# Minor swing with Django

**Cédric Declerfayt, Fred Pauchet**

# Table of Contents

# Licence

# Préface

> The only way to go fast, is to go well
>
> — Robert C. Martin

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, sans réellement préparer à la maintenance et au suivi de l'application nouvellement développée.

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 menant jusqu'au déploiement et de s'assurer du maintient correct de la base de code, en permettant à n'importe qui de reprendre ce qui aura déjà été écrit.

Ces idées ne s'appliquent pas uniquement à Django et à son cadre de travail, ni même au langage Python. Ces deux sujets sont cependant de bons candidats et leur cadre de travail est bien défini, documenté et suffisamment flexible.

Django se présente comme un *Framework Web pour perfectionnistes ayant des deadlines* [1] et suit ces quelques principes [2]:

- Faible couplage et forte cohésion, pour que chaque composant dispose de son indépendance, en n'ayant aucune connaissance des autres couches applicatives. Ainsi, le moteur de rendu ne connait absolument rien l'existence du moteur de base de données, tout comme le système de vues ne sait pas quel moteur de rendu est utilisé.
- Plus de fonctionnalités avec moins de code: chaque application Django doit utiliser le moins de code possible
- *Don't repeat yourself*, chaque concept ou morceau de code ne doit être présent qu'à un et un seul endroit de vos dépôts.
- Rapidité du développement, en masquant les aspects fastidieux du développement web actuel

Mis côte à côte, le suivi de ces principes permet une bonne stabilité du projet à moyen et long

terme.

Comme nous le verrons par la suite, et sans être parfait, Django offre une énorme flexibilité qui permet de se laisser le maximum d'options ouvertes tout en permettant d'expérimenter facilement plusieurs pistes, jusqu'au moment de prendre une vraie décision. Dans la majorité des cas problématiques pouvant être rencontrés lors du développement d'une application Web, Django proposera une solution pragmatique, compréhensible et facile à mettre en place. En résumé de ce paragraphe, pour tout problème commun, vous disposerez d'une solution logique. 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. Ces quelques points pourront être appliqués pour n'importe quel langage ou cadre de travail. 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, bien qu'encore un peu inutile.

**Dans la deuxième 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 troisième 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 quatrième partie**, nous aborderons les architectures typées *entreprise*, les services et les différentes manières de structurer notre application pour faciliter sa gestion et sa maintenance, tout en décrivant différents types de scénarii, en fonction des consommateurs de données.

**Dans la cinquième partie**, nous mettrons ces concepts en pratique en présentant le développement en pratique de deux applications, avec la description de problèmes

rencontrés et la solution qui a été choisie: définition des tables, gestion des utilisateurs, ... et mise à disposition.

## Pour qui ?

Ce livre s'adresse autant au néophyte qui souhaite se lancer dans le développement Web qu'à l'artisan qui a besoin d'un aide-mémoire et qui ne se rappelle plus toujours du bon ordre des paramètres, ou à l'expert qui souhaiterait avoir un aperçu d'une autre technologie que son domaine privilégié de compétences.

Beaucoup de concepts présentés peuvent être oubliés ou restés inconnus jusqu'au moment où ils seront réellement nécessaires. A ce moment-là, pour peu que votre mémoire ait déjà entraperçu le terme, il vous sera plus facile d'y revenir et de l'appliquer.

## Pour aller plus loin

Il existe énormément de ressources, autant spécifiques à Django que plus généralistes. Il ne sera pas possible de toutes les détailler; faites un tour sur

- https://duckduckgo.com,
- https://stackoverflow.com,
- https://ycombinator.com,
- https://lobste.rs/,
- https://lecourrierduhacker.com/
- ou https://www.djangoproject.com/.

Restez curieux, ne vous enclavez pas dans une technologie en particulier et gardez une bonne ouverture d'esprit.

## Conventions

Les notes indiquent des anecdotes.

Les conseils indiquent des éléments utiles, mais pas spécialement indispensables.

Les notes importantes indiquent des éléments à retenir.

Ces éléments indiquent des points d'attention. Les retenir vous fera gagner du temps en débuggage.

Les avertissements indiquent un (potentiel) danger ou des éléments pouvant amener des conséquences pas spécialement sympathiques.

# Let’s keep in touch

# Environnement et méthodes de travail

> Make it work, make it right, make it fast
>
> — Kent Beck

En fonction de vos connaissances et compétences, la création d'une nouvelle application est uneé tape relativement facile à mettre en place. Le code qui permet de faire tourner cette application peut ne pas être élégant, voire buggé jusqu'à la moëlle, il pourra fonctionner et faire "preuve de concept".

Les problèmes arriveront lorsqu'une nouvelle demande sera introduite, lorsqu'un bug sera découvert et devra être corrigé ou lorsqu'une dépendance cessera de fonctionner ou d'être disponible. Or, une application qui n'évolue pas, meurt. Tout application est donc destinée, soit à être modifiée, corrigée et suivie, soit à déperrir et à être délaissée par ses utilisateurs. Et c'est juste cette maintenance qui est difficile.

L'application des principes présentés et agrégés ci-dessous permet surtout de préparer correctement tout ce qui pourra arriver, sans aller jusqu'au « **You Ain't Gonna Need It** » (ou **YAGNI**), qui consiste à surcharger tout développement avec des fonctionnalités non demandées, juste « au cas ou ». Pour paraphraser une partie de l'introduction du livre *Clean Architecture* [8]:

> Getting software right is hard: it takes knowledge and skills that most young programmers don't take the time to develop. It requires a level of discipline and dedication that most programmers never dreamed they'd need. Mostly, it takes a passion for the craft and the desire to be a professional.

Le développement d'un logiciel nécessite une rigueur d'exécution et des connaissances précises dans des domaines extrêmement variés. Il nécessite également des intentions, des (bonnes) décisions et énormément d'attention. Indépendamment de l'architecture que vous aurez choisie, des technologies que vous aurez patiemment évaluées et mises en place, une architecture et une solution peuvent être cassées en un instant, en même temps que tout ce que vous aurez construit, dès que vous en aurez détourné le regard.

Un des objectifs ici est de placer les barrières et les gardes-fous (ou plutôt, les "**garde-vous**"), afin de péréniser au maximum les acquis, stabiliser les bases de tous les environnements (du développement à la production) qui pourraient accueillir notre application et fiabiliser les étapes de communication.

Dans cette partie, nous allons parler de **méthodes 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, et chaque étape doit être rendue la plus automatisée/automatisable possible. Dans son plus simple élément, une application pourrait être mise à jour simplement en envoyant son code sur un dépôt centralisé: ce déclencheur doit démarrer une chaîne de vérification d'utilisabilité/fonctionnalités/débuggabilité/sécurité, pour immédiatement la mettre à disposition de nouveaux utilisateurs si toute la chaîne indique que tout est OK.

Dans une version plus manuelle, cela pourrait se résumer à ces trois étapes (la dernière étant formellement facultative):

1. Démarrer un script,
2. Prévoir un rollback si cela plante (et si cela a planté, préparer un post-mortem de l'incident pour qu'il ne se produise plus)
3. Se préparer une tisane en regardant nos flux RSS (pour peu que 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. Architecture

> A software system that is hard to develop is not likely to have a long and healthy lifetime
>
> — Robert C. Martin, Clean Architecture

> If you think good architecture is expensive, try bad architecture
>
> — Brian Foote and Joseph Yoder

> A computer program is a detailed description of the policy by which inputs are transformed into outputs.
>
> — Robert C. Martin, Clean Architecture

Au delà des principes SOLID dont il est question plus haut, c'est à nouveau dans les ressources proposées et les cas démontrés que l'on comprend leur intérêt: plus que de la définition d'une architecture adéquate, c'est surtout dans la facilité de maintenance d'une application que ces principes s'identifient.

Derrière une bonne architecture, il y a aussi un investissement quant aux ressources qui seront nécessaires à faire évoluer l'application: ne pas investir dès qu'on le peut va juste lentement remplir la case de la dette technique.

Une bonne architecture va également rendre le système facile à lire, facile à développer, facile à maintenir et facile à déployer. L'objectif ultime étant de minimiser le coût de maintenance et de maximiser la productivité des développeurs. Un des autres objectifs d'une bonne architecture consiste également à se garder le plus d'options possibles, et à se concentrer sur les détails (le type de base de données, la conception concrète, ...), le plus tard possible, tout en conservant la politique principale en ligne de mire. Cela permet de délayer les choix techniques à « plus tard », ce qui permet également de concrétiser ces choix en ayant le plus d'informations possibles [8 pp. 137-141]

Une architecture ouverte et pouvant être étendue n'a d'intérêt que si le développement est suivi et que les gestionnaires (et architectes) s'engagent à économiser du temps et de la qualité lorsque des changements seront demandés pour l'évolution du projet.

## 1.1. Politiques et règles métiers

TODO: Un p'tit bout à ajouter sur les méthodes de conception ;)

## 1.2. Considération sur les frameworks

> Frameworks are tools to be used, not architectures to be conformed to. Your architecture should tell readers about the system, not about the frameworks you used in your system. If you are building a health care system, then when new programmers look at the source repository, their first impression should be, « oh, this is a health care system ». Those new programmers should be able to learn all the use cases of the system, yet still not know how the system is delivered.
>
> — Robert C. Martin, Clean Architecture

Le point soulevé ci-dessous est qu'un framework n'est qu'un outil, et pas une obligation de structuration. L'idée est que le framework doit se conformer à la définition de l'application, et non l'inverse. Dans le cadre de l'utilisation de Django, c'est un point critique à prendre en considération: une fois que vous aurez fait ce choix, vous aurez extrêmement difficile à faire machine arrière:

- Votre modèle métier sera largement couplé avec le type de base de données (relationnelle, indépendamment
- Votre couche de présentation sera surtout disponible au travers d'un navigateur
- Les droits d'accès et permissions seront en grosse partie gérés par le frameworks
- La sécurité dépendra de votre habilité à suivre les versions
- Et les fonctionnalités complémentaires (que vous n'aurez pas voulu/eu le temps de développer) dépendront de la bonne volonté de la communauté

Le point à comprendre ici n'est pas que "Django, c'est mal", mais qu'une fois que vous aurez défini la politique, les règles métiers, les données critiques et entités, et que vous aurez fait le choix de développer en âme et conscience votre nouvelle création en utilisant Django, vous serez bon gré mal gré, contraint de continuer avec. Cette décision ne sera pas irrévocable, mais difficile à contourner.

> At some point in their history most DevOps organizations were hobbled by tightly-coupled, monolithic architectures that while extremely successfull at helping them achieve product/market fit - put them at risk of organizational failure once they had to operate at scale (e.g. eBay's monolithic C++ application in 2001, Amazon's monolithic OBIDOS application in 2001, Twitter's monolithic Rails front-end in 2009, and LinkedIn's monolithic Leo application in 2011). In each of these cases, they were able to re-architect their systems and set the stage not only to survice, but also to thrise and win in the marketplace

[6 p. 182]

Ceci dit, Django compense ses contraintes en proposant énormément de flexibilité et de fonctionnalités **out-of-the-box**, c'est-à-dire que vous pourrez sans doute avancer vite et bien jusqu'à un point de rupture, puis revoir la conception et réinvestir à ce moment-là, mais en toute connaissance de cause.

> When any of the external parts of the system become obsolete, such as the database, or the web framework, you can replace those obsolete elements with a minimum of fuss.
>
> — Robert C. Martin, Clean Architecture

Avec Django, la difficulté à se passer du framework va consister à basculer vers « autre chose » et a remplacer chacune des tentacules qui aura pousser partout dans l'application.

A noter que les services et les « architectures orientées services » ne sont jamais qu'une définition d'implémentation des frontières, dans la mesure où un service n'est jamais qu'une fonction appelée au travers d'un protocole (rest, soap, ...). Une application monolotihique sera tout aussi fonctionnelle qu'une application découpée en microservices. (Services: great and small, page 243).

## 1.3. Un point sur l'inversion de dépendances

Dans la partie SOLID, nous avons évoqué plusieurs principes de développement. Django est un framework qui évolue, et qui a pu présenter certains problèmes liés à l'un de ces principes.

Les link:release notes[https://docs.djangoproject.com/en/2.0/releases/2.0/] de Django 2.0 date de décembre 2017; parmi ces notes, l'une d'elles cite l'abandon du support d'link:Oracle

11.2[https://docs.djangoproject.com/en/2.0/releases/2.0/#dropped-support-for-oracle-11-2]. En substance, cela signifie que le framework se chargeait lui-même de construire certaines parties de requêtes, qui deviennent non fonctionnelles dès lors que l'on met le framework ou le moteur de base de données à jour. Réécrit, cela signifie que:

1. Si vos données sont stockées dans un moteur géré par Oracle 11.2, vous serez limité à une version 1.11 de Django
2. Tandis que si votre moteur est géré par une version ultérieure, le framework pourra être mis à jour.

Nous sommes dans un cas concret d'inversion de dépendances ratée: le framework (et encore moins vos politiques et règles métiers) ne devraient pas avoir connaissance du moteur de base de données. Pire, vos politiques et données métiers ne devraient pas avoir connaissance **de la version** du moteur de base de données.

En conclusion, le choix d'une version d'un moteur technique (**la base de données**) a une incidence directe sur les fonctionnalités mises à disposition par votre application, ce qui va à l'encontre des 12 facteurs (et des principes de développement).

Ce point sera rediscuté par la suite, notamment au niveau de l'épinglage des versions, de la reproduction des environnements et de l'interdépendance entre des choix techniques et fonctionnels.

## 1.4. Evolutions

## 1.5. 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

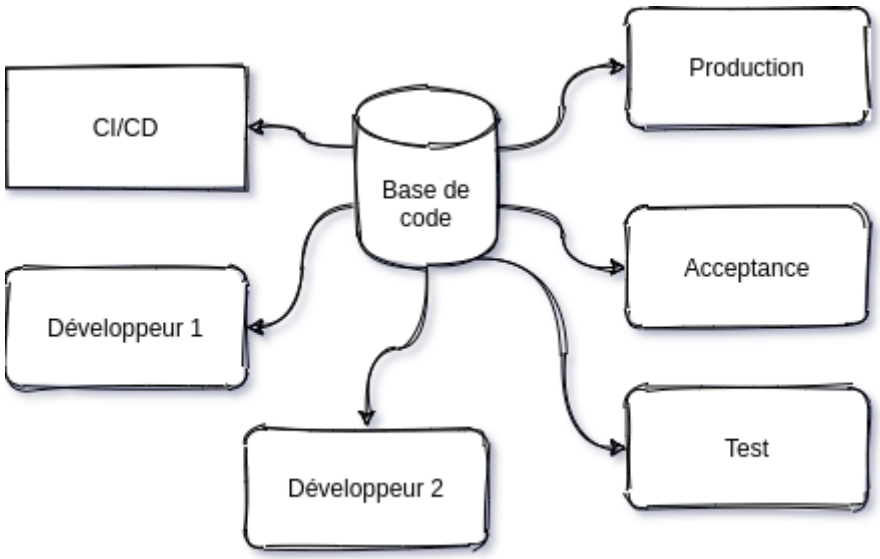
architecturale et une meilleure mise à l'échelle - *Vous avez 5000 utilisateurs en plus? Ajoutez un serveur et on n'en parle plus ;-).*

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.




Comme l'explique Eran Messeri [1], ingénieur dans le groupe Google Developer Infrastructure, un des avantages d'utiliser un dépôt unique de sources, est qu'il permet un accès facile et rapide à la forme la plus à jour du code, sans aucun besoin de coordination. Ce dépôt ne sert pas seulement au code source, mais également à d'autres artefacts et formes de connaissance:

- Standards de configuration (Chef recipes, Puppet manifests, ...)
- Outils de déploiement
- Standards de tests, y compris tout ce qui touche à la sécurité
- Outils de déploiement de pipeline
- Outils d'analyse et de monitoring
- Tutoriaux

**#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**

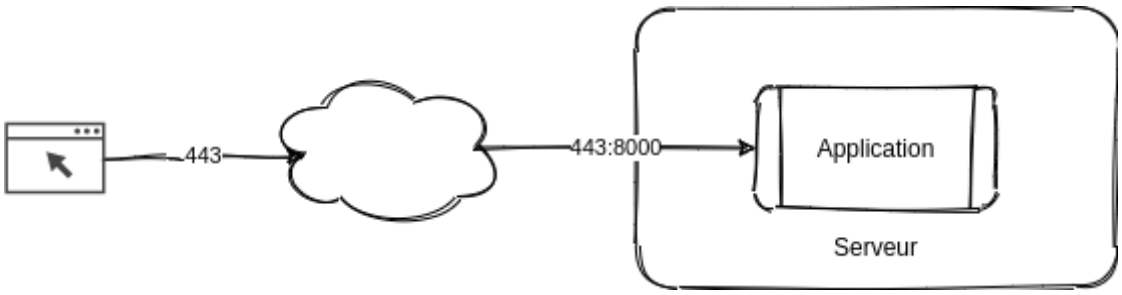
Toute information stockée en mémoire ou sur disque ne doit pas altérer le comportement futur de l'application, par exemple après un redémarrage non souhaité.

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.

## 1.6. Design for operations through codified non-functional requirements

Pour paraphraser une section du DevOps Handbook (Part V, Chapitre 20, Convert Local Discoveries into Global Improvements (page 293-294), une application devient nettement plus maintenable dès lors que l'équipe de développement suit de près les différentes étapes de sa conception, de la demande jusqu'à son aboutissement en production. Au fur et à mesure que le code est délibérément construit pour être maintenable, nous gagnons en rapidité, en qualité et en fiabilité de déploiement et les tâches liées aux opérations en sont facilitées. Ces prérequis sont les suivants:

- Activation d'une télémétrie suffisante dans les applications et les environnements.
- Conservation précise des dépendances nécessaires
- Résilience des services et plantage élégant (i.e. **sans finir sur un SEGFAULT avec l'OS dans les choux et un écran bleu**)
- Compatibilité entre les différentes versions (n+1, …)
- Gestion de l'espace de stockage associé à un environnement (pour éviter d'avoir un

environnement de production qui fait 157 Tera-octets)

- Activation de la recherche dans les logs
- Traces des requêtes provenant des utilisateurs, indépendamment des services utilisés
- Centralisation de la configuration (**via** ZooKeeper, par exemple)

## 1.7. Maintenabilité

## 1.8. Maintenance

> The primary cost of maintenance is in spelunking and risk [8 p. 139]
>
> — Robert C. Martin

En ayant connaissance de toutes les choses qui pourraient être modifiées par la suite, l'idée est de pousser le développement jusqu'au point où un service pourrait être nécessaire. A ce stade, l'architecture nécessitera des modifications, mais aura déjà intégré le fait que cette possibilité existe. Nous n'allons donc pas jusqu'au point où le service doit être créé (même s'il peut ne pas être nécessaire), ni à l'extrême au fait d'ignorer qu'un service pourrait être nécessaire, mais nous aboutissons à une forme de compromis. Une forme de comportement de Descartes, qui ne croit pas en Dieu, mais qui envisage quand même cette possibilité, ce qui lui ouvre le maximum de portes

Avec cette approche, les composants sont déjà découplés au niveau du code source, ce qui pourrait s'avérer suffisant jusqu'au stade où une modification ne pourra plus faire reculer l'échéance.

En terme de découpe, les composants peuvent l'être aux niveaux suivants:

@ code source @ déploiement, au travers de dll, jar, linked libraries, ... voire au travers de threads ou de processus locaux. @ services

Cette section se base sur deux ressources principales [11], qui 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.

### 1.8.1. 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.

### 1.8.2. 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.

### 1.8.3. 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.

### 1.8.4. 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**. === 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**.

## 1.9. Robustesse et flexibilité

> Un code mal pensé entraîne nécessairement une perte d'énergie et de temps. Il est plus simple de réfléchir, au moment de la conception du programme, à une architecture permettant une meilleure maintenabilité que de devoir corriger un code "sale" *a posteriori*. C'est pour aider les développeurs à rester dans le droit chemin que les principes SOLID ont été énumérés. GNU/Linux Magazine HS 104

[3 pp. 26-44]

Les principes SOLID, introduit par Robert C. Martin dans les années 2000 sont les suivants:

1. SRP - Single responsibility principle - Principe de Responsabilité Unique
2. OCP - Open-closed principle
3. LSP - Liskov Substitution
4. ISP - Interface ségrégation principle
5. DIP - Dependency Inversion Principle

En plus de ces principes de développement, il faut ajouter des principes au niveau des composants, puis un niveau plus haut, au niveau, au niveau architectural :

1. Reuse/release équivalence principle,
2. Common Closure Principle,
3. Common Reuse Principle.

### 1.9.1. Single Responsibility Principle

Le principe de responsabilité unique conseille de disposer de concepts ou domaines d'activité qui ne s'occupent chacun que d'une et une seule chose. Ceci rejoint un peu la Philosophie Unix, documentée par Doug McIlroy et qui demande de "*faire une seule chose, mais le faire bien*" [9]. Une classe ou un élément de programmtion ne doit donc pas avoir plus d'une raison de changer.

Il est également possible d'étendre ce principe en fonction d'acteurs:

> A module should be responsible to one and only one actor.

[8]

Plutôt que de centraliser le maximum de code à un seul endroit ou dans une seule classe par convenance ou commodité [3], le principe de responsabilité unique suggère que chaque classe soit responsable d'un et un seul concept. Une autre manière de voir les choses consiste à différencier les acteurs ou les intervenants: imaginez avoir une classe représentant des données de membres du personnel. Ces données pourraient être demandées par trois acteurs, le CFO, le CTO et le COO: ceux-ci ont tous besoin de données et d'informations relatives à une même base de données centralisées, mais ont chacun besoin d'une représentation différente ou de traitements distincts. [8] Nous sommes d'accord qu'il s'agit à chaque fois de données liées aux employés, mais elles vont un cran plus loin et pourraient nécessiter des ajustements spécifiques en fonction de l'acteur concerné. L'idée sous-jacente est simplement d'identifier dès que possible les différents acteurs, en vue de prévoir une modification qui pourrait être demandée par l'un d'entre eux. Dans le cas d'un

élément de code centralisé, une modification induite par un des acteurs pourrait ainsi avoir un impact sur les données utilisées par les autres.

```
class Document:
    def __init__(self, title, content, published_at):
        self.title = title
        self.content = content
        self.published_at = published_at

    def render(self, format_type):
        if format_type == "XML":
            return """<?xml version = "1.0"?>
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                self.title,
                self.content,
                self.published_at.isoformat()
            )

        if format_type == "Markdown":
            import markdown
            return markdown.markdown(self.content)

        raise ValueError("Format type '{}' is not known".format(format_type))
```

Lorsque nous devrons ajouter un nouveau rendu (Atom, OpenXML, ...), nous devrons modifier la classe `Document`, ce qui n'est pas vraiment intuitif. Une bonne pratique consisterait à créer une classe de rendu par type de format à gérer:

```
class Document:
    def __init__(self, title, content, published_at):
        self.title = title
        self.content = content
        self.published_at = published_at

class DocumentRenderer:
    def render(self, document):
        if format_type == "XML":
            return """<?xml version = "1.0"?>
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                self.title,
                self.content,
                self.published_at.isoformat()
            )

        if format_type == "Markdown":
            import markdown
            return markdown.markdown(self.content)

        raise ValueError("Format type '{}' is not known".format(format_type))
```

A présent, lorsque nous devrons ajouter un nouveau format de prise en charge, nous irons modifier la classe `DocumentRenderer`, sans que la classe `Document` ne soit impactée. En même temps, le jour où une instance de type `Document` sera liée à un champ `author`, rien ne dit que le rendu devra en tenir compte; nous modifierons donc notre classe pour y ajouter le nouveau champ sans que cela n'impacte nos différentes manières d'effectuer un rendu.

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 de structurer le code permet de centraliser la configuration d'un type d'évènement à un seul endroit, ce qui augmente ainsi la testabilité globale du projet.

Lorsque nous verrons les composants, le SRP deviendra le CCP. Au niveau architectural, ce sera la définition des frontières (boundaries).

### 1.9.2. Open Closed

> For software systems to be easy to change, they must be designed to allow the behavior to change by adding new code instead of changing existing code.

L'objectif est de rendre le système facile à étendre, en évitant que l'impact d'une modification ne soit trop grand.

Les exemples parlent d'eux-mêmes: des données doivent être présentées dans une page web. Et demain, ce seras dans un document PDF. Et après demain, ce sera dans un tableur Excel. La source de ces données restent la même (au travers d'une couche de présentation), mais la mise en forme diffère à chaque fois.

L'application n'a pas à connaître les détails d'implémentation: elle doit juste permettre une forme d'extension, sans avoir à appliquer une modification (ou une grosse modification) sur son cœur.

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.

Nous pouvons également appliquer ceci à notre exemple sur les rendus de document, où le code suivant

```
class DocumentRenderer:
    def render(self, document):
        if format_type == "XML":
            return """<?xml version = "1.0"?>
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                document.title,
                document.content,
                document.published_at.isoformat()
            )

        if format_type == "Markdown":
            import markdown
            return markdown.markdown(document.content)

        raise ValueError("Format type '{}' is not known".format(format_type))
```

devient le suivant:

```
class Renderer:
    def render(self, document):
        raise NotImplementedError

class XmlRenderer(Renderer):
    def render(self, document)
        return """<?xml version = "1.0"?>
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                document.title,
                document.content,
                document.published_at.isoformat()
            )

class MarkdownRenderer(Renderer):
    def render(self, document):
        import markdown
        return markdown.markdown(document.content)
```

Lorsque nous ajouterons notre nouveau type de rendu, nous ajouterons simplement une nouvelle classe de rendu qui héritera de `Renderer`.

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

### 1.9.3. Liskov Substitution

> Dans Clean Architecture, ce chapitre ci (le 9) est sans doute celui qui est le moins complet. Je suis d'accord avec les exemples donnés, dans la mesure où la définition concrète d'une classe doit dépendre d'une interface correctement définie (et que donc, faire hériter un carré d'un rectangle, n'est pas adéquat dans le mesure où cela induit l'utilisateur en erreur), mais il y est aussi question de la définition d'un style architectural pour une interface REST, mais sans donner de solution...

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`*", nous allons nous retrouver avec ceci:

```
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!")
```

Le principe de substitution de Liskov suggère qu'une classe doit toujours pouvoir être considérée comme une instance de sa classe parent, et **doit pouvoir s'y substituer**. Dans notre exemple, cela signifie que nous pourrons tout à fait accepter qu'un lion se comporte comme un canard et adore manger des plantes, insectes, graines, algues et du poisson. Miam ! Nous 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).

