Architecture de Sith: le nouveau site AE

Skia (Florent JACQUET)

Dernière version: 11 août 2019

Table des matières

1 Introduction

Il y a longtemps, au début des années 2000, l'Association des Étudiants a mis en place un site internet qui n'a eu de cesse d'évoluer au fil des ans. Grâce aux différents contributeurs qui s'y sont plongés, et qui ont pu y ajouter leurs bouts de code plus ou moins utiles, le site possède désormais un ensemble de fonctionnalités impressionnant.

De la comptabilité à la gestion de la laverie, en passant par le forum ou le Matmatronch', le site de l'AE prend actuellement en charge la quasi totalité de la gestion de l'argent, et c'est là un de ses rôles les plus importants.

Mais les vieilles technologies qu'il emploie, et l'entretien plus ou moins aléatoire qu'il a subit, en font un outil très difficile à maintenir à l'heure actuelle, et le besoin d'une refonte s'imposait de plus en plus.

Le choix de technologies récentes, maintenues, et éprouvées a donc été fait, et le développement a pu commencer dès Novembre 2015, avec l'objectif d'une mise en production dans l'été 2016, au moins dans une version incluant l'intégralité des fonctions liées à l'argent, qui sont les plus critiques.

Soutenant les projets libres, j'ai décidé de placer le projet sous licence MIT, assurant ainsi une pérénité aux source. Si quelqu'un dans le futur souhaite le relicencier sous GPL (ou autre licence plus restrictive que la MIT, voire contagieuse), cela reste possible, mais je n'impose au départ que très peu de restrictions [1] .

1. La seule condition en réalité, est de toujours garder à sa place une copie de la licence originale, à savoir le fichier LICENSE à la racine du site.

2 Les technologies

C'est là un des choix les plus important lors d'un tel projet, puisqu'il se fait au début, et qu'il n'est ensuite plus possible de revenir en arrière. Le PHP vieillissant, et piègeux [1] a donc été mis de côté au profit d'un language plus stable, le *Python* dans sa version 3.

2.1 Python3

Le site étant développé en *Python*, il est impératif d'avoir un environnement de développement approprié à ce language. L'outil `virtualenv` permet d'installer un environnement *Python* de manière locale, sans avoir besoin des droits root pour installer des packages. De plus cela permet d'avoir sur sa machine plusieurs environnements différents, adaptés à chaque projet, avec chacun des versions différentes des même paquets.

La procédure pour installer son `virtualenv` est décrite dans le fichier `README` situé à la racine du projet.

2.2 Django

Django est un framework web pour *Python*, apparu en 2005, et fournissant un grand nombre de fonctionnalités pour développer un site rapidement et simplement. Cela inclut entre autre un serveur Web, pour les échanges HTTP, un parseur d'URL, pour le routage des différentes URI du site, un ORM [2] pour la gestion de la base de donnée, ou encore un moteur de templates, pour les rendus HTML.

La version 1.8 de *Django* a été choisie pour le développement de ce projet, car c'est une version LTS (Long Term Support), c'est à dire qu'elle restera stable et maintenue plus longtemps que les autres (au moins jusqu'en Avril 2018).

La documentation est disponible à cette addresse : `https://docs.djangoproject.com/en/1.8/`. Bien que ce rapport présente dans les grandes lignes le fonction-

1. `https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/`
2. Object-Relational Mapper

nement de *Django*, il n'est pas et ne se veut pas exhaustif, et la documentation restera donc toujours la référence à ce sujet.

2.2.1 Le fichier de management et l'organisation d'un projet

Lors de la création d'un projet *Django*, plusieurs fichiers sont créés. Ces fichiers sont essentiels pour le projet, mais ne contiennent en général pas de code à proprement parler. Ce n'est pas là qu'on y développe quoi que ce soit.

manage.py

Le fichier `manage.py`, situé à la racine, permet de lancer toutes les tâches d'administration du site. Parmis elles :

- **startapp**
  Créer une application.
- **makemigrations**
  Parser les modèles pour créer les fichiers de migration de la base de donnée.
- **migrate**
  Appliquer les migrations sur la base de données.
- **runserver**
  Pour lancer le serveur Web, et donc le site en lui même, dans une version de développement.
- **makemessages**
  Pour générer les fichiers de traduction, dans le dossier `locale`.
- **compilemessages**
  Pour compiler les fichiers de traduction, dans le dossier `locale`, et passer de `django.po` à `django.mo`, sa version binaire, optimisée pour l'utilisation.

