# Minor swing with Django

Cédric Declerfayt, Fred Pauchet

# Table of Contents


Préface . . . 1
Environnement de travail . . . 2
1. Méthodologie de travail . . . 3
2. Environnements virtuels . . . 5
2.1. Création de l'environnement virtuel . . . 5
2.2. Installation de Django et création du répertoire de travail . . . 6
2.3. Gestion des dépendances . . . 7
2.4. Structure finale de l'environnement . . . 8
3. Django . . . 9
3.1. Gestion . . . 9
3.2. Structure d'une application . . . 9
3.3. Tests unitaires . . . 10
4. Construire des applications maintenables . . . 11
4.1. Au niveau des méthodes et fonctions . . . 11
4.2. Au niveau des classes . . . 11
4.3. Au niveau des composants . . . 11
4.4. De manière plus générale . . . 12
4.5. En pratique . . . 12
5. Chaîne d'outils . . . 13
5.1. PEP8 . . . 13
5.2. pep8, flake8, pylint . . . 19
5.3. Black . . . 21
5.4. pytest . . . 21
5.5. mypy . . . 21
6. Git . . . 22
7. graphviz . . . 23
8. En résumé . . . 24
Déploiement . . . 25
9. Un peu de théorie… Les principales étapes . . . 26
9.1. Définition de l'infrastructure . . . 26
9.2. Configuration et sécurisation de la machine hôte . . . 27
9.3. Mise à jour . . . 27
9.4. Supervision . . . 27
10. Déploiement sur CentOS . . . 28
10.1. Installation des dépendances systèmes . . . 28
10.2. Préparation de l'environnement utilisateur . . . 28
10.3. Configuration de l'application . . . 29
10.4. Création des répertoires de logs . . . 29

10.5. Création du répertoire pour le socket . . . 29
10.6. Gunicorn . . . 29
10.7. Supervision . . . 30
10.8. Ouverture des ports . . . 32
10.9. Installation d'Nginx . . . 32
10.10. Configuration des sauvegardes . . . 33
11. PostgreSQL . . . 35
12. MariaDB . . . 36
Modélisation . . . 37
13. Modélisation . . . 38
14. Queryset & managers . . . 39
15. Forms . . . 40
15.1. Dépendance avec le modèle . . . 40
15.2. Rendu et affichage . . . 41
15.3. Squelette par défaut . . . 41
15.4. Crispy-forms . . . 42
16. Validation des données . . . 43
17. En conclusion . . . 44
18. Migrations . . . 45
19. Modèle-vue-template . . . 46
19.1. Vues . . . 46
20. Logging . . . 58
21. Administration . . . 60
21.1. Quelques conseils . . . 60
Go Live ! . . . 62
22. Besoins utilisateurs . . . 63
23. Besoins fonctionnels . . . 64
23.1. Gestion des utilisateurs . . . 64
23.2. Gestion des listes . . . 64
23.3. Gestion des souhaits . . . 64
23.4. Gestion des réalisations de souhaits . . . 65
23.5. Gestion des personnes réalisants les souhaits et qui ne sont pas connues . . . 66
24. Tests unitaires . . . 72
24.1. Pourquoi s'ennuyer à écrire des tests? . . . 72
24.2. Why Bother with Test Discipline? . . . 72
24.3. What are you testing? . . . 73
24.4. Couverture de code . . . 73
24.5. Comment tester ? . . . 73
24.6. Quelques liens utiles . . . 76
25. A retenir . . . 77
25.1. Constructeurs . . . 77

25.2. Relations . . . . . 77
25.3. Querysets & managers . . . . . 79
26. Refactoring . . . . . 80
26.1. Classe abstraite . . . . . 80
26.2. Héritage classique . . . . . 81
26.3. Classe proxy . . . . . 82
Métamodèle . . . . . 84
27. Supervision des logs . . . . . 85
28. feedbacks utilisateurs . . . . . 86
En bonus . . . . . 87
29. Snippets utiles (et forcément dispensables) . . . . . 88
29.1. Récupération du dernier tag Git en Python . . . . . 88

# Préface

On ne va pas se mentir: il existe enormément de tutoriaux très bien réalisés sur "Comment réaliser une application Django" et autres "Déployer votre code en 2 minutes". On se disait juste que ces tutoriaux restaient relativement haut-niveau et se limitaient à un contexte donné.

L'idée du texte ci-dessous est de jeter les bases d'un bon développement, en survolant l'ensemble des outils permettant de suivre des lignes directrices reconnues, de maintenir une bonne qualité de code au travers des différentes étapes (du développement au déploiement) et de s'assurer du maintient correct de la base de code, en permettant à n'importe qui de reprendre le développement.

Ces idées ne s'appliquent pas uniquement à Django et à son cadre de travail, ni même au langage Python. Juste que ces deux bidules sont de bons candidats et que le cadre de travail est bien défini et suffisamment flexible.

Django se présente comme un `Framework Web pour perfectionnistes ayant des deadlines <https://www.djangoproject.com/>`_.

Django suit quelques principes <https://docs.djangoproject.com/en/dev/misc/design-philosophies/>:

- Faible couplage et forte cohésion, pour que chaque composant ait son indépendance.
- Moins de code, plus de fonctionnalités.
- `Don't repeat yourself <https://fr.wikipedia.org/wiki/Sec>`_: ne pas se répéter!
- Rapidité du développement (après une courbe d'apprentissage relativement ardue, malgré tout)

Mis côté à côté, l'application de ces principes permet une meilleure stabilité du projet. Dans la suite de ce chapitre, on verra comment configurer l'environnement, comment installer Django de manière isolée et comment démarrer un nouveau projet. On verra comment gérer correctement les dépendances, les versions et comment applique un score sur note code.

Finalement, on verra aussique la configuration proposée par défaut par le framework n'est pas idéale pour la majorité des cas.

Pour cela, on présentera différents outils (mypy, flake8, black, …), la rédaction de tests unitaires et d'intégration pour limiter les régressions, les règles de nomenclature et de contrôle du contenu, ainsi que les bonnes étapes à suivre pour arriver à un déploiement rapide et fonctionnel avec peu d'efforts.

Et tout ça à un seul et même endroit. Oui. :-)

Bonne lecture.

# Environnement de travail

Avant de démarrer le développement, il est nécessaire de passer un peu de temps sur la configuration de l’environnement.

Les morceaux de code seront développés pour Python3.4+ et Django 1.8+. Ils nécessiteront peut-être quelques adaptations pour fonctionner sur une version antérieure.

**Remarque** : les commandes qui seront exécutés dans ce livre le seront depuis un shell sous GNU/Linux. Certaines devront donc être adaptées si vous êtes dans un autre environnemnet.

# Chapitre 1. Méthodologie de travail

Pour la méthode de travail et de développement, on va se baser sur les The Twelve-factor App - ou plus simplement les **12 facteurs**.

L'idée derrière cette méthode consiste à pousser les concepts suivants (repris grossièrement de la page d'introduction :

1. Faciliter la mise en place de phases d'automatisation; plus simplement, faciliter les mises à jour applicatives, simplifier la gestion de l'hôte, diminuer la divergence entre les différents environnements d'exécution et offrir la possibilité d'intégrer le projet dans un processus d'intégration continue/déploiement continu
2. Faciliter la mise à pied de nouveaux développeurs ou de personnes souhaitant rejoindre le projet.
3. Faciliter
4. Augmenter l'agilité générale du projet, en permettant une meilleure évolutivité architecturale et une meilleure mise à l'échelle - *Vous avez 5000 utilisateurs en plus? Ajoutez un serveur et on n'en parle plus ;-)*.

En pratique, les idées planquées derrière les quelques phrases ci-dessus permettront de monter facilement un nouvel environnement - qu'il soit sur la machine du petit nouveau ou sur un serveur Azure, Heroku, Digital Ocean ou votre nouveau Raspberry Pi Zéro caché à la cave.

Pour reprendre de manière très brute les différentes idées derrière cette méthode, on a:

| **NOTE** | pensez à retravailler la partie ci-dessous; la version anglophone semble plus compréhensible... :-/ |
|---|---|

1. Une base de code suivie avec un système de contrôle de version, plusieurs déploiements
2. Déclarez explicitement et isolez les dépendances
3. Stockez la configuration dans l'environnement
4. Traitez les services externes comme des ressources attachées
5. Séparez strictement les étapes d'assemblage et d'exécution
6. Exécutez l'application comme un ou plusieurs processus sans état
7. Exportez les services via des associations de ports
8. Grossissez à l'aide du modèle de processus
9. Maximisez la robustesse avec des démarrages rapides et des arrêts gracieux
10. Gardez le développement, la validation et la production aussi proches que possible
11. Traitez les logs comme des flux d'évènements
12. Lancez les processus d'administration et de maintenance comme des one-off-processes

*Table 1. Concrètement*

| Concept | Outil | Description |
| --- | --- | --- |
| Base de code suivie avec un système de contrôle de version | Git, Mercurial, SVN, … | Chaque déploiement démarre à partir d'une base de code unique. Il n'y pas de dépôt "Prod", "Staging" ou "Dev". Il n'y en a qu'un et un seul. |
| Déclaration explicite et isolation des dépendances | Pyenv, environnements virtuels, RVM, … | Afin de ne pas perturber les dépendances systèmes, chaque application doit disposer d'un environnement sain par défaut. |
| Configuration dans l'environnement | Fichiers .ENV | Toute clé de configuration (nom du serveur de base de données, adresse d'un service Web externe, clé d'API pour l'interrogation d'une ressource, …) sera définie directement au niveau de l'hôte - à aucun moment, on ne doit trouver un mot de passe en clair dans le dépôt source ou une valeur susceptible d'évoluer, écrite en dur dans le code. |
| Services externes = ressources locales | Fichiers .ENV | Chaque ressource doit pouvoir être interchangeable avec une autre, sans modification du code source. La solution consiste à passer toutes ces informations (nom du serveur et type de base de données, clé d'authentification, …) directement via des variables d'environnement. |
| Bien séparer les étapes de construction des étapes de mise à disposition | Capistrano, Gitea, un serveur d'artefacts, … | L'idée est de pouvoir récupérer une version spécifique du code, sans que celle-ci ne puisse avoir été modifiée. Git permet bien de gérer des versions (au travers des tags), mais ces éléments peuvent sans doute être modifiés directement au travers de l'historique. |

# Chapitre 2. Environnements virtuels

On va commencer avec la partie la moins funky, mais la plus utile, dans la vie d'un développeur: la gestion et l'isolation des dépendances.

Il est tout à fait possible de s'en passer complètement dans le cadre de "petits" projets ou d'applications déployées sur des machines dédiées, et de fonctionner à grand renforts de "sudo" et d'installation globale des dépendances. Cette pratique est cependant fortement déconseillée pour plusieurs raisons:

1. Pour la reproductibilité d'un environnement spécifique. Cela évite notamment les réponses type "Ca juste marche chez moi", puisqu'on a la possibilité de construire un environnement sain et appliquer des dépendances identiques, quelle que soit la machine hôte.
2. Il est tout à fait envisagable que deux applications soient déployées sur un même hôte, et nécessitent chacune deux versions différentes d'une même dépendance.

> But it works on my machine! Then, we'll ship your machine.
>
> — A famous meme, And this is how Docker was born.

Nous allons utiliser le module `venv` afin de créer un `environnement virtuel <http://sametmax.com/les-environnement-virtuels-python-virtualenv-et-virtualenvwrapper/>`_ pour python.

**NOTE** auparavant, on utilisait `virtualenvwrapper`. mais cela fait plusieurs années que je ne l'utilise plus. On l'intègre quand même ?

## 2.1. Création de l'environnement virtuel

Commencons par créer un environnement virtuel, afin d'y stocker les dépendances. Lancez `python3 -m venv gwift-env`.

```
---
Intégrer les résultats de la création de l'environnement
---
```

Ceci créera l'arborescence de fichiers suivante, qui peut à nouveau être un peu différente en fonction du système d'exploitation:

a. code-block:: shell

```
$ ls .virtualenvs/gwift-env
bin include lib
```

Nous pouvons ensuite l'activer grâce à la commande `source gwift-env`.

```
---
Intégrer les résultats de l'accès de l'environnement
---
```

Le **shell** signale que nous sommes bien dans l'environnement `gwift-env` en l'affichant avant le `$`. Par la suite, nous considérerons que l'environnement virtuel est toujours activé, même si `gwift-env` n'est pas présent devant chaque `$`.

A présent, tous les binaires de cet environnement prendront le pas sur les binaires du système. De la même manière, une variable `PATH` propre est définie et utilisée, afin que les librairies Python y soient stockées. C'est donc dans cet environnement virtuel que nous retrouverons le code source de Django, ainsi que des librairies externes pour Python une fois que nous les aurons installées.

Pour désactiver l'environnement virtuel, il suffira d'utiliser la commande `deactivate`

## 2.2. Installation de Django et création du répertoire de travail

Comme l'environnement est activé, on peut à présent y installer Django. La librairie restera indépendante du reste du système, et ne polluera pas les autres projets.

C'est parti: `pip install django`!

a. code-block:: shell

```
$ pip install django
Collecting django
  Downloading Django-X.Y.Z
100% |################################|
Installing collected packages: django
Successfully installed django-X.Y.Z
```

Les commandes de création d'un nouveau site sont à présent disponibles, la principale étant `django-admin startproject`. Par la suite, nous utiliserons `manage.py`, qui constitue un **wrapper** autour de `django-admin`.