Pour revenir à nos exemples de rendus de documents, nous aurions pu faire hériter notre `MarkdownRenderer` de la classe `XmlRenderer`:

```
class XmlRenderer:
    def render(self, document)
        return """<?xml version = "1.0"?>
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                document.title,
                document.content,
                document.published_at.isoformat()
            )

class MarkdownRenderer(XmlRenderer):
    def render(self, document):
        import markdown
        return markdown.markdown(document.content)
```

Mais lorsque nous ajouterons une fonction d'entête, notre rendu en Markdown héritera irrémédiablement de cette même méthode:

```
class XmlRenderer:
    def header(self):
        return """<?xml version = "1.0"?>"""

    def render(self, document)
        return """{}
            <document>
                <title>{}</title>
                <content>{}</content>
                <publication_date>{}</publication_date>
            </document>""".format(
                self.header(),
                document.title,
                document.content,
                document.published_at.isoformat()
            )

class MarkdownRenderer(XmlRenderer):
    def render(self, document):
        import markdown
        return markdown.markdown(document.content)
```

A nouveau, lorsque nous invoquerons la méthode `header()` sur une instance de type `MarkdownRenderer`, nous obtiendrons un bloc de déclaration XML (`<?xml version = "1.0"?>`) pour un fichier Markdown.

### 1.9.4. Interface Segregation Principle

Le principe de ségrégation d'interface suggère de limiter la nécessité de recompiler un module, en n'exposant que les opérations nécessaires à l'exécution d'une classe. Ceci évite d'avoir à redéployer l'ensemble d'une application.

> The lesson here is that depending on something that carries baggage that you don't need can cause you troubles that you didn't except.

Ce principe stipule qu'un client ne doit pas 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.

GNU/Linux Magazine [3 pp. 37-42] propose un exemple d'interface permettant d'implémenter une imprimante:

```
interface IPrinter
{
    public abstract void printPage();

    public abstract void scanPage();

    public abstract void faxPage();
}

public class Printer
{
    protected string name;

    public Printer(string name)
    {
        this.name = name;
    }
}
```

L'implémentation d'une imprimante multifonction aura tout son sens:

```
public class AllInOnePrinter implements Printer extends IPrinter
{
    public AllInOnePrinter(string name)
    {
        super(name);
    }

    public void printPage()
    {
        System.out.println(this.name + ": Impression");
    }

    public void scanPage()
    {
        System.out.println(this.name + ": Scan");
    }

    public void faxPage()
    {
        System.out.println(this.name + ": Fax");
    }
}
```

Tandis que l'implémentation d'une imprimante premier-prix ne servira pas à grand chose:

```
public class FirstPricePrinter implements Printer extends IPrinter
{
    public FirstPricePrinter(string name)
    {
        super(name);
    }

    public void printPage()
    {
        System.out.println(this.name + ": Impression");
    }

    public void scanPage()
    {
        System.out.println(this.name + ": Fonctionnalité absente");
    }

    public void faxPage()
    {
        System.out.println(this.name + ": Fonctionnalité absente");
    }
}
```

L'objectif est donc de découpler ces différentes fonctionnalités en plusieurs interfaces bien spécifiques, implémentant chacune une opération isolée:

```
interface IPrinterPrinter
{
    public abstract void printPage();
}

interface IPrinterScanner
{
    public abstract void scanPage();
}

interface IPrinterFax
{
    public abstract void faxPage();
}
```

Cette réflexion s'applique finalement à n'importe quel composant: votre système

d'exploitation, les librairies et dépendances tierces, les variables déclarées, ... Quel que soit le composant que l'on utilise ou analyse, il est plus qu'intéressant de se limiter uniquement à ce dont nous avons besoin plutôt que

En Python, ce comportement est inféré lors de l'exécution, et donc pas vraiment d'application pour notre contexte d'étude: de manière plus générale, les langages dynamiques sont plus flexibles et moins couplés que les langages statiquement typés, pour lesquels l'application de ce principe-ci permettrait de mettre à jour une DLL ou un JAR sans que cela n'ait d'impact sur le reste de l'application.

Il est ainsi possible de trouver quelques horreurs, dans tous les langages:

```
/*!
 * is-odd <https://github.com/jonschlinkert/is-odd>
 *
 * Copyright (c) 2015-2017, Jon Schlinkert.
 * Released under the MIT License.
 */

'use strict';

const isNumber = require('is-number');

module.exports = function isOdd(value) {
  const n = Math.abs(value);
  if (!isNumber(n)) {
    throw new TypeError('expected a number');
  }
  if (!Number.isInteger(n)) {
    throw new Error('expected an integer');
  }
  if (!Number.isSafeInteger(n)) {
    throw new Error('value exceeds maximum safe integer');
  }
  return (n % 2) === 1;
};
```

Voire, son opposé, qui dépend évidemment du premier:

```
/*!
 * is-even <https://github.com/jonschlinkert/is-even>
 *
 * Copyright (c) 2015, 2017, Jon Schlinkert.
 * Released under the MIT License.
 */

'use strict';

var isOdd = require('is-odd');

module.exports = function isEven(i) {
  return !isOdd(i);
};
```

Il ne s'agit que d'un simple exemple, mais qui tend à une seule chose: gardez les choses simples (et, éventuellement, stupides) . Dans l'exemple ci-dessus, l'utilisation du module `is-odd` requière déjà deux dépendances: `is-even` et `is-number`. Imaginez la suite.

### 1.9.5. Dependency inversion Principle

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. L'objectif est que les interfaces soient les plus stables possibles, afin de réduire au maximum les modifications qui pourraient y être appliquées. De cette manière, toute modification fonctionnelle pourra être directement appliquée sur le composant de bas-niveau, sans que l'interface ne soit impactée.

> The dependency inversion principle tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.

[8]

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

Django est bourré de ce principe, que ce soit pour les *middlewares* ou pour les connexions aux bases de données. Lorsque nous écrivons ceci dans notre fichier de configuration,

```
# [snip]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

# [snip]
```

Django ira simplement récupérer chacun de ces middlewares, qui répondent chacun à une interface clairement définie, dans l'ordre. Il n'y a donc pas de magie; c'est le développeur qui va simplement brancher ou câbler des fonctionnalités au niveau du framework, en les déclarant au bon endroit. Pour créer un nouveau *middleware*, il suffira d'implémenter le code suivant et de l'ajouter dans la configuration de l'application:

```
def simple_middleware(get_response):
    # One-time configuration and initialization.

    def middleware(request):
        # Code to be executed for each request before
        # the view (and later middleware) are called.

        response = get_response(request)

        # Code to be executed for each request/response after
        # the view is called.

        return response

    return middleware
```

Dans d'autres projets écrits en Python, ce type de mécanisme peut être implémenté relativement facilement en utilisant les modules importlib et la fonction `getattr`.

Un autre exemple concerne les bases de données: pour garder un maximum de flexibilité, Django ajoute une couche d'abstraction en permettant de spécifier le moteur de base de données que vous souhaiteriez utiliser, qu'il s'agisse d'SQLite, MSSQL, Oracle, PostgreSQL ou MySQL/MariaDB [4].

> The database is really nothing more than a big bucket of bits where we store our data on a long term basis.

[8 p. 281]

D'un point de vue architectural, nous ne devons pas nous soucier de la manière dont les données sont stockées, s'il s'agit d'un disque magnétique, de ram, ... en fait, on ne devrait même pas savoir s'il y a un disque du tout. Et Django le fait très bien pour nous.

En termes architecturaux, ce principe autorise une définition des frontières, et en permettant une séparation claire en inversant le flux de dépendances et en faisant en sorte que les règles métiers n'aient aucune connaissance des interfaces graphiques qui les exploitent ou des moteurs de bases de données qui les stockent. Ceci autorise une forme d'immunité entre les composants.

### 1.9.6. Sources

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

## 1.10. au niveau des composants

De la même manière que pour les principes définis ci-dessus, Mais toujours en faisant attention qu'une fois que les frontières sont implémentés, elles sont coûteuses à maintenir. Cependant, il ne s'agit pas une décision à réaliser une seule fois, puisque cela peut être réévalué.

Et de la même manière que nous devons délayer au maximum les choix architecturaux et techniques,

> but this is not a one time decision. You don't simply decide at the start of a project which boundaries to implémentent and which to ignore. Rather, you watch. You pay attention as the system evolves. You note where boundaries may be required, and then carefully watch for the first inkling of friction because those boundaries don't exist. at that point, you weight the costs of implementing those boundaries versus the cost of ignoring them and you review that decision frequently. Your goal is to implement the boundaries right at the inflection point where the cost of implementing becomes less than the cost of ignoring.

En gros, il faut projeter sur la capacité à s'adapter en minimisant la maintenance. Le problème est qu'elle ne permettait aucune adaptation, et qu'à la première demande, l'architecture se plante complètement sans aucune malléabilité.

### 1.10.1. Reuse/release equivalence principle

```
Classes and modules that are grouped together into a component should be
releasable together
-- (Chapitre 13, Component Cohesion, page 105)
```

### 1.10.2. CCP

(= l'équivalent du SRP, mais pour les composants)

> If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same component

Il y a peut-être aussi un lien à faire avec « Your code as a crime scene » 🔪

```
La définition exacte devient celle-ci: « gather together those things that
change at the same times and for the same reasons. Separate those things that
change at different times or for different reasons ».
```

```
==== CRP
```

```
 ⋯ que l'on résumera ainsi: « don't depend on things you don't need » 
Au niveau des composants, au niveau architectural, mais également à d'autres
niveaux.
```

### 1.10.3. SDP

(Stable dependency principle) qui définit une formule de stabilité pour les composants, en fonction de sa faculté à être modifié et des composants qui dépendent de lui: au plus un composant est nécessaire, au plus il sera stable (dans la mesure où il lui sera difficile de changer). En C++, cela correspond aux mots clés #include. Pour faciliter cette stabilité, il convient de passer par des interfaces (donc, rarement modifiées, par définition).

En Python, ce ratio pourrait être calculé au travers des import, via les AST.

### 1.10.4. SAP

(= Stable abstraction principle) pour la définition des politiques de haut niveau vs les composants plus concrets. SAP est juste une modélisation du OCP pour les composants: nous plaçons ceux qui ne changent pas ou pratiquement pas le plus haut possible dans l'organigramme (ou le diagramme), et ceux qui changent souvent plus bas, dans le sens de stabilité du flux. Les composants les plus bas sont considérés comme volatiles

## 1.11. Intégrées

> Tests are part of the system. You can think of tests as the outermost circle in the architecture. Nothing within in the system depends on the tests, and the tests always depend inward on the components of the system ».
>
> — Robert C. Martin, Clean Architecture

[1] The DevOps Handbook, Part V, Chapitre 20, Convert Local Discoveries into Global Improvements

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

[3] Aussi appelé *God-Like object*

[4] http://howfuckedismydatabase.com/

# Chapitre 2. Le langage Python

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

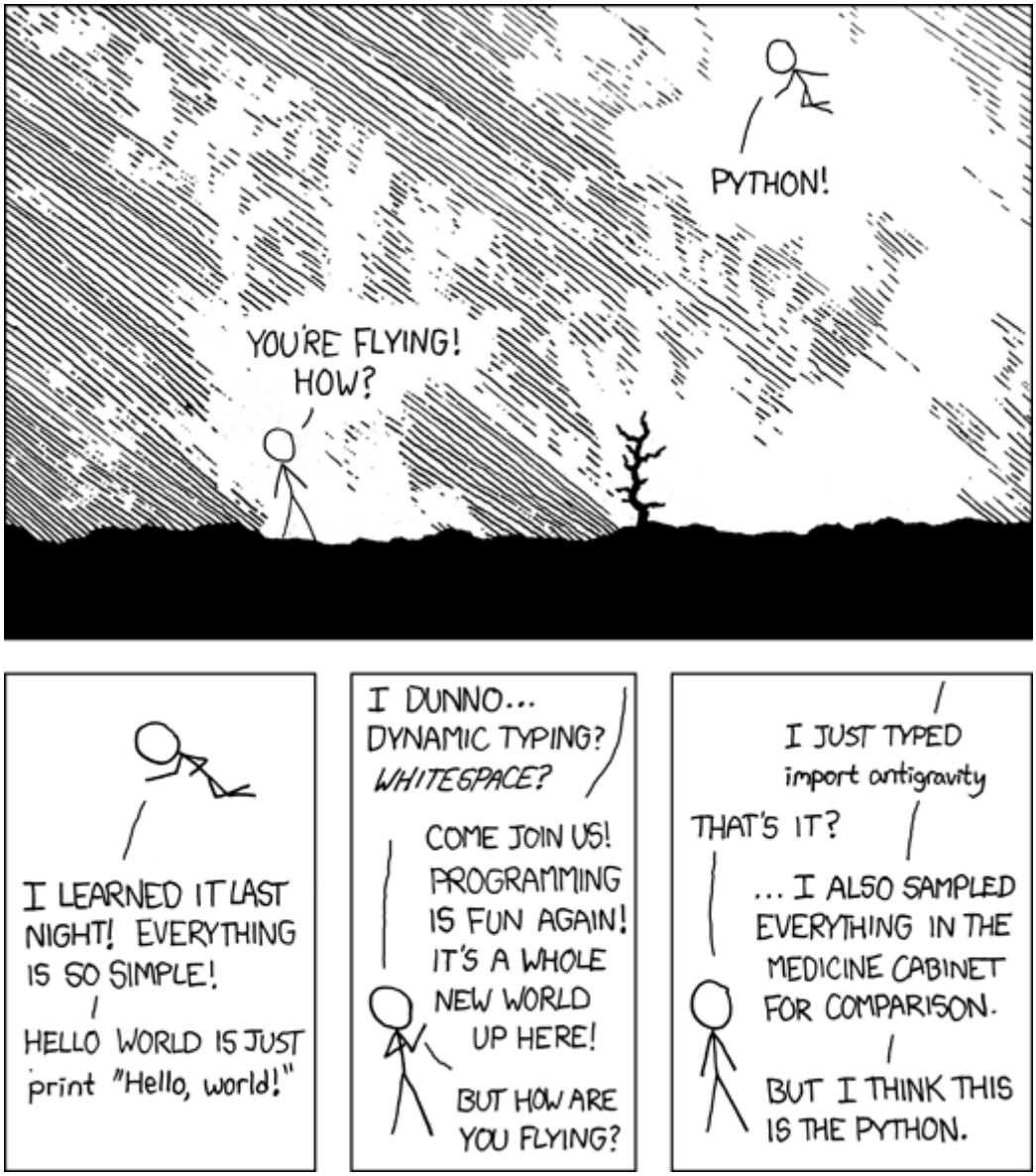



*Figure 1. https://xkcd.com/353/*

A première vue, et suivants les autres langages que vous connaitriez ou auriez déjà abordé,

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.

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.

Les morceaux de code que vous trouverez ci-dessous seront développés pour Python3.9+ et Django 3.2+. Ils nécessiteront peut-être quelques adaptations pour fonctionner sur une version antérieure.

# 2.1. Eléments de langage

En fonction de votre niveau d'apprentissage du langage, plusieurs ressources pourraient vous aider:

- **Pour les débutants**, Automate the Boring Stuff with Python [10], aka. *Practical Programming for Total Beginners*
- **Pour un (gros) niveau au dessus** et pour un état de l'art du langage, nous ne pouvons que vous recommander le livre Expert Python Programming [5], qui aborde énormément d'aspects du langage en détails (mais pas toujours en profondeur): les différents types d'interpréteurs, les éléments de langage avancés, différents outils de productivité, métaprogrammation, optimisation de code, programmation orientée évènements, multithreading et concurrence, tests, ... A ce jour, c'est le concentré de sujets liés au langage le plus intéressant qui ait pu arriver entre nos mains.

En parallèle, 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.

## 2.1.1. Protocoles de langage

Le modèle de données du langage spécifie un ensemble de méthodes qui peuvent être surchargées. Ces méthodes suivent une convention de nommage et leur nom est toujours encadré par un double tiret souligné; d'où leur nom de "*dunder methods*" ou "*double-*

*underscore methods*". La méthode la plus couramment utilisée est la méthode `init()`, qui permet de surcharger l'initialisation d'une instance de classe.

```
class CustomUserClass:
    def __init__(self, initiatization_argument):
        ...
```

[5 pp. 142-144]

Ces méthodes, utilisées seules ou selon des combinaisons spécifiques, constituent les *protocoles de langage*. Une liste complètement des *dunder methods* peut être trouvée dans la section `Data Model` de la documentation du langage Python.

All operators are also exposed as ordinary functions in the operators module. The documentation of that module gives a good overview of Python operators. It can be found at https://docs.python.org/3.9/library/operator.html

If we say that an object implements a specific language protocol, it means that it is compatible with a specific part of the Python language syntax.

The following is a table of the most common protocols within the Python language.

Protocol nameMethodsDescriptionCallable protocol*call*()Allows objects to be called with parentheses:instance()Descriptor protocols*set*(), *get*(), and *del*()Allows us to manipulate the attribute access pattern of classes (see the Descriptors section)Container protocol *contains*()Allows us to test whether or not an object contains some value using the in keyword:value in instance

Python in Comparison with Other LanguagesIterable protocol*iter*()Allows objects to be iterated using the forkeyword:for value in instance: ...Sequence protocol*getitem*(),*len*()Allows objects to be indexed with square bracket syntax and queried for length using a built-in function:item = instance[index]length = len(instance)Each operator available in Python has its own protocol and operator overloading happens by implementing the dunder methods of that protocol. Python provides over 50 overloadable operators that can be divided into five main groups:• Arithmetic operators • In-place assignment operators• Comparison operators• Identity operators• Bitwise operatorsThat's a lot of protocols so we won't discuss all of them here. We will instead take a look at a practical example that will allow you to better understand how to implement operator overloading on your own

The `add()` method is responsible for overloading the `+` (plus sign) operator and here it allows us to add two matrices together. Only matrices of the same dimensions can be added together. This is a fairly simple operation that involves adding all matrix elements one by one to form a new matrix.

The `sub()` method is responsible for overloading the `-` (minus sign) operator that will be

responsible for matrix subtraction. To subtract two matrices, we use a similar technique as in the – operator:

```
def __sub__(self, other):
    if (len(self.rows) != len(other.rows) or len(self.rows[0]) != len(other
.rows[0])):
        raise ValueError("Matrix dimensions don't match")
    return Matrix([[a - b for a, b in zip(a_row, b_row)] for a_row, b_row in
zip(self.rows, other.rows)    ])
```

And the following is the last method we add to our class:

```
def __mul__(self, other):
    if not isinstance(other, Matrix):
        raise TypeError(f"Don't know how to multiply {type(other)} with
Matrix")

    if len(self.rows[0]) != len(other.rows):
        raise ValueError("Matrix dimensions don't match")

    rows = [[0 for _ in other.rows[0]]  for _ in self.rows]

    for i in range(len (self.rows)):
        for j in range(len (other.rows[0])):
            for k in range(len (other.rows)):
                rows[i][j] += self.rows[i][k] * other.rows[k][j]

    return Matrix(rows)
```

The last overloaded operator is the most complex one. This is the * operator, which is implemented through the `mul()` method. In linear algebra, matrices don't have the same multiplication operation as real numbers. Two matrices can be multiplied if the first matrix has a number of columns equal to the number of rows of the second matrix. The result of that operation is a new matrix where each element is a dot product of the corresponding row of the first matrix and the corresponding column of the second matrix. Here we've built our own implementation of the matrix to present the idea of operators overloading. Although Python lacks a built-in type for matrices, you don't need to build them from scratch. The NumPy package is one of the best Python mathematical packages and among others provides native support for matrix algebra. You can easily obtain the NumPy package from PyPI

En fait, l'intérêt concerne surtout la représentation de nos modèles, puisque chaque classe du modèle est représentée par la définition d'un objet Python. Nous pouvons donc utiliser ces mêmes **dunder methods** (**double-underscores methods**) pour étoffer les protocoles du

langage.

## 2.2. 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!
```

## 2.3. 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.).

## 2.4. 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, la compréhension et la reprise du code par une tierce personne. Cela implique aussi de **tout** documenter: les modules, les paquets, les classes, les fonctions, méthodes, ... Ce qui peut également aller à contrecourant d'autres pratiques [7 pp. 53-74]; il y a une juste mesure à prendre entre "tout documenter" et "tout bien documenter":

- Inutile d'ajouter des watermarks, auteurs, ... Git ou tout VCS s'en sortira très bien et sera beaucoup plus efficace que n'importe quelle chaîne de caractères que vous pourriez indiquer et qui sera fausse dans six mois,
- Inutile de décrire quelque chose qui est évident; documenter la méthode `get_age()` d'une personne n'aura pas beaucoup d'intérêt
- S'il est nécessaire de décrire un comportement au sein-même d'une fonction, c'est que ce comportement pourrait être extrait dans une nouvelle fonction (qui, elle, pourra être documentée)

Documentation: be obsessed! Mais **le code reste la référence**

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 (par exemple, clize ne reconnait que du RestructuredText; l'auto-documentation de Django également). L'exemple donné dans les guides de style de Google 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, si vous le jugez nécessaire
4. Une ligne vide
5. La description des arguments et paramètres, des valeurs de retour, des exemples et les exceptions qui peuvent être levées.

Un exemple (encore) plus complet peut être trouvé dans le dépôt sphinxcontrib-napoleon. Et ici, nous tombons peut-être dans l'excès de zèle:

```
def module_level_function(param1, param2=None, *args, **kwargs):
    """This is an example of a module level function.

    Function parameters should be documented in the ``Args`` section. The name
    of each parameter is required. The type and description of each parameter
    is optional, but should be included if not obvious.

    If \*args or \*\*kwargs are accepted,
    they should be listed as ``*args`` and ``**kwargs``.

    The format for a parameter is::

        name (type): description
            The description may span multiple lines. Following
            lines should be indented. The "(type)" is optional.

            Multiple paragraphs are supported in parameter
            descriptions.

    Args:
        param1 (int): The first parameter.
        param2 (:obj:`str`, optional): The second parameter. Defaults to None.
            Second line of description should be indented.
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.

    Returns:
        bool: True if successful, False otherwise.

        The return type is optional and may be specified at the beginning of
        the ``Returns`` section followed by a colon.

        The ``Returns`` section may span multiple lines and paragraphs.
        Following lines should be indented to match the first line.

        The ``Returns`` section supports any reStructuredText formatting,
```

```
        including literal blocks::

            {
                'param1': param1,
                'param2': param2
            }

    Raises:
        AttributeError: The ``Raises`` section is a list of all exceptions
            that are relevant to the interface.
        ValueError: If `param2` is equal to `param1`.

    """
    if param1 == param2:
        raise ValueError('param1 may not be equal to param2')
    return True
```

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. Toute information pertinente peut donc lier le code à un cas d'utilisation concret.

### 2.4.1. 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* [5] 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 [6] 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, nous trouvons 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.

TODO: Voir si la sortie de pylint est obligatoirement 0 s'il y a un warning

TODO: parler de `pylint --errors-only`

## 2.5. 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 clarté du code, facilité d'installation et d'intégration et 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*".

## 2.6. 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.

## 2.7. Typage statique - PEP585

Nous vous disions ci-dessus que Python était un langage dynamique interprété. Concrètement, cela signifie que des erreurs pouvant être détectées à la compilation avec d'autres langages, ne le sont pas avec Python.

Il existe cependant une solution à ce problème, sous la forme de Mypy, qui peut (sous vous le souhaitez ;-)) vérifier une forme de typage statique de votre code source, grâce à une expressivité du code, basée sur des annotations (facultatives, elles aussi).

Ces vérifications se présentent de la manière suivante:

```
from typing import List


def first_int_elem(l: List[int]) -> int:
    return l[0] if l else None


if __name__ == "__main__":
    print(first_int_elem([1, 2, 3]))
    print(first_int_elem(['a', 'b', 'c']))
```

Est-ce que le code ci-dessous fonctionne correctement ? **Oui**:

```
λ python mypy-test.py
1
a
```

Malgré que nos annotations déclarent une liste d'entiers, rien ne nous empêche de lui envoyer une liste de caractères, sans que cela ne lui pose de problèmes.

Est-ce que Mypy va râler ? **Oui, aussi**. Non seulement nous retournons la valeur `None` si la liste est vide alors que nous lui annoncions un entier en sortie, mais en plus, nous l'appelons avec une liste de caractères, alors que nous nous attendions à une liste d'entiers:

```
λ mypy mypy-test.py
mypy-test.py:7: error: Incompatible return value type (got "Optional[int]",
expected "int")
mypy-test.py:12: error: List item 0 has incompatible type "str"; expected
"int"
mypy-test.py:12: error: List item 1 has incompatible type "str"; expected
"int"
mypy-test.py:12: error: List item 2 has incompatible type "str"; expected
"int"
Found 4 errors in 1 file (checked 1 source file)
```

Pour corriger ceci, nous devons:

1. Importer le type `Optional` et l'utiliser en sortie de notre fonction `first_int_elem`
2. Eviter de lui donner de mauvais paramètres ;-)

```python
from typing import List, Optional


def first_int_elem(l: List[int]) -> Optional[int]:
    return l[0] if l else None


if __name__ == "__main__":
    print(first_int_elem([1, 2, 3]))
```

```
λ mypy mypy-test.py
Success: no issues found in 1 source file
```

### 2.7.1. 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`:

```python
from django.test import TestCase


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

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

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

Comme indiqué ci-dessus, Django propose son propre cadre de tests, au travers du package `django.tests`. Une bonne pratique (parfois discutée) consiste cependant à switcher vers `pytest`, qui présente quelques avantages:

- Une syntaxe plus concise (au prix de quelques conventions, même si elles restent configurables): un test est une fonction, et ne doit pas obligatoirement faire partie d'une classe héritant de `TestCase` - la seule nécessité étant que cette fonction fasse partie d'un module commençant ou finissant par "test" (`test_example.py` ou `example_test.py`).
- Une compatibilité avec du code Python "classique" - vous ne devrez donc retenir qu'un seul ensemble de commandes ;-)
- Des *fixtures* faciles à réutiliser entre vos différents composants
- Une compatibilité avec le reste de l'écosystème, dont la couverture de code présentée ci-dessous.

Ainsi, après installation, il nous suffit de créer notre module `test_models.py`, dans lequel nous allons simplement tester l'addition d'un nombre et d'une chaîne de caractères (oui, c'est complètement biesse; on est sur la partie théorique ici):

```
def test_add():
    assert 1 + 1 == "argh"
```

Forcément, cela va planter. Pour nous en assurer (dès fois que quelqu'un en doute), il nous suffit de démarrer la commande `pytest`:

```
λ pytest
============================= test session starts
====================================
platform ...
rootdir: ...
plugins: django-4.1.0
collected 1 item

gwift\test_models.py F
[100%]

================================== FAILURES
=========================================
_______________________________ test_basic_add
________________________________________

    def test_basic_add():
>       assert 1 + 1 == "argh"
E       AssertionError: assert (1 + 1) == 'argh'

gwift\test_models.py:2: AssertionError

============================ short test summary info
==================================
FAILED gwift/test_models.py::test_basic_add - AssertionError: assert (1 + 1)
== 'argh'
============================== 1 failed in 0.10s
====================================
```

### 2.7.2. 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. Le paquet `coverage` se charge d'évaluer le pourcentage de code couvert par les tests.

Avec `pytest`, il convient d'utiliser le paquet `pytest-cov`, suivi de la commande `pytest --cov=gwift tests/`.

