Ubid - Documentation technique Livrable 2

Introduction

Le groupe n'ayant pas eu le temps de finaliser la partie front-end, avec les connections Angular -> API, nous vous livrons deux versions de notre travail.

Vous trouverez dans le dossier "rendu1" la dernière API fonctionnelle, sans que le Front ne soit implémenté malheureusement.

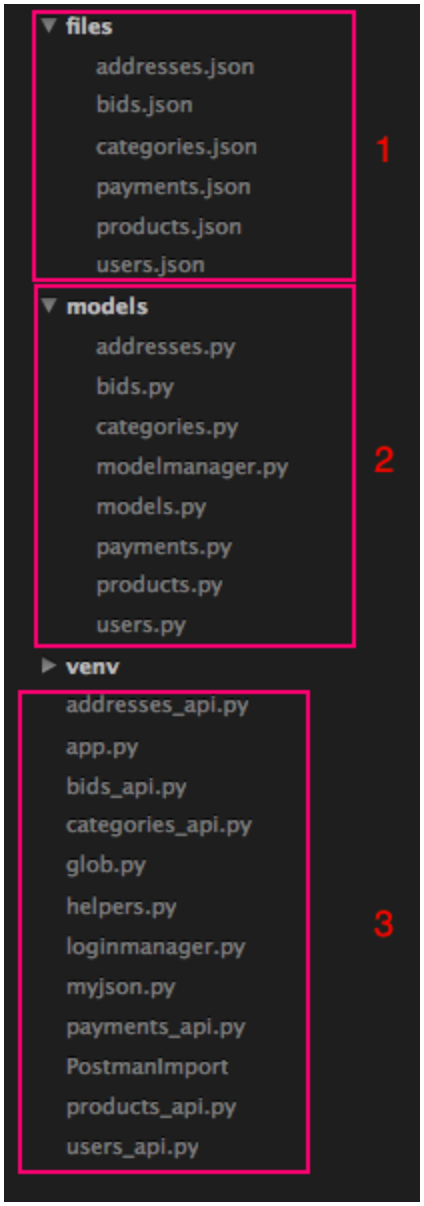
Vous trouverez dans le dossier "rendu2" une version de l'API plus ancienne, avec le front qui est fonctionnel.

Des documentations sont disponibles dans chacun de ces dossiers, nommées "documentation.pdf"

Dans cette documentation, nous nous attarderons sur l'API qui est disponible dans le dossier Rendu1

Architecture

Pour faire notre API, nous avons choisi de coder notre propre framework.



Nous pouvons voir que dans cette structure de fichier, nous avons séparé en trois parties :

1. Files : Les fichiers qui sont considérés comme "base de donnée"
2. Models : Les modèles de nos objets
3. ROOT_FOLDER : Les API qui permettent de faire la liaison entre nos modèles, notre base de donnée, et notre API.
4.

Les modèles

Nous avons choisi de recoder notre propre framework pour pouvoir avancer efficacement et se concentrer sur la tache principale : avoir une API fonctionnelle, sans avoir à tenir compte de la complexité de Flask ou de Python. Notre objectif était d'avoir le moins de code répliqué possible, et d'être efficace rapidement.

Supposons que nous voulions déclarer un modèle "User".

1. Créer un fichier models/users.py **Important : Les modèles sont TOUJOURS au pluriel**
2. Copier le contenu suivant dans ce fichier :

```
def __init__(self, json=None):
                self.fields = []
                self.unique = []
                self.mandatory = []
                self.intern_fields = []
                self.editable_fields = []
                self.has_many = []
                self.__password = None
                Models.__init__(self, json)
```

3. Déclarer le modèle à Flask : Dans le fichier app.py, rajouter ces lignes :

```
from users import Users
import users_api
app.register_blueprint(users_api.usr, url_prefix="/users")
```

4. Créer un fichier **users_api.py** dans le ROOT_FOLDER
5. Créer un fichier base de donnée **files/users.json**

Définition des champs d'un modèle

- Un utilisateur va avoir les champs suivants : name, firstname
- Un utilisateur va avoir les champs suivants, et qui sont uniques, plusieurs utilisateurs ne peuvent pas partager ces même champs : email, username
- Un utilisateur va avoir des champs obligatoires : password
- Un utilisateur va avoir des relations avec les modèles suivants : products, bids, addresses, payments