Un premier dossier

Un premier dossier est toujours créé, du nom du projet, et contenant plusieurs fichiers : `settings.py`, `urls.py`, et `wsgi.py`.

`settings.py` est un fichier *Python* servant à définir un grand nombre de constantes paramètrant le fonctionnement du site. L'avantage par rapport à un fichier de configuration classique est que ce dernier est executable, et on peut donc y mettre de la logique, afin d'avoir des paramètres dynamiques.

`urls.py` est le fichier principale contenant les routes du site, c'est à dire les URLs existantes. Il se charge en général d'inclure les fichiers `urls.py` de chaque application afin de garder une architecture modulaire et simple.

`wsgi.py` contient quant à lui les paramètres pour la mise en production du site en tant qu'application WSGI (Web Server Gateway Interface) pour tourner derrière un serveur Web.

2.2.2 Organisation d'une application

Lorsque l'on créer une application avec `./manage.py startapp`, on obtient une fois de plus un dossier type. On trouve dans celui-ci un certain nombre de

fichiers :

— **__init__.py**
Permet de définir le dossier comme un package *Python*. Ce fichier est généralement vide.

— **models.py**
C'est là que l'on définit tous les modèles, c'est à dire toutes les classes qui définissent des tables dans la base de donnée.

— **views.py**
Les vues y sont définies.

— **admin.py**
C'est là que l'on déclare quels modèles doivent apparaîtrent dans l'interface fournie par le module d'administration.

— **tests.py**
Ce dernier fichier sert à écrire les tests fonctionnels, unitaires, ou d'intégation à l'aide de la librairie de test de *Django*.

— **migrations**
Ce dossier sert à stocker les fichiers de migration de la base de donnée générés par `./manage.py makemigrations`.

On rajoute par la suite généralement plusieurs fichiers :

— **urls.py**
Pour y définir toutes les URLs de l'application, et ensuite inclure ce fichier dans le fichier `urls.py` global au projet.

— **templates**
Celui-ci est un dossier, et on y remet en général un sous dossier du nom de l'application afin de s'en servir de namespace pour les templates.

Dans le cas où un fichier *Python* deviendrait trop gros ou trop complexe, il est toujours possible de le diviser en plusieurs fichiers que l'on met dans un dossier du même nom que ce fichier de départ, et contenant en plus un fichier `__init__.py`. De plus, pour faciliter les imports depuis ce dossier, on peut mettre dans `__init__.py` la ligne [3] :

```
from .[nom_de_fichier_sans_le_.py] import *
```

2.2.3 Les modèles avec l'ORM

Le modèle en lui même

Rien ne vaudra un bon exemple pour comprendre comment sont construits les modèles avec *Django* :

```
class Club(models.Model): # (1)
    """
    The Club class, made as a tree to allow nice tidy organization
    """ # (2)
    name = models.CharField(_('name'), max_length=30) # (3)
    parent = models.ForeignKey('Club', related_name='children', null=True, blank=True) # (4)
    unix_name = models.CharField(_('unix name'), max_length=30, unique=True,
            validators=[ # (5)
```

3. Un exemple est disponible dans l'application core

```
            validators.RegexValidator(
                r'^[a-z0-9][a-z0-9._-]*[a-z0-9]$',
                _('Enter a valid unix name. This value may contain only '
                    'letters, numbers ./-/_ characters.')
                ),
            ],
        error_messages={ # (6)
            'unique': _("A club with that unix name already exists."),
            },
        )
address = models.CharField(_('address'), max_length=254)
email = models.EmailField(_('email address'), unique=True)
owner_group = models.ForeignKey(Group, related_name="owned_club",
                                default=settings.SITH_GROUP_ROOT_ID) # (7)
edit_groups = models.ManyToManyField(Group, related_name="editable_club", blank=True) # (8)
view_groups = models.ManyToManyField(Group, related_name="viewable_club", blank=True)
home = models.OneToOneField(SithFile, related_name='home_of_club', verbose_name=_("home"), null=True
        on_delete=models.SET_NULL) # (9)
```

Explications :

(1) Un modèle hérite toujours de `models.Model`. Il peut y avoir des intermédiaires, mais `Model` sera toujours en haut.

