heig-vd
Haute Ecole d'Ingénierie et de Gestion
du Canton de Vaud

Présentation WEB

Django

Un Python sur la toile

Reynald Borer & Murielle Savary
classe IL2007

22 mai 2007

Table des matières

1 Introduction

Django est un framework web, écrit en langage Python, et dédié à la publication sur Internet. Un framework consiste en un ensemble de bibliothèques permettant le développement rapide d'applications abouties. Ces briques logicielles sont organisées pour être utilisées en interaction les unes avec les autres [7].

La création et l'utilisation de ces frameworks est issue du besoin de développer rapidement des applications, sans devoir constamment réinventer la roue. Ceci est d'autant plus vrai pour les sites web, car la plupart utilisent une base de données et gèrent des formulaires pour que les utilisateurs puissent interagir avec le site. Un développeur web doit donc à chaque nouveau site concevoir un système pour la gestion de la base de données, de même pour les formulaires. Par l'utilisation d'un framework mettant déjà ces fonctionnalités à disposition, il peut se concentrer uniquement sur le fonctionnement du site.

De plus, la plupart des frameworks utilisent un modèle de conception précis, facilitant encore plus le découpage d'une application entre son fonctionnement interne et son affichage. Grâce à cela, il devient très facile de mettre en place la logique métier, puis de se focaliser à la fin uniquement sur l'apparence de l'application.

Dans la suite de ce rapport, nous allons étudier le framework Django. Nous commencerons par un bref rappel de quelques notions du langage de programmation Python. Puis nous nous plongerons plus en avant dans les fonctionnalités offertes par le framework Django, tout d'abord en regardant le découpage de la logique métier qu'il propose, mais aussi de toutes les facilités qu'il met à disposition.

Finalement, nous tenterons de faire un bref comparatif avec le framework web le plus connu du moment, Ruby on Rails.

2 Rappels sur le langage Python

Afin de pouvoir appréhender au mieux Django, il convient d'avoir quelques notions en Python. Nous allons ici en résumer les principaux concepts, sans prétendre vouloir couvrir le langage de manière exhaustive. Nous ne pouvons que vous recommander de lire l'excellente référence *Dive into Python*, disponible gratuitement sur [2].

Python est un langage de programmation interprété et orienté objet. Il fonctionne selon un typage dynamique fort, dispose d'une gestion automatique de la mémoire et d'un système de gestion d'exceptions. Il est supporté par la plupart des plates-formes informatiques, dont entre autre Mac OS X, Linux et Windows. Il existe de plus une multitude de librairies qui font de Python un langage complet et offrant un développement rapide de scripts en tout genre, mais aussi d'applications complètes. Notons qu'il permet aussi d'effectuer une programmation de style fonctionnelle, en voiçi un exemple :

```
l = [x**2 for x in range(10)]
```

La syntaxe du langage est conçue pour être extrêmement lisible. C'est pourquoi là ou certains langages utilisent de la ponctuation, python utilise des mots-clés anglais. De plus, les blocs sont identifiés par l'indentation et pas par des accolades, tandis que la fin d'une ligne ne comporte aucun caractère spécial. Voici un bref comparatif avec du C, tiré de wikipédia :

Code C	*Code Python*

`int factorielle(int x) {`
`  if (x == 0)`
`  return 1;`
`  else`
`  return x * factorielle(x-1);`
`}`
	`def factorielle(x):`
	`  if x == 0:`
	`  return 1`
	`  else:`
	`  return x * factorielle(x-1)`

2.1 Types offerts par le langage

Python offre plusieurs types de données natifs. Il existe bien évidemment les types numériques de base, soit `int`, `long`, `float` et `complex`. Mais il existe aussi un ensemble de types sur lesquels il est possible d'itérer. Ces types sont les suivants :

- **list**, représentant une liste d'objets de taille variable, et dont le type des objets n'est pas fixé (exemple : `liste = ["a", "b", 123]`) ;
- **tuple**, une variante de liste qui n'est pas modifiable (exemple : `t = ("a", 123)`) ;
- **str**, représentant une chaîne de caractères ;
- **dict**, dictionnaire mettant en relation un objet (la clé) avec un autre (la valeur) ; notons que la clé doit être unique et que, à l'instar des listes, les types des ob-

jets valeurs peuvent être différents au sein d'un même dictionnaire. Exemple : `dict = {"key1": "value1", "key2": 123}`.

2.2 Modules, classes et méthodes

Dans la terminologie Python, un module représente simplement un fichier de code, contenant une ou plusieurs classes. Il existe deux moyens d'importer des modules externes : le premier permet simplement d'indiquer à Python dans quels fichiers chercher les méthodes, les appels s'effectuant par l'utilisation du préfixe du nom du module. Cette méthode est mise en place en utilisant :

```
import MonModule
```

La deuxième méthode permet quant à elle d'importer directement les classes et méthodes dans l'espace de nom local, permettant ainsi d'éviter l'utilisation du préfixe. De plus, il est possible de spécifier quelles classes nous souhaitons rendre visible. Le code suivant est utilisé pour cela :

```
from MonModule import MaClasse
```

En Python, tout est objet, que ce soit nos propres classes, des instances de classes, mais aussi les types de base et les fonctions. De plus, Python supporte l'héritage multiple mais ne permet pas de surcharger les opérateurs.