Pour démarrer notre projet, nous lançons donc `django-admin startproject gwift`.

```
$ django-admin startproject gwift
```

Cette action a pour effet de créer un nouveau dossier `gwift`, dans lequel on trouve la structure suivante:

```
$ tree gwift
gwift
├── gwift
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

C’est sans ce répertoire que vont vivre tous les fichiers liés au projet. Le but est de faire en sorte que toutes les opérations (maintenance, déploiement, écriture, tests, …) puissent se faire à partir d’un seul point d’entrée. Tant qu’on y est, nous pouvons rajouter les répertoires utiles à la gestion de notre projet, à savoir la documentation, les dépendances et le README:

```
$ mkdir docs requirements
$ touch docs/README.md
```

```
$ tree gwift
gwift
├── gwift
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
|-- docs/
|-- requirements/
|-- README
```

Chacun de ces fichiers sert à:

- `settings.py` contient tous les paramètres globaux à notre projet.
- `urls.py` contient les variables de routes, les adresses utilisées et les fonctions vers lesquelles elles pointent.
- `manage.py`, pour toutes les commandes de gestion.
- `wsgi.py` contient la définition de l’interface `WSGI <https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface>`_, qui permettra à votre serveur Web (Nginx, Apache, …) de faire un pont vers votre projet.

**NOTE** déplacer la configuration dans un répertoire `config` à part.

## 2.3. Gestion des dépendances

Comme nous venons d’ajouter une dépendance à notre projet, nous allons créer un fichier

reprenant tous les dépendances de notre projet. Celles-ci sont normalement placées dans un fichier `requirements.txt`. Dans un premier temps, ce fichier peut être placé directement à la racine du projet, mais on préférera rapidement le déplacer dans un sous-répertoire spécifique (`requirements`), afin de grouper les dépendances en fonction de leur utilité:

- `base.txt`
- `dev.txt`
- `staging.txt`
- `production.txt`

Au début de chaque fichier, il suffira d'ajouter la ligne `-r base.txt`, puis de lancer l'installation grâce à un `pip install -r <nom du fichier>`. De cette manière, il est tout à fait acceptable de n'installer `flake8` et `django-debug-toolbar` qu'en développement par exemple. Dans l'immédiat, ajoutez simplement `django` dans le fichier `requirements/base.txt`.

```
$ echo django >> requirements/base.txt
```

Par la suite, il vous faudra **absolument** spécifier les versions à utiliser: les librairies que vous utilisez comme dépendances évoluent, de la même manière que vos projets. Des fonctions sont cassées, certaines signatures sont modifiées, des comportements sont altérés, etc. Si vous voulez être sûr et certain que le code que vous avez écrit continue à fonctionner, spécifiez la version de chaque librairie de dépendances. Avec les mécanismes d'intégration continue et de tests unitaires, on verra plus loin comment se prémunir d'un changement inattendu.

## 2.4. Structure finale de l'environnement

Nous avons donc la strucutre finale pour notre environnement de travail:

```
$ tree ~/gwift-project
gwift
├── docs
│   └── README.md
├── gwift
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
│   manage.py
└── requirements
    └── base.txt
```

# Chapitre 3. Django

Comme on l'a vu ci-dessus, `django-admin` permet de créer un nouveau projet. On fait ici une distinction entre un **projet** et une **application**:

- **Projet**: ensemble des applications, paramètres, pages HTML, middlwares, dépendances, etc., qui font que votre code fait ce qu'il est sensé faire.
- **Application**: **contexte** éventuellement indépendant, permettant d'effectuer une partie isolée de ce que l'on veut faire.

Pour `gwift`, on va notamment avoir une application pour la gestion des listes de souhaits et des éléments, une deuxième application pour la gestion des utilisateurs, voire une troisième application qui gérera les partages entre utilisateurs et listes.

On voit bien ici le principe de **contexte**: l'application viendra avec son modèle, ses tests, ses vues et son paramétrage. Elle pourra éventuellement être réutilisée dans un autre projet. C'est en ça que consistent les `paquets Django <https://www.djangopackages.com/>`_ déjà disponibles: ce sont simplement de petites applications empaquetées pour être réutilisées (eg. `Django-Rest-Framework <https://github.com/tomchristie/django-rest-framework>`, *`Django-Debug-Toolbar <https://github.com/django-debug-toolbar/django-debug-toolbar>`*, ...).

## 3.1. Gestion

Comme expliqué un peu plus haut, le fichier `manage.py` est un **wrapper** sur les commandes `django-admin`. A partir de maintenant, nous n'utiliserons plus que celui-là pour tout ce qui touchera à la gestion de notre projet:

- `manage.py check` pour vérifier (en surface...) que votre projet ne rencontre aucune erreur
- `manage.py runserver` pour lancer un serveur de développement
- `manage.py test` pour découvrir les tests unitaires disponibles et les lancer.

La liste complète peut être affichée avec `manage.py help`. Vous remarquerez que ces commandes sont groupées selon différentes catégories:

- **auth**: création d'un nouveau super-utilisateur, changer le mot de passe pour un utilisateur existant.
- **django**: vérifier la **compliance** du projet, lancer un **shell**, **dumper** les données de la base, effectuer une migration du schéma, ...
- **sessions**: suppressions des sessions en cours
- **staticfiles**: gestion des fichiers statiques et lancement du serveur de développement.

Nous verrons plus tard comment ajouter de nouvelles commandes.

## 3.2. Structure d'une application

Maintenant que l'on a vu à quoi servait `manage.py`, on peut créer notre nouvelle application grâce à

la commande `manage.py startapp <label>`.

Cette application servira à structurer les listes de souhaits, les éléments qui les composent et les parties que chaque utilisateur pourra offrir. Essayez de trouver un nom éloquent, court et qui résume bien ce que fait l'application. Pour nous, ce sera donc `wish`. C'est parti pour `manage.py startapp wish`!

```
$ python manage.py startapp wish
```

Résultat? Django nous a créé un répertoire `wish`, dans lequel on trouve les fichiers suivants:

```
$ ls -l wish
admin.py  __init__.py  migrations  models.py  tests.py  views.py
```

En résumé, chaque fichier a la fonction suivante:

- `admin.py` servira à structurer l'administration de notre application. Chaque information peut en effet être administrée facilement au travers d'une interface générée à la volée par le framework. On y reviendra par la suite.
- `init.py` pour que notre répertoire `wish` soit converti en package Python.
- `migrations/`, dossier dans lequel seront stockées toutes les différentes migrations de notre application.
- `models.py` pour représenter et structurer nos données.
- `tests.py` pour les tests unitaires.

## 3.3. Tests unitaires

Plein de trucs à compléter ici ;-) Est-ce qu'on passe par pytest ou par le framework intégré ? Quels sont les avantages de l'un % à l'autre ? * `views.py` pour définir ce que nous pouvons faire avec nos données.

# Chapitre 4. Construire des applications maintenables

Pour cette section, je me base d'un résumé de l'ebook **Building Maintenable Software** disponible chez `O'Reilly <http://shop.oreilly.com/product/0636920049555.do`_ qui vaut clairement le détour pour poser les bases d'un projet.

Ce livre répartit un ensemble de conseils parmi quatre niveaux de composants:

- Les méthodes et fonctions
- Les classes
- Les composants
- Et de manière plus générale.

## 4.1. Au niveau des méthodes et fonctions

- Gardez vos méthodes/fonctions courtes. Pas plus de 15 lignes, en comptant les commentaires. Des exceptions sont possibles, mais dans une certaine mesure uniquement (pas plus de 6.9% de plus de 60 lignes; pas plus de 22.3% de plus de 30 lignes, au plus 43.7% de plus de 15 lignes et au moins 56.3% en dessous de 15 lignes). Oui, c'est dur à tenir, mais faisable.
- Conserver une complexité de McCabe en dessous de 5, c'est-à-dire avec quatre branches au maximum. A nouveau, si on a une méthode avec une complexité cyclomatique de 15, la séparer en 3 fonctions avec une complexité de 5 conservera globalement le nombre 15, mais rendra le code de chacune de ces méthodes plus lisible, plus maintenable.
- N'écrivez votre code qu'une seule fois: évitez les duplications, copie, etc., c'est juste mal: imaginez qu'un bug soit découvert dans une fonction; il devra alors être corrigé dans toutes les fonctions qui auront été copiées/collées. C'est aussi une forme de régression.
- Conservez de petites interfaces. Quatre paramètres, pas plus. Au besoin, refactorisez certains paramètres dans une classe, plus facile à tester.

## 4.2. Au niveau des classes

- Privilégiez un couplage faible entre vos classes. Ceci n'est pas toujours possible, mais dans la mesure du possible, éclatez vos classes en fonction de leur domaine de compétences. L'implémentation du service `UserNotificationsService` ne doit pas forcément se trouver embarqué dans une classe `UserService`. De même, pensez à passer par une interface (commune à plusieurs classes), afin d'ajouter une couche d'abstraction. La classe appellante n'aura alors que les méthodes offertes par l'interface comme points d'entrée.

## 4.3. Au niveau des composants

- Tout comme pour les classes, il faut conserver un couplage faible au niveau des composants également. Une manière d'arriver à ce résultat est de conserver un nombre de points d'entrée

restreint, et d'éviter qu'on ne puisse contacter trop facilement des couches séparées de l'architecture. Pour une architecture n-tiers par exemple, la couche d'abstraction à la base de données ne peut être connue que des services; sans cela, au bout de quelques semaines, n'importe quelle couche de présentation risque de contacter directement la base de données, "juste parce qu'elle en a la possibilité". Vous pourrez également passer par des interfaces, afin de réduire le nombre de points d'entrée connus par un composant externe (qui ne connaîtra par exemple que `IFileTransfer` avec ses méthodes `put` et `get`, et non pas les détails d'implémentation complet d'une classe `FtpFileTransfer` ou `SshFileTransfer`).

- Conserver un bon balancement au niveau des composants: évitez qu'un composant **A** ne soit un énorme mastodonte, alors que le composant juste à côté n'est capable que d'une action. De cette manière, les nouvelles fonctionnalités seront mieux réparties parmi les différents systèmes, et les responsabilités plus faciles à gérer. Un conseil est d'avoir un nombre de composants compris entre 6 et 12 (idéalement, 12), et que ces composants soit approximativement de même taille.

## 4.4. De manière plus générale

- Conserver une densité de code faible: il n'est évidemment pas possible d'implémenter n'importe quelle nouvelle fonctionnalité en moins de 20 lignes de code; l'idée ici est que la réécriture du projet ne prenne pas plus de 20 hommes/mois. Pour cela, il faut (activement) passer du temps à réduire la taille du code existant: soit en faisant du refactoring (intensif?), soit en utilisant des librairies existantes, soit en explosant un système existant en plusieurs sous-systèmes communiquant entre eux. Mais surtout en évitant de copier/coller bêtement du code existant.
- Automatiser les tests, ajouter un environnement d'intégration continue dès le début du projet et vérifier par des outils les points ci-dessus.

## 4.5. En pratique

Par rapport aux points repris ci-dessus, l'environnement Python et le framework Django proposent un ensemble d'outils intégrés qui permettent de répondre à chaque point. Avant d'aller plus loin, donc, un petit point sur les conventions, les tests (unitaires, orientés comportement, basés sur la documentation, …), la gestion de version du code et sur la documentation. Plus que dans tout langage compilé, ceux-ci sont pratiquement obligatoires. Vous pourrez les voir comme une perte de temps dans un premier temps, mais nous vous promettons qu'ils vous en feront gagner par la suite.

# Chapitre 5. Chaîne d’outils

Le langage Python fonctionne avec un système d’améliorations basées sur des propositions: les PEP, ou “Python Enhancement Proposal”.

Celle qui va nous intéresser pour cette section est la PEP 8 — Style Guide for Python Code. Elle spécifie comment du code Python doit être organisé ou formaté, quelles sont les conventions pour l’indentation, le nommage des variables et des classes, … En bref, elle décrit comment écrire du code proprement pour que d’autres développeurs puissent le reprendre facilement, ou simplement que votre base de code ne dérive lentement vers un seuil de non-maintenabilité.

## 5.1. PEP8

Le langage Python fonctionne avec un système d’améliorations basées sur des propositions: les PEP, ou "**Python Enhancement Proposal**". Chacune d’entre elles doit être approuvée par le `Benevolent Dictator For Life <http://fr.wikipedia.org/wiki/Benevolent_Dictator_for_Life>`_.

La PEP qui nous intéresse plus particulièrement pour la suite est la `PEP-8 <https://www.python.org/dev/peps/pep-0008/>`_, ou "Style Guide for Python Code". Elle spécifie des conventions d’organisation et de formatage de code Python, quelles sont les conventions pour l’indentation, le nommage des variables et des classes, etc. En bref, elle décrit comment écrire du code proprement pour que d’autres développeurs puissent le reprendre facilement, ou simplement que votre base de code ne dérive lentement vers un seuil de non-maintenabilité.

Sur cette base, un outil existe et listera l’ensemble des conventions qui ne sont pas correctement suivies dans votre projet: pep8. Pour l’installer, passez par pip. Lancez ensuite la commande pep8 suivie du chemin à analyser (`.`, le nom d’un répertoire, le nom d’un fichier `.py`, …). Si vous souhaitez uniquement avoir le nombre d’erreur de chaque type, saisissez les options `--statistics -qq`.

a. code-block:: shell

```
$ pep8 . --statistics -qq
```

```
7       E101 indentation contains mixed spaces and tabs
6       E122 continuation line missing indentation or outdented
8       E127 continuation line over-indented for visual indent
23      E128 continuation line under-indented for visual indent
3       E131 continuation line unaligned for hanging indent
12      E201 whitespace after '{'
13      E202 whitespace before '}'
86      E203 whitespace before ':'
```

Si vous ne voulez pas être dérangé sur votre manière de coder, et que vous voulez juste avoir un retour sur une analyse de votre code, essayez `pyflakes`: il analysera vos sources à la recherche de sources d’erreurs possibles (imports inutilisés, méthodes inconnues, etc.).

Finalement, la solution qui couvre ces deux domaines existe et s'intitule `flake8 <https://github.com/PyCQA/flake8>`_. Sur base la même interface que `pep8`, vous aurez en plus tous les avantages liés à `pyflakes` concernant votre code source.

### 5.1.1. PEP257

a. todo:: à remplir avec `pydocstyle`.

### 5.1.2. Tests

Comme tout bon **framework** qui se respecte, Django embarque tout un environnement facilitant le lancement de tests; chaque application est créée par défaut avec un fichier **tests.py**, qui inclut la classe `TestCase` depuis le package `django.test`:

a. code-block:: python