(2) Toujours penser à commenter le modèle.

(3) Un premier attribut : `name`, de type `CharField`. Il constitue une colonne dans la base de donnée une fois que `./manage.py migrate` a été appliqué.

(4) Une `ForeignKey`, l'une des relations les plus utilisées. `related_name` précise le nom qui sert de retour vers cette classe depuis la classe pointée. Ici, elle est même récursive, puisque l'on pointe vers la classe que l'on est en train de définir, ce qui donne au final une structure d'arbre.

(5) On peut toujours préciser des `validators`, afin que le modèle soit contraint, et que *Django* maintienne toujours des informations cohérentes dans la base.

(6) Un message d'erreur peut être précisé pour expliciter à l'utilisateur les problèmes rencontrés.

(7) On utilise ici le champ `default` pour préciser une valeur par défaut au modèle, et celui-ci est affecté à une valeur contenue dans les `settings` de *Django*.

(8) Les `ManyToManyField` permettent de générer automatiquement une table intermédiaire de manière transparente afin d'avoir des relations doubles dans les deux classes mises en jeu.

(9) Le `OneToOneField` est très utilisé pour étendre une table avec des informations supplémentaires sans toucher à la table originale.

PRIMARY KEY Les plus observateurs d'entre vous auront remarqué qu'il n'y a pas ici de `PRIMARY KEY` de précisé. En effet, *Django* s'en occupe automatiquement en rajoutant un champ `id` jouant ce rôle. On peut alors y accèder en l'appelant par son nom, `id` la plupart du temps, sauf s'il a été personnalisé, ou bien par l'attribut générique `pk`, toujours présent pour désigner la `PRIMARY KEY` d'un modèle, quelle qu'elle soit.

Les migrations

Les migrations sont à lancer à chaque fois que l'on modifie un modèle. Elles permettent de conserver la base de donnée tout en la faisant évoluer dans sa structure, pour ajouter ou supprimer une colonne dans une table par exemple.

Lancer la commande `./manage.py makemigrations [nom de l'appli]` va permettre de générer un fichier *Python* automatiquement, qui sera mis à la suite des précédents, et qui sera appliqué sur la base au moment du lancement de `./manage.py migrate`.

2.2.4 Les vues

Les vues sont les parties de code s'occupant de l'interface avec l'utilisateur. Elles sont appelées par les URLs, et renvoient des réponses HTTP en fonction du traitement effectué.

Les URL

Les URLs sont définies par application, et centralisées dans le dossier du projet. Il s'agit à chaque fois d'une liste d'appel à la fonction `url()`, qui comprends toujours une expression rationnelle décrivant l'URL, une fonction passée en tant que callback qui sera appelé au moment où l'URL est résolue, et enfin un nom, permettant de s'y référer dans les fonctions de résolution inverse, comme dans les templates par exemple. Nous détaillerons cette utilisation plus tard.

Pour garder une organisation claire, les URLs sont classées par espaces de noms (namespace) afin d'avoir à éviter de préfixer tous les noms pour s'y retrouver. Le namespace d'une URL est généralement le même nom que celui de l'application dans laquelle elle se trouve.

Les fonctions de vue

Une fonction de vue prend toujours en paramètre une variable `request` et renvoie toujours un objet `HTTPResponse`, contenant un code de retour HTTP, ainsi qu'une chaîne de caractères contenant la réponse en elle même.

Entre temps, le traitement des informations permet de mettre à jour, de créer, ou de supprimer les objets définis dans les modèles, par le biais des paramètres passé dans la requête. Ainsi, on peut accèder aux informations des variables `GET` et `POST` très facilement en appelant respectivement `request.GET['ma_clef']` et `request.POST['ma_clef']`, ces deux variables fonctionnant comme des dictionnaires.