Par défaut, une classe met à disposition deux accesseurs pour ses attributs :

```
getattr(objet, 'nom de l'attribut')
setattr(objet, 'nom de l'attribut', nouvel_attribut)
```

Une classe se définit grâce au mot-clé `class`, l'héritage étant ensuite défini en utilisant le ou les noms des classes à hériter entre paranthèses, le tout séparé par des virgules le cas échéant.

Afin d'initialiser certaines données membres de la classe, il est possible de définir une méthode d'initialisation, appelée `__init__()`. Cette méthode n'est pas à considérer comme un constructeur, car lors de son appel implicite, l'objet est déjà créé. De même, comme il ne s'agit pas d'un constructeur, il convient d'appeler la méthode `__init__()` du parent le cas échéant. Consultez l'exemple de la figure 1 afin de voir sa syntaxe complète.

Les méthodes d'une classe Python ont toutes besoin d'un paramètre afin de se référer à l'objet courant, stocké dans le premier paramètre d'une fonction. Si nous regardons l'exemple ci-dessous, nous constatons que chaque méthode à comme premier paramètre `self`, représentant l'instance de la classe. Ce paramètre est automatiquement inséré lors

d'appels de méthodes par Python, et son nom n'est qu'une convention, vous pouvez donc le nommer comme bon vous semble.

Notons finalement que pour définir une classe vide, il convient de lui adjoindre le mot-clé `pass`, qui correspond à une instruction vide (équivalente à `{}` en C).

```
class Afficheur:
   def __str__(self):
      return "Je suis un " + self.__class__.__name__
   def __repr__(self):
      return repr(self.__str__())

class Forme:
   def __init__(self,couleur="blanc"):
      self.couleur=couleur
   def __str__(self):
      return "de couleur " + self.couleur

class Carre(Forme,Afficheur):
   def __init__(self, cote=0):
      Forme.__init__(self, "rouge")
      self.cote = cote
   def __str__(self):
      return Afficheur.__str__(self) + " " + Forme.__str__(self) +\
         " et de cote " + str(self.cote)

class Vide(Forme,Afficheur):
   pass

if __name__ == "__main__":
   tuple = (Carre(12), Forme(), Carre(), Vide())
   for item in tuple:
      print item
```

Fig. 1 – Exemple complet de code Python

3 Présentation du framework

Le développement de Django a débuté durant l'année 2003, dans le but de gérer efficacement plusieurs sites Internet d'actualité d'une société de presse américaine. Les développeurs de ces sites avaient besoin de pouvoir créer facilement et rapidement des sites s'interfaçant avec une base de données, ce qui donna naissance à ce framework. Développé d'abord en interne dans cette société, le framework fut finalement mis à disposition du grand public en juillet 2005, sous licence BSD, les développeurs ayant déterminé qu'il était assez mûr pour une utilisation grand public. Grâce à cette action, le framework reçoit à l'heure actuelle beaucoup de retour d'expérience et d'améliorations de la part de la communauté. Sa version actuelle est la 0.96, il est donc toujours en développement, tout en étant totalement utilisable. Ce framework tire son nom du défunt guitariste de jazz *Django Reinhardt*, considéré comme le meilleur guitariste de son époque.

3.1 Notion de projet

Django fonctionne selon un système de projets, chaque projet contenant ensuite une ou plusieurs applications (même s'il est possible d'utiliser une partie des fonctionnalités de Django sans créer d'application). Ces deux notions sont détaillées ci-après.

Un projet est une instance d'un certain nombre d'applications avec une configuration associée. La configuration, effectuée dans le fichier `settings.py` du projet, contient entre autre la liste des applications actives, les informations de connexion à la base de données, ainsi que le chemin d'accès pour les templates. Le projet met aussi en place le routage des URLs par l'intermédiaire du fichier `urls.py`.

Une application, quant à elle, est un ensemble de fonctionnalités Django qui sont portables (qui peuvent être utilisées dans un autre projet), comprenant généralement des modèles et des vues. Notons qu'une application peut être utilisée dans plus d'un projet, et qu'elle peut spécifier son propre routage des URLs, pour autant que le projet dans lequel elle est utilisée lui délègue ce travail.

3.2 Modèle de conception MVC

Django utilise le modèle de conception *Modèle - Vue - Contrôleur*. Pour rappel, cette architecture permet de découper les données, représentées par la couche modèle, de leur présentation, soit la couche vue. Le contrôleur quant à lui s'occupe de la logique métier, en synchronisant les événements afin de mettre à jour le modèle ou la vue.

Cependant, la terminologie utilisée par Django diffère quelque peu du modèle MVC. En effet, d'après les développeurs, les noms standards des couches sont discutables. En particulier, la vue ne représente pas forcément comment les données sont affichées, mais quelles données sont affichées. Dans le cas de Django, la vue est donc représentée par une

fonction Python appelée pour une URL particulière. L'affichage à proprement parler est quant à lui défini dans un template, qui est séparé de la vue afin de séparer le contenu de sa présentation. Et qu'en est-il du contrôleur dans cette architecture ? Le contrôleur est très certainement Django lui-même car, par l'intermédiaire de la configuration du routage des URLs, les requêtes sont redirigées vers la bonne vue.

