page qu'il faudrait remplacer par la nouvelle, mais juste une partie devant simplement être mise à jour suite aux opérations effectuées par le serveur. On comprend que l'information circulant entre le serveur et le client peut s'en trouver considérablement allégée.
Plus récemment, et de manière à rendre en effet l'interaction client/serveur plus efficace, la technologie Ajax (*Asynchronous JavaScript and XML*, que l'on doit à Google) permet au client de solliciter le serveur sans quitter la page active. Le résultat du serveur viendra juste s'insérer dans cette dernière sans qu'il faille entièrement la remplacer par une nouvelle version. C'est du JavaScript s'exécutant côté client ; il se charge de réaliser la connexion avec le serveur pour récupérer des informations et les insérer dans la page client courante. On parle d'**interaction asynchrone** car, lorsque le client envoie la requête, il n'est pas contraint d'attendre la réponse du serveur pour poursuivre sa tâche. C'est seulement quand le serveur aura terminé l'exécution de sa requête et qu'il renverra le résultat au client, que ce dernier s'en trouvera informé et agira en conséquence, par exemple en affichant ce résultat sur sa page.

CULTURE

Une des premières grandes exploitations d'Ajax fut « Google Suggest », la complétion automatique des mots-clés que les internautes commencent à écrire dans le moteur de recherche. Par la suite, Google Maps et Google Earth en firent un très large usage.

Ajax (*Asynchronous JavaScript and XML*), technologie apparue récemment dans le monde web, pallie ce manque de réactivité et ajoute une dose supplémentaire de dynamisme aux pages web. Plus précisément, Ajax envoie des requêtes HTTP au serveur *en coulisses*, c'est-à-dire sans que l'utilisateur n'ait explicitement soumis un formulaire ou cliqué sur un lien.

En ce qui concerne nos formulaires, lorsque l'utilisateur a fini d'encoder un champ, Ajax va discrètement interroger le serveur et lui demander si la valeur du champ est valide. On pourra ainsi immédiatement réagir et renvoyer à l'utilisateur un message d'erreur le cas échéant.

Ce mécanisme n'offre pas uniquement l'avantage d'une réactivité et d'une pro-activité accrue. Ajax apporte également aux applications une expérience utilisateur bien meilleure. En effet, ce dernier effectue bien plus d'actions avant que la page web ne soit rafraîchie (opération qui, même si elle ne prend qu'un dixième de seconde, paraît toujours durer une éternité). De plus, Ajax permet de ne rafraîchir et modifier qu'une petite portion de la page, ce qui, dans certains cas, réduit considérablement le trafic réseau.

DÉFINITION Ajax

Le terme Ajax résume quelques-unes des technologies qui sont utilisées pour mettre en place ce dynamisme. L'acronyme signifie *Asynchronous JavaScript and XML*. En d'autres termes, on va envoyer des requêtes via JavaScript en les formatant, préférablement en XML (c'était le cas à l'origine, ce n'est pas une obligation comme la suite du chapitre va nous le démontrer).

Afin d'étudier l'utilisation de cette technologie, nous allons agrémenter Trombinoscoop de deux fonctionnalités typiquement Ajax :

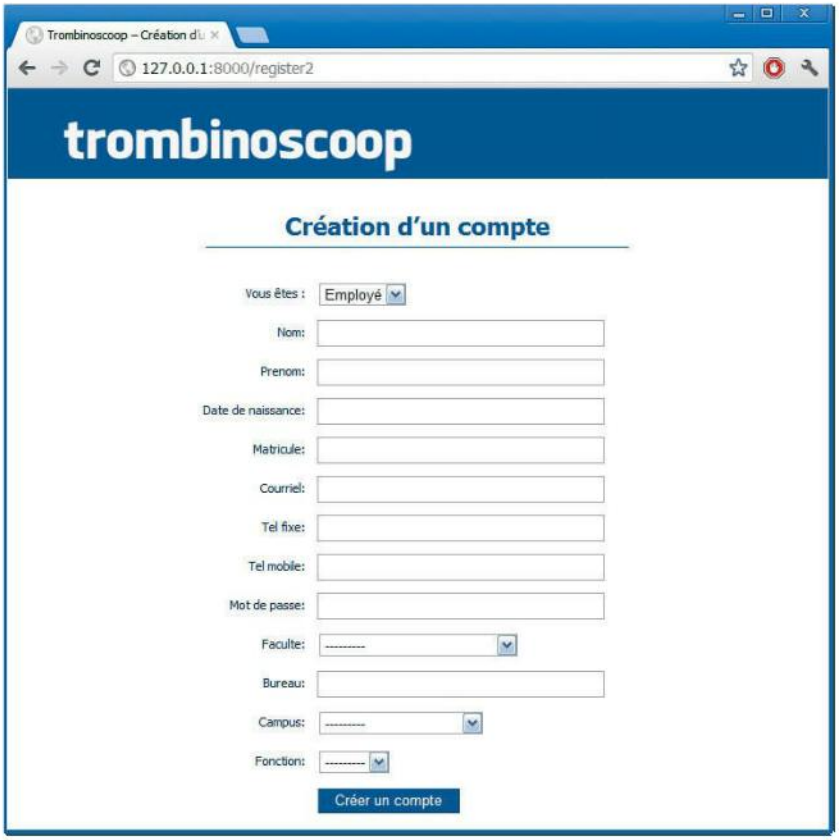
- valider un champ du formulaire de création de compte immédiatement après que l'utilisateur ait fini de le saisir ;
- sur la page d'accueil, rafraîchir la liste des amis sans qu'il faille recharger la page d'accueil en entier.

Validation de la création d'un compte avec Ajax

Les nouveautés se situeront cette fois-ci côté client ; côté serveur, la démarche sera identique à ce que nous avons déjà fait.

Reprenons notre page de création de compte qui, pour l'instant, ressemble à ceci :

Figure 13–2
Page de création de compte



Pour valider « en direct » le champ *Courriel*, dès que l'utilisateur l'a encodé, nous devons ajouter les parties suivantes :

- dans la page HTML, du code JavaScript qui s'active dès que l'utilisateur a terminé d'encoder le champ *courriel*. Il interroge le serveur pour vérifier la validité de ce qui a été encodé et, le cas échéant, afficher un message d'erreur au niveau du champ. Nous allons écrire ces lignes JavaScript en nous servant du framework jQuery vu au chapitre 4.
- au niveau Python, une nouvelle URL et une nouvelle vue afin de traiter la requête envoyée par le JavaScript. Le rôle de la vue est de vérifier que la donnée envoyée par le JavaScript est valide et de renvoyer si nécessaire une erreur.

Ajout d'une URL pour la requête Ajax

EXEMPLE 13.1 Trombinoscoop. Ajout d'une URL pour notre requête Ajax

```
from views import ajax_check_email_field

urlpatterns = patterns('',
  ('^$', welcome), # au lieu de login
  ('^login$', login),
  ('^welcome$', welcome),
  ('^register$', register),
  ('^addFriend$', addFriend),
  ('^showProfile$', showProfile),
  ('^modifyProfile$', modifyProfile),
  ('^ajax/checkEmailField$', ajax_check_email_field),
  ('^admin/', include(admin.site.urls))
)
```

Afin d'indiquer clairement qu'il s'agit d'une requête Ajax, on préfixe l'URL avec `ajax/`.

Vue traitant la requête Ajax

EXEMPLE 13.2 Trombinoscoop. Contenu de la vue ajax_check_email_field

```
from django.http import HttpResponse

def ajax_check_email_field(request):
  HTML_to_return = '' (1)
  if 'value' in request.GET:
    field = forms.EmailField() (2)
    try:
      field.clean(request.GET['value']) (3)
    except exceptions.ValidationError as ve: (4)
      HTML_to_return = '<ul class="errorlist">'
      for message in ve.messages:
```

```
            HTML_to_return += '<li>' + message + '</li>' (5)
        HTML_to_return += '</ul>'

    return HttpResponse(HTML_to_return)
```

Cette vue vérifie qu'une adresse de courriel passée en paramètre est valide. Deux valeurs de retour sont possibles :

- « rien » (un `string` vide) dans le cas où il n'y a pas d'erreur à signaler ;
- une liste HTML d'erreurs, chaque élément de la liste étant un message.

On commence par initialiser la valeur de retour avec le `string` vide indiquant qu'il n'y aucune erreur (1). Ensuite, on utilise la classe `Field` (2) de la bibliothèque `forms` de Django, car elle implémente déjà la validation d'un courriel (il vous faut pour cela importer deux nouvelles bibliothèques : `from django import forms` et `from django.core import exceptions`). Dans une instruction `try (...) except`, on appelle la méthode `clean` de cette classe en lui passant en paramètre la valeur à valider (3). Si la validation ne passe pas, `clean` lève une exception (4). On construit notre liste HTML en parcourant tous les messages d'erreur (5). Enfin, on retourne le tout au navigateur.

> **À LIRE Gestion des exceptions**
>
> La gestion d'exceptions est un mécanisme de programmation classique pour gérer les problèmes. Nous ne le détaillerons pas ici, mais vous le trouverez expliqué dans de nombreux manuels de programmation.

Nous pouvons très bien essayer cette URL de manière tout à fait indépendante en l'introduisant dans la barre d'adresse du navigateur. Voilà ce que cela donne si on passe en paramètre une adresse de courriel erronée (la liste contient un seul message d'erreur) :

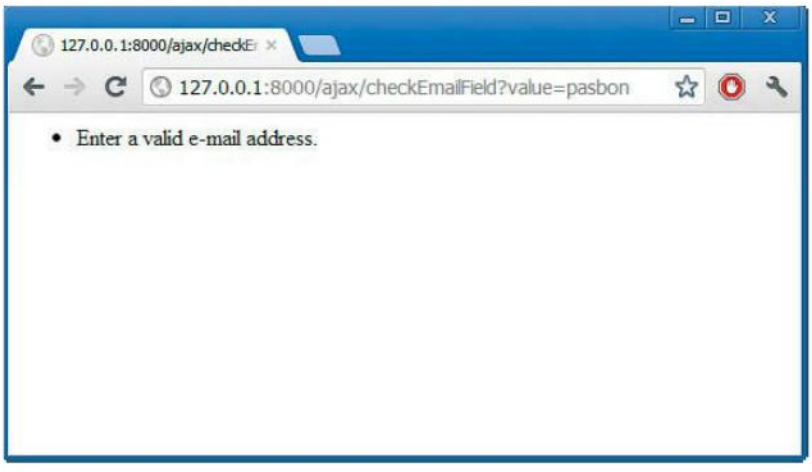


Figure 13–3
Essai de notre nouvelle URL de validation d'adresses de courriel

Ajout du code JavaScript

Installons notre code JavaScript dans le template `user_profile.html`. Nous allons progresser par étapes, afin de bien décomposer et comprendre chaque partie du code.

Détecter que l'utilisateur a bien terminé de remplir le champ courriel

Cela revient à intercepter l'événement `focusout` de l'élément `input` correspondant au champ courriel.

EXEMPLE 13.3 Trombinoscoop. Interception de l'événement focusout

```
{% block content %}

<script type="text/javascript">
  $("input#id_courriel").focusout(checkEmailField); (1)

  function checkEmailField()
  {
    alert('Courriel introduit. On va maintenant le valider.');
  }
</script>
```

Ce code spécifie simplement que lorsque l'événement `focusout` survient sur l'élément `input` d'`id` `id_courriel`, il faut exécuter la fonction `checkEmailField` (1).

> **EN PRATIQUE Des `id` différents pour les champs de courriels des étudiants et des employés**
>
> Dans le formulaire permettant de créer à la fois des comptes étudiants et des comptes employés, remarquez qu'il y a deux `id` différents pour le champ de courriel. Pour connaître les `id` créés par Django, il suffit de regarder les codes sources de la page web sur votre navigateur.

Toutefois, si nous exécutons ce code, nous constatons qu'il ne fonctionne pas. La première ligne est tout simplement exécutée « trop tôt ». Les navigateurs sont un peu nerveux et ont tendance à exécuter le JavaScript dès que possible, même si le document n'est pas encore entièrement chargé. Dès lors, lorsque cette ligne est exécutée, notre champ `input` n'existe pas encore.

Il existe un événement jQuery signalant quand le document est pleinement chargé ; il s'agit de l'événement `ready` de l'objet `document`. Nous allons donc plutôt écrire ce qui suit.

EXEMPLE 13.4 Trombinoscoop. Interception de l'événement focusout dans le document entièrement chargé

```
<script type="text/javascript">
  $(document).ready(function() 2
  {
    $("input#id_courriel").focusout(checkEmailField);
  });

  function checkEmailField()
  {
    alert('Courriel introduit. On va maintenant le valider.');
  }
</script>
```

EN PRATIQUE Astuce pour les fonctions anonymes

Généralement, lorsqu'un événement se produit, on appelle une fonction. C'est le cas avec l'événement `focusout`, pour lequel on appelle la fonction nommée `checkEmailField`.
En 2, plutôt que d'appeler une fonction nommée et définie ailleurs, on insère une fonction anonyme directement en paramètre de la fonction `ready`. Cela évite de devoir définir trop de fonctions qu'il faudrait nommer et n'utiliser qu'une seule fois.
Les fonctions anonymes sont très utilisées par les développeurs JavaScript et jQuery.

À présent, notre code fonctionne : quand on quitte le champ de courriel, un message apparaît.

Figure 13–4
Capture de l'événement focusout

Validation du courriel saisi

Nous pouvons dès lors dans la fonction `checkEmailField` solliciter le serveur et lui demander de vérifier l'adresse de courriel saisie. Nous ajoutons ce code juste après la fonction `checkEmailField`, toujours dans l'élément HTML `<script>`.

