SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

TME10. table d'index (partie 1) et gestion d'images (partie 2)

BM. Bui-Xuan

Le but principal du TME est de se préparer au rendu final de l'UE, dont la date butoir est le 30/03/2025. Nous utilisons Django et DjangoRestFramework pour mettre en oeuvre l'accès rapide aux contenus précalculés dans les moteurs de recherche.

- Le tutoriel recommandé est DjangoRestFramework (← cliquer sur les liens en bleu).
- La documentation recommandée est DjangoProject.
- Le code source du canevas de TME est disponible ici.

Les objectifs de la 1e partie de la séance sont:

- Fournir, sur localhost, un clone du service webAPI déjà disponible sur un certain serveur distant.
- Enrichir ce service avec un accès rapide à deux contenus supplémentaires: la liste de produits avec le champs `'sale'` mis à `true`; et la liste des produits avec le champs `'availability'` mis à `true`.
- Mettre à jour de façon régulière les contenus ci-dessus, sur localhost, en fonction des données originales, servies par le serveur distant.

Les objectifs de la 2e partie de la séance sont:

- Etendre le webAPI avec des URIs servant des images, dans un premier temps par une approche monolithique.
- API-zation du monolithe.

Partie 1. WebAPI

Exercice 1. Configurer l'environnement virtuel pour python

Télécharger le fichier canevas du TME et extraire son contenu dans un répertoire de travail. Nous pouvons configurer un environnement virtuel dans ce répertoire par ces commandes sur un terminal:

- `python3 -m venv myTidyVEnv`
- `source myTidyVEnv/bin/activate`
- `pip3 install django djangorestframework requests`

En suite, nous pouvons déployer le serveur du canevas sur localhost:

- `cd TME_webAPI_DAAR/mySearchEngine/`
- `python3 manage.py runserver`

Finalement, nous pouvons visualiser la réponse de cet API (du localhost) sur n'importe quel navigateur:

- `http://127.0.0.1:8000/products/`

Nous pouvons remarquer que la réponse de cet API du localhost est, d'après les lignes 7-11 de son code dans mytig/views.py, uniquement une redirection vers la réponse de l'URL du serveur distant, défini dans mytig/config.py. Sur ce serveur distant, le RestAPI est servi sur les URIs sous le format suivant:

- `.../products/` → liste de tous les produits.
- `.../product/id` → détails du produit avec `pk` égal à `id`. PS: pk est une abréviation de *primary key*.

Vérifier que l'API-cloné sur locahost sert bien les URIs décrits ci-dessus:

- `http://127.0.0.1:8000/products/`
- `http://127.0.0.1:8000/product/16` → huîtres.
- `http://127.0.0.1:8000/product/64` → *HTTP response not found*.

Exercice 2. Clone de shipPoints

Sur le serveur distant, le RestAPI sert également les URIs du format suivant:

- `.../shipPoints/` → liste de tous les points de relais pour la livraison.
- `.../shipPoint/id` → détails du point de relais avec `pk` égal à `id`.

Etendre le code de mytig/views.py afin que le clone sur localhost sert également les URIs décrits ci-dessus.

Exercice 3. Indexer la liste de tous les produits avec le champs `'availability'` mis à `true`

Dans une requête de type `.../product/id`, observer que le produit peut être disponible ou non, indiqué par la valeur du champs `'availability'`. Nous voudrions que l'API-cloné sur localhost serve ces nouveaux URIs:

- `.../availableproducts/` → liste de tous les produits qui sont actuellement disponibles.
- `.../availableproduct/id` → détails du produit avec `pk` égal à `id` dans la liste des produits disponibles.

Implémenter cet évolution. On peut s'inspirer du code déjà disponible dans le canevas pour servir les URIs suivants, qui sont très similaires:

- `.../onsaleproducts/` → liste de tous les produits qui sont actuellement en promotion.
- `.../onsaleproduct/id` → détails du produit avec `pk` égal à `id` dans la liste des produits en promotion.

Il est utile de lire de début de l'Exercice 4 ci-dessous afin d'avoir une vue globale sur les processus de mise à jour de la donnée avant de commencer l'Exercice 3. La liste des fichiers à modifier pour l'Exercice 3 comprend:

- mytig/views.py → similaire à `PromoList`
- mytig/models.py → similaire à `ProduitEnPromotion`
- mytig/serializers.py → similaire à `ProduitEnPromotionSerializer`
- mytig/urls.py → similaire à `onsaleproducts`

NB: afin que la base de donnée sur localhost en tient compte, après l'ajout d'une nouvelle classe à `mytig/models.py` il faut exécuter sur terminal:

- `python3 manage.py makemigrations mytig`
- `python3 manage.py migrate`

Exercice 4. Mise à jour automatique de la donnée

Dans l'API-cloné sur localhost, la liste des `ProduitEnPromotion` peut être mise à jour par appel à une commande Django:

- Usage: `python3 manage.py refreshOnSaleList`
- Code source: mytig/management/commands/refreshOnSaleList.py

Crontab peut alors être utilisé afin d'exécuter périodiquement la commande ci-dessus[1]:

- Shell script: adapter ce script à la situation spécifique de sa machine.
- Pour une mise à jour toutes les minutes (ce qui est pratique pour débugger, mais mauvais pour un usage réel), appeler `'crontab -e'` sur terminal et ajouter la ligne suivante (l'éditeur proposé par la commande, selon votre machine, est la plupart des cas `nano` ou `vi`, dont les deux commandes les plus utiles sont `'i'` pour insert et `':wq'` pour write-and-quit):
- `* * * * * cd ~ && ./mySearchEngineDataRefresh.sh`
- NB: nous pouvons vérifier la liste de tous les cron actuellement en exécution par la commande `'crontab -l'` du terminal.