C'est ainsi que Django peut être appelé un framework MTV, pour Modèle - Template - Vue, ce découpage étant plus approprié. Mais le fonctionnement général est très proche, voir identique, aux frameworks qui utilisent la terminologie MVC. La correspondance entre ces deux modèles est représentée sur la figure 2.

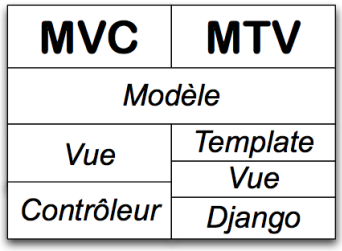

Fig. 2 – Modèle MVC adapté à Django

3.3 Philosophie du framework

La création du framework Django se base sur les philosophies suivantes :

Développement rapide
L'un des objectifs d'un framework consiste à simplifier la construction d'une application web. Le framework Django ne déroge pas à cette règle et permet de mettre en place une application web en quelques heures.

Couplage faible
Django favorise le couplage faible, c'est-à-dire que les divers composants de l'application ne savent rien des autres composants en dehors du strict nécessaire. Les composants sont ainsi interchangeables et communiquent entre eux via des API claires et concises. Il est donc tout à fait possible d'utiliser un modèle de données et un langage de templates différents de ceux proposés par Django.

Code concis
Une application Django ne nécessite que très peu de code. En cela, le framework sait exploiter les nombreux avantages fournis par le langage Python, tel que l'introspection.

DRY : *Don't Repeat Yourself* [1]
Django permet de s'extirper du piège de la redondance. Chaque objet n'existe qu'à un seul endroit, le framework effectue toutes les déductions nécessaires à partir d'un minimum d'informations.

Explicite est bien mieux qu'implicite
Le framework gère les composants de manière explicite de sorte que le développement est plus clair. Il n'y a, derrière son fonctionnement, aucune magie qui rendrait difficilement compréhensible ses différents aspects.

Les modèles incluent toute la logique de domaine appropriée
Les modèles encapsulent chaque aspect d'un objet, respectant ainsi le design pattern *Active Record* de Martin Fowler[2]. Les options d'administration spécifiques au modèle sont donc inclues dans le modèle lui-même, de même que les données relatives à ce modèle.

SQL efficace
L'API permettant de gérer la base de données optimise les requêtes SQL de manière à ce qu'elles soient effectuées le moins souvent possible. La couche d'abstraction pour la base de données n'est fournie qu'à titre de raccourci, et ne doit en aucun cas limiter les possibilités d'exécuter des requêtes SQL directement.

Gestion des URLs
Le framework simplifie la gestion des URLs et permet d'afficher un chemin clair. Le système d'URLs de Django permet également à une URL d'une même application d'être différente selon son contexte, de même, plusieurs URLs peuvent utiliser la même vue.

[1] `http://c2.com/cgi/wiki?DontRepeatYourself`
[2] `http://www.martinfowler.com/eaaCatalog/activeRecord.html`

4 Les modèles

4.1 Définition

Les modèles sont des ORM (Object-Relational Mapper) qui contiennent tous les champs et les comportements des données à stocker. En général, chaque modèle correspond à une seule table dans la base de données.

Django suit le principe DRY, le modèle de données n'est défini qu'à un seul endroit et tout le reste est dérivé automatiquement. Un modèle Django a les équivalences suivantes dans un SGBD :

Modèle Django	SGBD
classe	table
objet	ligne d'une table
champ	colonne d'une table

Chaque modèle est représenté par une classe Python qui étend la classe de base dénommée `django.db.models.Model`. Les modèles Django possèdent la particularité de stocker les métadonnées dans une classe interne appelée `Meta` tandis que les métadonnées propres à l'administration du site Django sont mises dans une classe interne `Admin`.

À partir des modèles, Django crée le schéma de la base de données ainsi qu'une API d'accès à la base de données en Python permettant de manipuler les objets sans utiliser de SQL. La souplesse de Django permet également d'utiliser d'autres librairies que celle fournie pour accéder à la base de données.

Voici un modèle permettant de définir une personne avec son nom et son prénom :

```
from django.db import models

class Personne(models.Model):
   nom    = models.CharField(maxlength=30)
   prenom = models.CharField(maxlength=30)
```

`nom` et `prenom` sont des champs du modèle. Chaque champ est spécifié comme un attribut de classe et correspond à une colonne dans la base de données.

La commmande

```
python manage.py syncdb
```

crée le modèle dans la base. Le code SQL exécuté est le suivant :

```
CREATE TABLE monapp_personne (
    "id"     serial      NOT NULL PRIMARY KEY,
    "nom"    varchar(30) NOT NULL,
    "prenom" varchar(30) NOT NULL )
```

Les instructions générées nécessitent quelques remarques :

- Le nom de la table `monapp_personne` est dérivé automatiquement mais peut être renommé.
- Un champ `id` est automatiquement ajouté, ce comportement peut toutefois être modifié. Par défaut, Django intègre dans chaque modèle le champ

  ```
  id = models.AutoField(primary_key=True)
  ```

  dont le résultat produit une clé primaire auto-incrémentée. L'utilisateur peut choisir sa propre clé primaire en spécifiant `(primary_key=True)` dans le champ dont il veut en faire une clé. Chaque modèle ne contient qu'un seul champ avec l'instruction `(primary_key=True)`, ce qui permet à Django de déterminer si une colonne `id` est nécessaire ou si une clé primaire est déjà explicitement définie ;
- le code SQL donné dans l'exemple ci-dessus utilise la syntaxe de PostgresSQL, mais Django supporte également MySQL et SQLite.