Si vous préférez rester avec le cadre de tests de Django, vous pouvez passer par le paquet django-coverage-plugin 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
[...]
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
```

←-- / partie obsolète -->

Ceci vous affichera non seulement la couverture de code estimée, et générera également vos fichiers sources avec les branches non couvertes.

### 2.7.3. Matrice de compatibilité

L'intérêt de la matrice de compatibilité consiste à spécifier un ensemble de plusieurs versions d'un même interpréteur (ici, Python), afin de s'assurer que votre application continue à fonctionner. Nous sommes donc un cran plus haut que la spécification des versions des librairies, puisque nous nous situons directement au niveau de l'interpréteur.

L'outil le plus connu est Tox, qui consiste en un outil basé sur virtualenv et qui permet:

1. de vérifier que votre application s'installe correctement avec différentes versions de Python et d'interpréteurs
2. de démarrer des tests parmi ces différents environnements

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py36,py37,py38,py39
skipsdist = true

[testenv]
deps =
    -r requirements/dev.txt
commands =
    pytest
```

Démarrez ensuite la commande `tox`, pour démarrer la commande `pytest` sur les environnements Python 3.6, 3.7, 3.8 et 3.9, après avoir installé nos dépendances présentes dans le fichier `requirements/dev.txt`.

pour que la commande ci-dessus fonctionne correctement, il sera nécessaire que vous ayez les différentes versions d'interpréteurs installées. Ci-dessus, la commande retournera une erreur pour chaque version non trouvée, avec une erreur type `ERROR: pyXX: InterpreterNotFound: pythonX.X`.

### 2.7.4. Configuration globale

Décrire le fichier setup.cfg

```
$ touch setup.cfg
```

### 2.7.5. Dockerfile

```
# Dockerfile

# Pull base image
#FROM python:3.8
FROM python:3.8-slim-buster

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
ENV DEBIAN_FRONTEND noninteractive
ENV ACCEPT_EULA=Y

# install Microsoft SQL Server requirements.
ENV ACCEPT_EULA=Y
RUN apt-get update -y && apt-get update \
  && apt-get install -y --no-install-recommends curl gcc g++ gnupg


# Add SQL Server ODBC Driver 17
RUN curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
RUN curl https://packages.microsoft.com/config/debian/10/prod.list >
/etc/apt/sources.list.d/mssql-release.list
RUN apt-get update \
  && apt-get install -y msodbcsql17 unixodbc-dev

# clean the install.
RUN apt-get -y clean

# Set work directory
WORKDIR /code

# Install dependencies
COPY ./requirements/base.txt /code/requirements/
RUN pip install --upgrade pip
RUN pip install -r ./requirements/base.txt

# Copy project
COPY . /code/
```

### 2.7.6. 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."
```

Pour la petite histoire, `make` peu sembler un peu désuet, mais reste extrêmement efficace.

## 2.8. 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.

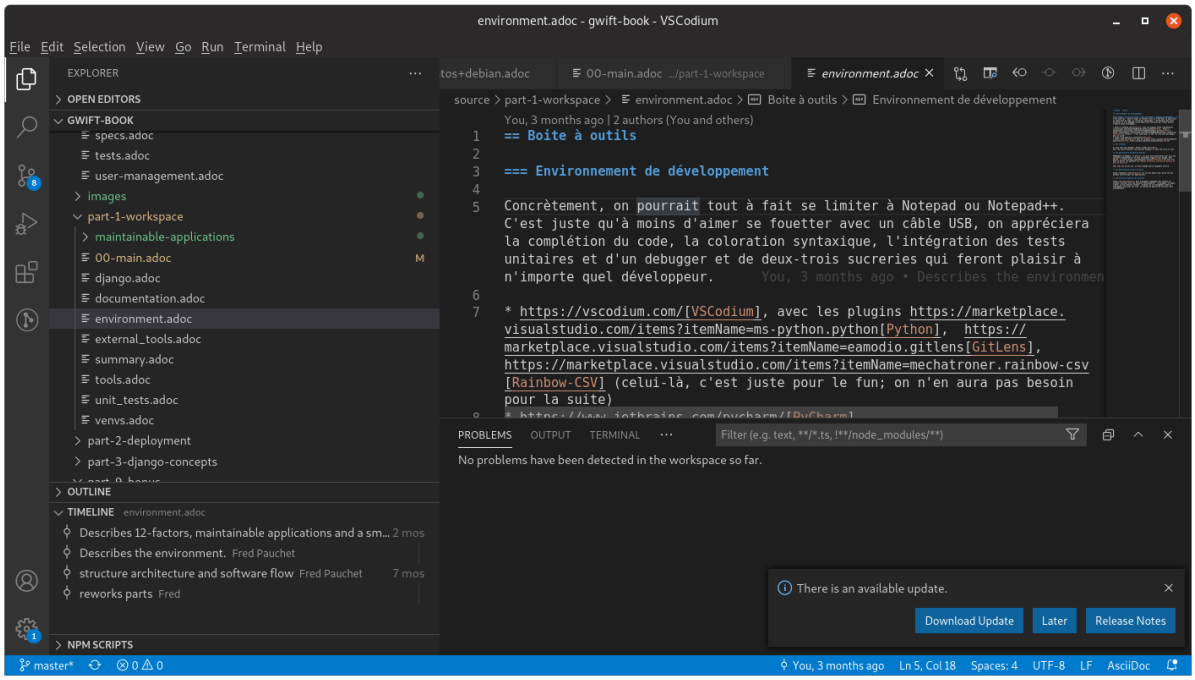



*Figure 3. Codium en action*

## 2.9. Un terminal

*A priori*, les IDE [7] 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.

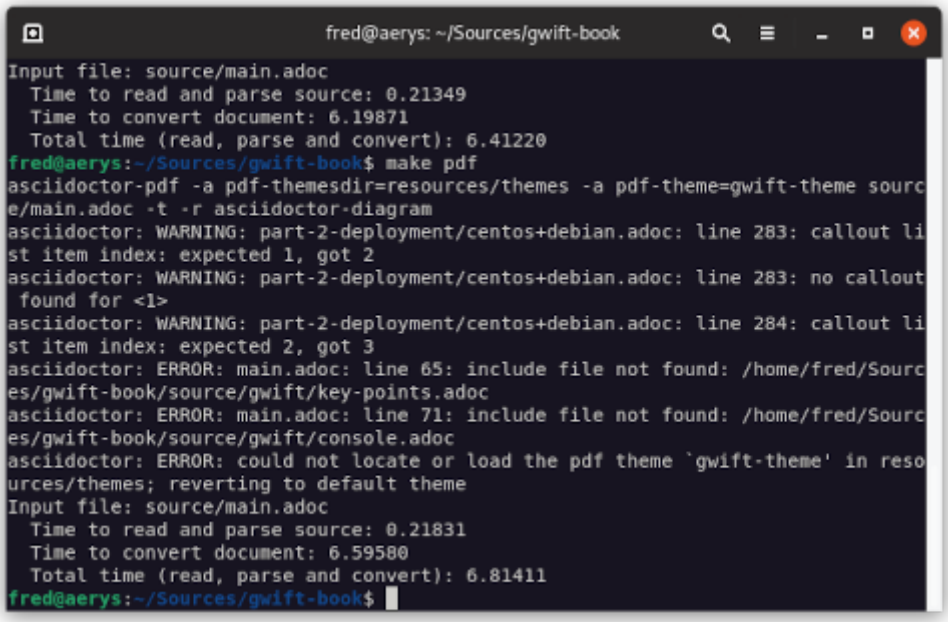



*Figure 4. Mise en abîme*

## 2.10. Un gestionnaire de base de données

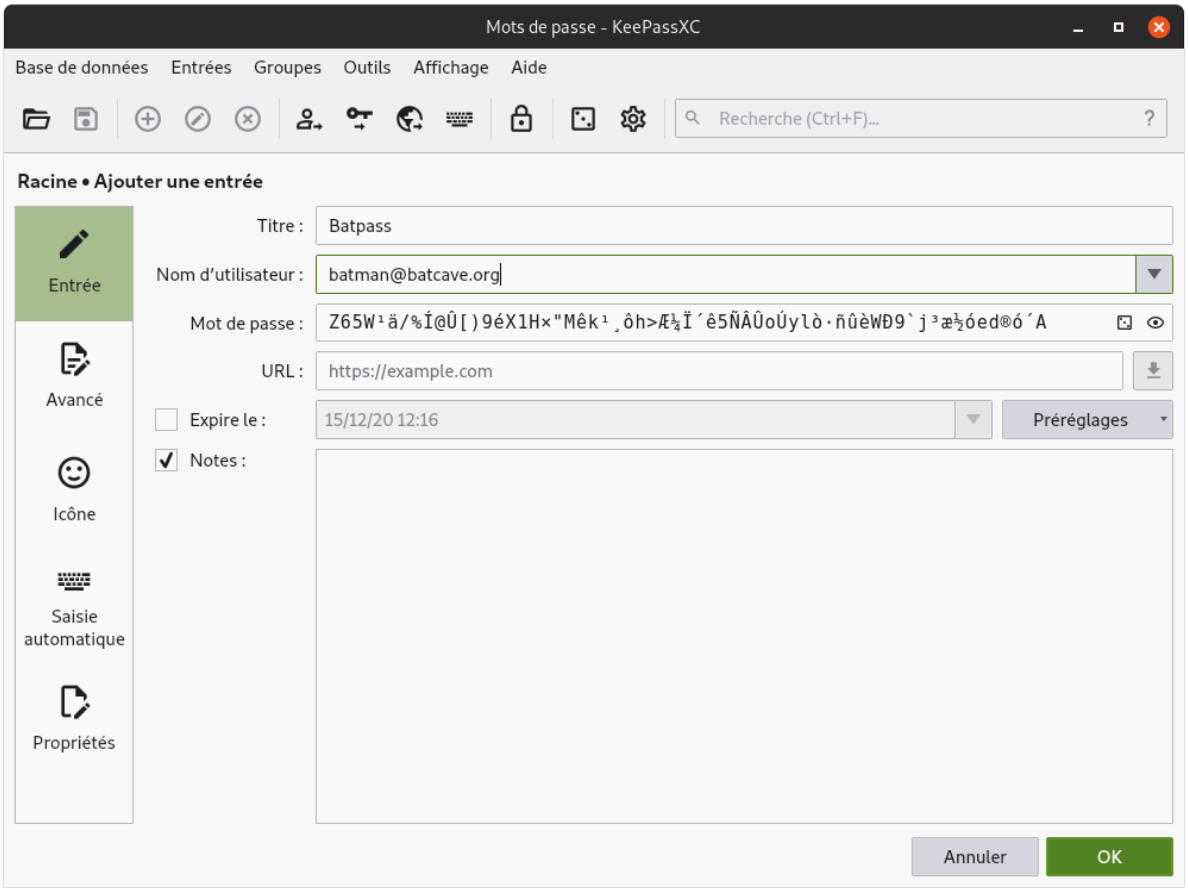
Django gère plusieurs moteurs de base de données. Certains sont gérés nativement par Django (PostgreSQL, MariaDB, SQLite); *a priori*, ces trois-là sont disponibles pour tous les systèmes d'exploitation. D'autres moteurs nécessitent des librairies tierces (Oracle, Microsoft SQL Server).

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.

## 2.11. 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.




## 2.12. 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*" [8] (*Feature A* et *Feature B*).

Chaque "**commit**" correspond à une sauvegarde atomique d'un état ou d'un ensemble de modifications cohérentes entre elles.[9] 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.

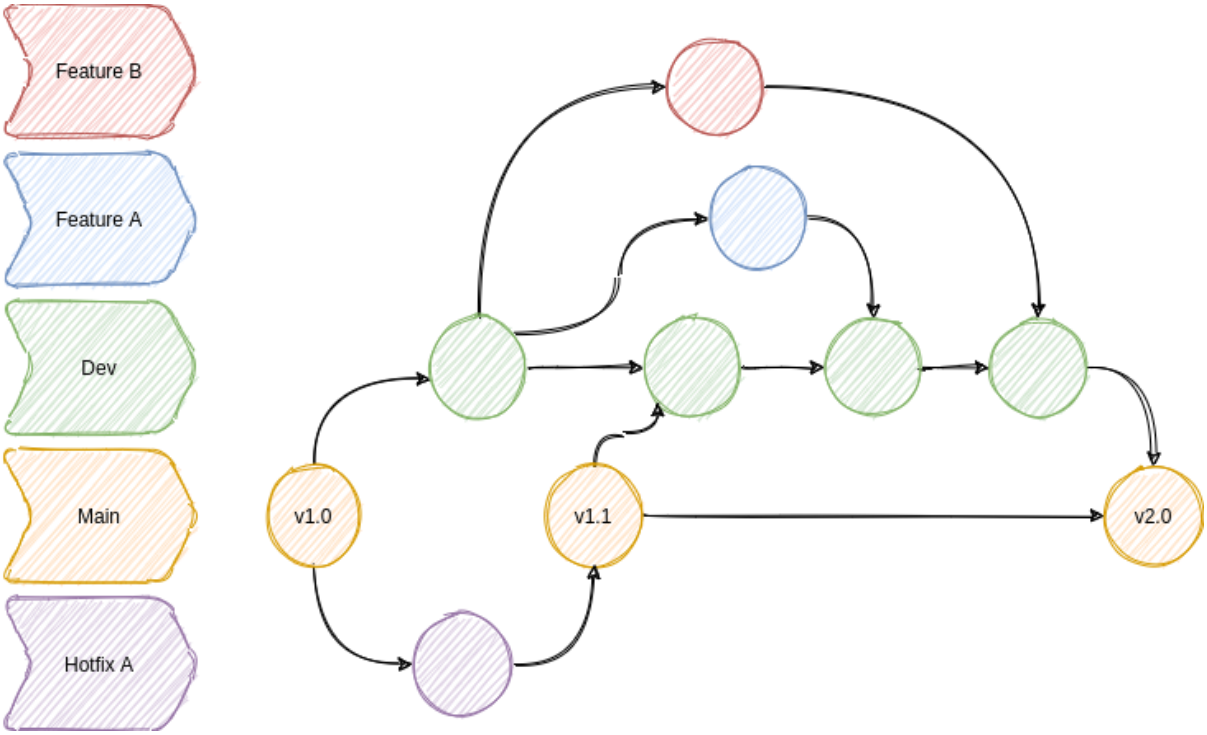



*Figure 6. Git en action*

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.

### 2.12.1. Décrire ses changements

La description d'un changement se fait *via* la commande `git commit`. Il est possible de lui passer directement le message associé à ce changement grâce à l'attribut `-m`, mais c'est une

pratique relativement déconseillée: un *commit* ne doit effectivement pas obligatoirement être décrit sur une seule ligne. Une description plus complète, accompagnée des éventuels tickets ou références, sera plus complète, plus agréable à lire, et plus facile à revoir pour vos éventuels relecteurs.

De plus, la plupart des plateformes de dépôts présenteront ces informations de manière ergonomique. Par exemple:


*Figure 7. Un exemple de commit affiché dans Gitea*

La première ligne est reprise comme titre (normalement, sur 50 caractères maximum); le reste est repris comme de la description.

## 2.13. Un système de virtualisation

Par "*système de virtualisation*", nous entendons n'importe quel application, système d'exploitation, système de containeurisation, ... qui permette de créer ou recréer un environnement de développement aussi proche que celui en production. Les solutions sont nombreuses:

- VirtualBox
- Vagrant
- Docker
- Linux Containers (LXC)
- Hyper-V

Ces quelques propositions se situent un cran plus loin que la "simple" isolation d'un environnement, puisqu'elles vous permettront de construire un environnement complet. Elles constituent donc une étape supplémentaires dans la configuration de votre espace de travail, mais en amélioreront la qualité.

Dans la suite, nous détaillerons Vagrant et Docker, qui constituent deux solutions automatisables et multiplateformes, dont la configuration peut faire partie intégrante de vos sources.

### 2.13.1. Vagrant

Vagrant consiste en un outil de création et de gestion d'environnements virtualisés, en respectant toujours une même manière de travailler, indépendamment des choix techniques et de l'infrastructure que vous pourriez sélectionner.

> Vagrant is a tool for building and managing virtual machine environments in a single workflow. With an easy-to-use workflow and focus on automation, Vagrant lowers development environment setup time, increases production parity, and makes the "works on my machine" excuse a relic of the past. [10]

La partie la plus importante de la configuration de Vagrant pour votre projet consiste à placer un fichier `Vagrantfile` - *a priori* à la racine de votre projet - et qui contiendra les information suivantes:

- Le choix du *fournisseur* (**provider**) de virtualisation (Virtualbox, Hyper-V et Docker sont natifs; il est également possible de passer par VMWare, AWS, etc.)
- Une *box*, qui consiste à lui indiquer le type et la version attendue du système virtualisé (Debian 10, Ubuntu 20.04, etc. - et il y a du choix).
- La manière dont la fourniture (**provisioning**) de l'environnement doit être réalisée: scripts Shell, fichiers, Ansible, Puppet, Chef, ... Choisissez votre favori :-) même s'il est toujours possible de passer par une installation et une maintenance manuelle, après s'être connecté sur la machine.
- Si un espace de stockage doit être partagé entre la machine virtuelle et l'hôte
- Les ports qui doivent être transmis de la machine virtuelle vers l'hôte.

La syntaxe de ce fichier `Vagrantfile` est en Ruby. Vous trouverez ci-dessous un exemple, généré (et nettoyé) après avoir exécuté la commande `vagrant init`:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
Vagrant.configure("2") do |config|

  config.vm.box = "ubuntu/bionic64"

  config.vm.network "forwarded_port", guest: 80, host: 8080, host_ip:
"127.0.0.1"

  config.vm.provider "virtualbox" do |vb|
    vb.gui = true
    vb.memory = "1024"
  end

  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install -y nginx
  SHELL
end
```

Dans le fichier ci-dessus, nous créons:

- Une nouvelle machine virtuelle (ie. *invitée*) sous Ubuntu Bionic Beaver, en x64
- Avec une correspondance du port `80` de la machine vers le port `8080` de l'hôte, en limitant l'accès à celui-ci - accédez à `localhost:8080` et vous accéderez au port `80` de la machine virtuelle.
- En utilisant Virtualbox comme backend - la mémoire vive allouée sera limitée à 1Go de RAM et nous ne voulons pas voir l'interface graphique au démarrage
- Et pour finir, nous voulons appliquer un script de mise à jour `apt-get update` et installer le paquet `nginx`

Par défaut, le répertoire courant (ie. le répertoire dans lequel notre fichier `Vagrantfile` se trouve) sera synchronisé dans le répertoire `/vagrant` sur la machine invitée.

### 2.13.2. Docker

(copié/collé de cookie-cutter-django)

```
version: '3'

volumes:
  local_postgres_data: {}
```

```
  local_postgres_data_backups: {}

services:
  django: &django
    build:
      context: .
      dockerfile: ./compose/local/django/Dockerfile
    image: khana_local_django
    container_name: django
    depends_on:
      - postgres
    volumes:
      - .:/app:z
    env_file:
      - ./.envs/.local/.django
      - ./.envs/.local/.postgres
    ports:
      - "8000:8000"
    command: /start

  postgres:
    build:
      context: .
      dockerfile: ./compose/production/postgres/Dockerfile
    image: khana_production_postgres
    container_name: postgres
    volumes:
      - local_postgres_data:/var/lib/postgresql/data:Z
      - local_postgres_data_backups:/backups:z
    env_file:
      - ./.envs/.local/.postgres

  docs:
    image: khana_local_docs
    container_name: docs
    build:
      context: .
      dockerfile: ./compose/local/docs/Dockerfile
    env_file:
      - ./.envs/.local/.django
    volumes:
      - ./docs:/docs:z
      - ./config:/app/config:z
      - ./khana:/app/khana:z
    ports:
      - "7000:7000"
    command: /start-docs
```

```
  redis:
    image: redis:5.0
    container_name: redis

  celeryworker:
    <<: *django
    image: khana_local_celeryworker
    container_name: celeryworker
    depends_on:
      - redis
      - postgres

    ports: []
    command: /start-celeryworker

  celerybeat:
    <<: *django
    image: khana_local_celerybeat
    container_name: celerybeat
    depends_on:
      - redis
      - postgres

    ports: []
    command: /start-celerybeat

  flower:
    <<: *django
    image: khana_local_flower
    container_name: flower
    ports:
      - "5555:5555"
    command: /start-flower
```

```
# docker-compose.yml
version: '3.8'

services:
  web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - 8000:8000
    depends_on:
      - slqserver
  slqserver:
    image: mcr.microsoft.com/mssql/server:2019-latest
    environment:
      - "ACCEPT_EULA=Y"
      - "SA_PASSWORD=sqklgjqihagrtdgqk12§!"
    ports:
      - 1433:1433
    volumes:
      - ../sqlserver/data:/var/opt/mssql/data
      - ../sqlserver/log:/var/opt/mssql/log
      - ../sqlserver/secrets:/var/opt/mssql/secrets
```

```
FROM python:3.8-slim-buster

ENV PYTHONUNBUFFERED 1
ENV PYTHONDONTWRITEBYTECODE 1

RUN apt-get update \
  # dependencies for building Python packages
  && apt-get install -y build-essential \
  # psycopg2 dependencies
  && apt-get install -y libpq-dev \
  # Translations dependencies
  && apt-get install -y gettext \
  # cleaning up unused files
  && apt-get purge -y --auto-remove -o
APT::AutoRemove::RecommendsImportant=false \
  && rm -rf /var/lib/apt/lists/*

# Requirements are installed here to ensure they will be cached.
COPY ./requirements /requirements
RUN pip install -r /requirements/local.txt

COPY ./compose/production/django/entrypoint /entrypoint
RUN sed -i 's/\r$//g' /entrypoint
RUN chmod +x /entrypoint

COPY ./compose/local/django/start /start
RUN sed -i 's/\r$//g' /start
RUN chmod +x /start

COPY ./compose/local/django/celery/worker/start /start-celeryworker
RUN sed -i 's/\r$//g' /start-celeryworker
RUN chmod +x /start-celeryworker

COPY ./compose/local/django/celery/beat/start /start-celerybeat
RUN sed -i 's/\r$//g' /start-celerybeat
RUN chmod +x /start-celerybeat

COPY ./compose/local/django/celery/flower/start /start-flower
RUN sed -i 's/\r$//g' /start-flower
RUN chmod +x /start-flower

WORKDIR /app

ENTRYPOINT ["/entrypoint"]
```

Voir comment nous pouvons intégrer toutes ces commandes au niveau de la CI et au niveau du déploiement (Docker-compose ?)

### 2.13.3. Base de données

Parfois, SQLite peut être une bonne option:

> Write througput is the area where SQLite struggles the most, but there's not a ton of compelling data online about how it fares, so I got some of my own: I spun up a Equinix m3.large.x86 instance, and ran a slightly modified1 version of the SQLite kvtest2 program on it. Writing 512 byte blobs as separate transactions, in WAL mode with synchronous=normal3, temp_store=memory, and mmap enabled, I got 13.78µs per write, or ~72,568 writes per second. Going a bit larger, at 32kb writes, I got 303.74µs per write, or ~3,292 writes per second. That's not astronomical, but it's certainly way more than most websites being used by humans need. If you had 10 million daily active users, each one could get more than 600 writes per day with that.

> Looking at read throughput, SQLite can go pretty far: with the same test above, I got a read throughput of ~496,770 reads/sec (2.013µs/read) for the 512 byte blob. Other people also report similar results — Expensify reports that you can get 4M QPS if you're willing to make some slightly more involved changes and use a beefier server. Four million QPS is enough that every internet user in the world could make ~70 queries per day, with a little headroom left over4. Most websites don't need that kind of throughput. [4]

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

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

[7] Integrated Development Environment

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

[9] 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.

[10] https://www.vagrantup.com/intro

# Chapitre 3. Démarrer un nouveau projet

## 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 occasioner des conflits entre les applications déployées:

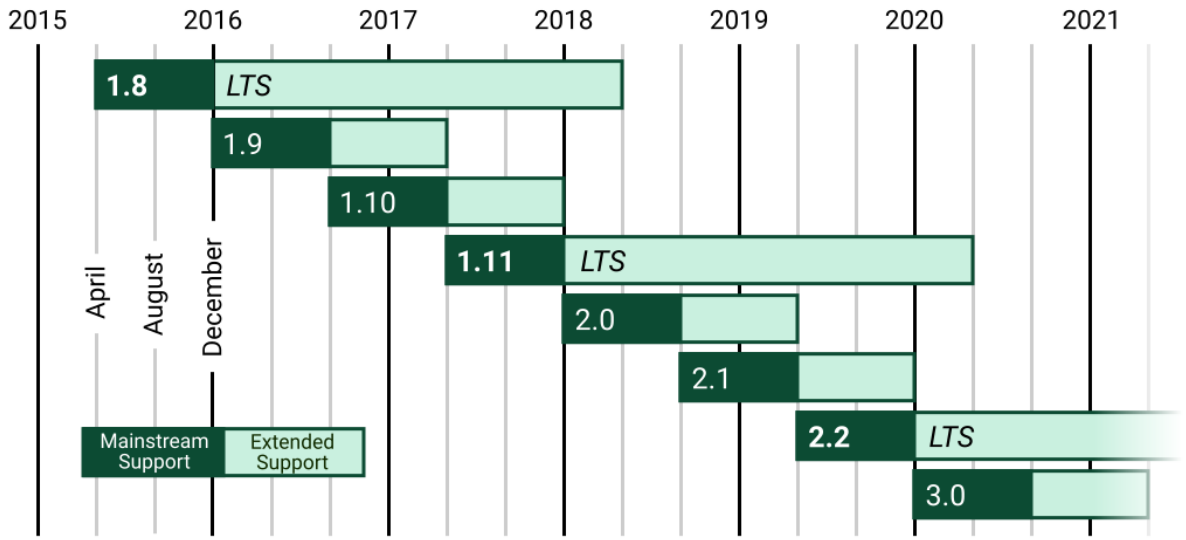
1. Il est tout à fait envisagable que deux applications différentes soient déployées sur un même hôte, et nécessitent chacune deux versions différentes d'une même dépendance.
2. Pour la reproductibilité d'un environnement spécifique, cela évite notamment les réponses type "Ca juste marche chez moi", 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. Roulements de versions

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, mais nous le verrons plus tard...) des dépendances à installer, pour peu que l'on reste sous un certain seuil.

Dans les étapes ci-dessous, nous épinglerons une version LTS afin de nous assurer une certaine sérénité d'esprit (= dont nous ne occuperons pas pendant les 3 prochaines années).