```
from django.test import TestCase
```

```
class TestModel(TestCase):
    def test_str(self):
        raise NotImplementedError('Not implemented yet')
```

Idéalement, chaque fonction ou méthode doit être testée afin de bien en valider le fonctionnement, indépendamment du reste des composants. Cela permet d'isoler chaque bloc de manière unitaire, et permet de ne pas rencontrer de régression lors de l'ajout d'une nouvelle fonctionnalité ou de la modification d'une existante. Il existe plusieurs types de tests (intégration, comportement, …); on ne parlera ici que des tests unitaires.

Avoir des tests, c'est bien. S'assurer que tout est testé, c'est mieux. C'est là qu'il est utile d'avoir le pourcentage de code couvert par les différents tests, pour savoir ce qui peut être amélioré.

### 5.1.3. Couverture de code

La couverture de code est une analyse qui donne un pourcentage lié à la quantité de code couvert par les tests. Attention qu'il ne s'agit pas de vérifier que le code est **bien** testé, mais juste de vérifier **quelle partie** du code est testée. En Python, il existe le paquet `coverage <https://pypi.python.org/pypi/coverage/>`_, qui se charge d'évaluer le pourcentage de code couvert par les tests. Ajoutez-le dans le fichier `requirements/base.txt`, et lancez une couverture de code grâce à la commande `coverage`. La configuration peut se faire dans un fichier `.coveragerc` que vous placerez à la racine de votre projet, et qui sera lu lors de l'exécution.

a. code-block:: shell

```
# requirements/base.text
[...]
coverage
django_coverage_plugin
```

b. code-block:: shell

```
# .coveragerc to control coverage.py
[run]
branch = True
omit = ../*migrations*
plugins =
    django_coverage_plugin
```

```
[report]
ignore_errors = True
```

```
[html]
directory = coverage_html_report
```

c. todo:: le bloc ci-dessous est à revoir pour isoler la configuration.

d. code-block:: shell

```
$ coverage run --source "." manage.py test
$ coverage report
```

```
Name                        Stmts   Miss  Cover
---------------------------------------------
gwift\gwift\__init__.py         0      0   100%
gwift\gwift\settings.py        17      0   100%
gwift\gwift\urls.py             5      5     0%
gwift\gwift\wsgi.py             4      4     0%
gwift\manage.py                 6      0   100%
gwift\wish\__init__.py          0      0   100%
gwift\wish\admin.py             1      0   100%
gwift\wish\models.py           49     16    67%
gwift\wish\tests.py             1      1     0%
gwift\wish\views.py             6      6     0%
---------------------------------------------
TOTAL                          89     32    64%
```

```
$ coverage html
```

Ceci vous affichera non seulement la couverture de code estimée, et générera également vos fichiers sources avec les branches non couvertes. Pour gagner un peu de temps, n'hésitez pas à créer un fichier `Makefile` que vous placerez à la racine du projet. L'exemple ci-dessous permettra, grâce à la commande `make coverage`, d'arriver au même résultat que ci-dessus:

a. code-block:: shell

```
# Makefile for gwift
#
```

```
# User-friendly check for coverage
ifeq ($(shell which coverage >/dev/null 2>&1; echo $$?), 1)
  $(error The 'coverage' command was not found. Make sure you have coverage
installed)
endif
```

```
.PHONY: help coverage
```

```
help:
    @echo "  coverage to run coverage check of the source files."
```

```
coverage:
    coverage run --source='.' manage.py test; coverage report; coverage html;
    @echo "Testing of coverage in the sources finished."
```

### 5.1.4. Complexité de McCabe

La `complexité cyclomatique <https://fr.wikipedia.org/wiki/Nombre_cyclomatique>`_ (ou complexité de McCabe) peut s'apparenter à mesure de difficulté de compréhension du code, en fonction du nombre d'embranchements trouvés dans une même section. Quand le cycle d'exécution du code rencontre une condition, il peut soit rentrer dedans, soit passer directement à la suite. Par exemple:

a. code-block:: python

```
if True == True:
    pass # never happens
```

```
# continue ...
```

La condition existe, mais on ne passera jamais dedans. A l'inverse, le code suivant aura une complexité pourrie à cause du nombre de conditions imbriquées:

a. code-block:: python

```
def compare(a, b, c, d, e):
    if a == b:
        if b == c:
            if c == d:
                if d == e:
                    print('Yeah!')
                    return 1
```

Potentiellement, les tests unitaires qui seront nécessaires à couvrir tous les cas de figure seront au nombre de quatre: le cas par défaut (a est différent de b, rien ne se passe), puis les autres cas, jusqu'à arriver à l'impression à l'écran et à la valeur de retour. La complexité cyclomatique d'un bloc est évaluée sur base du nombre d'embranchements possibles; par défaut, sa valeur est de 1. Si on rencontre une condition, elle passera à 2, etc.

Pour l'exemple ci-dessous, on va en fait devoir vérifier au moins chacun des cas pour s'assurer que la couverture est complète. On devrait donc trouver:

1. Un test pour entrer (ou non) dans la condition `a == b`
2. Un test pour entrer (ou non) dans la condition `b == c`
3. Un test pour entrer (ou non) dans la condition `c == d`
4. Un test pour entrer (ou non) dans la condition `d == e`
5. Et s'assurer que n'importe quel autre cas retournera la valeur `None`.

On a donc bien besoin de minimum cinq tests pour couvrir l'entièreté des cas présentés.

Le nombre de tests unitaires nécessaires à la couverture d'un bloc est au minimum égal à la complexité cyclomatique de ce bloc. Une possibilité pour améliorer la maintenance du code est de faire baisser ce nombre, et de le conserver sous un certain seuil. Certains recommandent de le garder sous une complexité de 10; d'autres de 5.

a. note::

```
Evidemment, si on refactorise un bloc pour en extraire une méthode, cela
n'améliorera pas sa complexité cyclomatique globale
```

A nouveau, un greffon pour `flake8` existe et donnera une estimation de la complexité de McCabe pour les fonctions trop complexes. Installez-le avec `pip install mccabe`, et activez-le avec le

paramètre `--max-complexity`. Toute fonction dans la complexité est supérieure à cette valeur sera considérée comme trop complexe.

### 5.1.5. Documentation

Il existe plusieurs manières de générer la documentation d'un projet. Les plus connues sont `Sphinx <http://sphinx-doc.org/>`_ et `MkDocs <http://www.mkdocs.org/>`. *Le premier a l'avantage d'être plus reconnu dans la communauté Python que l'autre, de pouvoir **parser** le code pour en extraire la documentation et de pouvoir lancer des `tests orientés documentation <https://duckduckgo.com/?q=documentation+driven+development&t=ffsb>`*. A contrario, votre syntaxe devra respecter `ReStructuredText <https://en.wikipedia.org/wiki/ReStructuredText>`_. Le second a l'avantage d'avoir une syntaxe plus simple à apprendre et à comprendre, mais est plus limité dans son résultat.

Dans l'immédiat, nous nous contenterons d'avoir des modules documentés (quelle que soit la méthode Sphinx/MkDocs/…). Dans la continuié de `Flake8`, il existe un greffon qui vérifie la présence de commentaires au niveau des méthodes et modules développés.

a. note::

```
voir si il ne faudrait pas mieux passer par pydocstyle.
```

b. code-block:: shell

```
$ pip install flake8_docstrings
```

Lancez ensuite `flake8` avec la commande `flake8 . --exclude="migrations"`. Sur notre projet (presque) vide, le résultat sera le suivant:

a. code-block:: shell

```
$ flake8 . --exclude="migrations"
.\src\manage.py:1:1: D100  Missing docstring in public module
.\src\gwift\__init__.py:1:1: D100  Missing docstring in public module
.\src\gwift\urls.py:1:1: D400  First line should end with a period (not 'n')
.\src\wish\__init__.py:1:1: D100  Missing docstring in public module
.\src\wish\admin.py:1:1: D100  Missing docstring in public module
.\src\wish\admin.py:1:1: F401 'admin' imported but unused
.\src\wish\models.py:1:1: D100  Missing docstring in public module
.\src\wish\models.py:1:1: F401 'models' imported but unused
.\src\wish\tests.py:1:1: D100  Missing docstring in public module
.\src\wish\tests.py:1:1: F401 'TestCase' imported but unused
.\src\wish\views.py:1:1: D100  Missing docstring in public module
.\src\wish\views.py:1:1: F401 'render' imported but unused
```

Bref, on le voit: nous n'avons que très peu de modules, et aucun d'eux n'est commenté.

En plus de cette méthode, Django permet également de rendre la documentation accessible depuis son interface d'administration.

## 5.2. pep8, flake8, pylint

Un outil existe et listera l'ensemble des conventions qui ne sont pas correctement suivies dans votre projet: pep8. Pour l'installer, passez par pip. Lancez ensuite la commande pep8 suivie du chemin à analyser (., le nom d'un répertoire, le nom d'un fichier .py, ...).

Si vous ne voulez pas être dérangé sur votre manière de coder, et que vous voulez juste avoir un retour sur une analyse de votre code, essayez pyflakes: il analaysera vos sources à la recherche d'erreurs (imports inutilsés, méthodes inconnues, etc.).

Et finalement, si vous voulez grouper les deux, il existe flake8. Sur base la même interface que pep8, vous aurez en plus des avertissements concernant le code source.

```
from datetime import datetime

"""On stocke la date du jour dans la variable ToD4y"""

ToD4y = datetime.today()

def print_today(ToD4y):
    today = ToD4y
    print(ToD4y)

def GetToday():
    return ToD4y


if __name__ == "__main__":
    t =   Get_Today()
    print(t)
```

L'exécution de la commande flake8 . retourne ceci:

```
test.py:7:1: E302 expected 2 blank lines, found 1
test.py:8:5: F841 local variable 'today' is assigned to but never used
test.py:11:1: E302 expected 2 blank lines, found 1
test.py:16:8: E222 multiple spaces after operator
test.py:16:11: F821 undefined name 'Get_Today'
test.py:18:1: W391 blank line at end of file
```

On trouve des erreurs:

- de **conventions**: le nombre de lignes qui séparent deux fonctions, le nombre d'espace après un opérateur, une ligne vide à la fin du fichier, ... Ces *erreurs* n'en sont pas vraiment, elles indiquent juste de potentiels problèmes de communication si le code devait être lu ou compris

par une autre personne.

- de **définition**: une variable assignée mais pas utilisée ou une lexème non trouvé. Cette dernière information indique clairement un bug potentiel.

L’étape d’après consiste à invoquer pylint. Lui, il est directement moins conciliant:

```
$ pylint test.py
************* Module test
test.py:16:6: C0326: Exactly one space required after assignment
    t =   Get_Today()
      ^ (bad-whitespace)
test.py:18:0: C0305: Trailing newlines (trailing-newlines)
test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test.py:3:0: W0105: String statement has no effect (pointless-string-statement)
test.py:5:0: C0103: Constant name "ToD4y" doesn't conform to UPPER_CASE naming style
(invalid-name)
test.py:7:16: W0621: Redefining name 'ToD4y' from outer scope (line 5) (redefined-
outer-name)
test.py:7:0: C0103: Argument name "ToD4y" doesn't conform to snake_case naming style
(invalid-name)
test.py:7:0: C0116: Missing function or method docstring (missing-function-docstring)
test.py:8:4: W0612: Unused variable 'today' (unused-variable)
test.py:11:0: C0103: Function name "GetToday" doesn't conform to snake_case naming
style (invalid-name)
test.py:11:0: C0116: Missing function or method docstring (missing-function-docstring)
test.py:16:4: C0103: Constant name "t" doesn't conform to UPPER_CASE naming style
(invalid-name)
test.py:16:10: E0602: Undefined variable 'Get_Today' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at -5.45/10
```

En gros, j’ai programmé comme une grosse bouse anémique (et oui, le score d’évaluation du code permet bien d’aller en négatif). En vrac, on trouve des problèmes liés:

- au nommage (C0103) et à la mise en forme (C0305, C0326, W0105)
- à des variables non définies (E0602)
- de la documentation manquante (C0114, C0116)
- de la redéfinition de variables (W0621).

Pour reprendre la documentation, chaque code possède sa signification (ouf!):

- C convention related checks
- R refactoring related checks
- W various warnings
- E errors, for probable bugs in the code

- F fatal, if an error occurred which prevented pylint from doing further* processing.

PyLint est la version ++, pour ceux qui veulent un code propre et sans bavure.

## 5.3. Black

## 5.4. pytest

## 5.5. mypy

# Chapitre 6. Git

| NOTE | insérer ici une description de Gitflow + quelques exemples types création, ajout, suppression, historique, branches, … et quelques schémas qui-vont-bien. |
| --- | --- |

Il existe plusiseurs outils permettant de gérer les versions du code, dont les plus connus sont `git <https://git-scm.com/>`_ et `mercurial <https://www.mercurial-scm.org/>`_.

Dans notre cas, nous utilisons git et hebergons le code et le livre directement sur le gitlab de `framasoft <https://git.framasoft.org/>`_

# Chapitre 7. graphviz

En utilisant django_extensions (! bien suivre les étapes d’installation !).