Les modèles sont stockés dans un fichier `models.py` et sont représentés par des classes Python. Ce fichier est propre à chaque application.

4.2 Champs

Les champs définissent les champs requis dans la base de données. Ils sont spécifiés par des attributs de classe. Ces champs permettent de définir le type des données (entier, texte, ...), les options du champs (null, clé primaire, ...), ainsi que les relations entre les tables.

```
class Sportif(models.Model):
   nom    = models.CharField(maxlength=30)
   prenom = models.CharField(maxlength=30)
   sport  = models.CharField(maxlength=30)
```

4.2.1 Restrictions

Il existe deux restrictions dans la nomenclature des champs, la première est inhérente à l'utilisation de Python, les mots réservés en Python ne peuvent pas être utilisés comme nom d'un champ. La seconde restriction est liée à la façon dont Django parcourt les requêtes à la recherche d'une syntaxe connue, le double underscore est proscrit. Ces

limitations peuvent toutefois être contournées car, avec Django, le nom des champs ne doit pas nécessairement correspondre au nom des colonnes de la base [3].

4.2.2 Types

Chaque champ est représenté par l'instance d'une classe `models.*Field` permettant ainsi à Django de savoir quel type de donnée est contenu dans un champ.

Django possède de nombreux types de données permettant d'identifier un champ, quelques uns sont détaillés ci-dessous, le lecteur se reportera à la documentation de référence pour de plus amples détails.

Type	Explication
`IntegerField`	Pour représenter un entier.
`AutoField`	Un `IntegerField` automatiquement incrémenté.
`BooleanField`	Une variable booléenne.
`CharField`	Un champ texte.
`TextField`	Un champ texte de plus grande taille.
`DateField`	Un champ date.
`EmailField`	Un `CharField` dont le contenu est vérifié de sorte à ce qu'il corresponde à adresse email valide.
`URLField`	Vérifie que l'url existe si le chargement de la page ne produit pas une réponse 404.

4.2.3 Options

Les options permettent de préciser si un champ peut avoir une entrée `null` dans la base de données, ou s'il ne peut prendre de valeurs que dans une liste de choix. Il existe de nombreuses options sur les champs, la documentation de référence les détaille toute. C'est dans les options du champ qu'il est également possible de déclarer une clé primaire ou de fixer une valeur par défaut au champ.

4.2.4 Relations

Django offre des méthodes afin de gérer les trois types de relations dans une base de données : un-à-un, un-à-plusieurs, plusieurs-à-plusieurs.

Un-à-un

Une relation un-à-un entre deux modèles (tables) est établie à l'aide de `OneToOneField`. Cette commande s'utilise comme un type de champ en l'incluant comme un attribut

[3] L'option de champ `db_column` permet de préciser le nom de la colonne de la base de données associé au champ.

dans le modèle, et requiert comme argument la classe à laquelle le modèle est lié.

Exemple Dans cet exemple, un restaurant ne peut se trouver qu'à un seul site et un site ne peut contenir que un seul restaurant.

```
from django.db import models

class Site(models.Model):
   nom     = models.CharField(maxlength=50)
   adresse = models.CharField(maxlength=80)

class Restaurant(models.Model):
   nom  = models.CharField(maxlength=50)
   site = models.OneToOneField(Site)
```

Un-à-plusieurs

Une relation un-à-plusieurs se définit à l'aide de `ForeignKey`. Cette méthode prend en argument la classe qui la lie au modèle.

Exemple Un fabriquant d'ordinateurs construit plusieurs ordinateurs tandis qu'un ordinateur n'est construit que par un seul fabriquant :

```
class Fabriquant(models.Model):
    # ...

class Ordinateur(models.Model):
    fabriquant = models.ForeignKey(Fabriquant)
    # ...
```

Il est préférable, mais pas exigé, d'utiliser le nom de la classe liée en minuscule comme nom du champ qui y fait référence (`fabriquant` qui lie la classe `Fabriquant`).

Plusieurs-à-plusieurs

La relation plusieurs-à-plusieurs s'obtient par `ManyToManyField`. L'argument de cette méthode est la classe à laquelle le modèle est lié.

Exemple Une pizza est composée de plusieurs ingrédients et un ingrédient compose plusieurs pizzas :

```
class Ingredient(models.Model):
    # ...

class Pizza(models.Model):
    ingredients = models.ManyToManyField(Ingredient)
    # ...
```

Il est préférable, mais pas exigé, d'utiliser le nom de la classe liée sous sa forme plurielle afin de nommer le champ qui y fait référence dans une relation plusieurs-à-plusieurs.

4.3 Administration

Les modèles contiennent des informations supplémentaires destinées à l'administration. Ainsi, par le simple ajout d'une classe interne `Admin` dans le modèle cela précise à Django comment afficher le modèle dans le site d'administration. Les options disponibles sont décrites dans la documentation de référence. Cette classe interne est optionnelle.

4.4 Méthodes

Les modèles peuvent contenir des méthodes afin de fournir des fonctionnalités supplémentaires aux objets. Cette conception s'intègre parfaitement dans l'idée de conserver la logique métier : le modèle.

