HappySchool

Version 0.1

Manuel Tondeur

déc. 07, 2022

Table des matières

HappySchool est une application web de gestion administrative et pédagogique pour les écoles primaires et secondaires. Fruit d'une expérience de terrain, elle se veut simple d'utilisation tout en restant configurable afin de s'adapter aux particularités des établissements scolaires.

HappySchool est constitué d'une série d'applications indépendantes qui peuvent être activés/désactivés selon les besoins :

- Un **annuaire** des élèves et des enseignants (la seule partie obligatoire). Elle permet de trouver une personne par nom/classe et d'afficher des informations la concernant, de générer un trombinoscope par classe ou encore d'avoir un rapide aperçu des aux autres applications.
- Un **dossier des élèves** qui permet d'assurer un suivi des élèves que ce soit au niveau des informations personnelles ou des sanctions. Il propose également de gérer le suivi des sanctions, les demandes, organiser des conseils disciplinaires ainsi que le vérifier la présence pour les retenues.
- Une prise des **absences des élèves** qui fonctionne sur tablette et hors-connexion. Deux interfaces ont été développées, une pour les éducateurs et une pour les enseignants.
- Une gestion des arrivées et sorties de l'**infirmerie** avec une notification automatique par courriel aux différents responsables de l'école.
- Une gestion des **appels** au travers de notifications. Par exemple, en notifiant un éducateur qu'un élève serait absent pour de cause de maladie suite à l'appel d'un parent.
- Un suivi des **absences des professeurs** pour la gestion administrative.
- Un suivi des **changements horaires** inhabituelles des enseignants afin d'avoir une connaissance centralisé d'où doivent se trouver les élèves. Ceux-ci peuvent être affichés sur un grand écran d'affichage.

HappySchool s'appuie sur toute une série de projets opensource dont les frameworks Django et Vue.js.

CHAPITRE 1

Installation

Happyschool a été prévu pour fonctionner sur le plus d'environnement possible, il peut donc techniquement tourner sur linux, Windows et OSX mais sans certification pour ces deux derniers. Cependant, il a été activement développé sous linux et c'est donc ce type système d'exploitation, et en particulier Ubuntu qui est recommandé et qui sera utilisé dans ce manuel.

Configuration minimale :

- Ubuntu Server 18.04 ou 20.04
- 2GO de mémoire vive

1.1 Développement

HappySchool propose deux solutions pour installer un environnement de développement. Soit par une image Docker, soit par une installation manuelle.

1.1.1 Docker

Docker est une solution de conteneurisation qui permet d'embarquer tous les logiciels nécessaire au bon fonctionnement d'HappySchool, il peut être vu comme une solution *légère* de virtualisation. Il permet ainsi de lancer HappySchool de manière indépendante par rapport au système.

Prérequis

La première étape est donc d'"installer Docker et docker-compose. Par exemple, sur ubuntu la commande suivante devrait installer les deux logiciels.

```
sudo apt install docker.io docker-compose
```

HappySchool

L'étape suivante est de télécharger le code d'HappySchool avec le logiciel git. La commande suivante va télécharger et créer un dossier happyschool avec la dernière version d'HappySchool dans le répertoire courant.

```
git clone https://github.com/ISLNamur/happyschool
```

Il faut ensuite préparer le fichier de configuration d'HappySchool. Placez-vous dans le nouveau dossier `happyschool` et créer votre fichier de configuration à partir du fichier exemple avec la commande suivante.

```
cp happyschool/setting.example.py happyschool/setting.py
```

Construire les images docker

Tout est maintenant prêt pour créer les conteneurs avec la commande suivante. Par défaut dans ubuntu, la commande `docker` doit être utilisé avec les droits administrateurs d'où l'utilisation de `sudo` dans les prochaines commandes.

```
sudo docker-compose build --no-cache --force-rm
```

Démarrer les images

Pour démarrer les images dockers, il nous reste plus qu'à lancer :

```
sudo docker-compose up
```

Vous pouvez maintenant accéder à HappySchool sur http://127.0.0.1:8000 et vous connecter avec l'utilisateur *admin* et le mot de passe *admin*.

Si vous voulez accéder au conteneur pour créer/appliquer une migration django ou bien pour générer le code javascript. La commande suivante vous-y donnera accès (vous pouvez faire `sudo docker ps` pour voir quelles sont les conteneurs actifs à accéder).

```
sudo docker exec -ti happyschool_django_1 bash
```

1.1.2 Installation manuelle

Base de donnée

Happyschool utilise une base de donnée *postgresql* avec une extension pour gérer les caractères accentués.

Pour l'installer :

```
sudo apt install postgresql postgresql-contrib
```

Pour activer l'extension pour toutes les futures base de données :

```
sudo -u postgres bash -c "psql -U postgres -d 'template1' -c 'CREATE EXTENSION␣
↪unaccent;'"
```

Puis créer un utilisateur et une base de donnée. D'abord accéder au shell `psql` :

```
sudo -u postgres bash -c "psql"
```

Et introduire les commandes SQL suivantes en spécifiant utilisateur, mot de passe et nom de la base de donnée :

```
CREATE USER newuser WITH PASSWORD 'yourpassword';
CREATE DATABASE mydb WITH OWNER newuser;
ALTER USER newuser CREATEDB;
\quit
```

Paquets système

Happyschool utilise principalement du python pour fonctionner mais pas seulement. Il utilise entre autres `git` pour la collecte et les mises à jour du code et un serveur `redis` pour communiquer entre différents processus.

Pour les installer :

```
sudo apt install curl
curl -sL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt install libldap2-dev libsasl2-dev python3-pip git python3-dateutil ttf-
↪bitstream-vera redis-server build-essential libssl-dev zlib1g-dev libbz2-dev␣
↪libreadline-dev libsqlite3-dev wget llvm libncurses5-dev libncursesw5-dev xz-utils␣
↪tk-dev libffi-dev liblzma-dev nodejs
```

HappySchool

Le code d'Happyschool est disponible sur github. Pour l'installer, placez-vous dans le dossier où vous vous voulez l'installer puis récupérer le code avec git :

```
cd /home/user/mon/dossier
git clone https://github.com/ISLNamur/happyschool.git
```

HappySchool s'appuie sur le framework Django ainsi que toutes une série de modules python. Afin des les gérer ainsi que leur versions, *pipenv* et *pyenv* sont utilisés. Pour les installer avec un shell bash :

```
pip3 install --user pipenv
curl https://pyenv.run | bash
echo 'export PATH="$HOME/.pyenv/bin:$HOME/.local/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(pyenv init -)"' >> ~/.bashrc
echo 'eval "$(pyenv virtualenv-init -)"' >> ~/bashrc
cd happyschool
PIPENV_YES=1 pipenv install
```

Il existe plusieurs niveaux de configurations pour Happyschool, le plus bas niveau est `happyschool/settings.py` (chemin relatif au dossier racine d'Happyschool). Un fichier exemple est disponible et peut être copié :

```
cp happyschool/settings.example.py happyschool/settings.py
```

Dans celui-ci vous retrouverez la possibilité d'activer/désactiver les applications, configurer l'accès à la base de donnée (pensez à mettre le nom de la db, l'utilisateur et le mot de passe définit plus haut !), configurer le serveur d'envoi

d'email, configurer l'authentification à un serveur LDAP/ActiveDirectory, etc. Plus de détails sont disponibles dans la section *Configuration*.

Pour installer les dépendances javascript et les compiler, dans le dossier racine (cela peut prendre un peu de temps) :

```
npm install
./node_modules/.bin/webpack --config webpack.dev.js
```

Pour écrire les schémas dans la base de donnée :

```
pipenv run ./manage.py migrate
```

Certaines applications ont besoin que les groupes soient déjà accessibles pour pouvoir fonctionner. La commande suivante permet de les générer à partir du fichier `happyschool/settings.py` :

```
pipenv run ./manage.py creategroups
```

Vous pouvez créer un super utilisateur en répondant aux questions posées par :

```
pipenv run ./manage.py createsuperuser
```

Finalement, pour la lancer le serveur de test :

```
pipenv run ./manage.py runserver --nostatic
```

HappySchool devrait maintenant être accessible à l'adresse suivante : http://127.0.0.1:8000. La prochaine étape est la *Configuration* d'Happyschool.

1.2 Production

HappySchool est construit sur un certain nombre de brique logiciel, ses dépendances. Par souci de simplicité, il est conseillé d'installer sur une machine réservée à l'emploi d'HappySchool même si, au vu de la plupart des logiciels utilisés, il est tout à fait possible d'utiliser une machine existante. Une machine virtuelle (ou une instance docker) sera donc tout à fait appropriée et recommandé pour l'installation. La manière la plus rapide et la plus simple pour l'installation est d'utiliser un playbook ansible pour automatiser l'installation et la configuration d'HappySchool. Cependant celle-ci se limite pour le moment à ubuntu 18.04. Pour une installation sur un autre système ou plus poussé, une installation manuelle est décrite ci-dessous. A vous à l'adapter selon les spécificités de votre machine.

1.2.1 Ansible

Ansible est un outil puissant qui permet d'automatiser l'installation et la configuration d'un ou plusieurs serveurs. En règle générale, il s'utilise à distance à travers une session *ssh* vers le ou les serveurs mais peut très bien s'utiliser en local. Un *role* pour installer HappySchool est disponible pour n'importe quelle utilisation. Un script *shell* est également fourni pour faciliter l'installation en local. Pour télécharger le role et l'inclure dans votre propre playbook clonez le dépôt correspondant :

```
git clone https://github.com/ISLNamur/happyschool-ansible
```

La configuration de votre instance (superutilisateur, applications actives, etc) se fait dans un *playbook* que vous pouvez créer à partir du fichier `happyschool.example.yml`. Les possibilités de configuration se trouve dans `roles/common/defaults/main.yml`. Ensuite, à la racine du dépôt exécutez le script suivant qui utilisera un *playbook* `happyschool.yml` :

```
./recipe.sh
```

Celui-ci devrait vous demander votre mot de passe pour l'installation des paquets système. Au final, HappySchool sera installé dans `/home/utilisateur/happyschool`.

1.2.2 Installation manuelle

Base de donnée

Happyschool utilise une base de donnée *postgresql* avec une extension pour gérer les caractères accentués.