### 3.1.2. Environnements virtuels

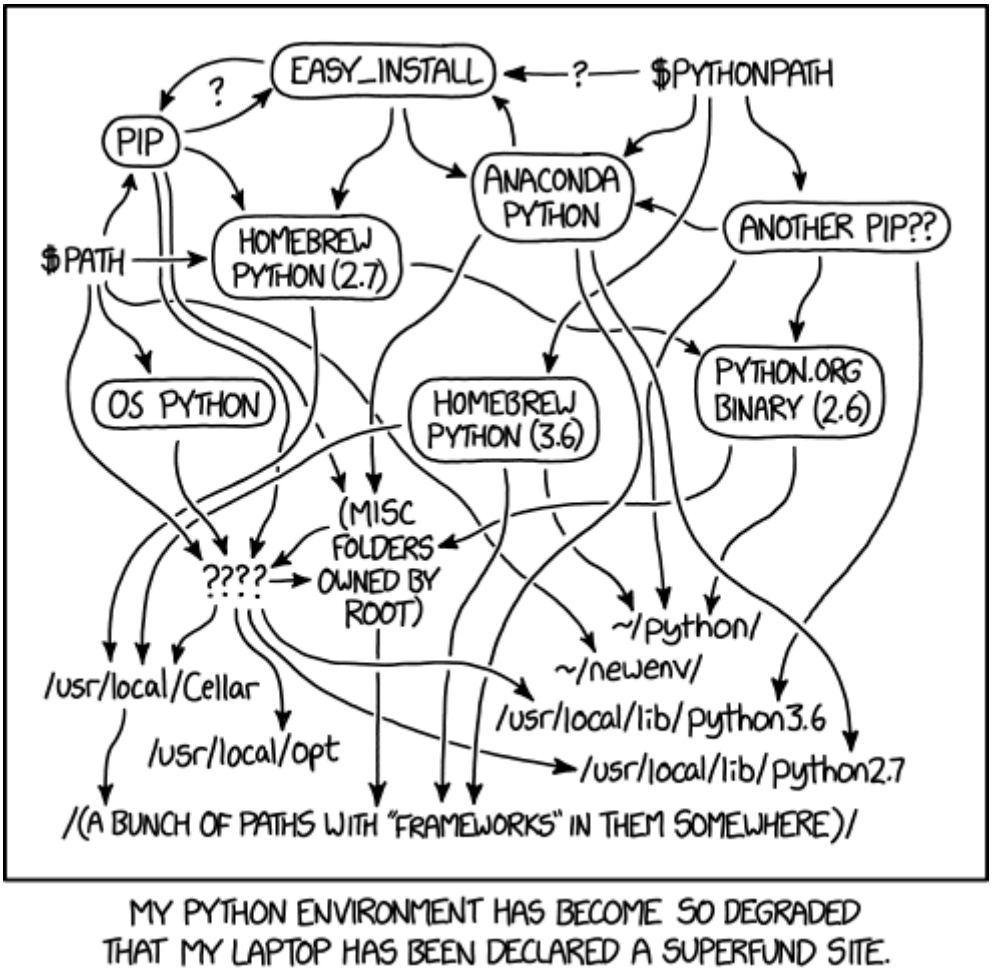



*Figure 8. https://xkcd.com/1987*

Un des reproches que l'on peut faire au langage concerne sa versatilité: il est possible de réaliser beaucoup de choses, mais celles-ci ne sont pas toujours simples ou directes. Pour quelqu'un qui débarquererait, la quantité d'options différentes peut paraître rebutante. Nous pensons notamment aux environnements virtuels: ils sont géniaux à utiliser, mais on est passé par virtualenv (l'ancêtre), virtualenvwrapper (sa version améliorée et plus ergonomique), `venv` (la version intégrée depuis la version 3.3 de l'interpréteur, et la manière recommandée de créer un environnement depuis la 3.5).

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

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.

Poetry se propose de gérer le projet au travers d'un fichier pyproject.toml. TOML (du nom de

son géniteur, Tom Preston-Werner, légèrement CEO de GitHub à ses heures), se place comme alternative aux formats comme JSON, YAML ou INI.

```
La commande poetry new <project> créera une structure par défaut relativement
compréhensible:

$ poetry new django-gecko
$ tree django-gecko/
django-gecko/
├── django_gecko
│   └── __init__.py
├── pyproject.toml
├── README.rst
└── tests
    ├── __init__.py
    └── test_django_gecko.py

2 directories, 5 files
```

Ceci signifie que nous avons directement (et de manière standard):

- Un répertoire django-gecko, qui porte le nom de l'application que vous venez de créer
- Un répertoires tests, libellé selon les standards de pytest
- Un fichier README.rst (qui ne contient encore rien)
- Un fichier pyproject.toml, qui contient ceci:

```
[tool.poetry]
name = "django-gecko"
version = "0.1.0"
description = ""
authors = ["... <...@grimbox.be>"]

[tool.poetry.dependencies]
python = "^3.9"

[tool.poetry.dev-dependencies]
pytest = "^5.2"

[build-system]
requires = ["poetry-core>=1.0.0"]
build-backend = "poetry.core.masonry.api"
```

La commande `poetry init` permet de générer interactivement les fichiers nécessaires à son

intégration dans un projet existant.

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. Elle évite également de rendre ce répertoire "visible" - il ne s'agit au fond que d'un paramètre de configuration lié uniquement à votre environnement de développement; les environnements virtuels étant disposables, il n'est pas conseillé de trop les lier au projet qui l'utilise comme base. Dans la suite de ce chapitre, je considérerai ces mêmes répertoires, mais n'hésitez pas à les modifier.

DANGER: Indépendamment de l'endroit où vous stockerez le répertoire contenant cet environnement, il est primordial de **ne pas le conserver dans votre dépôt de stockager**. Cela irait à l'encontre des douze facteurs, cela polluera inutilement vos sources et créera des conflits avec l'environnement des personnes qui souhaiteraient intervenir sur le projet.

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.

a manière recommandée pour la gestion des dépendances consiste à les épingler dans un fichier requirements.txt, placé à la racine du projet. Ce fichier reprend, ligne par ligne, chaque dépendance et la version nécessaire. Cet épinglage est cependant relativement basique, dans la mesure où les opérateurs disponibles sont ==, ! et >=.

Poetry propose un épinglage basé sur SemVer. Les contraintes qui peuvent être appliquées aux dépendances sont plus touffues que ce que proposent pip -r, avec la présence du curseur ^, qui ne modifiera pas le nombre différent de zéro le plus à gauche:

```
^1.2.3 (où le nombre en question est 1) pourra proposer une mise à jour
jusqu'à la version juste avant la version 2.0.0
^0.2.3 pourra être mise à jour jusqu'à la version juste avant 0.3.0.
...
```

L'avantage est donc que l'on spécifie une version majeure - mineure - patchée, et que l'on pourra spécifier accepter toute mise à jour jusqu'à la prochaine version majeure - mineure patchée (non incluse !).

Une bonne pratique consiste également, tout comme pour npm, à intégrer le fichier de lock (poetry.lock) dans le dépôt de sources: de cette manière, seules les dépendances testées (et intégrées) seront considérées sur tous les environnements de déploiement.

Il est alors nécessaire de passer par une action manuelle (poetry update) pour mettre à jour

le fichier de verrou, et assurer une mise à jour en sécurité (seules les dépendances testées sont prises en compte) et de qualité (tous les environnements utilisent la même version d'une dépendance).

L'ajout d'une nouvelle dépendance à un projet se réalise grâce à la commande `poetry add <dep>`:

```
$ poetry add django
Using version ^3.2.3 for Django

Updating dependencies
Resolving dependencies... (5.1s)

Writing lock file

Package operations: 8 installs, 1 update, 0 removals

  • Installing pyparsing (2.4.7)
  • Installing attrs (21.2.0)
  • Installing more-itertools (8.8.0)
  • Installing packaging (20.9)
  • Installing pluggy (0.13.1)
  • Installing py (1.10.0)
  • Installing wcwidth (0.2.5)
  • Updating django (3.2 -> 3.2.3)
  • Installing pytest (5.4.3)
```

Elle est ensuite ajoutée à notre fichier `pyproject.toml`:

```
[...]

[tool.poetry.dependencies]
python = "^3.9"
Django = "^3.2.3"

[...]
```

Et contrairement à `pip`, pas besoin de savoir s'il faut pointer vers un fichier (`-r`) ou un dépôt VCS (`-e`), puisque Poetry va tout essayer, [dans un certain ordre](https://python-poetry.org/docs/cli/#add). L'avantage également (et cela m'arrive encore souvent, ce qui fait hurler le runner de Gitlab), c'est qu'il n'est plus nécessaire de penser à épingler la dépendance que l'on vient d'installer parmi les fichiers de requirements, puisqu'elles s'y ajoutent automatiquement grâce à la commande `add`.

### 3.1.3. Python packaging made easy

Cette partie dépasse mes compétences et connaissances, dans la mesure où je n'ai jamais rien packagé ni publié sur [pypi.org](pypi.org). Ce n'est pas l'envie qui manque, mais les idées et la nécessité 🤷. Ceci dit, Poetry propose un ensemble de règles et une préconfiguration qui (doivent) énormément facilite(r) la mise à disposition de librairies sur Pypi - et rien que ça, devrait ouvrir une partie de l'écosystème.

Les chapitres 7 et 8 de [Expert Python Programming - Third Edtion](#), écrit par Michal Jaworski et Tarek Ziadé en parlent très bien:

> Python packaging can be a bit overwhelming at first. The main reason for that is the confusion about proper tools for creating Python packages. Anyway, once you create your first package, you will se that this is as hard as it looks. Also, knowing propre, state-of-the-art packaging helps a lot.

En gros, c'est ardu-au-début-mais-plus-trop-après. Et c'est heureusement suivi et documenté par la PyPA (**Python Packaging Authority**).

Les étapes sont les suivantes:

1. Utiliser setuptools pour définir les projets et créer les distributions sources,
2. Utiliser **wheels** pour créer les paquets,
3. Passer par **twine** pour envoyer ces paquets vers PyPI
4. Définir un ensemble d'actions (voire, de plugins nécessaires - lien avec le VCS, etc.) dans le fichier `setup.py`, et définir les propriétés du projet ou de la librairie dans le fichier `setup.cfg`.

Avec Poetry, deux commandes suffisent (théoriquement - puisque je n'ai pas essayé 🤷): `poetry build` et `poetry publish`:

```
$ poetry build
Building geco (0.1.0)
  - Building sdist
  - Built geco-0.1.0.tar.gz
  - Building wheel
  - Built geco-0.1.0-py3-none-any.whl

$ tree dist/
dist/
 ├──── geco-0.1.0-py3-none-any.whl
 └──── geco-0.1.0.tar.gz

0 directories, 2 files
```

Ce qui est quand même 'achement plus simple que d'appréhender tout un écosystème.

### 3.1.4. 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:

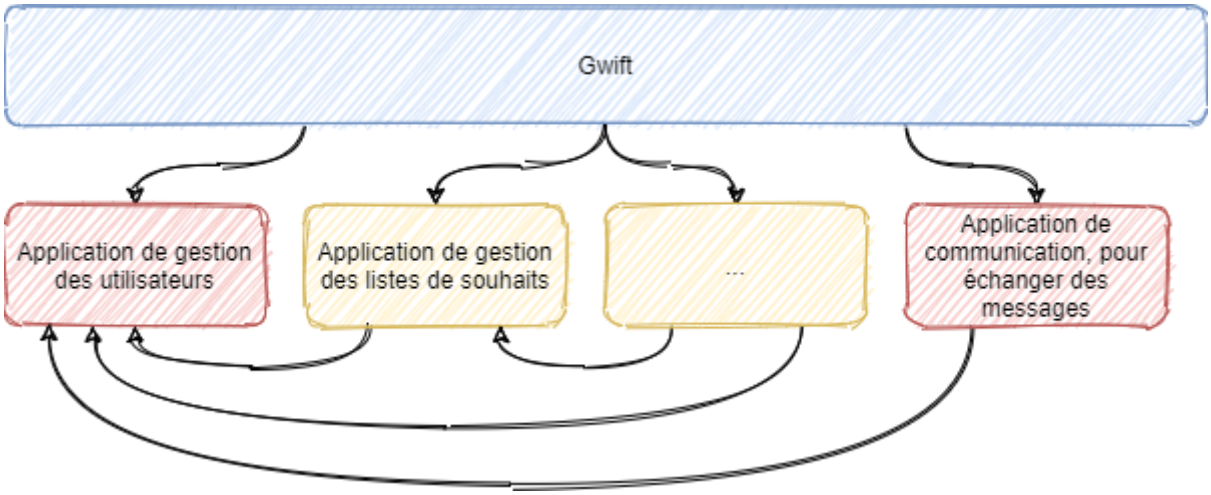



*Figure 9. Django Projet vs Applications*

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:

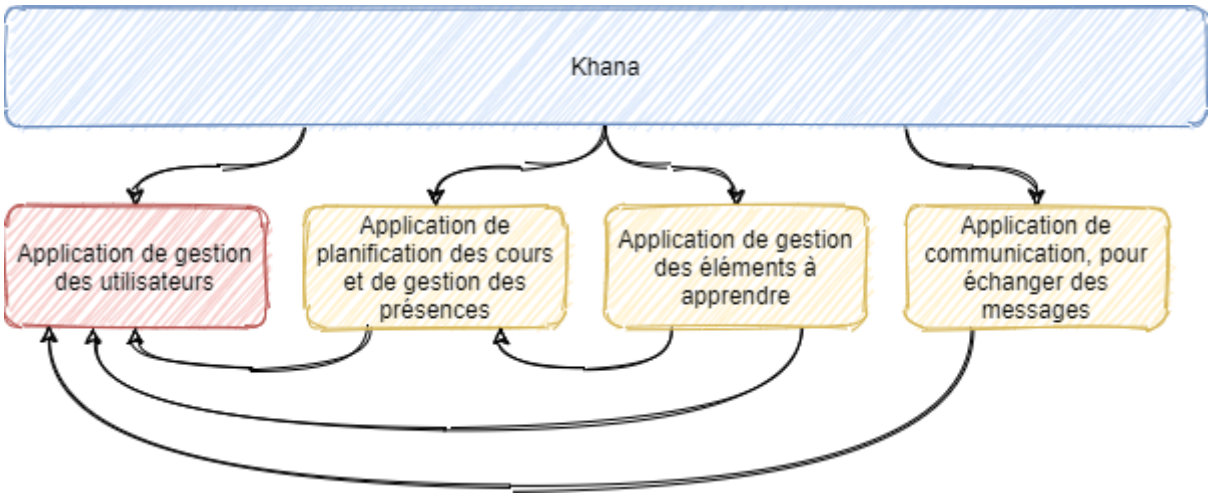



*Figure 10. Django Project vs Applications*

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:

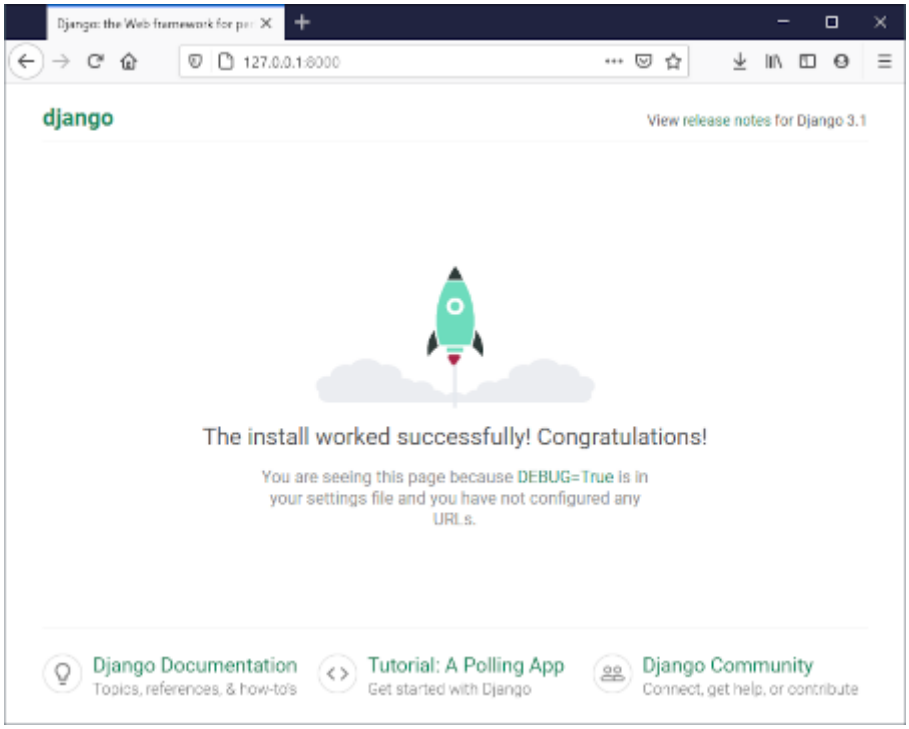



*Figure 11. python manage.py runserver (Non, ce n'est pas Challenger)*

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/apps.py` qui contient la configuration de l'application et qui permet notamment de fixer un nom ou un libellé https://docs.djangoproject.com/en/stable/ref/applications/
- `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

Le métier de programmeur est devenu de plus en plus complexe. Il y a 20 ans, nous pouvions nous contenter d'une simple page PHP dans laquelle nous mixions l'ensemble des actios à réaliser: requêtes en bases de données, construction de la page, ... La recherche d'une solution a un problème n'était pas spécialement plus complexe - dans la mesure où le rendu des enregistrements en direct n'était finalement qu'une forme un chouia plus évoluée du `print()` ou des `System.out.println()` - mais c'était l'évolutivité des applications qui en prenait un coup: une grosse partie des tâches étaient dupliquées entre les différentes pages, et l'ajout d'une nouvelle fonctionnalité était relativement ardue.

Django (et d'autres cadriciels) résolvent ce problème en se basant ouvertement sur le principe de `Don't repeat yourself` [11]. Chaque morceau de code ne doit apparaitre qu'une seule fois, afin de limiter au maximum la redite (et donc, l'application d'un même correctif à

différents endroits).

Le chemin parcouru par une requête est expliqué en (petits) détails ci-dessous.

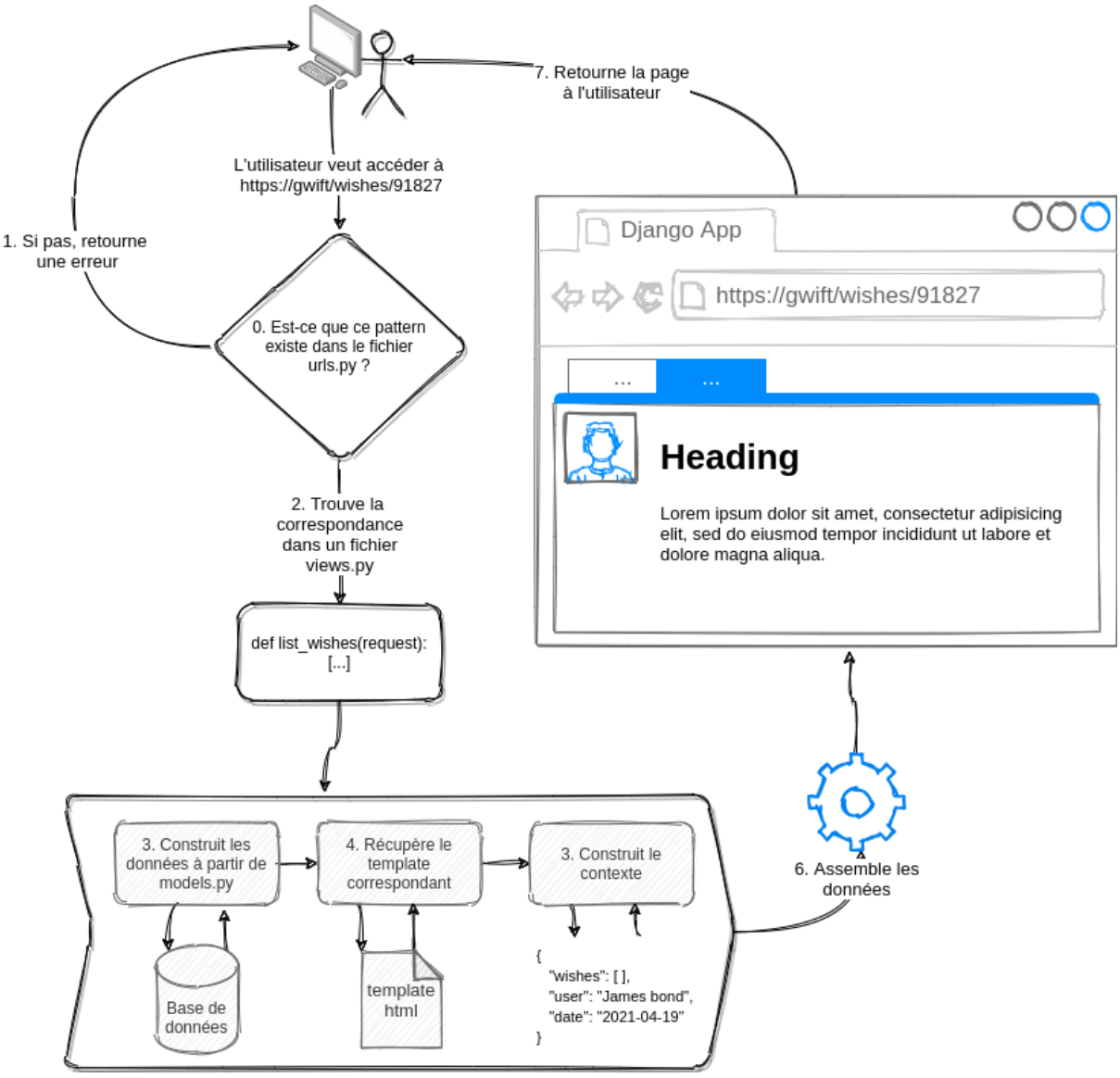



*Figure 12. How it works*

**1. Un utilisateur ou un visiteur souhaite accéder à une URL hébergée et servie par notre application**. Ici, nous prenons l'exemple de l'URL fictive `https://gwift/wishes/91827`. Lorsque cette URL "arrive" dans notre application, son point d'entrée se trouvera au niveau des fichiers `asgi.py` ou `wsgi.py`. Nous verrons cette partie plus tard, et nous pouvons nous concentrer sur le chemin interne qu'elle va parcourir.

**Etape 0** - La première étape consiste à vérifier que cette URL répond à un schéma que nous avons défini dans le fichier `gwift/urls.py`.

**Etape 1** - Si ce n'est pas le cas, l'application n'ira pas plus loin et retournera une erreur à l'utilisateur.

**Etape 2** - Django va parcourir l'ensemble des *patterns* présents dans le fichier `urls.py` et s'arrêtera sur le premier qui correspondra à la requête qu'il a reçue. Ce cas est relativement trivial: la requête `/wishes/91827` a une correspondance au niveau de la ligne `path("wishes/<int:wish_id>` dans l'exemple ci-dessous. Django va alors appeler la fonction [12] associée à ce *pattern*, c'est-à-dire `wish_details` du module `gwift.views`.

```
from django.contrib import admin
from django.urls import path

from gwift.views import wish_details ①

urlpatterns = [
    path('admin/', admin.site.urls),
    path("wishes/<int:wish_id>", wish_details), ②
]
```

① Nous importons la fonction `wish_details` du module `gwift.views`

② Champomy et cotillons! Nous avons une correspondance avec `wishes/details/91827`

TODO: En fait, il faudrait quand même s'occuper du modèle ici. TODO: et de la mise en place de l'administration, parce que nous en aurons besoin pour les étapes de déploiement.

~~Nous n'allons pas nous occuper de l'accès à la base de données pour le moment (nous nous en occuperons dans un prochain chapitre) et nous nous contenterons de remplir un canevas avec un ensemble de données.~~

Le module `gwift.views` qui se trouve dans le fichier `gwift/views.py` peut ressembler à ceci:

```
[...]

from datetime import datetime


def wishes_details(request: HttpRequest, wish_id: int) -> HttpResponse:
    context = {
        "user_name": "Bond,"
        "user_first_name": "James",
        "now": datetime.now()
    }

    return render(
        request,
        "wish_details.html",
        context
    )
```

Pour résumer, cette fonction permet:

1. De construire un *contexte*, qui est représenté sous la forme d'un dictionnaire associant des clés à des valeurs. Les clés sont respectivement `user_name`, `user_first_name` et `now`, tandis que leurs valeurs respectives sont `Bond`, `James` et le `moment présent` [13].
2. Nous passons ensuite ce dictionnaire à un canevas, `wish_details.html`
3. L'application du contexte sur le canevas nous donne un résultat.

```
<!-- fichier wish_details.html -->
<!DOCTYPE html>
<html>
<head>
  <title>Page title</title>
</head>
<body>
  <h1> Hi!</h1>
  <p>My name is {{ user_name }}. {{ user_first_name }} {{ user_name }}.</p>
  <p>This page was generated at {{ now }}</p>
</body>
</html>
```

Après application de notre contexte sur ce template, nous obtiendrons ce document, qui sera renvoyé au navigateur de l'utilisateur qui aura fait la requête initiale:

```
<!DOCTYPE html>
<html>
<head>
  <title>Page title</title>
</head>
<body>
  <h1> Hi!</h1>
  <p>My name is Bond. James Bond.</p>
  <p>This page was generated at 2027-03-19 19:47:38</p>
</body>
</html>
```

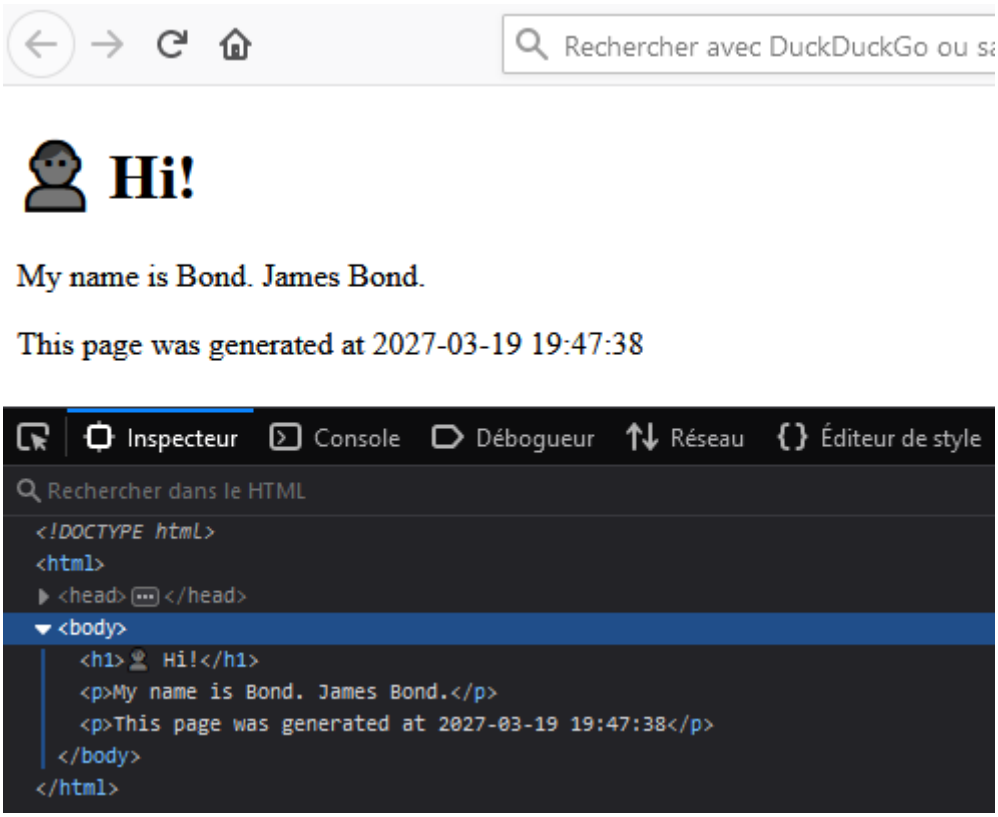



*Figure 13. Résultat*

### 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

### 3.2.5. setup.cfg

(Repris de cookie-cutter-django)

```
[flake8]
max-line-length = 120
exclude = .tox,.git,*/migrations/*,*/static/CACHE/*,docs,node_modules,venv

[pycodestyle]
max-line-length = 120
exclude = .tox,.git,*/migrations/*,*/static/CACHE/*,docs,node_modules,venv

[mypy]
python_version = 3.8
check_untyped_defs = True
ignore_missing_imports = True
warn_unused_ignores = True
warn_redundant_casts = True
warn_unused_configs = True
plugins = mypy_django_plugin.main

[mypy.plugins.django-stubs]
django_settings_module = config.settings.test

[mypy-*.migrations.*]
# Django migrations should not produce any errors:
ignore_errors = True

[coverage:run]
include = khana/*
omit = *migrations*, *tests*
plugins =
    django_coverage_plugin
```

## 3.3. Structure finale de notre environnement

Nous avons donc la structure 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

Pfiou! Ca en fait des commandes et du boulot pour "juste" démarrer un nouveau projet, non? Sachant qu'en plus, nous avons dû modifier des fichiers, déplacer des dossiers, ajouter des dépendances, configurer une base de données, ...