Des vues basées sur des classes

Les vues avec *Django* peuvent aussi être définies comme des classes. Elles héritent alors à ce moment là toutes de la classe `View`, mais ont toutefois souvent beaucoup d'intermédiaires et n'héritent donc pas directement de cette dernière.

L'avantage de ces vues sous forme de classe est de pouvoir séparer toute la chaîne de traitement entre les différentes méthodes, et ainsi permettre, en jouant avec l'héritage, de fournir alors très peu d'informations à la classe, tout en lui permettant d'effectuer un travail correct.

Ainsi, on retrouve de base, dans les filles de `View`, un grand nombre de classes prédéfinies pour la plupart des comportement. `DetailView`, `CreateView`, `ListView`, sont quelques exemples de classes renvoyant respectivement un objet en détails, un formulaire pour créer un nouvel objet, et enfin une liste d'objets. Il existe cependant un bon nombre de ces vues fournissant d'autres fonctionnalités, et si malgré tout, aucune ne peut convenir, il reste possible de se baser sur l'une d'elle et surcharger l'une de ses fonctions pour l'adapter à ses besoins.

L'écosystème des `class-based views` étant toutefois assez complexe et riche, un site web a été créé afin d'offrir un bon résumé de la situation : *Classy class-based views*, accessible à l'adresse `http://ccbv.co.uk/`.

2.3 Jinja2

Jinja2 est un moteur de template écrit en *Python* qui s'inspire fortement de la syntaxe des templates de *Django*, mais qui apporte toutefois son lot d'améliorations non négligeables. En effet, l'ajout des macros, par exemple, permet de factoriser une grande partie du code.

Un moteur de template permet de générer du contenu textuel de manière procédural en fonction des données à afficher. Cela permet en pratique de pouvoir inclure du code proche de *Python* dans la syntaxe au milieu d'un document contenant principalement du HTML. Ainsi, si on a une liste d'objets, on peut facilement executer une boucle `for` afin de faire afficher simplement tous les objets selon le même format. De même, il est facile d'inclure un `if` pour décider à l'execution d'afficher ou non un lien en fonction des droits que l'utilisateur possède sur le site, par exemple.

En plus des structures conditionnelles classiques, un moteur de templates permet de formater des données plus simplement, comme par exemple des dates, en fonction de la langue actuellement utilisée par l'utilisateur.

Enfin, bien utilisés, les templates permettent d'utiliser des fonctions d'inclusion, ce qui permet de hiérarchiser les fichiers, et de s'assurer de l'unité de certaines parties du site. Ainsi, les *headers*, les *footers*, et autres menus que l'on retrouve sur toutes les pages du site sont définis chacun dans un seul fichier et inclus dans tous les autres.

2.3.1 Exemple de template Jinja2

```
{% extends "core/base.jinja" %} {# (1) #}

{% block title %} {# (2) #}
{% trans user_name=user.get_display_name() %}{{ user_name }}'s tools{% endtrans %} {# (3) #}
{% endblock %}

{% block content %}
    <h3>{% trans %}User Tools{% endtrans %}</h3> {# (4) #}
<p><a href="{{ url('core:user_profile', user_id=request.user.id) }}{# (5) #}">
    {% trans %}Back to profile{% endtrans %}</a>
</p>

<h4>{% trans %}Sith management{% endtrans %}</h4>
```

```
<ul>
{% if user.is_root %} {# (6) #}
    <li><a href="{{ url('core:group_list') }}">{% trans %}Groups{% endtrans %}</a></li>
{% endif %}
{% if user.is_in_group(settings.SITH_GROUP_ACCOUNTING_ADMIN_ID) %}
    <li><a href="{{ url('accounting:bank_list') }}">Accounting</a></li>
{% endif %}
{% if user.is_in_group(settings.SITH_MAIN_BOARD_GROUP) or user.is_root %}
    <li><a href="{{ url('subscription:subscription') }}">Subscriptions</a></li>
    <li><a href="{{ url('counter:admin_list') }}">Counters management</a></li>
{% endif %}
</ul>

<h4>{% trans %}Club tools{% endtrans %}</h4>
<ul>
    {% for m in user.memberships.filter(end_date=None).all() %} {# (7) #}
    <li><a href="{{ url('club:tools', club_id=m.club.id) }}">{{ m.club }}</a></li>
    {% endfor %}
</ul>
{% endblock %}
```

(1) Nous faisons ici une extension d'un template existant afin de bénéficier des blocs déjà défini, et afin d'intégrer le contenu de ce template dans celui déjà défini.