EXEMPLE 13.5 Trombinoscoop. Appel Ajax à la fonction de validation

```
function checkEmailField()
{
  $fieldValue = $("input#id_courriel").val(); (1)
  $.ajax({ (2)
    url: '/ajax/checkEmailField', (3)
    data: ({value : $fieldValue}) , (4)
    type: 'GET', 5
    success: function($data, $textStatus, $XMLHttpRequest) { (6)
        if ($data != '')
        {
          alert($data);
        }
      }
    }
  )
}
```

La première ligne est la plus simple à comprendre : il s'agit simplement de récupérer la valeur du champ contenant l'adresse de courriel (1).

Ensuite, la syntaxe se complique. On fait appel à la fonction `ajax` de jQuery (2), qui prend en paramètre un tableau de paires clés/valeurs servant à paramétrer l'appel Ajax. La première valeur à renseigner est l'URL à appeler (3). Ensuite, on spécifie un tableau contenant tous les champs à envoyer en paramètres à l'URL. Dans notre cas, nous n'avons qu'un champ : l'adresse à valider (4). Le troisième paramètre de la requête Ajax est son type : `POST` ou `GET` (5). Enfin, le dernier paramètre, et non des moindres, est la fonction à appeler une fois que la requête Ajax s'est terminée avec succès (6).

À nouveau, nous avons travaillé avec une fonction anonyme. Cette fonction, définie par jQuery, reçoit trois paramètres, dont le premier, `data`, est le plus important. Il contient la réponse du serveur, dans notre cas la liste des éventuels messages d'erreur.

Si des erreurs se présentent, on affiche une boîte de message reprenant ces erreurs. On s'occupera de les insérer dans la page HTML par la suite. Testons le résultat.

Comme nous pouvons le constater, si l'adresse est erronée, une boîte de dialogue apparaît affichant le code HTML de la liste contenant toutes les erreurs.

Figure 13–5
Affichage de la réponse de la requête Ajax

Insérer la liste des erreurs au-dessus du champ « fautif »

EXEMPLE 13.6 Trombinoscoop. Affichage des erreurs retournées par l'appel Ajax

```
function checkEmailField()
{
  $fieldValue = $("input#id_courriel").val();
  $.ajax({
    url: '/ajax/checkEmailField',
    data: ({value : $fieldValue}) ,
    type: 'GET',
    success: function($data, $textStatus, $XMLHttpRequest) {
        if ($data != '')
        {
          $("input#id_courriel").parent().prev('.errorlist').remove();
          $("input#id_courriel").parent().before($data); (1)
        }
      }
    }
  )
}
```

Nous avons ajouté deux lignes : la première pour supprimer une éventuelle liste d'erreurs qui serait déjà présente, la deuxième pour insérer la nouvelle liste d'erreurs fraîchement reçue.

Afin de bien comprendre ces deux instructions, regardons exactement où doit se situer la liste d'erreurs dans le document HTML. Lorsque c'est le serveur qui valide l'ensemble des champs et que des erreurs sont trouvées, voici ce que Django produit :

EXEMPLE 13.7 Trombinoscoop. Erreurs imprimées par Django

```
<ul class="errorlist">
  <li>Enter a valid e-mail address.</li>
</ul>
<p>
  <label for="id_courriel">Courriel:</label>
  <input id="id_courriel" type="text" name="courriel" value="pas bon"
maxlength="75" />
</p>
```

Nous le voyons bien, la liste d'erreurs se trouve juste avant le paragraphe qui contient le champ, au même niveau que le paragraphe. Ainsi, pour insérer la liste, on part de l'élément `input`, on remonte au parent et on l'insère juste avant. C'est exactement ce qu'exprime la ligne 1 du code Ajax.

Pour réaliser la suppression, on part de l'élément `input`, on remonte au parent, on prend l'élément qui précède et on le supprime.

Et voilà ! Notre validation Ajax est terminée. Voici le résultat :

Figure 13–6
Le formulaire est validé en direct.

Ajout d'un ami via Ajax

Voyons maintenant comment ajouter un ami directement sur la page d'accueil par le biais d'une requête asynchrone en arrière-plan.

Pour rappel, notre page d'accueil ressemble à ceci :

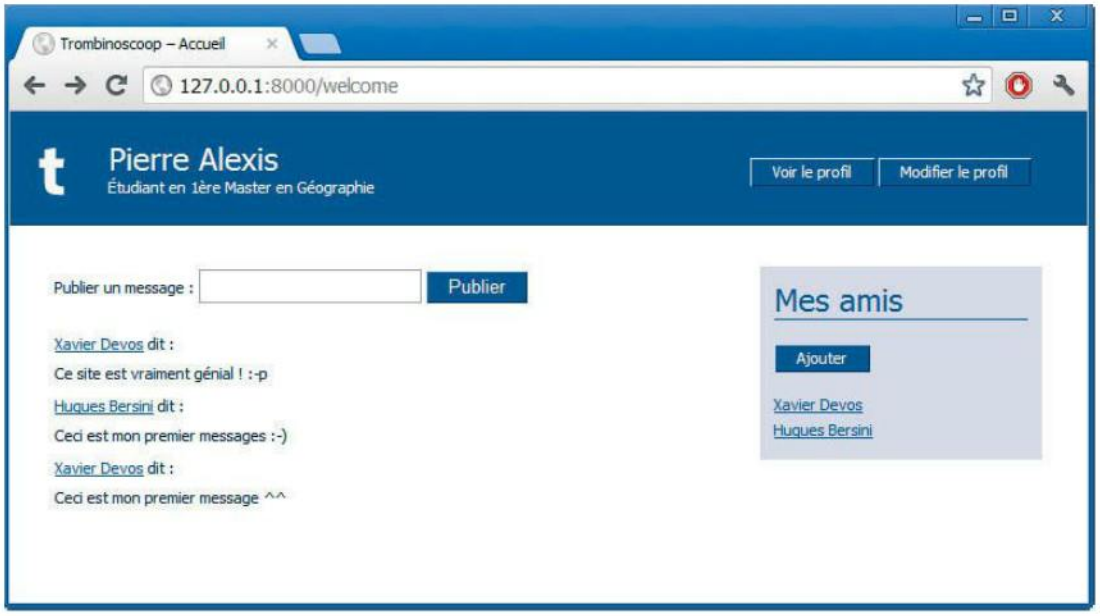


Figure 13–7 Notre page d'accueil

Ajout d'un champ texte et d'un lien

Il nous faut d'abord ajouter un champ texte dans un formulaire permettant de saisir directement l'adresse de la personne à ajouter. Nous insérons ce champ juste avant le lien *Ajouter*, qu'il va falloir transformer en bouton de soumission. En effet, il s'agit bien d'un lien hypertexte (balise `a`), même s'il ressemble à un bouton.

Ajoutons le formulaire dans le template `welcome.html` :

EXEMPLE 13.8 Trombinoscoop. Liste des amis dans welcome.html

```
<section id="friendList">
  <p class="title">Mes amis</p>
  <form action="welcome" method="get" class="inlineForm">
    <input type="email" name="newFriend" id="newFriendInput"
      placeholder="Courriel" />
    <input type="submit" value="Ajouter" />
```

```
  </form>
  <ul>
  {% for ami in logged_user.amis.all %}
    <li><a href="showProfile?userToShow={{ ami.id }}">{{ ami.prenom }}
                                                ➥ {{ ami.nom }}</a></li>
  {% endfor %}
  </ul>
</section>
```

EN PRATIQUE Pourquoi avoir conservé un formulaire ?

Nous aurions très bien pu ajouter simplement un champ texte et un bouton et les gérer entièrement en JavaScript. À quoi bon ajouter un formulaire, puisque nous sommes en Ajax ? De plus, si on clique sur le bouton de soumission, la page `welcome` sera rappelée avec, en paramètre, l'adresse de courriel de l'ami à ajouter.

La réponse est simple : pour les visiteurs de votre site qui auraient désactivé le JavaScript et pour ceux-là seulement (un programmeur averti en vaut deux), il faut maintenir un formulaire traditionnel. Aussi « désactivateurs de JavaScript » qu'ils soient, ils pourront eux aussi ajouter des amis.

C'est ce qu'on appelle la programmation « défensive » ou « prévoyante » et qui se dégrade « gracefully », comme disent les anglophones.

Dès lors, si le JavaScript est activé, l'ami risque d'être ajouté deux fois : une fois en Ajax et juste après, lorsque le formulaire sera soumis. Nous désactiverons l'envoi du formulaire avec JavaScript. Vous verrez comment dans la suite.

Ajout de l'URL et de la vue

EXEMPLE 13.9 Trombinoscoop. Ajout d'une URL permettant d'ajouter un ami

```
from views import ajax_check_email_field, ajax_add_friend

urlpatterns = patterns('',
  # Autres URL
  ('^ajax/checkEmailField$', ajax_check_email_field),
  ('^ajax/addFriend$', ajax_add_friend),
)
```

Le rôle de la vue correspondante sera double :

- d'une part ajouter l'ami à la liste des amis dans la base de données ;
- d'autre part retourner la portion de code HTML permettant d'insérer le nouvel ami dans le code de la page HTML. Cela se fait exactement comme dans l'exercice précédent : le HTML contenant les erreurs et récupéré du serveur est injecté dans le HTML de la page.

EXEMPLE 13.10 Trombinoscoop. Contenu de la vue ajax_add_friend

```
def ajax_add_friend(request):
  HTML_to_return = '' (6)
  logged_user = get_logged_user_from_request(request)
  if not logged_user is None: (1)
    if 'email' in request.GET:
      new_friend_email = request.GET['email'] (2)
      if len(Personne.objects.filter(courriel=new_friend_email)) == 1: (3)
        new_friend = Personne.objects.get(courriel=new_friend_email)
        logged_user.amis.add(new_friend) (4)
        logged_user.save()

        HTML_to_return = '<li><a href="showProfile?userToShow=' (5)
        HTML_to_return += str(new_friend.id)
        HTML_to_return += '">'
        HTML_to_return += new_friend.prenom + ' ' + new_friend.nom
        HTML_to_return += '</a></li>'

  return HttpResponse(HTML_to_return) (6)
```

On vérifie d'abord qu'on est bien authentifié (1). On récupère ensuite le paramètre contenant l'adresse courriel du nouvel ami (2). On vérifie que ce nouvel ami se trouve bien dans la base de données (3). Si c'est le cas, on l'ajoute comme ami à l'utilisateur authentifié (4). Et pour terminer, on construit le HTML qu'on va devoir insérer dans la page d'accueil pour y ajouter le nouvel ami. Si quelque chose s'est mal passé, on renvoie un `string` vide (6).

Création du JavaScript d'ajout d'un ami

Pour l'instant, contentons-nous de récupérer la réponse du serveur. Nous l'insérerons dans la page par la suite. Le code est le suivant et on le place où on veut dans la page, par exemple juste avant la balise `section` qui contient notre liste d'amis.

EXEMPLE 13.11 Trombinoscoop. JavaScript d'ajout d'un ami

```
<script type="text/javascript">
  $(document).ready(function()
  {
    $("#friendList form").submit(addFriend); (1)
  });

  function addFriend()
  {
    $fieldValue = $("#newFriendInput").val(); (2)
```

```
    $.ajax({ ❸
      url: '/ajax/addFriend', ❹
      data: ({email : $fieldValue}) , ❺
      type: 'GET',
      success: function($data, $textStatus, $XMLHttpRequest) { ❻
          alert($data);
        }
      }
    );
    return false;
  }
</script>
```

Ce code n'est pas si éloigné de celui de l'exercice précédent. On commence par intercepter les soumissions du formulaire d'ajout d'amis. Chaque fois que le formulaire est soumis, on appelle la fonction `addFriend` ❶. Dans cette fonction, on récupère d'abord la valeur du champ contenant l'adresse courriel du nouvel ami ❷. Ensuite, on envoie une requête Ajax ❸ qui appelle l'URL `/ajax/addFriend` ❹ en passant en paramètre le courriel de l'ami à ajouter ❺. Si la requête Ajax se passe au mieux, on affiche le HTML produit par le serveur ❻. En voici le résultat :

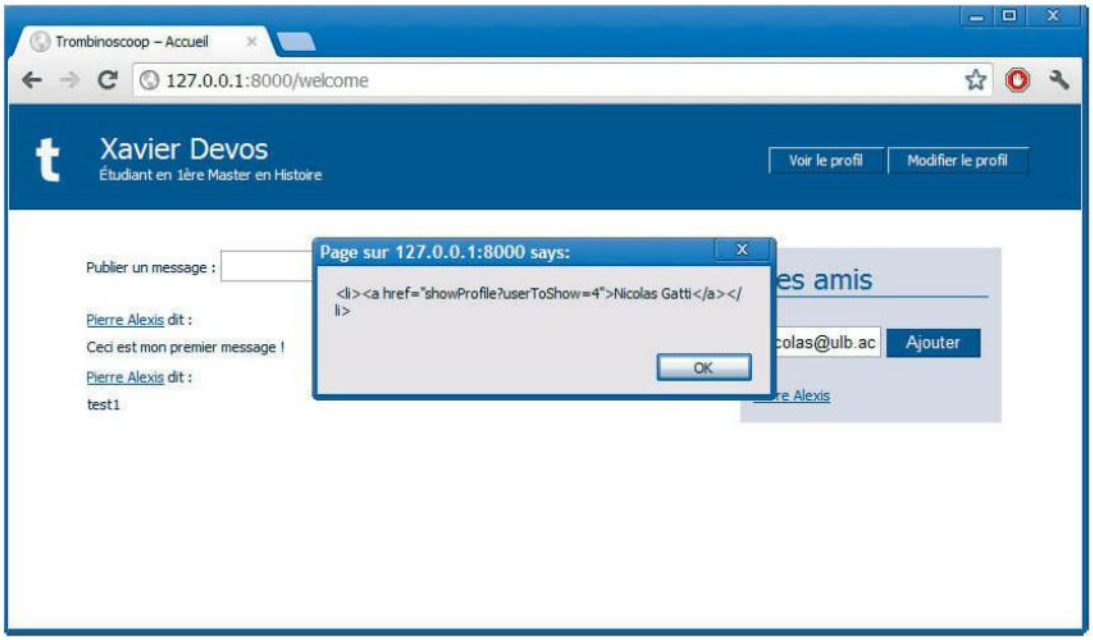


Figure 13–8 HTML renvoyé par le serveur lors de l'ajout d'un ami

Insertion du HTML dans la page web

EXEMPLE 13.12 **Exemple 13.12. Trombinoscoop. Ajout du nouvel ami dans la page**

```
function addFriend()
{
  $fieldValue = $("#newFriendInput").val();
  $.ajax({
    url: '/ajax/addFriend',
    data: ({email : $fieldValue}) ,
    type: 'GET',
    success: function($data, $textStatus, $XMLHttpRequest) {
        if ($data != '')
        {
          $('#friendList ul').prepend($data); (1)
        }
      }
    }
  );
  return false; (2)
}
```

On se positionne au niveau de la liste `ul` et on utilise la fonction `prepend` de jQuery qui va ajouter le HTML reçu par le serveur comme premier élément de la liste.

