Minor swing with Django

Cédric Declerfayt, Fred Pauchet

Table of Contents

Préface

Nous n'allons pas vous mentir: il existe enormément de tutoriaux très bien réalisés sur "*Comment réaliser une application Django*" et autres "*Déployer votre code en 2 minutes*". Nous nous disions juste que ces tutoriaux restaient relativement haut-niveaux 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 sujets 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" et suit ces quelques principes:

- Faible couplage et forte cohésion, pour que chaque composant dispose de son indépendance.
- Moins de code, plus de fonctionnalités.
- Don't repeat yourself: on ne se répète pas !
- Rapidité du développement (après une petite courbe d'apprentissage un peu ardue au début ;-))

Mis côte à côte, l'application de ces principes permet une meilleure stabilité du projet à moyen et long terme. Tout pour plaire à n'importe quel directeur IT.

Dans la première partie, nous verrons comment partir d'un environnement sain, comment le configurer correctement, comment installer Django de manière isolée et comment démarrer un nouveau projet. Nous verrons rapidement comment gérer les dépendances, les versions et comment appliquer et suivre un score de qualité de notre code. Nous verrons aussi que la configuration proposée par défaut par le framework n'est pas idéale dans la majorité des cas.

Pour cela, nous présenterons différents outils, la rédaction de tests unitaires et d'intégration pour limiter les régressions, les règles de nomenclature et de contrôle du contenu, comment partir d'un squelette plus complet, ainsi que les bonnes étapes à suivre pour arriver à un déploiement rapide et fonctionnel avec peu d'efforts.

A la fin de cette partie, vous disposerez d'un code propre et d'un projet fonctionnel (mais encore inutile, parce qu'encore vide).

Dans la deuxième partie, nous détaillerons précisément les étapes de déploiement, avec la description et la configuration de l'infrastructure, des exemples concrets de mise à disposition sur deux distributions principales (Debian et CentOS), sur une **Plateform as a Service**, ainsi que l'utilisation de Docker et Docker-Compose.

Nous aborderons également la supervision et la mise à jour d'une application existante, en respectant les bonnes pratiques d'administration système.

Dans la troisième partie, nous aborderons les grands principes de modélisation, en suivant les lignes de conduites du cadre de travail. Nous aborderons les concepts clés qui permettent à une application de rester maintenable, les formulaires, leurs validations, comment gérer les données en entrée, les migrations de données et l'administration.

Dans la quatrième partie, nous mettrons ces concepts en pratique en présentant le développement de deux "vraies" applications: définition des tables, gestion des utilisateurs, … et mise à disposition!

Et tout ça à un seul et même endroit.[1] Oui. :-)

Bonne lecture.

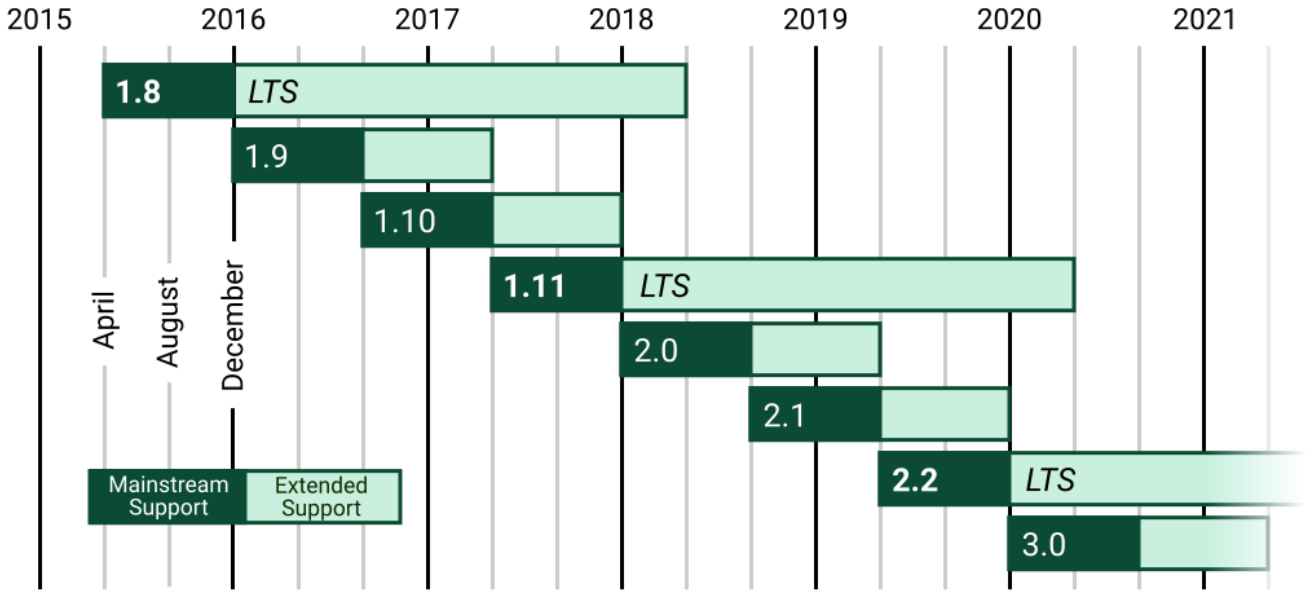
[1] Avec un peu d'XKCD dedans

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 que vous trouverez ci-dessous seront développés pour Python3.6+ et Django 3.0+. Ils nécessiteront peut-être quelques adaptations pour fonctionner sur une version antérieure.

Django fonctionne sur un roulement de trois versions mineures pour une version majeure, clôturé par une version LTS (*Long Term Support*).



La version utilisée sera une bonne indication à prendre en considération pour nos dépendances, puisqu'en visant une version particulière, nous ne devrons pratiquement pas nous soucier (bon, un peu quand même…) des dépendances à installer, pour peu que l'on reste sous un certain seuil.

Dans cette partie, nous allons parler de **méthode de travail**, avec comme objectif d'éviter que l'application ne tourne que sur notre machine et que chaque déploiement ne soit une plaie à gérer. Chaque mise à jour doit être réalisable de la manière la plus simple possible:

1. démarrer un script,
2. prévoir un rollback si cela plante
3. se préparer une tisane en regardant nos flux RSS (si cette technologie existe encore…).

La plupart des commandes qui seront présentées dans ce livre le seront depuis un shell sous GNU/Linux. Certaines d'entre elles pourraient devoir être adaptées si vous utilisez un autre système d'exploitation (macOS) ou n'importe quelle autre grosse bouse commerciale.

Chapitre 1. Construire des applications maintenables

1.1. 12 facteurs

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

L'idée derrière cette méthode, et indépendamment des langages de développement utilisés, consiste à suivre un ensemble de douze concepts, afin de:

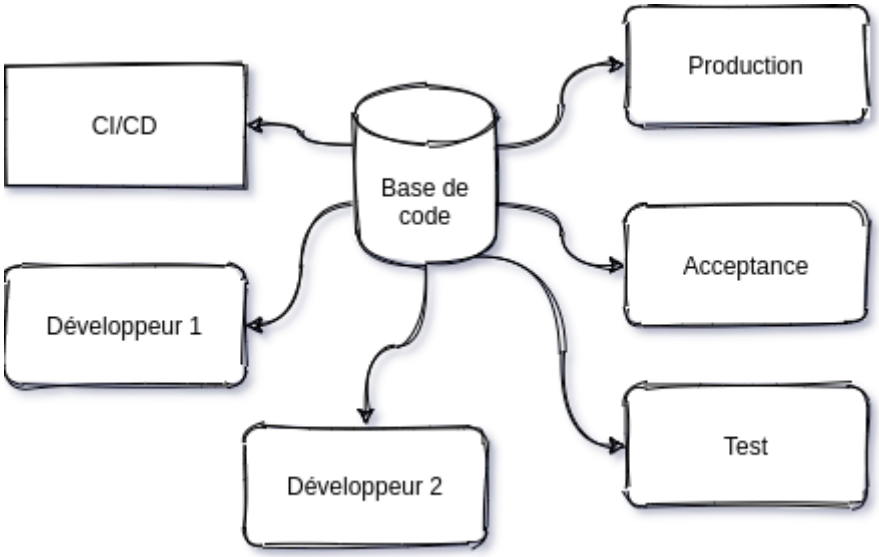
1. **Faciliter la mise en place de phases d'automatisation**; plus concrètement, de faciliter les mises à jour applicatives, simplifier la gestion de l'hôte, diminuer la divergence entre les différents environnements d'exécution et offrir la possibilité d'intégrer le projet dans un processus d'intégration continue ou déploiement continu
2. **Faciliter la mise à pied de nouveaux développeurs ou de personnes souhaitant rejoindre le projet**, dans la mesure où la mise à disposition d'un environnement sera grandement facilitée.
3. **Minimiser les divergences entre les différents environnemens sur lesquels un projet pourrait être déployé**
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 points ci-dessus permettront de monter facilement un nouvel environnement - qu'il soit sur la machine du petit nouveau dans l'équipe, sur un serveur Azure/Heroku/Digital Ocean ou votre nouveau Raspberry Pi Zéro caché à la cave - et vous feront gagner un temps précieux.

Pour reprendre de manière très brute les différentes idées derrière cette méthode, nous avons:

#1 - Une base de code unique, suivie par un système de contrôle de versions.

Chaque déploiement de l'application se basera sur cette source, afin de minimiser les différences que l'on pourrait trouver entre deux environnements d'un même projet. On utilisera un dépôt Git - Github, Gitlab, Gitea, ... Au choix.



#2 - Déclarez explicitement les dépendances nécessaires au projet, et les isoler du reste du système lors de leur installation

Chaque installation ou configuration doit toujours être faite de la même manière, et doit pouvoir être répétée quel que soit l'environnement cible.

Cela permet d'éviter que l'application n'utilise une dépendance qui soit déjà installée sur un des sytèmes de développement, et qu'elle soit difficile, voire impossible, à répercuter sur un autre environnement. Dans notre cas, cela pourra être fait au travers de PIP - Package Installer for Python ou Poetry.

Mais dans tous les cas, chaque application doit disposer d'un environnement sain, qui lui est assigné, et vu le peu de ressources que cela coûte, il ne faut pas s'en priver.

Chaque dépendance pouvant être déclarée et épinglée dans un fichier, il suffira de créer un nouvel environment vierge, puis d'utiliser ce fichier comme paramètre pour installer les prérequis au bon fonctionnement de notre application et vérifier que cet environnement est bien reproductible.

Il est important de bien "épingler" les versions liées aux dépendances de l'application. Cela peut éviter des effets de bord comme une nouvelle version d'une librairie dans laquelle un bug aurait pu avoir été introduit.[2]

#3 - Sauver la configuration directement au niveau de l'environnement

Nous voulons éviter d'avoir à recompiler/redéployer l'application parce que:

1. l'adresse du serveur de messagerie a été modifiée,
2. un protocole a changé en cours de route
3. la base de données a été déplacée
4. ...

En pratique, toute information susceptible de modifier un lien applicatif doit se trouver dans un fichier ou dans une variable d'environnement, et doit être facilement modifiable. En allant un pas plus loin, cela permettra de paramétrer facilement un container, en modifiant une variable de

configuration qui spécifierait la base de données sur laquelle l'application devra se connecter.

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, nous ne devons trouver un mot de passe en clair dans le dépôt source ou une valeur susceptible d'évoluer, écrite en dur dans le code.

#4 - Traiter les ressources externes comme des ressources attachées

Nous parlons de bases de données, de services de mise en cache, d'API externes, ... L'application doit être capable d'effectuer des changements au niveau de ces ressources sans que son code ne soit modifié. Nous parlons alors de **ressources attachées**, dont la présence est nécessaire au bon fonctionnement de l'application, mais pour lesquelles le **type** n'est pas obligatoirement défini.

Nous voulons par exemple "une base de données" et "une mémoire cache", et pas "une base MariaDB et une instance Memcached". De cette manière, les ressources peuvent être attachées et détachées d'un déploiement à la volée.

Si une base de données ne fonctionne pas correctement (problème matériel?), l'administrateur pourrait simplement restaurer un nouveau serveur à partir d'une précédente sauvegarde, et l'attacher à l'application sans que le code source ne soit modifié. une 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.

#5 - Séparer proprement les phases de construction, de mise à disposition et d'exécution

1. La **construction** (*build*) convertit un code source en un ensemble de fichiers exécutables, associé à une version et à une transaction dans le système de gestion de sources.
2. La **mise à disposition** (*release*) associe cet ensemble à une configuration prête à être exécutée,
3. tandis que la phase d'**exécution** (*run*) démarre les processus nécessaires au bon fonctionnement de l'application.

Parmi les solutions possibles, nous pourrions nous pourrions nous baser sur les *releases* de Gitea, sur un serveur d'artefacts ou sur Capistrano.

#6 - Les processus d'exécution ne doivent rien connaître ou conserver de l'état de l'application

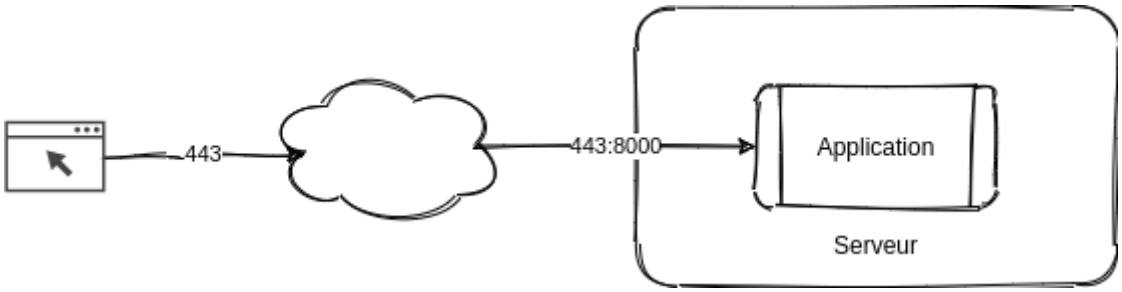
Toute information stockée en mémoire ou sur disque ne doit pas altérer le comportement futur de l'application, par exemple après un redémarrage non souhaité.

Pratiquement, si l'application devait rencontrer un problème, nous pourrions la redémarrer sur un autre serveur. Toute information qui aurait été stockée durant l'exécution de l'application sur le premier hôte serait donc perdue. Si une réinitialisation devait être nécessaire, l'application ne devra pas compter sur la présence d'une information au niveau du nouveau système.

Il serait également difficile d'appliquer une mise à l'échelle de l'application si une donnée indispensable à son fonctionnement devait se trouver sur une seule machine où elle est exécutée.

#7 - Autoriser la liaison d'un port de l'application à un port du système hôte

Les applications 12-factors sont auto-contenues et peuvent fonctionner en autonomie totale. L'idée est qu'elles puissent être joignables grâce à un mécanisme de ponts, où l'hôte effectue la redirection vers l'un des ports ouverts par l'application, typiquement, en HTTP ou via un autre protocole.