Les méthodes sont écrites en Python dans la classe associée au modèle. Chaque modèle peut ensuite appeler ses méthodes implémentées.

Il n'est cependant pas nécessaire d'écrire les méthodes de base qui sont fournies par Django, ainsi, la création, la suppression et l'enregistrement des objets sont disponibles sans écrire de code.

4.5 Génération du modèle depuis la base

Il est parfois nécessaire d'intégrer Django dans une base de données supportée déjà existante, ceci ne pose pas de problème. En effet, le framework possède un utilitaire pour créer le modèle par introspection d'une base existante. Il convient cependant de bien relire le modèle généré, surtout car les références ne sont pas forcément toutes détectées.

```
python manage.py inspectdb
```

5 Les vues

Dans la philosophie Django, une vue est un type de page web servant généralement à une action spécifique, et utilisant un template de présentation spécifique. Les vues sont toute des fonctions Python, stockées dans le fichier `views.py` du répertoire de l'application.

Cependant, là ou d'autres frameworks font un lien direct entre le nom de la vue et l'adresse à utiliser, Django utilise un autre système, dans lequel il nous faut spécifier, pour chaque URL, quelle vue y associer.

5.1 Gestion des URLs

Pour cela, avant de commencer à développer le code pour les vues, il convient en premier lieu de définir clairement les fonctionnalités de l'application, ainsi que les URLs d'accès à ces diverses fonctionnalités. En effet, Django permet de définir des adresses propres et lisibles, pour lesquelles il est possible de changer en cours de route de vue, tout en conservant la forme de l'URL. De récentes études ont démontré à quel point des adresses propres (soit sans virgules et autres points d'interrogations) et persistantes étaient importantes sur le web[4].

La gestion des URLs s'effectue dans le fichier `urls.py` du projet, lequel peut ensuite inclure d'autres fichiers spécifiques à une application. De même, l'emplacement et le nom de ce fichier peuvent être modifiés dans le fichier `settings.py` du projet. Ce fichier contient un attribut `urlpatterns`, résultant de l'appel de la méthode `patterns()`, laquelle prend en paramètre un ensemble de règles. Chaque règle est de la forme suivante :

```
(expression régulière, fonction Python [, dictionnaire optionnel])
```

Les adresses sont donc vérifiées selon des expressions régulières, lesquelles sont compilées lors du premier accès au fichier `urls.py` afin de gagner en performances. Ces expressions régulières permettent non seulement de définir la forme des adresses, mais aussi d'associer une partie de l'adresse à une variable utilisable dans la vue. Considérons l'exemple suivant :

```
from django.conf.urls.defaults import *
urlpatterns = patterns('',
    (r'^blog/$', 'monprojet.blog.views.index'),
    (r'^blog/details/(?P<post>\w+)/$', 'monprojet.blog.views.detail'),
    (r'^blog/(?P<year>\d{4})/$', 'monprojet.blog.views.year'),
    (r'^blog/(?P<year>\d{4})/(?P<month>[a-z]{3})/$', 'monprojet.blog.views.month'),
)
```

[4] `http://www.w3.org/Provider/Style/URI`

Cet ensemble de règles définit un système simple d'affichage d'un blog. Lorsque l'adresse est `/blog/`, la vue `index` est appelée, afin d'afficher la page d'accueil du blog. Notons que ni le protocole ni le domaine ne sont à spécifier dans les règles (pas de `http://www.test.com/blog/`), et que le caractère / de fin est obligatoire.

Si par contre l'adresse est `/blog/details/why-django-rocks`, la vue `detail` est appelée, avec comme paramètre `post='why-django-rocks'`. L'expression régulière `\w+` définit donc qu'il doit y avoir un ou plusieurs caractères alphanumériques, tandis que `?P<post>` permet d'indiquer à Django de passer cette partie de l'url dans le paramètre `post` de la vue. De même, `\d{4}` précise qu'il doit y avoir 4 caractères numériques, ce qui permet d'afficher les archives annuelles ou mensuelles du blog de manière très simple. Consultez [8] pour obtenir la syntaxe complète des expressions régulières en Python.

5.2 Fonction représentant une vue

Une vue reçoit en premier paramètre un objet de type `HttpRequest`, puis, selon la configuration des URLs, zéro ou plusieurs paramètres. Elle retourne à la fin de son traitement un objet de type `HttpResponse`. À cette fin, il est possible soit de retourner un nouvel objet de ce type, contenant le texte à rendre au client, soit, en utilisant la méthode `django.shortcuts.render_to_response()`, de retourner une réponse en utilisant le système de template, présenté dans le prochain chapitre.

Django met à disposition un grand nombre de fonctions d'aide, permettant par exemple de retourner une page web par l'intermédiaire d'un template, mais aussi de faire une recherche dans un modèle en fonction d'un paramètre passé et de retourner une erreur 404 si aucune donnée concernant la recherche n'existe. L'exemple suivant montre l'utilisation de ces fonctions.

```
from django.shortcuts import render_to_response, get_object_or_404
from mysite.polls.models import Poll

def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p})
```

Ce code prend en paramètre un numéro de sondage, celui-ci ayant été défini dans la gestion des URLs. Il vérifie ensuite si le sondage existe dans le système, et dans l'affirmative l'affiche par l'intermédiaire du template `polls/detail.html` en lui passant un dictionnaire contenant la référence du sondage en paramètre. Si par contre le sondage n'existe pas, alors une erreur 404 est levée, et une page d'erreur est affichée. Il est bien évidemment possible de créer sa propre page d'erreur.