Dernière remarque : on renvoie `false` comme valeur de retour (2). Ceci informe le navigateur qu'il ne doit surtout pas soumettre le formulaire. L'ajout de l'ami a été fait en Ajax, inutile de le refaire en rechargeant la totalité de la page !

Si le JavaScript est désactivé, le formulaire sera simplement soumis et la page d'accueil sera rechargée, un paramètre lui ayant été passé (l'adresse courriel du nouvel ami). Nous ne le ferons pas, mais cela doit être traité dans Django au niveau de la vue qui gère l'affichage de la page d'accueil. Cette vue doit également s'occuper de l'ajout d'un ami.

Le résultat final de notre page d'accueil est le suivant, lorsqu'un ami a été ajouté en Ajax.

Et voilà qui termine notre site Trombinoscoop.

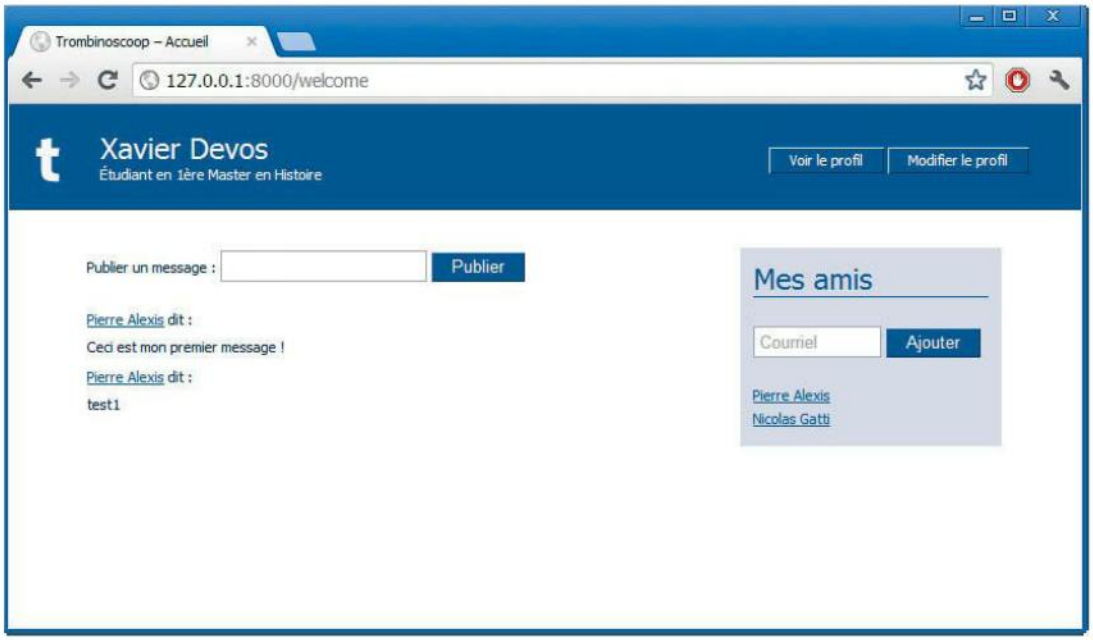


Figure 13–9 L'ami Nicolas a été ajouté dynamiquement

Conclusions

Nous le savons, la réalisation finale de Trombisnoscoop n'est finalement pas une si mince affaire. Elle vous est peut-être apparue, par moments, comme un véritable parcours du combattant. Nous nous voulions exhaustifs au possible ; au-delà de Django, nous avions l'ambition de vous familiariser à toutes les facettes et tous les métiers de la programmation web. Si vous avez réalisé et compris ce projet web jusqu'au bout, vous pouvez fièrement annoncer à tout futur employeur que vous maîtrisez déjà certaines ficelles de la programmation web, et pas seulement en Django ! À la lecture de ce livre, vous avez affronté et résolu de nombreux problèmes qui vous seraient apparus tout autant avec d'autres solutions technologiques, issues par exemple du monde Java ou .Net.

Au fil des pages et de la mise en forme de notre Trombisnoscoop, vous avez peut-être dû vous procurer d'autres sources d'information pour la programmation Python, HTML5, CSS, JavaScript, Ajax, le modèle relationnel, etc. Notre choix de l'exhaustivité, au détriment parfois de la profondeur de vue, vous aura permis de prendre le recul nécessaire, de remettre chaque outil à sa place, de comprendre le pourquoi de cette profusion technologique plutôt confondante qui caractérise le monde de la programmation web. Comme nombre de vos collègues, vous aurez fait vôtre cette multi-

tude de sigles qui, de HTML, CSS à Ajax, sont autant d'atouts dans votre main. Et si un soir de vague à l'âme, l'envie vous vient de jeter ce livre et tous ses codes Python et HTML par la fenêtre, rappelez-vous la belle histoire de Marc Van Saltberg, cet étudiant bruxellois. Elle devrait vous remonter le moral, vous convaincre d'aller de l'avant et de tourner la page.

Ai-je bien compris ?

- Quand on parle de site web « dynamique », décrivez deux niveaux de dynamisme vus dans ce livre.
- Quels sont les avantages d'Ajax ?
- Quelle différence faites-vous entre Ajax, JavaScript et jQuery ?

A

Installation de l'environnement de développement

Cette annexe décrit étape par étape l'installation des différents éléments logiciels nécessaires à la réalisation du projet Trombinoscoop.

SOMMAIRE

- Annexe décrivant l'installation des différents outils nécessaires au développement web sous Django : Python, Java, Django, Eclipse et plug-ins permettant l'exécution de code Python dans l'environnement Eclipse.
- Cette installation est décrite pour les trois systèmes d'exploitation les plus répandus : Windows, Mac OS et Linux.

Avant de se lancer dans la programmation Python/Django, il est nécessaire de configurer son environnement de travail. L'objectif de ce chapitre est d'expliquer chacune des étapes qui permettront d'installer et de configurer les divers outils nécessaires pour se constituer un environnement de développement optimal. Les explications sont déclinées pour trois systèmes d'exploitation :

- Microsoft Windows 7 ;
- Mac OS X Mountain Lion ;
- Ubuntu 12.

Nous espérons couvrir la grande majorité des systèmes d'exploitation utilisés aujourd'hui, sachant que les explications sont facilement adaptables pour leurs variantes. Par exemple, la procédure décrite pour Mac OS X Moutain Lion reste valable pour les versions inférieures Lion et Snow Leopard.

Que faut-il installer ?

Python

Le premier outil à installer, et dont on ne pourra pas se passer, est le langage de programmation Python qui nous permettra de compiler et d'exécuter des programmes écrits dans ce langage.

Python est accompagné d'une importante *bibliothèque standard* offrant de nombreux modules pré-écrits (types de données complexes, cryptographie, traitement d'images, traitement audio, envoi de courriels, etc.). C'est par l'instruction `import` qu'on appelle ces différents utilitaires dans les codes qui en font usage.

Il existe plusieurs implémentations du langage Python et de sa bibliothèque standard. Nous utiliserons l'implémentation « traditionnelle » de référence, proposée sur le site web python.org et disponible pour de nombreuses plates-formes (Windows, Linux, etc.).

Python, comme tout langage de programmation qui se respecte, est en constante évolution ; il en existe plusieurs versions. La dernière version majeure est la 3 et mérite qu'on s'y attarde un peu. Elle a eu notamment pour objectif de simplifier le langage en lui retirant certaines constructions jugées redondantes. La compatibilité ascendante a été brisée. En d'autres mots, et à la grande déconvenue de nombreux programmeurs, le compilateur Python 3 n'est plus capable de compiler tous les programmes écrits en Python 2. Or, Django a été écrit à la base pour Python 2 et sa migration vers Python 3 n'a pas encore abouti. Nous installerons donc la version 2.7 de Python, qui est la dernière précédant la version 3. Toutefois rassurez-vous, l'adaptation de Django à la der-

nière version de Python, qui se fera bien un jour, ne devrait pas modifier pour l'essentiel notre enseignement et le projet que nous vous proposons.

Django

Une fois Python installé, il faut l'enrichir avec le framework Django, qui n'est pas prévu de base dans la bibliothèque standard Python. Ce framework se télécharge sur le site officiel de Django.

Un des avantages de Django est l'intégration d'un serveur web léger. On n'est donc pas obligé d'installer un serveur web tiers, tel Apache. Bien entendu, si le serveur web léger inclus dans Django suffit amplement pour tester son site durant la phase de développement, il n'en sera pas de même lorsqu'on ouvrira son site au public. Un serveur web tiers plus robuste sera nécessaire, car il permettra de mieux supporter un nombre élevé de visites sur le site et permettra une configuration plus fine des utilitaires web.

Eclipse

Python et Django suffiraient à débuter le développement de notre premier site web. À l'aide d'un simple éditeur de texte et d'une interface en ligne de commande, nous pourrions écrire notre premier code et le lancer. Ce serait néanmoins laborieux et peu convivial. C'est pourquoi nous allons installer un *environnement de développement intégré* ou IDE (*Integrated Development Environment*).

> **DÉFINITION IDE**
>
> Un environnement de développement intégré est un ensemble d'outils facilitant et rendant plus convivial le développement d'applications.

Généralement, les IDE offrent au moins les utilitaires suivants, plutôt précieux :

- un éditeur de texte capable de colorer le code, de détecter les erreurs de syntaxe en ligne ou d'aider à la saisie de code en affichant toutes les constructions possibles ;
- une interface graphique conviviale pour simplifier la compilation d'un programme ou le lancement de l'application ;
- un débogueur graphique permettant d'exécuter pas à pas un programme et d'observer son état à tout instant (valeurs des variables, position dans le code, etc.).

Notre choix d'environnement de développement intégré s'est porté sur Eclipse car il s'agit d'un environnement très populaire et complet, capable de gérer plusieurs langages de programmation, dont Python et Django.

Eclipse est un IDE écrit en Java. C'est aussi à ce jour l'environnement de développement Java le plus utilisé, sachant que Java est actuellement le langage de programmation le plus répandu et le plus enseigné. Le moteur d'exécution Java, qui permet de lancer des programmes écrits dans ce langage, devra donc faire partie de la panoplie d'outils à installer, sans quoi Eclipse ne pourra pas être lancé.

L'installation de base d'Eclipse ne contient pas les modules permettant de gérer le langage Python et le framework Django. Ces modules, regroupés dans le plug-in nommé PyDev, doivent être ajoutés manuellement par la suite, ce que nous ferons également.

En résumé

La mise en place de notre environnement de développement passera par l'installation successive de ces cinq composants :

- le langage Python, dans sa version 2.7 ;
- le framework Django ;
- le moteur d'exécution Java (*Java Runtime Environment*) ;
- l'IDE Eclipse ;
- le plug-in Eclipse PyDev.

Les sections suivantes sont consacrées à l'installation de ces éléments sous différents systèmes d'exploitation. Vous pouvez bien entendu ne lire que les sections qui correspondent à votre environnement.

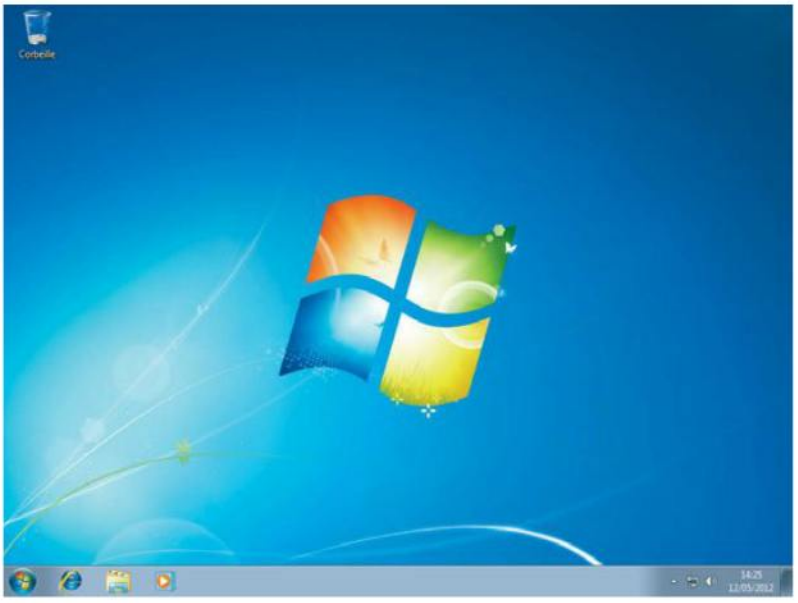


Figure A-1
Windows 7 professionnel 32 bits

Les différentes étapes ont été réalisées et testées sous Windows 7 Professionnel 32 bits, sans *service pack* installé, Mac OS X Lion et Ubuntu 12.04 dans sa version 32 bits. Elles devraient rester pour l'essentiel identiques pour les autres versions des systèmes d'exploitation.