#8 - Faites confiance aux processus systèmes pour l'exécution de l'application

Comme décrit plus haut, l'application doit utiliser des processus *stateless* (sans état). Nous pouvons créer et utiliser des processus supplémentaires pour tenir plus facilement une lourde charge, ou dédier des processus particuliers pour certaines tâches: requêtes HTTP *via* des processus Web; *long-running* jobs pour des processus asynchrones, … Si cela existe au niveau du système, ne vous fatiguez pas: utilisez le.

#9 - Améliorer la robustesse de l'application grâce à des arrêts élégants et à des démarrages rapides

Par "arrêt élégant", nous voulons surtout éviter le `kill -9 <pid>` ou tout autre arrêt brutal d'un processus qui nécessiterait une intervention urgente du superviseur. De cette manière, les requêtes en cours pourront se terminer au mieux, tandis que le démarrage rapide de nouveaux processus améliorera la balance d'un processus en cours d'extinction vers des processus tout frais.

L'intégration de ces mécanismes dès les premières étapes de développement limitera les perturbations et facilitera la prise en compte d'arrêts inopinés (problème matériel, redémarrage du système hôte, etc.).

#10 - Conserver les différents environnements aussi similaires que possible, et limiter les divergences entre un environnement de développement et de production

L'exemple donné est un développeur qui utilise macOS, NGinx et SQLite, tandis que l'environnement de production tourne sur une CentOS avec Apache2 et PostgreSQL. L'idée derrière ce concept limite les divergences entre environnements, facilite les déploiements et limite la casse et la découverte de modules non compatibles dès les premières phases de développement.

Pour vous donner un exemple tout bête, SQLite utilise un mécanisme de stockage dynamique, associée à la valeur plutôt qu'au schéma, *via* un système d'affinités. Un autre moteur de base de données définira un schéma statique et rigide, où la valeur sera déterminée par son contenant. Un champ `URLField` proposé par Django a une longeur maximale par défaut de 200 caractères. Si vous faites vos développements sous SQLite et que vous rencontrez une URL de plus de 200 caractères, votre développement sera passera parfaitement bien, mais plantera en production (ou en *staging*, si vous faites les choses un peu mieux) parce que les données seront tronquées…

Conserver des environements similaires limite ce genre de désagréments.

#11 - Gérer les journeaux d'évènements comme des flux

Une application ne doit jamais se soucier de l'endroit où ses évènements seront écrits, mais simplement de les envoyer sur la sortie `stdout`. De cette manière, que nous soyons en développement sur le poste d'un développeur avec une sortie console ou sur une machine de production avec un envoi vers une instance Greylog ou Sentry, le routage des journaux sera réalisé en fonction de sa nécessité et de sa criticité, et non pas parce que le développeur l'a spécifié en dur dans son code.

#12 - Isoler les tâches administratives du reste de l'application

Evitez qu'une migration ne puisse être démarrée depuis une URL de l'application, ou qu'un envoi massif de notifications ne soit accessible pour n'importe quel utilisateur: les tâches administratives ne doivent être accessibles qu'à un administrateur. Les applications 12facteurs favorisent les langages qui mettent un environnement REPL (pour *Read*, *Eval*, *Print* et *Loop*) à disposition (au hasard: Python ou Kotlin), ce qui facilite les étapes de maintenance.

1.2. Bonnes pratiques

Pour cette section, nous nous basons sur un résumé de l'ebook **Building Maintenable Software** disponible chez O'Reilly.

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

- Les méthodes et fonctions
- Les classes
- Les composants
- Et des conseils plus généraux.

Ces conseils sont valables pour n'importe quel langage.

1.2.1. 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 une méthode présente 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, qui sera plus facile à tester.

1.2.2. 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 respectif. 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.

1.2.3. 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'il ne soit possible de 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 pourriez 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 ne soit capable que d'une action. De cette manière, les nouvelles fonctionnalités seront mieux réparties parmi les différents systèmes, et les responsabilités seront plus faciles à gérer. Un conseil est d'avoir un nombre de composants compris entre 6 et 12 (idéalement, 12), et que chacun de ces composants soit approximativement de même taille.

1.2.4. 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**.

1.3. SOLID

1. S : SRP (Single Responsibility
2. O : Open closed
3. L : LSP (Liskov Substitution)
4. I : Interface Segregation

5. D : Dependency Inversion

1.3.1. Single Responsibility Principle

Le principe de responsabilité unique définit que chaque concept ou domaine d'activité ne s'occupe que d'une et d'une seule chose. En prenant l'exemple d'une méthode qui communique avec une base de données, ce ne sera pas à cette méthode à gérer l'inscription d'une exception à un emplacement quelconque. Cette action doit être prise en compte par une autre classe (ou un autre concept), qui s'occupera elle de définir l'emplacement où l'évènement sera enregistré (dans une base de données, une instance Graylog, un fichier, ...).

Cette manière d'organiser le code ajoute une couche de flemmardise (ie. Une fonction ou une méthode doit pouvoir se dire "*I don't care*" et s'occuper uniquement de ses propres oignons) sur certains concepts. Ceci permet de centraliser la configuration d'un type d'évènement à un seul endroit, ce augmente la testabilité du code.

1.3.2. Open Closed

Un des principes essentiels en programmation orientée objets concerne l'héritage de classes et la surcharge de méthodes: plutôt que de partir sur une série de comparaisons pour définir le comportement d'une instance, il est parfois préférable de définir une nouvelle sous-classe, qui surcharge une méthode bien précise. Pour l'exemple, on pourrait ainsi définir trois classes:

- Une classe `Customer`, pour laquelle la méthode `GetDiscount` ne renvoit rien;
- Une classe `SilverCustomer`, pour laquelle la méthode revoit une réduction de 10%;
- Une classe `GoldCustomer`, pour laquelle la même méthode renvoit une réduction de 20%.

Si nous rencontrons un nouveau type de client, il suffit de créer une nouvelle sous-classe. Cela évite d'avoir à gérer un ensemble conséquent de conditions dans la méthode initiale, en fonction d'une autre variable (ici, le type de client).

Nous passerions ainsi de:

```
class Customer():
    def __init__(self, customer_type: str):
        self.customer_type = customer_type


def get_discount(customer: Customer) -> int:
    if customer.customer_type == "Silver":
        return 10
    elif customer.customer_type == "Gold":
        return 20
    return 0


>>> jack = Customer("Silver")
>>> jack.get_discount()
10
```

A ceci:

```
class Customer():
    def get_discount(self) -> int:
        return 0


class SilverCustomer(Customer):
    def get_discount(self) -> int:
        return 10


class GoldCustomer(Customer):
    def get_discount(self) -> int:
        return 20


>>> jack = SilverCustomer()
>>> jack.get_discount()
10
```

En anglais, dans le texte : "Putting in simple words, the “Customer” class is now closed for any new modification but it’s open for extensions when new customer types are added to the project.". En résumé: nous fermons la classe `Customer` à toute modification, mais nous ouvrons la possibilité de créer de nouvelles extensions en ajoutant de nouveaux types [héritant de `Customer`].

De cette manière, nous simplifions également la maintenance de la méthode `get_discount`, dans la mesure où elle dépend directement du type dans lequel elle est implémentée.

Ce point sera très utile lorsque nous aborderons les modèles proxy.

1.3.3. Liskov Substitution

Le principe de substitution fait qu'une classe héritant d'une autre classe doit se comporter de la même manière que cette dernière. Il n'est pas question que la sous-classe n'implémente pas certaines méthodes, alors que celles-ci sont disponibles sa classe parente.

> [...] if S is a subtype of T, then objects of type T in a computer program may be replaced with objects of type S (i.e., objects of type S may be substituted for objects of type T), without altering any of the desirable properties of that program (correctness, task performed, etc.). (Source: Wikipédia).

> Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T. (Source: Wikipédia aussi)

Ce n'est donc pas parce qu'une classe **a besoin d'une méthode définie dans une autre classe** qu'elle doit forcément en hériter. Cela bousillerait le principe de substitution, dans la mesure où une instance de cette classe pourra toujours être considérée comme étant du type de son parent.

Petit exemple pratique: si nous définissons une méthode `walk` et une méthode `eat` sur une classe `Duck`, et qu'une réflexion avancée (et sans doute un peu alcoolisée) nous dit que "Puisqu'un `Lion` marche aussi, faisons le hériter de notre classe `Canard`":

```
class Duck():
    def walk(self):
        print("Kwak")

    def eat(self, thing):
        if thing in ("plant", "insect", "seed", "seaweed", "fish"):
            return "Yummy!"

        raise IndigestionError("Arrrh")

class Lion(Duck):
    def walk(self):
        print("Roaaar!")
```

UnNous vous laissons tester la structure ci-dessus en glissant une antilope dans la boite à goûter du lion, ce qui nous donnera quelques trucs bizarres (et un lion atteint de botulisme).

1.3.4. Interface Segregation

Ce principe stipule qu'un client ne peut en aucun cas dépendre d'une méthode dont il n'a pas besoin. Plus simplement, plutôt que de dépendre d'une seule et même (grosse) interface présentant un ensemble conséquent de méthodes, il est proposé d'exploser cette interface en plusieurs (plus petites) interfaces. Ceci permet aux différents consommateurs de n'utiliser qu'un sous-ensemble précis d'interfaces, répondant chacune à un besoin précis.

Un exemple est d'avoir une interface permettant d'accéder à des éléments. Modifier cette interface pour permettre l'écriture impliquerait que toutes les applications ayant déjà accès à la première, obtiendraient (par défaut) un accès en écriture, ce qui n'est pas souhaité/souhaitable.

Pour contrer ceci, on aurait alors une première interface permettant la lecture, tandis qu'une deuxième (héritant de la première) permettrait l'écriture. On aurait alors le schéma suivant :

- A : lecture
- B (héritant de A) : lecture (par A) et écriture.

1.3.5. Dependency inversion

Dans une architecture conventionnelle, les composants de haut-niveau dépendent directement des composants de bas-niveau. L'inversion de dépendances stipule que c'est le composant de haut-niveau qui possède la définition de l'interface dont il a besoin, et le composant de bas-niveau qui l'implémente.

Le composant de haut-niveau peut définir qu'il s'attend à avoir un `Publisher`, afin de publier du contenu vers un emplacement particulier. Plusieurs implémentation de cette interface peuvent alors être mise en place:

- Une publication par SSH
- Une publication par FTP
- Une publication
- …

L'injection de dépendances est un patron de programmation qui suit le principe d'inversion de dépendances.

1.3.6. Sources

- Understanding SOLID principles on CodeProject
- Software Craftmanship
- Dependency Injection is NOT the same as dependency inversion
- Injection de dépendances

1.4. Complexité de McCabe

La complexité 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:

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

# continue ...
```

TODO: faut vraiment reprendre un cas un peu plus lisible. Là, c'est naze.

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

```
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 cinq:

1. le cas par défaut (a est différent de b, rien ne se passe),
2. le cas où `a` est égal à `b`, mais où `b` est différent de `c`
3. le cas où `a` est égal à `b`, `b` est égal à `c`, mais `c` est différent de `d`
4. le cas où `a` est égal à `b`, `b` est égal à `c`, `c` est égal à `d`, mais `d` est différent de `e`
5. le cas où `a` est égal à `b`, `b` est égal à `c`, `c` est égal à `d` et `d` est égal à `e`

La complexité cyclomatique d'un bloc est évaluée sur base du nombre d'embranchements possibles; par défaut, sa valeur est de 1. Si nous rencontrons une condition, elle passera à 2, etc.

Pour l'exemple ci-dessous, nous allons devoir vérifier au moins chacun des cas pour nous assurer que la couverture est complète. Nous devrions donc trouver:

1. Un test où rien de se passe (`a != b`)
2. Un test pour entrer dans la condition `a == b`
3. Un test pour entrer dans la condition `b == c`
4. Un test pour entrer dans la condition `c == d`
5. Un test pour entrer dans la condition `d == e`

Nous avons 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 fonctionnel 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 noter que refactoriser un bloc pour en extraire une méthode n'améliorera pas la complexité cyclomatique globale de l'application. Mais nous visons ici une amélioration **locale**.

[2] Au conditionnel du futur plus-que-parfait antérieur. Mais ça arrive. Et tout le temps au mauvais moment.

Chapitre 2. Boite à outils

2.1. Python

Le langage Python est un langage de programmation interprété, interactif, orienté objet (souvent), fonctionnel (parfois), open source, multi-plateformes, flexible, facile à apprendre et difficile à maîtriser.

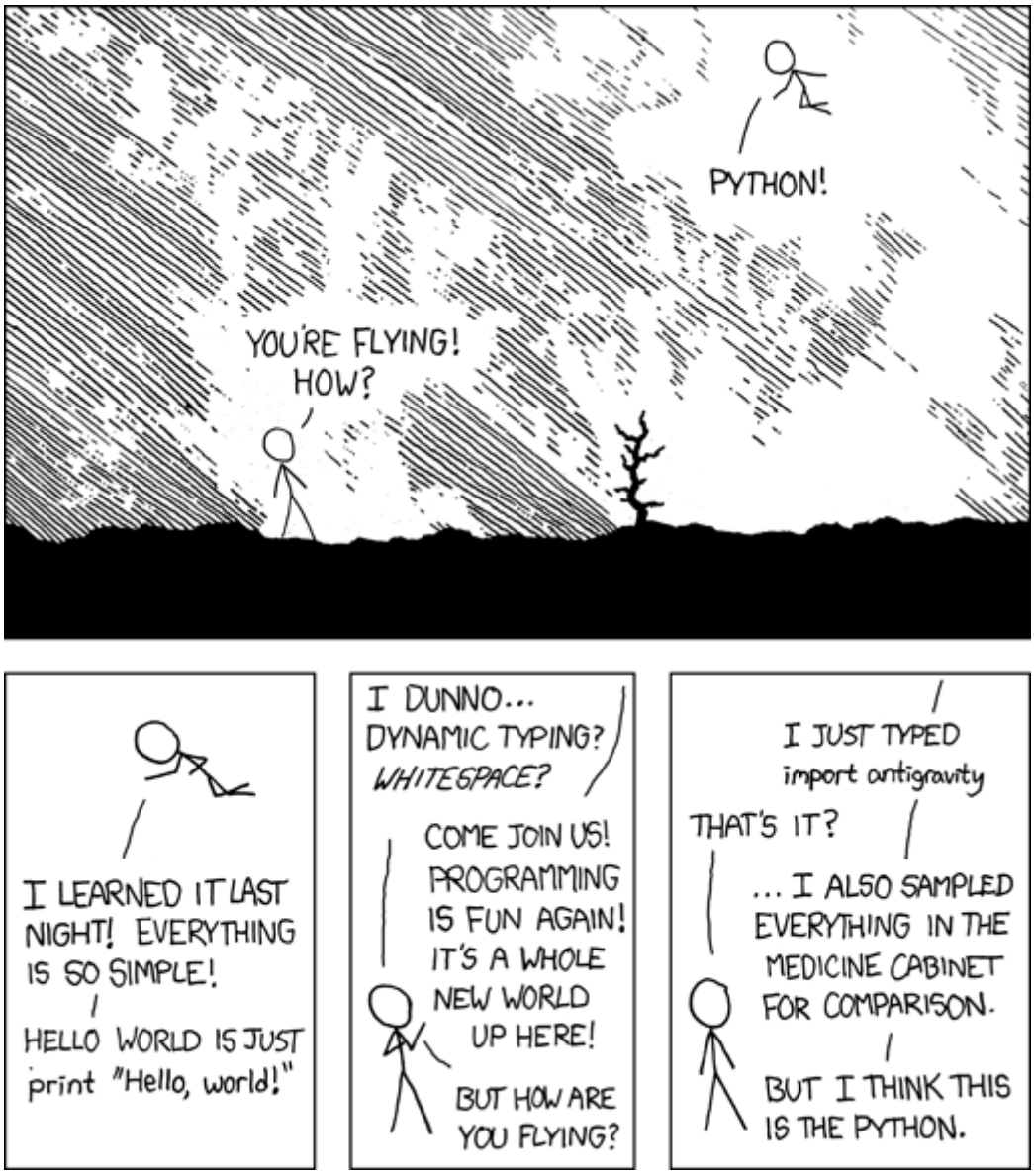


Figure 1. https://xkcd.com/353/

A première vue, certains concepts restent difficiles à aborder: l'indentation définit l'étendue d'un bloc (classe, fonction, méthode, boucle, condition, ...), il n'y a pas de typage fort des variables et le compilateur n'est pas là pour assurer le filet de sécurité avant la mise en production (puisqu'il n'y a pas de compilateur). Et malgré ces quelques points, Python reste un langage généraliste accessible et "bon partout", et de pouvoir se reposer sur un écosystème stable et fonctionnel.