```
C:\app\graphviz\bin\dot.exe graph.dot -Tpng -o graph.png
```

# Chapitre 8. En résumé

En résumé, la création d'un **nouveau** projet Django demande plus ou moins toujours les mêmes actions:

1. Configurer un environnement virtuel
2. Installer les dépendances et les ajouter dans le fichier `requirements.txt`
3. Configurer le fichier `settings.py`

C'est ici que le projet CookieCutter va être intéressant: les X premières étapes peuvent être **bypassées** par une simple commande.

# Déploiement

Et sécurisation du serveur.

# Chapitre 9. Un peu de théorie… Les principales étapes

On va déjà parler de déploiement. Le serveur que django met à notre disposition est prévu uniquement pour le développement: inutile de passer par du code Python pour charger des fichiers statiques (feuilles de style, fichiers JavaScript, images, …). De même, la base de donnée doit supporter plus qu'un seul utilisateur: SQLite fonctionne très bien dès lors qu'on se limite à un seul utilisateur… Sur une application Web, il est plus que probable que vous rencontriez rapidement des erreurs de base de données verrouillée pour écriture par un autre processus. Il est donc plus que bénéfique de passer sur quelque chose de plus solide.

Si vous avez suivi les étapes jusqu'ici, vous devriez à peine disposer d'un espace de travail proprement configuré, d'un modèle relativement basique et d'une configuration avec une base de données simpliste. En bref, vous avez quelque chose qui fonctionne, mais qui ressemble de très loin à ce que vous souhaitez au final.

Il y a une raison très simple à aborder le déploiement dès maintenant: à trop attendre et à peaufiner son développement en local, on en oublie que sa finalité sera de se retrouver exposé sur un serveur. On risque d'avoir oublié une partie des désidérata, d'avoir zappé une fonctionnalité essentielle ou simplement de passer énormément de temps à adapter les sources pour qu'elles fonctionnent sur un environnement en particulier.

Aborder le déploiement maintenant permet également de rédiger dès le début les procédures d'installation, de mise à jour et de sauvegardes. Déploier une nouvelle version sera aussi simple que de récupérer la dernière archive depuis le dépôt, la placer dans le bon répertoire, appliquer des actions spécifiques (et souvent identiques entre deux versions), puis redémarrer les services adéquats.

Dans cette partie, on abordera les points suivants:

- La définition de l'infrastructure nécessaire à notre application
- La configuration de l'hôte, qui hébergera l'application: dans une machine physique, virtuelle ou dans un container. On abordera aussi rapidement les déploiements via Ansible, Chef, Puppet ou Salt.
- Les différentes méthodes de supervision de l'application: comment analyser les fichiers de logs et comment intercepter correctement une erreur si elle se présente et comment remonter l'information.
- Une partie sur la sécurité et la sécurisation de l'hôte.

## 9.1. Définition de l'infrastructure

Comme on l'a vu dans la première partie, Django est un framework complet, intégrant tous les mécanismes nécessaires à la bonne évolution d'une application. On peut ainsi commencer petit, et suivre l'évolution des besoins en fonction de la charge estimée ou ressentie, ajouter un mécanisme de mise en cache, des logiciels de suivi, …

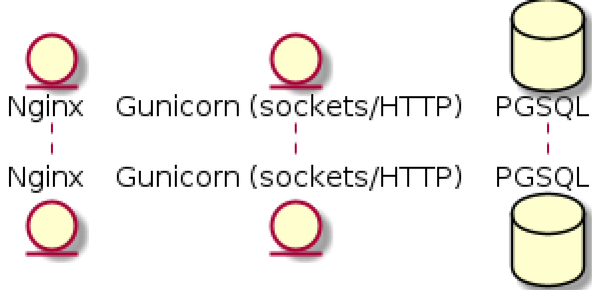
Pour une mise ne production, le standard **de facto** est le suivant:

- Nginx comme serveur principal
- Gunicorn comme serveur d’application
- Supervisorctl pour le monitoring
- PostgreSQL comme base de données.

C’est celle-ci que nous allons décrire ci-dessous.

## 9.2. Configuration et sécurisation de la machine hôte

Supervisor, nginx, gunicorn, utilisateurs, groupes, …




Aussi : Docker, Heroku, Digital Ocean, Scaleway, OVH, … Bref, sur Debian et CentOS pour avoir un panel assez large. On oublie Windows.

## 9.3. Mise à jour

Script de mise à jour.

## 9.4. Supervision

Qu’est-ce qu’on fait des logs après ? :-)

# Chapitre 10. Déploiement sur CentOS

```
yum update
groupadd --system webapps
groupadd --system gunicorn_sockets
useradd --system --gid webapps --shell /bin/bash --home /home/medplan medplan
mkdir -p /home/medplan
chown medplan:webapps /home/medplan
```

## 10.1. Installation des dépendances systèmes

```
yum install python36 git tree -y

# CentOS 7 ne dispose que de la version 3.7 d'SQLite. On a besoin d'une version 3.8 au
minimum:
wget https://kojipkgs.fedoraproject.org//packages/sqlite/3.8.11/1.fc21/x86_64/sqlite-
devel-3.8.11-1.fc21.x86_64.rpm
wget https://kojipkgs.fedoraproject.org//packages/sqlite/3.8.11/1.fc21/x86_64/sqlite-
3.8.11-1.fc21.x86_64.rpm
sudo yum install sqlite-3.8.11-1.fc21.x86_64.rpm sqlite-devel-3.8.11-1.fc21.x86_64.rpm
-y
```

## 10.2. Préparation de l'environnement utilisateur

```
su - medplan
cp /etc/skel/.bashrc .
cp /etc/skel/.bash_profile .
ssh-keygen
mkdir bin
mkdir .venvs
mkdir webapps
python3.6 -m venv .venvs/medplan
source .venvs/medplan/bin/activate
cd /home/medplan/webapps
git clone git@vmwmedtools:institutionnel/medplan.git
```

La clé SSH doit ensuite être renseignée au niveau du dépôt, afin de pouvoir y accéder.

A ce stade, on devrait déjà avoir quelque chose de fonctionnel en démarrant les commandes suivantes:

```
# en tant qu'utilisateur 'medplan'

source .venvs/medplan/bin/activate
pip install -U pip
pip install -r requirements/base.txt
pip install gunicorn
cd webapps/medplan
gunicorn config.wsgi:application --bind localhost:3000 --settings
=config.settings_production
```

## 10.3. Configuration de l'application

```
SECRET_KEY=<set your secret key here>
ALLOWED_HOSTS=*
STATIC_ROOT=/var/www/medplan/static
```

## 10.4. Création des répertoires de logs

```
mkdir -p /var/www/medplan/static
```

## 10.5. Création du répertoire pour le socket

Dans le fichier `/etc/tmpfiles.d/medplan.conf`:

```
D /var/run/webapps 0775 medplan gunicorn_sockets -
```

Suivi de la création par systemd :

```
systemd-tmpfiles --create
```

## 10.6. Gunicorn

```
#!/bin/bash

# defines settings for gunicorn
NAME="Medplan"
DJANGODIR=/home/medplan/webapps/medplan
SOCKFILE=/var/run/webapps/gunicorn_medplan.sock
USER=medplan
GROUP=gunicorn_sockets
NUM_WORKERS=5
DJANGO_SETTINGS_MODULE=config.settings_production
DJANGO_WSGI_MODULE=config.wsgi

echo "Starting $NAME as `whoami`"

source /home/medplan/.venvs/medplan/bin/activate
cd $DJANGODIR
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

exec gunicorn ${DJANGO_WSGI_MODULE}:application \
--name $NAME \
--workers $NUM_WORKERS \
--user $USER \
--bind=unix:$SOCKFILE \
--log-level=debug \
--log-file=-
```

## 10.7. Supervision

```
yum install supervisor -y
```

On crée ensuite le fichier `/etc/supervisord.d/medplan.ini`:

```
[program:medplan]
command=/home/medplan/bin/start_gunicorn.sh
user=medplan
stdout_logfile=/var/log/medplan/medplan.log
autostart=true
autorestart=unexpected
redirect_stdout=true
redirect_stderr=true
```

Et on crée les répertoires de logs, on démarre supervisord et on vérifie qu'il tourne correctement:

```
mkdir /var/log/medplan
chown medplan:nagios /var/log/medplan

systemctl enable supervisord
systemctl start supervisord.service
systemctl status supervisord.service
● supervisord.service - Process Monitoring and Control Daemon
   Loaded: loaded (/usr/lib/systemd/system/supervisord.service; enabled; vendor
preset: disabled)
   Active: active (running) since Tue 2019-12-24 10:08:09 CET; 10s ago
  Process: 2304 ExecStart=/usr/bin/supervisord -c /etc/supervisord.conf (code=exited,
status=0/SUCCESS)
 Main PID: 2310 (supervisord)
   CGroup: /system.slice/supervisord.service
           ├─2310 /usr/bin/python /usr/bin/supervisord -c /etc/supervisord.conf
           ├─2313 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
           ├─2317 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
           ├─2318 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
           ├─2321 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
           ├─2322 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
           └─2323 /home/medplan/.venvs/medplan/bin/python3
/home/medplan/.venvs/medplan/bin/gunicorn config.wsgi:...
ls /var/run/webapps/
```

On peut aussi vérifier que l'application est en train de tourner, à l'aide de la commande `supervisorctl`:

```
$$$ supervisorctl status gwift
gwift                            RUNNING    pid 31983, uptime 0:01:00
```

Et pour gérer le démarrage ou l'arrêt, on peut passer par les commandes suivantes:

```
$$$ supervisorctl stop gwift
gwift: stopped
root@ks3353535:/etc/supervisor/conf.d# supervisorctl start gwift
gwift: started
root@ks3353535:/etc/supervisor/conf.d# supervisorctl restart gwift
gwift: stopped
gwift: started
```

## 10.8. Ouverture des ports

```
firewall-cmd --permanent --zone=public --add-service=http
firewall-cmd --permanent --zone=public --add-service=https
firewall-cmd --reload
```

## 10.9. Installation d'Nginx

```
yum install nginx -y
usermod -a -G gunicorn_sockets nginx
```

On configure ensuite le fichier `/etc/nginx/conf.d/medplan.conf`:

```
upstream medplan_app {
        server unix:/var/run/webapps/gunicorn_medplan.sock fail_timeout=0;
}

server {
        listen 80;
        server_name <server_name>;
        root /var/www/medplan;
        error_log /var/log/nginx/medplan_error.log;
        access_log /var/log/nginx/medplan_access.log;

        client_max_body_size 4G;
        keepalive_timeout 5;

        gzip on;
        gzip_comp_level 7;
        gzip_proxied any;
        gzip_types gzip_types text/plain text/css text/xml text/javascript
application/x-javascript application/xml;


        location /static/ {
                access_log off;
                expires 30d;
                add_header Pragma public;
                add_header Cache-Control "public";
                add_header Vary "Accept-Encoding";
                try_files $uri $uri/ =404;
        }

        location / {
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header Host $http_host;
                proxy_redirect off;

                proxy_pass http://medplan_app;
        }
}
```

## 10.10. Configuration des sauvegardes

Les sauvegardes ont été configurées avec borg: `yum install borgbackup`.

C’est l’utilisateur medplan qui s’en occupe.

```
mkdir -p /home/medplan/borg-backups/
cd /home/medplan/borg-backups/
borg init medplan.borg -e=none
borg create medplan.borg::{now} ~/bin ~/webapps
```

Et dans le fichier crontab :

```
0 23 * * * /home/medplan/bin/backup.sh
```

On l'a déjà vu, Django se base sur un pattern type ActiveRecords pour l'ORM.

**NOTE** à vérifier ;-)

et supporte les principaux moteurs de bases de données connus: MariaDB (en natif depuis Django 3.0), PostgreSQL au travers de psycopg2 (en natif aussi), Microsoft SQLServer grâce aux drivers […à compléter] ou Oracle via cx_Oracle.

**WARNING** Chaque pilote doit être utilisé précautionneusement ! Chaque version de Django n'est pas toujours compatible avec chacune des versions des pilotes, et chaque moteur de base de données nécessite parfois une version spécifique du pilote. De fait, vous serez parfois bloqué sur une version de Django, simplement parce que votre serveur de base de données se trouvera dans une version spécifique (eg. Django 2.3 à cause d'un Oracle 12.1).

Ci-dessous, quelques procédures d'installation pour mettre un serveur à disposition. Les deux plus simples seront MariaDB et PostgreSQL, qu'on couvrira ci-dessous. Oracle et Microsoft SQLServer se trouveront en annexes.

# Chapitre 11. PostgreSQL

On commence par installer PostgreSQL.

Par exemple, dans le cas de debian, on exécute la commande suivante:

```
$$$ aptitude install postgresql postgresql-contrib
```

Ensuite, on crée un utilisateur pour la DB:

```
$$$ su - postgres
postgres@gwift:~$ createuser --interactive -P
Enter name of role to add: gwift_user
Enter password for new role:
Enter it again:
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
postgres@gwift:~$
```

Finalement, on peut créer la DB:

```
postgres@gwift:~$ createdb --owner gwift_user gwift
postgres@gwift:~$ exit
logout
$$$
```

**NOTE** penser à inclure un bidule pour les backups.

# Chapitre 12. MariaDB

Idem, installation, configuration, backup, tout ça. A copier de grimboite, je suis sûr d’avoir des notes là-dessus.

# Modélisation

Dans ce chapitre, on va parler de plusieurs concepts utiles au développement rapide d’une application. On parlera de modélisation, de migrations, d’administration auto-générée.

# Chapitre 13. Modélisation

On va aborder la modélisation des objets en elle-même, qui s'apparente à la conception de la base de données.

Django utilise un modèle ORM - c'est-à-dire que chaque objet peut s'apparenter à une table SQL, mais en ajoutant une couche propre au paradigme orienté objet. Il sera ainsi possible de définir facilement des notions d'héritage (tout en restant dans une forme d'héritage simple), la possibilité d'utiliser des propriétés spécifiques, des classes intermédiaires, …

L'avantage de tout ceci est que tout reste au niveau du code. Si l'on revient sur la méthodologie des douze facteurs, ce point concerne principalement la minimisation de la divergence entre les environnements d'exécution. Déployer une nouvelle instance de l'application pourra être réalisé directement à partir d'une seule et même commande, dans la mesure où **tout est embarqué au niveau du code**.

Assez de blabla, on démarre !

# Chapitre 14. Queryset & managers

L'ORM de Django propose par défaut deux objets hyper importants:

- Les managers, qui consistent en un point d'entrée pour accéder aux objets persistants
- Les querysets, qui permettent de filtrer des ensembles ou sous-ensemble d'objets. Les querysets peuvent s'imbriquer, pour ajouter d'autres filtres à des filtres existants.

En plus de cela, il faut bien tenir compte des propriétés `Meta` de la classe: si elle contient déjà un ordre par défaut, celui-ci sera pris en compte pour l'ensemble des requêtes effectuées sur cette classe.

# Chapitre 15. Forms

Ou comment valider proprement des données entrantes.

**NOTE** intégrer le dessin XKCD avec Little Bobby Table sur l'assainissement des données en entrée :-p

Quand on parle de `forms`, on ne parle pas uniquement de formulaires Web. On pourrait considérer qu'il s'agit de leur objectif principal, mais on peut également voir un peu plus loin: on peut en fait voir les `forms` comme le point d'entrée pour chaque donnée arrivant dans notre application: il s'agit en quelque sorte d'un ensemble de règles complémentaires à celles déjà présentes au niveau du modèle.

L'exemple le plus simple est un fichier `.csv`: la lecture de ce fichier pourrait se faire de manière très simple, en récupérant les valeurs de chaque colonne et en l'introduisant dans une instance du modèle.

Mauvaise idée.

Les données fournies par un utilisateur **doivent toujours** être validées avant introduction dans la base de données. Notre base de données étant accessible ici par l'ORM, la solution consiste à introduire une couche supplémentaire de validation.

Le flux à suivre est le suivant:

1. Création d'une instance grâce à un dictionnaire
2. Validation des données et des informations reçues
3. Traitement, si la validation a réussi.

Ils jouent également plusieurs rôles:

1. Validation des données, en plus de celles déjà définies au niveau du modèle
2. Contrôle sur le rendu à appliquer aux champs

Ils agissent come une glue entre l'utilisateur et la modélisation de vos structures de données.

## 15.1. Dépendance avec le modèle

Un **form** peut dépendre d'une autre classe Django. Pour cela, il suffit de fixer l'attribut `model` au niveau de la `class Meta` dans la définition.