Pour l'installer :

```
sudo apt install postgresql postgresql-contrib
```

Pour activer l'extension pour toutes les futures base de données :

```
sudo -u postgres bash -c "psql -U postgres -d 'template1' -c 'CREATE EXTENSION␣
↪unaccent;'"
```

Puis créer un utilisateur et une base de donnée. D'abord accéder au shell `psql` :

```
sudo -u postgres bash -c "psql"
```

Et introduire les commandes SQL suivantes en spécifiant utilisateur, mot de passe et nom de la base de donnée :

```
CREATE USER newuser WITH PASSWORD 'yourpassword';
CREATE DATABASE mydb WITH OWNER newuser;
\quit
```

Paquets système

Happyschool utilise principalement du python pour fonctionner mais pas seulement. Il utilise entre autres `git` pour la collecte et les mises à jour du code et un serveur `redis` pour communiquer entre différents processus.

Pour les installer :

```
curl -sL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt install libldap2-dev libsasl2-dev python3-pip git python3-dateutil ttf-
↪bitstream-vera redis-server build-essential libssl-dev zlib1g-dev libbz2-dev␣
↪libreadline-dev libsqlite3-dev wget curl llvm libncurses5-dev libncursesw5-dev xz-
↪utils tk-dev libffi-dev liblzma-dev python-openssl nodejs
```

HappySchool

Le code d'Happyschool est disponible sur github. Pour l'installer, placez-vous dans le dossier où vous vous voulez l'installer puis récupérer le code avec git :

```
cd /home/user/mon/dossier
git clone https://github.com/ISLNamur/happyschool.git
```

HappySchool s'appuie sur le framework Django ainsi que toutes une série de modules python. Afin des les gérer ainsi que leur versions, *pipenv* et *pyenv* sont utilisés. Pour les installer avec un shell bash :

```
pip3 install --user pipenv
curl https://pyenv.run | bash
echo 'export PATH="$HOME/.pyenv/bin:$PATH"' >> /.bashrc
echo 'eval "$(pyenv init -)"' >> /.bashrc
echo 'eval "$(pyenv virtualenv-init -)"' >> /bashrc
cd happyschool
PIPENV_YES=1 pipenv install
```

Il existe plusieurs niveaux de configurations pour Happyschool, le plus bas niveau est `happyschool/settings.py` (chemin relatif au dossier racine d'Happyschool). Un fichier exemple est disponible et peut être copié :

```
cp happyschool/settings.example.py happyschool/settings.py
```

Dans celui-ci vous retrouverez la possibilité d'activer/désactiver une application, configurer l'accès à la base de donnée (pensez à mettre le nom de la db, l'utilisateur et le mot de passe définit plus haut), configurer le serveur d'envoi d'email, configurer l'authentification à un serveur LDAP/ActiveDirectory, etc. Plus de détails sont disponibles dans la section *Configuration*.

Pour installer les dépendances javascript et les compiler, dans le dossier racine (cela peut prendre un peu de temps) :

```
npm install
./node_modules/.bin/webpack --config webpack.dev.js
```

Pour écrire les schémas dans la base de donnée :

```
pipenv run ./manage.py migrate
```

Certaines applications ont besoin que les groupes soient déjà accessibles pour pouvoir fonctionner. La commande suivante permet de les générer à partir du fichier `happyschool/settings.py` :

```
pipenv run ./manage.py creategroups
```

Vous pouvez créer un super utilisateur en répondant aux questions posées par :

```
pipenv run ./manage.py createsuperuser
```

Ensuite récupérez les fichiers statiques (css,...) utilisés par django et ses applications. Pour cela, assurez-vous que `DEBUG = FALSE` dans votre fichier `happyschool/settings.py` et lancez la commande suivante :

```
pipen run ./manage.py collectstatic
```

Supervisord

Supervisord est un système de gestion des processus. Il a pour but de coordoner le (re)démarrage et l'arrêt des différents processus utilisés pour le bon fonctionnement d'Happyschool. Il s'installe avec `pip` :

```
sudo pip3 install supervisor
```

Supervisor se configure avec le fichier `/etc/supervisor/supervisord.conf` (à créer) pour la configuration générale :

```
[unix_http_server]
file=/var/run/supervisor.sock   ; the path to the socket file
chown = root
```

(suite sur la page suivante)

(suite de la page précédente)

```
[supervisord]
nodaemon = False
childlogdir = /var/log/happyschool
user = root
pidfile = /var/run/supervisord.pid

[rpcinterface:supervisor]
supervisor.rpcinterface_factory = supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl = unix:///var/run/supervisor.sock ; use a unix:// URL  for a unix socket

[include]
files = /etc/supervisor/conf.d/*.conf
```

Ainsi qu'un fichier pour chacun des processus que supervisor doit gérer. `/etc/supervisor/conf.d/daphne.conf` :

```
[program:daphne]
command=/home/myuser/.local/bin/pipenv run daphne -b 0.0.0.0 -p 8080 happyschool.
↪asgi:application ; Remplacer 'myuser' par l'utilisateur courant !
directory=/home/myuser/happyschool            ; Remplacer 'myuser' par l'utilisateur␣
↪courant !
autostart=true
autorestart=true
environment=HOME="/home/myuser",USER="myuser"   ; Remplacer 'myuser' par l
↪'utilisateur courant !
user=myuser                                   ; Remplacer 'myuser' par l'utilisateur␣
↪courant !
stdout_logfile_maxbytes=10MB
```

et `/etc/supervisor/conf.d/celery.conf` :

```
[program:celery]
command=/home/myuser/.local/bin/pipenv run celery -A happyschool worker -l info ;␣
↪Remplacer 'myuser' par l'utilisateur courant !
directory=/home/myuser/happyschool            ; Remplacer 'myuser' par l'utilisateur␣
↪courant !
autostart=true
autorestart=true
environment=HOME="/home/myuser",USER="myuser"   ; Remplacer 'myuser' par l
↪'utilisateur courant !
user=myuser                                   ; Remplacer 'myuser' par l'utilisateur␣
↪courant !
stdout_logfile_maxbytes=10MB
```

Vérifiez que les chemins d'accès à Happyschool ainsi que le nom d'utilisateur sont correctement configurés.

Pour s'assurer que supervisor est bien lancé au démarrage de la machine, vous pouvez créer un service dans `/etc/systemd/system/supervisord.service` :

```
[Unit]
Description=Supervisor process control system for UNIX
Documentation=http://supervisord.org
After=network.target

[Service]
```

(suite sur la page suivante)

(suite de la page précédente)

```
ExecStart=/usr/local/bin/supervisord -n -c /etc/supervisor/supervisord.conf
ExecStop=/usr/local/bin/supervisorctl shutdown
ExecReload=/usr/local/bin/supervisorctl reload
KillMode=process
Restart=on-failure
RestartSec=50s

[Install]
WantedBy=multi-user.target
```

Que vous pouvez activer avec

```
sudo systemctl enable supervisord
sudo systemctl start supervisord
```

Nginx

Nginx va nous permettre de répartir les différentes demandes entre les contenus dynamiques que va gérer daphne, et les contenus statiques (images, css, js,...). Il s'installe simplement avec :

```
sudo apt install nginx
```

Ensuite pour le configurer, modifiez le fichier `/etc/nginx/sites-available/default` :

```
server {
        listen 80 default_server;
        listen [::]:80 default_server;
        server_name mon.domaine 10.32.141.6; # Nom de domaine du serveur, l'ip n'est␣
↪pas nécessaire. À MODIFIER.
        client_max_body_size 100m;

        location /static/ {
                add_header Service-Worker-Allowed "/";
                alias /home/user/happyschool/static/; # Mettre le chemin vers les␣
↪fichiers statiques. À MODIFIER.
        }

        location /media/ {
                alias /home/user/happyschool/media/; # Mettre le chemin vers les␣
↪fichiers media (upload,...). À MODIFIER.
        }

        location /favicon.ico {
                alias /home/user/happyschool/static/favicon.ico; # Mettre le chemin␣
↪correct.
        }

        # On transmet le reste à daphne.
        location / {
                proxy_pass http://0.0.0.0:8080; # Le port d'écoute de daphne.
                proxy_set_header Upgrade $http_upgrade;
                proxy_set_header Connection "upgrade";

                proxy_redirect     off;
                proxy_set_header   Host $host;
```

(suite sur la page suivante)

(suite de la page précédente)

```
                proxy_set_header   X-Real-IP $remote_addr;
                proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_set_header   X-Forwarded-Host $server_name;
        }
}
```

Pour vérifier qu'il n'y a pas de faute de syntaxe, la commande `sudo nginx -t` est bien utile. Ensuite pour charger la nouvelle configuration :

```
sudo systemctl reload nginx
```

Happyschool devrait maintenant être accessible à l'adresse IP ou au nom de domaine que vous avez choisi. La prochaine étape est la *Configuration* Happyschool que ce soit pour l'envoi automatique des courriels ou pour le choix des applications.

CHAPITRE 2

Configuration

Happyschool disposent de trois différents niveaux de configuration :

— Le plus bas niveau, dans `happyschool/settings.py`, est la configuration liée à django et se rapporte à la base de donnée, les applications utilisées, le serveur de courriel sortant, etc.
— Le niveau intermédiare est simplement des entrées dans la base de donnée. Par exemple, les types de sanctions ou les motifs des appels.
— Finalement, le plus haut niveau est accessible directement depuis l'interface web.

2.1 settings.py

Ce fichier python se situe dans le dossier `happyschool` et **doit** être configuré pour assurer le bon fonctionnement d'Happyschool.