Il 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.

Si vous avez besoin d'un aide-mémoire ou d'une liste exhaustive des types et structures de données du langage, référez-vous au lien suivant: Python Cheat Sheet.

Le langage Python utilise un typage dynamique appelé **duck typing**: "*When I see a bird that quacks like a duck, walks like a duck, has feathers and webbed feet and associates with ducks — I'm certainly going to assume that he is a duck*" (Source: Wikipedia (as usual)).

2.1.1. PEP8 - Style Guide for Python Code

La première PEP qui va nous intéresser 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, afin que d'autres développeurs puissent le reprendre facilement, ou simplement que votre base de code ne dérive lentement vers un seuil de non-maintenabilité.

Dans cet objectif, 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`.

```
$ 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`: cette librairie analysera vos sources à la recherche de sources d'erreurs possibles (imports inutilisés, méthodes inconnues, etc.).

2.1.2. PEP257 - Docstring Conventions

Python étant un langage interprété fortement typé, il est plus que conseillé, au même titre que les tests unitaires que nous verrons plus bas, de documenter son code. Cela impose une certaine rigueur, mais améliore énormément la qualité (et la reprise) du code par une tierce personne. Cela implique aussi de **tout** documenter: les modules, les paquets, les classes, les fonctions, méthodes, ... Tout doit avoir un **docstring** associé :-).

Documentation: be obsessed!

Il existe plusieurs types de conventions de documentation:

1. PEP 257
2. Numpy
3. Google Style (parfois connue sous l'intitulé `Napoleon`)
4. ...

Les conventions proposées par Google nous semblent plus faciles à lire que du RestructuredText, mais sont parfois moins bien intégrées que les docstrings officiellement supportées (typiquement, par exemple par clize qui ne reconnait que du RestructuredText). L'exemple donné dans les styleguide est celui-ci:

```
def fetch_smalltable_rows(table_handle: smalltable.Table,
                          keys: Sequence[Union[bytes, str]],
                          require_all_keys: bool = False,
) -> Mapping[bytes, Tuple[str]]:
    """Fetches rows from a Smalltable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by table_handle.  String keys will be UTF-8 encoded.

    Args:
        table_handle: An open smalltable.Table instance.
        keys: A sequence of strings representing the key of each table
          row to fetch.  String keys will be UTF-8 encoded.
        require_all_keys: Optional; If require_all_keys is True only
          rows with values set for all keys will be returned.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {b'Serak': ('Rigel VII', 'Preparer'),
         b'Zim': ('Irk', 'Invader'),
         b'Lrrr': ('Omicron Persei 8', 'Emperor')}

        Returned keys are always bytes.  If a key from the keys argument is
        missing from the dictionary, then that row was not found in the
        table (and require_all_keys must have been False).

    Raises:
        IOError: An error occurred accessing the smalltable.
    """
```

C'est-à-dire:

1. Une courte ligne d'introduction, descriptive, indiquant ce que la fonction ou la méthode réalise. Attention, la documentation ne doit pas indiquer *comment* la fonction/méthode est implémentée, mais ce qu'elle fait concrètement (et succintement).

2. Une ligne vide
3. Une description plus complète et plus verbeuse
4. Une ligne vide
5. La description des arguments et paramètres, des valeurs de retour (+ exemples) et les exceptions qui peuvent être levées.

Un exemple (encore) plus complet peut être trouvé dans le dépôt sphinxcontrib-napoleon.

Pour ceux que cela pourrait intéresser, il existe une extension pour Codium, comme nous le verrons juste après, qui permet de générer automatiquement le squelette de documentation d'un bloc de code:

Figure 2. autodocstring

Nous le verrons plus loin, Django permet de rendre la documentation immédiatement accessible depuis son interface d'administration.

2.1.3. Linters

Il existe plusieurs niveaux de *linters*:

1. Le premier niveau concerne pycodestyle (anciennement, pep8 justement…), qui analyse votre code à la recherche d'erreurs de convention.
2. Le deuxième niveau concerne pyflakes. Pyflakes est un *simple* [3] programme qui recherchera des erreurs parmi vos fichiers Python.
3. Le troisième niveau est Flake8, qui regroupe les deux premiers niveaux, en plus d'y ajouter flexibilité, extensions et une analyse de complexité de McCabe.
4. Le quatrième niveau [4] est PyLint.

PyLint est le meilleur ami de votre *moi* futur, un peu comme quand vous prenez le temps de faire la vaisselle pour ne pas avoir à la faire le lendemain: il rendra votre code soyeux et brillant, en posant

des affirmations spécifiques. A vous de les traiter en corrigeant le code ou en apposant un *tag* indiquant que vous avez pris connaissance de la remarque, que vous en avez tenu compte, et que vous choisissez malgré tout de faire autrement.

Pour vous donner une idée, voici ce que cela pourrait donner avec un code pas très propre et qui ne sert à rien:

```
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)
```

Avec Flake8, nous obtiendrons 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
```

Nous trouvons 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. Ne pas en tenir compte nuira sans doute à la santé de votre code (et risque de vous réveiller à cinq heures du mat', quand votre application se prendra méchamment les pieds dans le tapis).

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.

TODO: Expliquer comment faire pour tagger une explication.

2.1.4. Formatage de code

Nous avons parlé ci-dessous de style de codage pour Python (PEP8), de style de rédaction pour la documentation (PEP257), d'un *linter* pour nous indiquer quels morceaux de code doivent absolument être revus, … Reste que ces tâches sont ~~parfois~~ (très) souvent fastidieuses: écrire un code propre et systématiquement cohérent est une tâche ardue. Heureusement, il existe des outils pour nous aider (un peu).

A nouveau, il existe plusieurs possibilités de formatage automatique du code. Même si elle n'est pas parfaite, Black arrive à un compromis entre la clarté du code, la facilité d'installation et d'intégration et un résultat.

Est-ce que ce formatage est idéal et accepté par tout le monde ? Non. Même Pylint arrivera parfois à râler. Mais ce formatage conviendra dans 97,83% des cas (au moins).

> By using Black, you agree to cede control over minutiae of hand-formatting. In return, Black gives you speed, determinism, and freedom from pycodestyle nagging about formatting. You will save time and mental energy for more important matters.
>
> Black makes code review faster by producing the smallest diffs possible. Blackened code looks the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead.

Traduit rapidement à partir de la langue de Batman: "*En utilisant Black, vous cédez le contrôle sur le formatage de votre code. En retour, Black vous fera gagner un max de temps, diminuera votre charge mentale et fera revenir l'être aimé*". Mais la partie réellement intéressante concerne le fait que "*Tout code qui sera passé par Black aura la même forme, indépendamment du project sur lequel vous serez en train de travailler. L'étape de formatage deviendra transparente, et vous pourrez vous concentrer sur le contenu*".

2.1.5. Complexité cyclomatique

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.

2.1.6. Typage statique

→ Mypy

2.1.7. Tests unitaires

→ **PyTest**

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`:

```
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.

TODO: Vérifier comment les applications sont construites. Type DRF, Django Social Auth, tout ça.

2.1.8. 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, 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.

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

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

[report]
ignore_errors = True

[html]
directory = coverage_html_report
```

```
$ 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.

2.1.9. Matrice de compatibilité

Décrire un fichier tox.ini

```
$ touch tox.ini
```

2.1.10. Configuration globale

Décrire le fichier setup.cfg

```
$ touch setup.cfg
```

2.1.11. Makefile

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:

```
# 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."
```

2.1.12. The Zen of Python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.2. Environnement de développement

Concrètement, nous pourrions tout à fait nous limiter à Notepad ou Notepad++. Mais à moins d'aimer se fouetter avec un câble USB, nous apprécions la complétion du code, la coloration syntaxique, l'intégration des tests unitaires et d'un debugger, ainsi que deux-trois sucreries qui feront plaisir à n'importe quel développeur.

Si vous manquez d'idées ou si vous ne savez pas par où commencer:

- VSCodium, avec les plugins Pythonet GitLens
- PyCharm
- Vim avec les plugins Jedi-Vim et nerdtree

Si vous hésitez, et même si Codium n'est pas le plus léger (la faute à Electron...), il fera correctement son travail (à savoir: faciliter le vôtre), en intégrant suffisament de fonctionnalités qui gâteront les papilles émoustillées du développeur impatient.

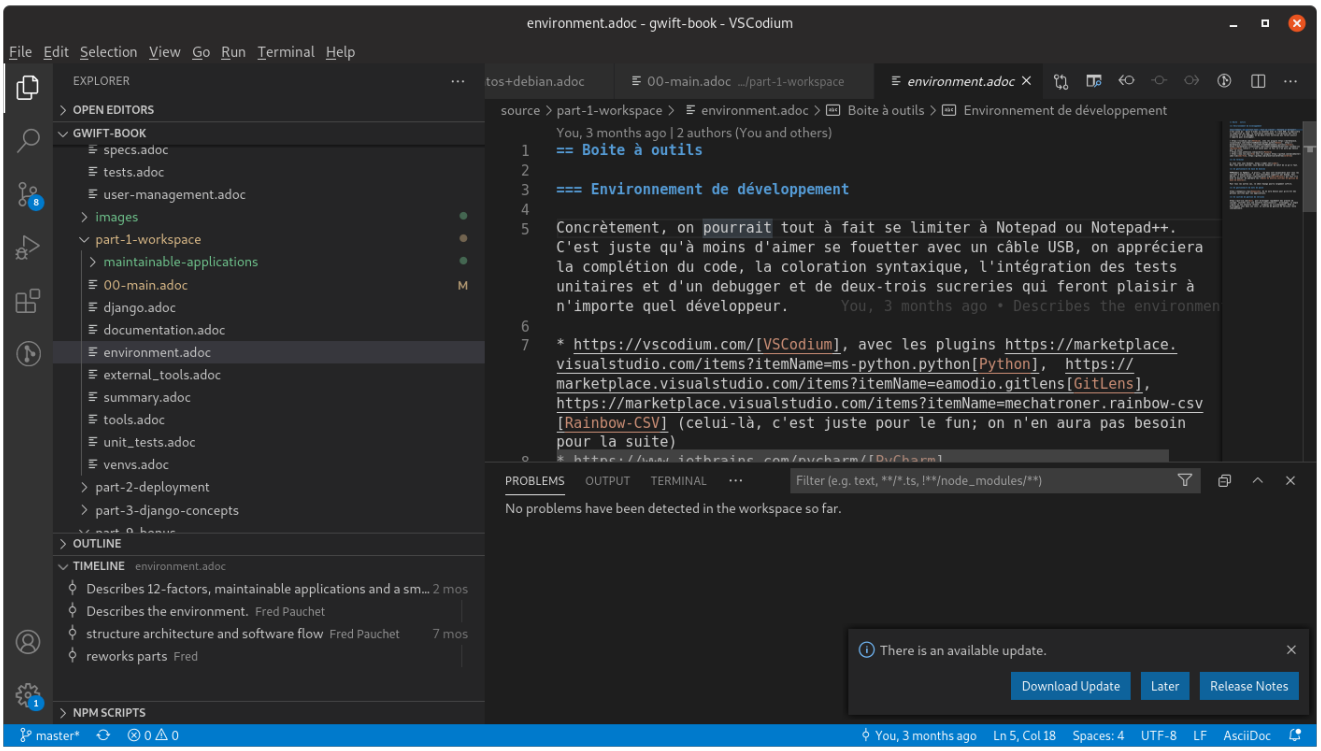


Figure 3. Codium en action

2.3. Un terminal

A priori, les IDE [5] proposés ci-dessus fournissent par défaut ou *via* des greffons un terminal intégré. Ceci dit, disposer d'un terminal séparé facilite parfois certaines tâches.

A nouveau, si vous manquez d'idées:

1. Si vous êtes sous Windows, téléchargez une copie de Cmder. Il n'est pas le plus rapide, mais propose une intégration des outils Unix communs (`ls`, `pwd`, `grep`, `ssh`, `git`, ...) sans trop se fouler.
2. Pour tout autre système, vous devriez disposer en natif de ce qu'il faut.

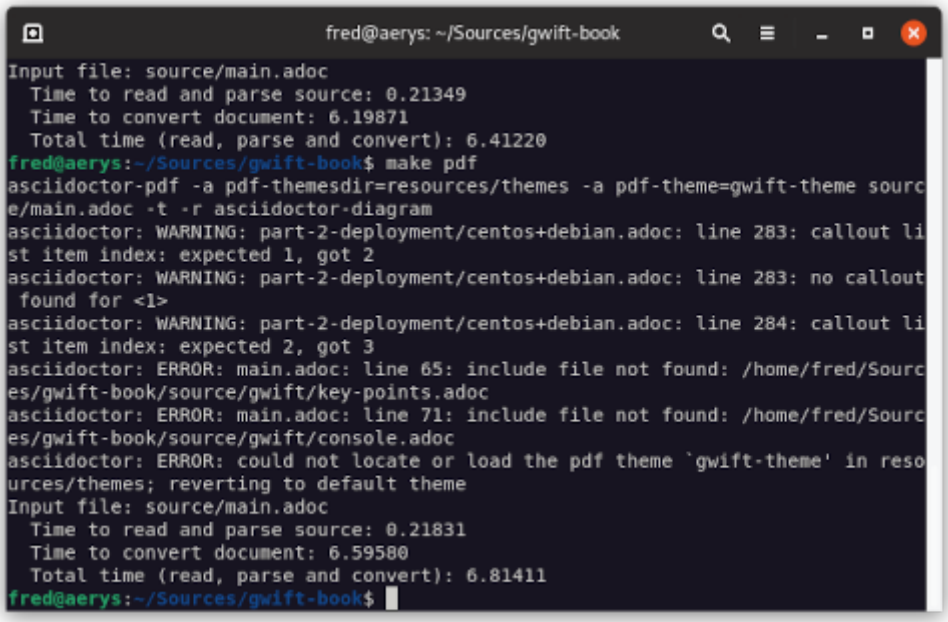


Figure 4. Mise en abîme

2.4. Un gestionnaire de base de données

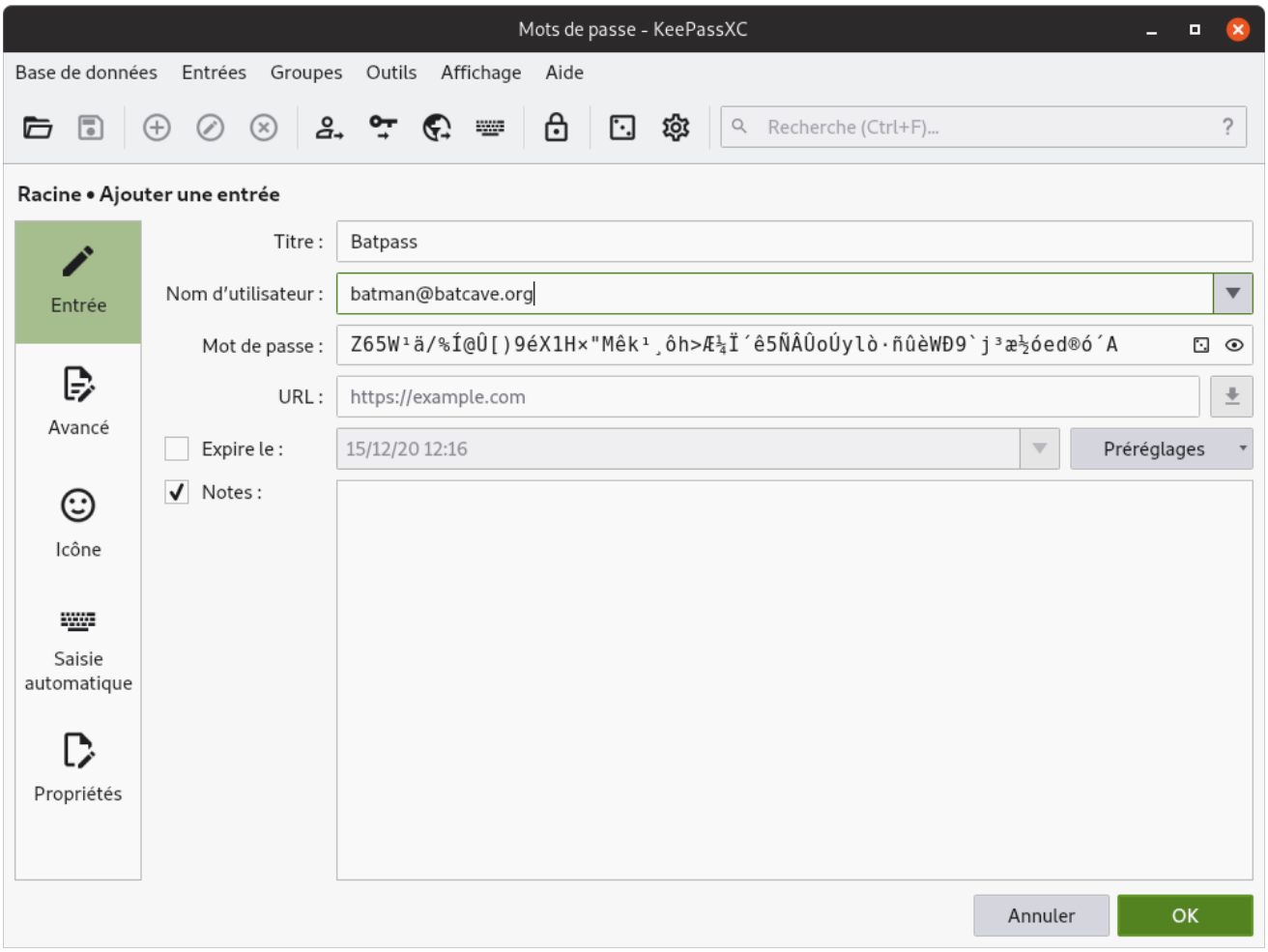
Django gère plusieurs moteurs de base de données. Certains sont gérés nativement par Django (PostgreSQL, MariaDB, SQLite); *a priori*, ces trois-là sont disponibles pour tous les systèmes d'exploitation. D'autres moteurs nécessitent des librairies tierces (Oracle, Microsoft SQL Server).

Il n'est pas obligatoire de disposer d'une application de gestion pour ces moteurs: pour les cas d'utilisation simples, le shell Django pourra largement suffire (nous y reviendrons). Mais pour faciliter la gestion des bases de données elles-même, et si vous n'êtes pas à l'aise avec la ligne de commande, choisissez l'une des applications d'administration ci-dessous en fonction du moteur de base de données que vous souhaitez utiliser.

- Pour **PostgreSQL**, il existe pgAdmin
- Pour **MariaDB** ou **MySQL**, partez sur PHPMyAdmin
- Pour **SQLite**, il existe SQLiteBrowser PHPMyAdmin ou PgAdmin.

2.5. Un gestionnaire de mots de passe

Nous en auront besoin pour gé(né)rer des phrases secrètes pour nos applications. Si vous n'en utilisez pas déjà un, partez sur KeepassXC: il est multi-plateformes, suivi et s'intègre correctement aux différents environnements, tout en restant accessible.



2.6. Un système de gestion de versions

Il existe plusieurs systèmes de gestion de versions. Le plus connu à l'heure actuelle est Git, notamment pour sa (très) grande flexibilité et sa rapidité d'exécution. Il est une aide précieuse pour développer rapidement des preuves de concept, switcher vers une nouvelle fonctionnalité, un bogue à réparer ou une nouvelle release à proposer au téléchargement. Ses deux plus gros défauts concerneraient peut-être sa courbe d'apprentissage pour les nouveaux venus et la complexité des actions qu'il permet de réaliser.

Figure 5. https://xkcd.com/1597/

Même pour un développeur solitaire, un système de gestion de versions (quel qu'il soit) reste indispensable.

Chaque "**branche**" correspond à une tâche à réaliser: un bogue à corriger (*Hotfix A*), une nouvelle fonctionnalité à ajouter ou un "*truc à essayer*" [6] (*Feature A* et *Feature B*). Chaque "**commit**" correspond à une sauvegarde atomique d'un état ou d'un ensemble de modifications cohérentes entre elles.[7] De cette manière, il est beaucoup plus facile pour le développeur de se concenter sur un sujet en particulier, dans la mesure où celui-ci ne doit pas obligatoirement être clôturé pour appliquer un changement de contexte.

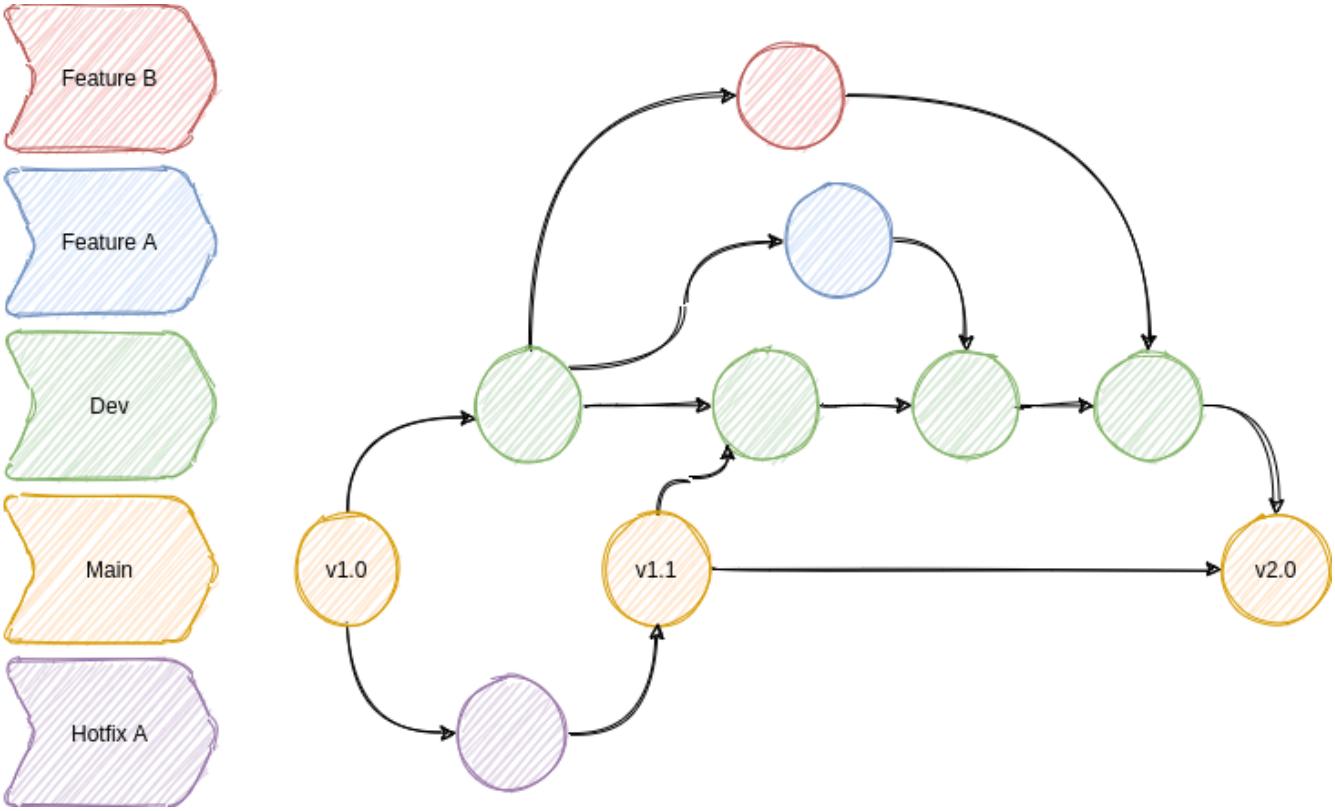


Figure 6. Git en action

Cas pratique: vous développez cette nouvelle fonctionnalité qui va révolutionner le monde de demain et d'après-demain, quand, tout à coup (!), vous vous rendez compte que vous avez perdu votre conformité aux normes PCI parce les données des titulaires de cartes ne sont pas isolées correctement. Il suffit alors de:

1. sauver le travail en cours (`git add . && git commit -m [WIP]`)
2. revenir sur la branche principale (`git checkout main`)
3. créer un "hotfix" (`git checkout -b hotfix/pci-compliance`)
4. solutionner le problème (sans doute un `;` en trop ?)
5. sauver le correctif sur cette branche (`git add . && git commit -m "Did it!"`)
6. récupérer ce correctif sur la branche principal (`git checkout main && git merge hotfix/pci-compliance`)
7. et revenir tranquillou sur votre branche de développement pour fignoler ce générateur de noms de dinosaures rigolos que l'univers vous réclame à cor et à a cri (`git checkout features/dinolol`)

Finalement, sachez qu'il existe plusieurs manières de gérer ces flux d'informations. Les plus connus sont Gitflow et Threeflow.

2.7. Décrire ses changements

→ parler de la bonne structure d'un commit.

[3] Ce n'est pas moi qui le dit, c'est la doc du projet

[4] Oui, en Python, il n'y a que quatre cercles à l'Enfer

[5] Integrated Development Environment

[6] Oui, comme dans "Attends, j'essaie vite un truc, si ça marche, c'est beau."

[7] Il convient donc de s'abstenir de modifier le CSS d'une application et la couche d'accès à la base de données, sous peine de se faire huer par ses relecteurs au prochain stand-up.

Chapitre 3. Un projet Django

3.1. Travailler en isolation

Nous allons aborder la gestion et l'isolation des dépendances. Cette section est aussi utile pour une personne travaillant seule, que pour transmettre les connaissances à un nouveau membre de l'équipe ou pour déployer l'application elle-même.

Il en était déjà question au deuxième point des 12 facteurs: même dans le cas de petits projets, il est déconseillé de s'en passer. Cela évite les déploiements effectués à l'arrache à grand renfort de `sudo` et d'installation globale de dépendances, pouvant potentiellement occasionner des conflits entre les applications déployées:

1. Il est tout à fait envisagable que deux applications différentes soient déployées sur un même hôte, et nécessitent chacune deux versions différentes d'une même dépendance.
2. Pour la reproductibilité d'un environnement spécifique, cela évite notamment les réponses type "Ca juste marche chez moi", puisque la construction d'un nouvel environnement fait partie intégrante du processus de construction et de la documentation du projet; grâce à elle, nous avons la possibilité de construire un environnement sain et d'appliquer des dépendances identiques, quelle que soit la machine hôte.

Dans la suite de ce chapitre, nous allons considérer deux projets différents:

1. Gwift, une application permettant de gérer des listes de souhaits
2. Khana, une application de suivi d'apprentissage pour des élèves ou étudiants.

3.1.1. Environnement virtuel

Depuis la version 3.5 de Python, le module `venv` est la manière recommandée pour créer un environnement virtuel.

Il existe plusieurs autres modules permettant d'arriver au même résultat, avec quelques avantages et inconvénients pour chacun d'entre eux. Le plus prometteur d'entre eux est Poetry, qui dispose d'une interface en ligne de commande plus propre et plus moderne que ce que PIP propose.

Pour créer un nouvel environnement, vous aurez donc besoin:

1. D'une installation de Python - https://www.python.org/
2. D'un terminal - voir le point Un terminal

J'ai pour habitude de conserver mes projets dans un répertoire `~/Sources/` et mes environnements virtuels dans un répertoire `~/.venvs/`. Cette séparation évite que l'environnement virtuel ne se trouve dans le même répertoire que les sources, ou ne soit accidentellement envoyé vers le système de gestion de versions. Dans la suite de ce chapitre, je considérerai ces mêmes répertoires, mais n'hésitez pas à les modifier.

Pur créer notre répertoire de travail et notre environnement virtuel, exécutez les commandes suivantes:

```
mkdir ~/.venvs/
python -m venv ~/.venvs/gwift-venv
```

Ceci aura pour effet de créer un nouveau répertoire (`~/.venvs/gwift-env/`), dans lequel vous trouverez une installation complète de l'interpréteur Python. Votre environnement virtuel est prêt, il n'y a plus qu'à indiquer que nous souhaitons l'utiliser, grâce à l'une des commandes suivantes:

```
# GNU/Linux, macOS
source ~/.venvs/gwift-venv/bin/activate

# MS Windows, avec Cmder
~/.venvs/gwift-venv/Scripts/activate.bat

# Pour les deux
(gwift-env) fred@aerys:~/Sources/.venvs/gwift-env$ ①
```

① Le terminal signale que nous sommes bien dans l'environnement `gwift-env`.

A présent que l'environnement est activé, 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 les curieux, un environnement virtuel n'est jamais qu'un répertoire dans lequel se trouve une installation fraîche de l'interpréteur, vers laquelle pointe les liens symboliques des binaires. Si vous recherchez l'emplacement de l'interpréteur avec la commande `which python`, vous recevrez comme réponse `/home/fred/.venvs/gwift-env/bin/python`.

Pour sortir de l'environnement virtuel, exécutez la commande `deactivate`. Si vous pensez ne plus en avoir besoin, supprimer le dossier. Si nécessaire, il suffira d'en créer un nouveau.

Pour gérer des versions différentes d'une même librairie, il nous suffit de jongler avec autant d'environnements que nécessaires. Une application nécessite une version de Django inférieure à la 2.0 ? On crée un environnement, on l'active et on installe ce qu'il faut.

Cette technique fonctionnera autant pour un poste de développement que sur les serveurs destinés à recevoir notre application.

Par la suite, nous considérerons que l'environnement virtuel est toujours activé, même si `gwift-env` n'est pas indiqué.

3.1.2. Gestion des dépendances, installation de Django et création d'un nouveau projet

Comme nous en avons déjà discuté, PIP est la solution que nous avons choisie pour la gestion de nos dépendances. Pour installer une nouvelle librairie, vous pouvez simplement passer par la commande `pip install <my_awesome_library>`. Dans le cas de Django, et après avoir activé l'environnement, nous pouvons à présent y installer Django. Comme expliqué ci-dessus, la librairie restera indépendante du reste du système, et ne polluera aucun autre projet. nous exécuterons donc la commande suivante:

```
$ source ~/.venvs/gwift-env/bin/activate # ou ~/.venvs/gwift-env/Scrips/activate.bat
pour Windows.
$ pip install django
Collecting django
    Downloading Django-3.1.4
100% |################################|
Installing collected packages: django
Successfully installed django-3.1.4
```

Ici, la commande `pip install django` récupère la **dernière version connue disponible dans les dépôts https://pypi.org/** (sauf si vous en avez définis d'autres. Mais c'est hors sujet). Nous en avons déjà discuté: il est important de bien spécifier la version que vous souhaitez utiliser, sans quoi vous risquez de rencontrer des effets de bord.

L'installation de Django a ajouté un nouvel exécutable: `django-admin`, que l'on peut utiliser pour créer notre nouvel espace de travail. Par la suite, nous utiliserons `manage.py`, qui constitue un **wrapper** autour de `django-admin`.

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

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

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

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

C'est dans 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.

L'utilité de ces fichiers est définie ci-dessous:

- `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.
- `asgi.py` contient la définition de l'interface ASGI, le protocole pour la passerelle asynchrone entre votre application et le serveur Web.
- `wsgi.py` contient la définition de l'interface WSGI, qui permettra à votre serveur Web (Nginx, Apache, ...) de faire un pont vers votre projet.

Indiquer qu'il est possible d'avoir plusieurs structures de dossiers et qu'il n'y a pas de "magie" derrière toutes ces commandes.

Tant que nous y sommes, nous pouvons ajouter un répertoire dans lequel nous stockerons les dépendances et un fichier README:

```
(gwift) $ mkdir requirements
(gwift) $ touch README.md
(gwift) $ tree gwift
gwift
├── gwift
│   ├── asgi.py
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── requirements ①
├── README.md ②
└── manage.py
```

① Ici

② Et là

Comme nous venons d'ajouter une dépendance à notre projet, profitons-en pour 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 environnement de destination:

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

Au début de chaque fichier, il suffit 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, nous allons ajouter `django` dans une version strictement inférieure à la version 3.2 dans le fichier `requirements/base.txt`.

```
$ echo 'django<3.2' > requirements/base.txt
$ echo '-r base.txt' > requirements/prod.txt
$ echo '-r base.txt' > requirements/dev.txt
```

Prenez directement l'habitude de spécifier la version ou les versions compatibles: les librairies que vous utilisez comme dépendances évoluent, de la même manière que vos projets. Pour être sûr et certain le code que vous avez écrit continue à fonctionner, spécifiez la version de chaque librairie de dépendances. Entre deux versions d'une même librairie, des fonctions sont cassées, certaines signatures sont modifiées, des comportements sont altérés, etc. Il suffit de parcourir les pages de *Changements incompatibles avec les anciennes versions dans Django* (par exemple ici pour le passage de la 3.0 à la 3.1) pour réaliser que certaines opérations ne sont pas anodines, et que sans filet de sécurité, c'est le mur assuré. Avec les mécanismes d'intégration continue et de tests unitaires, nous verrons plus loin comment se prémunir d'un changement inattendu.

3.2. Django

Comme nous l'avons vu ci-dessus, `django-admin` permet de créer un nouveau projet. Nous faisons ici une distinction entre un **projet** et une **application**:

- **Un projet** représente l'ensemble des applications, paramètres, pages HTML, middlewares, dépendances, etc., qui font que votre code fait ce qu'il est sensé faire.
- **Une application** est un contexte d'exécution, idéalement autonome, d'une partie du projet.

Pour `gwift`, nous aurons:

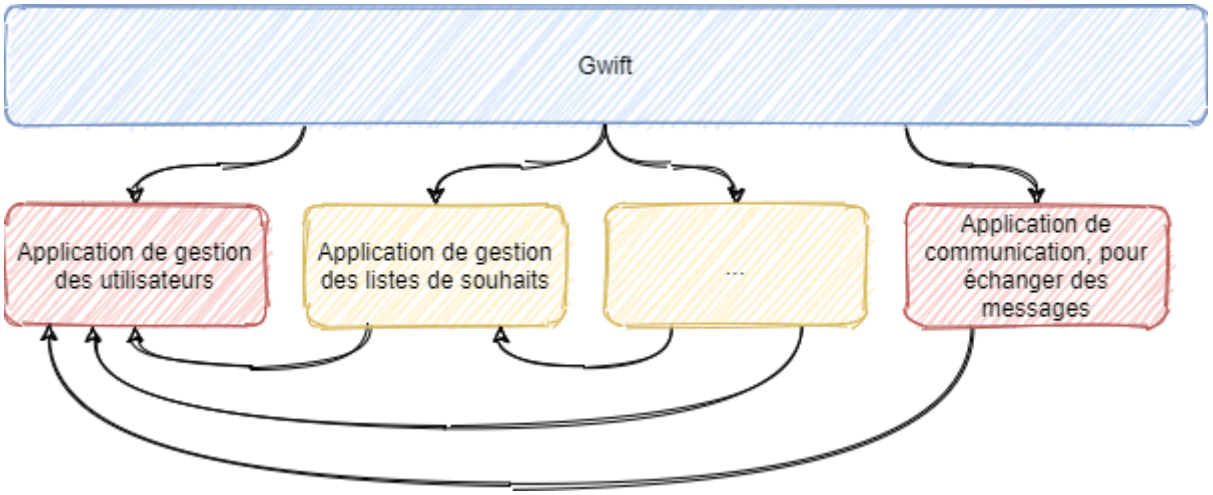


Figure 7. Django Projet vs Applications

1. une première application pour la gestion des listes de souhaits et des éléments,
2. une deuxième application pour la gestion des utilisateurs,
3. voire une troisième application qui gérera les partages entre utilisateurs et listes.

Nous voyons également que la gestion des listes de souhaits et éléments aura besoin de la gestion des utilisateurs - elle n'est pas autonome -, tandis que la gestion des utilisateurs n'a aucune autre dépendance qu'elle-même.

Pour `khana`, nous pourrions avoir quelque chose comme ceci:

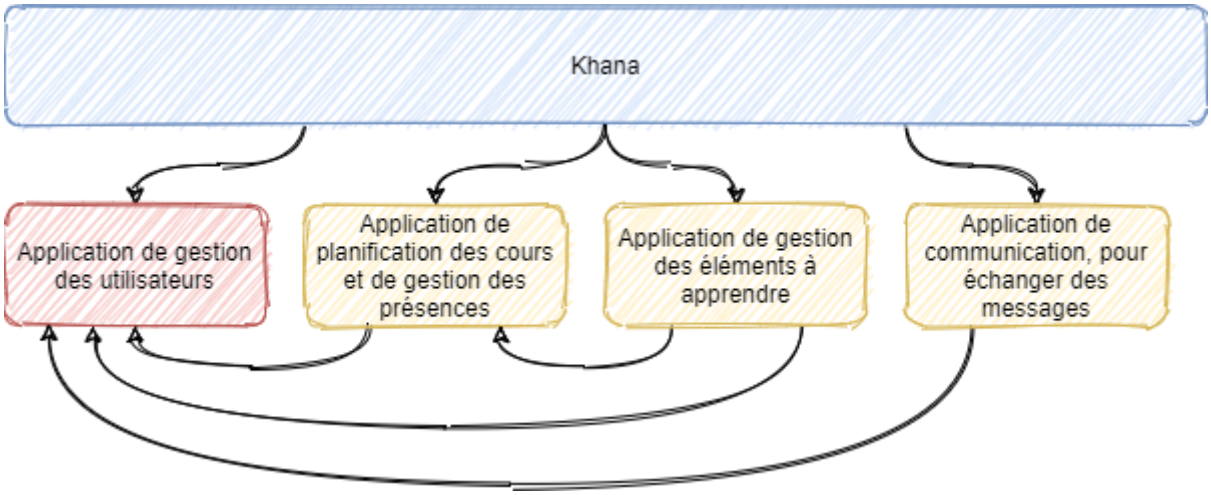


Figure 8. Django Project vs Applications

En rouge, vous pouvez voir quelque chose que nous avons déjà vu: la gestion des utilisateurs et la possibilité qu'ils auront de communiquer entre eux. Ceci pourrait être commun aux deux applications. Nous pouvons clairement visualiser le principe de **contexte** pour une application: celle-ci viendra avec son modèle, ses tests, ses vues et son paramétrage et pourrait ainsi être réutilisée dans un autre projet. C'est en ça que consistent les paquets Django déjà disponibles: ce sont "*simplement*" de petites applications empaquetées et pouvant être réutilisées dans différents contextes (eg. Django-Rest-Framework, Django-Debug-Toolbar, ...).

3.2.1. manage.py

Le fichier `manage.py` que vous trouvez à la racine de votre projet 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 évidente
- `manage.py check --deploy`, pour vérifier (en surface aussi) que l'application est prête pour un déploiement
- `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.

Si nous démarrons la commande `python manage.py runserver`, nous verrons la sortie console suivante:

```
$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

[...]

December 15, 2020 - 20:45:07
Django version 3.1.4, using settings 'gwift.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Si nous nous rendons sur la page http://127.0.0.1:8000 (ou http://localhost:8000) comme le propose si gentiment notre (nouveau) meilleur ami, nous verrons ceci:

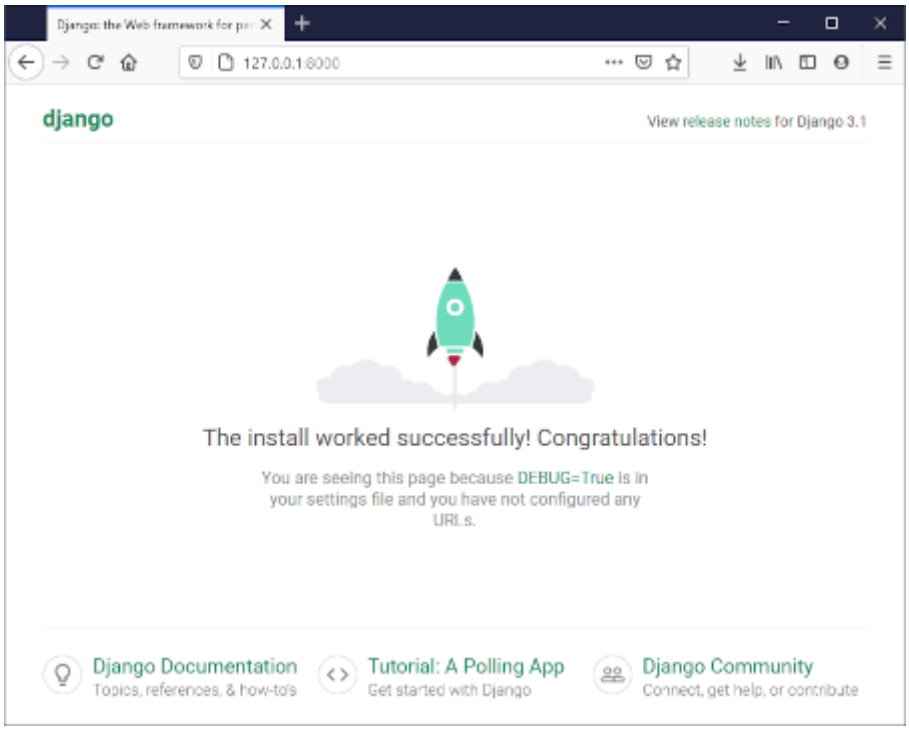


Figure 9. python manage.py runserver (Non, ce n'est pas Challenger)

Nous avons mis un morceau de la sortie console entre crochet `[⋯]` ci-dessus, car elle concerne les migrations. Si vous avez suivi les étapes jusqu'ici, vous avez également dû voir un message type `You have 18 unapplied migration(s). [⋯] Run 'python manage.py migrate' to apply them.` Cela concerne les migrations, et c'est un point que nous verrons un peu plus tard.

3.2.2. Création d'une nouvelle application

Maintenant que nous avons a vu à quoi servait `manage.py`, nous pouvons créer notre nouvelle

application grâce à la commande `manage.py startapp <label>`.

Notre première application servira à structurer les listes de souhaits, les éléments qui les composent et les parties que chaque utilisateur pourra offrir. De manière générale, 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 nous trouvons les fichiers et dossiers suivants:

- `wish/__init__.py` pour que notre répertoire `wish` soit converti en package Python.
- `wish/admin.py` servira à structurer l'administration de notre application. Chaque information peut être administrée facilement au travers d'une interface générée à la volée par le framework. Nous y reviendrons par la suite.
- `wish/migrations/` est le dossier dans lequel seront stockées toutes les différentes migrations de notre application (= toutes les modifications que nous apporterons aux données que nous souhaiterons manipuler)
- `wish/models.py` représentera et structurera nos données, et est intimement lié aux migrations.
- `wish/tests.py` pour les tests unitaires.