```python
from django import forms

from wish.models import Wishlist

class WishlistCreateForm(forms.ModelForm):
    class Meta:
        model = Wishlist
        fields = ('name', 'description')
```

De cette manière, notre form dépendra automatiquement des champs déjà déclarés dans la classe Wishlist. Cela suit le principe de DRY <don't repeat yourself>`_, et évite qu'une modification ne pourrisse le code: en testant les deux champs présent dans l'attribut `fields, nous pourrons nous assurer de faire évoluer le formulaire en fonction du modèle sur lequel il se base.

## 15.2. Rendu et affichage

Le formulaire permet également de contrôler le rendu qui sera appliqué lors de la génération de la page. Si les champs dépendent du modèle sur lequel se base le formulaire, ces widgets doivent être initialisés dans l'attribut Meta. Sinon, ils peuvent l'être directement au niveau du champ.

```python
from django import forms
from datetime import date
from .models import Accident

class AccidentForm(forms.ModelForm):
    class Meta:
        model = Accident
        fields = ('gymnast', 'educative', 'date', 'information')
        widgets = {
            'date' : forms.TextInput(
                     attrs={
                        'class' : 'form-control',
                        'data-provide' : 'datepicker',
                        'data-date-format' : 'dd/mm/yyyy',
                        'placeholder' : date.today().strftime("%d/%m/%Y")
                     }),
            'information' : forms.Textarea(
                            attrs={
                                'class' : 'form-control',
                                'placeholder' : 'Context (why, where, ...)'
                            })
```

## 15.3. Squelette par défaut

On a d'un côté le {{ form.as_p }} ou {{ form.as_table }}, mais il y a beaucoup mieux que ça ;-) Voir les templates de Vitor.

## 15.4. Crispy-forms

Comme on l'a vu à l'instant, les forms, en Django, c'est le bien. Cela permet de valider des données reçues en entrée et d'afficher (très) facilement des formulaires à compléter par l'utilisateur.

Par contre, c'est lourd. Dès qu'on souhaite peaufiner un peu l'affichage, contrôler parfaitement ce que l'utilisateur doit remplir, modifier les types de contrôleurs, les placer au pixel près, ... Tout ça demande énormément de temps. Et c'est là qu'intervient `Django-Crispy-Forms <http://django-crispy-forms.readthedocs.io/en/latest/>`_. Cette librairie intègre plusieurs frameworks CSS (Bootstrap, Foundation et uni-form) et permet de contrôler entièrement le **layout** et la présentation.

(c/c depuis le lien ci-dessous)

Pour chaque champ, crispy-forms va :

- utiliser le `verbose_name` comme label.
- vérifier les paramètres `blank` et `null` pour savoir si le champ est obligatoire.
- utiliser le type de champ pour définir le type de la balise `<input>`.
- récupérer les valeurs du paramètre `choices` (si présent) pour la balise `<select>`.

http://dotmobo.github.io/django-crispy-forms.html

# Chapitre 16. Validation des données

**NOTE** parler ici des méthodes `clean`.

# Chapitre 17. En conclusion

1. Toute donnée entrée par l’utilisateur **doit** passer par une instance de `form`.
2. euh ?

# Chapitre 18. Migrations

Les migrations (comprendre les "migrations du schéma de base de données") sont intimement liées à la représentation d’un contexte fonctionnel. L’ajout d’une nouvelle information, d’un nouveau champ ou d’une nouvelle fonction peut s’accompagner de tables de données à mettre à jour ou de champs à étendre.

Toujours dans une optique de centralisation, les migrations sont directement embarquées au niveau du code. Le développeur s’occupe de créer les migrations en fonction des actions à entreprendre; ces migrations peuvent être retravaillées, *squashées*, … et feront partie intégrante du processus de mise à jour de l’application.

# Chapitre 19. Modèle-vue-template

Dans un **pattern** MVC classique, la traduction immédiate du **contrôleur** est une **vue**. Et comme on le verra par la suite, la **vue** est en fait le **template**. Les vues agrègent donc les informations à partir d'un des composants et les font transiter vers un autre. En d'autres mots, la vue sert de pont entre les données gérées par la base et l'interface utilisateur.

## 19.1. Vues

Une vue correspond à un contrôleur dans le pattern MVC. Tout ce que vous pourrez définir au niveau du fichier `views.py` fera le lien entre le modèle stocké dans la base de données et ce avec quoi l'utilisateur pourra réellement interagir (le `template`).

Chaque vue peut etre représentée de deux manières: soit par des fonctions, soit par des classes. Le comportement leur est propre, mais le résultat reste identique. Le lien entre l'URL à laquelle l'utilisateur accède et son exécution est faite au travers du fichier `gwift/urls.py`, comme on le verra par la suite.

### 19.1.1. Function Based Views

Les fonctions (ou `FBV` pour **Function Based Views**) permettent une implémentation classique des contrôleurs. Au fur et à mesure de votre implémentation, on se rendra compte qu'il y a beaucoup de répétitions dans ce type d'implémentation: elles ne sont pas obsolètes, mais dans certains cas, il sera préférable de passer par les classes.

Pour définir la liste des `WishLists` actuellement disponibles, on précédera de la manière suivante:

1. Définition d'une fonction qui va récupérer les objets de type `WishList` dans notre base de données. La valeur de retour sera la construction d'un dictionnaire (le **contexte**) qui sera passé à un template HTML. On démandera à ce template d'effectuer le rendu au travers de la fonction `render`, qui est importée par défaut dans le fichier `views.py`.
2. Construction d'une URL qui permettra de lier l'adresse à l'exécution de la fonction.
3. Définition du squelette.

**Définition de la fonction**

```
# wish/views.py

from django.shortcuts import render
from .models import Wishlist

def wishlists(request):
    w = Wishlist.objects.all()
    return render(request, 'wish/list.html',{ 'wishlists': w })
```

### Construction de l’URL

```
# gwift/urls.py

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

from wish import views as wish_views

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', wish_views.wishlists, name='wishlists'),
]
```

### Définition du squelette

A ce stade, vérifiez que la variable `TEMPLATES` est correctement initialisée dans le fichier `gwift/settings.py` et que le fichier `templates/wish/list.html` ressemble à ceci:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <title></title>
    </head>
    <body>
        <p>Mes listes de souhaits</p>
        <ul>
        {% for wishlist in wishlists %}
            <li>{{ wishlist.name }}: {{ wishlist.description }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

### Exécution

A présent, ajoutez quelques listes de souhaits grâce à un **shell**, puis lancez le serveur:

```
$ python manage.py shell
>>> from wish.models import Wishlist
>>> Wishlist.create('Décembre', "Ma liste pour les fêtes de fin d'année")
<Wishlist: Wishlist object>
>>> Wishlist.create('Anniv 30 ans', "Je suis vieux! Faites des dons!")
<Wishlist: Wishlist object>
```

Lancez le serveur grâce à la commande `python manage.py runserver`, ouvrez un navigateur quelconque et rendez-vous à l'adresse `http://localhost:8000 <http://localhost:8000>`_. Vous devriez obtenir le résultat suivant:

a. image:: mvc/my-first-wishlists.png :align: center

Rien de très sexy, aucune interaction avec l'utilisateur, très peu d'utilisation des variables contextuelles, mais c'est un bon début! =)

## 19.1.2. Class Based Views

Les classes, de leur côté, implémente le **pattern** objet et permettent d'arriver facilement à un résultat en très peu de temps, parfois même en définissant simplement quelques attributs, et rien d'autre. Pour l'exemple, on va définir deux classes qui donnent exactement le même résultat que la fonction `wishlists` ci-dessus. Une première fois en utilisant une classe générique vierge, et ensuite en utilisant une classe de type `ListView`.

#### Classe générique

blah

#### ListView

Les classes génériques implémentent un aspect bien particulier de la représentation d'un modèle, en utilisant très peu d'attributs. Les principales classes génériques sont de type `ListView`, [...]. L'implémentation consiste, exactement comme pour les fonctions, à:

1. Définir la classe
2. Créer l'URL
3. Définir le squelette.

```python
# wish/views.py

from django.views.generic import ListView

from .models import Wishlist

class WishListList(ListView):
    context_object_name = 'wishlists'
    model = Wishlist
    template_name = 'wish/list.html'
```

```python
# gwift/urls.py

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

from wish.views import WishListList

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', WishListList.as_view(), name='wishlists'),
]
```

C’est tout. Lancez le serveur, le résultat sera identique. Par inférence, Django construit beaucoup d’informations: si on n’avait pas spécifié les variables `context_object_name` et `template_name`, celles-ci auraient pris les valeurs suivantes:

- `context_object_name`: `wishlist_list` (ou plus précisément, le nom du modèle suivi de `_list`)
- `template_name`: `wish/wishlist_list.html` (à nouveau, le fichier généré est préfixé du nom du modèle). === Templates

Avant de commencer à interagir avec nos données au travers de listes, formulaires et IHM sophistiquées, quelques mots sur les templates: il s’agit en fait de **squelettes** de présentation, recevant en entrée un dictionnaire contenant des clés-valeurs et ayant pour but de les afficher dans le format que vous définirez. En intégrant un ensemble de **tags**, cela vous permettra de greffer les données reçues en entrée dans un patron prédéfini.

Une page HTML basique ressemble à ceci:

```html
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <title></title>
    </head>
    <body>
        <p>Hello world!</p>
    </body>
</html>
```

Notre première vue permettra de récupérer la liste des objets de type `Wishlist` que nous avons définis dans le fichier `wish/models.py`. Supposez que cette liste soit accessible **via** la clé `wishlists` d’un dictionnaire passé au template. Elle devient dès lors accessible grâce aux tags `{% for wishlist in wishlists %}`. A chaque tour de boucle, on pourra directement accéder à la variable `{{ wishlist }}`. De même, il sera possible d’accéder aux propriétés de cette objet de la même manière: `{{ wishlist.id }}`, `{{ wishlist.description }}`, … et d’ainsi respecter la mise en page que nous souhaitons.

En reprenant l'exemple de la page HTML définie ci-dessus, on pourra l'agrémenter de la manière suivante:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <title></title>
    </head>
    <body>
        <p>Mes listes de souhaits</p>
        <ul>
        {% for wishlist in wishlists %}
            <li>{{ wishlist.name }}: {{ wishlist.description }}</li>
        {% endfor %}
        </ul>
    </body>
</html>
```

Vous pouvez déjà copier ce contenu dans un fichier `templates/wsh/list.html`, on en aura besoin par la suite.

### 19.1.3. Structure et configuration

Il est conseillé que les templates respectent la structure de vos différentes applications, mais dans un répertoire à part. Par convention, nous les placerons dans un répertoire `templates`. La hiérarchie des fichiers devient alors celle-ci:

```
$ tree templates/
templates/
└── wish
    └── list.html
```

Par défaut, Django cherchera les templates dans les répertoirer d'installation. Vous devrez vous éditer le fichier `gwift/settings.py` et ajouter, dans la variable `TEMPLATES`, la clé `DIRS` de la manière suivante:

```
TEMPLATES = [
    {
        ...
        'DIRS': [ 'templates' ],
        ...
    },
]
```

### 19.1.4. Builtins

Django vient avec un ensemble de **tags**. On a vu la boucle `for` ci-dessus, mais il existe beaucoup d'autres tags nativement présents. Les principaux sont par exemple:

- `{% if … %} … {% elif … %} … {% else %} … {% endif %}`: permet de vérifier une condition et de n'afficher le contenu du bloc que si la condition est vérifiée.
- Opérateurs de comparaisons: `<`, `>`, `==`, `in`, `not in`.
- Regroupements avec le tag `{% regroup … by … as … %}`.
- `{% url %}` pour construire facilement une URL
- …

### 19.1.5. Non-builtins

En plus des quelques tags survolés ci-dessus, il est également possible de construire ses propres tags. La structure est un peu bizarre, car elle consiste à ajouter un paquet dans une de vos applications, à y définir un nouveau module et à y définir un ensemble de fonctions. Chacune de ces fonctions correspondra à un tag appelable depuis vos templates.

Il existe trois types de tags **non-builtins**:

1. Les filtres - on peut les appeler grâce au **pipe** `|` directement après une valeur dans le template.
2. Les tags simples - ils peuvent prendre une valeur ou plusieurs en paramètre et retourne une nouvelle valeur. Pour les appeler, c'est **via** les tags `{% nom_de_la_fonction param1 param2 … %}`.
3. Les tags d'inclusion: ils retournent un contexte (ie. un dictionnaire), qui est ensuite passé à un nouveau template.