— `SECRET_KEY` est utilisé pour la protection des mots de passes. Si vous utilisez directement le système d'authentification de django (sans passer par un serveur LDAP/ActiveDirectory) vous **devez** générer vous-même une clé aléatoire.
— `DEBUG` comme son nom l'indique permet d'afficher des informations de ... debug lorsqu'il y a une erreur. À mettre sur `True` uniquement lorsque vous développez.
— `ALLOWED_HOSTS` indique l'ip ou le nom de domaine avec lequel les utilisateurs se connectent. Vous pouvez utiliser un joker comme ceci `ALLOWED_HOSTS = ['*']`.
— `INSTALLED_APPS` est la liste des applications installées. Par défaut, seul l'annuaire est activé. Il vous suffit de décommenter les applications que vous désirez (n'oubliez pas d'effectuer la création/migration de la base de donnée si vous ajoutez une application).
— `DATABASES` vous permet de configurer l'accès à votre base de donnée, vous pouvez ainsi spécifier le nom de la base de donnée (`NAME`), l'utilisateur (`USER`) et le mot de passe (`PASSWORD`) pour y accéder.
— `AUTHENTICATION_BACKENDS` reprend tous les types de système d'authentification. Par défaut, le *backend* intégré à django est activés.
— `EMAIL_ADMIN` est le courriel de l'administrateur. Il est principalement utilisé pour tout ce qui debug et administration.
— `EMAIL_HOST`, `EMAIL_PORT`, `EMAIL_HOST_USER`, `EMAIL_HOST_PASSWORD`, `EMAIL_FROM` pour la configuration du serveur d'envoi du courriel. Il n'est pas utilisé pour l'envoi de masse avec l'application `mail_notification`.
— `LOGIN_REDIRECT_URL` est l'url de redirection après authentification. Par défault, renvoie vers l'annuaire.

2.1.1 Configuration avancée

— Les variables `****_GROUP` font correspondre le nom des groupes utilisés dans Happyschool et ceux dans le serveur LDAP/ActiveDirectory.
— `USE_LDAP_INFO` active la prise en compte de l'utilisation d'un serveur LDAP/ActiveDirectory.
— `LDAP_HOST` et `LDAP_DOMAIN` renseignent le serveur LDAP/ActiveDirectory.
— `AUTH_LDAP_***` permettent de configurer l'accès au serveur LDAP/ActiveDirectory et sont utilisées par django-auth-ldap.
— `MAILGUN_KEY` et `SPARKPOST_KEY`. Mailgun et Sparkpost sont pour le moment les deux services supportés pour l'envoi de masse de courriels. Ces deux variables vous permettent de configurer votre accès à ces services.
— `MEDIA_SYNC` est la commande de synchronisation entre le serveur local et distant pour les pièces jointes des courriels.
— `MAIL_ANSWER` est la commande de synchronisation entre les modèles du serveur local et distant.
— `SYNC_FDB` active la synchronisation de ProEco avec libreschoolfdb.

2.2 Interface admin django

Django propose une interface d'administration pour gérer certains éléments de la base de donnée comme les utilisateurs, groupes et permissions mais également les objects utilisés par les différentes applications. HappySchool stocke également la configuration des applications dans la base de donnée et sont donc modifiable à partir de cette interface.

Seules deux applications sont nécessaires au bon fonctionnement d'HappySchool, `annuaire` et `core`. La première contient toutes les fonctionnalités de recherches utilisées par la plupart des applications. Tandis que la deuxième, comme son nom l'indique, est la brique centrale sur laquelle se construit les autres applications.

L'interface d'administration Django est accessible à l'adresse suivante : http://ip-du-serveur/admin

2.2.1 Core

Paramètres généraux

C'est l'objet `CoreSettingsModel` qui permet d'éditer les paramètres. S'il n'existe aucun objet, créez-en un et **un seul**.

Vous pourrez ici modifier le nom de l'école qui apparaîtra à différents endroits dont les documents générés ainsi que l'url d'accès d'HappySchool.

Si vous utilisez un serveur distant, vous pouvez également indiquer son url ainsi que le *token* de l'utilisateur distant à utiliser pour la synchronisation.

Établissement/enseignement

HappySchool permet de gérer plusieurs établissements et/ou enseignements afin de compartimenter les accès en fonction des applications. Par exemple, une école possédant deux établissements qui posséde une infirmerie commune devra gérer les élèves des deux parties contrairement aux dossiers des élèves qui seront traités de manière distinctes. Ceci se reflète dans les objects `TeachingModel` qui seront associés aux élèves, enseignants et classes. Au moins un objet est nécessaire.

Responsables

Les responsables représentent tous les non-étudiants, qu'ils soient professeurs, éducateurs, membres de la direction ou encore secrétaires. Ils ne pourront se connecter que si un utilisateur leur est associé. L'objet associé est le `ResponsibleModel`.

Étudiants

Les objets concernant les étudiants sont divisés en deux modèles. Le `StudentModel` contient les données essentielles de l'élève. Tandis que `AdditionalStudentInfo` contient quant à lui les informations personnelles/confidentielles. Le but étant d'avoir une distinction clair entre les deux, HappySchool permettant de fonctionner en binôme entre un serveur local à l'école et une instance dans le cloud avec un minimum d'informations.

Courriels

Certaines applications permettent d'envoyer un courriel. Le modèle `EmailModel` permet de configurer ces différents courriels. En particulier, le champ `year` indique la *responsabilité* vis à vis d'une année. Ainsi, si une application envoie un courriel concernant un élève aux responsables de l'élève, HappySchool va regarder la correspondance entre l'année dans laquelle se trouve l'élève et le champ `year` de l'objet `EmailModel` pour savoir s'il doit être envoyé à cette adresse.

2.2.2 Utilisateurs, groupes et permissions

Une des volontés derrière HappySchool est de répondre au besoin de granularité dans l'accès à l'information. Pour cela, HappySchool utilise le système de permissions de Django. C'est la section *Authentification et autorisation* dans l'interface d'administration qui contient tous les objects qui nous intéresse.

Même si l'authentification se fait par service externe (LDAP, GoogleAuth, ...), chaque utilisateur se retrouve ici. Il peut être intéressant de préenregistrer tous les futures utilisateurs afin des les associer avec un objet `ResponsibleModel`.

Afin de gérer les accès en lectures et/ou écritures pour chacune des applications, HappySchool utilise des permissions au niveau des groupes. Ainsi chaque utilisateur peut appartenir à un ou plusieurs groupes avec des permissions différentes.

De manière générale, la permission `Can view model` où `model` est le nom du modèle principal de l'application, donne accès à l'application et `Can add/change/delete model` donne accès en écriture à l'application.

Direction et sysadmin

Afin de se concentrer sur l'information utile, les responsables n'ont accès qu'aux informations concernant leurs élèves (déduit à partir de leurs classes et années). Les groupes `direction` et `sysadmin` font exceptions à cette règle et ont accès à tous les élèves. Il se peut également que l'application permette d'outrepasser cette règle.

2.3 Dossier des élèves

Le dossier des élèves est permet de gérer les informations personnelles et éducatives (dont les sanctions) des élèves. La volonté derrière cette application est double. D'une part, d'avoir un accès rapide de l'historique d'un élève. D'autre part, d'avoir un suivi journalier des informations. Il permet ainsi le suivi des sanctions (la retenue a-t-elle bien été faite ?) ainsi que la communication, concernant un élève, aux professeurs et aux parents.

2.3.1 Paramètres

Les paramètres sont gérés par le modèle `DossierEleveSettingsModel`.

Demandes de sanction

HappySchool propose deux modes de fonctionnement vis-à-vis des sanctions. Un mode « simple » sans suivi où ce qui ajouté au dossier est considéré comme fait. Et un mode de « demande » où certaines sanctions, typiquement les retenues, passent par une ou deux étapes de validation. Ainsi une sanction peut être validée par l'équipe éducative (conseil de discipline) avant sa communication avec l'élève et ses parents, mais elle peut aussi être actée après sa réalisation et ainsi s'assurer que ce qui se trouve dans le dossier correspond bien à la réalité.

Par défaut, c'est le mode « Demande » qui est activé avec le paramètre `enable_submit_sanctions`.

Conseil de discipline

Si les demandes de sanctions activées, la possibilité de passer par une étape de confirmation (facultative) par l'équipe éducative peut être activée par le paramètre `enable_disciplinary_council`.

Visibilité

Le dossier des élèves dispose d'un système relativement flexible d'accès à l'"**information**. Ainsi il est possible d'obliger certains groupes à donner la visibilité des infos à d'autres groupes ou de laisser la possibilité à l'utilisateur de rendre accessible à d'autres groupes. Par exemple, toute information laisser par un éducateur pourrait obligatoirement être visible aux membres de la direction mais la visibilité aux professeurs se ferait au cas par cas. Ou encore, une information ne pourrait être accessible qu'à celui qui l'a postée.

Pour le moment, les sanctions ne sont pas affectées par ce système de visibilité.

C'est la série de paramètres `group_allow_visibility_to` et `group_force_visibility_to` qui permet de contrôler le comportement d'accès. `group_allow_visibility_to` indique la liste des groupes qui seront visibles lors de l'ajout d'une information par le membre de `group`. Tandis que `group_force_visibility_to` oblige l'information à être visible par le groupe. Si un groupe ce trouve simultanément dans `allow` et `force`, le groupe sera bien dans la liste visible par l'utilisateur mais sera coché et non-modifiable.

Accès enseignant

Par défaut, seuls les titulaires ont accès aux informations de leurs élèves. Toutefois, il peut être préférable que l'ensemble des professeurs qui ont la classe ou qui donnent cours à un élève aient accès aux informations les concernant. Le paramètre `filter_teacher_entries_by_tenure` permet de gérer cet accès.

Courriel de l'enseignant

Le paramètre `use_school_email` permet de choisir, lors de l'envoit de courriel, entre le courriel personnel du professeur ou celui de l'école (champ `email_school` dans le modèle `ResponsibleModel`).

CHAPITRE 3

Utilisation

En cours d'écriture...

3.1 Utilisation

CHAPITRE 4

Développement

La volonté derrière l'architecture d'HappySchool est de garder une certaine simplicité autour de composants matures et flexibles. De manière générale, HappySchool peut être divisé en deux parties. Un *back-end* qui s'occupe du stockage des données et de la logique interne. Et un *front-end* qui s'occupe d'afficher les données à l'utilisateur final ainsi que de la logique de l'interface.