(2) `title` est un bloc défini dans le template `base.jinja`. Le redéfinir joue alors le même rôle qu'une surcharge de méthode dans de l'héritage, et permet de remplacer le contenu du bloc, tout en conservant sa place dans le template parent.

(3) La variable `user` faisant ici partie du contexte, nous pouvons donc appeler une de ses méthodes pour obtenir un contenu dynamiquement.

(4) Les blocs `{% trans %}` et `{% endtrans %}` permettent de définir les chaînes qui vont ensuite être traduites.

(5) L'appel à la fonction `url()` permet de résoudre la route afin d'obtenir l'adresse appropriée en fonction des arguments passé. Cette fonction fait généralement partie du contexte global, et est donc accessible dans tous les templates.

(6) Les structures conditionnelles permettent d'afficher ou pas un élément en fonction de la valeur d'une variable ou du retour d'une fonction.

(7) Le `for` permet, comme en Python, d'itérer sur les éléments d'une liste. Ici, on fait même une requête via l'ORM de *Django* en utilisant un filtre pour obtenir directement des valeurs depuis la base de donnée de manière transparente.

2.3.2 Le contexte

Le contexte dans lequel le template s'execute influe beaucoup sur la capacité de *Jinja* à s'adapter dynamiquement au contenu. Plus on a de variables disponibles, plus on va pouvoir générer un contenu s'y adaptant.

Il est possible de définir le contexte global, et donc ce qui est accessible dans tous les templates, comme il est possible d'ajouter manuellement et spécifiquement des variables au contexte pour un template particulier, dans une vue particulière.

3 Organisation du projet

Après cette présentations des différentes technologies employées dans le projet, passons maintenant à une partie plus spécifique à Sith en lui même.

3.1 Les options spécifiques

3.1.1 Django-jinja

Jinja n'étant pas inclus de base dans *Django*, le paquet *Django-jinja* a été mis en place afin de bénéficier au mieux des performances de chacun, comme les filtres personnalisés de *Django* dans la puissance des macros de *Jinja*. Tout cela se trouve dans la variable `TEMPLATES`.

Jinja a été ajouté afin de s'occuper uniquement des fichiers ayant l'extension `.jinja` dans les dossiers `templates` de chaque application.

Un certain nombre de variables et fonctions ont été ajoutés au contexte global. Parmis elles, l'ensemble des filtres que *Django* fournit, mais aussi un filtre pour passer de *Markdown* à *HTML*, l'ensemble du contenu de `settings`, et enfin des fonction utiles dont voici la liste :

- **can_edit_prop** permet, en fonction d'une variable `user` et d'un objet, de savoir si l'utilisateur donnée peut modifier les propriétés de cet objet.
- **can_edit** permet, en fonction d'une variable `user` et d'un objet, de savoir si l'utilisateur donnée peut éditer l'objet.
- **can_view** permet, en fonction d'une variable `user` et d'un objet, de savoir si l'utilisateur donnée peut voir l'objet.

3.1.2 Le fichier settings_custom.py

Afin de faciliter la configuration des différentes instances du projet, un fichier `settings_custom.py` a été créé à côté de `settings.py`. Celui-ci est automatiquement inclu à la fin de `settings.py`, pour que tout ce qui est défini dans le `_custom` vienne remplacer les valeurs par défaut. Seul `settings.py` est versionné dans *Git*, et est exhaustif concernant la configuration et les variables requises. `settings_custom.py` quant à lui n'est même pas obligatoire, et s'il

existe, ne peut contenir que quelques variables qui diffèrent de la configuration par défaut [1].

3.2 Les commandes ajoutées

Si cela ne suffit pas, il est possible d'enrichir de nouvelles commandes le script `manage.py`. Cela a été fait pour *Sith*, afin de pouvoir très rapidement déployer un environnement en ayant déjà les quelques données nécéssaires au fonctionnement du projet, comme le groupe `root` par exemple.