5.3 Vues génériques

Il existe beaucoup de vues que nous répétons régulièrement, comme par exemple pour afficher une liste d'objets ou bien leurs détails, gérer l'affichage d'archives annuelles,

mensuelles et journalières d'articles, etc. De même, la création, l'édition et la suppression d'objets divers reviennent fréquemment sur un site web.

C'est pour cela que Django met à disposition un ensemble de vues génériques permettant d'effectuer toutes ces opérations sans devoir écrire de code Python. Leur mise en place est simple, il suffit en premier lieu d'identifier les URLs correspondant à ces vues, puis de leur affecter la vue générique dans le contrôleur des URLs. Chaque vue prend en paramètre au minimum un dictionnaire contenant le modèle sur lequel travailler, mais il est possible de lui adjoindre d'autres informations. Il est par exemple possible d'ajouter des paramètres récupérés depuis l'adresse, mais aussi d'autres informations tel qu'une adresse de redirection, des détails de pagination, et si la page doit s'afficher même si aucun objet n'existe dans le modèle. De plus, chaque vue se voit dotée d'un template propre pour l'affichage.

Si nous reprenons notre exemple précédent avec le blog, le gestionnaire d'url pourrait se présenter de la façon suivante :

```
from django.conf.urls.defaults import *

info_dict = { 'queryset': Article.objects.all() }

urlpatterns = patterns('django.views.generic.date_based',
    (r'^blog/$', 'archive_index', info_dict),
    (r'^blog/details/(?P<post>\w+)/$', 'object_detail', \
            dict (info_dict, slug_field='post')),
    (r'^blog/(?P<year>\d{4})/$', 'archive_year', info_dict),
)
```

Les noms des templates peuvent être spécifiés, sinon ceux par défaut sont utilisés, se basant sur le nom du modèle et de la vue. Dans notre cas, il s'agit, par ordre d'apparition des règles, de `article_archive.html`, `article_detail.html` et `article_archive_year.html`.

Pour plus d'informations au sujet des vues génériques, nous vous invitons à consulter la documentation sur ce sujet, à l'adresse [3]. Celle-ci contient l'ensemble des vues génériques disponibles, avec pour chacune les paramètres acceptés ainsi que le nom du template utilisé par défaut. De même, n'hésitez pas à consulter le site officiel ([5]) pour plus d'informations sur le système des vues.

6 Les templates

Les templates sont intimement liés aux vues, ils s'occupent uniquement de la représentation des données de la vue pour l'utilisateur. À cette fin, les développeurs de Django ont créé leur propres balises, afin qu'il soit possible de créer les templates sans avoir à connaitre le langage Python.

Un template est un simple fichier texte qui permet de générer n'importe quel format texte (HTML, XML, CVS, etc.). Il contient des variables qui prennent une valeur lors de l'évaluation du template et des tags qui contrôlent la logique du template.

6.1 Variables

Une variable s'utilise de la manière suivante : `{{ variable }}`.

Lorsque le moteur du template rencontre une variable, il l'évalue et la remplace par sa valeur. Ainsi, si une variable est déclarée comme `{{ personne.nom }}`, son évaluation aura pour effet de remplacer la variable par l'attribut `nom` contenu dans l'objet `personne`. Si la variable utilisée n'existe pas, le système de template lui attribue une valeur par défaut (chaîne de caractères vide).

6.2 Filtres

Les filtres permettent de modifier l'affichage d'une variable. Un filtre se présente sous la forme : `{{ nom|lower }}`. Ce filtre a pour effet d'afficher une variable `nom` qui après son passage dans le filtre `lower` se retrouve en minuscule. Django possède de nombreux filtres définis, la documentation de référence les présentent tous.

6.3 Tags

Les tags sont de la forme : `{% tag %}`. Bien que presque de la même forme qu'une variable, ils sont plus complexes puisqu'ils sont responsables de la logique du template. Certains tags permettent de présenter le texte en sortie tandis que certains contrôlent les flux à l'aide de boucle. Les tags offrent également la possibilité de charger des informations dans le template afin de les utiliser via des variables.

Il existe de nombreux tags définis, ces derniers sont présentés dans la documentation de référence de Django. Il est également possible de créer ses propres tags sous la seule condition de savoir écrire du code Python.

Exemple Afficher le nom de tous les sportifs inscrits dans la liste `sportif_list` :

```
<ul>
{% for sportif in sportif_list %}
    <li>{{ sportif.nom }}</li>
{% endfor %}
</ul>
```

6.4 Héritage de templates

C'est l'une des parties les plus puissantes du moteur de templates de Django, mais c'est également la plus complexe. L'héritage de template permet la création d'un squelette de template contenant tous les éléments communs du site tout en définissant des blocs redéfinissables par ses enfants. Cette rédéfinition par les héritiers n'est toutefois pas obligatoire.

Le template parent indique au moteur de template quelles parties peuvent être modifiées par ses enfants à l'aide des balises `{% block nomblock %}` et `{% endblock %}`.