4.1 Développement

4.1.1 Back-end

Le cœur du *back-end* est pris en charge par Django, un framework python abordable et complet. Sa documentation est très fournie et est une aide précieuse pour le développement. Le stockage et le traitement des données est quant à lui gérer par PostgreSQL. Django propose une couche d'abstraction qui génère en interne les commandes SQL, il n'y a donc pas – ou très peu – d'interaction directe avec PostgreSQL. En fait, Django s'inspire très largement du principe MVC, il propose ainsi cette couche d'abstraction pour gérer l'aspect *modèle* ainsi que toute la logique autour des données – le *contrôleur*. Django propose également un système de gabarit – pour la partie *vue*. Toutefois, la gestion du rendu côté serveur (et donc par django et son système de gabarit) rend difficile et surtout peu pratique le dynamisme de l'interface de l'utilisateur final (la page web). C'est Vue.js, un framework javascript, qui va se charger de la partie interface – la partie *vue*. L'interaction entre les deux va se faire au travers d'une API REST avec une sérialisation en JSON. C'est l'excellent Django REST framework qui va permettre la création de l'API et s'occuper de la sérialisation/désérialisation.

4.1.2 Front-end

C'est donc le framework Vue.js qui va se charger de l'interface web, les deux points forts de celui-ci sont, sa facilité de mise en œuvre et la réactivité aux changements qu'il offre. Son principe est de définir une série de données et les transformations sur la partie html lorsque ces données changent. Cette façon de faire permet une séparation claire des différentes parties du code et donc une meilleure vue d'ensemble (sans mauvais jeu de mots). En ce qui concerne le style, le *css*, c'est le très connu Bootstrap qui est utilisé et qui s'intègre à Vue.js avec le module Bootstrap-vue. Finalement, c'est Webpack, un empaqueteur, qui orchestre toute la partie javascript (interprétation, assemblage, minification, etc).

4.1.3 Création d'une application

Au lieu d'expliquer techniquement point par point le fonctionnement d'Happyschool, expliquons comment créer une application, tant la partie *back-end* que *front-end*. Tout le code peut se retrouver sur notre dépôt.

Supposons que nous voulons créer une application simple de présence/d'absence des élèves. La première étape est de créer les fichiers de bases en python. Pour cela, Django propose une commande de création, dans le dossier racine d'Happyschool :

```
python3 manage.py startapp student_absence
```

Ceci va créer une arborescence dans un nouveau dossier `student_absence`. Le fichier `manage.py` permet de faire pas mal de choses en relation avec Django et le projet, comme un accès à un shell python pour interagir avec les modèles créés ou encore d'appliquer des changements de notre modèle.

Tout d'abord, créons notre modèle dans le fichier `student_absence/models.py` :

```
# Notre modèle d'absence.
class StudentAbsenceModel(models.Model):
    # L'étudiant qui sera absent.
    student = models.ForeignKey(StudentModel, on_delete=models.CASCADE)
    date_absence_start = models.DateField("Date de début de l'absence")
    date_absence_end = models.DateField("Date de fin de l'absence")
    # Indique si l'étudiant est absent le matin.
    morning = models.BooleanField("Absence le matin", default=True)
    # Indique si l'étudiant est absent l'après-midi.
    afternoon = models.BooleanField("Absence l'après-midi", default=True)
    user = models.CharField("Utilisateur qui a créé l'absence", max_length=100)
    datetime_creation = models.DateTimeField("Date et heure de création de l'absence",
                                             auto_now_add=True)
    datetime_update = models.DateTimeField("Date et heure de mise à jour de l'absence
↪",
                                           auto_now=True)
```

Détaillons le code : - `from django.db import models` permet d'importer la couche d'abstraction de Django concernant la base de donnée. - `from core.models import StudentModel` permet d'utiliser les étudiants qui sont dans la base de donnée. - `class StudentAbsenceModel(models.Model)` est la déclaration de notre modèle, elle hérite de `models.Model` qui est la couche d'abstraction de django. Nous devons définir tous les champs de notre modèle (les colonnes dans la base de donnée). - `student = models.ForeignKey(StudentModel, on_delete=models.CASCADE)` est notre premier champ, il permet de faire une liaison avec une autre entrée de notre base de donnée, ici, un étudiant (`StudentModel`). L'argument `on_delete=models.CASCADE` indique que si un étudiant est supprimé, l'absence sera également supprimée. - `date_absence_start` et `date_absence_end` sont de simples champs indiquant la date de début et de fin de l'absence. - `morning` et `afternoon` sont des booléens qui indiquent si l'absence a lieu le matin et/ou l'après-midi. - `user`, `datetime_creation` et `datetime_update` enregistrent des données concernant l'enregistrement d'une

entrée. `auto_now` et `auto_now_add` permet d'automatiquement enregistrer la date et heure de modification et création.

Une fois notre fichier sauvegardé, nous devons demander à Django de créer le schéma correspondant dans la base de donnée. Il faut donc d'abord spécifier à Django que nous voulons utiliser cette nouvelle application. C'est dans le fichier `happyschool/settings.py` et plus spécifiquement la variable `INSTALLED_APPS` auquel nous devons ajouter notre nouvelle application :

```
INSTALLED_APPS = [
    ...,
    'appels',
    'absence_prof',
    'dossier_eleve',
    'mail_notification',
    'mail_answer',
    'student_absence,
]
```

Cela fait, nous pouvons créer le schéma grâce à `manage.py`, nous pouvons exécuter les deux commandes suivantes :

```
python3 manage.py makemigrations student_absence
```

qui va générer les commandes SQL nécessaire à la création/modification de la base de donnée. Et finalement :

```
python3 manage.py migrate student_absence
```

qui va executer les commandes SQL.

Passons maintenant à la logique autour du modèle et de ce qui sera exposé par l'API REST.

Définissons d'abord ce qui sera exposé dans le modèle, pour cela créons un fichier `serializers.py` :

```
from rest_framework import serializers

from core.serializers import StudentSerializer
from core.models import StudentModel
from student_absence.models import StudentAbsenceModel


class StudentAbsenceSerializer(serializers.ModelSerializer):
    student = StudentSerializer(read_only=True)
    student_id = serializers.PrimaryKeyRelatedField(queryset=StudentModel.objects.
↪all(),
                                                    source='student', required=False,
                                                    allow_null=True)

    class Meta:
        model = StudentAbsenceModel
        exclude = ('user',)
        read_only_fields = ('datetime_creation', 'datetime_update',)
```

Regardons le code, nous commonçons par importer toutes les classes qui vont nous être nécessaire à notre propre *sérialiseur*. Ensuite, nous créons notre *sérialiseur*, `StudentAbsenceSerializer` qui hérite de `serializers.ModelSerializer` un *sérialiseur* qui se base sur un modèle.

Remarquez que le nom de notre classe suit une certaine convention de nommage, l'écriture est de type camel case ensuite sa fonction est inclue dans son nom, `Serializer`, ainsi que ce à quoi elle se rapporte `StudentAbsence`. Dans un projet collaboratif, il devient vite nécessaire d'établir certaines conventions, le style d'écriture en fait parti.

HappySchool essaye tant bien que mal de suivre un style conforme au PEP8 même si par souci de clarté quelques entorses sont parfaitement autorisées.

Continuons notre analyse du code et passons directement à `class Meta` qui permet une génération de notre classe de manière dynamique. Nous avons donc mis dans cette partie, le modèle auquel nous nous référons, `StudentAbsenceModel`, les champs à exclure de la sérialisation, `('user',)` ainsi que les champs en lecture seul `('datetime_creation', 'datetime_update',)`. Nous aurions pu au contraire, spécifier les champs à exposer par la variable `fields = ('un_champ', ...)`. Toute la documentation se sur la sérialisation se trouve ici.

Finalement, jetons un œil à `student` et `student_id`. À priori, le champ `student` doit normalement déjà être inclut dans la sérialisation puisqu'il n'est pas mentionné dans `exclude`. Cependant, nous aimerions avoir un comportement différent pour la création/mise à jour d'une entrée où nous voulons juste indiquer le matricule de l'étudiant et pour la lecture d'une entrée où nous voulons avoir des informations supplémentaires sur l'étudiant comme son nom, sa classe, son établissement/enseignement. `student` sera donc le champ en lecture seul avec toutes les informations et `student_id` sera le champ du matricule de l'élève nécessaire uniquement pour la création/modification d'une entrée dans la base de donnée.

Avant d'arriver à la partie *vue* de notre application, mettons en place un système de configuration pour notre application pour, par exemple, spécifier l'enseignement/établissement qui aura accès aux absences. Afin de profiter des possibilités de django, créons un modèle qui n'aura qu'une seule entrée, les paramètres de StudentAbsence.

```
from core.models import StudentModel, TeachingModel

# Les paramètres de notre application.
class StudentAbsenceSettingsModel(models.Model):
    # Les enseignements/établissements utilisés par l'application.
    # Ne pas oublier de mettre une valeur par défaut pour la création automatique.
    teachings = models.ManyToManyField(TeachingModel, default=None)
```

Ceci rajoute simplement un modèle, `StudentAbsenceSettingsModel` avec un seul champ, `teachings`, qui peut être relier à plusieurs instances de `TeachingModel`, d'où le `ManyToManyField`. Par défaut, aucun `TeachingModel` ne sera sélectioné et aucune entrée ne sera affichée. Il faudra donc que l'administrateur mette explicitement et manuellement au moins une entrée.

Comme pour `StudentAbsenceModel`, il faut appliquer les changements sur notre base de donnée avec :

```
python3 manage.py makemigrations
python3 manage.py migrate
```

Passons maintenant au cœur de notre application avec la partie *vue*, c'est-à-dire exposer notre modèle au travers d'une API REST. La classe `ModelViewSet` du DRF, permet de nous faciliter grandement le travail. En effet, en lui donnant le *sérialiseur* ainsi que quelques paramètres, il nous crée automatiquement une interface http en gérant les requêtes `GET`, `POST`, `PUT`, `DELETE`. Une des particularité d'Happyschool étant de gérer les permissions d'accès, la classe `BaseMovelViewSet` va hériter de `ModelViewSet` et gérer les accès automatiquement, un éducateur du 2ème niveau ne verra que les élèves de ce niveau. Il est évidemment possible de passer outre en surchargeant la méthode `get_group_all_access` qui attend comme retour un `QuerySet` de `Group` ayant accès à tous les niveaux. Les paramètres attendus par notre class `StudentAbsenceViewSet(BaseModelViewSet)` sont, le *sérialiseur* `serializer_class`, la requête de base à la base de donnée `queryset` (qui servira également de cache), les permissions avec `permission_classes`, les champs qui peuvent être ordonés `ordering_fields` et les filtres que nous pouvons appliquer sur nos données, `filter_class`, objet que détaillerons par la suite.