3.2.1 setup

La fonction `setup` s'occupe simplement de supprimer le fichier `db.sqlite3`, qui est le fichier de base de donnée utilisé pour le développement, et relance une procédure de migration pour reconstruire une base de donnée propre avant de la peupler.

Cette commande permet donc de s'assurer que la base de donnée utilisée est neuve et non corrompue.

3.2.2 populate

`populate` permet de remplir la base de donnée avec dans un premier temps les données **nécessaires** au bon fonctionnement du site. Cela comprend notamment un superutilisateur, les groupes définis dans `settings.SITH_GROUP_*_ID`, dont le groupe `root` fait partie, une première page de Wiki, ainsi qu'un club racine, l'AE dans notre cas.

Cette fonction prend un éventuel argument, `--prod`, qui lui permet de mettre en place le strict minimum énoncé précédemment. Sinon, elle continue en ajoutant un certain nombre de données pratiques pour le développement, comme un certain nombre d'utilisateurs avec différents droits, de nouvelles pages dans le Wiki, de nouveaux clubs, des comptoirs et des produits, ou encore des données de comptabilité.

L'argument `--prod` peut, en outre, être passé directement depuis la fonction `setup`.

1. Typiquement, ajouter **DEBUG=True**

4 Les applications

Chaque application va être détaillée ici. Cela permet de mettre en valeur le rôle de chacune, et de signaler les éventuelles particularités qui peuvent s'y trouver.

4.1 Core

4.1.1 Résumé

L'application *Core* est de loin la plus importante de toutes. C'est elle qui gère les utilisateurs ainsi que leurs droits. Le CMS y est aussi définit pour tout ce qui est pages de Wiki, pages statiques, ou l'ajout du filtre `markdown` pour les templates.

4.1.2 Liste des modèles

— **Group**
Ce modèle se subdivise en deux : RealGroup et MetaGroup, décrivants respectivement un vrai groupe géré à la main dans la liste des groupes, et un meta-groupe, géré automatiquement, en général par les clubs, ou bien par les cotisations.

— **User**
Le modèle des utilisateurs, qui est ensuite décliné ou référencé dans beaucoup d'applications pour les utilisations spécifiques. C'est toutefois ici que sont déclarés les fonctions de gestion des droits des utilisateurs, afin de pouvoir les utiliser partout ailleurs.

— **AnonymousUser**
Cette classe n'est pas un modèle stocké en base, puisqu'elle sert à instancier la variable `user` lorsqu'aucun utilisateur n'est connecté au site.

— **Page**
Décrit une entité page, servant dans le Wiki ou pour les pages statiques du site. Cette classe s'occupe des méta-données de la page, comme ses droits, mais son contenu est en réalité stocké dans un objet `PageRev`.

— **PageRev**
Décrit une révision de page. Utiliser une autre classe avec une `ForeignKey` permet de gérer facilement un historique des révisions.

4.1.3 La gestion des droits

La gestion des droits est implémentée de manière globale dans l'application Core.

On trouve en effet dans `views/__init__.py` un certain nombre de mixins [1] s'occupant de cela, en se basant sur un modèle général permettant de rendre compatible rapidement n'importe quel modèle que l'on voudrait protéger. Il suffit alors de déclarer dans la classe une certain nombre de méthodes et/ou d'attributs, le reste étant simplement déjà pris en charge par les mixins suivants :

— **CanCreateMixin**
Cette classe est à mettre en parente d'une classe héritant de `CreateView`, afin d'empêcher quelqu'un n'ayant pas les droits de créer un objet.
Méthode correspondante à créer dans les modèles :
`def can_be_created_by(user):`
(Attention, ce n'est pas une méthode prenant `self` en premier paramètre !)