Inspirer de l'exemple ci-dessus de `refreshOnSaleList` et implémenter une mise à jour automatique de la donnée afin de rafraîchir la liste des produits disponibles implémentée dans l'Exercice 3.

Exercice 5 (Bonus). Préparer l'API backend pour le projet `mygutenberg`

Le webapp décrit ci-dessus, `mytig`, a été créé par l'exécution de la commande suivante depuis le répertoire racine du code Django de `mySearchEngine` (où l'on trouve, entre autre, le fichier `manage.py`):

- `python3 manage.py startapp mytig`

Nous voudrions maintenant créer l'API backend pour notre future projet `mygutenberg`, qui sera un clone et une extension comme moteur de recherche bibliothécaire de l'API distant suivant:

[1] Si votre SI comprend de nombreux équipements, une meilleure gestion est offerte par des outils comme Jenkins. Essentiellement, un crontab est spécifique à une machine (et à un utilisateur), ainsi, l'administrateur système ne sera pas enclin à aller visiter le parc de machines serveurs du SI et vérifier chaque crontab lors de chaque bug de surcharge. Ceci dit, pour les besoins très minimalistes de notre API-cloné sur localhost, une utilisation de crontab est largement suffisant.

- `https://www.gutenberg.org/ebooks/` → liste de tous les livres.
- `https://www.gutenberg.org/ebooks/id` → détails du livre avec `pk` égale à `id`.

En particulier, nous voudrions que le nouveau API `mygutenberg` sert les URIs suivants:

- `.../books/` → liste de tous les livres.
- `.../book/id` → détails du livre avec `pk` égale à `id`.
- `.../frenchbooks/` → liste de tous les livres en Français.
- `.../frenchbook/id` → détails du livre avec `pk` égal à `id` dans la liste des livres en Français.
- `.../englishbooks/` → liste de tous les livres en Anglais.
- `.../englishbook/id` → détails du livre avec `pk` égal à `id` dans la liste des livres en Anglais.

Commencer un nouveau webapp avec la commande suivante, puis implémenter un API sur localhost servant les URIs décrit ci-dessus:

- `python3 manage.py startapp mygutenberg`

Partie 2. Monolithe et microservices

Le stockage des contenus multimédia a depuis toujours suscité maintes discussions, aussi passionnées qu'aptes à des débordements incontrôlés... Nous allons adopter la stratégie suivante, sans polémiquer.

- Nous enregistrons les emplacements de fichiers multimédia dans la base de donnée.
- Nous enregistrons les fichiers multimédia dans le système de gestion de fichiers (sur le disque local ou sur n'importe quel serveur distant: on peut notamment noter que les deux canevas du TME délèguent ce stockage à des serveurs distants).

Dans les deux canevas de la deuxième partie du TME10, les emplacements de fichiers multimédia sont manipulés sous la forme des URLs. Cependant, ces derniers ne sont pas stocké proprement dans la base de donnée gérée par DjangoRestFramework sur localhost. Ils sont représentés par un bouchon qui se trouve dans `mytig/config.py`.

Exercice 6. Configurer l'environnement virtuel pour python

Attention: *Nous supposons dans ce qui suit que l'Exercice 1 du TME10 est terminé.*

Télécharger le premier canevas pour la partie 2 du TME10, extraire son contenu, et faire la comparaison avec celui du canevas de la 1e partie du TME10. Il y a modification sur uniquement trois fichiers: `mytig/config.py`, `mytig/urls.py`, and `mytig/views.py`. Dépendant de sa machine, on peut avoir besoin d'un paquet python supplémentaire:

- `source myTidyVEnv/bin/activate`
- `pip3 install secrets`

Déployer le canevas sur localhost:

- `cd TME_webAPI_DAAR/mySearchEngine/`

- `python3 manage.py runserver`

Observer la réponse de l'API sur n'importe quel navigateur:

- `http://127.0.0.1:8000/product/1/image/`
- `http://127.0.0.1:8000/product/1/image/1/`
- `http://127.0.0.1:8000/product/1/image/2/`
- `http://127.0.0.1:8000/product/2/image/2/`

Pourquoi les deux dernières requêtes retournent la même réponse? Quelles sont les lignes dans `mytig/views.py` reponsables de ce comportement?

Ceci dit, notre objectif pour l'instant n'est pas le couplage parfait entre les URLs et les produits présents dans l'API. Nous voudrions dans un premier temps focaliser sur un point important dans l'Exercice 7 suivant.