En ce qui concerne, `permission_classes`, nous pouvons demander que l'utilisateur soit connecté avec `IsAuthenticated` et utilisé le système de permission de django pour gérer l'écriture/modification/suppression qui accessible par l'interface d'admin de django.

Finalement, intéressons-nous aux capacités de filtres. Le système offert par l'application `` `django_filters <https://django-filter.readthedocs.io/en/master/>`__ `` permet une grande souplesse dans les types de filtres. Pour cela la classe

fournie par Happyschool, `BaseFilters` qui hérite de `django_filters`, permet d'indiquer les champs à filtrer de manière exacte mais également des filtres personnalisés. Dans notre application nous avons ajouté un filtre par classe.

Nous obtenons alors le code suivant :

```
import json

from rest_framework.permissions import IsAuthenticated, DjangoModelPermissions
from django_filters import rest_framework as filters

from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import TemplateView

from core.views import BaseModelViewSet, BaseFilters
from core.models import ResponsibleModel
from core.people import get_classes
from core.utilities import get_menu

from student_absence.models import StudentAbsenceModel, StudentAbsenceSettingsModel
from student_absence.serializers import StudentAbsenceSerializer,␣
↪StudentAbsenceSettingsSerializer

class StudentAbsenceFilter(BaseFilters):
    classe = filters.CharFilter(method='classe_by')

    class Meta:
        fields_to_filter = ('student_id', 'date_absence_start', 'date_absence_end')
        model = StudentAbsenceModel
        # Permet de génèrer correctement les filtres avec prises en comptes des␣
↪accents.
        fields = BaseFilters.Meta.generate_filters(fields_to_filter)
        filter_overrides = BaseFilters.Meta.filter_overrides

    def classe_by(self, queryset, name, value):
        if not value[0].isdigit():
            return queryset

        teachings = ResponsibleModel.objects.get(user=self.request.user).teaching.
↪all()
        classes = get_classes(list(map(lambda t: t.name, teachings)), True, self.
↪request.user)
        queryset = queryset.filter(student__classe__in=classes)

        if len(value) > 0:
            queryset = queryset.filter(student__classe__year=value[0])
            if len(value) > 1:
                queryset = queryset.filter(student__classe__letter=value[1].lower())
        return queryset


class StudentAbsenceViewSet(BaseModelViewSet):
    queryset = StudentAbsenceModel.objects.filter(student__isnull=False)

    serializer_class = StudentAbsenceSerializer
    permission_classes = (IsAuthenticated, DjangoModelPermissions,)
    filter_class = StudentAbsenceFilter
    ordering_fields = ('datetime_creation',)
```

Il ne nous reste plus qu'à exposer notre API par un accès http, une URL. Nous voulons tout d'abord que tout ce qui concerne notre application soit de la forme `http://mon.domaine.org/student_absence/...`, pour cela il faut ajouter au fichier `happyschool/urls.py`, l'application `student_absence` à la liste `app` du fichier. Ensuite, créons le fichier `/student_absence/urls.py` et mettons-y :

```
from rest_framework.routers import DefaultRouter

from . import views

urlpatterns = [
]

router = DefaultRouter()
router.register(r'api/student_absence', views.StudentAbsenceViewSet)

urlpatterns += router.urls
```

qui va se charger de créer les bonnes urls. Ainsi pour avoir la liste des absences il faudra faire http://localhost:8000/student_absence/api/student_absence/ si vous avez lancé le serveur de développement en local. Pour accéder à une entrée en particulier, qui a comme *id* 42, nous irons sur http://localhost:8000/student_absence/api/student_absence/42/. DRF crée automatiquement une interface web de notre API accessible depuis un navigateur, il suffit d'aller sur les liens précédents.

Pour tester notre API, django fournit un serveur de développement qui peut être lancer avec :

```
python3 manage.py runserver
```

et qui se rechargera à chaque modification de fichiers.

Nous avons maintenant notre partie *back-end* prête à l'emploi, il nous reste à développer la partie *front-end* qui sera principalement écrite en javascript avec le framework Vue.js. Pour la suite, il est conseillé d'avoir lu, au moins en partie, sa documentation et sa philosophie.

Pour notre *front-end* nous avons tout d'abord besoin d'un point d'entrée, une page html pour servir notre code javascript ainsi que le contexte de notre application *i.e.* ses paramètres. Pour cela, ajoutons à notre fichier `views.py` les éléments suivants :

```
def get_settings():
    settings_student_absence = StudentAbsenceSettingsModel.objects.first()
    if not settings_student_absence:
        # Create default settings.
        StudentAbsenceSettingsModel.objects.create().save()

    return settings_student_absence


class StudentAbsenceView(LoginRequiredMixin,
                         TemplateView):
    template_name = "student_absence/student_absence.html"
    filters = [
        {'value': 'student_id', 'text': 'Matricule'},
        {'value': 'classe', 'text': 'Classe'},
        {'value': 'date_absence_start', 'text': 'Début absence'},
        {'value': 'date_absence_end', 'text': 'Fin absence'},
    ]

    def get_context_data(self, **kwargs):
        # Add to the current context.
```

(suite sur la page suivante)

(suite de la page précédente)

```
        context = super().get_context_data(**kwargs)
        context['menu'] = json.dumps(get_menu(self.request, "student_absence"))
        context['filters'] = json.dumps(self.filters)
        context['settings'] = json.dumps((StudentAbsenceSettingsSerializer(get_
↪settings()).data))
        return context
```

La fonction `get_settings()` permet de rapatrier les paramètres de l'application et de créer le modèle correspondant s'il ne l'est pas encore. Quant à la classe `StudentAbsenceView` va exposer notre page html. Django utilise un système de template (ou gabarit) qui permet de générer dynamiquement une page pour y introduire quelques variables (paramètres, utilisateur, etc). Notre template aura la forme suivante (`student_absence/templates/student_absence/student_absence.html`) :

```
{% extends "core/base_vue.html" %}

{% block header %}
<title>HappySchool : Absence des élèves</title>
{% endblock %}
{% block content %}
<script>
    const current_app = "student_absence";
    const settings = JSON.parse('{{ settings|safe }}');
    const menu = {{ menu|safe }};
    const filters = JSON.parse('{{ filters|safe }}');
</script>
<div id="vue-app"></div>
{% load render_bundle from webpack_loader %}
{% render_bundle 'student_absence' %}

{% endblock %}
```

Le langage de gabarit utilisé par django permet non seulement d'insérer des variables avec `{{ ma_variable }}` mais également de faire des opérations logiques `{% function/logique %}`. La première ligne hérite d'un autre gabarit `core/base_vue.html` qui s'occupe de charger les certaines librairies commune à toutes les applications mais également d'exposer l'utilisateur et les groupes auxquels il appartient. `{% block header %}...{% endblock %}` permet d'insérer du code html dans la partie header de la page, ici le titre de la page. Quant à `{% block content %}...{% endblock %}` il permet d'insérer du code html dans la balise `<body>` de la page. C'est dans la balise `<script>` que le *context* de la page va être *traduit* en javascript (`{{ settings|safe }}`, ...), le filtre `` `safe <https://docs.djangoproject.com/fr/2.1/ref/templates/builtins/#safe>`__ `` indique à django de ne pas échapper les caractères (accent, guillement, etc).

`<div id="vue-app"></div>` servira à Vue.js comme nous le verons par la suite. Quant à `{% load render_bundle from webpack_loader %}` et `{% render_bundle 'student_absence' %}`, ils insérent le code généré par Vue.js.

Revenons maintenant à notre fichier `views.py` et notre class `StudentAbsenceView`. Tout d'abord, elle hérite de `LoginRequiredMixin` et de `TemplateView`. La première classe implique qu'il faut être connecté en tant qu'utilisateur pour afficher la page. La seconde est une classe générique fournie par Django pour afficher une page basée sur un gabarit. Elle demande juste de fournir le chemin vers le template avec la variable `template_name`. La fonction `get_context_data()` quant à elle, permet de passer au gabarit certaines variables, ici les paramètres, les applications à afficher dans le menu ainsi que les filtres disponibles pour l'application.

Pour que l'url sur notre classe il rajouter la ligne suivante dans le fichier `urls.py` :

```
from django.urls import path

```

(suite sur la page suivante)

(suite de la page précédente)

```
urlpatterns = [
    path('', views.StudentAbsenceView.as_view(), name='student_absence'),
]
```

Et c'est tout pour le code python. Passons au javascript !

Afin de structurer le code en différents modules, mutualiser le chargement des librairies externes mais aussi minimiser le code pour le rendre moins lourd à charger, nous utiliserons Webpack. Nous allons pour le moment nous contenter de rajouter notre application et en particulier le code javascript que nous allons écrire. Pour cela, regardons le fichier `webpack.common.js` et particulier les deux parties suivantes :

```
entry: {
        babelPolyfill: "babel-polyfill",
        menu: './assets/js/menu',
        annuaire: './assets/js/annuaire',
        appels: './assets/js/appels',
        mail_notification: './assets/js/mail_notification',
        mail_notification_list: './assets/js/mail_notification_list',
        members: './assets/js/members',
        mail_answer: './assets/js/mail_answer',
        answer: './assets/js/answer',
        dossier_eleve: './assets/js/dossier_eleve',
        ask_sanctions: './assets/js/ask_sanctions',
        student_absence: './assets/js/student_absence',
    },
```

Où nous avons rajouter le point d'entrée `./assets/js/student_absence`.

```
name: "commons",
            chunks: ["menu", "schedule_change", "appels", "mail_notification",
                "mail_notification_list", "members", "mail_answer", "dossier_eleve",
                "ask_sanctions", "annuaire", "student_absence",
            ],
```

Où nous avons rajouter notre application dans la liste des applications mutualisables.

Créons donc un simple point d'entrée :

```
import Vue from 'vue';

import StudentAbsence from '../student_absence/student_absence.vue';

var studentAbsenceApp = new Vue({
    el: '#vue-app',
    data: {},
    template: '<student-absence/>',
    components: { StudentAbsence },
})
```

La première ligne importe `Vue` et la deuxième notre composant, qui sera le cœur de la partie front-end. Finalement, la variable `studentAbsenceApp` est une application *Vue.js* qui s'attache à l'élément `<div id="vue-app">` de notre gabarit.