Bonne nouvelle! Il existe des générateurs, permettant de démarrer rapidement un nouveau projet sans (trop) se prendre la tête. Le plus connu (et le plus personnalisable) est Cookie-Cutter, qui se base sur des canevas *type Jinja2*, pour créer une arborescence de dossiers et fichiers conformes à votre manière de travailler. Et si vous avez la flemme de créer votre propre canevas, vous pouvez utiliser ceux qui existent déjà.

Pour démarrer, créez un environnement virtuel (comme d'habitude):

```
λ python -m venv .venvs\cookie-cutter-khana
λ .venvs\cookie-cutter-khana\Scripts\activate.bat
(cookie-cutter-khana) λ pip install cookiecutter

  Collecting cookiecutter
  [...]
  Successfully installed Jinja2-2.11.2 MarkupSafe-1.1.1 arrow-0.17.0
binaryornot-0.4.4 certifi-2020.12.5 chardet-4.0.0 click-7.1.2 cookiecutter-
1.7.2 idna-2.10 jinja2-time-0.2.0 poyo-0.5.0 python-dateutil-2.8.1 python-
slugify-4.0.1 requests-2.25.1 six-1.15.0 text-unidecode-1.3 urllib3-1.26.2

(cookie-cutter-khana) λ cookiecutter https://github.com/pydanny/cookiecutter-
django

  [...]

 [SUCCESS]: Project initialized, keep up the good work!
```

Si vous explorez les différents fichiers, vous trouverez beaucoup de similitudes avec la configuration que nous vous proposions ci-dessus. En fonction de votre expérience, vous serez tenté de modifier certains paramètres, pour faire correspondre ces sources avec votre utilisation ou vos habitudes.

Il est aussi possible d'utiliser l'argument `--template`, suivie d'un argument reprenant le nom de votre projet (`<my_project>`), lors de l'initialisation d'un projet avec la commande `startproject` de `django-admin`, afin de calquer votre arborescence sur un projet existant. La documentation à ce sujet est assez complète.

```
django-admin.py startproject --template=https://[...].zip <my_project>
```

[11] DRY

[12] Qui ne sera pas toujours une fonction. Django s'attend à trouver un *callable*, c'est-à-dire n'importe quel élément qu'il peut appeler comme une fonction.

[13] Non, pas celui d'Eckhart Tolle

# Principes fondamentaux

Dans ce chapitre, nous allons parler de plusieurs concepts fondamentaux au développement rapide d'une application. Nous parlerons de modélisation, de métamodèle, de migrations, d'administration auto-générée, de traductions et de cycle de vie des données.

Django est un framework Web qui propose 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, et en gérant ses dépendances de manière autonome. Pour un néophyte, la courbe d'apprentissage sera relativement ardue: à côté de concepts clés de Django, il conviendra également d'assimiler correctement les structures de données du langage Python, le cycle de vie des requêtes HTTP et le B.A-BA des principes de sécurité.

En restant dans les sentiers battus, votre projet suivra un patron de conception dérivé du modèle `MVC` (Modèle-Vue-Controleur), où la variante concerne les termes utilisés: Django les nomme respectivement Modèle-Template-Vue et leur contexte d'utilisation. 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, lorsqu'une requête est émise par un utilisateur, la première étape va consister à trouver une *route* qui correspond à cette requête, c'est à dire à trouver la correspondance entre l'URL qui est demandée par l'utilisateur et la fonction du langage qui sera exécutée pour fournir le résultat attendu. 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 4. Modélisation

Ce chapitre aborde la modélisation des objets et les options qui y sont liées. Avec Django, la modélisation est en lien direct avec la conception et le stockage, sous forme d'une base de données relationnelle, et la manière dont ces données s'agencent et communiquent entre elles.

Django utilise un paradigme de persistence des données de type ORM - c'est-à-dire que chaque type d'objet manipulé peut s'apparenter à une table SQL, tout en respectant une approche propre à la programmation orientée object. Plus spécifiquement, l'ORM de Django suit le patron de conception Active Records, comme le font par exemple Rails pour Ruby ou EntityFramework pour .Net.

Le modèle de données de Django est sans doute la (seule ?) partie qui soit tellement couplée au framework qu'un changement à ce niveau nécessitera une refonte complète de beaucoup d'autres briques de vos applications; là où un pattern de type Repository permettrait justement de découpler le modèle des données de l'accès à ces mêmes données, un pattern Active Record lie de manière extrêmement forte le modèle à sa persistence. Architecturalement, c'est sans doute la plus grosse faiblesse de Django, à tel point que **ne pas utiliser cette brique de fonctionnalités** peut remettre en question le choix du framework. Conceptuellement, c'est pourtant la manière de faire qui permettra d'avoir quelque chose à présenter très rapidement: à partir du moment où vous aurez un modèle de données, vous aurez accès, grâce à cet ORM à:

1. Des migrations de données,
2. Un découplage complet entre le moteur de données relationnel et le modèle de données,
3. Une interface d'administration auto-générée
4. Un mécanisme de formulaires HTML qui soit complet, pratique à utiliser, orienté objet et facile à faire évoluer,
5. Une définition des notions d'héritage (tout en restant dans une forme d'héritage simple).

Comme tout ceci reste au niveau du code, cela suit également la méthodologie des douze facteurs, concernant la minimisation des divergences entre environnements d'exécution: comme tout se trouve au niveau du code, il n'est plus nécessaire d'avoir un DBA qui doive démarrer un script sur un serveur au moment de la mise à jour, de recevoir une release note de 512 pages en PDF reprenant les modifications ou de nécessiter l'intervention de trois

équipes différentes lors d'une modification majeure du code. Déployer une nouvelle instance de l'application pourra être réalisé directement à partir d'une seule et même commande.

## 4.1. Active Records

Il faut noter que l'implémentation d'Active Records reste une forme hybride entre une structure de données brutes et une classe: là où une classe va exposer ses données derrière une forme d'abstraction et n'exposer que les fonctions qui opèrent sur ces données, une structure de données ne va exposer que ses champs et propriétés, et ne va pas avoir de functions significatives.

L'exemple ci-dessous présente trois structure de données, qui exposent chacune leurs propres champs:

```
class Square:
    def __init__(self, top_left, side):
        self.top_left = top_left
        self.side = side

class Rectangle:
    def __init__(self, top_left, height, width):
        self.top_left = top_left
        self.height = height
        self.width = width

class Circle:
    def __init__(self, center, radius):
        self.center = center
        self.radius = radius
```

Si nous souhaitons ajouter une fonctionnalité permettant de calculer l'aire pour chacune de ces structures, nous aurons deux possibilités:

1. Soit ajouter une classe de *visite* qui ajoute cette fonction de calcul d'aire
2. Soit modifier notre modèle pour que chaque structure hérite d'une classe de type Shape, qui implémentera elle-même ce calcul d'aire.

Dans le premier cas, nous pouvons procéder de la manière suivante:

```
class Geometry:
    PI = 3.141592653589793

    def area(self, shape):
        if isinstance(shape, Square):
            return shape.side * shape.side

        if isinstance(shape, Rectangle):
            return shape.height * shape.width

        if isinstance(shape, Circle):
            return PI * shape.radius**2

        raise NoSuchShapeException()
```

Dans le second cas, l'implémentation pourrait évoluer de la manière suivante:

```python
class Shape:
    def area(self):
        pass

class Square(Shape):
    def __init__(self, top_left, side):
        self.__top_left = top_left
        self.__side = side

    def area(self):
        return self.__side * self.__side

class Rectangle(Shape):
    def __init__(self, top_left, height, width):
        self.__top_left = top_left
        self.__height = height
        self.__width = width

    def area(self):
        return self.__height * self.__width

class Circle(Shape):
    def __init__(self, center, radius):
        self.__center = center
        self.__radius = radius

    def area(self):
        PI = 3.141592653589793
        return PI * self.__radius**2
```

On le voit: une structure brute peut être rendue abstraite au travers des notions de programmation orientée objet. Dans l'exemple géométrique ci-dessus, repris de [7 pp. 95-97], l'accessibilité des champs devient restreinte, tandis que la fonction `area()` bascule comme méthode d'instance plutôt que de l'isoler au niveau d'un visiteur. Nous ajoutons une abstraction au niveau des formes grâce à un héritage sur la classe `Shape`; indépendamment de ce que nous manipulerons, nous aurons la possibilité de calculer son aire.

Une structure de données permet de facilement gérer des champs et des propriétés, tandis qu'une classe gère et facilite l'ajout de fonctions et de méthodes.

Le problème d'Active Records est que chaque classe s'apparente à une table SQL et revient donc à gérer des *DTO* ou *Data Transfer Object*, c'est-à-dire des objets de correspondance pure et simple entre les champs de la base de données et les propriétés de la programmation orientée objet, c'est-à-dire également des classes sans fonctions. Or, chaque classe a également la possibilité d'exposer des possibilités d'interactions au niveau de la persistence,

en enregistrant ses propres données ou en en autorisant leur suppression. Nous arrivons alors à un modèle hybride, mélangeant des structures de données et des classes d'abstraction, ce qui restera parfaitement viable tant que l'on garde ces principes en tête et que l'on se prépare à une éventuelle réécriture du code.

Lors de l'analyse d'une classe de modèle, nous pouvons voir que Django exige un héritage de la classe `django.db.models.Model`. Nous pouvons regarder les propriétés définies dans cette classe en analysant le fichier `lib\site-packages\django\models\base.py`. Outre 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()`, ... En résumé, toutes les méthodes qui font qu'une instance sait **comment** interagir avec la base de données.

## 4.2. Types de champs

## 4.3. Relations et clés étrangères

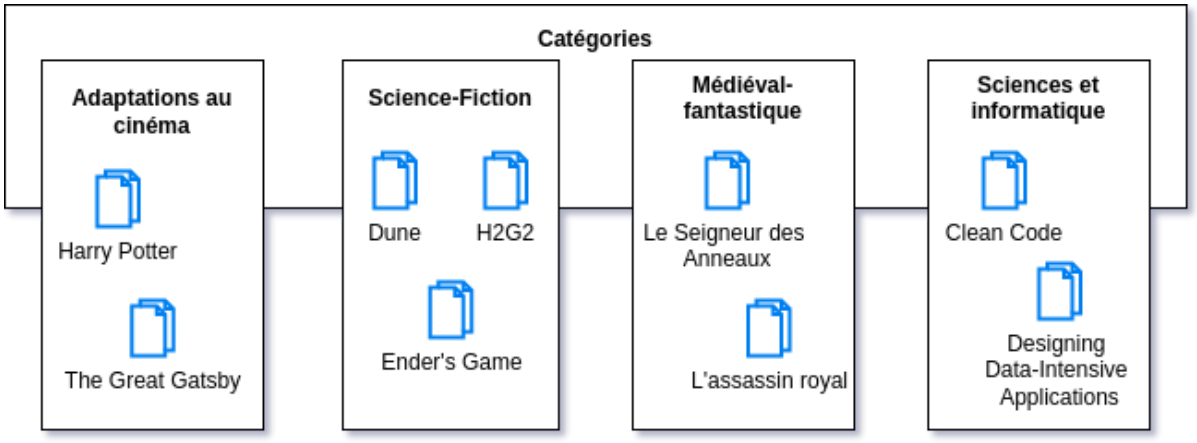
Nous l'avons vu plus tôt, Python est un langage dynamique et fortement typé. Django, de son côté, ajoute une couche de typage statique exigé par le lien sous-jacent avec le moteur de base de données relationnelle. Dans le domaine des bases de données relationnelles, un point d'attention est de toujours disposer d'une clé primaire pour nos enregistrements. Si aucune clé primaire n'est spécifiée, Django s'occupera d'en ajouter une automatiquement et la nommera (par convention) `id`. Elle sera ainsi accessible autant par cette propriété que par la propriété `pk`.

Chaque champ du modèle est donc typé et lié, soit à une primitive, soit à une autre instance au travers de sa clé d'identification.

Grâce à toutes ces informations, nous sommes en mesure de représenter facilement des livres liés à des catégories:

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

class Book(models.Model):
    title = models.CharField(max_length=255)
    category = models.ForeignKey(Category, on_delete=models.CASCADE)
```

Par défaut, et si aucune propriété ne dispose d'un attribut `primary_key=True`, Django s'occupera d'ajouter un champ `id` grâce à son héritage de la classe `models.Model`. Les autres champs nous permettent d'identifier une catégorie (`Category`) par un nom, tandis qu'un livre (`Book`) le sera par ses propriétés `title` et une clé de relation vers une catégorie. Un livre est donc lié à une catégorie, tandis qu'une catégorie est associée à plusieurs livres.




En termes de code d'initialisation, cela revient écrire ceci:

```
from library.models import Book, Category

movies = Category.objects.create(name="Adaptations au cinéma")
medieval = Category.objects.create(name="Médiéval-Fantastique")
science_fiction = Category.objects.create(name="Sciences-fiction")
computers = Category.objects.create(name="Sciences Informatiques")

books = {
    "Harry Potter": movies,
    "The Great Gatsby": movies,
    "Dune": science_fiction,
    "H2G2": science_fiction,
    "Ender's Game": science_fiction,
    "Le seigneur des anneaux": medieval,
    "L'Assassin Royal", medieval,
    "Clean code": computers,
    "Designing Data-Intensive Applications": computers
}

for book_title, category in books.items:
    Book.objects.create(name=book_title, category=category)
```

Nous nous rendons rapidement compte qu'un livre peut appartenir à plusieurs catégories: *Dune* a été adapté au cinéma en 1973 et en 2021, de même que *Le Seigneur des Anneaux*, *The Great Gatsby*, et sans doute que nous pourrons étoffer notre bibliothèque avec une catégorie spéciale "Baguettes magiques et trucs phalliques", à laquelle nous pourrons associer la saga *Harry Potter*. En clair, notre modèle n'est pas adapté, et nous devons le modifier pour que notre clé étrangère accepte plusieurs valeurs. Ceci peut être fait au travers d'un champ de type `ManyToMany`, c'est-à-dire qu'un livre peut être lié à plusieurs catégories, et qu'une catégorie peut être liée à plusieurs livres.

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

class Book(models.Model):
    title = models.CharField(max_length=255)
    category = models.ManyManyField(Category, on_delete=models.CASCADE)
```

Notre code d'initialisation reste par contre identique: Django s'occupe parfaitement de gérer la transition.

### 4.3.1. Accès aux relations

```
# 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')
```

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, nous fixons une valeur à l'attribut `related_name`. Par facilité (et par conventions), prenez l'habitude de toujours ajouter cet attribut: votre modèle gagnera en cohérence et en lisibilité. Si cette relation inverse n'est pas nécessaire, il est possible de l'indiquer (par convention) au travers de l'attribut `related_name="+"`.

A partir de maintenant, nous pouvons accéder à nos propriétés de la manière suivante:

```
# python manage.py shell

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

### 4.3.2. N+1 Queries

## 4.4. Unicité

## 4.5. Indices

### 4.5.1. Conclusions

Dans les examples ci-dessus, nous avons vu les relations multiples (1-N), représentées par des clés étrangères (**ForeignKey**) d'une classe A vers une classe B. Pour représenter d'autres types de relations, il existe également les champs de type **ManyToManyField**, afin de représenter une relation N-N. Il existe également un type de champ spécial pour les clés étrangères, qui est le Les champs de type **OneToOneField**, pour représenter une relation 1-1.

### 4.5.2. Metamodèle et introspection

Comme 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 Nous définissons 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.
- `contraints` (Voir ici-), par exemple

```
    constraints = [ # constraints added
        models.CheckConstraint(check=models.Q(year_born__lte=datetime.date
.today().year-18), name='will_be_of_age'),
    ]
```

### 4.5.3. Choix

Voir ici

```
class Runner(models.Model):

    # this is new:
    class Zone(models.IntegerChoices):
        ZONE_1 = 1, 'Less than 3.10'
        ZONE_2 = 2, 'Less than 3.25'
        ZONE_3 = 3, 'Less than 3.45'
        ZONE_4 = 4, 'Less than 4 hours'
        ZONE_5 = 5, 'More than 4 hours'

    name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    email = models.EmailField()
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=
False)
    start_zone = models.PositiveSmallIntegerField(choices=Zone.choices,
default=Zone.ZONE_5, help_text="What was your best time on the marathon in
last 2 years?") # this is new
```

### 4.5.4. Validateurs

### 4.5.5. 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 :-))
```

## 4.6. Conclusion

Le modèle proposé par Django est un composant extrêmement performant, mais fort couplé avec le coeur du framework. Si tous les composants peuvent être échangés avec quelques manipulations, le cas du modèle sera plus difficile à interchanger.

A côté de cela, il permet énormément de choses, et vous fera gagner un temps précieux, tant en rapidité d'essais/erreurs, que de preuves de concept.

Une possibilité peut également être de cantonner Django à un framework

# Chapitre 5. Migrations

L'intégration des migrations a été réalisée dans la version 1.7 de Django. Avant cela, il convenait de passer par une librairie tierce intitulée South.

Dans cette section, nous allons voir comment fonctionnent les migrations. Lors d'une première approche, elles peuvent sembler un peu magiques, puisqu'elles centralisent un ensemble de modifications pouvant être répétées sur un schéma de données, en tenant compte de ce qui a déjà été appliqué et en vérifiant quelles migrations devaient encore l'être pour mettre l'application à niveau.

Une analyse en profondeur montrera qu'elles ne sont pas plus complexes à suivre et à comprendre qu'un ensemble de fonctions de gestion appliquées à notre application.

Prenons l'exemple de notre liste de souhaits; nous nous rendons (bêtement) compte que nous avons oublié d'ajouter un champ de `description` à une liste. Historiquement, cette action nécessitait l'intervention d'un administrateur système ou d'une personne ayant accès au schéma de la base de données, à partir duquel ce-dit utilisateur pouvait jouer manuellement un script SQL. Cet enchaînement d'étapes nécessitait une bonne coordination d'équipe, mais également une bonne confiance dans les scripts à exécuter. Et souvenez-vous (cf. ref-à-insérer), que l'ensemble des actions doit être répétable et automatisable.

Bref, dans les années '80, il convenait de jouer ceci après s'être connecté à la base de données:

```
ALTER TABLE WishList ADD COLUMN Description nvarchar(MAX);
```

Et là, nous nous rappelons qu'un utilisateur tourne sur Oracle et pas sur MySQL, et qu'il a donc besoin de son propre script d'exécution, parce que le type du nouveau champ n'est pas exactement le même entre les deux moteurs:

```

```

Bref, vous voyez le(s) problème(s):

1. Aucune autonomie

2. Aucune automatisation possible (à moins d'écrire un programme, qu'il faudra également maintenir et intégrer au niveau des tests)
3. Nécessiter de maintenir autant de scripts différents qu'il y a de moteurs de base de données supportés
4. Aucune possibilité de vérifier si le script a déjà été exécuté ou non (à moins de maintenir un programme supplémentaire, à nouveau)
5. ...

Les migrations résolvent la plupart de ces soucis: le framework embarque ses propres applications, dont les migrations, qui gèrent elles-mêmes l'arbre de dépendances entre les modifications devant être appliquées. Une migration consiste donc à appliquer un ensemble de modifications (ou **opérations**), qui exercent un ensemble de transformations, pour que le schéma de base de données corresponde au modèle de l'application sous-jacente. 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.

Une migration est donc une classe Python, présentant *a minima* deux propriétés:

1. `dependencies`, qui décrit les opérations précédentes devant obligatoirement avoir été appliquées
2. `operations`, qui consiste à décrire précisément ce qui doit être exécuté.

Pour reprendre notre exemple d'ajout d'un champ `description` sur le modèle `WishList`, la migration ressemblera à ceci:

```
from django.db import migrations, models
import django.db.models.deletion
import django.utils.timezone


class Migration(migrations.Migration):

    dependencies = [
        ('gwift', '0004_name_value'),
    ]

    operations = [
        migrations.AddField(
            model_name='wishlist',
            name='description',
            field=models.TextField(default="", null=True)
            preserve_default=False,
        ),
    ]
```

Nous avions un modèle reprenant quelques classes, elles-mêmes saupoudrées de quelques propriétés.

## 5.1. Réinitialisation d'une ou plusieurs migrations

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.
> Pour repartir de notre exemple ci-dessus, nous avions un modèle reprenant quelques classes, saupoudrées de propriétés décrivant nos différents champs. Pour être prise en compte par le moteur de base de données, chaque modification doit être

## 5.2. Description d'une migration

1. Décrite, grâce à la commande `makemigrations`

## 5.3. Application d'une ou plusieurs migrations

1. Appliquée, avec la commande `migrate`.

## 5.4. Analyse

Nous allons ci-dessous analyser exactement les modifications appliquées au schéma de la base de données, en fonction des différents cas, et comment ils sont gérés par les pilotes de Django. Nous utiliserons Sqlite Browser et la commande `sqldump`, qui nous présentera le schéma tel qu'il sera compris.

### 5.4.1. Création de nouveaux champs

### 5.4.2. Modification d'un champ existant

### 5.4.3. Suppression d'un champ existant

# Chapitre 6. Shell

# Chapitre 7. 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, un *back office* sans transpirer; 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.

Cette fonctionnalité 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 aussi 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 de données
2. Les validateurs

3. Les formulaires
4. Les widgets

## 7.1. Le modèle de données

Comme expliqué ci-dessus, le modèle de données est constité d'un ensemble de champs typés et de relations. L'administration permet de décrire les données qui peuvent être modifiées, en y associant un ensemble (basique) de permissions.

Si vous vous rappelez de l'application que nous avions créée dans la première partie, les URLs reprenaient déjà la partie suivante:

```
from django.contrib import admin
from django.urls import path

from gwift.views import wish_details

urlpatterns = [
    path('admin/', admin.site.urls), ①
    [...]
]
```

① Cette URL signifie que la partie `admin` est déjà active et accessible à l'URL `<mon_site>/admin`

C'est le seul prérequis pour cette partie.

Chaque application nouvellement créée contient par défaut un fichier `admin.py`, dans lequel il est possible de déclarer quel ensemble de données sera accessible/éditable. Ainsi, si nous partons du modèle basique que nous avions détaillé plus tôt, avec des souhaits et des listes de souhaits:

```
# gwift/wish/models.py

from django.db import models


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


class Item(models.Model):
    name = models.CharField(max_length=255)
    wishlist = models.ForeignKey(WishList, on_delete=models.CASCADE)
```

Nous pouvons facilement arriver au résultat suivant, en ajoutant quelques lignes de configuration dans ce fichier `admin.py`:

```
from django.contrib import admin

from .models import Item, WishList ①


admin.site.register(Item) ②
admin.site.register(WishList)
```

① Nous importons les modèles que nous souhaitons gérer dans l'admin

② Et nous les déclarons comme gérables. Cette dernière ligne implique aussi qu'un modèle pourrait ne pas être disponible du tout, ce qui n'activera simplement aucune opération de lecture ou modification.