Figure A-2
Mac OS X Lion

Figure A-3
Ubuntu 12.04 32 bits

Installation de Python

Commençons par une bonne nouvelle : sous Ubuntu, Python est déjà pré-installé.

Pour Windows et Mac OS X, la première étape consiste à installer l'implémentation officielle *CPython*.

EN PRATIQUE « Vieilles » versions de Python sous Mac OS X

Mac OS X est livré en standard avec Python. Malheureusement, celui-ci n'est mis à jour qu'à chaque sortie d'une nouvelle version du système, soit environ tous les deux ans. On se retrouve souvent avec une version de Python largement dépassée. Il est donc indispensable d'installer la version qui nous intéresse à côté de celle existante et qui cohabitera en très bon voisinage avec cette dernière.

EN PRATIQUE Version de Python

La version à installer est la 2.7 et non la 3, cette dernière n'étant pas compatible avec Django.
Nous recommandons d'installer la dernière version de la lignée 2.7. À l'heure où ces lignes sont écrites, il s'agit de la version 2.7.3.

Pour la télécharger, il faut se rendre dans la section *Download* de www.python.org. Différents paquets sont disponibles. Nous avons choisi :

- pour Windows, le paquet `Windows x86 MSI Installer`, comprenant un installeur destiné aux Windows 32 bits ;
- pour Mac OS X, le paquet `Mac OS X 64-bit/32-bit x86-64/i386 Installer`. C'est en réalité une image disque `DMG` qu'il suffit de monter en l'ouvrant.

▸ www.python.org

Pour Windows

Une fois le paquet téléchargé, il suffit de l'exécuter et de suivre les étapes de l'assistant d'installation. À la première étape, l'assistant vous demande si vous désirez installer Python pour tous les utilisateurs de l'ordinateur ou juste pour l'utilisateur courant ; il est recommandé de choisir *Install for all users*.

L'assistant demande ensuite de choisir un emplacement pour l'installation. Afin de respecter les standards Windows, nous recommandons d'installer Python dans `Program Files` et non à la racine du disque système, comme proposé par défaut par l'installeur.

Figure A-4
Première étape de l'assistant d'installation

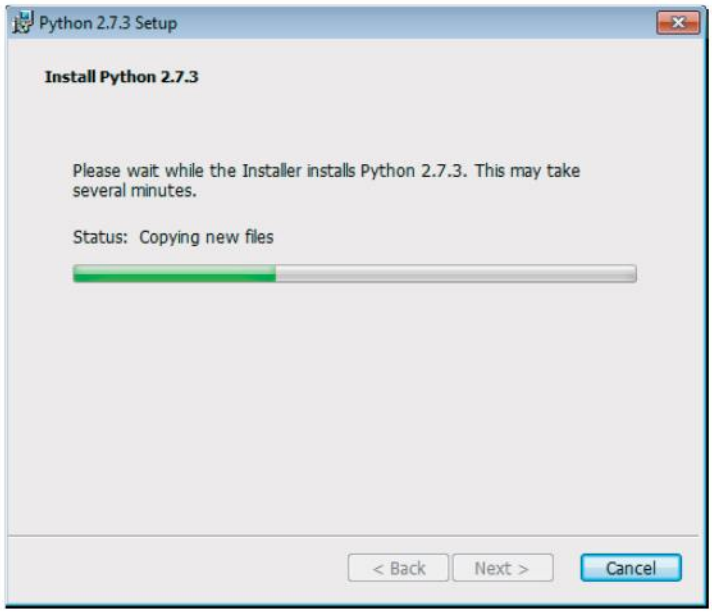
Figure A-5
Deuxième étape de l'assistant d'installation

À la troisième étape, l'installeur demande quels composants de Python installer. Nous recommandons de tout sélectionner, afin de tout installer.

Figure A-6
Troisième étape de l'assistant d'installation

À l'étape suivante, l'installation commence.

Figure A-7
Quatrième étape de l'assistant d'installation



Python demeurant un langage et un environnement somme toute très léger, l'installation est terminée après quelques minutes. Il ne reste plus qu'à cliquer sur le bouton *Finish* pour fermer l'assistant.

Figure A-8
Dernière étape de l'assistant d'installation

Pour Mac OS X

Une fois l'image montée, il suffit d'exécuter l'application `Python.mpkg` et de suivre les étapes de l'assistant d'installation. À la fin du processus, on obtient cette fenêtre :

Figure A-9
Dernière étape de l'assistant d'installation

Cet installateur met Python dans le dossier `Applications` de votre Mac.

Vérification de l'installation

Il reste à valider la bonne installation du langage en lançant par exemple la ligne de commande graphique Python :

- *Démarrer > Tous les programmes > Python > IDLE (Python GUI)* sous Windows ;
- *Applications > Python 2.7 > IDLE* sous Mac OS X.

Ensuite, il suffit de saisir un peu de code, par exemple le sempiternel « hello world », bien connu des programmeurs :

EXEMPLE A.1 Petit programme de test

```
print('Hello World !')
```

Ce code devrait afficher le texte « Hello World ! » dans la console, comme illustré sur la figure suivante. Vous pourriez refaire l'exercice avec « Bonjour Monde ! », mais sachez que cela fait tout de suite beaucoup moins pro.

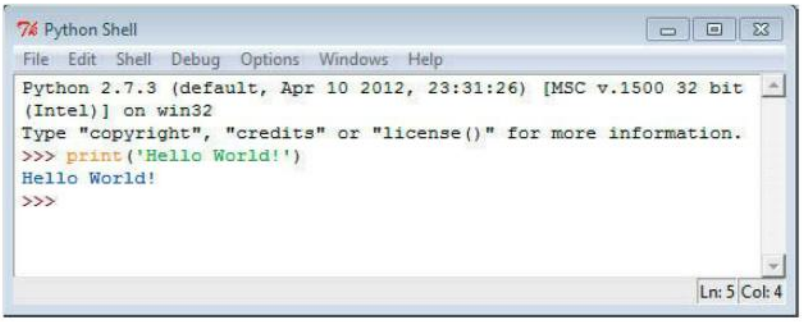


Figure A-10
Test de l'installation

Sous Ubuntu, il suffit de lancer un terminal (via le *Tableau de bord*, on lance l'application *Terminal*) et de taper la commande `python`. Apparaît alors le numéro de version du langage :

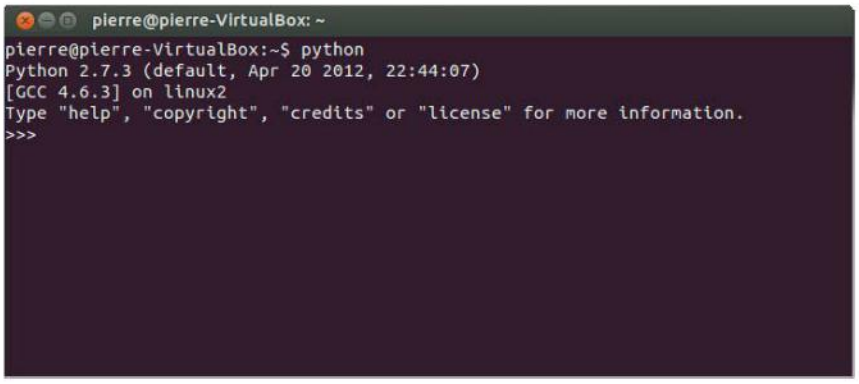


Figure A-11
Version de Python incluse dans Ubuntu

Installation de Django

Django, écrit en Python, n'est pas livré en standard avec ce dernier. Il s'agit d'un framework tiers à installer manuellement.

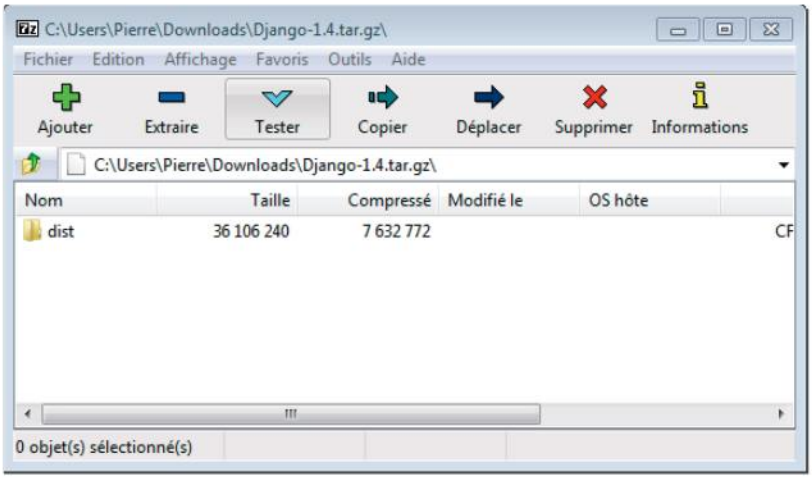
Pour Windows et Mac OS X, le framework est disponible en téléchargement sur le site du projet Django. Il faut se rendre dans la section *Download* et prendre la dernière version officielle. À l'heure où ces lignes sont écrites, il s'agit de la version 1.4. Le fichier à télécharger se présente sous la forme d'un fichier `tar.gz`, en d'autres mots une archive TAR qui a été compressée en format GZIP.

▸ www.djangoproject.com

Pour Windows

Il faut utiliser une application tierce qui va ouvrir, décompresser et extraire les fichiers d'installation de Django que renferme le fichier `tar.gz`. L'application que nous avons utilisée est *7-Zip*, disponible gratuitement. D'autres décompresseurs et extracteurs peuvent bien entendu être utilisés. Ouvrez le fichier `.gz` à l'aide de l'application 7-Zip. Un seul dossier se présente dans l'explorateur de l'archive : `dist`.

Figure A-12
Contenu de l'archive TAR



Double-cliquez sur le dossier `dist` pour en afficher le contenu. Un seul fichier s'y trouve : `Django-1.4.tar`. À nouveau, double-cliquez sur ce fichier TAR pour en afficher le contenu. Dans cette archive, se trouve un seul dossier, nommé `Django-1.4` ; cliquez sur le bouton *Extraire* de 7-Zip. Un emplacement destiné à accueillir le dossier extrait vous est demandé ; choisissez celui qui vous convient le mieux.

L'archive extraite, nous pouvons lancer l'installeur qui s'y trouve. Cet installeur étant écrit en Python, on ne peut pas le lancer comme un exécutable Windows traditionnel en double-cliquant dessus. Il va falloir le lancer en ligne de commande en utilisant Python.

Il faut d'abord lancer une ligne de commande Windows, avec les droits d'administrateur. La ligne de commande Windows se trouve dans *Menu démarrer>Tous les programmes>Accessoires>Invite de commande*. Pour la lancer avec les droits d'administrateur, il faut cliquer droit sur son raccourci et choisir *Exécuter en tant qu'administrateur*.

Une fois l'invite de commande lancée, il faut se rendre, à l'aide de la commande `cd`, dans le dossier d'installation de Django que l'on vient d'extraire. Par exemple, ayant extrait ce dossier dans `C:\Users\Pierre\Downloads\`, nous avons tapé la ligne de commande suivante :

EXEMPLE A.2 Positionnement dans le dossier d'installation

```
cd Downloads\Django-1.4
```

Nous n'avons pas spécifié le chemin absolu dans la commande, car en ouvrant la console, celle-ci nous avait déjà positionnés dans le dossier `c:\Users\Pierre`.

Pour lancer l'installation de Django, il faut entrer la commande suivante :

SYNTAXE. Lancement de l'installation

```
"c:\Program Files\Python27\python.exe" setup.py install
```

La figure suivante illustre l'utilisation de ces deux commandes.

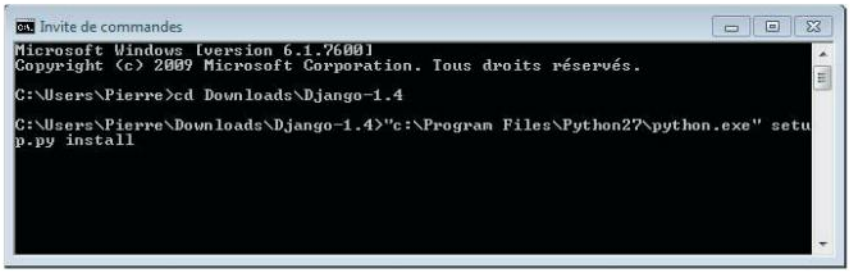
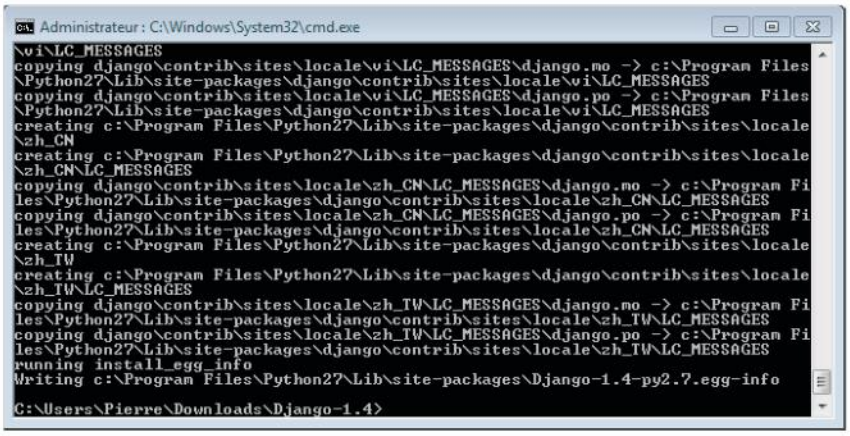


Figure A-13
Lancement de l'installation de Django

Si aucune erreur ne survient, c'est que l'installation s'est bien déroulée.