— **CanEditPropMixin**
Cette classe protège l'objet pour l'édition avancée. Par exemple : éditer les droits sur une page, ou éditer les droits accordé à un utilisateur.
Attribut correspondant à créer dans les modèles :
```
owner_group = models.ForeignKey(Group,
     related_name="owned_user", default=settings.SITH_GROUP_ROOT_ID)
```
Méthode correspondante à créer dans les modèles :
`def is_owned_by(self, user):`

— **CanEditMixin**
Cette classe protège l'objet pour l'édition non avancée. Par exemple : éditer une page, ou éditer le profil d'un utilisateur.
Attribut correspondant à créer dans les modèles :
```
edit_groups = models.ManyToManyField(Group,
     related_name="editable_user", blank=True)
```
Méthode correspondante à créer dans les modèles :
`def can_be_edited_by(self, user):`

— **CanViewMixin**
Cette classe protège l'objet pour la vue. Par exemple : consulter une page, ou voir le profil d'un utilisateur.
Attribut correspondant à créer dans les modèles :
```
view_groups = models.ManyToManyField(Group,
     related_name="viewable_user", blank=True)
```

1. Un mixin est, dans *Django*, un terme désignant une classe abstraite qui ne peut pas servir de parente seule. Elle permet de surcharger certaines méthodes d'une autre classe abstraite afin de l'adapter à un comportement plus spécifique, mais reste totalement inutile quand elle est seule. La gestion des droits est un bon exemple puisqu'elle ne s'occupe pas vraiment de traitement des données comme les autres vues le feraient, elle permet simplement d'ajouter une condition à une autre classe où cette dernière renverrait un *403 Forbidden*

Méthode correspondante à créer dans les modèles :
`def can_be_viewed_by(self, user):`

Pour savoir si l'on doit implémenter les méthodes, les attributs, ou les deux, il faut simplement se poser la question de savoir si l'objet en question requiert une gestion des droits à l'échelle de la classe ou à l'échelle de l'objet, et si cette gestion peut être calculé par de la logique.

Si on a besoin d'une gestion pour la classe, ou si du code peut être implémenter pour déterminer qui peut avoir tel droit, alors la méthode suffira. Mais si on a besoin d'une gestion au niveau de l'objet, alors il faudra certainement recourir aux attributs.

Exemples :

— Les comptes en banque sont gérés uniquement par les personnes faisant partie du groupe `admin-compta`. Ils ont donc tous les mêmes droits, c'est une gestion au niveau de la classe, donc les méthodes suffisent.

— Les classeurs de comptabilité sont gérés par les trésoriers des clubs, ils n'ont pas tous les mêmes droits, mais cela peut tout de même se calculer en fonction des postes dans les clubs correspondants. On a donc besoin des méthodes uniquement.

— Les pages n'appartiennent pas forcément à un club, ni à une quelconque entité, mais ont tout de même besoin de gestion des droits au niveau de l'objet. L'ajout des attributs est donc nécessaire pour pouvoir gérer cela au cas par cas.

4.2 Subscription

4.2.1 Résumé

Cette application ajoute le support des cotisations. Elle fournit également les interfaces de cotisation.

4.2.2 Liste des modèles

— **Subscription**
Un modèle cotisation, pour stocker ces dernières. Cette classe fait automatiquement les calculs de début et de fin de cotisation en fonction de la date du jour, du type de cotisation, et de la durée en semestre de cotisation.

4.3 Accounting

4.3.1 Résumé

Cette application sert à gérer la comptabilité. Elle est architecturée de façon hiérarchique, avec en haut, les comptes bancaires réels, qui contiennent eux des comptes de clubs, permettant de les diviser en plusieurs petits comptes en interne, et enfin les classeurs de trésorerie, propres à chaque compte club, permettant de faire les comptes en triant par semestre.

De plus, cette application définit un nouveau type de champ dans la base de donnée : le champ `CurrencyField`, permettant de stocker de valeurs monétaires.

4.3.2 Liste des modèles

— **BankAccount**
Le modèle des comptes bancaires.
— **ClubAccount**
Le modèle des comptes clubs.
— **GeneralJournal**
Le modèle des classeurs de comptabilité, généralement semestriels, mais ils peuvent toutefois fonctionner en année pour les activités plus longues comme le Gala.
— **AccountingType**
Le modèle pour stocker les types comptables, servant à remplir les opérations.
— **SimpleAccountingType**
Le modèle pour stocker les types comptables simplifiés, servant à remplir les opérations.
— **Label**
Le modèle pour stocker les étiquettes, servant à classifer les opérations.
— **Operation**
Le modèle des opérations, servant à remplir les classeurs comptables. Une opération peut être un débit ou un crédit, et permet ensuite d'éditer des factures, par exemple.