Pour l'implémentation:

1. On prend l'application `wish` et on y ajoute un répertoire `templatetags`, ainsi qu'un fichier `init.py`.
2. Dans ce nouveau paquet, on ajoute un nouveau module que l'on va appeler `tools.py`
3. Dans ce module, pour avoir un aperçu des possibilités, on va définir trois fonctions (une pour chaque type de tags possible).

```
[Inclure un tree du dossier template tags]
```

```
# wish/tools.py

from django import template

from wish.models import Wishlist

register = template.Library()

@register.filter(is_safe=True)
def add_xx(value):
    return '%sxx' % value

@register.simple_tag
def current_time(format_string):
    return datetime.datetime.now().strftime(format_string)

@register.inclusion_tag('wish/templatetags/wishlists_list.html')
def wishlists_list():
    return { 'list': Wishlist.objects.all() }
```

Pour plus d’informations, la [documentation officielle est un bon début](). === Mise en page

Pour que nos pages soient un peu plus **eye-candy** que ce qu’on a présenté ci-dessus, nous allons modifié notre squelette pour qu’il se base sur Bootstrap <http://getbootstrap.com/>`_. Nous placerons une barre de navigation principale, la possibilité de se connecter pour l’utilisateur et définirons quelques emplacements à utiliser par la suite. Reprenez votre fichier `base.html et modifiez le comme ceci:

```
{% load staticfiles %}

<!DOCTYPE html>
<!--[if IE 9]><html class="lt-ie10" lang="en" > <![endif]-->
<html class="no-js" lang="en">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
rel="stylesheet">
    <script src="//code.jquery.com/jquery.min.js"></script>
    <script src=
"//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"></script>
    <link href='https://fonts.googleapis.com/css?family=Open+Sans' rel='stylesheet'
type='text/css'>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/font-
awesome/4.4.0/css/font-awesome.min.css">
    <link href="{% static 'css/style.css' %}" rel="stylesheet">
    <link rel="icon" href="{% static 'img/favicon.ico' %}" />
    <title>Gwift</title>
```

```html
</head>

<body class="base-body">

    <!-- navigation -->
    <div class="nav-wrapper">
        <div id="nav">
            <nav class="navbar navbar-default navbar-static-top navbar-shadow">
                <div class="container-fluid">
                    <div class="navbar-header">
                        <button type="button" class="navbar-toggle" data-
toggle="collapse" data-target="#menuNavbar">
                            <span class="icon-bar"></span>
                            <span class="icon-bar"></span>
                            <span class="icon-bar"></span>
                        </button>
                        <a class="navbar-brand" href="/">
                            <img src="{% static 'img/gwift-20x20.png' %}" />
                        </a>
                    </div>
                    <div class="collapse navbar-collapse" id="menuNavbar">
                        {% include "_menu_items.html" %}
                    </div>
                </div>
            </nav>
        </div>
    </div>
    <!-- end navigation -->

    <!-- content -->
    <div class="container">
        <div class="row">
            <div class="col-md-8">
                {% block content %}{% endblock %}
            </div>
        </div>
    </div>
    <!-- end content -->

    <!-- footer -->
    <footer class="footer">
        {% include "_footer.html" %}
    </footer>
    <!-- end footer -->
</body>
</html>
```

Quelques remarques:

- La première ligne du fichier inclut le **tag** `{% load staticfiles %}`. On y reviendra par la suite, mais en gros, cela permet de faciliter la gestion des fichiers statiques, notamment en les

appelent grâce à la commande `{% static 'img/header.png' %}` ou `{% static 'css/app_style.css' %}`.

- La balise `<head />` est bourée d'appel vers des ressources stockées sur des :abbr:`CDN (Content Delivery Networks)`.
- Les balises `{% block content %} {% endblock %}` permettent de faire hériter du contenu depuis une autre page. On l'utilise notamment dans notre page `templates/wish/list.html`.
- Pour l'entête et le bas de page, on fait appel aux balises `{% include 'nom_du_fichier.html' %}`: ces fichiers sont des fichiers physiques, placés sur le filesystem, juste à côté du fichier `base.html`. De façon bête et méchante, cela inclut juste du contenu HTML. Le contenu des fichiers `_menu_items.html` et `_footer.html` est copié ci-dessous.

```
<!-- gwift/templates/wish/list.html -->

{% extends "base.html" %}

{% block content %}
    <p>Mes listes de souhaits</p>
    <ul>
    {% for wishlist in wishlists %}
        <li>{{ wishlist.name }}: {{ wishlist.description }}</li>
    {% endfor %}
    </ul>
{% endblock %}
```

```
<!-- gwift/templates/_menu_items.html -->
<ul class="nav navbar-nav">
    <li class="">
        <a href="#">
            <i class="fa fa-calendar"></i> Mes listes
        </a>
    </li>
</ul>
<ul class="nav navbar-nav navbar-right">
    <li class="">
        <a href="#">
            <i class="fa fa-user"></i> Login / Register
        </a>
    </li>
</ul>
```

```
<!-- gwift/templates/_footer.html -->
<div class="container">
    Copylefted '16
</div>
```

En fonction de vos affinités, vous pourriez également passer par `PluCSS <http://plucss.pluxml.org/>`, `*Pure* <*http://purecss.io/*>`, `Knacss <http://knacss.com/>`, `*Cascade* <*http://www.cascade-framework.com/*>`, `Semantic <http://semantic-ui.com/>`_ ou `Skeleton <http://getskeleton.com/>`_. Pour notre plus grand bonheur, les frameworks de ce type pullulent. Reste à choisir le bon.

**A priori**, si vous relancez le serveur de développement maintenant, vous devriez déjà voir les modifications… Mais pas les images, ni tout autre fichier statique.

### 19.1.6. Fichiers statiques

Si vous ouvrez la page et que vous lancez la console de développement (F12, sur la majorité des navigateurs), vous vous rendrez compte que certains fichiers ne sont pas disponibles. Il s'agit des fichiers suivants:

- `/static/css/style.css`
- `/static/img/favicon.ico`
- `/static/img/gwift-20x20.png`.

En fait, par défaut, les fichiers statiques sont récupérés grâce à deux handlers:

1. `django.contrib.staticfiles.finders.FileSystemFinder` et `django.contrib.staticfiles.finders.AppDirectoriesFinder`.

En fait, Django va considérer un répertoire `static` à l'intérieur de chaque application. Si deux fichiers portent le même nom, le premier trouvé sera pris. Par facilité, et pour notre développement, nous placerons les fichiers statiques dans le répertoire `gwift/static`. On y trouve donc:

```
[inclure un tree du répertoire gwift/static]
```

Pour indiquer à Django que vous souhaitez aller y chercher vos fichiers, il faut initialiser la `variable <https://docs.djangoproject.com/en/stable/ref/settings/#std:setting-STATICFILES_DIRS>`_ `STATICFILES_DIRS` dans le fichier `settings/base.py`. Vérifiez également que la variable `STATIC_URL` est correctement définie.

```
# gwift/settings/base.py

STATIC_URL = '/static/'
```

```
# gwift/settings/dev.py

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

En production par contre, nous ferons en sorte que le contenu statique soit pris en charge par le front-end Web (Nginx), raison pour laquelle cette variable n'est initialisée que dans le fichier des paramètres liés au développement.

Au final, cela ressemble à ceci:

a. image:: mvc/my-first-wishlists.png :align: center === URLs

La gestion des URLs permet **grosso modo** d'assigner une adresse paramétrée ou non à une fonction Python. La manière simple consiste à modifier le fichier `gwift/settings.py` pour y ajouter nos correspondances. Par défaut, le fichier ressemble à ceci:

```
# gwift/urls.py

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

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
]
```

Le champ `urlpatterns` associe un ensemble d'adresses à des fonctions. Dans le fichier **nu**, seul le **pattern** `admin`_ est défini, et inclut toutes les adresses qui sont définies dans le fichier `admin.site.urls`. Reportez-vous à l'installation de l'environnement: ce fichier contient les informations suivantes:

a. _`admin`: Rappelez-vous de vos expressions régulières: `^` indique le début de la chaîne.
b. code-block:: python

```
# admin.site.urls.py
```

### 19.1.7. Reverse

En associant un nom ou un libellé à chaque URL, il est possible de récupérer sa **traduction**. Cela implique par contre de ne plus toucher à ce libellé par la suite…

Dans le fichier `urls.py`, on associe le libellé `wishlists` à l'URL `r'^$` (c'est-à-dire la racine du site):

```
from wish.views import WishListList

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^$', WishListList.as_view(), name='wishlists'),
]
```

De cette manière, dans nos templates, on peut à présent construire un lien vers la racine avec le

tags suivant:

```
<a href="{% url 'wishlists' %}">{{ yearvar }} Archive</a>
```

De la même manière, on peut également récupérer l'URL de destination pour n'importe quel libellé, de la manière suivante:

```
from django.core.urlresolvers import reverse_lazy

wishlists_url = reverse_lazy('wishlists')
```

**NOTE** Ne pas oublier de parler des sessions. Mais je ne sais pas si c'est le bon endroit.

# Chapitre 20. Logging

Si on veut propager les logs entre applications, il faut bien spécifier l'attribut `propagate`, sans quoi on s'arrêtera au module sans prendre en considération les sous-modules.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
        'file': {
            'level': 'DEBUG',
            'class': 'logging.FileHandler',
            'filename': os.path.join(SRC_DIR, 'log', 'log.txt'),
        },
    },
    'loggers': {
        'mv': {
            'handlers': ['file'],
            'level': 'DEBUG',
            'propagate': True,
        },
    },
}
```

Par exemple:

```
'loggers': {
    'mv': {  # Parent
        'handlers': ['file'],
        'level': 'DEBUG',
    },
    'mv.models': { # Enfant
        'handlers': ['console'],
        'level': 'DEBUG',
        'propagate': True,
    }
},
```

Et dans le fichier `mv/models.py`, on a ceci:

```
logger = logging.getLogger(__name__)
logger.debug('helloworld');
```

Le log sera écrit dans la console **ET** dans le fichier.

Par contre, si on retire l'attribut `propagate: True` (ou qu'on le change en `propagate: False`), le même code ci-dessus n'écrit que dans la console. Simplement parce que le log associé à un package considère par défaut ses enfants, alors que le log associé à un module pas. [Par exemple](https://docs.djangoproject.com/en/2.1/topics/logging/#examples).

# Chapitre 21. Administration

Woké. On va commencer par la **partie à ne *surtout* (*surtout* !!) pas faire en premier dans un projet Django**. Mais on va la faire quand même. La raison principale est que cette partie est tellement puissante et performante, qu'elle pourrait laisser penser qu'il est possible de réaliser une application complète rien qu'en configurant l'administration.

C'est faux.

L'administration est une sorte de tour de contrôle évoluée; elle se base sur le modèle de données programmé et construit dynamiquement les formulaires qui lui est associé. Elle joue avec les clés primaires, étrangères, les champs et types de champs par introspection.

Son problème est qu'elle présente une courbe d'apprentissage asymptotique. Il est **très** facile d'arriver rapidement à un bon résultat, au travers d'un périmètre de configuration relativement restreint. Mais quoi que vous fassiez, il y a un moment où la courbe de paramétrage sera tellement ardue que vous aurez plus facile à développer ce que vous souhaitez ajouter en utilisant les autres concepts de Django.

Elle doit rester dans les mains d'administrateurs ou de gestionnaires, et dans leurs mains à eux uniquement: il n'est pas question de donner des droits aux utilisateurs finaux (même si c'est extrêment tentant durant les premiers tours de roues). Indépendamment de la manière dont vous allez l'utiliser et la configurer, vous finirez par devoir développer une "vraie" application, destinée aux utilisateurs classiques, et répondant à leurs besoins uniquement.

Une bonne idée consiste à développer l'administration dans un premier temps, en **gardant en tête qu'il sera nécessaire de développer des concepts spécifiques**. Dans cet objectif, l'administration est un outil exceptionel, qui permet de valider un modèle, de créer des objets rapidement et de valider les liens qui existent entre eux. C'est un excellent outil de prototypage et de preuve de concept.

## 21.1. Quelques conseils

1. Surchargez la méthode `str(self)` pour chaque classe que vous aurez définie dans le modèle. Cela permettra de construire une représentation textuelle qui représentera l'instance de votre classe. Cette information sera utilisée un peu partout dans le code, et donnera une meilleure idée de ce que l'on manipule. En plus, cette méthode est également appelée lorsque l'administration historisera une action (et comme cette étape sera inaltérable, autant qu'elle soit fixée dans le début).
2. La méthode `get_absolute_url(self)` retourne l'URL à laquelle on peut accéder pour obtenir les détails d'une instance. Par exemple:

```python
def get_absolute_url(self):
    return reverse('myapp.views.details', args=[self.id])
```

1. Les attributs `Meta`:

```python
class Meta:
    ordering = ['-field1', 'field2']
    verbose_name = 'my class in singular'
    verbose_name_plural = 'my class when is in a list!'
```

1. Le titre:
    - Soit en modifiant le template de l'administration
    - Soit en ajoutant l'assignation suivante dans le fichier `urls.py`: `admin.site.site_header = "SuperBook Secret Area`.
2. Prefetch

https://hackernoon.com/all-you-need-to-know-about-prefetching-in-django-f9068ebe1e60?gi=7da7b9d3ad64

https://medium.com/@hakibenita/things-you-must-know-about-django-admin-as-your-app-gets-bigger-6be0b0ee9614

En gros, le problème de l'admin est que si on fait des requêtes imbriquées, on va flinguer l'application et le chargement de la page. La solution consiste à utiliser la propriété `list_select_related` de la classe d'Admin, afin d'appliquer une jointure par défaut et et gagner en performances.

# Go Live !

Pour commencer, nous allons nous concentrer sur la création d'un site ne contenant qu'une seule application, même si en pratique le site contiendra déjà plusieurs applications fournies pas django, comme nous le verrons plus loin.

Pour prendre un exemple concret, nous allons créer un site permettant de gérer des listes de souhaits, que nous appellerons `gwift` (pour `GiFTs and WIshlisTs` :)).

La première chose à faire est de définir nos besoins du point de vue de l'utilisateur, c'est-à-dire ce que nous souhaitons qu'un utilisateur puisse faire avec l'application.

Ensuite, nous pourrons traduire ces besoins en fonctionnalités et finalement effectuer le développement

# Chapitre 22. Besoins utilisateurs

Nous souhaitons développer un site où un utilisateur donné peut créer une liste contenant des souhaits et où d’autres utilisateurs, authentifiés ou non, peuvent choisir les souhaits à la réalisation desquels ils souhaitent participer.

Il sera nécessaire de s’authentifier pour :

- Créer une liste associée à l’utilisateur en cours
- Ajouter un nouvel élément à une liste

Il ne sera pas nécessaire de s’authentifier pour :

- Faire une promesse d’offre pour un élément appartenant à une liste, associée à un utilisateur.

L’utilisateur ayant créé une liste pourra envoyer un email directement depuis le site aux personnes avec qui il souhaite partager sa liste, cet email contenant un lien permettant d’accéder à cette liste.

A chaque souhait, on pourrait de manière facultative ajouter un prix. Dans ce cas, le souhait pourrait aussi être subdivisé en plusieurs parties, de manière à ce que plusieurs personnes puissent participer à sa réalisation.

Un souhait pourrait aussi être réalisé plusieurs fois. Ceci revient à dupliquer le souhait en question.

# Chapitre 23. Besoins fonctionnels

## 23.1. Gestion des utilisateurs

Pour gérer les utilisateurs, nous allons faire en sorte de surcharger ce que Django propose: par défaut, on a une la possibilité de gérer des utilisateurs (identifiés par une adresse email, un nom, un prénom, …) mais sans plus.

Ce qu'on peut souhaiter, c'est que l'utilisateur puisse s'authentifier grâce à une plateforme connue (Facebook, Twitter, Google, etc.), et qu'il puisse un minimum gérer son profil.