Figure A-14
Installation terminée de Django



Pour Mac OS X

Pour l'archive sous Mac, il faut utiliser la ligne de commande. Ouvrez d'abord un *Terminal* (dans *Applications > Utilitaires*). Rendez-vous ensuite dans le dossier où a été téléchargée l'archive, à l'aide de la commande `cd` :

EXEMPLE A.3 Positionnement dans le dossier de téléchargement

```
cd Downloads
```

Puis lancez la décompression de l'archive et rendez-vous dans le dossier créé :

SYNTAXE. Décompression de l'archive

```
tar xzvf Django-1.4.tar.gz
cd Django-1.4
```

Ensuite, lancez l'installation de Django à l'aide de l'interpréteur Python que nous venons d'installer (et non de celui fourni de base dans Mac OS X). Il se trouve dans *Library > Frameworks*. Utilisez la commande `sudo` car l'installation doit se faire en tant qu'administrateur.

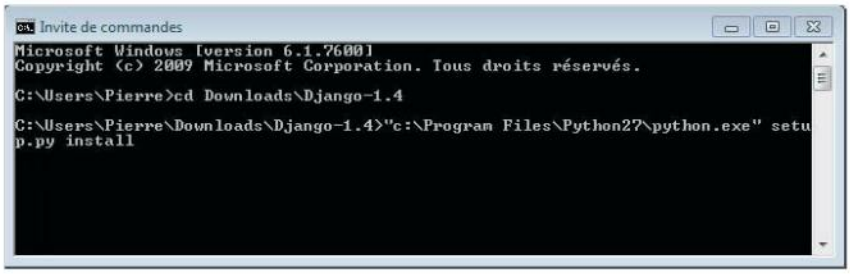
SYNTAXE. Installation de Django

```
sudo /Library/Frameworks/Python.framework/Versions/2.7/bin/python
  setup.py install
```

Attention, il va falloir entrer le mot de passe administrateur.

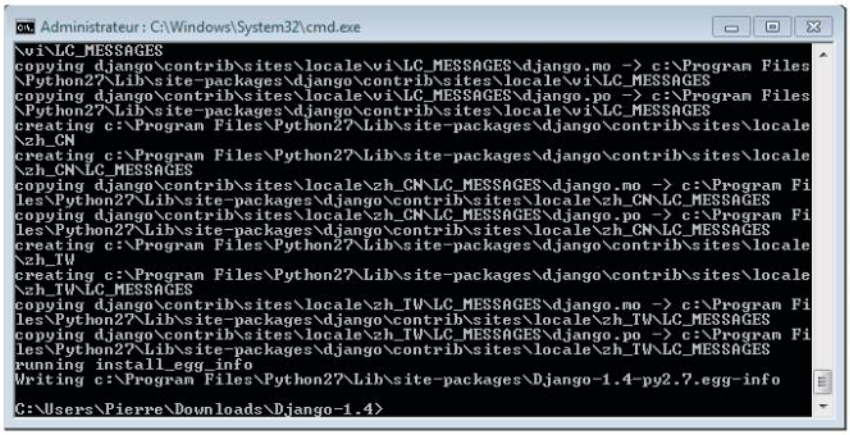
La figure suivante illustre l'utilisation de cette commande.

Figure A-15
Lancement de líinstallation de Django



Si aucune erreur ne survient, l'installation s'est bien déroulée.

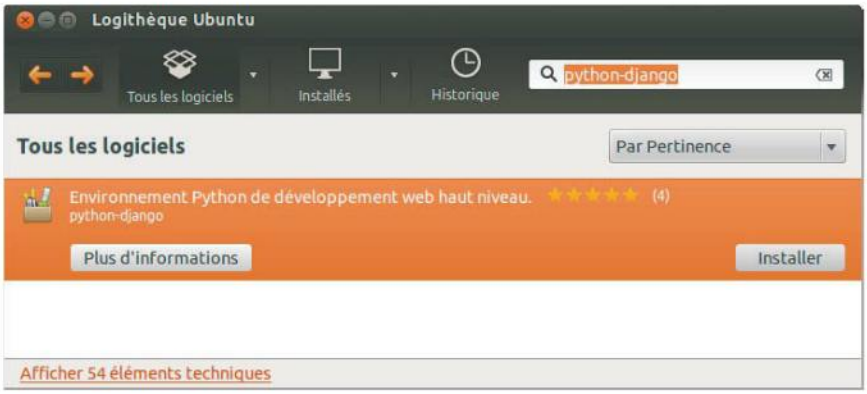
Figure A-16
Installation terminée de Django



Pour Ubuntu

Django est disponible dans la *Logithèque* d'Ubuntu. Il suffit de chercher le logiciel `python-django` et, celui-ci trouvé, de cliquer sur *Installer*.

Figure A-17
Django dans la Logithèque Ubuntu



Vérification de l'installation

Sous Windows et Mac OS X, on peut tester si l'installation s'est exécutée correctement en lançant une console Python et en tapant le code suivant :

SYNTAXE. Vérification de la bonne installation de Django sous Windows et Mac OS X

```
>>> import django
>>> print(django.get_version())
1.4
```

Le code doit afficher la version de Django, soit « 1.4 », comme illustré à la figure suivante.

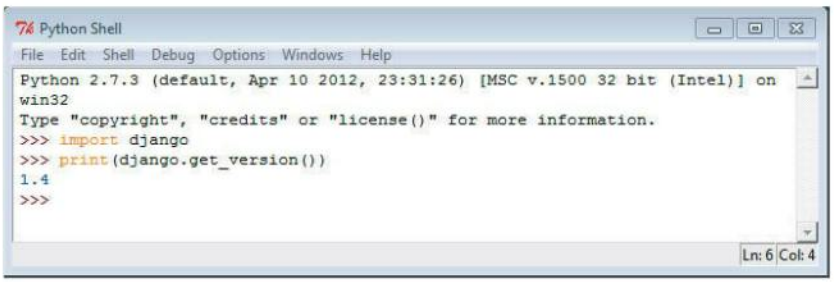


Figure A-18
Vérification de la bonne installation de Django sous Windows et Mac OS X

Sous Ubuntu, on peut tester la bonne installation de Django en lançant un terminal et en tapant le code suivant :

SYNTAXE. Vérification de la bonne installation de Django sous Ubuntu

```
python
>>> import django
>>> print(django.get_version())
1.3.1
```

Le code doit afficher la version de Django, soit « 1.3.1 », comme illustré à la figure suivante.

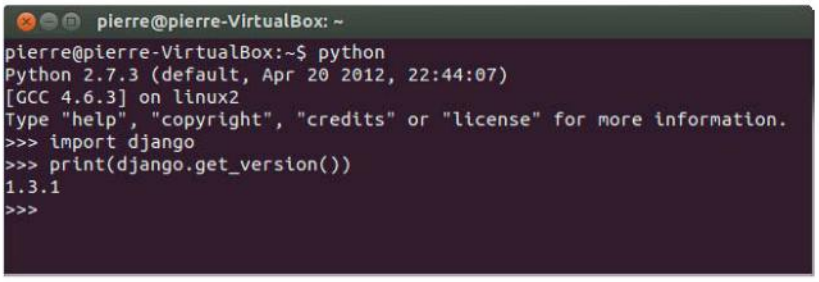


Figure A-19
Vérification de la bonne installation de Django sous Ubuntu

Installation de Java

Si Java n'est pas déjà installé sur votre ordinateur (ce qui est peu probable), vous devrez y remédier.

Pour Windows et Mac OS X

Sur la page d'accueil du site officiel de Java, un lien mis en évidence vous permet de télécharger l'installateur. Choisissez la version de Java correspondant à votre système (32 ou 64 bits). L'installation ne nécessite pas de paramétrage particulier.

▸ www.java.com

Pour Ubuntu

Rendez-vous à nouveau dans la *Logithèque*. Il suffit d'y chercher « Java », de sélectionner *OpenJDK Java 7 Runtime* et de cliquer sur *Installer*.

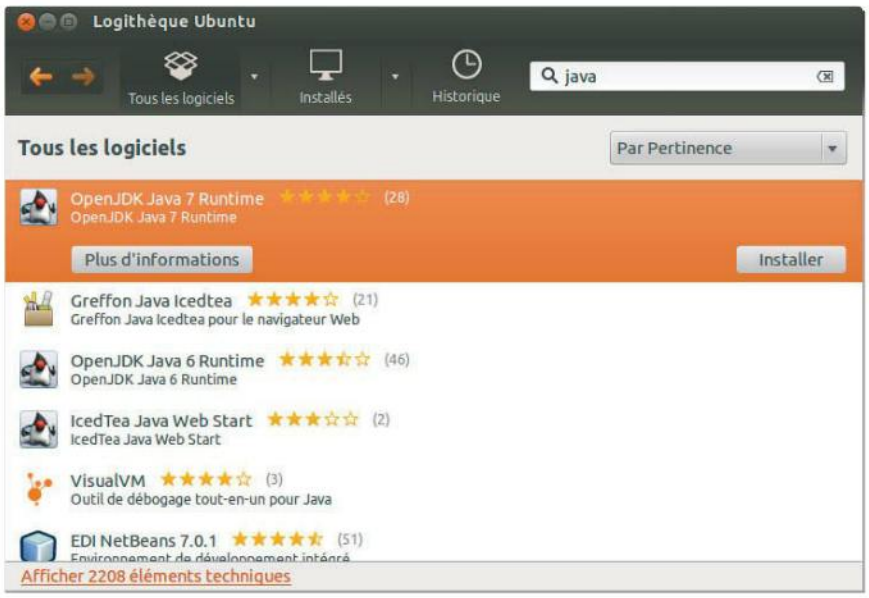


Figure A-20
Installation de Java avec Ubuntu

Installation d'Eclipse

Nous allons maintenant installer Eclipse, notre environnement de développement intégré.

Le téléchargement d'Eclipse se fait via le site officiel du projet pour Windows et Mac OS X. On retrouve sur cette page plusieurs versions de l'outil, chacune ciblant un langage de programmation ou un usage bien précis. Elles ne sont en fait que la version de base de l'outil sur laquelle ont été pré-installés les plug-ins ad hoc.

▸ http://www.eclipse.org/downloads/

Comme il n'en existe pas pour Python et Django, nous allons simplement télécharger la version de base, « Eclipse Classic » pour Windows ou Mac (version 3.7.2 au moment où nous écrivons ces lignes), et nous installerons par la suite les plug-ins qui nous intéressent.

Une fois téléchargé, Eclipse se présente sous la forme d'une archive (ZIP pour Windows et tar.gz pour Mac), qu'il suffit de décompresser comme nous l'avons fait pour Django. Elle contient un dossier reprenant tous les fichiers de l'application. Aucun installateur n'est donc à lancer ; il suffit de déplacer le dossier que l'on a décompressé de l'archive à l'emplacement de son choix. Pour notre part, nous avons choisi de déplacer le dossier dans `C:\Program Files` sur Windows et *Applications* pour Mac.

Pour Ubuntu, nous allons faire usage une fois encore de la *Logithèque*. Il suffit de chercher *Eclipse*, de le sélectionner et de cliquer sur *Installer*.

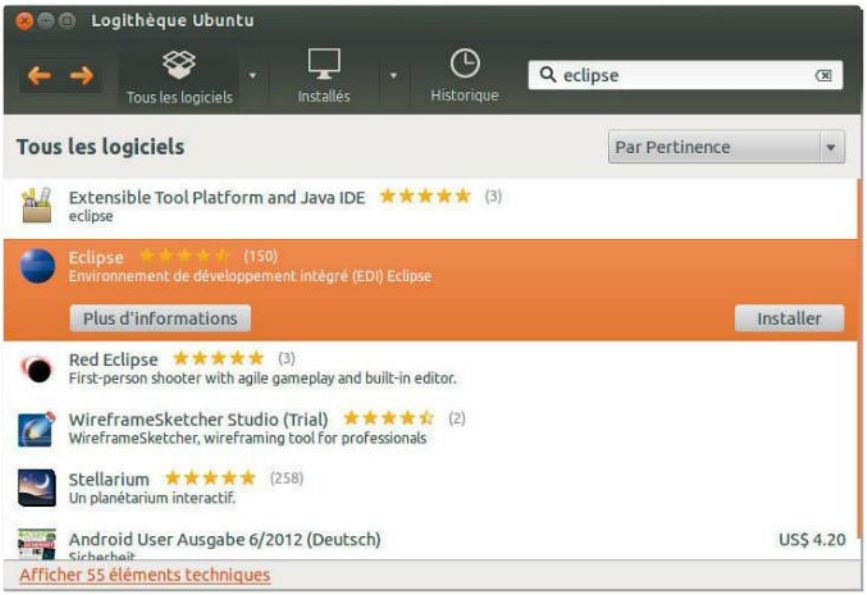


Figure A-21
Installation d'Eclipse sous Ubuntu

Nous pouvons maintenant lancer Eclipse :

- Sous Windows, double-cliquez sur `eclipse.exe`.
- Sous Mac, si Java n'est pas installé à ce stade, *Mise à jour de logiciels* permet de le faire en affichant un dialogue le proposant.
- Sous Ubuntu, démarrez Eclipse à l'aide du tableau de bord.