Ajoutons donc notre composant `assets/student_absence/student_absence.vue` :

```
<template>
    <div>
```

(suite sur la page suivante)

(suite de la page précédente)

```
        <div class="loading" v-if="!loaded"></div>
        <app-menu :menu-info="menuInfo"></app-menu>
        <b-container v-if="loaded">
            <b-row>
                <h2>Absence des élèves</h2>
            </b-row>
            <b-row>
                <b-col>
                    <b-form-group>
                        <div>
                            <b-btn variant="primary">
                                <icon name="plus" scale="1" class="align-middle"></
↪icon>
                                Nouvelle absence
                            </b-btn>
                            <b-btn variant="outline-secondary">
                                <icon name="search" scale="1"></icon>
                                Ajouter des filtres
                            </b-btn>
                        </div>
                    </b-form-group>
                </b-col>
            </b-row>
        </b-container>
    </div>
</template>

<script>
import Vue from 'vue';
import BootstrapVue from 'bootstrap-vue'
Vue.use(BootstrapVue);

import 'vue-awesome/icons'
import Icon from 'vue-awesome/components/Icon.vue'
Vue.component('icon', Icon);

import Menu from '../common/menu.vue'

export default {
    data: function () {
        return {
            menuInfo: {},
            loaded: false,
        }
    },
    methods: {
    },
    mounted: function () {
        this.menuInfo = menu;
        this.loaded = true;
    },
    components: {
        'app-menu': Menu,
    },
}
</script>
```

(suite sur la page suivante)

(suite de la page précédente)

```
<style>
.loading {
  content: " ";
  display: block;
  position: absolute;
  width: 80px;
  height: 80px;
  background-image: url(/static/img/spin.svg);
  background-size: cover;
  left: 50%;
  top: 50%;
}
</style>
```

Un composant vue possède trois parties : `<template>` qui est également un gabarit mais cette fois-ci pour le code js, `<script>` pour toute la partie logique et `<style>` pour le style *css*.

Pour dire à webpack de *compiler* le code, la commande suivante permet de le faire ainsi que de relancer la compilation à chaque changement de fichier :

```
./node_modules/.bin/webpack --config webpack.dev.js --watch
```

Si l'on pointe maintenant notre navigateur vers http://localhost:8000/student_absence et si le serveur de développement a été lancé (`python3 manage.py runserver`), notre application s'affiche enfin ! Pour faciliter le développement, il existe une extension pour navigateurs qui permet d'afficher l'état de notre application Vue.js, les composants ainsi que les différentes variables. Il est fortement conseillé de l'utiliser !

Dans ce premier jet, c'est une page simple avec un menu, un titre et deux bouttons. La partie *template* utilise beaucoup de composants venant de la librairie `BootstrapVue <https://bootstrap-vue.js.org/docs>`__ mais également le composant `Menu` qui est propre à HappySchool. Vous pouvez remarquer que la page affiche une image de chargement. Celle-ci est liée à la variable `loaded` qui initialement fausse et qui permute lorsque le composant est chargé (dans la fonction `mounted`).

La fonction principale étant d'afficher les absences, rajoutons une méthode pour rapatrier les données et les assigner à `entries`. Profitons-en pour mettre `loaded = true` lorsque les données ont été rapatriées.

```
data: function () {
    return {
        menuInfo: {},
        entriesCount: 0,
        entries: [],
        loaded: false,
    }
},
methods: {
    loadEntries: function () {
        // Get current absences.
        axios.get('/student_absence/api/student_absence/')
        .then(response => {
            this.entries = response.data.results;
            // Everything is ready, hide the loading icon and show the content.
            this.loaded = true;
        });
    },
}
```

Et modifions `mounted` :

```
mounted: function () {
    this.menuInfo = menu;

    this.loadEntries();
},
```

Si nous rechargeons la page, visuellement, rien n'a changé mais si nous regardons dans les requêtes faites à notre serveur de développement, nous voyons qu'une requête vers notre API a été effectuée. Pour le moment aucune entrée n'a encore été créée donc rien n'est rapatrié. Pour changer la donne, allons sur page d'administration de django et créons une entrée manuellement. Une fois fait, notre page devrait rapatrier notre première entrée. Vérifiez bien que cela est le cas en utilisant l'extension vuejs devtools. Et vous verrez dans le composant StudentAbsence : `entries:Array[1]`.

Créons maintenant un composant pour afficher notre absence, `assets/student_absence/studentAbsenceEntry.vue` :

```
<template>
    <div>
        <transition appear name="fade">
            <b-card>
                <b-row>
                    <b-col><strong><a href="#" @click="filterStudent">{{ rowData.
↪student.display }}</a> : </strong>
                    Absent du {{ rowData.date_absence_start }} au {{ rowData.date_
↪absence_end}}.</b-col>
                    <b-col sm="2"><div class="text-right">
                        <b-btn variant="light" size="sm" @click="editEntry" class=
↪"card-link">
                            <icon scale="1.3" name="edit" color="green" class="align-
↪text-bottom"></icon>
                        </b-btn>
                        <b-btn variant="light" size="sm" @click="deleteEntry"class=
↪"card-link">
                            <icon scale="1.3" name="trash" color="red" class="align-
↪text-bottom"></icon>
                        </b-btn>
                    </div></b-col>
                </b-row>
            </b-card>
        </transition>
    </div>
</template>

<script>
export default {
    props: {
        rowData : {type: Object},
    },
    data: function () {
        return {
        }
    },
    methods: {
        deleteEntry: function () {
            this.$emit('delete');
        },
        editEntry: function () {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            this.$emit('edit');
        },
        filterStudent: function () {
            this.$emit('filterStudent', this.rowData.student_id);
        },
    },
}
</script>

<style>
.fade-enter-active {
  transition: opacity .7s
}
.fade-enter, .fade-leave-to .fade-leave-active {
  opacity: 0
}
</style>
```

Analysons notre composant. Tout d'abord dans la partie *template*, la balise `<transition>` permet d'ajouter un effet lors de l'apparition du composant ; effet qui est décrit dans la partie *style*. À l'intérieur, le reste des balises servent principalement à décrire notre entrée. À noter toutefois, l'appel des différentes méthodes lorsque les bouttons sont pressés. Ce qui nous amène à la partie *script*, qui elle comporte un *props*, les données brutes de l'absence fournie par le composant parent et trois méthodes qui remontent au composant parent (*StudentAbsence*), lorsqu'un des boutons est pressé.

Insérons donc notre composant dans notre application. Dans la partie *template* en dessous de la ligne contenant les boutons :

```
    ...
    <b-row>
        <b-col>
            <student-absence-entry
            v-for="(entry, index) in entries"
            v-bind:key="entry.id"
            v-bind:row-data="entry"
            @delete="askDelete(entry)"
            @edit="editEntry(index)"
            @filterStudent="filterStudent($event)"
                >
            </student-absence-entry>
        </b-col>
    </b-row>
</b-container>
...
```

Dans la partie *script* :

```
<script>
...
import InfirmerieEntry from './infirmerieEntry.vue'
Vue.component('infirmerie-entry', InfirmerieEntry);

export default {
    data: function () {
        return {
            menuInfo: {},
```

(suite sur la page suivante)

(suite de la page précédente)

```
            entriesCount: 0,
            entries: [],
            loaded: false,
            currentEntry: null,
        }
    },
    methods: {
        filterStudent: function (matricule) {
        },
        askDelete: function (entry) {
            this.currentEntry = entry;
        },
        editEntry: function(index) {
            this.currentEntry = this.entries[index];
        },
        deleteEntry: function () {
            const token = { xsrfCookieName: 'csrftoken', xsrfHeaderName: 'X-CSRFToken
↪'};
            axios.delete('/student_absence/api/student_absence/' + this.currentEntry.
↪id + '/', token)
            .then(response => {
                this.loadEntries();
            });

            this.currentEntry = null;
        },
        loadEntries: function () {
            // Get current absences.
            axios.get('/student_absence/api/student_absence/')
            .then(response => {
                this.entries = response.data.results;
                // Everything is ready, hide the loading icon and show the content.
                this.loaded = true;
            });
        },
    mounted: function () {
        this.menuInfo = menu;

        this.loadEntries();
    },
    components: {
        'app-menu': Menu,
    },
}
</script>
```

Notons que la variable `currentEntry` a été ajoutée, elle permet de retenir l'entrée en cours modification/suppression. Nous savons maintenant enfin afficher une absence !

Afin de prévenir une suppression inopiné de la part de l'utilisateur, il serait pertinent d'afficher une fenêtre demandant la confirmation de la suppression d'où la méthode `askDelete`. Une telle fenêtre s'appelle un modal. Ce qui se traduit en code dans la partie *template* par :

```
...
</b-container>
<b-modal ref="deleteModal" cancel-title="Annuler" hide-header centered
    @ok="deleteEntry" @cancel="currentEntry = null">
```

(suite sur la page suivante)

(suite de la page précédente)

```
    Êtes-vous sûr de vouloir supprimer définitivement cette entrée ?
</b-modal>
...
```

Et pour la méthode `askDelete` :

```
askDelete: function (entry) {
    this.currentEntry = entry;
    this.$refs.deleteModal.show();
},
```

Après confirmation, le modal va appeler la méthode `deleteEntry` qui supprimera pour de bon l'entrée.