## 23.2. Gestion des listes

### 23.2.1. Modèlisation

Les données suivantes doivent être associées à une liste:

- un identifiant
- un identifiant externe (un GUID, par exemple)
- un nom
- une description
- le propriétaire, associé à l'utilisateur qui l'aura créée
- une date de création
- une date de modification

### 23.2.2. Fonctionnalités

- Un utilisateur authentifié doit pouvoir créer, modifier, désactiver et supprimer une liste dont il est le propriétaire
- Un utilisateur doit pouvoir associer ou retirer des souhaits à une liste dont il est le propriétaire
- Il faut pouvoir accéder à une liste, avec un utilisateur authentifier ou non, **via** son identifiant externe
- Il faut pouvoir envoyer un email avec le lien vers la liste, contenant son identifiant externe
- L'utilisateur doit pouvoir voir toutes les listes qui lui appartiennent

## 23.3. Gestion des souhaits

### 23.3.1. Modélisation

Les données suivantes peuvent être associées à un souhait:

- un identifiant

- identifiant de la liste
- un nom
- une description
- le propriétaire
- une date de création
- une date de modification
- une image, afin de représenter l'objet ou l'idée
- un nombre (1 par défaut)
- un prix facultatif
- un nombre de part, facultatif également, si un prix est fourni.

### 23.3.2. Fonctionnalités

- Un utilisateur authentifié doit pouvoir créer, modifier, désactiver et supprimer un souhait dont il est le propriétaire.
- On ne peut créer un souhait sans liste associée
- Il faut pouvoir fractionner un souhait uniquement si un prix est donné.
- Il faut pouvoir accéder à un souhait, avec un utilisateur authentifié ou non.
- Il faut pouvoir réaliser un souhait ou une partie seulement, avec un utilisateur authentifié ou non.
- Un souhait en cours de réalisation et composé de différentes parts ne peut plus être modifié.
- Un souhait en cours de réalisation ou réalisé ne peut plus être supprimé.
- On peut modifier le nombre de fois qu'un souhait doit être réalisé dans la limite des réalisations déjà effectuées.

## 23.4. Gestion des réalisations de souhaits

### 23.4.1. Modélisation

Les données suivantes peuvent être associées à une réalisation de souhait:

- identifiant du souhait
- identifiant de l'utilisateur si connu
- identifiant de la personne si utilisateur non connu
- un commentaire
- une date de réalisation

### 23.4.2. Fonctionnalités

- L'utilisateur doit pouvoir voir si un souhait est réalisé, en partie ou non. Il doit également avoir

un pourcentage de complétion sur la possibilité de réalisation de son souhait, entre 0% et 100%.

- L'utilisateur doit pouvoir voir la ou les personnes ayant réalisé un souhait.
- Il y a autant de réalisation que de parts de souhait réalisées ou de nombre de fois que le souhait est réalisé.

## 23.5. Gestion des personnes réalisants les souhaits et qui ne sont pas connues

### 23.5.1. Modélisation

Les données suivantes peuvent être associées à une personne réalisant un souhait:

- un identifiant
- un nom
- une adresse email facultative

### 23.5.2. Fonctionnalités

```
Modélisation
```

L'ORM de Django permet de travailler uniquement avec une définition de classes, et de faire en sorte que le lien avec la base de données soit géré uniquement de manière indirecte, par Django lui-même. On peut schématiser ce comportement par une classe = une table.

Comme on l'a vu dans la description des fonctionnalités, on va **grosso modo** avoir besoin des éléments suivants:

- Des listes de souhaits
- Des éléments qui composent ces listes
- Des parts pouvant composer chacun de ces éléments
- Des utilisateurs pour gérer tout ceci.

Nous proposons dans un premier temps d'éluder la gestion des utilisateurs, et de simplement se concentrer sur les fonctionnalités principales. Cela nous donne ceci:

a. code-block:: python

```
# wish/models.py
```

```
from django.db import models
```

```
class Wishlist(models.Model):
    pass
```

```
class Item(models.Model):
    pass
```

```
class Part(models.Model):
    pass
```

Les classes sont créées, mais vides. Entrons dans les détails.

Listes de souhaits

Comme déjà décrit précédemment, les listes de souhaits peuvent s'apparenter simplement à un objet ayant un nom et une description. Pour rappel, voici ce qui avait été défini dans les spécifications:

- un identifiant
- un identifiant externe
- un nom
- une description
- une date de création
- une date de modification

Notre classe `Wishlist` peut être définie de la manière suivante:

a. code-block:: python

```
# wish/models.py
```

```
class Wishlist(models.Model):
```

```
name = models.CharField(max_length=255)
description = models.TextField()
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
external_id = models.UUIDField(unique=True, default=uuid.uuid4, editable=False)
```

Que peut-on constater?

- Que s'il n'est pas spécifié, un identifiant `id` sera automatiquement généré et accessible dans le modèle. Si vous souhaitez malgré tout spécifier que ce soit un champ en particulier qui devienne la clé primaire, il suffit de l'indiquer grâce à l'attribut `primary_key=True`.
- Que chaque type de champs (`DateTimeField`, `CharField`, `UUIDField`, etc.) a ses propres paramètres d'initialisation. Il est intéressant de les apprendre ou de se référer à la documentation en cas de doute.

Au niveau de notre modélisation:

- La propriété `created_at` est gérée automatiquement par Django grâce à l'attribut `auto_now_add`: de cette manière, lors d'un **ajout**, une valeur par défaut ("**maintenant**") sera attribuée à cette propriété.
- La propriété `updated_at` est également gérée automatique, cette fois grâce à l'attribut `auto_now` initialisé à `True`: lors d'une **mise à jour**, la propriété se verra automatiquement assigner la valeur du moment présent. Cela ne permet évidemment pas de gérer un historique complet et ne nous dira pas **quels champs** ont été modifiés, mais cela nous conviendra dans un premier temps.
- La propriété `external_id` est de type `UUIDField`. Lorsqu'une nouvelle instance sera instanciée, cette propriété prendra la valeur générée par la fonction `uuid.uuid4()`. **A priori**, chacun des types de champs possède une propriété `default`, qui permet d'initialiser une valeur sur une nouvelle instance.

Souhaits

Nos souhaits ont besoin des propriétés suivantes:

- un identifiant
- l'identifiant de la liste auquel le souhait est lié
- un nom
- une description
- le propriétaire
- une date de création
- une date de modification
- une image permettant de le représenter.
- un nombre (1 par défaut)
- un prix facultatif
- un nombre de part facultatif, si un prix est fourni.

Après implémentation, cela ressemble à ceci:

a. code-block:: python

```
# wish/models.py
```

```
class Wish(models.Model):
```

```
wishlist = models.ForeignKey(Wishlist)
name = models.CharField(max_length=255)
description = models.TextField()
created_at = models.DateTimeField(auto_now_add=True)
updated_at = models.DateTimeField(auto_now=True)
picture = models.ImageField()
numbers_available = models.IntegerField(default=1)
number_of_parts = models.IntegerField(null=True)
estimated_price = models.DecimalField(max_digits=19, decimal_places=2,
                                        null=True)
```

A nouveau, que peut-on constater ?

- Les clés étrangères sont gérées directement dans la déclaration du modèle. Un champ de type `ForeignKey <https://docs.djangoproject.com/en/1.8/ref/models/fields/#django.db.models.ForeignKey>`_ permet de déclarer une relation 1-N entre deux classes. Dans la même veine, une relation 1-1 sera représentée par un champ de type `OneToOneField <https://docs.djangoproject.com/en/1.8/topics/db/examples/one_to_one/>`*, alors qu'une relation N-N utilisera un `ManyToManyField <https://docs.djangoproject.com/en/1.8/topics/db/examples/many_to_many/>`*.
- L'attribut `default` permet de spécifier une valeur initiale, utilisée lors de la construction de l'instance. Cet attribut peut également être une fonction.
- Pour rendre un champ optionnel, il suffit de lui ajouter l'attribut `null=True`.
- Comme cité ci-dessus, chaque champ possède des attributs spécifiques. Le champ `DecimalField` possède par exemple les attributs `max_digits` et `decimal_places`, qui nous permettra de représenter une valeur comprise entre 0 et plus d'un milliard (avec deux chiffres décimaux).
- L'ajout d'un champ de type `ImageField` nécessite l'installation de `pillow` pour la gestion des images. Nous l'ajoutons donc à nos pré-requis, dans le fichier `requirements/base.txt`.

```
Parts
```

Les parts ont besoins des propriétés suivantes:

- un identifiant
- identifiant du souhait
- identifiant de l'utilisateur si connu
- identifiant de la personne si utilisateur non connu

- un commentaire
- une date de réalisation

Elles constituent la dernière étape de notre modélisation et représente la réalisation d'un souhait. Il y aura autant de part d'un souhait que le nombre de souhait à réaliser fois le nombre de part.

Elles permettent à un utilisateur de participer au souhait émis par un autre utilisateur. Pour les modéliser, une part est liée d'un côté à un souhait, et d'autre part à un utilisateur. Cela nous donne ceci:

a. code-block:: python

```
from django.contrib.auth.models import User
```

```
class WishPart(models.Model):
```

```
wish = models.ForeignKey(Wish)
user = models.ForeignKey(User, null=True)
unknown_user = models.ForeignKey(UnknownUser, null=True)
comment = models.TextField(null=True, blank=True)
done_at = models.DateTimeField(auto_now_add=True)
```

La classe `User` référencée au début du snippet correspond à l'utilisateur qui sera connecté. Ceci est géré par Django. Lorsqu'une requête est effectuée et est transmise au serveur, cette information sera disponible grâce à l'objet `request.user`, transmis à chaque fonction ou **Class-based-view**. C'est un des avantages d'un framework tout intégré: il vient **batteries-included** et beaucoup de détails ne doivent pas être pris en compte. Pour le moment, nous nous limiterons à ceci. Par la suite, nous verrons comment améliorer la gestion des profils utilisateurs, comment y ajouter des informations et comment gérer les cas particuliers.

La classe `UnknownUser` permet de représenter un utilisateur non enregistré sur le site et est définie au point suivant.

Utilisateurs inconnus

a. todo:: je supprimerais pour que tous les utilisateurs soient gérés au même endroit.

Pour chaque réalisation d'un souhait par quelqu'un, il est nécessaire de sauver les données suivantes, même si l'utilisateur n'est pas enregistré sur le site:

- un identifiant
- un nom
- une adresse email. Cette adresse email sera unique dans notre base de données, pour ne pas créer une nouvelle occurence si un même utilisateur participe à la réalisation de plusieurs

souhaits.

Ceci nous donne après implémentation:

a. code-block:: python

```
class UnkownUser(models.Model):
```

```
name = models.CharField(max_length=255)
email = models.CharField(email = models.CharField(max_length=255, unique=True)
```

# Chapitre 24. Tests unitaires

## 24.1. Pourquoi s’ennuyer à écrire des tests?

Traduit grossièrement depuis un article sur `https://medium.com <https://medium.com/javascript-scene/what-every-unit-test-needs-f6cd34d9836d#.kfyvxyb21>`_:

```
Vos tests sont la première et la meilleure ligne de défense contre les défauts de
programmation. Ils sont
```

```
Les tests unitaires combinent de nombreuses fonctionnalités, qui en fait une arme
secrète au service d'un développement réussi:
```

1. Aide au design: écrire des tests avant d’écrire le code vous donnera une meilleure perspective sur le design à appliquer aux API.
2. Documentation (pour les développeurs): chaque description d’un test
3. Tester votre compréhension en tant que développeur:
4. Assurance qualité: des tests, 5.

## 24.2. Why Bother with Test Discipline?

Your tests are your first and best line of defense against software defects. Your tests are more important than linting & static analysis (which can only find a subclass of errors, not problems with your actual program logic). Tests are as important as the implementation itself (all that matters is that the code meets the requirement—how it’s implemented doesn’t matter at all unless it’s implemented poorly).

Unit tests combine many features that make them your secret weapon to application success:

1. Design aid: Writing tests first gives you a clearer perspective on the ideal API design.
2. Feature documentation (for developers): Test descriptions enshrine in code every implemented feature requirement.
3. Test your developer understanding: Does the developer understand the problem enough to articulate in code all critical component requirements?
4. Quality Assurance: Manual QA is error prone. In my experience, it’s impossible for a developer to remember all features that need testing after making a change to refactor, add new features, or remove features.
5. Continuous Delivery Aid: Automated QA affords the opportunity to automatically prevent broken builds from being deployed to production.

Unit tests don’t need to be twisted or manipulated to serve all of those broad-ranging goals. Rather, it is in the essential nature of a unit test to satisfy all of those needs. These benefits are all side-

effects of a well-written test suite with good coverage.

## 24.3. What are you testing?

1. What component aspect are you testing?
2. What should the feature do? What specific behavior requirement are you testing?

## 24.4. Couverture de code

On a vu au chapitre 1 qu'il était possible d'obtenir une couverture de code, c'est-à-dire un pourcentage.

## 24.5. Comment tester ?

Il y a deux manières d'écrire les tests: soit avant, soit après l'implémentation. Oui, idéalement, les tests doivent être écrits à l'avance. Entre nous, on ne va pas râler si vous faites l'inverse, l'important étant que vous le fassiez. Une bonne métrique pour vérifier l'avancement des tests est la couverture de code.

Pour l'exemple, nous allons écrire la fonction `percentage_of_completion` sur la classe `Wish`, et nous allons spécifier les résultats attendus avant même d'implémenter son contenu. Prenons le cas où nous écrivons la méthode avant son test:

```python
class Wish(models.Model):

    [...]

    @property
    def percentage_of_completion(self):
        """
        Calcule le pourcentage de complétion pour un élément.
        """
        number_of_linked_parts = WishPart.objects.filter(wish=self).count()
        total = self.number_of_parts * self.numbers_available
        percentage = (number_of_linked_parts / total)
        return percentage * 100
```

Lancez maintenant la couverture de code. Vous obtiendrez ceci:

```
$ coverage run --source "." src/manage.py test wish
$ coverage report

Name                              Stmts   Miss Branch BrPart  Cover
--------------------------------------------------------------------
src\gwift\__init__.py                 0      0      0      0   100%
src\gwift\settings\__init__.py        4      0      0      0   100%
src\gwift\settings\base.py           14      0      0      0   100%
src\gwift\settings\dev.py             8      0      2      0   100%
src\manage.py                         6      0      2      1    88%
src\wish\__init__.py                  0      0      0      0   100%
src\wish\admin.py                     1      0      0      0   100%
src\wish\models.py                   36      5      0      0    88%
--------------------------------------------------------------------
TOTAL                                69      5      4      1    93%
```

Si vous générez le rapport HTML avec la commande `coverage html` et que vous ouvrez le fichier `coverage_html_report/src_wish_models_py.html`, vous verrez que les méthodes en rouge ne sont pas testées. **A contrario**, la couverture de code atteignait **98%** avant l'ajout de cette nouvelle méthode.