Au démarrage, Eclipse nous demande de choisir un *workspace* (espace de travail) et nous propose, par défaut, d'utiliser `C:\Users\XXXXX\workspace` pour Windows et Ubuntu ou `/Users/XXXXX/Documents/workspace` pour Mac. Un workspace regroupe différents projets, spécifie leur emplacement sur le disque et enregistre un certain nombre de préférences propres à Eclipse. Définir plusieurs *workspaces* permet donc d'avoir des environnements de travail différents (paramétrés différemment) contenant des projets différents. Tout au long de ce livre, nous n'utilisons qu'un seul workspace.

Vous pouvez très bien utiliser le workspace proposé par défaut. Néanmoins, nous avons préféré créer et utiliser un nom plus convivial : `C:\Users\Pierre\Projets` pour Windows et Ubuntu ou `/Users/Pierre/Documents/Projets` pour Mac. Pour créer un nouveau workspace, il suffit d'entrer son nom dans la boîte de dialogue de choix de workspace.

Vous pouvez également cocher la case *Use this as the default and do not ask again* afin qu'au prochain démarrage, Eclipse ne vous demande plus quel workspace utiliser.

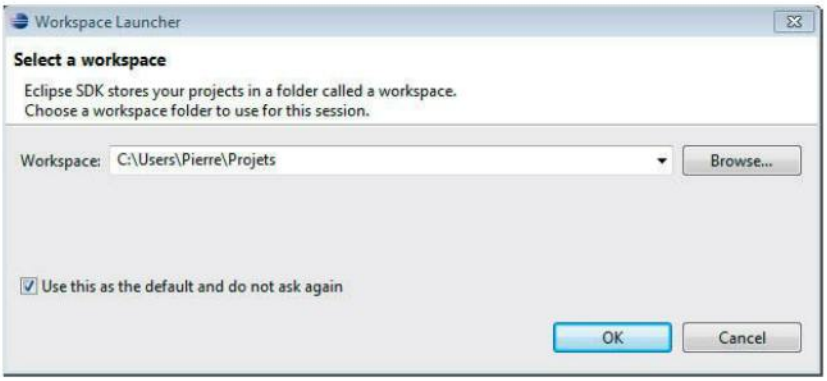


Figure A-22
Création et utilisation du workspace C:\Users\Pierre\Projets (Windows)

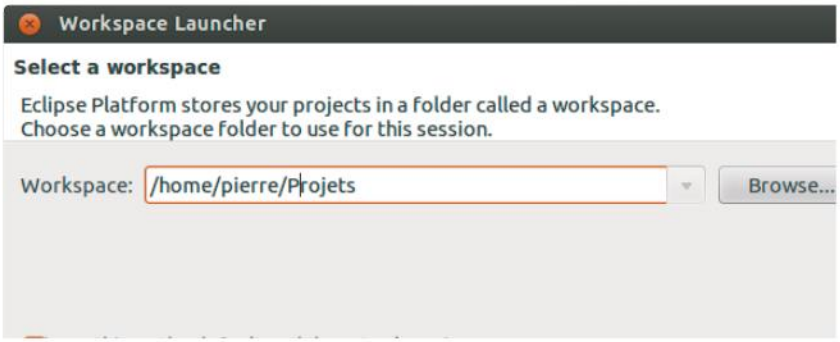


Figure A-23
Création et utilisation du workspace C:\Users\Pierre\Projets (Ubuntu)

Il reste à cliquer sur *OK*. Eclipse s'ouvre alors et affiche sa page d'accueil.

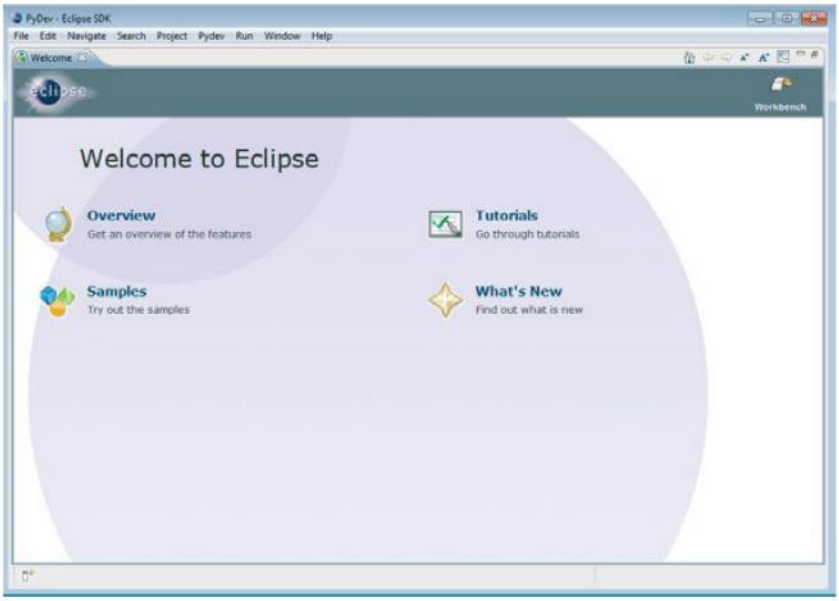


Figure A-24
Écran d'accueil d'Eclipse

Et voilà ! Eclipse est installé, Nous n'avons plus à nous occuper que du plug-in PyDev.

Installation du plug-in Eclipse PyDev

Avant de pouvoir commencer à développer en Python et Django dans Eclipse, il nous faut installer le plug-in PyDev.

Les plug-ins s'installent via le menu *Help > Install New Software...* La fenêtre suivante apparaît :

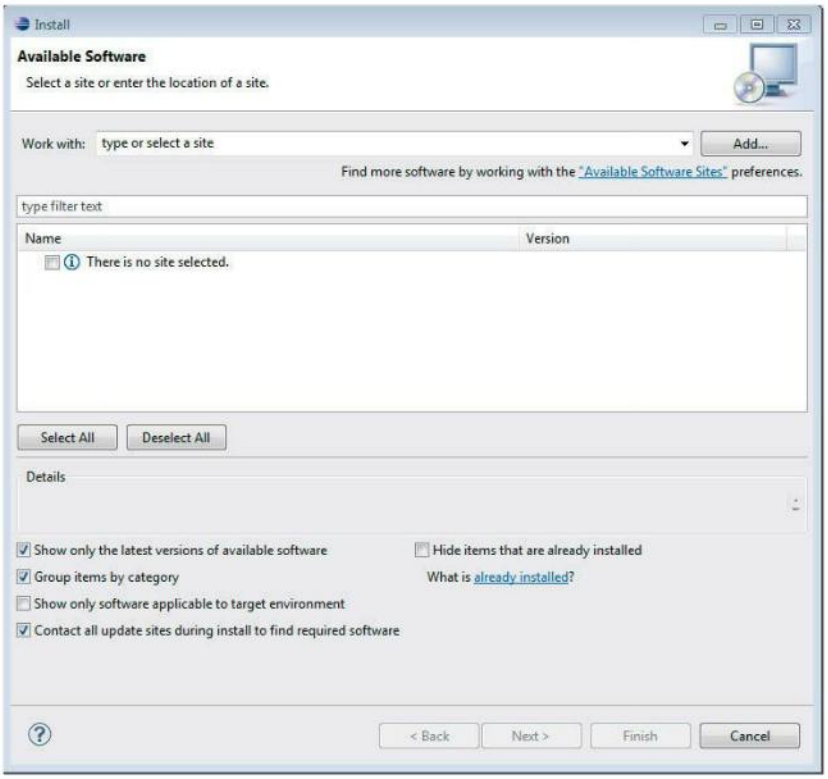


Figure A-25
Écran d'installation de plug-ins

Tout d'abord, nous devons renseigner le *site* sur lequel se trouve le plug-in PyDev. Cliquez sur le bouton *Add...* et, dans la fenêtre qui apparaît, inscrivez `PyDev` et `http://pydev.org/updates`.

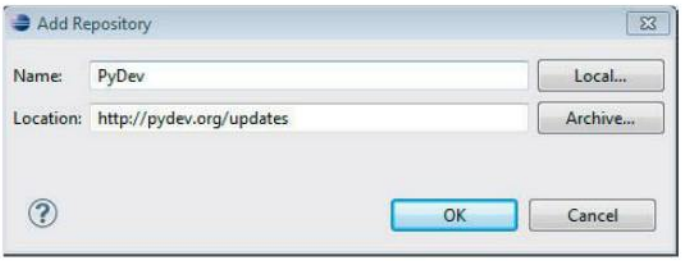
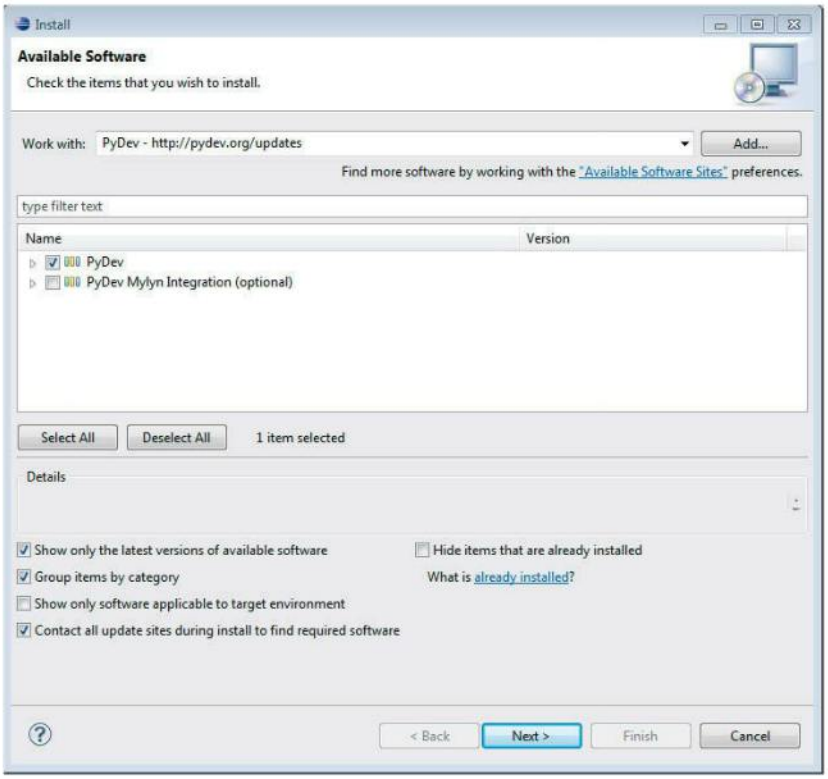


Figure A-26
Ajout du site pour le plug-in PyDev

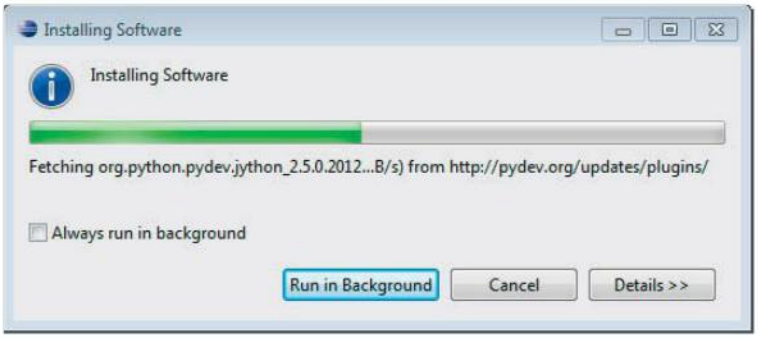
Cliquez sur *OK*. Choisissez ensuite *PyDev* dans la liste des plug-ins disponibles à installer.

Figure A-27
Choix de PyDev



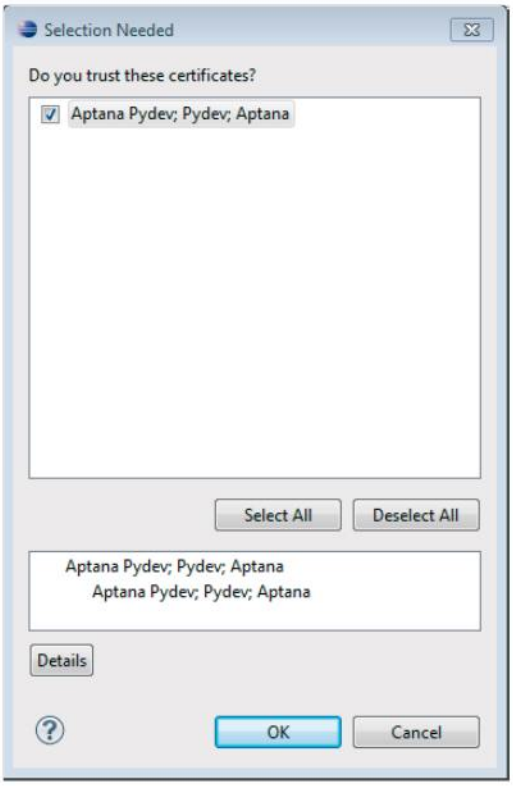
On clique sur *Next* deux fois, on accepte la licence, on clique sur *Finish* et l'installation débute.

Figure A-28
Installation de PyDev



Au cours de l'installation, Eclipse vous demandera de valider un certificat. Il suffit de s'assurer qu'ils sont tous sélectionnés et de cliquer sur *OK*.

Figure A-29
Installation de PyDev



Il est possible qu'à ce stade, on vous demande de redémarrer votre ordinateur, pour que les plug-ins s'installent vraiment. Vous pouvez vous contenter de réexécuter Eclipse.

Il reste maintenant à configurer PyDev pour lui indiquer où se trouve le compilateur Python et ses bibliothèques (que nous avons installées au tout début de ce chapitre).

Pour ce faire, il faut aller dans le menu *Window > Preference* (ou *Eclipse > Preference*). Dans l'arborescence de paramètres, rendez-vous à la section *PyDev > Interpreter – Python*.