Exemple Le template `base.html` ci-dessous définit un document HTML de base. Les trois balises `{% block titre %}`, `{% block menu %}` et `{% block contenu %}` établissent les parties redéfinissables du template :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <link rel="stylesheet" href="style.css" />
    <title>{% block titre %}Mon super site de base{% endblock %}</title>
</head>

<body>
    <div id="sidebar">
        {% block menu %}
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/blog/">Blog</a></li>
        </ul>
        {% endblock %}
    </div>

    <div id="content">
        {% block contenu %}{% endblock %}
    </div>
</body>
</html>
```

Le template enfant hérite de `base.html` à l'aide de la balise `{% extends "base.html" %}`. Cela permet au moteur de template de localiser le template parent. Lors de l'évaluation du parent, le moteur remarque les trois balises de bloc et remplace ces blocs par le contenu du template enfant. Dans cet exemple, le template enfant redéfinit le titre, et le contenu (inexistant chez son parent) mais pas le menu :

```
{% extends "base.html" %}

{% block titre %}Mon super blog{% endblock %}

{% block contenu %}
{% for entry in blog_entries %}
    <h2>{{ entry.title }}</h2>
    <p>{{ entry.body }}</p>
{% endfor %}
{% endblock %}
```

Il existe de nombreuses manières d'exploiter cette capacité d'héritage, l'une des plus répandues est l'approche en trois niveaux suivante :

- création d'un template de base `base.html` qui maintient le *look-and-feel* du site ;
- création d'une base de template `base_nomsection.html` pour chaque section du site. Par exemple, `base_news.html`, `base_sports.html`. Ces templates héritent du premier template, `base.html`. L'apparence et la cohérence du site sont ainsi conservées, tout en permettant l'incorporation d'un style spécifique aux sections ;
- création de templates individuels pour chaque type de page (page d'articles, blog, etc.). Ces templates héritent, quant à eux, du template de la section appropriée.

L'un des gros avantages de cet héritage de templates et de permettre la réutilisabilité du code au maximum. De plus, il rend la présentation du site entièrement modulaire en permettant l'ajout d'éléments de façon très simple.

6.5 Autres librairies de tags et de filtres

Django offre la possibilité de charger plusieurs autres librairies de filtres et de tags à l'aide de la balise `{% load %}`. Il est également possible de créer ses propres librairies en Python. La documentation de Django fournit toutes les indications tant sur les librairies existantes que sur leur création.

De même, il est possible de se passer totalement du système de templates fourni, et d'utiliser une des nombreuses autres librairies Python pour gérer cette partie de la vue.

7 Les plus de Django

Django fournit, en plus du framework de base, un ensemble de fonctionnalités supplémentaires dédiées à certaines tâches répétitives de la conception de sites web. Ces utilitaires se retrouvent dans deux catégories : les applications à proprement parler, dont nous connaissons déjà le fonctionnement, mais aussi les middlewares (ou plugins). Nous allons détailler ces deux catégories ci-après.

7.1 Les applications

Django vient avec un ensemble d'applications complètes qu'il est possible d'utiliser dans notre projet. Le meilleur exemple reste sans doute l'interface d'administration, qui est une application séparée capable d'interagir avec les modèles de données de notre application. Cette interface ne doit pas être confondue avec des possibilités de création, affichage, modification et suppression de contenu, qui peuvent être réalisées par l'intermédiaire des vues génériques. Il est donc tout à fait possible de ne pas utiliser cette interface, mais si nous l'utilisons, des possibilités de gestion des modèles sont mises en place très rapidement.

Afin d'activer une application, il convient de modifier le fichier `settings.py` du projet pour le renseigner sur les applications actives du projet.

En plus de cette interface d'administration, Django met plusieurs autres applications à disposition. Citons entre autre :

- **un système d'authentification**, permettant de gérer facilement des utilisateurs et des groupes, avec la possibilité de définir des droits et l'envoi de messages personnels ;
- **un système de commentaires**, à utiliser par exemple dans le cadre d'un blog ;
- **un créateur de flux RSS**, permettant très facilement d'extraire un flux RSS depuis son ou ses modèles ;
- **une mise en forme du texte**, système permettant à partir d'un langage avec des balises simples, de le transformer en HTML ;
- et bien d'autres... consultez le site officiel [5] pour avoir la liste complète.

Ces applications permettent bien souvent de mettre très facilement en place un système qui aurait pris beaucoup plus de temps à développer si ces facilités n'étaient pas disponibles.

7.2 Les middlewares

Les middlewares sont une autre possibilité d'ajouter des fonctionnalités à Django. Ces plugins permettent d'effectuer des actions directement sur les requêtes et les réponses, avant que celles-ci n'arrivent à la vue respectivement au client. Pour activer un middleware, il faut modifier le fichier `settings.py` du projet uniquement, rien n'est à modifier

du côté du code de l'application.

Les middlewares fourni par Django couvrent entre autre :

- **des possibilités de cache**, afin d'éviter par exemple de surcharger la base de données ;
- **la compression des pages**, pour accélérer le téléchargement de ces dernières chez le client ;
- **une gestion des sessions**, pour gérer les données envoyées et reçues par les cookies ;
- **l'authentification par HTTP** ;