Pour cela, on va utiliser un fichier `tests.py` dans notre application `wish`. **A priori**, ce fichier est créé automatiquement lorsque vous initialisez une nouvelle application.

```python
from django.test import TestCase

class TestWishModel(TestCase):
    def test_percentage_of_completion(self):
        """
        Vérifie que le pourcentage de complétion d'un souhait
        est correctement calculé.

        Sur base d'un souhait, on crée quatre parts et on vérifie
        que les valeurs s'étalent correctement sur 25%, 50%, 75% et 100%.
        """
        wishlist = Wishlist(name='Fake WishList',
                            description='This is a faked wishlist')
        wishlist.save()

        wish = Wish(wishlist=wishlist,
                    name='Fake Wish',
                    description='This is a faked wish',
                    number_of_parts=4)
        wish.save()

        part1 = WishPart(wish=wish, comment='part1')
        part1.save()
        self.assertEqual(25, wish.percentage_of_completion)

        part2 = WishPart(wish=wish, comment='part2')
        part2.save()
        self.assertEqual(50, wish.percentage_of_completion)

        part3 = WishPart(wish=wish, comment='part3')
        part3.save()
        self.assertEqual(75, wish.percentage_of_completion)

        part4 = WishPart(wish=wish, comment='part4')
        part4.save()
        self.assertEqual(100, wish.percentage_of_completion)
```

L'attribut `@property` sur la méthode `percentage_of_completion()` va nous permettre d'appeler directement la méthode `percentage_of_completion()` comme s'il s'agissait d'une propriété de la classe, au même titre que les champs `number_of_parts` ou `numbers_available`. Attention que ce type de méthode contactera la base de données à chaque fois qu'elle sera appelée. Il convient de ne pas surcharger ces méthodes de connexions à la base: sur de petites applications, ce type de comportement a très peu d'impacts, mais ce n'est plus le cas sur de grosses applications ou sur des méthodes fréquemment appelées. Il convient alors de passer par un mécanisme de **cache**, que nous aborderons plus loin.

En relançant la couverture de code, on voit à présent que nous arrivons à 99%:

```
$ coverage run --source='.' src/manage.py test wish; coverage report; coverage html;
.
----------------------------------------------------------------------
Ran 1 test in 0.006s

OK
Creating test database for alias 'default'...
Destroying test database for alias 'default'...
Name                                 Stmts   Miss Branch BrPart  Cover
------------------------------------------------------------------
src\gwift\__init__.py                    0      0      0      0   100%
src\gwift\settings\__init__.py           4      0      0      0   100%
src\gwift\settings\base.py              14      0      0      0   100%
src\gwift\settings\dev.py                8      0      2      0   100%
src\manage.py                            6      0      2      1    88%
src\wish\__init__.py                     0      0      0      0   100%
src\wish\admin.py                        1      0      0      0   100%
src\wish\models.py                      34      0      0      0   100%
src\wish\tests.py                       20      0      0      0   100%
------------------------------------------------------------------
TOTAL                                   87      0      4      1    99%
```

En continuant de cette manière (ie. Ecriture du code et des tests, vérification de la couverture de code), on se fixe un objectif idéal dès le début du projet. En prenant un développement en cours de route, fixez-vous comme objectif de ne jamais faire baisser la couverture de code.

## 24.6. Quelques liens utiles

- `Django factory boy <https://github.com/rbarrois/django-factory_boy/tree/v1.0.0>`_

# Chapitre 25. A retenir

## 25.1. Constructeurs

Si vous décidez de définir un constructeur sur votre modèle, ne surchargez pas la méthode `init`: créez plutôt une méthode static de type `create()`, en y associant les paramètres obligatoires ou souhaités:

```
class Wishlist(models.Model):

    @staticmethod
    def create(name, description):
        w = Wishlist()
        w.name = name
        w.description = description
        w.save()
        return w

class Item(models.Model):

    @staticmethod
    def create(name, description, wishlist):
        i = Item()
        i.name = name
        i.description = description
        i.wishlist = wishlist
        i.save()
        return i
```

Mieux encore: on pourrait passer par un `ModelManager` pour limiter le couplage; l''accès à une information stockée en base de données ne se ferait dès lors qu'au travers de cette instance et pas directement au travers du modèle. De cette manière, on limite le couplage des classes et on centralise l'accès.

## 25.2. Relations

### 25.2.1. Types de relations

- ForeignKey
- ManyToManyField
- OneToOneField

Dans les examples ci-dessus, nous avons vu les relations multiples (1-N), représentées par des **ForeignKey** d'une classe A vers une classe B. Il existe également les champs de type **ManyToManyField**, afin de représenter une relation N-N. Les champs de type **OneToOneField**, pour représenter une relation 1-1. Dans notre modèle ci-dessus, nous n'avons jusqu'à présent eu

besoin que des relations 1-N: la première entre les listes de souhaits et les souhaits; la seconde entre les souhaits et les parts.

### 25.2.2. Mise en pratique

Dans le cas de nos listes et de leurs souhaits, on a la relation suivante:

```
# wish/models.py

class Wishlist(models.Model):
    pass


class Item(models.Model):
    wishlist = models.ForeignKey(Wishlist)
```

Depuis le code, à partir de l'instance de la classe `Item`, on peut donc accéder à la liste en appelant la propriété `wishlist` de notre instance. **A contrario**, depuis une instance de type `Wishlist`, on peut accéder à tous les éléments liés grâce à `<nom de la propriété>_set`; ici `item_set`.

Lorsque vous déclarez une relation 1-1, 1-N ou N-N entre deux classes, vous pouvez ajouter l'attribut `related_name` afin de nommer la relation inverse.

```
# wish/models.py

class Wishlist(models.Model):
    pass


class Item(models.Model):
    wishlist = models.ForeignKey(Wishlist, related_name='items')
```

A partir de maintenant, on peut accéder à nos propriétés de la manière suivante:

```
# python manage.py shell

>>> from wish.models import Wishlist, Item
>>> w = Wishlist('Liste de test', 'description')
>>> w = Wishlist.create('Liste de test', 'description')
>>> i = Item.create('Element de test', 'description', w)
>>>
>>> i.wishlist
<Wishlist: Wishlist object>
>>>
>>> w.items.all()
[<Item: Item object>]
```

Remarque: si, dans une classe A, plusieurs relations sont liées à une classe B, Django ne saura pas à quoi correspondra la relation inverse. Pour palier à ce problème et pour gagner en cohérence, on fixe alors une valeur à l'attribut `related_name`.

## 25.3. Querysets & managers

- http://stackoverflow.com/questions/12681653/when-to-use-or-not-use-iterator-in-the-django-orm
- https://docs.djangoproject.com/en/1.9/ref/models/querysets/#django.db.models.query.QuerySet.iterator
- http://blog.etianen.com/blog/2013/06/08/django-querysets/

# Chapitre 26. Refactoring

On constate que plusieurs classes possèdent les mêmes propriétés `created_at` et `updated_at`, initialisées aux mêmes valeurs. Pour gagner en cohérence, nous allons créer une classe dans laquelle nous définirons ces deux champs, et nous ferons en sorte que les classes `Wishlist`, `Item` et `Part` en héritent. Django gère trois sortes d'héritage:

- L'héritage par classe abstraite
- L'héritage classique
- L'héritage par classe proxy.

## 26.1. Classe abstraite

L'héritage par classe abstraite consiste à déterminer une classe mère qui ne sera jamais instanciée. C'est utile pour définir des champs qui se répèteront dans plusieurs autres classes et surtout pour respecter le principe de DRY. Comme la classe mère ne sera jamais instanciée, ces champs seront en fait dupliqués physiquement, et traduits en SQL, dans chacune des classes filles.

```python
# wish/models.py

class AbstractModel(models.Model):
    class Meta:
        abstract = True

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)


class Wishlist(AbstractModel):
    pass


class Item(AbstractModel):
    pass


class Part(AbstractModel):
    pass
```

En traduisant ceci en SQL, on aura en fait trois tables, chacune reprenant les champs `created_at` et `updated_at`, ainsi que son propre identifiant:

```
--$ python manage.py sql wish
BEGIN;
CREATE TABLE "wish_wishlist" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "created_at" datetime NOT NULL,
    "updated_at" datetime NOT NULL
)
;
CREATE TABLE "wish_item" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "created_at" datetime NOT NULL,
    "updated_at" datetime NOT NULL
)
;
CREATE TABLE "wish_part" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "created_at" datetime NOT NULL,
    "updated_at" datetime NOT NULL
)
;

COMMIT;
```

## 26.2. Héritage classique

L'héritage classique est généralement déconseillé, car il peut introduire très rapidement un problème de performances: en reprenant l'exemple introduit avec l'héritage par classe abstraite, et en omettant l'attribut `abstract = True`, on se retrouvera en fait avec quatre tables SQL:

- Une table `AbstractModel`, qui reprend les deux champs `created_at` et `updated_at`
- Une table `Wishlist`
- Une table `Item`
- Une table `Part`.

A nouveau, en analysant la sortie SQL de cette modélisation, on obtient ceci:

```
--$ python manage.py sql wish

BEGIN;
CREATE TABLE "wish_abstractmodel" (
    "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
    "created_at" datetime NOT NULL,
    "updated_at" datetime NOT NULL
)
;
CREATE TABLE "wish_wishlist" (
    "abstractmodel_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES
"wish_abstractmodel" ("id")
)
;
CREATE TABLE "wish_item" (
    "abstractmodel_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES
"wish_abstractmodel" ("id")
)
;
CREATE TABLE "wish_part" (
    "abstractmodel_ptr_id" integer NOT NULL PRIMARY KEY REFERENCES
"wish_abstractmodel" ("id")
)
;

COMMIT;
```

Le problème est que les identifiants seront définis et incrémentés au niveau de la table mère. Pour obtenir les informations héritées, nous seront obligés de faire une jointure. En gros, impossible d'obtenir les données complètes pour l'une des classes de notre travail de base sans effectuer un **join** sur la classe mère.

Dans ce sens, cela va encore... Mais imaginez que vous définissiez une classe `Wishlist`, de laquelle héritent les classes `ChristmasWishlist` et `EasterWishlist`: pour obtenir la liste complètes des listes de souhaits, il vous faudra faire une jointure **externe** sur chacune des tables possibles, avant même d'avoir commencé à remplir vos données. Il est parfois nécessaire de passer par cette modélisation, mais en étant conscient des risques inhérents.

## 26.3. Classe proxy

Lorsqu'on définit une classe de type **proxy**, on fait en sorte que cette nouvelle classe ne définisse aucun nouveau champ sur la classe mère. Cela ne change dès lors rien à la traduction du modèle de données en SQL, puisque la classe mère sera traduite par une table, et la classe fille ira récupérer les mêmes informations dans la même table: elle ne fera qu'ajouter ou modifier un comportement dynamiquement, sans ajouter d'emplacements de stockage supplémentaires.

Nous pourrions ainsi définir les classes suivantes:

```python
# wish/models.py

class Wishlist(models.Model):
    name = models.CharField(max_length=255)
    description = models.CharField(max_length=2000)
    expiration_date = models.DateField()

    @staticmethod
    def create(self, name, description, expiration_date=None):
        wishlist = Wishlist()
        wishlist.name = name
        wishlist.description = description
        wishlist.expiration_date = expiration_date
        wishlist.save()
        return wishlist

class ChristmasWishlist(Wishlist):
    class Meta:
        proxy = True

    @staticmethod
    def create(self, name, description):
        christmas = datetime(current_year, 12, 31)
        w = Wishlist.create(name, description, christmas)
        w.save()


class EasterWishlist(Wishlist):
    class Meta:
        proxy = True

    @staticmethod
    def create(self, name, description):
        expiration_date = datetime(current_year, 4, 1)
        w = Wishlist.create(name, description, expiration_date)
        w.save()
```

Gestion des utilisateurs

Dans les spécifications, nous souhaitions pouvoir associer un utilisateur à une liste (**le propriétaire**) et un utilisateur à une part (**le donateur**). Par défaut, Django offre une gestion simplifiée des utilisateurs (pas de connexion LDAP, pas de double authentification, …): juste un utilisateur et un mot de passe. Pour y accéder, un paramètre par défaut est défini dans votre fichier de settings: `AUTH_USER_MODEL`.

Jouons un peu avec la console

# Métamodèle

Sous ce titre franchement pompeux, on va un peu parler de la modélisation du modèle. Quand on prend une classe (par exemple, `Wishlist` que l'on a défini ci-dessus), on voit qu'elle hérite par défaut de `models.Model`. On peut regarder les propriétés définies dans cette classe en analysant le fichier `lib\site-packages\django\models\base.py`. On y voit notamment que `models.Model` hérite de `ModelBase` au travers de `six <https://pypi.python.org/pypi/six>`_ pour la rétrocompatibilité vers Python 2.7.

Cet héritage apporte notamment les fonctions `save()`, `clean()`, `delete()`, ... Bref, toutes les méthodes qui font qu'une instance est sait **comment** interagir avec la base de données. La base d'un `ORM <https://en.wikipedia.org/wiki/Object-relational_mapping>`_, en fait.

D'autre part, chaque classe héritant de `models.Model` possède une propriété `objects`. Comme on l'a vu dans la section **Jouons un peu avec la console**, cette propriété permet d'accéder aux objects persistants dans la base de données.

# Chapitre 27. Supervision des logs

# Chapitre 28. feedbacks utilisateurs

# En bonus

# Chapitre 29. Snippets utiles (et forcément dispensables)

## 29.1. Récupération du dernier tag Git en Python

L'idée ici est simplement de pouvoir afficher le numéro de version ou le hash d'exécution du code, sans avoir à se connecter au dépôt. Cela apporte une certaine transparence, **sous réserve que le code soit géré par Git**. Si vous suivez scrupuleusement les 12 facteurs, la version de l'application déployée n'est plus sensée conserver un lien avec votre dépôt d'origine… Si vous déployez votre code en utilisant un `git fetch` puis un `git checkout <tag_name>`, le morceau de code ci-dessous pourra vous intéresser :-)