Il nous reste une seule étape à réaliser: créer un nouvel utilisateur. Pour cet exemple, notre gestion va se limiter à une gestion manuelle; nous aurons donc besoin d'un *super-utilisateur*, que nous pouvons créer grâce à la commande `python manage.py createsuperuser`.

```
λ python manage.py createsuperuser
Username (leave blank to use 'fred'): fred
Email address: fred@root.org
Password: ******
Password (again): ******
Superuser created successfully.
```

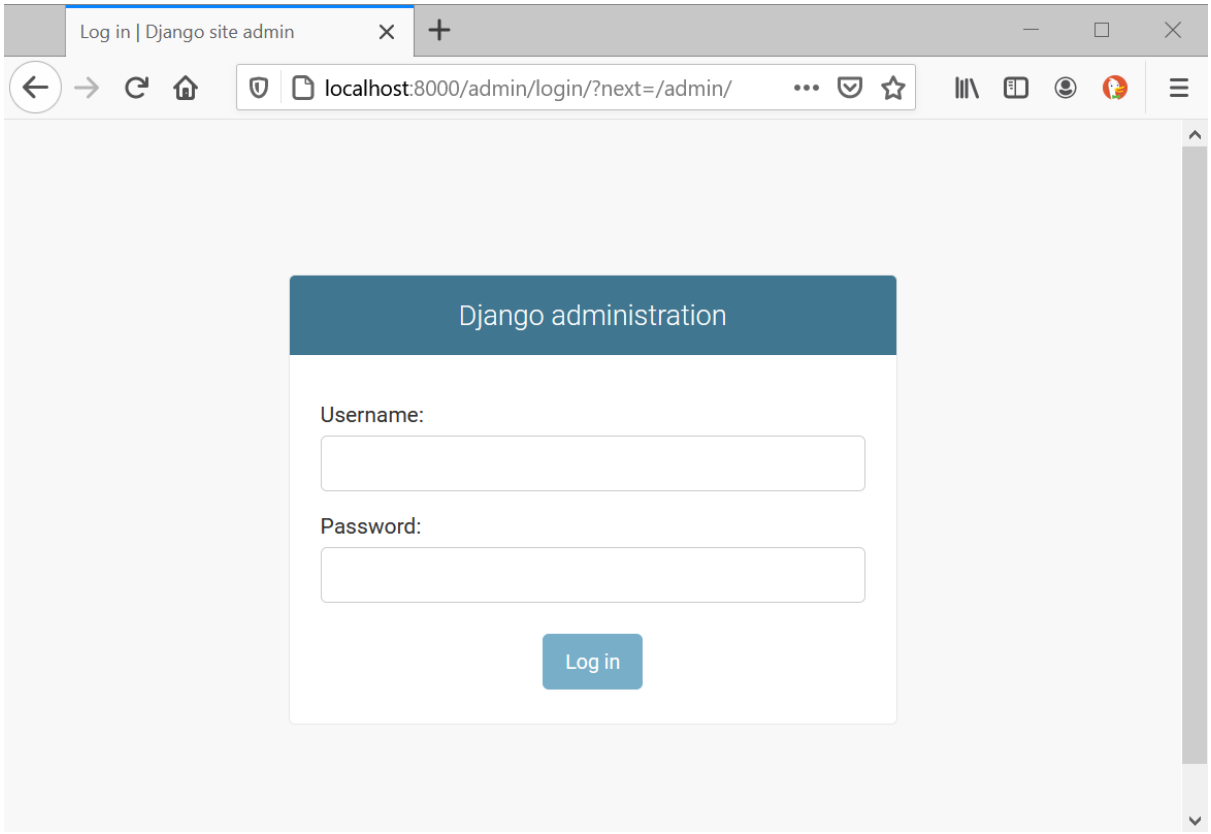



*Figure 14. Connexion au site d'administration*

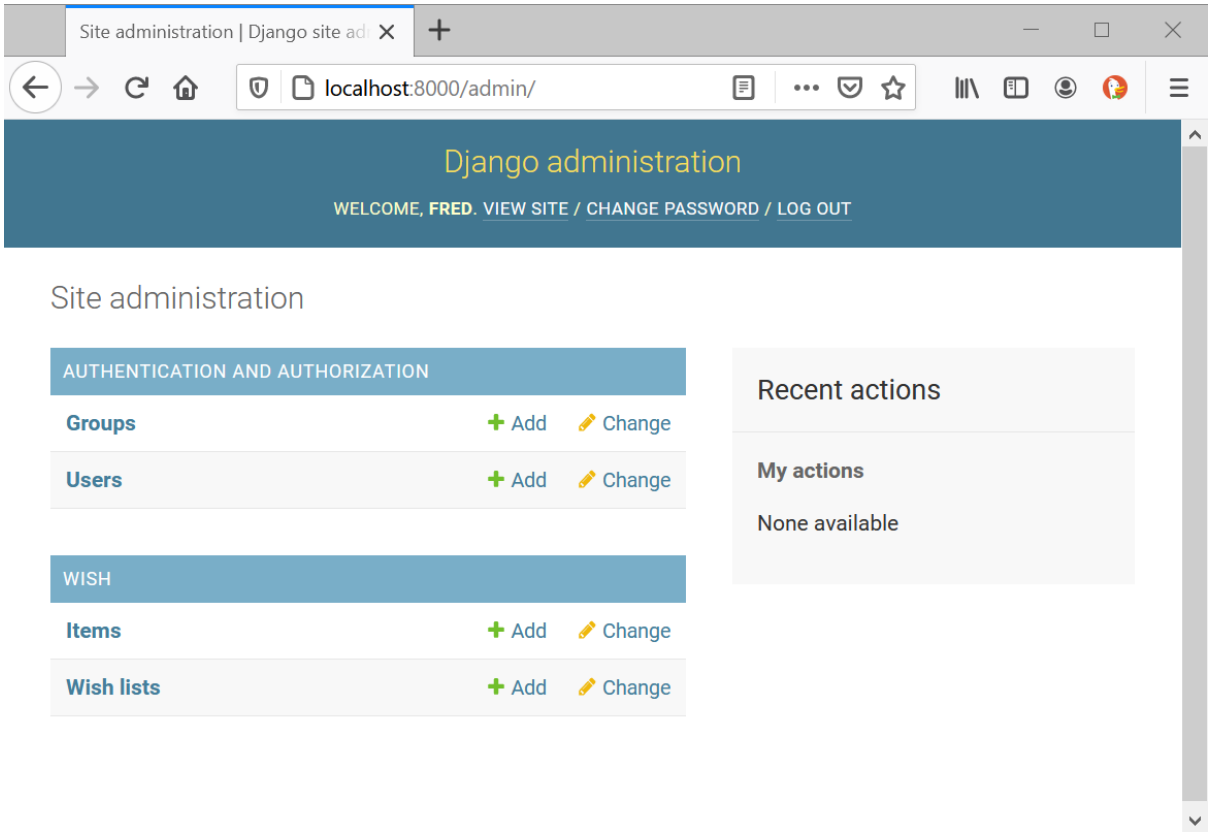



*Figure 15. Administration*

## 7.2. Quelques conseils de base

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.

## 7.3. 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)

## 7.4. 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 ?

## 7.5. Les filtres

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

## 7.6. 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 :-)

# 7.7. Les relations

## 7.7.1. 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.

## 7.7.2. Les auto-suggestions et auto-complétions

Parler de l'intégration de select2.

## 7.8. La présentation

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

## 7.9. 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))
```

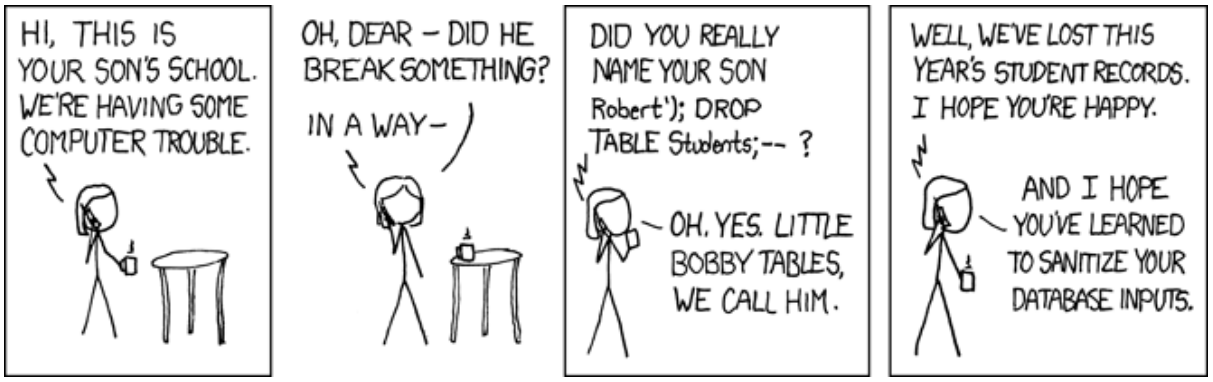
## 7.10. La documentation

Nous l'avons dit plus haut, l'administration de Django a également la possibilité de rendre accessible la documentation associée à un modèle de données. Pour cela, il suffit de suivre les bonnes pratiques, puis d'activer la documentation à partir des URLs:

```
```

# Chapitre 8. 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.

## 8.1. Flux de validation

| .Validation | .is_valid | .clean_fields ↓ .clean_fields_machin

A compléter ;-)

## 8.2. 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.

## 8.3. 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, ...)'
                            })
        }
```

## 8.4. 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`.

## 8.5. 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

## 8.6. En conclusion

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

# Chapitre 9. 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.

## 9.1. 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**.

## 9.2. 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.

## 9.3. 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.

## 9.4. 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>`.

### 9.4.1. Backend

### 9.4.2. 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`.

### 9.4.3. Social-Authentification

Voir ici : python social auth

### 9.4.4. 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.

```
INSTALLED_APPS = [
    "django.contrib..."
]
```

peut être splitté en plusieurs parties:

```
INSTALLED_APPS = [

]

THIRD_PARTIES = [

]

MY_APPS = [

]
```

## 9.5. Tests

> Tests are part of the system.
>
> — Robert C. Martin, Clean Architecture

### 9.5.1. Types de tests

Les **tests unitaires** ciblent typiquement une seule fonction, classe ou méthode, de manière isolée, en fournissant au développeur l'assurance que son code réalise ce qu'il en attend. Pour plusieurs raisons (et notamment en raison de performances), les tests unitaires utilisent souvent des données stubbées - pour éviter d'appeler le "vrai" service.

> The aim of a unit test is to show that a single part of the application does what programmer intends it to.

Les **tests d'acceptance** vérifient que l'application fonctionne comme convenu, mais à un plus haut niveau (fonctionnement correct d'une API, validation d'une chaîne d'actions effectuées par un humain, ...).

> The objective of acceptance tests is to prove that our application does what the customer meant it to.

Les **tests d'intégration** vérifient que l'application coopère correctement avec les systèmes

périphériques.

De manière plus générale, si nous nous rendons compte que les tests sont trop compliqués à écrire ou nous coûtent trop de temps, c'est sans doute que l'architecture de la solution n'est pas adaptée et que les composants sont couplés les uns aux autres. Dans ces cas, il sera nécessaire de refactoriser le code, afin que chaque module puisse être testé indépendamment des autres. [8]

> Martin Fowler observes that, in general, "a ten minute build [and test process] is perfectly within reason... [We first] do the compilation and run tests that are more localized unit tests with the database completely stubbed out. Such tests can run very fast, keeping within the ten minutes guideline. However any bugs that involve larger scale intercations, particularly those involving the real database, won't be found. The second stage build runs a different suite of tests [acceptance tests] that do hit the real database and involve more end-to-end behavior. This suite may take a couple of hours to run.
>
> — Robert C. Martin, Clean Architecture

Au final, le plus important est de toujours corréler les phases de tests indépendantes du reste du travail (de développement, ici), en l'automatisant au plus près de sa source de création.

En résumé, il est recommandé de:

1. Tester que le nommage d'une URL (son attribut `name` dans les fichiers `urls.py`) corresponde à la fonction que l'on y a définie
2. Tester que l'URL envoie bien vers l'exécution d'une fonction (et que cette fonction est celle que l'on attend)

TODO: Voir comment configurer une `memoryDB` pour l'exécution des tests.

### 9.5.2. Tests de nommage

```
from django.core.urlresolvers import reverse
from django.test import TestCase


class HomeTests(TestCase):
    def test_home_view_status_code(self):
        url = reverse("home")
        response = self.client.get(url)
        self.assertEquals(response.status_code, 200)
```

### 9.5.3. Tests d'urls

```
from django.core.urlresolvers import reverse
from django.test import TestCase

from .views import home


class HomeTests(TestCase):
    def test_home_view_status_code(self):
        view = resolve("/")
        self.assertEquals(view.func, home)
```

# Chapitre 10. Conclusions

# Déploiement

> To be effective, a software system must be deployable. The higher the cost of deployements, the less useful the system is. A goal of a software architecture, then, should be to make a system that can be easily deployed with a single action. Unfortunately, deployment strategy is seldom considered during initial development. This leads to architectures that may be make the system easy to develop, but leave it very difficult to deploy.
>
> — Robert C. Martin, Clean Architecture

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 et accessible depuis 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 puissent être mises à disposition sur un environnement en particulier, une fois que leur développement aura été finalisé, testé et validé. Un bon déploiement ne doit pas dépendre de dizaines de petits scripts éparpillés sur le disque. L'objectif est qu'il soit rapide et fiable. Ceci peut être atteint au travers d'un partitionnement correct, incluant le fait que le composant principal s'assure que chaque sous-composant est correctement démarré intégré et supervisé.

Aborder le déploiement dès le début permet également de rédiger dès le début les procédures d'installation, de mises à jour et de sauvegardes. A la fin de chaque intervalle de développement, les fonctionnalités auront dû avoir été intégrées, testées, fonctionnelles et un code propre, démontrable dans un environnement similaire à un environnement de production, et créées à partir d'un tronc commun au développement [6].

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.

> Because value is created only when our services are running into production, we must ensure that we are not only delivering fast flow, but that our deployments can also be performed without causing chaos and disruptions such as service outages, service impairments, or security or compliance failures.
>
> — DevOps Handbook, Introduction

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 simultanés. 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, qui répond à un besoin, mais qui va attirer la grogne de ses utilisateurs pour des problèmes de latences, pour des erreurs de verrou ou simplement parce que le serveur répondra trop lentement.

L'objectif de cette partie est de parcourir les différentes possibilités qui s'offrent à nous en termes de déploiement, tout en faisant en sorte que le code soit le moins couplé possible à sa destination de production. L'objectif est donc de faire en sorte qu'une même application puisse être hébergées par plusieurs hôtes sans avoir à subir de modifications. Nous vous renvoyons vers les 12-facteurs dont nous avons déjà parlé et qui vous énormément nous aider, puisque ce sont des variables d'environnement qui vont réellement piloter le câblage entre l'application, ses composants et son hébergeur.

RedHat proposait récemment un article intitulé **What Is IaaS**, qui présentait les principales différences entre types d'hébergement.

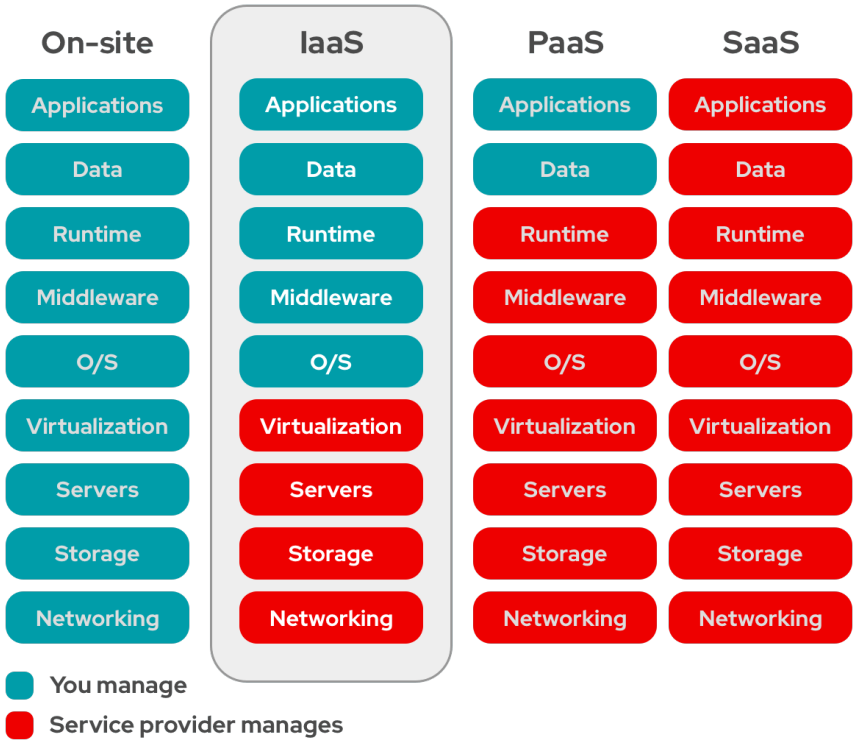



*Figure 16. L'infrastructure en tant que service, cc. RedHat Cloud Computing*

Ainsi, on trouve:

1. Le déploiment *on-premises* ou *on-site*
2. Les *Infrastructures as a service* ou *IaaS*
3. Les *Platforms as a service* ou *PaaS*
4. Les *Softwares as a service* ou *SaaS*, ce dernier point nous concernant moins, puisque c'est nous qui développons le logiciel.

Dans cette partie, nous aborderons les points suivants:

1. Définir l'infrastructure et les composants nécessaires à notre application
2. Configurer l'hôte qui hébergera l'application et y déployer notre application: dans une machine physique, virtuelle ou dans un container. Nous aborderons aussi les déploiements via Ansible et Salt. A ce stade, nous aurons déjà une application disponible.
3. 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 correctement l'information.

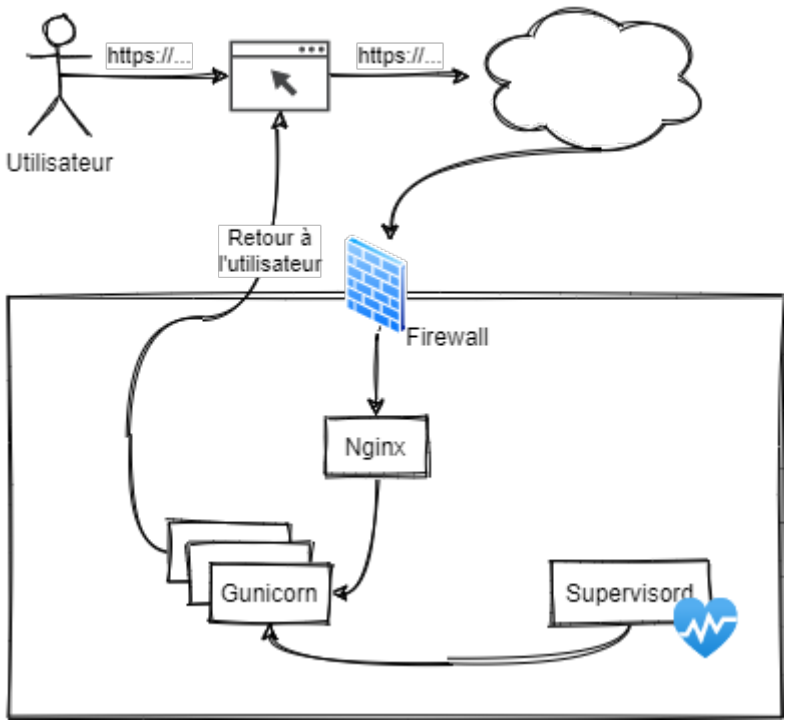
# Chapitre 11. Infrastructure & composants

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

- Nginx comme reverse proxy
- HAProxy pour la distribution de charge
- Gunicorn ou Uvicorn comme serveur d'application
- Supervisor pour le monitoring
- PostgreSQL ou MySQL/MariaDB comme bases 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).
- Sentry, pour le suivi des bugs

Si nous schématisons l'infrastructure et le chemin parcouru par une requête, nous pourrions arriver à la synthèse suivante:

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 un des *workers* (= un processus Python) instancié par Gunicorn. Si l'un de ces travailleurs venait à planter, il serait automatiquement réinstancié par Supervisord.
6. Qui la transmet ensuite à l'un de ses *workers* (= un processus Python).
7. Après exécution, une réponse est renvoyée à l'utilisateur.




## 11.1. Reverse proxy

Le principe du **proxy inverse** est de pouvoir rediriger du trafic entrant vers une application hébergée sur le système. Il serait tout à fait possible de rendre notre application directement accessible depuis l'extérieur, mais le proxy a aussi l'intérêt de pouvoir élever la sécurité du serveur (SSL) et décharger le serveur applicatif grâce à un mécanisme de cache ou en compressant certains résultats [14]

## 11.2. Load balancer

## 11.3. Workers

## 11.4. Supervision des processus

## 11.5. Base de données

## 11.6. Tâches asynchrones

## 11.7. Mise en cache

[14] https://fr.wikipedia.org/wiki/Proxy_inverse

# Chapitre 12. 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 13. Outils de supervision et de mise à disposition

## 13.1. Logs

# Chapitre 14. Logging

La structure des niveaux de journaux est essentielle.

> When deciding whether a message should be ERROR or WARN, imagine being woken up at 4 a.m. Low printer toner is not an ERROR.
>
> — Dan North, former ToughtWorks consultant

- **DEBUG**: Il s'agit des informations qui concernent tout ce qui peut se passer durant l'exécution de l'application. Généralement, ce niveau est désactivé pour une application qui passe en production, sauf s'il est nécessaire d'isoler un comportement en particulier, auquel cas il suffit de le réactiver temporairement.
- **INFO**: Enregistre les actions pilotées par un utilisateur - Démarrage de la transaction de paiement, ...
- **WARN**: Regroupe les informations qui pourraient potentiellement devenir des erreurs.
- **ERROR**: Indique les informations internes - Erreur lors de l'appel d'une API, erreur interne, ...
- **FATAL** (ou **EXCEPTION**): ... généralement suivie d'une terminaison du programme ;-) - Bind raté d'un socket, etc.

La configuration des *loggers* est relativement simple, un peu plus complexe si nous nous penchons dessus, et franchement complète si nous creusons encore. Il est ainsi possible de définir des formattages, gestionnaires (*handlers*) et loggers distincts, en fonction de nos applications.

Sauf que comme nous l'avons vu avec les 12 facteurs, nous devons traiter les informations de notre application comme un flux d'évènements. Il n'est donc pas réellement nécessaire de chipoter la configuration, puisque la seule classe qui va réellement nous intéresser concerne les `StreamHandler`. La configuration que nous allons utiliser est celle-ci:

1. Formattage: à définir - mais la variante suivante est complète, lisible et pratique: `{levelname} {asctime} {module} {process:d} {thread:d} {message}`
2. Handler: juste un, qui définit un `StreamHandler`
3. Logger: pour celui-ci, nous avons besoin d'un niveau (`level`) et de savoir s'il faut propager

les informations vers les sous-paquets, auquel cas il nous suffira de fixer la valeur de `propagate` à `True`.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'verbose': {
            'format': '{levelname} {asctime} {module} {process:d} {thread:d}
{message}',
        },
        'simple': {
            'format': '{levelname} {asctime} {module} {message}',
        },
    },
    'handlers': {
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
            'formatter': "verbose"
        }
    },
    'loggers': {
        'khana': {
            'handlers': ['console'],
            'level': env("LOG_LEVEL", default="DEBUG"),
            'propagate': True,
        },
    }
}
```

Pour utiliser nos loggers, il suffit de copier le petit bout de code suivant:

```
import logging

logger = logging.getLogger(__name__)

logger.debug('helloworld')
```

Par exemples.

## 14.1. Sentry ! :-D

```
SENTRY_DSN = env("SENTRY_DSN", default=None)

if SENTRY_DSN is not None:
    import sentry_sdk
    from sentry_sdk.integrations.django import DjangoIntegration

    sentry_sdk.init(
        dsn=SENTRY_DSN,
        integrations=[DjangoIntegration()],

        # Set traces_sample_rate to 1.0 to capture 100%
        # of transactions for performance monitoring.
        # We recommend adjusting this value in production.
        traces_sample_rate=1.0,

        # If you wish to associate users to errors (assuming you are using
        # django.contrib.auth) you may enable sending PII data.
        send_default_pii=True,
        ca_certs=<path_to_pem_file>,
    )
```

## 14.2. Logging

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

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

# Chapitre 15. 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.

# Chapitre 16. Déploiement sur Debian

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 [15].

```
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

## 16.1. 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.

## 16.2. Installation de la base 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.

### 16.2.1. 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.

### 16.2.2. MariaDB

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

### 16.2.3. Microsoft SQL Server

### 16.2.4. Oracle

## 16.3. 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
```

## 16.4. 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.

## 16.5. Création des répertoires de logs

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

## 16.6. 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
```

## 16.7. 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=-
```

## 16.8. 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
```

## 16.9. Configuration du firewall et 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).

## 16.10. 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/ { ①
                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;
②
                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

## 16.11. 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
kill -HUP `ps -C gunicorn fch -o pid | head -n 1` ①
```

① https://stackoverflow.com/questions/26902930/how-do-i-restart-gunicorn-hup-i-dont-know-masterpid-or-location-of-pid-file

## 16.12. 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
```

## 16.13. 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.

## 16.14. Ansible

TODO

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

# Chapitre 17. Déploiement sur Heroku

Heroku est une *Plateform As A Service* , où vous choisissez le *service* dont vous avez besoin (une base de données, un service de cache, un service applicatif, ...), vous lui envoyer les paramètres nécessaires et le tout démarre gentiment sans que vous ne deviez superviser l'hôte. Ce mode démarrage ressemble énormément aux 12 facteurs dont nous avons déjà parlé plus tôt - raison de plus pour que notre application soit directement prête à y être déployée, d'autant plus qu'il ne sera pas possible de modifier un fichier une fois qu'elle aura démarré: si vous souhaitez modifier un paramètre, cela reviendra à couper l'actuelle et envoyer de nouveaux paramètres et recommencer le déploiement depuis le début.

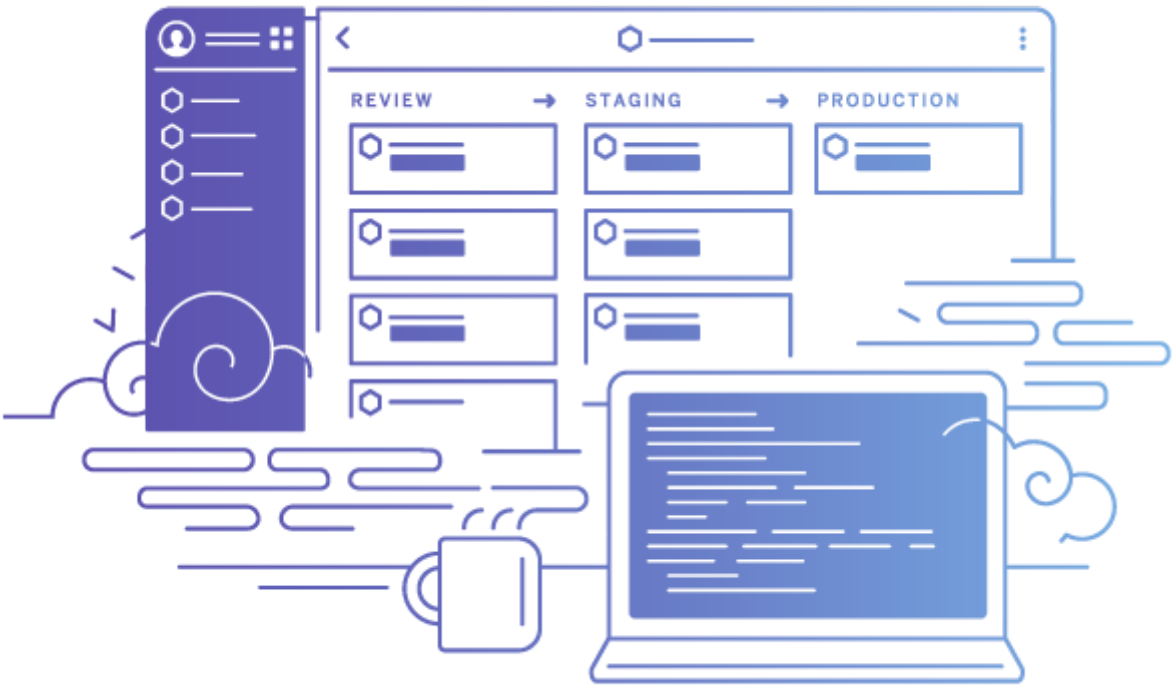



*Figure 17. Invest in apps, not ops. Heroku handles the hard stuff — patching and upgrading, 24/7 ops and security, build systems, failovers, and more — so your developers can stay focused on building great apps.*

Pour un projet de type "hobby" et pour l'exemple de déploiement ci-dessous, il est tout à fait possible de s'en sortir sans dépenser un kopek, afin de tester nos quelques idées ou mettre

rapidement un *Most Valuable Product* en place. La seule contrainte consistera à pouvoir héberger des fichiers envoyés par vos utilisateurs - ceci pourra être fait en configurant un *bucket compatible S3*, par exemple chez Amazon, Scaleway ou OVH.

Le fonctionnement est relativement simple: pour chaque application, Heroku crée un dépôt Git qui lui est associé. Il suffit donc d'envoyer les sources de votre application vers ce dépôt pour qu'Heroku les interprête comme étant une nouvelle version, déploie les nouvelles fonctionnalités - sous réserve que tous les tests passent correctement - et les mettent à disposition. Dans un fonctionnement plutôt manuel, chaque déploiement est initialisé par le développeur ou par un membre de l'équipe. Dans une version plus automatisée, chacun de ces déploiements peut être placé en fin de *pipeline*, lorsque tous les tests unitaires et d'intégration auront été réalisés.