4.4 Counter

4.4.1 Résumé

Cette application s'occupe de la gestion des comptoirs. Elle définit ainsi des produits, et ajoute également le support du compte AE pour les utilisateurs.

4.4.2 Liste des modèles

— **Customer**
Ce modèle étend l'utilisateur pour lui rajouter un compte AE. Il est lié à la classe `User` par un `OneToOneField`.
— **ProductType**
Ce modèle ajoute des types de produits afin de catégoriser ces derniers.
— **Product**
Ce modèle décrit les produits pouvant être vendus dans les différents comptoirs.
— **Counter**
Ce modèle décrit les comptoirs, qui permettent de générer des recharges de compte et des ventes de produits.
— **Refilling**
Ce modèle permet de stocker les rechargements de compte.

— **Selling**
Ce modèle permet d'enregistrer toutes les ventes de produits.

4.5 Club

4.5.1 Résumé

Cette application permet de générer les clubs et les adhésions des utilisateurs à ceux-ci.

4.5.2 Liste des modèles

— **Club**
Le modèle des clubs.
— **Membership**
Le modèle des adhésions. Stocker cela dans un modèle à part permet de conserver un historique des personnes ayant eu un rôle quelconque dans un club quelconque.

5 Conclusion

Encore une fois, ce rapport ne se veut absolument pas exhaustif sur quoi que ce soit.

Concernant *Python*, *Django*, ou *Jinja*, les documentations respectives sont toujours très bien faites, et permettront de répondre à toutes les questions techniques concernant les technologies.

Concernant le projet *Sith* en lui-même, ce rapport n'est pas non plus exhaustif. Pour cela, lire le code des différentes sections sera le meilleur moyen de comprendre le fonctionnement des différentes applications. Pour obtenir plus rapidement un résumé à jour des sources, le fichier `Doxyfile` présent à la racine du site permet de regénérer de la documentation exhaustive rapidement à l'aide de *Doxygen* (voir la section correspondante dans le README).

J'espère toutefois que même s'il n'est pas complet, ce rapport permettra à tout futur contributeur de rentrer plus rapidement dans le projet.

L'idéal serait également de maintenir à jour ce rapport du mieux possible en même temps que le développement avance, même si je ne me fais guère d'illusions en pratique.

5.1 Pour le futur...

En l'état actuel des choses, un grand nombre d'éléments sont encore manquants au site :

- Une API REST pour pouvoir facilement intégrer d'autres outils autour du site.
- Du CSS, pour qu'il soit un peu plus joli à regarder.
- Du Javascript, et particulièrement de l'AJAX pour améliorer l'ergonomie de certaines pages.
- Un grand nombre de vues pour aider à gérer les données plus efficacement, ou à les gérer tout court dans certains cas.
- Quelques applications utiles à qui existent sur le site actuel, mais que je n'ai pas encore eu le temps de développer : un forum, une galerie de photos, une gestion basique des fichiers pour uploader des documents dans les pages ou le forum, un système de news, une newsletter (Weekmail), une gestion des sondages et des élections, etc...

5.2 Apports personnels

Même s'il est vrai que j'ai beaucoup appris en développant ce site, cela reste avant tout un travail de quantité plus que de qualité : on définit des modèles, puis les vues correspondantes en terminant par les templates, et on répète l'opération pour l'application suivante.

Mais j'ai tout de même pu mettre en place de l'intégration continue pour ce projet, ce qui a été certes, très rapide, mais toutefois très enrichissant, étant donnée que ces méthodes sont très en vogue ces derniers temps.

J'ai également pu me familiariser d'avantage avec le fonctionnement d'un ORM, et la magie noire que cela permet de faire [1].

Enfin, j'espère que ce projet va, en plus d'être correctement mené au bout, pouvoir être repris par la suite par les futurs membres de l'équipe info de l'AE, et pourquoi pas par d'autres contributeurs...

Skia <skia@libskia.so> - 2016

1. Je trouve toujours magique de pouvoir faire une requête SQL au milieu d'un template sans que cela paraisse affreux :)