- **un système de transactions**, permettant, si une vue se termine avec succès de terminer la transaction sur la base de données, et de l'annuler en cas de problème ;
- et bien d'autres...

De plus, il est tout à fait possible de développer ses propres middleware, leur création étant très simple. Nous vous invitons à consulter la documentation officielle [4] pour plus de détails sur ce sujet.

7.3 Les librairies

Mais la grande force de Django est aussi de permettre l'utilisation de toutes les librairies du langage Python, sans aucune restriction.

Notons que Django met à disposition une librairie extrêmement pratique dans le cadre des applications web, à savoir `django.newforms`, permettant de gérer très facilement un formulaire HTML. En effet, cette librairie utilise une syntaxe très proche des modèles, et permet très simplement de générer le formulaire HTML sous diverses présentations, stocker le résultat et indiquer ensuite si celui-ci est valide par rapport aux types de données attendus pour les champs.

8 Comparatif avec Rails

Nous allons maintenant tenter de faire un petit comparatif objectif entre Django et Ruby on Rails. Celui-ci se base sur notre faible expérience de ces deux frameworks, mais aussi sur les avis d'autres développeurs, évoqués entre autre sur [1] et [6]. Ces deux frameworks sont très proches en terme de fonctionnalités, et se basent les deux sur des langages orientés objets à la syntaxe très explicite.

La documentation semble être ce qui pèche le plus pour Rails, en comparaison avec la documentation officielle de Django. En effet, pour apprendre Rails, il est quasiment nécessaire d'acheter un livre, seule source d'information. Django au contraire met à disposition une quantité étonnante de documentation et tutoriaux, et il existe même un livre, en cours de rédaction, qui est librement disponible sur le web ([9]).

Au niveau du fonctionnement du framework, Rails semble avoir un côté très magique. En effet, nous n'avons pas à nous soucier des URLs, et on obtient très rapidement un résultat. Par contre, le système proposé par Django semble être beaucoup plus puissant, car c'est au développeur de décider de la forme de ses adresses. Il est par contre évident que cette méthode est légèrement plus compliquée à mettre en place.

Le support d'AJAX dans Rails est un plus, celui-ci n'étant pas supporté par Django, en tout cas pas de manière officielle. L'intégration poussée d'AJAX dans Rails permet de l'utiliser très facilement, sans connaître le javascript.

La gestion de la base de données est aussi différente. Là où Rails vous force à définir auparavant vos tables, et donc à connaitre le SQL, Django vous permet de n'utiliser que du Python pour cela. De plus, l'approche Rails de vérifier la structure de la base de données à chaque appel est certainement moins performante que le modèle de Django.

La gestion des templates de Django semble être plus propre et mieux pensée, car il est impossible d'utiliser du code Python dans un template. Cet avis est cependant totalement subjectif.

L'interface d'administration de Django est clairement un plus, même si son but n'est pas de fournir une interface répondant à tous les besoins. Avec Rails, il faut obligatoirement définir quelques vues avant de pouvoir obtenir un semblant de gestion des données, alors qu'avec Django, si l'interface d'administration convient à l'application, c'est une chose de moins à développer.

Finalement, des mesures de performances montrent que Django a une grande avance par rapport à Rails. Mais ces deux frameworks sont globalement très proches, et la meilleure chose à faire est de les tester afin de se faire son propre avis sur la question.

9 Conclusion

Nous voici arrivés au terme de cette présentation, au cours de laquelle nous avons appris à connaitre les différents composants du framework Django, non sans commencer en premier lieu par un bref rappel sur le langage Python. Nous avons aussi brièvement comparé Django et Rails.

Django est un framework très puissant, offrant un grand nombre de fonctionnalités tout en restant relativement simple d'accès. De plus, il fournit d'office un grand nombre d'outils facilitant le développement, dont entre autre les vues génériques et des applications existantes. En se basant sur le langage Python, lui aussi très puissant tout en restant simple d'accès, Django permet de développer rapidement des applications web en ne se concentrant que sur la logique métier.

Certes, Django n'est pas parfait, mais face au nombreux autres frameworks disponibles à l'heure actuelle, il conserve une place de choix pour les applications web.

10 Bibliographie

[1] Constructive reasons to use Django instead of Rails, `http://jesusphreak.infogami.com/blog/why_django`, dernière visite : 21.05.2007

[2] Dive Into Python, Mark Pilgrim, 2004, 413 p., `http://www.diveintopython.org/toc/index.html`, dernière visite : 19.05.2007

[3] Django | Generic views | Django Documentation, `http://www.djangoproject.com/documentation/generic_views/`, dernière visite : 20.05.2007

[4] Django | Middleware | Django Documentation, `http://www.djangoproject.com/documentation/middleware/`, dernière visite : 20.05.2007

[5] Django | The Web framework for perfectionists with deadlines, `http://www.djangoproject.com/`, dernière visite : 19.05.2007

[6] Encore une comparaison Django/Ruby on Rails, `http://www.biologeek.com/journal/index.php/encore-une-comparaison-django-ruby-on-rails-les-deux-frameworks-web-qui-buzzent`, dernière visite : 21.05.2007

[7] Framework - Wikipédia, `http://fr.wikipedia.org/wiki/Framework`, dernière visite : 19.05.2007

[8] Regular expressions in Python, `http://www.python.org/doc/current/lib/module-re.html`, dernière visite : 20.05.2007

[9] The Django Book, `http://www.djangobook.com/`, dernière visite : 21.05.2007