Exercice 7. Extraire la gestion des fichiers multimédia du premier canevas

Après la 1e partie du TME10, ainsi que l'Exercice 6 ci-dessus, nous avons vu grandir la webapp `mytig`, qui inclut maintenant les trois fonctionnalités suivantes:

- Redirection des services `.../products/` et `.../shipPoints/` vers le serveur distant original.
- Nouveaux services `.../availableproducts/` et `.../onsaleproducts/`
- Nouveaux services `.../product/<id>/image/` et `.../product/<id>/image/<image_id>/`

Si la webapp n'excède pas 200 lignes fonctionnelles[2] (vive DRF!), il n'est pas des *best practices* des SI de cumuler multiples services dans une même webapp. Afin de prévenir `mytig` de devenir un futur monstre monolithique, nous allons extraire tout service relatif aux fichiers multimédia dans une nouvelle webapp, appelée `myImageBank`, qui serviront les URIs suivants:

- `.../myImage/random/`
- `.../myImage/<image_id>/`

Ensuite, `mytig` de la 1e partie du TME10 pourra juste mettre à jour des pointeurs vers la nouvelle webapp `myImageBank` pour toute requête relative au contenu multimédia. Nous rappelons qu'une nouvelle webapp peut être créée dans Django par la commande suivante, exécutée dans le répertoire racine (celui contenant le fichier `manage.py`):

- `python3 manage.py startapp myImageBank`

Une façon d'implémenter cette extraction est disponible dans ce second canevas pour la 2e partie du TME10. Télécharger, extraire, et déployer le nouveau canevas sur localhost

Extendre le canevas afin que la webapp `myImageBank` serialise tous les URLs du bouchon `myImageBank/config.py` dans une nouvelle classe dans `myImageBank/models.py`, avec du code approprié dans `myImageBank/serializers.py`.

On veillera à ne pas oublier d'étendre `myImageBank/views.py` afin de prendre en compte la nouvelle classe dans le `myImageBank/models.py`.

Exercice 8. Personnaliser `myImageBank`

Nous observons que les trois requêtes suivantes donnent la même réponse:

[2] `find mySearchEngine/mytig -iname *py | grep -v migrations | xargs wc -l` retourne 195 lignes au total.

- `.../product/1/image/2/`
- `.../product/2/image/2/`
- `.../product/1664/image/2/`

ce qui est pourtant assez moyen. Ceci est d'autant plus vrai en observant que la requête `.../product/1664/` donne une réponse de type `{'detail: 'Not found'}`.

Créer une nouvelle webapp, appelée `myImageEngine`, servant les URIs suivants:

- `.../myImageFromString/<str:name>/` → l'adresse URL d'une image correspondant à `name` comme description. Pour des raisons pratiques, on peut penser à remplacer dans `name` tout caractère d'espace par un caractère spécial de type *"magic number"*.

Il est à noter que le moteur de recherche d'image le plus simpliste consiste en un stockage statique dans la base de donnée locale. Par exemple avec:

- `<int:pk><str:name><str:url>`
- où l'image présente à l'adresse `url` correspond à la description `name`.
- NB: une même description `name` peut correspondre à plusieurs `url`; en revanche, une adresse `url` correspond idéalement à une unique description `name`[3].

Des exemples de moteurs de recherche d'images plus sophistiqués comprennent: Google Images, Flickr, Giphy.

Exercice 9. Mise à jour automatique de la donnée

S'inspirer de l'Exercice 4 du TME10 et implémenter une mise à jour automatique de la donnée locale pour `myImageEngine`. En particulier, `myImageEngine` doit donner une réponse positive à toute nouvelle requête `.../myImageFromString/<newName>` dès que le serveur distant original sert un nouveau `.../product/<id>/` dont le champs `'name'` est égal à `newName`.

Exercice 10 (Bonus). Continuer `mygutenberg`

Le stockage des fichiers multimédia du site The Gutenberg Project est très bien organisé! Nous pouvons notamment retrouver tout contenu relatif à `.../ebooks/56667/` sur l'emplacement suivant:

- `http://gutenberg.org/files/56667/56667-h/images/`
- en particulier, nous avons une image de la page de garde, un certain joli plan de ville, voire même des mesures réelles des mûrs d'une forteresse en cas d'absolu besoin...

Etendre la table d'index de la webapp `myImageEngine` afin qu'elle sert également des futures requêtes d'image depuis la webapp `mygutenberg`. Etendre `mygutenberg` afin de servir également les URIs suivants:

- `.../book/<id>/coverImage/` → URL correspondant à la page de garde du livre avec `pk` égal à `id`.
- `.../book/<id>/image/` → URL correspondant à une image aléatoire prise dans le livre avec `pk` égal à `id`.
- `.../book/<id>/image/<image_id>/` → URL correspondant à une image spécifique, d'identifiant `image_id`, prise dans le livre avec `pk` égal à `id`.

[3] Cependant, plusieurs catégories d'huîtres peuvent toujours avoir la même image avatar, par exemple.