Il est parfois utile de partager les données d'une variable entre tous les composants, par exemple les paramètres de l'application ou les filtres appliqués sur les requêtes. C'est l'excellente librairie Vuex qui va nous offrir ces possibilités, c'est-à-dire un gestionnaire d'état. Pour qu'il soit utilisable par tous les composants, c'est dans l'application *root* qu'il doit être implémenté (`assets/js/students_absence/`) :

```
import Vue from 'vue';

import Vuex from 'vuex';
Vue.use(Vuex);

import StudentAbsence from '../student_absence/student_absence.vue';

const store = new Vuex.Store({
  state: {
    settings: settings,
    filters: [],
  },
  mutations: {
      addFilter: function (state, filter) {
          // If filter is a matricule, remove name filter to avoid conflict.
          if (filter.filterType === 'matricule_id') {
              this.commit('removeFilter', 'name');
          }

          // Overwrite same filter type.
          this.commit('removeFilter', filter.filterType);

          state.filters.push(filter);
      },
      removeFilter: function (state, key) {
          for (let f in state.filters) {
              if (state.filters[f].filterType === key) {
                  state.filters.splice(f, 1);
                  break;
              }
          }
      }
  }
});

var studentAbsenceApp = new Vue({
    el: '#vue-app',
    data: {},
    store,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    template: '<student-absence/>',
    components: { StudentAbsence },
})
```

C'est la définition de la variable `store` (qui est ajouté à notre application Vue) qui va créer notre gestionnaire d'état. C'est donc tout naturellement que `state` définit les différents états à gérer. Afin de s'assurer d'une gestion robuste des états, tous les changements d'état sont décrits explicitement. Dans notre cas, les paramètres, `settings`, ne doivent en aucun être modifiés par l'utilisateur, il n'y a donc rien à décrire. Par contre, les filtres doivent pouvoir être dynamiquement ajoutés/retirés. D'où les méthodes `addFilter` et `removeFilter` dans les `mutations`.

Voyons en pratique ce que cela donne et rajoutons un système de filtre à notre application. HappySchool fournit un composant qui repose justement sur l'utilisation du gestionnaire d'état. Le code dans *template* donne :

```
<b-row>
    <b-col>
        <b-form-group>
            <div>
                <b-btn variant="primary" @click="">
                    <icon name="plus" scale="1" class="align-middle"></icon>
                    Nouvelle absence
                </b-btn>
                <b-btn variant="outline-secondary" v-b-toggle.filters>
                    <icon name="search" scale="1"></icon>
                    Ajouter des filtres
                </b-btn>
            </div>
        </b-form-group>
    </b-col>
</b-row>
<b-row>
    <b-col>
        <b-collapse id="filters" v-model="showFilters">
            <b-card>
                <filters app="student_absence" model="student_absence" ref="filters"␣
↪@update="applyFilter"></filters>
            </b-card>
        </b-collapse>
    </b-col>
</b-row>
<b-pagination class="mt-1" :total-rows="entriesCount" v-model="currentPage" @change=
↪"changePage" :per-page="20">
</b-pagination>
<b-row>
    <b-col>
        <student-absence-entry
        v-for="(entry, index) in entries"
        v-bind:key="entry.id"
        v-bind:row-data="entry"
        @delete="askDelete(entry)"
        @edit="editEntry(index)"
        @filterStudent="filterStudent($event)"
            >
        </student-absence-entry>
    </b-col>
</b-row>
```

Et dans la partie *scripts* :

```
import Filters from '../common/filters.vue'

import StudentAbsenceEntry from './studentAbsenceEntry.vue'

export default {
    data: function () {
        return {
            menuInfo: {},
            currentPage: 1,
            entriesCount: 0,
            entries: [],
            filter: '',
            ordering: '&ordering=-datetime_creation',
            loaded: false,
            showFilters: false,
            currentEntry: null,
        }
    },
    methods: {
        changePage: function (page) {
            this.currentPage = page;
            this.loadEntries();
            // Move to the top of the page.
            scroll(0, 0);
            return;
        },
        applyFilter: function () {
            this.filter = "";
            let storeFilters = this.$store.state.filters
            for (let f in storeFilters) {
                if (storeFilters[f].filterType.startsWith("date")
                    || storeFilters[f].filterType.startsWith("time")) {
                    let ranges = storeFilters[f].value.split("_");
                    this.filter += "&" + storeFilters[f].filterType + "__gt=" +␣
↪ranges[0];
                    this.filter += "&" + storeFilters[f].filterType + "__lt=" +␣
↪ranges[1];
                } else {
                    this.filter += "&" + storeFilters[f].filterType + "=" +␣
↪storeFilters[f].value;
                }
            }
            this.currentPage = 1;
            this.loadEntries();
        },
        filterStudent: function (matricule) {
            this.showFilters = true;
            this.$store.commit('addFilter',
                {filterType: 'student_id', tag: matricule, value: matricule}
            );
            this.applyFilter()
        },
        loadEntries: function () {
            // Get current absences.
            axios.get('/student_absence/api/student_absence/?page=' + this.
↪currentPage + this.filter + this.ordering)
            .then(response => {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                this.entriesCount = response.data.count;
                this.entries = response.data.results;
                // Everything is ready, hide the loading icon and show the content.
                this.loaded = true;
            });
        },
        ...
```

En plus du filtre, un système de pagination a été ajouté, d'où la définition des variables `currentPage` et `entriesCount` ainsi que la méthode `changePage`. Pour le filtre en lui-même, deux variables ont été ajoutées, `filter` qui représente la chaîne finale à rajouter à la requête et `showFilters` qui indique la visibilité du composant `Filters`. Quant aux méthodes, `applyFilter` récupère du gestionnaire d'état les filtres appliqués, produit la variable `filter` et recharge les entrées à afficher. `filterStudent`, lui, va juste ajouter un filtre manuellement.

Normalement, si l'on clique sur le nom de l'étudiant dans une entrée, le filtre matricule devrait automatiquement être ajouté et le composant filtre affiché.

Passons maintenant à notre dernier composant, l'ajout d'une entrée. Pour cela, créons un nouveau composant, `assets/student_absence/addStudentModal.vue`, qui proposera à l'utilisateur d'ajouter/modifier une absence :

```
<template>
<div>
    <b-modal size="lg" title="Nouvelle absence"
        ok-title="Soumettre" cancel-title="Annuler"
        ref="addStudentModal"
        :ok-disabled="!student.matricule || (!form.morning && !form.afternoon)"
        @ok="addAbsence" @hidden="resetAbsence"
        >
        <b-form>
            <b-form-row>
                <b-col sm="8">
                    <b-form-group label="Étudiant :" label-for="input-student" :state=
↪"inputStates.student">
                        <multiselect id="input-name"
                            :internal-search="false"
                            :options="studentOptions"
                            @search-change="getStudentOptions"
                            :loading="studentLoading"
                            placeholder="Rechercher un étudiant..."
                            select-label=""
                            selected-label="Sélectionné"
                            deselect-label=""
                            label="display"
                            track-by="matricule"
                            v-model="student"
                            >
                            <span slot="noResult">Aucune personne trouvée.</span>
                            <span slot="noOptions"></span>

                        </multiselect>
                        <span slot="invalid-feedback">{{ errorMsg('student_id') }}</
↪span>
                    </b-form-group>
                </b-col>
                <b-col sm="4">
                    <b-form-group label="Matricule :" label-for="input-matricule">
```

(suite sur la page suivante)

(suite de la page précédente)

```
                        <b-form-input id="input-matricule" type="text" v-model=
↪"student.matricule" readonly></b-form-input>
                    </b-form-group>
                </b-col>
            </b-form-row>
            <b-form-row class="mt-4">
                <b-col>
                    <b-form-row>
                        <b-form-group label="À partir du" :state="inputStates.date_
↪absence_start">
                            <input type="date" v-model="form.date_absence_start" :max=
↪"form.date_absence_end"/>
                            <span slot="invalid-feedback">{{ errorMsg('date_absence_
↪start') }}</span>
                        </b-form-group>
                    </b-form-row>
                </b-col>
                <b-col>
                    <b-form-row>
                        <b-form-group label="Jusqu'au" :state="inputStates.date_
↪absence_end">
                            <input type="date" v-model="form.date_absence_end" :min=
↪"form.date_absence_start"/>
                            <span slot="invalid-feedback">{{ errorMsg('date_absence_
↪end') }}</span>
                        </b-form-group>
                    </b-form-row>
                </b-col>
            </b-form-row>
            <b-form-row>
                <b-form-group label="Matin/Après-midi :">
                        <b-form-checkbox v-model="form.morning">
                            Matin
                        </b-form-checkbox>
                        <b-form-checkbox v-model="form.afternoon">
                            Après-midi
                        </b-form-checkbox>
                </b-form-group>
            </b-form-row>
        </b-form>
    </b-modal>
</div>
</template>

<script>
import Multiselect from 'vue-multiselect'
import 'vue-multiselect/dist/vue-multiselect.min.css'

import axios from 'axios';
window.axios = axios;
window.axios.defaults.baseURL = window.location.origin; // In order to have httpS.

export default {
    props: ['entry'],
    data: function () {
        return {
            form: {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                student_id: null,
                date_absence_start: null,
                date_absence_end: null,
                morning: true,
                afternoon: true,
            },
            student: {matricule: null},
            studentOptions: [],
            studentLoading: false,
            inputStates: {
                student: null,
                date_absence_start: null,
                date_absence_end: null,
            },
            errors: {},
            searchId: -1,
        }
    },
    watch: {
        'form.date_absence_start': function (date) {
            if (this.form.date_absence_end === null) this.form.date_absence_end =␣
↪date;
        },
        entry: function (entry, oldEntry) {
            this.setEntry(entry);
        },
        errors: function (newErrors, oldErrors) {
            let inputs = Object.keys(this.inputStates);
            for (let u in inputs) {
                if (inputs[u] in newErrors) {
                    this.inputStates[inputs[u]] = newErrors[inputs[u]].length == 0;
                } else {
                    this.inputStates[inputs[u]] = null;
                }
            }
        },
    },
    methods: {
        show: function () {
            this.$refs.addStudentModal.show();
        },
        hide: function () {
            this.$refs.addStudentModal.hide();
        },
        resetAbsence: function () {
            this.$emit('reset');

            this.form = {
                student_id: null,
                date_absence_start: null,
                date_absence_end: null,
                morning: true,
                afternoon: true,
            };
            this.student =  {matricule: null};
        },
        setEntry: function (entry) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            if (entry) {
                this.student = entry.student;
                this.form = {
                    student_id: entry.student.matricule,
                    date_absence_start: entry.date_absence_start,
                    date_absence_end: entry.date_absence_end,
                    morning: entry.morning,
                    afternoon: entry.afternoon,
                    id: entry.id,
                }
            } else {
                this.resetAbsence();
            }
        },
        addAbsence: function (evt) {
            // Prevent form to be sent.
            evt.preventDefault();

            this.form.student_id = this.student.matricule;

            let modal = this;
            const token = { xsrfCookieName: 'csrftoken', xsrfHeaderName: 'X-CSRFToken
↪'};
            let path = '/student_absence/api/student_absence/';
            if (this.entry) path += this.entry.id + '/'

            const send = this.entry ? axios.put(path, this.form, token) : axios.
↪post(path, this.form, token);
            send.then(response => {
                this.hide();
                this.errors = {};
                this.$emit('update');
            }).catch(function (error) {
                modal.errors = error.response.data;
            });

            this.entry = null;
        },
        getStudentOptions: function (query) {
            let app = this;
            this.searchId += 1;
            let currentSearch = this.searchId;
            this.studentLoading = true;

            const token = { xsrfCookieName: 'csrftoken', xsrfHeaderName: 'X-CSRFToken
↪'};
            const data = {
                query: query,
                teachings: this.$store.state.settings.teachings,
                people: 'student',
                check_access: false,
            };
            axios.post('/annuaire/api/people/', data, token)
            .then(response => {
                // Avoid that a previous search overwrites a faster following search␣
↪results.
                if (this.searchId !== currentSearch)
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    return;

                const options = response.data.map(p => {
                    // Format entries.
                    let entry = {display: p.last_name + " " + p.first_name,␣
↪matricule: p.matricule};
                    // It's a student.
                    entry.display += " " + p.classe.year + p.classe.letter.
↪toUpperCase();
                    entry.display += " - " + p.teaching.display_name;
                    return entry;
                });
                this.studentLoading = false;
                this.studentOptions = options;
            })
            .catch(function (error) {
                alert(error);
                app.studentLoading = false;
            });
        },
        errorMsg(err) {
            if (err in this.errors) {
                return this.errors[err][0];
            } else {
                return "";
            }
        },
    },
    components: {Multiselect},
    mounted: function () {
        if (this.entry) this.setEntry(this.entry);

        this.show();
    },
}
</script>
```