Par soucis de clarté, vous pouvez déplacer ce nouveau répertoire `wish` dans votre répertoire `gwift` existant. C'est une forme de convention.

La structure de vos répertoires devient celle-ci:

```
(gwift-env) fred@aerys:~/Sources/gwift$ tree .
.
├── gwift
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── wish ①
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   └── wsgi.py
├── Makefile
├── manage.py
├── README.md
├── requirements
│   ├── base.txt
│   ├── dev.txt
│   └── prod.txt
├── setup.cfg
└── tox.ini

5 directories, 22 files
```

① Notre application a bien été créée, et nous l'avons déplacée dans le répertoire `gwift` !

3.2.3. Fonctionement général

→ diagramme django

3.2.4. 12 facteurs et configuration globale

→ Faire le lien avec les settings → Faire le lien avec les douze facteurs → Construction du fichier setup.cfg

3.3. Structure finale de notre environnement

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

```
(gwift-env) fred@aerys:~/Sources/gwift$ tree .
.
├── gwift
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py
│   ├── urls.py
│   ├── wish ①
│   │   ├── __init__.py
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── migrations
│   │   │   └── __init__.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   └── wsgi.py
├── Makefile
├── manage.py
├── README.md
├── requirements
│   ├── base.txt
│   ├── dev.txt
│   └── prod.txt
├── setup.cfg
└── tox.ini
```

===

3.4. Cookie cutter

- Créez systématiquement un environnement virtuel pour chaque projet sur lequel vous travaillez
- La description des dépendances utilisées pour un projet doivent faire partie intégrante des sources

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

Unresolved directive in part-1-workspace/_main.adoc - include::venvs.adoc[]

Unresolved directive in part-1-workspace/_main.adoc - include::unit_tests.adoc[]

Unresolved directive in part-1-workspace/_main.adoc - include::tools.adoc[]

Unresolved directive in part-1-workspace/_main.adoc - include::external_tools.adoc[]

Unresolved directive in part-1-workspace/_main.adoc - include::summary.adoc[]

Déploiement

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. Il est du coup probable d'oublier une partie des désidérata, de zapper 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 dès le début permet également de rédiger dès le début les procédures d'installation, de mises 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, et la procédure complète se résumera à quelques lignes d'un script bash.

Le serveur que django met à notre disposition *via* la commande `runserver` est extrêmement pratique, mais il est uniquement prévu pour la phase développement: en production, il est inutile de passer par du code Python pour charger des fichiers statiques (feuilles de style, fichiers JavaScript, images, ...). De même, Django propose par défaut une base de données SQLite, qui fonctionne parfaitement dès lors que l'on connait ses limites et que l'on se limite à un utilisateur à la fois. En production, il est légitime que la base de donnée soit capable de supporter plusieurs utilisateurs et connexions simultanément... En restant avec les paramètres par défaut, il est plus que probable que vous rencontriez rapidement des erreurs de verrou parce qu'un autre processus a déjà pris la main pour écrire ses données. En bref, vous avez quelque chose qui fonctionne, mais qui ressemble de très loin à ce dont vous aurez besoin au final.

Dans cette partie, nous aborderons les points suivants:

- Définir l'infrastructure nécessaire à notre application et configurer l'hôte qui hébergera l'application: dans une machine physique, virtuelle ou dans un container. Nous aborderons aussi les déploiements via Ansible et Salt.
- Déployer notre code source
- Configurer les outils nécessaires à la bonne exécution de ce code et de ses fonctionnalités: les différentes méthodes de supervision de l'application, comment analyser les fichiers de logs, comment intercepter correctement une erreur si elle se présente et comment remonter l'information.
- Rendre notre application accessible depuis l'extérieur.

Chapitre 4. Infrastructure

Si on schématise l'infrastructure et le chemin parcouru par une éventuelle requête, nous devrions arriver à quelque chose de synthéthique:

1. l'utilisateur fait une requête via son navigateur (Firefox ou Chrome)
2. le navigateur envoie une requête http, sa version, un verbe (GET, POST, …), un port et éventuellement du contenu
3. le firewall du serveur (Debian GNU/Linux, CentOS, …) vérifie si la requête peut être prise en compte
4. la requête est transmise à l'application qui écoute sur le port (probablement 80 ou 443; et *a priori* Nginx)
5. elle est ensuite transmise par socket et est prise en compte par Gunicorn
6. qui la transmet ensuite à l'un de ses *workers* (= un processus Python)
7. après exécution, une réponse est renvoyée à l'utilisateur.

Chapitre 5. Code source

Au niveau logiciel (la partie mise en subrillance ci-dessus), la requête arrive dans les mains du processus Python, qui doit encore

1. effectuer le routage des données,
2. trouver la bonne fonction à exécuter,
3. récupérer les données depuis la base de données,
4. effectuer le rendu ou la conversion des données,
5. et renvoyer une réponse à l'utilisateur.

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

Chapitre 6. Outils de supervision et de mise à disposition

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

- Nginx comme reverse proxy
- Gunicorn ou Uvicorn comme serveur d'application
- Supervisorctl pour le monitoring
- PostgreSQL ou MariaDB comme base de données.
- Celery et RabbitMQ pour l'exécution de tâches asynchrones
- Redis / Memcache pour la mise à en cache (et pour les sessions ? A vérifier).

Chapitre 7. Méthode de déploiement

Nous allons détailler ci-dessous trois méthodes de déploiement:

- Sur une machine hôte, en embarquant tous les composants sur un même serveur. Ce ne sera pas idéal, puisqu'il ne sera pas possible de configurer un *load balancer*, de routeur plusieurs basées de données, mais ce sera le premier cas de figure.
- Dans des containers, avec Docker-Compose.
- Sur une **Plateforme en tant que Service** (ou plus simplement, **PaaS**), pour faire abstraction de toute la couche de configuration du serveur.

7.1. Sur une machine hôte

La première étape pour la configuration de notre hôte consiste à définir les utilisateurs et groupes de droits. Il est faut absolument éviter de faire tourner une application en tant qu'utilisateur **root**, car la moindre faille pourrait avoir des conséquences catastrophiques.

Une fois que ces utilisateurs seront configurés, nous pourrons passer à l'étape de configuration, qui consistera à:

1. Déployer les sources
2. Démarrer un serveur implémentant une interface WSGI (**Web Server Gateway Interface**), qui sera chargé de créer autant de ~~petits lutins~~ travailleurs que nous le désirerons.
3. Démarrer un superviseur, qui se chargera de veiller à la bonne santé de nos petits travailleurs, et en créer de nouveaux s'il le juge nécessaire
4. Configurer un proxy inverse, qui s'occupera d'envoyer les requêtes d'un utilisateur externe à la machine hôte vers notre serveur applicatif, qui la communiquera à l'un des travailleurs.

La machine hôte peut être louée chez Digital Ocean, Scaleway, OVH, Vultr, ... Il existe des dizaines d'hébergements typés VPS (**Virtual Private Server**). A vous de choisir celui qui vous convient [8].

7.2. Déploiement sur Debian

```
apt update
groupadd --system webapps ①
groupadd --system gunicorn_sockets ②
useradd --system --gid webapps --shell /bin/bash --home /home/gwift gwift ③
mkdir -p /home/gwift ④
chown gwift:webapps /home/gwift ⑤
```

① On ajoute un groupe intitulé `webapps`

② On crée un groupe pour les communications via sockets

③ On crée notre utilisateur applicatif; ses applications seront placées dans le répertoire `/home/gwift`

④ On crée le répertoire home/gwift

⑤ On donne les droits sur le répertoire /home/gwift

7.2.1. Installation des dépendances systèmes

La version 3.6 de Python se trouve dans les dépôts officiels de CentOS. Si vous souhaitez utiliser une version ultérieure, il suffit de l'installer en parallèle de la version officiellement supportée par votre distribution.

Pour CentOS, vous avez donc deux possibilités :

```
yum install python36 -y
```

Ou passer par une installation alternative:

```
sudo yum -y groupinstall "Development Tools"
sudo yum -y install openssl-devel bzip2-devel libffi-devel

wget https://www.python.org/ftp/python/3.8.2/Python-3.8.2.tgz
cd Python-3.8*/
./configure --enable-optimizations
sudo make altinstall ①
```

① **Attention !** Le paramètre `altinstall` est primordial. Sans lui, vous écraserez l'interpréteur initialement supporté par la distribution, et cela pourrait avoir des effets de bord non souhaités.

7.2.2. Préparation de l'environnement utilisateur

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

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 'gwift'

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

7.2.3. Configuration de l'application

```
SECRET_KEY=<set your secret key here> ①
ALLOWED_HOSTS=*
STATIC_ROOT=/var/www/gwift/static
DATABASE= ②
```

① La variable `SECRET_KEY` est notamment utilisée pour le chiffrement des sessions.

② On fait confiance à django_environ pour traduire la chaîne de connexion à la base de données.

7.2.4. Création des répertoires de logs

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

7.2.5. Création du répertoire pour le socket

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

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

Suivi de la création par systemd :

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

7.2.6. Gunicorn

```
#!/bin/bash

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

echo "Starting $NAME as `whoami`"

source /home/gwift/.venvs/gwift/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=-
```

7.2.7. Supervision, keep-alive et autoreload

Pour la supervision, on passe par Supervisor. Il existe d'autres superviseurs,

```
yum install supervisor -y
```

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

```
[program:gwift]
command=/home/gwift/bin/start_gunicorn.sh
user=gwift
stdout_logfile=/var/log/gwift/gwift.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/gwift
chown gwift:nagios /var/log/gwift

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/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/bin/gunicorn config.wsgi:...
           ├─2317 /home/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/bin/gunicorn config.wsgi:...
           ├─2318 /home/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/bin/gunicorn config.wsgi:...
           ├─2321 /home/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/bin/gunicorn config.wsgi:...
           ├─2322 /home/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/bin/gunicorn config.wsgi:...
           └─2323 /home/gwift/.venvs/gwift/bin/python3
/home/gwift/.venvs/gwift/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
```

7.2.8. Ouverture des ports

```
et 443 (HTTPS).
```

```
firewall-cmd --permanent --zone=public --add-service=http ①
firewall-cmd --permanent --zone=public --add-service=https ②
firewall-cmd --reload
```

① On ouvre le port 80, uniquement pour autoriser une connexion HTTP, mais qui sera immédiatement redirigée vers HTTPS

② Et le port 443 (forcément).

7.2.9. Installation d'Nginx

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

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

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

server {
        listen 80;
        server_name <server_name>;
        root /var/www/gwift;
        error_log /var/log/nginx/gwift_error.log;
        access_log /var/log/nginx/gwift_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/ { ②
                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; ③
                proxy_set_header Host $http_host;
                proxy_redirect off;

                proxy_pass http://gwift_app;
        }
}
```

① Ce répertoire sera complété par la commande `collectstatic` que l'on verra plus tard. L'objectif est que les fichiers ne demandant aucune intelligence soit directement servis par Nginx. Cela évite d'avoir un processus Python (relativement lent) qui doive être instancié pour servir un simple fichier statique.

② Afin d'éviter que Django ne reçoive uniquement des requêtes provenant de 127.0.0.1

7.3. Mise à jour

Script de mise à jour.

```
su - <user>
source ~/.venvs/<app>/bin/activate
cd ~/webapps/<app>
git fetch
git checkout vX.Y.Z
pip install -U requirements/prod.txt
python manage.py migrate
python manage.py collectstatic
gunicorn reload -HUP
```

7.4. Configuration des sauvegardes

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

C'est l'utilisateur gwift qui s'en occupe.

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

Et dans le fichier crontab :

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

7.5. Rotation des jounaux

```
/var/log/gwift/* {
        weekly
        rotate 3
        size 10M
        compress
        delaycompress
}
```

Puis on démarre logrotate avec # logrotate -d /etc/logrotate.d/gwift pour vérifier que cela fonctionne correctement.

7.6. Ansible

7.7. Heroku

7.8. Docker-Compose

(c/c Ced' - 2020-01-24)

Ça y est, j'ai fait un test sur mon portable avec docker et cookiecutter pour django.

D'abords, après avoir installer docker-compose et les dépendances sous debian, tu dois t'ajouter dans le groupe docker, sinon il faut être root pour utiliser docker. Ensuite, j'ai relancé mon pc car juste relancé un shell n'a pas suffit pour que je puisse utiliser docker avec mon compte.

Bon après c'est facile, un petit virtualenv pour cookiecutter, suivit d'une installation du template django. Et puis j'ai suivi sans t https://cookiecutter-django.readthedocs.io/en/latest/developing-locally-docker.html

Alors, il télécharge les images, fait un petit update, installe les dépendances de dev, install les requirement pip ...

Du coup, ça prend vite de la place: image.png

L'image de base python passe de 179 à 740 MB. Et là j'en ai pour presque 1,5 GB d'un coup.

Mais par contre, j'ai un python 3.7 direct et postgres 10 sans rien faire ou presque.

le serveur de déploiement ne doit avoir qu'un accès en lecture au dépôt source.

On peut aussi passer par fabric, ansible, chef ou puppet.

[8] Personnellement, j'ai un petit faible pour Hetzner Cloud

Chapitre 8. Supervision

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

1. Sentry via sentry_sdk
2. Nagios
3. LibreNMS
4. Zabbix

Il existe également Munin, Logstash, ElasticSearch et Kibana (ELK-Stack) ou Fluentd.

Chapitre 9. Autres outils

Voir aussi devpi, circus, uswgi, statsd.

See https://mattsegal.dev/nginx-django-reverse-proxy-config.html

Chapitre 10. Ressources

- https://zestedesavoir.com/tutoriels/2213/deployer-une-application-django-en-production/
- Déploiement.
- Let’s Encrypt !

Chapitre 11. Bases de données

On l'a déjà vu, Django se base sur un pattern type ActiveRecords pour la gestion de la persistance des données et supporte les principaux moteurs de bases de données connus:

- SQLite (en natif, mais Django 3.0 exige une version du moteur supérieure ou égale à la 3.8)
- MariaDB (en natif depuis Django 3.0),
- PostgreSQL au travers de psycopg2 (en natif aussi),
- Microsoft SQLServer grâce aux drivers [...à compléter]
- Oracle via cx_Oracle.

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. Par ce 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.

11.1. 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
$$$
```

penser à inclure un bidule pour les backups.

11.2. MariaDB

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

11.3. Microsoft SQL Server

11.4. Oracle

Django

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. C'est un framework Web proposant une très bonne intégration des composants, et une flexibilité bien pensée: chacun des composants permet de définir son contenu de manière poussée, en respectant des contraintes logiques et faciles à retenir.

En restant dans les sentiers battus, votre projet suivra le patron de conception `MVC` (Modèle-Vue-Controleur), avec une petite variante sur les termes utilisés: Django les nomme respectivement Modèle-Template-Vue:

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**.