Cliquez sur le bouton *Auto Config*. Dans la fenêtre qui apparaît, choisissez tous les éléments du dossier d'installation de Python. Cliquez sur *OK* pour valider.

Si *Auto Config* ne marche pas, il suffit alors de faire *New* et de spécifier comme emplacement l'endroit où se trouve l'exécutable de l'interpréteur Python que l'on désire utiliser. Par exemple, pour Mac OS X, il s'agira de `/Library/Frameworks/Python.framework/Versions/2.7/bin/python`.

La configuration est maintenant terminée, cliquez sur *OK* pour fermer la fenêtre de préférences.

Figure A-30
Configuration de PyDev

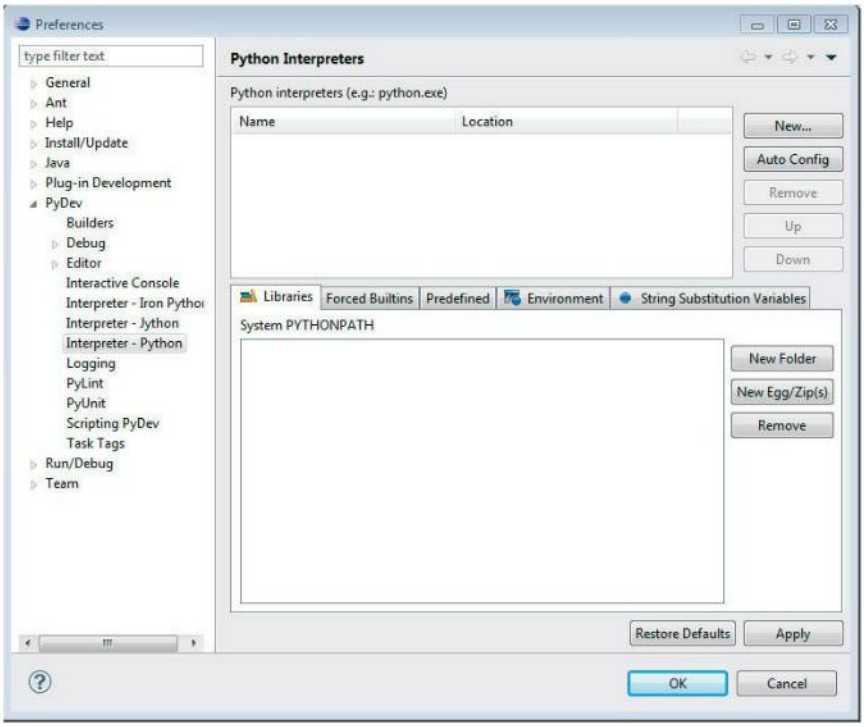


Figure A-31
Configuration de PyDev

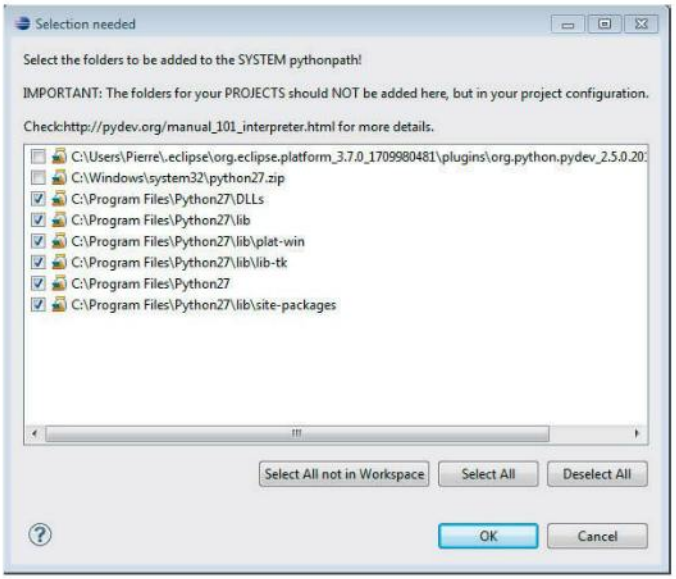
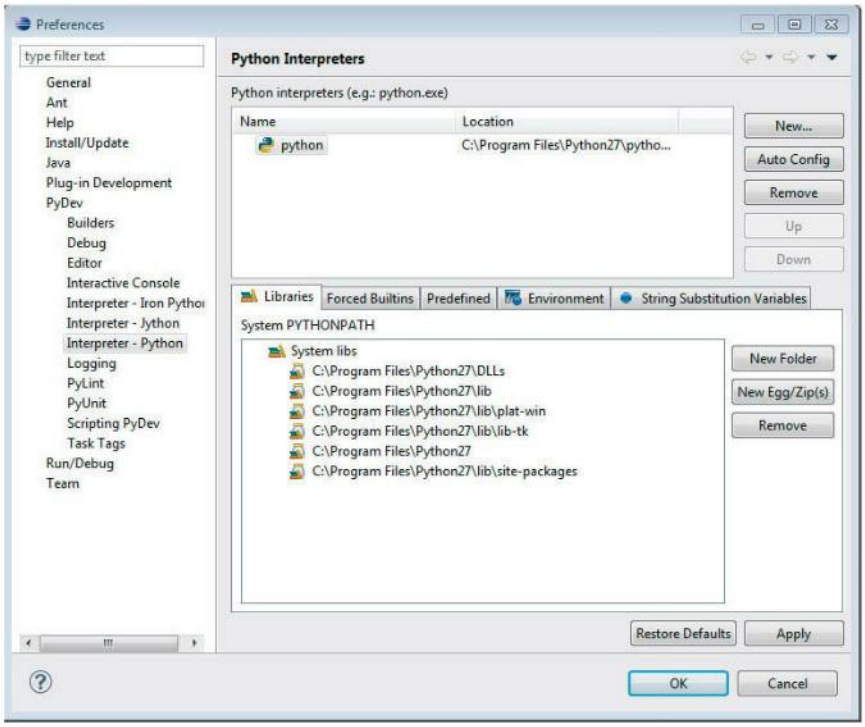


Figure A-32
Configuration de PyDev



Installation de Web Developer Tools pour Eclipse

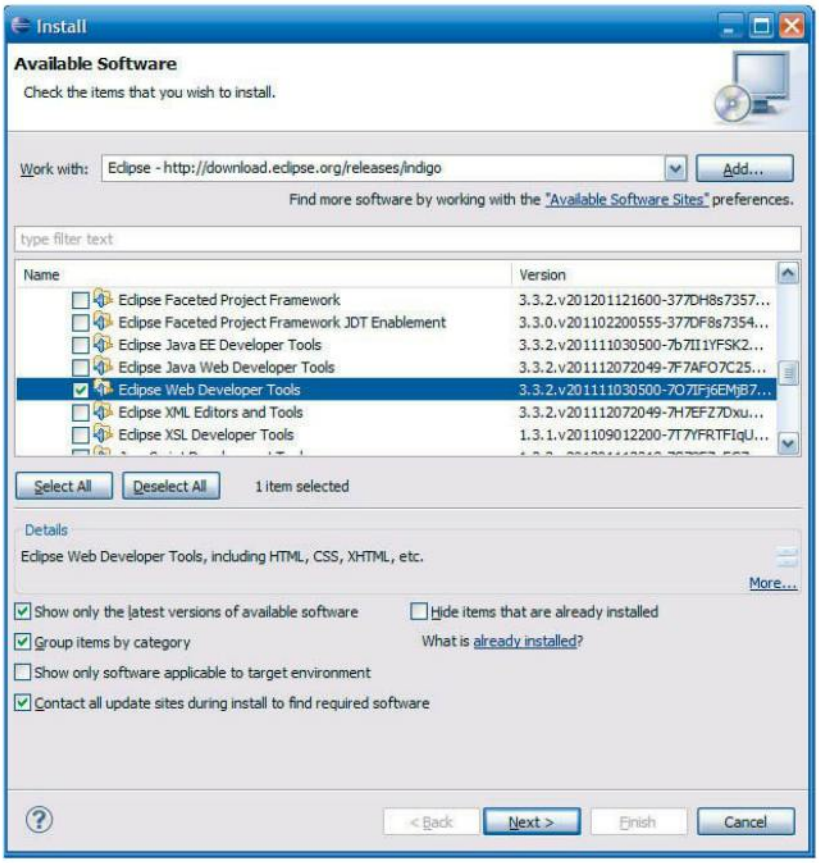
Ce paquet supplémentaire dans Eclipse nous sert dans le chapitre 8 pour éditer directement nos HTML. Il suffit de procéder de la même manière que pour PyDev.

Rendez-vous dans le menu *Help > Install New Software* et cliquez sur le bouton *Add*. Comme nom de site, indiquez ce que vous voulez, par exemple *Eclipse*. Comme emplacement (*location*), mettez http://download.eclipse.org/releases/indigo (du moins, si vous travaillez avec la version Indigo d'Eclipse). Si vous utilisez une autre version, il faut remplacer indigo dans l'adresse par le nom de celle-ci (regardez bien l'écran de démarrage d'Eclipse, il y est indiqué).

Le site ajouté, sélectionnez-le et patientez pendant que la liste des paquets disponibles se met à jour. Rendez-vous ensuite dans *Web, XML, Java EE and OSGi Enterprise Development* et sélectionnez le paquet *Eclipse Web Developer Tools*.

Suivez les instructions de l'assistant d'installation et répondez aux questions qu'il vous pose. Après un redémarrage d'Eclipse, vous voilà en mesure d'éditer vos HTML directement dans Eclipse.

Figure A-33
Installation du plugin « Web Developer Tools »



Premier projet de test

Les différentes installations terminées, nous pouvons réaliser un premier petit projet de test. Cela permet de valider que tout fonctionne à la perfection. Ce petit projet n'a d'autre but que de réaliser un programme Python affichant toujours notre sempiternel texte « Hello World ! » à la console. Pour créer un nouveau projet, cliquez sur le menu *File > New > Project...* puis choisissez *PyDev Project*.

Après avoir cliqué sur *Next*, un assistant de création de projet se lance. À la première étape, il suffit de donner un nom au projet.

Cliquez alors sur *Finish*. Notre projet est créé et apparaît dans l'écran principal d'Eclipse.