Il y a pas mal de chose à dire concernant ce composant. De manière générale, il est composé de composants venant de *Bootstrap-vue* à l'exception de `multiselect <https://vue-multiselect.js.org/>`__ qui propose un champ de recherche et de sélection modulable. Si nous y regardons de plus près, la propriété `@search-change` indique quelle méthode est appelée lorsqu'une recherche est effectué, dans notre cas `getStudentOptions`. Celle-ci fait un appel à notre API `/annuaire/api/people/` puis formate la réponse reçue et assigne le résultat à la variable `studentOptions` que le composant *multiselect* va utiliser.

Autre particularité, la gestion des erreurs. En effet, s'il manque une donnée ou que l'une d'entre-elles est mal formaté/incorrecte, le *sérialiseur* de notre API va retourner une erreur et indiquer de quelle type elle est. Nous pouvons voir dans la méthode `addAbsence`, qu'en cas d'erreur le retour est assigné à la variable `errors` du composant :

```
let modal = this;

send.then(response => {
    this.hide();
    this.errors = {};
    this.$emit('update');
}).catch(function (error) {
    modal.errors = error.response.data;
});
```

Du côté de l'interface utilisateur, chaque `input` possède une propriété `state`, indiquant si le champ possède une erreur (trois valeurs possible, `null`, `true`, `false`). Vue permet de réagir dès qu'une variable change, dès qu'une erreurs est détectée dans le `watch` du composant, l'état est mis à jour de manière appropriée. La méthode `errorMsg()` quant à elle, permettra d'afficher la bonne erreur pour l'*input* correspondant.

De manière similaire, si la *props* `entry` est modifiée, par exemple lorsque l'utilisateur veut modifier une entrée, le formulaire sera pré-rempli ou vidé après la création/modification d'une entrée afin de préparer le prochain ajout/modification.

Il ne nous reste plus qu'à intégrer ce *modal* à notre instance Vue principale :

```
<template>
    <div>
        <div class="loading" v-if="!loaded"></div>
        <app-menu :menu-info="menuInfo"></app-menu>
        <b-container v-if="loaded">
            <b-row>
                <h2>Absence des élèves</h2>
            </b-row>
            <b-row>
                <b-col>
                    <b-form-group>
                        <div>
                            <b-btn variant="primary" @click="openDynamicModal('add-
↪student-modal')">
                                <icon name="plus" scale="1" class="align-middle"></
↪icon>
                                Nouvelle absence
                            </b-btn>
                            <b-btn variant="outline-secondary" v-b-toggle.filters>
                                <icon name="search" scale="1"></icon>
                                Ajouter des filtres
                            </b-btn>
                        </div>
                    </b-form-group>
                </b-col>
            </b-row>
            <b-row>
                <b-col>
                    <b-collapse id="filters" v-model="showFilters">
                        <b-card>
                            <filters app="student_absence" model="student_absence"␣
↪ref="filters" @update="applyFilter"></filters>
                        </b-card>
                    </b-collapse>
                </b-col>
            </b-row>
            <b-pagination class="mt-1" :total-rows="entriesCount" v-model="currentPage
↪" @change="changePage" :per-page="20">
            </b-pagination>
            <b-row>
                <b-col>
                    <student-absence-entry
                    v-for="(entry, index) in entries"
                    v-bind:key="entry.id"
                    v-bind:row-data="entry"
                    @delete="askDelete(entry)"
                    @edit="editEntry(index)"
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    @filterStudent="filterStudent($event)"
                        >
                    </student-absence-entry>
                </b-col>
            </b-row>
        </b-container>
        <component
            v-bind:is="currentModal" ref="dynamicModal"
            :entry="currentEntry"
            @update="loadEntries"
            @reset="currentEntry = null">
        </component>
        <b-modal ref="deleteModal" cancel-title="Annuler" hide-header centered
            @ok="deleteEntry" @cancel="currentEntry = null">
            Êtes-vous sûr de vouloir supprimer définitivement cette entrée ?
        </b-modal>
    </div>
</template>

<script>
import Vue from 'vue';
import BootstrapVue from 'bootstrap-vue'
Vue.use(BootstrapVue);

import 'vue-awesome/icons'
import Icon from 'vue-awesome/components/Icon.vue'
Vue.component('icon', Icon);

import axios from 'axios';

import Filters from '../common/filters.vue'
import Menu from '../common/menu.vue'

import StudentAbsenceEntry from './studentAbsenceEntry.vue'
import AddStudentModal from './addStudentModal.vue'

export default {
    data: function () {
        return {
            menuInfo: {},
            currentPage: 1,
            entriesCount: 0,
            entries: [],
            filter: '',
            ordering: '&ordering=-datetime_creation',
            loaded: false,
            showFilters: false,
            currentModal: '',
            currentEntry: null,
        }
    },
    methods: {
        changePage: function (page) {
            this.currentPage = page;
            this.loadEntries();
            // Move to the top of the page.
            scroll(0, 0);
```

(suite sur la page suivante)

(suite de la page précédente)

```
            return;
        },
        openDynamicModal: function (modal) {
            this.currentModal = modal;
            if ('dynamicModal' in this.$refs) this.$refs.dynamicModal.show();
        },
        applyFilter: function () {
            this.filter = "";
            let storeFilters = this.$store.state.filters
            for (let f in storeFilters) {
                if (storeFilters[f].filterType.startsWith("date")
                    || storeFilters[f].filterType.startsWith("time")) {
                    let ranges = storeFilters[f].value.split("_");
                    this.filter += "&" + storeFilters[f].filterType + "__gt=" +␣
↪ranges[0];
                    this.filter += "&" + storeFilters[f].filterType + "__lt=" +␣
↪ranges[1];
                } else {
                    this.filter += "&" + storeFilters[f].filterType + "=" +␣
↪storeFilters[f].value;
                }
            }
            this.currentPage = 1;
            this.loadEntries();
        },
        filterStudent: function (matricule) {
            this.showFilters = true;
            this.$store.commit('addFilter',
                {filterType: 'student_id', tag: matricule, value: matricule}
            );
            this.applyFilter()
        },
        loadEntries: function () {
            // Get current absences.
            axios.get('/student_absence/api/student_absence/?page=' + this.
↪currentPage + this.filter + this.ordering)
            .then(response => {
                this.entriesCount = response.data.count;
                this.entries = response.data.results;
                // Everything is ready, hide the loading icon and show the content.
                this.loaded = true;
            });
        },
        askDelete: function (entry) {
            this.currentEntry = entry;
            this.$refs.deleteModal.show();
        },
        editEntry: function(index) {
            this.currentEntry = this.entries[index];
            this.openDynamicModal('add-student-modal');
        },
        deleteEntry: function () {
            const token = { xsrfCookieName: 'csrftoken', xsrfHeaderName: 'X-CSRFToken
↪'};
            axios.delete('/student_absence/api/student_absence/' + this.currentEntry.
↪id + '/', token)
            .then(response => {
```

(suite sur la page suivante)

(suite de la page précédente)

```
                this.loadEntries();
            });

            this.currentEntry = null;
        },
    },
    mounted: function () {
        this.menuInfo = menu;

        this.loadEntries();
    },
    components: {
        'filters': Filters,
        'app-menu': Menu,
        'student-absence-entry': StudentAbsenceEntry,
        'add-student-modal': AddStudentModal,
    },
}
</script>

<style>
.loading {
  content: " ";
  display: block;
  position: absolute;
  width: 80px;
  height: 80px;
  background-image: url(/static/img/spin.svg);
  background-size: cover;
  left: 50%;
  top: 50%;
}
</style>
```

L'ajout est somme toute assez simple. Nous importons le composant et nous rajoutons une méthode pour ouvrir le *modal*. Petite particularité, non nécessaire pour notre application, nous avons laissé la possibilité d'ouvrir différents *modals*. En effet, le type de *modal*, et donc de composant, est dynamiquement chargé et affiché grâce à la propriété `currentModal` qui retient le composant courant en mémoire. Nous aurions très bien pu utiliser ce mécanisme pour la demande de suppression d'une entrée.

Ceci clôture cette première approche de la création d'une application. Pour aller plus loin, un coup d'œil à l'application *infirmerie* ou, encore plus complexe, *dossier_eleve* qui donnent une bonne vision de ce que peut être en condition réelle une application dans HappySchool. N'hésitez surtout pas à apporter des corrections à ces explications et à HappySchool de manière générale.

CHAPITRE 5

Indices and tables

— genindex
— modindex
— search