Pour définir ces relations, je vais préciser mon modèle user avec ces informations :

```
self.fields = ['name']
                self.unique = ['email', 'username']
                self.mandatory = ['password']
                self.intern_fields = ['token']
                self.editable_fields = ['name', 'password', 'email']
                self.has_many = ['products', 'bids', 'addresses', 'payments']
```

Je vais également vouloir controler que le mot de passe fait 6 caractères au minimum, et ce de manière automatique. Donc à chaque fois que je vais utiliser le setter de mon password de la manière suivante :
user.password = "abcdef"
je veux que mon mot de passe soit controlé, et si ce qui est entré est inférieur à 6 caractères, je veux qu'une erreur soit soulevé automatiquement; à l'inverse, si le password est bon, je veux qu'il soit automatiquement encodé en Base64. Pour se faire, je précise à mon objet les options suivantes :

```
        def __get_password(self):
                return self.__password

        # When we set the password, we encode it with base64
        def __set_password(self, password):
                if len(password) < 6:
                        abort(make_response('Password too short, 400))
                self.__password = base64.b64encode(password)

        # Define the delete callback
        def __del_password(self):
                del self.__password

        # We define the callback, and say :
        # Hey password field, when I set you like "user.password = abcdef", I want you to call the method "__get_password()" for me !
        password = property(__get_password, __set_password, __del_password, '')
```

Définition des relations d'un modèle

Je sais que mon utilisateur va pouvoir créer plusieurs produits. Je veux donc que sur chaque produit, il y ait un user_id qui soit crée, et que je puisse récupérer la liste de ces produits en faisant
`user.products`

ou à l'inverse
`product.user`

Pour cela, il suffit de définir les relations has_many et belongs_to.
Un utilisateur a plusieurs (has_many) produits
Un produit est rattaché (belongs_to) à un utilisateur

Je vais simplement définir dans mes modèles :

- products.py :
  `self.belongs_to = ['user']`
- users.py :
  `self.has_many = ['products']`

Grâce à ces relations, dans la base de donnée sera crée automatiquement un champ user_id pour les products, et je pourrai accéder à la liste de mes produits depuis mes utilisateurs de manière simple.

Chargement de modèles en mémoire automatiquement

Nous avons implémenté un système d'auto chargement des modèles grâce à des notions en Python.

Ces chargement sont effectués dans les fichiers modelmanager.py et models.py.

modelmanager.py : Récupère la liste des fichiers dans le dossier models, et va les importer automatiquement (Ligne 52)

```
module = importlib.import_module(f)
```

Il va également définir les champs has_many et belongs_to de tout nos modèles. (Ligne 21 à 37).

```
def set_has_many(modele, field):
    setattr(modele, field, Models().getBy(field, modele.__class__.__name__.lower()[:-1] + "_id", modele.id))

def set_belongs_to(modele, field):
    setattr(modele, field, Models().getBy(field + "s", "id", eval("modele." + field + "_id")))

# Define the references between 2 models, the 'belongs_to' and the 'has_many'.
def init_references(models):
    for model_name in models:
        mods = Models().get(model_name)
        for model in mods: # All models in "user"
            if hasattr(model, 'has_many'):
                for field in model.has_many: # All fields in "has_many"
                    set_has_many(model, field)
            if hasattr(model, 'belongs_to'):
                for field in model.belongs_to :
                    set_belongs_to(model, field)
```

models.py : Tout les modèles héritent de cette classe. Cela nous permet d'éviter de dupliquer du code. Cette classe contient toute les méthodes get, set, save, edit, delete pour un objet depuis la base de donnée. Elle contient également quelques méthodes privées utiles pour le bon fonctionnement de nos modèles, entre autre l'enregistrement automatique de nos champs depuis la récupération de la description du modèle, comme pour le modèle users.py par exemple entre les lignes 12 et 17.

```
self.fields = ['name'] #Default fields, can be blank in the request.
                self.unique = ['email', 'username', 'rate', 'rateNumber'] #Unique fields are also mandatory // Rate = note de l'utilisateur & rate_number = nombre de vote (pour le calcul coté client)
                self.mandatory = ['password'] # Mandatory fields.
                self.intern_fields = ['token']
                self.editable_fields = ['name', 'password', 'email']
                self.has_many = ['products', 'bids', 'addresses', 'payments']
                self.__password = None # Special field with a callback
```