Figure A-34
Création d'un projet de test

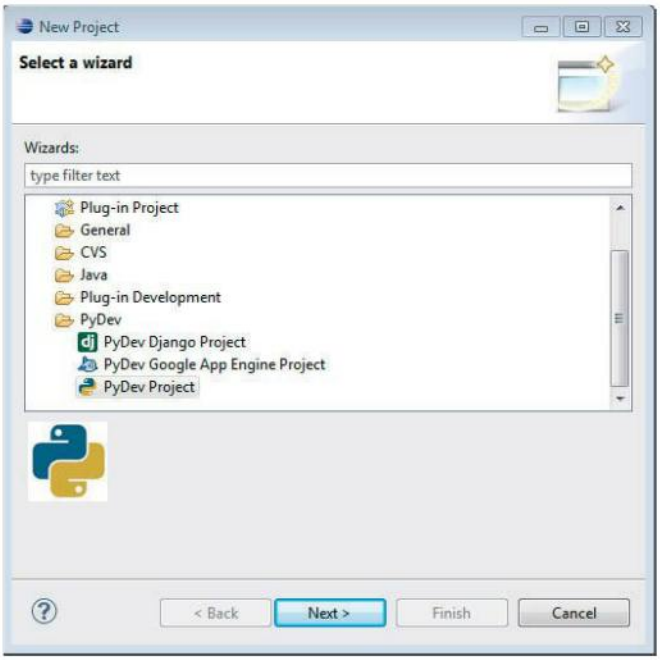


Figure A-35
Création d'un projet de test

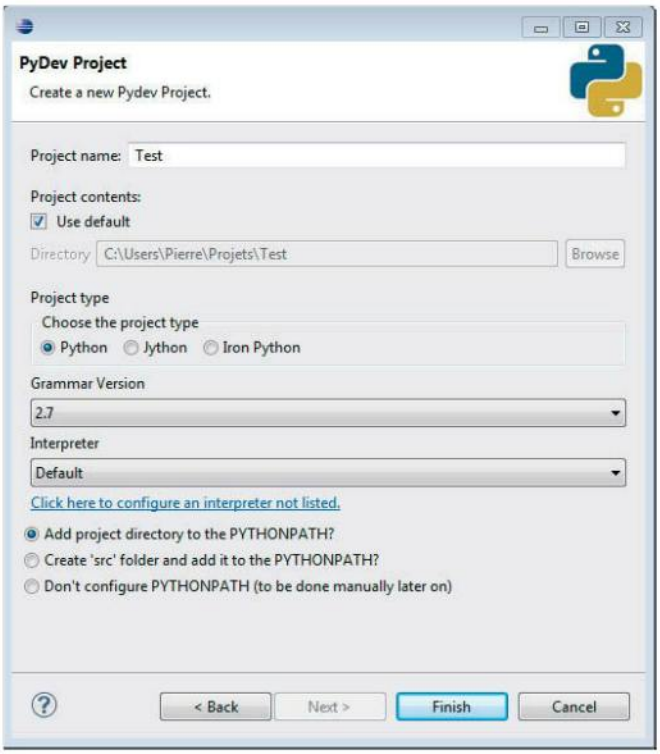
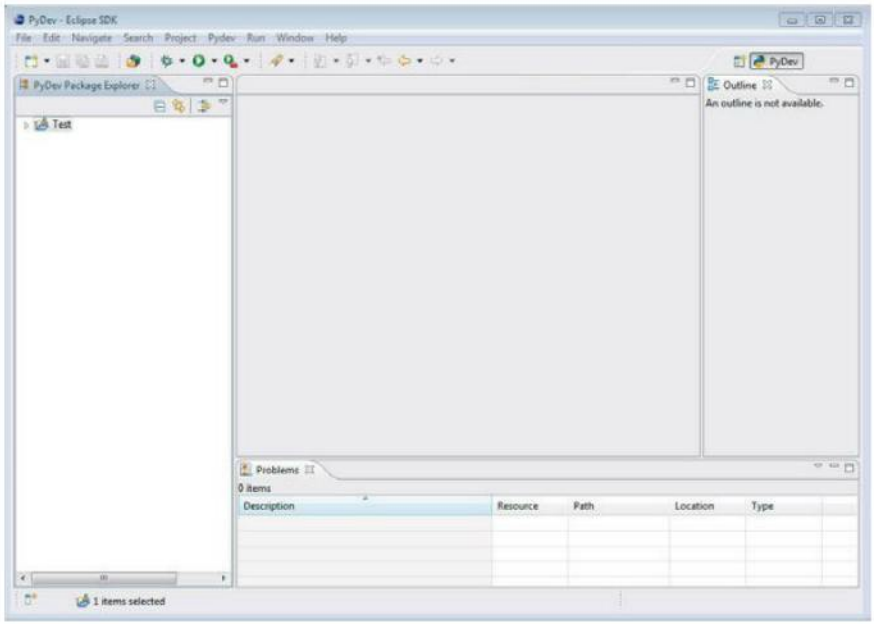
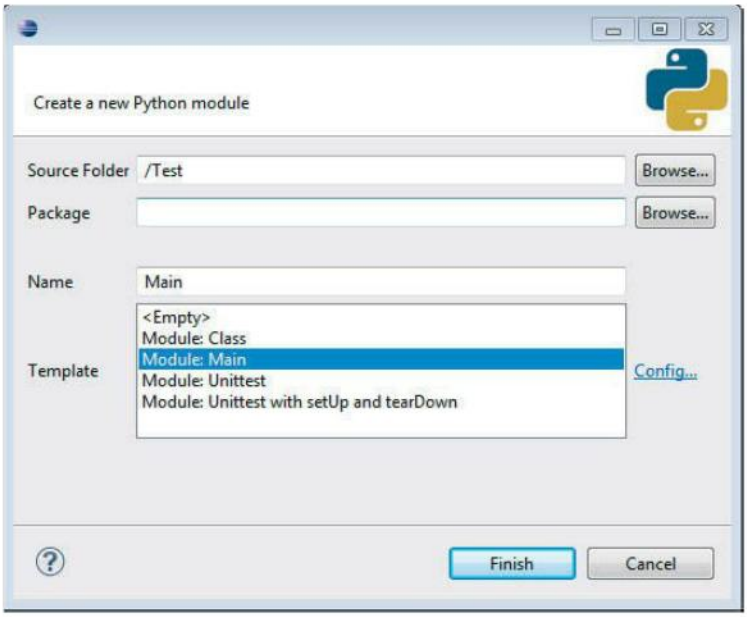


Figure A-36
Création d'un projet de test



Il reste à ajouter à notre projet un fichier `py` qui va contenir nos sources. Cliquez-droit sur le projet `Test` et choisissez *New > PyDev Module*. Une fenêtre apparaît dans laquelle on précise un nom pour le projet (`Main`) et un *template* de base ; dans notre cas, nous allons choisir le template *Module: Main*, car nous allons écrire notre code au niveau du point d'entrée du programme.

Figure A-37
Création d'un projet de test



Cliquez sur *Finish*, le fichier est alors créé. Supprimez-en tout le contenu, en particulier les commentaires ajoutés par défaut par Eclipse, car pour peu qu'ils contiennent des accents, Python plantera. Nous verrons comment gérer correctement les accents dans le chapitre 8. Cela nous permettra de garder le code proposé en standard dans ces fichiers.

Il reste à insérer le code suivant dans le fichier fraîchement créé :

EXEMPLE A.4 HelloWorld ! en Python

```
if __name__ == '__main__':
    print('Hello World !')
```

On peut maintenant exécuter le programme en utilisant l'icône *Run* représentée par un triangle blanc sur un rond vert. Une fois exécuté, le programme affiche « HelloWorld ! » à la console.

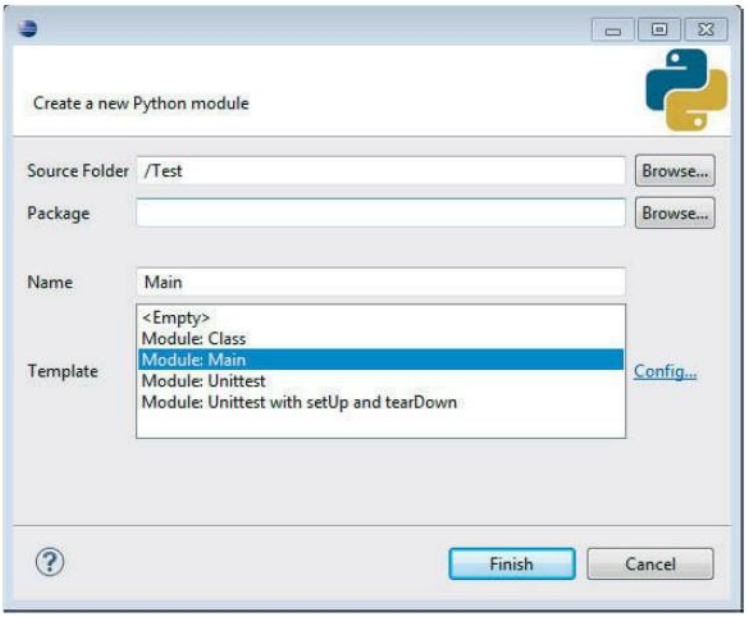


Figure A-38
Création d'un projet de test

Et voilà ! Nous avons créé un programme Python en utilisant Eclipse et son plug-in PyDev. Nous sommes prêts pour des projets plus complexes ayant recours au framework Django.

Ai-je bien compris ?

- Pourquoi faut-il installer Java alors que l'on va programmer en Python ?
- Eclipse est-il obligatoire pour développer un site en Django ?
- À quoi sert le plug-in Eclipse « PyDev » ?

Index

A

Access 131
Ajax (Asynchronous JavaScript and XML) 270

B

base de données 12
  nettoyer 234
base de données relationnelle 214
  clé primaire 114, 119
  lien 208
  relation 1-1 116
  relation 1-n 114
  relation n-n 117

C

cardinalité d'un lien 122
cas d'utilisation 104
CGI (Common Gateway Interface) 126, 131, 133, 145
cgi-bin 133, 139
classe 45
  attribut 45
  attribut statique 48
  constructeur 47
clé
  primaire 115, 129
cookie 233, 238
  durée de vie 235
CSS (Cascaded Style Sheets) 58, 60
  * (sélecteur) 83
  absolute 90
  active 82
  background-color 79
  block 88
  border 86
  border-width 79
  dimension des éléments 86
  display 98
  font-family 79
  height 79, 86
  hover 82
  inline 88
  link 82
  marges par défaut 95
  margin 86
  padding 86
  position 89
  positionnement par défaut 88
  propriété 79
  relative 89
  sélecteur 79
  sortir un élément du flux 89
  static 89
  top 89
  visited 82
  width 86

D

DHTML (Dynamic HTML) 96–97
diagramme de classes 202
Django 23, 289
  __startswith 212
  __str__ 218
  __unicode__ 218
  _set 213
  action 186, 188
  admin.py 215
  ajax/ 272
  all 248
  as_p 193
  as_table 193
  autodiscover() 215

blank 208, 211
BooleanField 192
clean 198, 273
cleaned_data 198
commit 222
configuration 157, 160, 168, 234–235
datetime 170
DEFAULT CONTENT TYPE 160
DEFAULT_CHARSET 157
delete 213
django-admin.py cleanup 234
EmailField 191
emplacement des fichiers projet 180
errorList 196
Field 273
filter 212
focusout 274–275
Form 191, 221
forms 190, 254
forms.py 221
formulaire 186, 188, 221, 225–227, 280
GET 187
get 212
hidden 229
HttpRequest 187
HttpResponse 154
HttpResponseRedirect 188
installation 297
is_bound 228
is_valid 195
message d'erreur 189
Model 203
model 150
modèle 202
ModelForm 220, 226, 263
models 204
models.ForeignKey 207
models.ManyToManyField 207
models.OneToOneField 207
models.py 203
none 239
PasswordInput 192
patterns 152
raise 198
ready 274–275
redémarrer l'application 169
redirection 186
render_to_response 167, 186
request 154, 187, 195
save 221
session 233
settings.py 157, 160, 180, 204, 215–216
super 198
Sync DB 205, 211, 241
template 150, 166
TEMPLATE_DIRS 168
try...except 273
tuple 152
urlpatterns 152
urls.py 152, 158, 171, 178, 215, 222, 250
valider un courriel 197
valider un formulaire 194
view 150
views.py 153, 158, 168, 172, 179
vue 151, 153, 192
XHTML 5 160
document HTML
encodage 65
structure 63

E

Eclipse 155, 289
installation 303
plug-in PyDev 305, 310
encodage
ISO 8859-1 66
UTF-8 66, 138, 157, 167
événement 96
exception 273
expression régulière 153

F

faille de sécurité 232
feuille de styles 78
fonction anonyme 275
formulaire 270
framework
Django 150

H

héritage 17, 120
HTML (HyperText Markup Language) 58–59, 126, 138, 145
  a 70
  action 76
  alt 72
  article 66
  aside 68
  attribut 62
  balise 60
  body 63
  change 96
  class 81
  click 96
  dblclick 96
  dl 259
  em 73
  encoding 65
  figcaption 72
  figure 72
  footer 68
  form 75
  formulaire 75
  h1, h2, h3 67
  head 63
  header 68
  height 72
  html 63
  id 80
  image 72
  img 72
  input 75
  lang 62
  li 71
  link 85
  liste 71
  method 76
  mise en évidence de texte 73
  mouseover 96
  name 76
  ol 71
  onclick 99
  p 66
  script 99
  section 66
  src 72
  strong 73
  style 83–84
  submit 75–76
  text 75
  title 61
  type 75
  ul 71
  value 76
  width 72
  XHTML 60
  xmlns 63
HTTP (HyperText Transfer Protocol)
  redirection 186
  requête 75, 187

I

Informix 131
instruction Python
  __init__ 45
  __str__ 47
  append 35
  class 45
  def...return 43
  del 36
  for...in 41
  if...elif...else 39
  if...else 37
  import 288
  indentation 38
  insert 35
  len 35
  print 31, 138
  self 47
  type 31
  while 40
instruction SQL
  DELETE FROM...WHERE 132
  INSERT INTO...VALUES 132
  SELECT...FROM...WHERE 131
  UPDATE...SET...WHERE 132
intégrité des données 233

interactivité client-serveur 268–269
Internet 4
istruction SQL
  ORDER BY 132

J
Java
  installation 302
JavaScript 58, 96, 270, 275, 280
  getElementById 98
jQuery 99, 229, 272, 274–276
  ajax 276
  manipulation d'élément 101
  prepend 283
  sélection d'élément 101

L
langage
  SQL (Structured Query Language) 131
langage de programmation
  Python 145, 288
  script 96
lien hypertexte 4, 60, 70

M
modèle de données 104, 113, 119, 122, 127
modèle Django
  accéder à des objets liés 213
  création 202
  créer un enregistrement 211
  récupérer plusieurs enregistrements 212
  récupérer un enregistrement 212
  superuser 206
  supprimer des enregistrements 213
  trier des données 212
  utilisation 211
MTV (Model Templates Views) 150, 166
MVC (Modèle Vue Contrôleur) 58, 150
MVC (Modèle-Vue-Contrôleur) 18

N
navigateur web 6, 11

O
Oracle 131

P
page web 4
  dynamique 10, 135
  passage de paramètres 13
  statique 134
PK (Primary Key) 116
port 162
port de communication 133
programmation
  orientée Objet 17
  procédurale 17
programmation Objet 45
  association entre classes 47
  héritage 51
  polymorphisme 53
programme modulaire 16
protection des pages 238
protocole HTTP (HyperText Transfer Protocol) 4, 7
Python 17, 26–27
  bibliothèque 54
  installation 292
  interpréteur 138
  ordre des classes 210
  script 126, 133

S
scenario d'un site 112
sécurité des données 233
serveur de bases de données 12
serveur web 6, 9
serveur web Python 133
session 231–233, 238
  configuration 234
  données 237
  utilisation 235
SGBD (Système de gestion de bases de données) 12
site web
  dynamique 9, 11, 13
  page de déconnexion 234
  page par défaut 250
  statique 5

SQL (Structured Query Language) 126, 131, 133, 145, 202
SQLite 129, 139, 214
  sqlite.db 206, 211, 217
Sybase 131

T

template
  {%block%}{%endblock%} 175
  {%extends%} 176
  {%for%}{%endfor%} 174
  {%if%}{%else%}{%endif%} 173
  {{current_date_time}} 170
  {{logged_user_name}} 166
  attribut de variable 172
  bodyId 177
  condition 173
  content 177
  élément d'une liste 172
  first 173
  genericPage 177
  headerContent 177
  héritage 174
  langage 172
  length 173
  lower 173
  principe 166
  title 177
  variable 172

U

UML (Unified Modeling Language) 20
URL (Uniform Resource Locator) 4, 8
use case 20–21, 105, 127

V

variable
  chaîne de caractères 33
  de classe 48
  déclaration 28
  dictionnaire 36
  entier 31
  liste 35
  locale/globale 44
  réel 32
  transtypage 29
  type 29
Von Neumann 29

W

Web 4
Web Developer Tools 167
wireframe 104, 106, 127
World Wide Web 4

X

XHTML 138