- Le modèle (`models.py`) fait le lien avec la base de données et permet de définir les champs et leur type à associer à une table. *Grosso modo**, une table SQL correspondra à une classe d'un modèle Django.
- La vue (`views.py`), qui joue le rôle de contrôleur: *a priori*, tous les traitements, la récupération des données, etc. doit passer par ce composant et ne doit (pratiquement) pas être généré à la volée, directement à l'affichage d'une page. En d'autres mots, la vue sert de pont entre les données gérées par la base et l'interface utilisateur.
- Le template, qui s'occupe de la mise en forme: c'est le composant qui va s'occuper de transformer les données en un affichage compréhensible (avec l'aide du navigateur) pour l'utilisateur.

Pour reprendre une partie du schéma précédent, on a une requête qui est émise par un utilisateur. La première étape consiste à trouver une route qui correspond à cette requête, c'est à dire à trouver la correspondance entre l'URL demandée et la fonction qui sera exécutée. Cette fonction correspond au **contrôleur** et s'occupera de construire le **modèle** correspondant.

En simplifiant, Django suit bien le modèle MVC, et toutes ces étapes sont liées ensemble grâce aux différentes routes, définies dans les fichiers `urls.py`.

Chapitre 12. 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 !

12.1. Types de champs

12.2. Clés étrangères et relations

1. ForeignKey
2. ManyToManyField
3. 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.

```
# 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`.

12.3. Querysets et managers

```
NOTE : faudra sortir les queryset du chapitre...
```

- 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/

L'ORM de Django (et donc, chacune des classes qui composent votre modèle) 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.

Ces deux propriétés vont de paire; par défaut, chaque classe de votre modèle propose un attribut `objects`, qui correspond à un manager (ou un gestionnaire, si vous préférez). Ce gestionnaire constitue l'interface par laquelle vous accéderez à la base de données. Mais pour cela, vous aurez aussi besoin d'appliquer certains requêtes ou filtres. Et pour cela, vous aurez besoin des `querysets`, qui consistent en des ... ensembles de requêtes :-).

Si on veut connaître la requête SQL sous-jacente à l'exécution du queryset, il suffit d'appeler la fonction str() sur la propriété `query`:

```
queryset = Wishlist.objects.all()

print(queryset.query)
```

Conditions AND et OR sur un queryset

```
Pour un `AND`, il suffit de chaîner les conditions. ** trouver un exemple ici ** :-)
```

```
Mais en gros : bidule.objects.filter(condition1, condition2)
```

```
Il existe deux autres options : combiner deux querysets avec l'opérateur `&` ou
combiner des Q objects avec ce même opérateur.
```

Soit encore combiner des filtres:

```
from core.models import Wish

Wish.objects ①
Wish.objects.filter(name__icontains="test").filter(name__icontains="too") ②
```

① Ca, c'est notre manager.

② Et là, on chaîne les requêtes pour composer une recherche sur tous les souhaits dont le nom contient (avec une casse insensible) la chaîne "test" et dont le nom contient la chaîne "too".

Pour un 'OR', on a deux options :

1. Soit passer par deux querysets, typiuqment `queryset1 | queryset2`
2. Soit passer par des `Q objects`, que l'on trouve dans le namespace `django.db.models`.

```
from django.db.models import Q

condition1 = Q(...)
condition2 = Q(...)

bidule.objects.filter(condition1 | condition2)
```

L'opérateur inverse (*NOT*)

Idem que ci-dessus : soit on utilise la méthode `exclude` sur le queryset, soit l'opérateur `~` sur un Q object;

Ajouter les sujets suivants :

1. Prefetch
2. select_related

12.4. Aggregate vs. Annotate

https://docs.djangoproject.com/en/3.1/topics/db/aggregation/

12.5. Metamodèle

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 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, 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, au travers d'un `ModelManager`.

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.

```
class Wish(models.Model):
    name = models.CharField(max_length=255)

    class Meta:
        ordering = ('name',) ①
```

① On définit un ordre par défaut, directement au niveau du modèle. Cela ne signifie pas qu'il ne

sera pas possible de modifier cet ordre (la méthode `order_by` existe et peut être chaînée à n'importe quel queryset). D'où l'intérêt de tester ce type de comportement, dans la mesure où un `top 1` dans votre code pourrait être modifié simplement par cette petite information.

Pour sélectionner un objet au pif : `return Category.objects.order_by("?").first()`

Les propriétés de la classe Meta les plus utiles sont les suivates:

- `ordering` pour spécifier un ordre de récupération spécifique.
- `verbose_name` pour indiquer le nom à utiliser au singulier pour définir votre classe
- `verbose_name_plural`, pour le pluriel.

12.6. 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.

A noter que les migrations n'appliqueront de modifications que si le schéma est impacté. Ajouter une propriété `related_name` sur une ForeignKey n'engendrera aucune nouvelle action de migration, puisque ce type d'action ne s'applique que sur l'ORM, et pas directement sur la base de données: au niveau des tables, rien ne change. Seul le code et le modèle sont impactés.

reset migrations.

> ```
> En gros, soit on supprime toutes les migrations (en conservant le fichier
> __init__.py), soit on réinitialise proprement les migrations avec un --fake
> -initial (sous réserve que toutes les personnes qui utilisent déjà le projet s'y
> conforment... Ce qui n'est pas gagné.
> ```

12.7. Shell

12.8. Les validateurs

12.9. A retenir

12.9.1. 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.

```
class ItemManager(...):
    (de mémoire, je ne sais plus exactement :-))
```

Chapitre 13. Shell

Chapitre 14. 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. Mais 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, et présente tout ce qu'il faut pour avoir du CRUD, c'est-à-dire tout ce qu'il faut pour ajouter, lister, modifier ou supprimer des informations.

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.

Elle se base sur plusieurs couches que l'on a déjà (ou on va bientôt) aborder (suivant le sens de lecture que vous préférez):

1. Le modèle et les validateurs
2. Les formulaires
3. Les widgets

14.1. 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 pour chaque 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:

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

1. Les attributs `Meta`:

```
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.

14.2. admin.ModelAdmin

La classe `admin.ModelAdmin` que l'on retrouvera principalement dans le fichier `admin.py` de chaque application contiendra la définition de ce que l'on souhaite faire avec nos données dans l'administration. Cette classe (et sa partie Meta)

14.3. L'affichage

Comme l'interface d'administration fonctionne (en trèèèès) gros comme un CRUD auto-généré, on trouve par défaut la possibilité de :

1. Créer de nouveaux éléments
2. Lister les éléments existants
3. Modifier des éléments existants

4. Supprimer un élément en particulier.

Les affichages sont donc de deux types: en liste et par élément.

Pour les affichages en liste, le plus simple consiste à jouer sur la propriété `list_display`.

Par défaut, la première colonne va accueillir le lien vers le formulaire d'édition. On peut donc modifier ceci, voire créer de nouveaux liens vers d'autres éléments en construisant des URLs dynamiquement.

(Insérer ici l'exemple de Medplan pour les liens vers les postgradués :-))

Voir aussi comment personnaliser le fil d'Ariane ?

14.4. Les filtres

1. list_filter
2. filter_horizontal
3. filter_vertical
4. date_hierarchy

14.5. Les permissions

On l'a dit plus haut, il vaut mieux éviter de proposer un accès à l'administration à vos utilisateurs. Il est cependant possible de configurer des permissions spécifiques pour certains groupes, en leur autorisant certaines actions de visualisation/ajout/édition ou suppression.

Cela se joue au niveau du `ModelAdmin`, en implémentant les méthodes suivantes:

```
def has_add_permission(self, request):
    return True

def has_delete_permission(self, request):
    return True

def has_change_permission(self, request):
    return True
```

On peut accéder aux informations de l'utilisateur actuellement connecté au travers de l'objet `request.user`.

a. NOTE: ajouter un ou deux screenshots :-)

14.6. Les relations

14.6.1. Les relations 1-n

Les relations 1-n sont implémentées au travers de formsets (que l'on a normalement déjà décrits plus haut). L'administration permet de les définir d'une manière extrêmement simple, grâce à quelques propriétés.

L'implémentation consiste tout d'abord à définir le comportement du type d'objet référencé (la relation -N), puis à inclure cette définition au niveau du type d'objet référençant (la relation 1-).

```
class WishInline(TabularInline):
    model = Wish


class Wishlist(admin.ModelAdmin):
    ...
    inlines = [WishInline]
    ...
```

Et voilà : l'administration d'une liste de souhaits (*Wishlist*) pourra directement gérer des relations multiples vers des souhaits.

14.6.2. Les auto-suggestions et auto-complétions

Parler de l'intégration de select2.

14.7. La présentation

Parler ici des `fieldsets` et montrer comment on peut regrouper des champs dans des groupes, ajouter un peu de javascript, ...

14.8. Les actions sur des sélections

Les actions permettent de partir d'une liste d'éléments, et autorisent un utilisateur à appliquer une action sur une sélection d'éléments. Par défaut, il existe déjà une action de **suppression**.

Les paramètres d'entrée sont :

1. L'instance de classe
2. La requête entrante
3. Le queryset correspondant à la sélection.

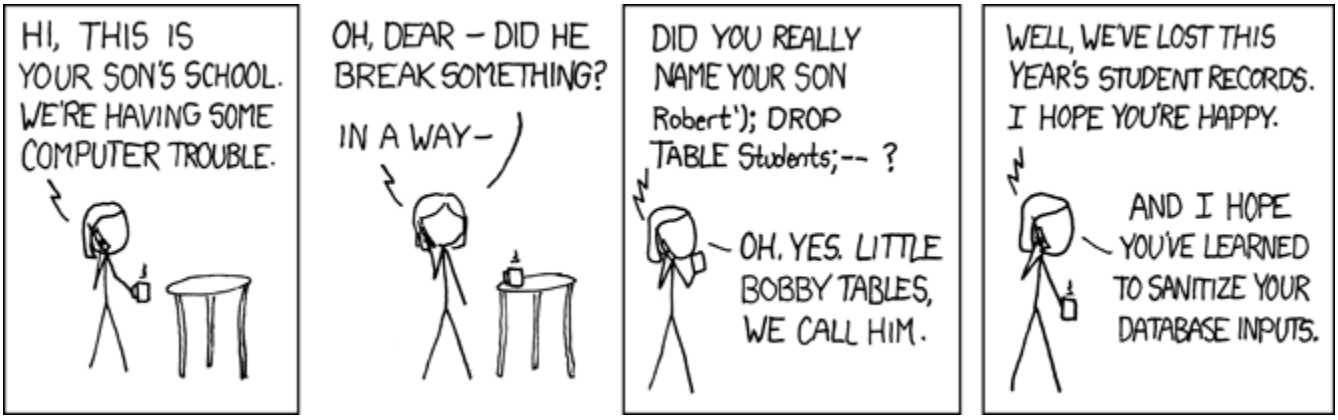
```
def double_quantity(self, request, queryset):
    for obj in queryset.all():
        obj.field += 1
        obj.save()
double_quantity.short_description = "Doubler la quantité des souhaits."
```

Et pour informer l'utilisateur de ce qui a été réalisé, on peut aussi lui passer un petit message:

```
if rows_updated = 0:
    self.message_user(request, "Aucun élément n'a été impacté.")
else:
    self.message_user(request, "{} élément(s) mis à jour".format(rows_updated))
```

Chapitre 15. Forms

Ou comment valider proprement des données entrantes.



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. On peut proposer trois versions d'un même code, de la version simple (lecture du fichier csv et jonglage avec les indices de colonnes), puis une version plus sophistiquée (et plus lisible, à base de DictReader), et la version + à base de form.

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 deux rôles importants:

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

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

15.1. Flux de validation

| .Validation | .is_valid | .clean_fields □ .clean_fields_machin

A compléter ;-)

15.2. 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.

```
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.

15.3. 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.

```
from datetime import date

from django import forms

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, ...)'
                            })
        }
```

15.4. 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 et en passant par `widget-tweaks`.

15.5. 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. 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

15.6. En conclusion

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

Chapitre 16. 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.

16.1. 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.

```
# 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 })
```

Rien qu'ici, on doit déjà tester deux choses:

1. Qu'on construit bien le modèle attendu - la liste de tous les souhaits déjà émis.
2. Que le template `wish/list.html` existe bien - sans quoi, on va tomber sur une erreur de type `TemplateDoesNotExist` dans notre environnement de test, et sur une erreur 500 en production.

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>
```

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! =)

16.2. 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`.

Voir Classy Class Based Views.

L'idée derrière les classes est de définir des fonctions **par convention plutôt que par configuration**.

à compléter ici :-)

16.2.1. 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 une sous-classe de celle que l'on souhaite utiliser
2. Câbler l'URL qui lui sera associée
3. Définir le squelette.

```
# 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'
```

Il est même possible de réduire encore ce morceau de code en définissant juste le snippet suivant :

```
# wish/views.py

from django.views.generic import ListView

from .models import Wishlist

class WishListList(ListView):
    context_object_name = 'wishlists'
```

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).

En l'état, par rapport à notre précédente vue basée sur une fonction, on y gagne sur les conventions utilisées et le nombre de tests à réaliser. A vous de voir la déclaration que vous préférez, en fonction de vos affinités et du résultat que vous souhaitez atteindre.

un petit tableau de différence entre les deux ? :-)

```
# 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.

Chapitre 17. Templates

Avant de commencer à interagir avec nos données au travers de listes, formulaires et d'interfaces 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 selon 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.

(je ne sais plus ce que je voulais dire ici)

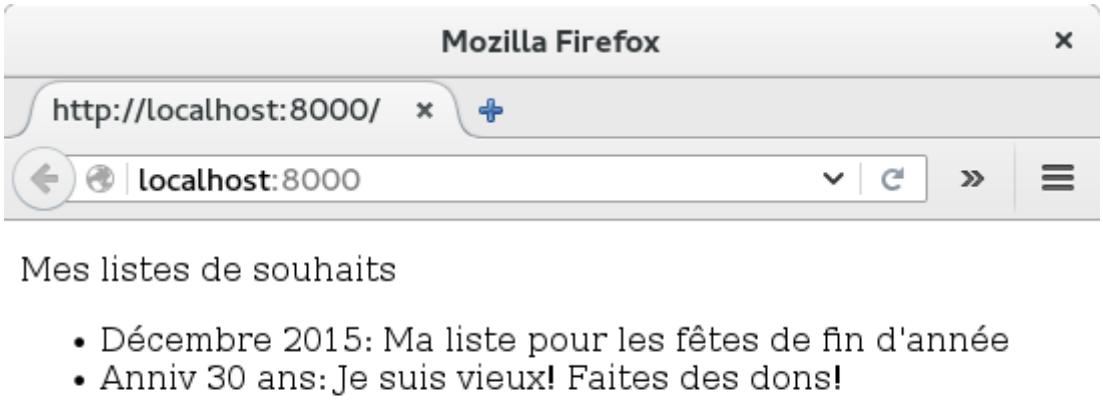
Un squelette de page HTML basique ressemble à ceci:

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



Mais plutôt que de réécrire à chaque fois le même entête, on peut se simplifier la vie en implémentant un héritage au niveau des templates. Pour cela, il suffit de définir des blocs de contenu, et d'**étendre** une page de base, puis de surcharger ces mêmes blocs.

Par exemple, si on repart de notre page de base ci-dessus, on va y définir deux blocs réutilisables:

```
<!-- templates/base.html -->

<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <meta http-equiv="x-ua-compatible" content="ie=edge">
        <title>{% block title %}Gwift{% endblock %}</title> ①
    </head>
    <body>
        {% block body %}<p>Hello world!</p>{% endblock %} ②
    </body>
</html>
```

① Un bloc `title`

② Un bloc `body`

La page HTML pour nos listes de souhaits devient alors:

```
<!-- templates/wishlist/wishlist_list.html -->

{% extends "base.html" %} ①

{% block title %}{{ block.super }} - Listes de souhaits{% endblock %} ②

{% block body %} ③
<p>Mes listes de souhaits</p>
<ul>
{% for wishlist in wishlists %}
    <li>{{ wishlist.name }}: {{ wishlist.description }}</li>
{% endfor %}
</ul>
```

① On étend/hérite de notre page `base.html`

② On redéfinit le titre (mais on réutilise le titre initial en appelant `block.super`)

③ On définit uniquement le contenu, qui sera placé dans le bloc `body`.

17.1. 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' ],
        ...
    },
]
```

17.2. Builtins

Django vient avec un ensemble de **tags** ou **template 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 à partir de son nom
- `urlize` qui permet de remplacer des URLs trouvées dans un champ de type CharField ou TextField par un lien cliquable.
- ...