Au travers de la commande `heroku create`, vous associez donc une nouvelle référence à votre code source, comme le montre le contenu du fichier `.git/config` ci-dessous:

```
$ heroku create
Creating app... done, ⬢ young-temple-86098
https://young-temple-86098.herokuapp.com/ | https://git.heroku.com/young-
temple-86098.git

$ cat .git/config
[core]
        repositoryformatversion = 0
        filemode = false
        bare = false
        logallrefupdates = true
        symlinks = false
        ignorecase = true
[remote "heroku"]
        url = https://git.heroku.com/still-thicket-66406.git
        fetch = +refs/heads/*:refs/remotes/heroku/*
```

IMPORTANT:

```
Pour définir de quel type d'application il s'agit, Heroku nécessite un minimum
de configuration.
Celle-ci se limite aux deux fichiers suivants:

* Déclarer un fichier `Procfile` qui va simplement décrire le fichier à passer
au protocole WSGI
* Déclarer un fichier `requirements.txt` (qui va éventuellement chercher ses
propres dépendances dans un sous-répertoire, avec l'option `-r`)
```

Après ce paramétrage, il suffit de pousser les changements vers ce nouveau dépôt grâce à la commande `git push heroku master`.

Heroku propose des espaces de déploiements, mais pas d'espace de stockage. Il est possible d'y envoyer des fichiers utilisateurs (typiquement, des media personnalisés), mais ceux-ci seront perdus lors du redémarrage du container. Il est donc primordial de configurer correctement l'hébergement des fichiers média, de préférences sur un stockage compatible S3.

Prêt à vous lancer ? Commencez par créer un compte: https://signup.heroku.com/python.

## 17.1. Configuration du compte Heroku

+ Récupération des valeurs d'environnement pour les réutiliser ci-dessous.

Vous aurez peut-être besoin d'un coup de pouce pour démarrer votre première application; heureusement, la documentation est super bien faite:

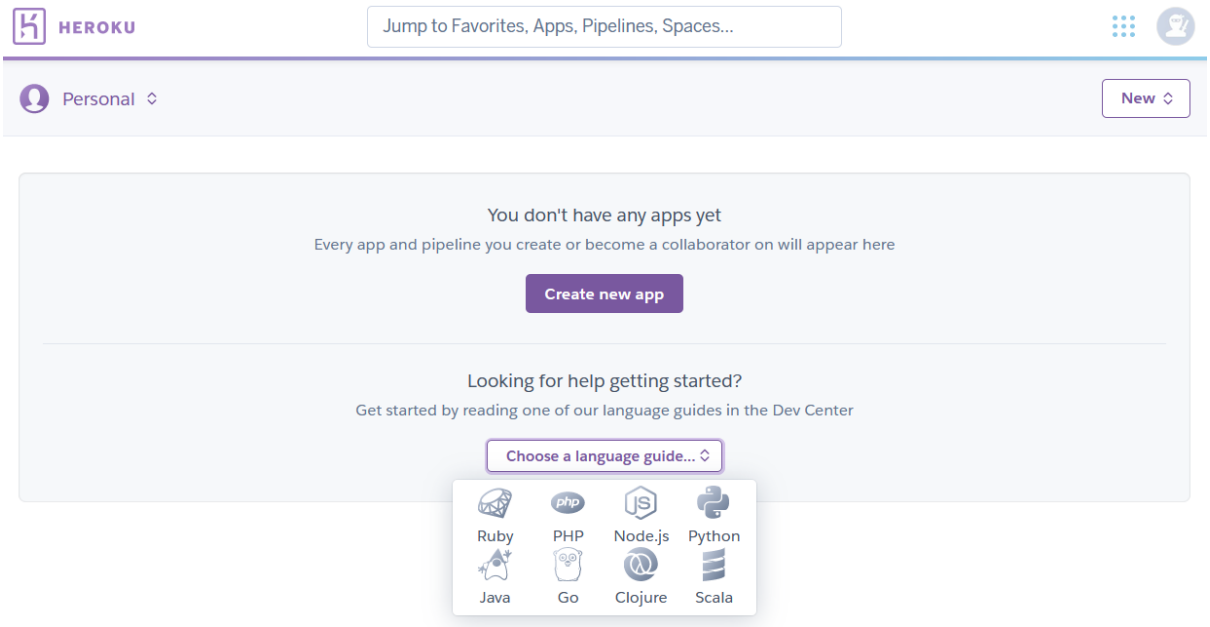



*Figure 18. Heroku: Commencer à travailler avec un langage*

Installez ensuite la CLI (*Command Line Interface*) en suivant la documentation suivante.

Au besoin, cette CLI existe pour:

1. macOS, *via* `brew`
2. Windows, grâce à un binaire x64 (la version 32 bits existe aussi, mais il est peu probable que vous en ayez besoin)
3. GNU/Linux, via un script Shell `curl https://cli-assets.heroku.com/install.sh | sh` ou

sur SnapCraft.

Une fois installée, connectez-vous:

```
$ heroku login
```

Et créer votre application:

```
$ heroku create
Creating app... done, ⬢ young-temple-86098
https://young-temple-86098.herokuapp.com/ | https://git.heroku.com/young-
temple-86098.git
```

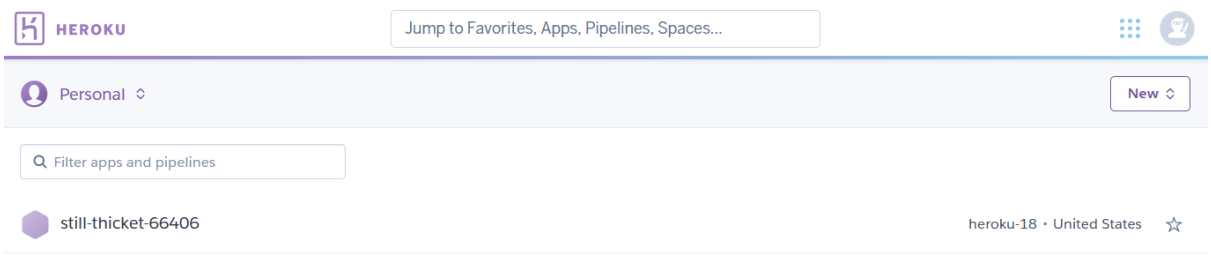



*Figure 19. Notre application est à présent configurée!*

Ajoutons lui une base de données, que nous sauvegarderons à intervalle régulier:

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on ⬢ still-thicket-66406... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-clear-39693 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation

$ heroku pg:backups schedule --at '14:00 Europe/Brussels' DATABASE_URL
Scheduling automatic daily backups of postgresql-clear-39693 at 14:00
Europe/Brussels... done
```

TODO: voir comment récupérer le backup de la db :-p

```
# Copié/collé de https://cookiecutter-
django.readthedocs.io/en/latest/deployment-on-heroku.html
heroku create --buildpack https://github.com/heroku/heroku-buildpack-python

heroku addons:create heroku-redis:hobby-dev

heroku addons:create mailgun:starter

heroku config:set PYTHONHASHSEED=random

heroku config:set WEB_CONCURRENCY=4

heroku config:set DJANGO_DEBUG=False
heroku config:set DJANGO_SETTINGS_MODULE=config.settings.production
heroku config:set DJANGO_SECRET_KEY="$(openssl rand -base64 64)"

# Generating a 32 character-long random string without any of the visually
similar characters "IOl01":
heroku config:set DJANGO_ADMIN_URL="$(openssl rand -base64 4096 | tr -dc 'A-
HJ-NP-Za-km-z2-9' | head -c 32)/"

# Set this to your Heroku app url, e.g. 'bionic-beaver-28392.herokuapp.com'
heroku config:set DJANGO_ALLOWED_HOSTS=

# Assign with AWS_ACCESS_KEY_ID
heroku config:set DJANGO_AWS_ACCESS_KEY_ID=

# Assign with AWS_SECRET_ACCESS_KEY
heroku config:set DJANGO_AWS_SECRET_ACCESS_KEY=

# Assign with AWS_STORAGE_BUCKET_NAME
heroku config:set DJANGO_AWS_STORAGE_BUCKET_NAME=

git push heroku master

heroku run python manage.py createsuperuser

heroku run python manage.py check --deploy

heroku open
```

## 17.2. Configuration

Pour qu'Heroku comprenne le type d'application à démarrer, ainsi que les commandes à exécuter pour que tout fonctionne correctement. Pour un projet Django, cela comprend, à

placer à la racine de votre projet:

1. Un fichier `requirements.txt` (qui peut éventuellement faire appel à un autre fichier, **via** l'argument `-r`)
2. Un fichier `Procfile` ([sans extension](https://devcenter.heroku.com/articles/procfile)!), qui expliquera la commande pour le protocole WSGI.

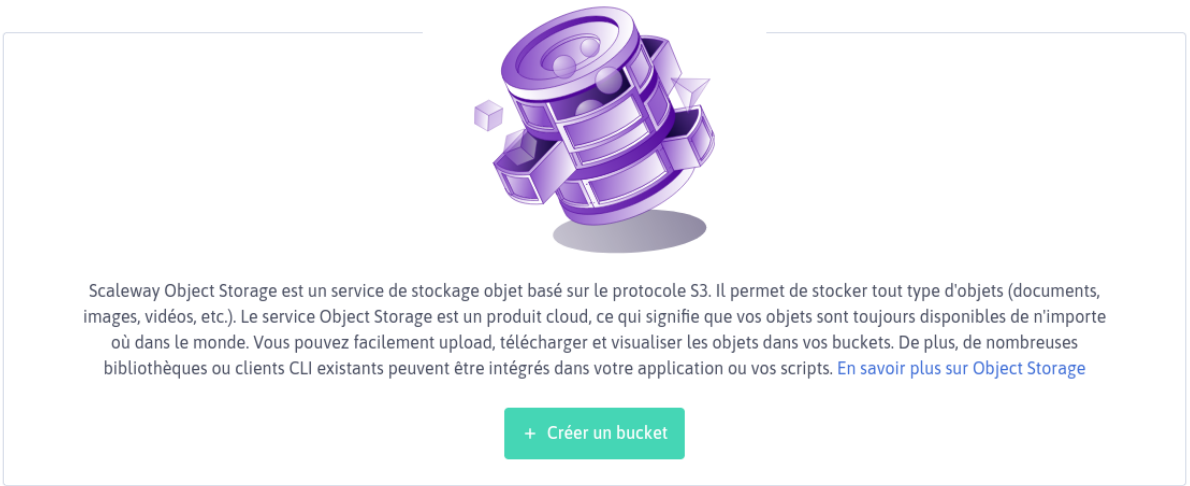
Dans notre exemple:

```
# requirements.txt
django==3.2.8
gunicorn
boto3
django-storages
```

```
# Procfile
release: python3 manage.py migrate
web: gunicorn gwift.wsgi
```

## 17.3. Hébergement S3

Pour cette partie, nous allons nous baser sur l'Object Storage de Scaleway. Ils offrent 75GB de stockage et de transfert par mois, ce qui va nous laisser suffisament d'espace pour jouer un peu .




L'idée est qu'au moment de la construction des fichiers statiques, Django aille simplement les héberger sur un espace de stockage compatible S3. La complexité va être de configurer correctement les différents points de terminaison. Pour héberger nos fichiers sur notre **bucket** S3, il va falloir suivre et appliquer quelques étapes dans l'ordre:

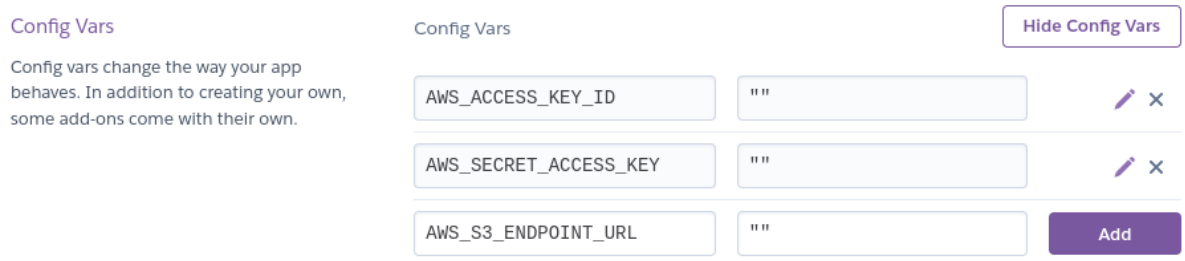
1. Configurer un bucket compatible S3 - je parlais de Scaleway, mais il y en a - **littéralement** - des dizaines.
2. Ajouter la librairie `boto3`, qui s'occupera de "parler" avec ce type de protocole
3. Ajouter la librairie `django-storage`, qui va elle s'occuper de faire le câblage entre le fournisseur (**via** `boto3`) et Django, qui s'attend à ce qu'on lui donne un moteur de gestion **via** la clé [`DJANGO_STATICFILES_STORAGE`](https://docs.djangoproject.com/en/3.2/ref/settings/#std:setting-STATICFILES_STORAGE).

La première étape consiste à se rendre dans [la console Scaleway](https://console.scaleway.com/project/credentials), pour gérer ses identifiants et créer un jeton.


Selon la documentation de django-storages, de boto3 et de Scaleway, vous aurez besoin des clés suivantes au niveau du fichier `settings.py`:

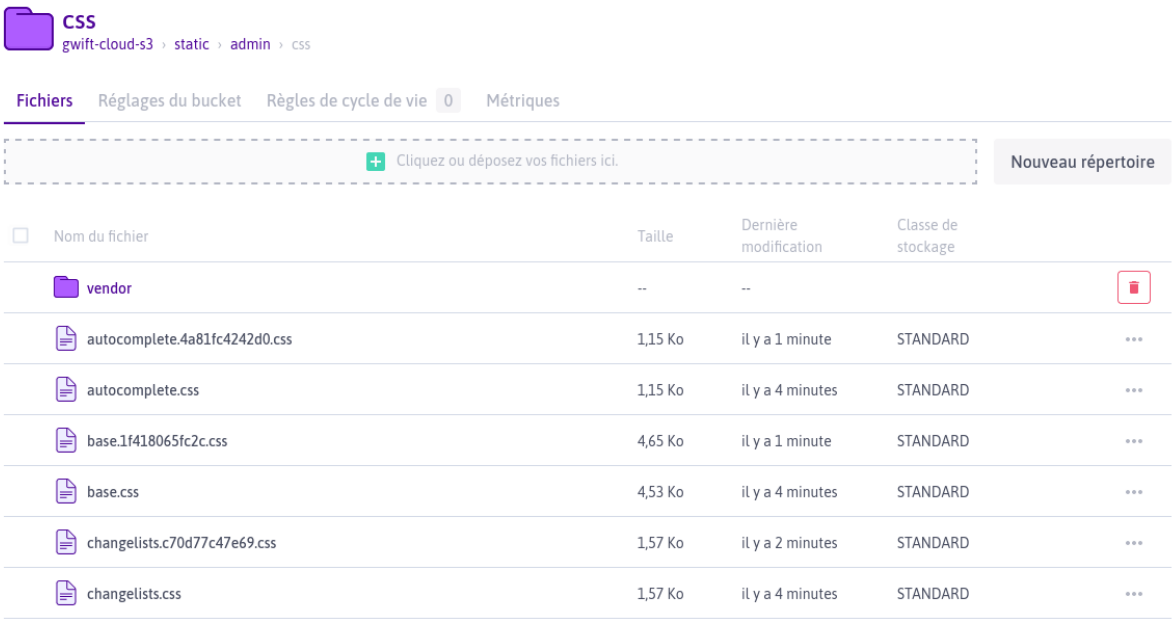
```
AWS_ACCESS_KEY_ID = os.getenv('ACCESS_KEY_ID')
AWS_SECRET_ACCESS_KEY = os.getenv('SECRET_ACCESS_KEY')
AWS_STORAGE_BUCKET_NAME = os.getenv('AWS_STORAGE_BUCKET_NAME')
AWS_S3_REGION_NAME = os.getenv('AWS_S3_REGION_NAME')

AWS_DEFAULT_ACL = 'public-read'
AWS_LOCATION = 'static'
AWS_S3_SIGNATURE_VERSION = 's3v4'

AWS_S3_HOST = 's3.%s.scw.cloud' % (AWS_S3_REGION_NAME,)
AWS_S3_ENDPOINT_URL = 'https://%s' % (AWS_S3_HOST, )

DEFAULT_FILE_STORAGE = 'storages.backends.s3boto3.S3Boto3Storage'
STATICFILES_STORAGE = 'storages.backends.s3boto3.S3ManifestStaticStorage'

STATIC_URL = '%s/%s/' % (AWS_S3_ENDPOINT_URL, AWS_LOCATION)

# General optimization for faster delivery
AWS_IS_GZIPPED = True
AWS_S3_OBJECT_PARAMETERS = {
    'CacheControl': 'max-age=86400',
}
```

Configurez-les dans la console d'administration d'Heroku:




Lors de la publication, vous devriez à présent avoir la sortie suivante, qui sera confirmée par le **bucket**:

```
remote: -----> $ python manage.py collectstatic --noinput
remote:        128 static files copied, 156 post-processed.
```




Sources complémentaires:

- [How to store Django static and media files on S3 in production](https://coderbook.com/@marcus/how-to-store-django-static-and-media-files-on-s3-in-production/)
- [Using Django and Boto3](https://www.simplecto.com/using-django-and-boto3-with-scaleway-object-storage/)

# 17.4. 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.

La partie ci-dessous a été reprise telle quelle de la documentation de cookie-cutter-django.

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.

# Chapitre 18. Autres outils

Voir aussi devpi, circus, uswgi, statsd.

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

# Chapitre 19. Ressources

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

# Services Oriented Applications

Nous avons fait exprès de reprendre l'acronyme d'une *Services Oriented Architecture* pour cette partie. L'objectif est de vous mettre la puce à l'oreille quant à la finalité du développement: que l'utilisateur soit humain, bot automatique ou client Web, l'objectif est de fournir des applications résilientes, disponibles et accessibles.

Dans cette partie, nous aborderons les vues, la mise en forme, la mise en page, la définition d'une interface REST, la définition d'une interface GraphQL et le routage d'URLs.

# Chapitre 20. 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`).

De manière très basique, une vue consiste en un objet Python dont le retour sera une instance de type `HttpResponse:

```
from django.http import HttpResponse

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

Techniquement, une vue pourrait retourner autre chose qu'une réponse Http, mais cela bloquera au niveau des middlewares, qui s'attendent tous à traiter un objet de ce type-là. Les APIs de type REST, SOAP ou GraphQL ne font finalement qu'une chose: encapsuler leur résultat dans un objet de type HttpResponse.

Le paramètre `request` est lui de type `HttpRequest` et embarque énormément d'informations, dont le schéma, les cookies, les verbes http,

détailler les informations de l'objet request :-p

Chaque vue peut etre représentée de deux manières:

1. Soit par des fonctions,
2. 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.

## 20.1. 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):
    wishlists = Wishlist.objects.all()
    return render(
        request,
        'wish/list.html',
        {
            'wishlists': wishlists
        }
    )
```

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! =)

## 20.2. 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 :-)

### 20.2.1. 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 21. 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)

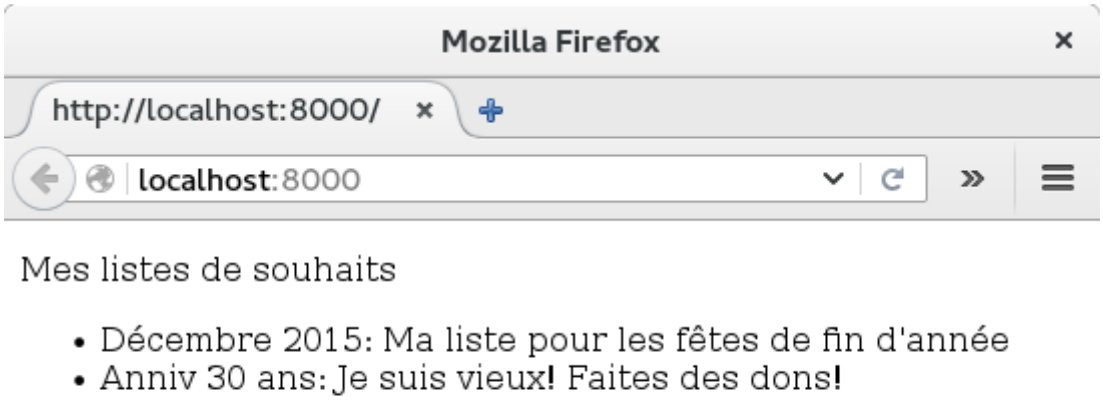
Un squelette de page HTML basique ressemble à ceci:

```
<!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 -->
{% load static %}<!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`.

# 21.1. Structure et configuration

## 21.1.1. Répertoires de découverte des templates

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

### 21.1.2. Fichiers statiques

(à compléter)

## 21.2. 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:

```python
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)
```

## 21.3. 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.

### 21.3.1. 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
```

### 21.3.2. 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)
```

### 21.3.3. 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() }
```

## 21.4. 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',
        ....
    ],
},
```

## 21.5. Querysets & managers

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

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, et fonctionnent comme un super jeu d'abstraction pour accéder à nos données (persistentes).

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

### 21.5.1. Gestionnaire de models (managers) et opérations

Chaque définition de modèle utilise un `Manager`, afin d'accéder à la base de données et traiter nos demandes. Indirectement, une instance de modèle ne **connait pas** la base de données: c'est son gestionnaire qui a cette tâche. Il existe deux exceptions à cette règle: les méthodes `save()` et `update()`.

- Instanciation: MyClass()
- Récupération: MyClass.objects.get(pk=...)

- Sauvegarde : MyClass().save()
- Création: MyClass.objects.create(...)
- Liste des enregistrements: MyClass.objects.all()

Par défaut, le gestionnaire est accessible au travers de la propriété `objects`. Cette propriété a une double utilité:

1. Elle est facile à surcharger - il nous suffit de définir une nouvelle classe héritant de ModelManager, puis de définir, au niveau de la classe, une nouvelle assignation à la propriété `objects`
2. Il est tout aussi facile de définir d'autres propriétés présentant des filtres bien spécifiques.

### 21.5.2. Requêtes

DANGER: Les requêtes sont sensibles à la casse, **même** si le moteur de base de données ne l'est pas. C'est notamment le cas pour Microsoft SQL Server; faire une recherche directement via les outils de Microsoft ne retournera pas obligatoirement les mêmes résultats que les managers, qui seront beaucoup plus tatillons sur la qualité des recherches par rapport aux filtres paramétrés en entrée.

### 21.5.3. Jointures

Pour appliquer une jointure sur un modèle, nous pouvons passer par les méthodes `select_related` et `prefetch_related`. Il faut cependant faire **très** attention au prefetch related, qui fonctionne en fait comme une grosse requête dans laquelle nous trouvons un `IN (⋯)`. Càd que Django va récupérer tous les objets demandés initialement par le queryset, pour ensuite prendre toutes les clés primaires, pour finalement faire une deuxième requête et récupérer les relations externes.

Au final, si votre premier queryset est relativement grand (nous parlons de 1000 à 2000 éléments, en fonction du moteur de base de données), la seconde requête va planter et vous obtiendrez une exception de type `django.db.utils.OperationalError: too many SQL variables`.

Nous pourrions penser qu'utiliser un itérateur permettrait de combiner les deux, mais ce n'est pas le cas...

Comme l'indique la documentation:

```
Note that if you use iterator() to run the query, prefetch_related() calls
will be ignored since these two optimizations do not make sense together.
```

Ajouter un itérateur va en fait forcer le code à parcourir chaque élément de la liste, pour

l'évaluer. Il y aura donc (à nouveau) autant de requêtes qu'il y a d'éléments, ce que nous cherchons à éviter.

```
informations = (
  <MyObject>.objects.filter(<my_criteria>)
  .select_related(<related_field>)
  .prefetch_related(<related_field>)
  .iterator(chunk_size=1000)
)
```

## 21.6. Aggregate vs. Annotate

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

# Chapitre 22. 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`.

Django fonctionne avec des **expressions rationnelles** simplifiées (des **expressions régulières** ou **regex**) pour trouver une correspondance entre une URL et la fonction qui recevra la requête et retournera une réponse. Nous utilisons l'expression `^$` pour déterminer la racine de notre application, mais nous pourrions appliquer d'autres regroupements (`/home`, `users/<profile_id>`, `articles/<year>/<month>/<day>`, ...). Chaque **variable** déclarée dans l'expression régulière sera apparenté à un paramètre dans la fonction correspondante. Ainsi,

```
# 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'),
]
```

Dans la mesure du possible, essayez toujours de **nommer** chaque expression. Cela permettra notamment de les retrouver au travers de la fonction `reverse`, mais permettra également de simplifier vos templates.

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 <code><a href="https://twitter.com/<user&gt" class="bare">https://twitter.com/<user&gt</a>;</code>. Sauf que les pages <code>about</code> et <code>help</code> 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 <code><a href="https://twitter.com/users/<user&gt" class="bare">https://twitter.com/users/<user&gt</a>;</code>.

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.

## 22.1. 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:

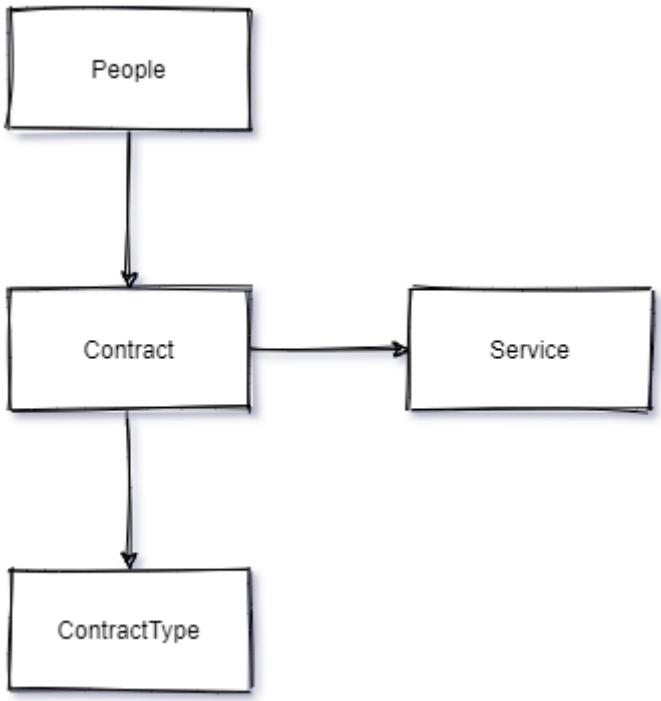
```
from django.core.urlresolvers import reverse_lazy

wishlists_url = reverse_lazy('wishlists')
```

# Chapitre 23. Application Programming Interface

Au niveau du modèle, nous allons partir de quelque chose de très simple: des personnes, des contrats, des types de contrats, et un service d'affectation. Quelque chose comme ceci:

```
# models.py

from django.db import models


class People(models.Model):
    CIVILITY_CHOICES = (
        ("M", "Monsieur"),
        ("Mme", "Madame"),
        ("Dr", "Docteur"),
        ("Pr", "Professeur"),
        ("", "")
    )

    last_name = models.CharField(max_length=255)
    first_name = models.CharField(max_length=255)
    civility = models.CharField(
        max_length=3,
        choices=CIVILITY_CHOICES,
        default=""
    )

    def __str__(self):
        return "{}, {}".format(self.last_name, self.first_name)


class Service(models.Model):
    label = models.CharField(max_length=255)

    def __str__(self):
        return self.label
```

```
class ContractType(models.Model):
    label = models.CharField(max_length=255)
    short_label = models.CharField(max_length=50)

    def __str__(self):
        return self.short_label


class Contract(models.Model):
    people = models.ForeignKey(People, on_delete=models.CASCADE)
    date_begin = models.DateField()
    date_end = models.DateField(blank=True, null=True)
    contract_type = models.ForeignKey(ContractType, on_delete=models.CASCADE)
    service = models.ForeignKey(Service, on_delete=models.CASCADE)

    def __str__(self):
        if self.date_end is not None:
            return "A partir du {}, jusqu'au {}, dans le service {} ({})"
.format(
                self.date_begin,
                self.date_end,
                self.service,
                self.contract_type
            )

        return "A partir du {}, à durée indéterminée, dans le service {}
({})".format(
                self.date_begin,
                self.service,
                self.contract_type
            )
```


People
Contract
Service
ContractType

# Chapitre 24. Configuration

La configuration des points de terminaison de notre API est relativement touffue. Il convient de:

1. Configurer les sérialiseurs, càd. les champs que nous souhaitons exposer au travers de l'API,
2. Configurer les vues, càd le comportement de chacun des points de terminaison,
3. Configurer les points de terminaison eux-mêmes, càd les URLs permettant d'accéder aux ressources.
4. Et finalement ajouter quelques paramètres au niveau de notre application.