Chacune de ces fonctions peut être utilisée autant au niveau des templates qu'au niveau du code. Il suffit d'aller les chercher dans le package `django.template.defaultfilters`. Par exemple:

```
from django.db import models
from django.template.defaultfilters import urlize


class Suggestion(models.Model):
    """Représentation des suggestions.
    """
    subject = models.TextField(verbose_name="Sujet")

    def urlized_subject(self):
        """
        Voir https://docs.djangoproject.com/fr/3.0/howto/custom-template-tags/
        """
        return urlize(self.subject, autoescape=True)
```

17.3. 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. Type `{% include '…' … %}`.

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]
```

Pour plus d'informations, la documentation officielle est un bon début.

17.3.1. Filtres

```
# 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
```

17.3.2. Tags simples

```
# wish/tools.py

from django import template

from wish.models import Wishlist


register = template.Library()


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

17.3.3. Tags d'inclusion

```
# wish/tools.py

from django import template

from wish.models import Wishlist


register = template.Library()


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

17.4. Contexts Processors

Un `context processor` permet d'ajouter des informations par défaut à un contexte (le dictionnaire qu'on passe de la vue au template). L'idée est d'ajouter une fonction à un module Python à notre projet, puis de le référencer parmi les CONTEXT_PROCESSORS de nos paramètres généraux. Cette fonction doit peupler un dictionnaire, et les clés de ce dictionnaire seront directement ajoutées à tout autre dictionnaire/contexte passé à une vue. Par exemple:

(cf. StackOverflow - à retravailler)

```
from product.models import SubCategory, Category


def add_variable_to_context(request):
    return {
        'subCategories': SubCategory.objects.order_by('id').all(),
        'categories': Category.objects.order_by("id").all(),
    }
```

```
'OPTIONS': {
    'context_processors': [
        ....
        'core.context_processors.add_variable_to_context',
        ....
    ],
},
```

17.5. 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>
</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.

17.5.1. 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

Chapitre 18. URLs et espaces de noms

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)),
]
```

La variable `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`.

petit mot d'explication sur les expressions rationnelles.

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

Pour reprendre l'exemple où on en était resté:

```
# 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'),
]
```

A présent, on doit tester que l'URL racine de notre application mène bien vers la fonction `wish_views.wishlists`.

Prenons par exemple l'exemple de Twitter : quand on accède à une URL, elle est de la forme `https://twitter.com/<user>`. Sauf que les pages `about` et `help` existent également. Pour implémenter ce type de précédence, il faudrait implémenter les URLs de la manière suivante:

```
| about
| help
| <user>
```

Mais cela signifie aussi que les utilisateurs `about` et `help` (s'ils existent...) ne pourront jamais accéder à leur profil. Une dernière solution serait de maintenir une liste d'authorité des noms d'utilisateur qu'il n'est pas possible d'utiliser.

D'où l'importance de bien définir la séquence de déinition de ces routes, ainsi que des espaces de noms.

L'idée des espaces de noms ou *namespaces* est de définir un *sous-répertoire* dans lequel on trouvera nos nouvelles routes. Cette manière de procéder permet notamment de répondre au problème ci-dessous, en définissant un sous-dossier type `https://twitter.com/users/<user>`.

De là, découle une autre bonne pratique: l'utilisation de *breadcrumbs* (https://stackoverflow.com/questions/826889/how-to-implement-breadcrumbs-in-a-django-template) ou de guidelines de navigation.

18.1. 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')
```

Chapitre 19. Authentification

Comme on l'a vu dans la partie sur le modèle, nous souhaitons que le créateur d'une liste puisse retrouver facilement les éléments qu'il aura créé. Ce dont nous n'avons pas parlé cependant, c'est la manière dont l'utilisateur va pouvoir créer son compte et s'authentifier. La documentation est très complète, nous allons essayer de la simplifier au maximum. Accrochez-vous, le sujet peut être complexe.

19.1. Mécanisme d'authentification

On peut schématiser le flux d'authentification de la manière suivante :

En gros:

1. La personne accède à une URL qui est protégée (voir les décorateurs @login_required et le mixin LoginRequiredMixin)
2. Le framework détecte qu'il est nécessaire pour la personne de se connecter (grâce à un paramètre type LOGIN_URL)
3. Le framework présente une page de connexion ou un mécanisme d'accès pour la personne (template à définir)
4. Le framework récupère les informations du formulaire, et les transmets aux différents backends d'authentification, dans l'ordre
5. Chaque backend va appliquer la méthode `authenticate` en cascade, jusqu'à ce qu'un backend réponde True ou qu'aucun ne réponde
6. La réponse de la méthode authenticate doit être une instance d'un utilisateur, tel que définit parmi les paramètres généraux de l'application.

En résumé (bis):

1. Une personne souhaite se connecter;
2. Les backends d'authentification s'enchaîne jusqu'à trouver une bonne correspondance. Si aucune correspondance n'est trouvée, on envoie la personne sur les roses.
3. Si OK, on retourne une instance de type current_user, qui pourra être utilisée de manière uniforme dans l'application.

Ci-dessous, on définit deux backends différents pour mieux comprendre les différentes possibilités:

1. Une authentification par jeton
2. Une authentification LDAP

```
from datetime import datetime

from django.contrib.auth import backends, get_user_model
from django.db.models import Q

from accounts.models import Token  ①


UserModel = get_user_model()


class TokenBackend(backends.ModelBackend):
    def authenticate(self, request, username=None, password=None, **kwargs):
        """Authentifie l'utilisateur sur base d'un jeton qu'il a reçu.

        On regarde la date de validité de chaque jeton avant d'autoriser l'accès.
        """
        token = kwargs.get("token", None)

        current_token = Token.objects.filter(token=token, validity_date__gte=datetime
.now()).first()

        if current_token:
            user = current_token.user

            current_token.last_used_date = datetime.now()
            current_token.save()

            return user

        return None
```

① Sous-entend qu'on a bien une classe qui permet d'accéder à ces jetons ;-)

```
from django.contrib.auth import backends, get_user_model

from ldap3 import Server, Connection, ALL
from ldap3.core.exceptions import LDAPPasswordIsMandatoryError

from config import settings


UserModel = get_user_model()


class LdapBackend(backends.ModelBackend):
    """Implémentation du backend LDAP pour la connexion des utilisateurs à l'Active
Directory.
    """
    def authenticate(self, request, username=None, password=None, **kwargs):
        """Authentifie l'utilisateur au travers du serveur LDAP.
        """

        ldap_server = Server(settings.LDAP_SERVER, get_info=ALL)
        ldap_connection = Connection(ldap_server, user=username, password=password)

        try:
            if not ldap_connection.bind():
                raise ValueError("Login ou mot de passe incorrect")
        except (LDAPPasswordIsMandatoryError, ValueError) as ldap_exception:
            raise ldap_exception

        user, _ = UserModel.objects.get_or_create(username=username)
```

On peut résumer le mécanisme d'authentification de la manière suivante:

- Si vous voulez modifier les informations liées à un utilisateur, orientez-vous vers la modification du modèle. Comme nous le verrons ci-dessous, il existe trois manières de prendre ces modifications en compte. Voir également ici.
- Si vous souhaitez modifier la manière dont l'utilisateur se connecte, alors vous devrez modifier le **backend**.

19.2. Modification du modèle

Dans un premier temps, Django a besoin de manipuler des instances de type `django.contrib.auth.User`. Cette classe implémente les champs suivants:

- `username`
- `first_name`
- `last_name`
- `email`

- `password`
- `date_joined`.

D'autres champs, comme les groupes auxquels l'utilisateur est associé, ses permissions, savoir s'il est un super-utilisateur, ... sont moins pertinents pour le moment. Avec les quelques champs déjà définis ci-dessus, nous avons de quoi identifier correctement nos utilisateurs. Inutile d'implémenter nos propres classes, puisqu'elles existent déjà :-)

Si vous souhaitez ajouter un champ, il existe trois manières de faire.

19.3. Extension du modèle existant

Le plus simple consiste à créer une nouvelle classe, et à faire un lien de type `OneToOne` vers la classe `django.contrib.auth.User`. De cette manière, on ne modifie rien à la manière dont Django authentife ses utlisateurs: tout ce qu'on fait, c'est un lien vers une table nouvellement créée, comme on l'a déjà vu au point [...voir l'héritage de modèle]. L'avantage de cette méthode, c'est qu'elle est extrêmement flexible, et qu'on garde les mécanismes Django standard. Le désavantage, c'est que pour avoir toutes les informations de notre utilisateur, on sera obligé d'effectuer une jointure sur le base de données, ce qui pourrait avoir des conséquences sur les performances.

19.4. Substitution

Avant de commencer, sachez que cette étape doit être effectuée **avant la première migration**. Le plus simple sera de définir une nouvelle classe héritant de `django.contrib.auth.User` et de spécifier la classe à utiliser dans votre fichier de paramètres. Si ce paramètre est modifié après que la première migration ait été effectuée, il ne sera pas pris en compte. Tenez-en compte au moment de modéliser votre application.

```
AUTH_USER_MODEL = 'myapp.MyUser'
```

Notez bien qu'il ne faut pas spécifier le package `.models` dans cette injection de dépendances: le schéma à indiquer est bien `<nom de l□application>.<nom de la classe>`.

19.4.1. Backend

19.4.2. Templates

Ce qui n'existe pas par contre, ce sont les vues. Django propose donc tout le mécanisme de gestion des utilisateurs, excepté le visuel (hors administration). En premier lieu, ces paramètres sont fixés dans le fichier `settings <https://docs.djangoproject.com/en/1.8/ref/settings/#auth>`_. On y trouve par exemple les paramètres suivants:

- `LOGIN_REDIRECT_URL`: si vous ne spécifiez pas le paramètre `next`, l'utilisateur sera automatiquement redirigé vers cette page.
- `LOGIN_URL`: l'URL de connexion à utiliser. Par défaut, l'utilisateur doit se rendre sur la page `/accounts/login`.

19.4.3. Social-Authentification

Voir ici : python social auth

19.4.4. Un petit mot sur OAuth

OAuth est un standard libre définissant un ensemble de méthodes à implémenter pour l'accès (l'autorisation) à une API. Son fonctionnement se base sur un système de jetons (Tokens), attribués par le possesseur de la ressource à laquelle un utilisateur souhaite accéder.

Le client initie la connexion en demandant un jeton au serveur. Ce jeton est ensuite utilisée tout au long de la connexion, pour accéder aux différentes ressources offertes par ce serveur. `wikipedia <http://en.wikipedia.org/wiki/OAuth>`_.

Une introduction à OAuth est disponible ici. Elle introduit le protocole comme étant une `valet key`, une clé que l'on donne à la personne qui va garer votre voiture pendant que vous profitez des mondanités. Cette clé donne un accès à votre voiture, tout en bloquant un ensemble de fonctionnalités. Le principe du protocole est semblable en ce sens: vous vous réservez un accès total à une API, tandis que le système de jetons permet d'identifier une personne, tout en lui donnant un accès restreint à votre application.

L'utilisation de jetons permet notamment de définir une durée d'utilisation et une portée d'utilisation. L'utilisateur d'un service A peut par exemple autoriser un service B à accéder à des ressources qu'il possède, sans pour autant révéler son nom d'utilisateur ou son mot de passe.

L'exemple repris au niveau du workflow est le suivant : un utilisateur(trice), Jane, a uploadé des photos sur le site faji.com (A). Elle souhaite les imprimer au travers du site beppa.com (B). Au moment de la commande, le site beppa.com envoie une demande au site faji.com pour accéder aux ressources partagées par Jane. Pour cela, une nouvelle page s'ouvre pour l'utilisateur, et lui demande d'introduire sa "pièce d'identité". Le site A, ayant reçu une demande de B, mais certifiée par l'utilisateur, ouvre alors les ressources et lui permet d'y accéder.

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,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d}
{message}',
            'style': '{',
        },
        'simple': {
            'format': '{levelname} {asctime} {module} {message}',
            'style': '{',
        },
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': "simple"
        }
    },
    'loggers': {
        'connexys': {
            'handlers': ['console'],
            'level': 'DEBUG',
            'propagate': True,
        },
        'stp': {
            'handlers': ['console'],
            '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).

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

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 21. 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 22. Besoins fonctionnels

22.1. 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.

22.2. Gestion des listes

22.2.1. 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

22.2.2. 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

22.3. Gestion des souhaits

22.3.1. 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.

22.3.2. 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.

22.4. Gestion des réalisations de souhaits

22.4.1. 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

22.4.2. 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.

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

22.5.1. Modélisation

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

- un identifiant
- un nom
- une adresse email facultative

22.5.2. 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 23. Tests unitaires

23.1. 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.

23.2. 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.

23.3. What are you testing?

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

23.4. 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.

23.5. 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:

```
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.

```
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.

23.6. Quelques liens utiles

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

Unresolved directive in main.adoc - include::gwift/key-points.adoc[]

Chapitre 24. 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.

24.1. 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.

```
# 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;
```

24.2. 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.

24.3. 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:

```
# 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`.

Unresolved directive in main.adoc - include::gwift/console.adoc[]