## 24.1. Sérialiseurs

```
# serializers.py

from django.contrib.auth.models import User, Group
from rest_framework import serializers

from .models import People, Contract, Service


class PeopleSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = People
        fields = ("last_name", "first_name", "contract_set")


class ContractSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Contract
        fields = ("date_begin", "date_end", "service")


class ServiceSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Service
        fields = ("name",)
```

## 24.2. Vues

```
# views.py

from django.contrib.auth.models import User, Group
from rest_framework import viewsets
from rest_framework import permissions

from .models import People, Contract, Service
from .serializers import PeopleSerializer, ContractSerializer,
ServiceSerializer


class PeopleViewSet(viewsets.ModelViewSet):
    queryset = People.objects.all()
    serializer_class = PeopleSerializer
    permission_class = [permissions.IsAuthenticated]


class ContractViewSet(viewsets.ModelViewSet):
    queryset = Contract.objects.all()
    serializer_class = ContractSerializer
    permission_class = [permissions.IsAuthenticated]


class ServiceViewSet(viewsets.ModelViewSet):
    queryset = Service.objects.all()
    serializer_class = ServiceSerializer
    permission_class = [permissions.IsAuthenticated]
```

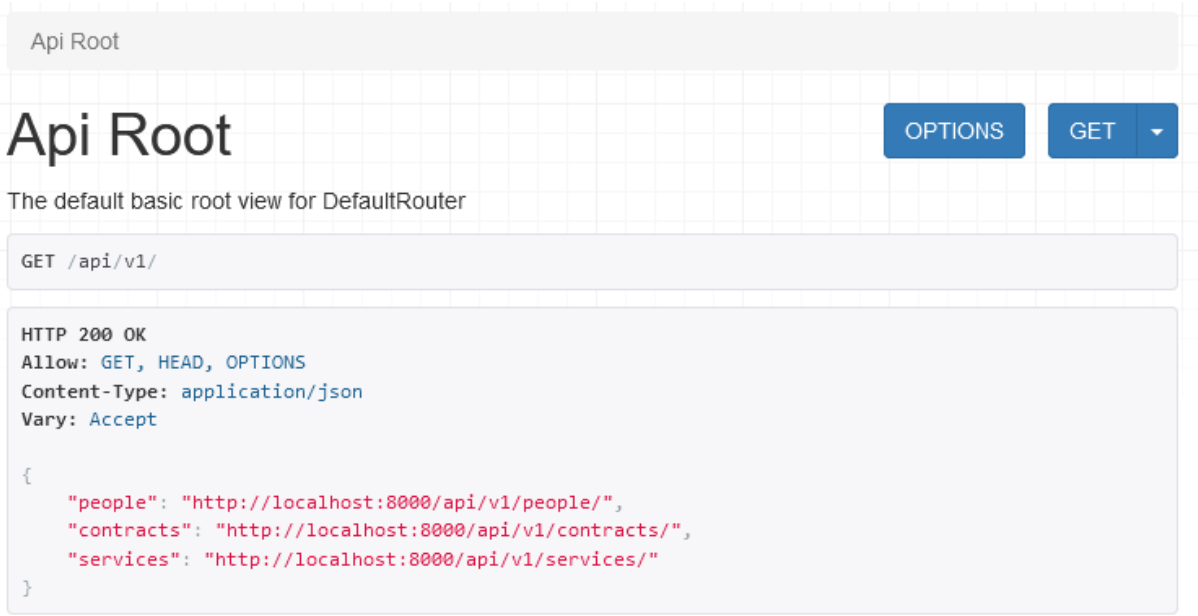
## 24.3. URLs

```
# urls.py

from django.contrib import admin
from django.urls import path, include

from rest_framework import routers

from core import views


router = routers.DefaultRouter()
router.register(r"people", views.PeopleViewSet)
router.register(r"contracts", views.ContractViewSet)
router.register(r"services", views.ServiceViewSet)

urlpatterns = [
    path("api/v1/", include(router.urls)),
    path('admin/', admin.site.urls),
]
```

## 24.4. Paramètres

```
# settings.py

INSTALLED_APPS = [
    ...
    "rest_framework",
    ...
]

...

REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

A ce stade, en nous rendant sur l'URL http://localhost:8000/api/v1, nous obtiendrons ceci:


Api Root
Api Root
OPTIONS
GET
The default basic root view for DefaultRouter
GET /api/v1/
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"people": "http://localhost:8000/api/v1/people/",
"contracts": "http://localhost:8000/api/v1/contracts/",
"services": "http://localhost:8000/api/v1/services/"
}

# Chapitre 25. Modèles et relations

Plus haut, nous avons utilisé une relation de type `HyperlinkedModelSerializer`. C'est une bonne manière pour autoriser des relations entre vos instances à partir de l'API, mais il faut reconnaître que cela reste assez limité. Pour palier à ceci, il existe [plusieurs manières de représenter ces relations](https://www.django-rest-framework.org/api-guide/relations/): soit **via** un hyperlien, comme ci-dessus, soit en utilisant les clés primaires, soit en utilisant l'URL canonique permettant d'accéder à la ressource. La solution la plus complète consiste à intégrer la relation directement au niveau des données sérialisées, ce qui nous permet de passer de ceci (au niveau des contrats):

```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "last_name": "Bond",
            "first_name": "James",
            "contract_set": [
                "http://localhost:8000/api/v1/contracts/1/",
                "http://localhost:8000/api/v1/contracts/2/"
            ]
        }
    ]
}
```

à ceci:

```
{
    "count": 1,
    "next": null,
    "previous": null,
    "results": [
        {
            "last_name": "Bond",
            "first_name": "James",
            "contract_set": [
                {
                    "date_begin": "2019-01-01",
                    "date_end": null,
                    "service": "http://localhost:8000/api/v1/services/1/"
                },
                {
                    "date_begin": "2009-01-01",
                    "date_end": "2021-01-01",
                    "service": "http://localhost:8000/api/v1/services/1/"
                }
            ]
        }
    ]
}
```

La modification se limite à **surcharger** la propriété, pour indiquer qu'elle consiste en une instance d'un des sérialiseurs existants. Nous passons ainsi de ceci

```
class ContractSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Contract
        fields = ("date_begin", "date_end", "service")


class PeopleSerializer(serializers.HyperlinkedModelSerializer):

    class Meta:
        model = People
        fields = ("last_name", "first_name", "contract_set")
```

à ceci:

```
class ContractSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = Contract
        fields = ("date_begin", "date_end", "service")


class PeopleSerializer(serializers.HyperlinkedModelSerializer):
    contract_set = ContractSerializer(many=True, read_only=True)

    class Meta:
        model = People
        fields = ("last_name", "first_name", "contract_set")
```

Nous ne faisons donc bien que redéfinir la propriété `contract_set` et indiquons qu'il s'agit à présent d'une instance de `ContractSerializer`, et qu'il est possible d'en avoir plusieurs. C'est tout.

# Chapitre 26. Filtres et recherches

A ce stade, nous pouvons juste récupérer des informations présentes dans notre base de données, mais à part les parcourir, il est difficile d'en faire quelque chose.

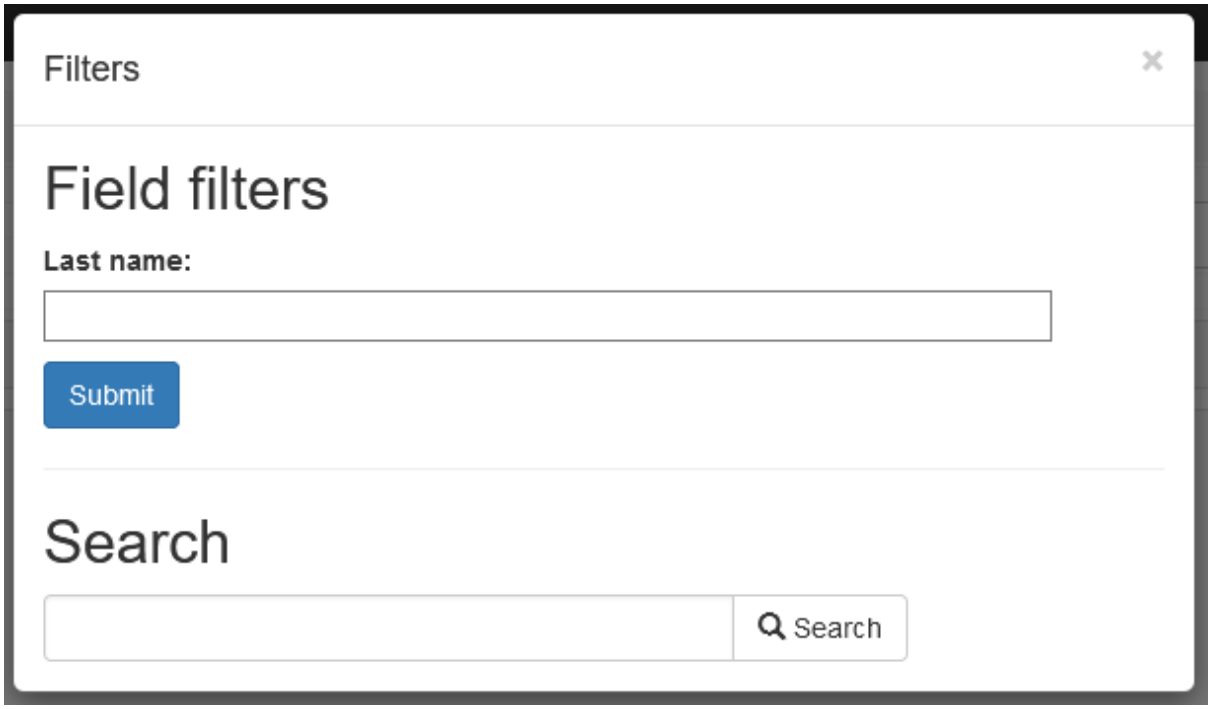
Il est possible de jouer avec les URLs en définissant une nouvelle route ou avec les paramètres de l'URL, ce qui demanderait alors de programmer chaque cas possible - sans que le consommateur ne puisse les déduire lui-même. Une solution élégante consiste à autoriser le consommateur à filtrer les données, directement au niveau de l'API. Ceci peut être fait. Il existe deux manières de restreindre l'ensemble des résultats retournés:

1. Soit au travers d'une recherche, qui permet d'effectuer une recherche textuelle, globale et par ensemble à un ensemble de champs,
2. Soit au travers d'un filtre, ce qui permet de spécifier une valeur précise à rechercher.

Dans notre exemple, la première possibilité sera utile pour rechercher une personne répondant à un ensemble de critères. Typiquement, `/api/v1/people/?search=raymond bond` ne nous donnera aucun résultat, alors que `/api/v1/people/?search=james bond` nous donnera le célèbre agent secret (qui a bien entendu un contrat chez nous...).

Le second cas permettra par contre de préciser que nous souhaitons disposer de toutes les personnes dont le contrat est ultérieur à une date particulière.

Utiliser ces deux mécanismes permet, pour Django-Rest-Framework, de proposer immédiatement les champs, et donc d'informer le consommateur des possibilités:




## 26.1. Recherches

La fonction de recherche est déjà implémentée au niveau de Django-Rest-Framework, et aucune dépendance supplémentaire n'est nécessaire. Au niveau du `viewset`, il suffit d'ajouter deux informations:

```
...
from rest_framework import filters, viewsets
...

class PeopleViewSet(viewsets.ModelViewSet):
    ...
    filter_backends = [filters.SearchFilter]
    search_fields = ["last_name", "first_name"]
    ...
```

## 26.2. Filtres

Nous commençons par installer [le paquet `django-filter`](https://www.django-rest-framework.org/api-guide/filtering/#djangofilterbackend) et nous l'ajoutons parmi les applications installées:

```
λ pip install django-filter
Collecting django-filter
  Downloading django_filter-2.4.0-py3-none-any.whl (73 kB)
     |████████████████████████████████| 73 kB 2.6 MB/s
Requirement already satisfied: Django>=2.2 in c:\users\fred\sources\.venvs\r
ps\lib\site-packages (from django-filter) (3.1.7)
Requirement already satisfied: asgiref<4,>=3.2.10 in c:\users\fred\sources
\.venvs\rps\lib\site-packages (from Django>=2.2->django-filter) (3.3.1)
Requirement already satisfied: sqlparse>=0.2.2 in c:\users\fred\sources\.
venvs\rps\lib\site-packages (from Django>=2.2->django-filter) (0.4.1)
Requirement already satisfied: pytz in c:\users\fred\sources\.venvs\rps\lib
\site-packages (from Django>=2.2->django-filter) (2021.1)
Installing collected packages: django-filter
Successfully installed django-filter-2.4.0
```

Une fois l'installée réalisée, il reste deux choses à faire:

1. Ajouter `django_filters` parmi les applications installées:
2. Configurer la clé `DEFAULT_FILTER_BACKENDS` à la valeur `['django_filters.rest_framework.DjangoFilterBackend']`.

Vous avez suivi les étapes ci-dessus, il suffit d'adapter le fichier `settings.py` de la manière suivante:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10,
    'DEFAULT_FILTER_BACKENDS':
['django_filters.rest_framework.DjangoFilterBackend']
}
```

Au niveau du viewset, il convient d'ajouter ceci:

```
...
from django_filters.rest_framework import DjangoFilterBackend
from rest_framework import viewsets
...

class PeopleViewSet(viewsets.ModelViewSet):
    ...
    filter_backends = [DjangoFilterBackend]
    filterset_fields = ('last_name',)
    ...
```

A ce stade, nous avons deux problèmes:

1. Le champ que nous avons défini au niveau de la propriété `filterset_fields` exige une correspondance exacte. Ainsi, `/api/v1/people/?last_name=Bon` ne retourne rien, alors que `/api/v1/people/?last_name=Bond` nous donnera notre agent secret préféré.
2. Il n'est pas possible d'aller appliquer un critère de sélection sur la propriété d'une relation. Notre exemple proposant rechercher uniquement les relations dans le futur (ou dans le passé) tombe à l'eau.

Pour ces deux points, nous allons définir un nouveau filtre, en surchargeant une nouvelle classe dont la classe mère serait de type `django_filters.FilterSet`.

TO BE CONTINUED.

A noter qu'il existe un paquet [Django-Rest-Framework-filters](https://github.com/philipn/django-rest-framework-filters), mais il est déprécié depuis Django 3.0, puisqu'il se base sur `django.utils.six` qui n'existe à présent plus. Il faut donc le faire à la main (ou patcher le paquet...).

## 26.3. Arborescences

```
```

```
# <app>/management/commands/rebuild.py

"""This command manages Closure Tables implementation

It adds new levels and cleans links between entities.
This way, it's relatively easy to fetch an entire tree with just one tiny
request.

"""

from django.core.management.base import BaseCommand

from rps.structure.models import Entity, EntityTreePath


class Command(BaseCommand):
    def handle(self, *args, **options):
        entities = Entity.objects.all()

        for entity in entities:
            breadcrumb = [node for node in entity.breadcrumb()]

            tree = set(EntityTreePath.objects.filter(descendant=entity))

            for idx, node in enumerate(breadcrumb):
                tree_path, _ = EntityTreePath.objects.get_or_create(
                    ancestor=node, descendant=entity, weight=idx + 1
                )

                if tree_path in tree:
                    tree.remove(tree_path)

            for tree_path in tree:
                tree_path.delete()
```

# Chapitre 27. Conclusions

De part son pattern `MVT`, Django ne fait pas comme les autres frameworks.

# 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 28. Gwift

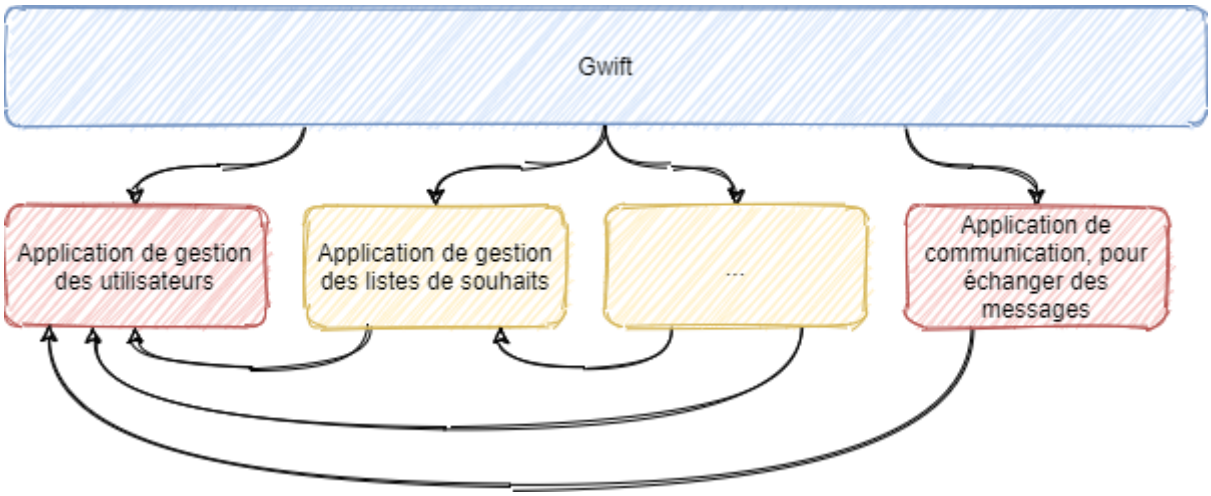



*Figure 20. Gwift*

# Chapitre 29. 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 30. Besoins fonctionnels

## 30.1. 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.

## 30.2. Gestion des listes

### 30.2.1. 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

### 30.2.2. 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

## 30.3. Gestion des souhaits

### 30.3.1. 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.

### 30.3.2. 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.

# 30.4. Gestion des réalisations de souhaits

## 30.4.1. 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

## 30.4.2. 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.

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

## 30.5.1. Modélisation

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

- un identifiant
- un nom
- une adresse email facultative

## 30.5.2. 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 31. Tests unitaires

## 31.1. 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.

## 31.2. 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.

## 31.3. What are you testing?

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

## 31.4. 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.

## 31.5. 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.

## 31.6. Quelques liens utiles

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

# Chapitre 32. 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.

## 32.1. 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;
```

## 32.2. 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.

## 32.3. 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`.

# Chapitre 33. Khana

Khana est une application de suivi d'apprentissage pour des élèves ou étudiants. Nous voulons pouvoir:

1. Lister les élèves
2. Faire des listes de présence pour les élèves
3. Pouvoir planifier ses cours
4. Pouvoir suivre l'apprentissage des élèves, les liens qu'ils ont entre les éléments à apprendre:
5. pour écrire une phrase, il faut pouvoir écrire des mots, connaître la grammaire, et connaître la conjugaison
6. pour écrire des mots, il faut savoir écrire des lettres
7. ...

Plusieurs professeurs s'occupent d'une même classe; il faut pouvoir écrire des notes, envoyer des messages aux autres professeurs, etc.

Il faut également pouvoir définir des dates de contrôle, voir combien de semaines il reste pour s'assurer d'avoir vu toute la matièrte.

Et pouvoir encoder les points des contrôles.

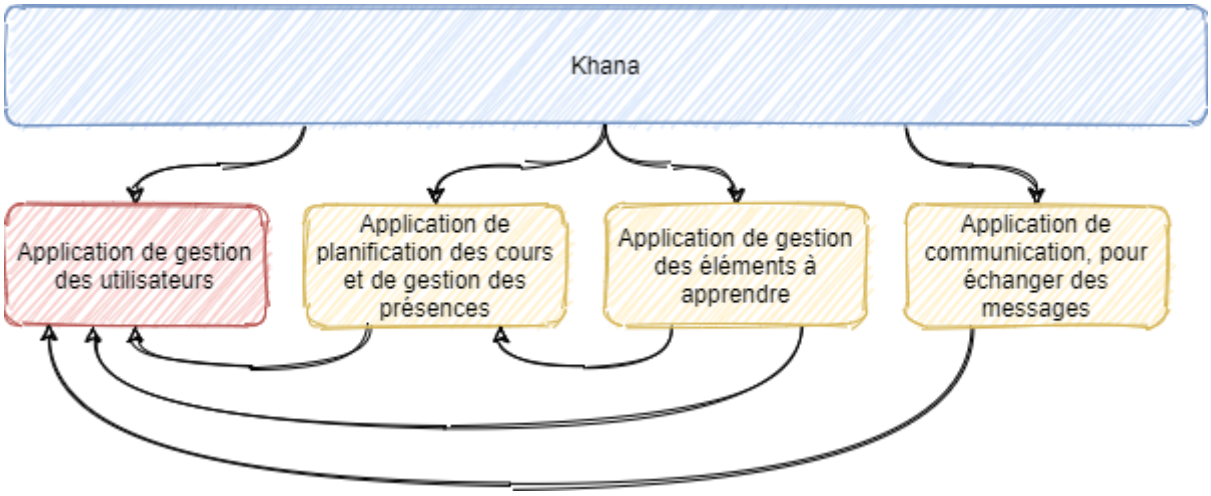



*Figure 21. Khana*

Unresolved directive in part-5-go-live/*index.adoc* - *include::legacy/_index*.adoc[]

# Ressources et bibliographie

Unresolved directive in part-9-resources/_index.adoc - include::code-snippets.adoc[]

# Chapitre 34. Applications *Legacy*

Quand on intègre une nouvelle application Django dans un environement existant, la première étape est de se câbler sur la base de données existantes;

1. Soit l'application sur laquelle on se greffe restera telle quelle;
2. Soit l'application est **remplacée** par la nouvelle application Django.

```
Dans le premier cas, il convient de créer une application et de spécifier
pour chaque classe l'attribute `managed = False` dans le `class Meta:` de
la définition.
Dans le second, il va falloir câbler deux-trois éléments avant d'avoir une
intégration complète (comprendre: avec une interface d'admin, les
migrations, les tests unitaires et tout le brol :))
```

```
`python manage.py inspectdb > models.py`
```

# Glossaire

**http**

*HyperText Transfer Protocol*, ou plus généralement le protocole utilisé (et détourné) pour tout ce qui touche au **World Wide Web**. Il existe beaucoup d'autres protocoles d'échange de données, comme Gopher, FTP ou SMTP.

**IaaS**

*Infrastructure as a Service*, où un tiers vous fournit des machines (généralement virtuelles) que vous devrez ensuite gérer en bon père de famille. L'IaaS propose souvent une API, qui vous permet d'intégrer la durée de vie de chaque machine dans vos flux - en créant, augmentant, détruisant une machine lorsque cela s'avère nécessaire.

**MVC**

Le modèle *Model-View-Controler* est un patron de conception autorisant un faible couplage entre la gestion des données (le *Modèle*), l'affichage et le traitement de celles (la *Vue*) et la glue entre ces deux composants (au travers du *Contrôleur*). Wikipédia

**ORM**

*Object Relational Mapper*, où une instance est directement (ou à proximité) liée à un mode de persistance de données.

**PaaS**

*Platform as a Service*, qui consiste à proposer les composants d'une plateforme (Redis, PostgreSQL, ...) en libre service et disponibles à la demande (quoiqu'après avoir communiqué son numéro de carte de crédit...).

**POO**

La *Programmation Orientée Objet* est un paradigme de programmation informatique. Elle consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. Il possède une structure interne et un comportement, et il sait interagir avec ses pairs. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Dès lors, l'étape de modélisation revêt une importance majeure et nécessaire pour la POO. C'est elle qui permet de transcrire les éléments du réel sous forme virtuelle. Wikipédia

### S3

Amazon *Simple Storage Service* consiste en un système d'hébergement de fichiers, quels qu'ils soient. Il peut s'agir de fichiers de logs, de données applications, de fichiers média envoyés par vos utilisateurs, de vidéos et images ou de données de sauvegardes.

*https://aws.amazon.com/fr/s3/*

# Index

# Chapitre 35. Bibliographie

[1] “The web framework for perfectionists with deadlines.” [Online]. Available: https://www.djangoproject.com/.

[2] “Design Philosophies.” [Online]. Available: https://docs.djangoproject.com/en/dev/misc/design-philosophies/.

[3] “Les cinq règles pour écrire du code maintenable.” 2019, [Online]. Available: https://boutique.ed-diamond.com/les-hors-series/1402-gnulinux-magazine-hs-104.html.

[4] “Consider SQLite.” 2021, [Online]. Available: https://blog.wesleyac.com/posts/consider-sqlite.

[5] M. Jaworski and T. Ziadé, *Expert Python Programming*, 4Th edition. Packt Publishing, 2021.

[6] G. Kim, J. Humble, P. Debois, and J. Willis, *The DevOps Handbook, How to create World-class Agility, Reliability, & Security in Technology Organizations*. IT Revolution, 2016.

[7] R. C. Martin, *Clean Code, a Handbook of Agile Software Craftmanship*. Pearson, 2009.

[8] R. C. Martin, *Clean Architecture, A Craftman’s Guide to Software Structure and Design*. Pearson, 2018.

[9] E. S. Raymond, *Basics of the Unix Philosophy, The Art of Unix Programming*. Addison-Wesley Professional, 2004.

[10] A. Sweigart, *Automate the Boring Stuff with Python*, 2Nd edition. William Pollock, 2020.

[11] J. Visser, *Building Maintainable Software*, C# Edition, first edition. O’Reilly Media, Inc., 2016.