

Bases de Données Avancées,

Rapport Phase 4 :

Projet final

Guilhem ALBALADEJO
Info 4A
Polytech Marseille

TABLE DES MATIÈRES

I	Introduction	1
II	Création et configuration	1
II-A	Installation	1
II-B	Création	1
II-C	Configuration	1
III	Architecture de l'application	1
III-A	Architecture django	1
III-B	Navigation	2
IV	Services	2
IV-A	Service SQLite	2
IV-B	Services mongoDB	3
V	Views et logique métier	3
VI	Interface utilisateur	3
VI-A	base.html	3
VI-B	Composants	4
VI-B1	Movie card	4
VI-B2	Movie page	4
VI-C	Templates	4
VI-C1	Home	4
VI-C2	movies	4
VI-C3	movies/<id>	5
VI-C4	search	5
VI-C5	stats	5
VII	Difficultés rencontrées	6
VIII	Conclusion	6

Bases de Données Avancées,

Rapport Phase 4 :

Projet final

I. INTRODUCTION

Nous avons dans les phases précédentes entièrement créé et configuré nos bases de données avec notre base SQLite et notre cluster de trois replica sets mongoDB. Cette phase finale visera à mettre à profit nos données en les connectant à une application django qui permettra à ses utilisateurs de naviguer dans notre répertoire de films.

Le projet est disponible au lien github : <https://github.com/guilhem-alb/Cineexplorer>.

II. CRÉATION ET CONFIGURATION

A. Installation

Pour notre application, on utilisera :

- Le framework python django pour le développement de notre application.
- La librairie tailwind-css pour le style de notre interface.
- Le package python pymongo pour interagir avec notre base mongoDB, en effet django ne possède aucun engine mongoDB par défaut.

On crée un environnement virtuel python à la racine de notre projet, dans lequel on installe les packages nécessaires et leurs dépendances. On les copie ensuite dans un fichier requirements.txt à la racine de notre projet aussi à l'aide la commande pip freeze pour que n'importe qui reprenant le projet puisse installer les packages sans avoir de problème de compatibilité.

B. Création

Une fois nos packages installé, on peut créer le projet django dans le répertoire courant à l'aide de la commande

```
django-admin startproject config .
```

Puis pour créer notre application, que l'on appellera movies :

```
python manage.py startapp movies
```

Dans config/settings.py on ajoute notre application movies à la liste "INSTALLED_APPS".

Il nous faut à présent créer l'application tailwind-css à l'aide de django-tailwind. Pour cela on ajoute à "INSTALLED_APPS" l'application 'tailwind' et on lance la commande :

```
python manage.py tailwind init
```

Cette commande demande une méthode d'installation de tailwind-css. Pour cette application, on choisit la méthode "npm-based", mais il faut tout de même noter qu'elle requiert

d'avoir nodejs et npm d'installé sur la machine pour fonctionner. On nous demande également le nom de l'application tailwind-css que l'on veut créer. Un nom commun est 'theme' c'est donc celui choisi pour ce projet.

Une fois les dépendances installées, on ajoute notre application 'theme' dans INSTALLED_APPS et on ajoute dans settings.py la variable :

```
TAILWIND_APP_NAME = 'theme'
```

Finalement on installe les dépendances de tailwind avec :

```
python manage.py tailwind install
```

C. Configuration

On ajoute nos bases de données dans la liste "DATABASES" de settings.py pour pouvoir y accéder globalement depuis le reste de notre application :

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'data' / 'imdb.db',
    },
    'mongo': {
        'HOST': 'mongodb://localhost:27017',
        'NAME': 'imdb'
    }
}
```

Dans le fichier urls.py de l'application 'config', on redirige toutes les urls vers notre application 'movies' :

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

Finalement dans notre application 'theme', on supprime le template "base.html" puisque l'on créera notre propre template de base pour notre application.

III. ARCHITECTURE DE L'APPLICATION

A. Architecture django

Django est un framework suivant l'architecture "model-view-templates" :

- Les "models" correspondent à une abstraction des données de l'application, ils permettent d'accéder aux données via l'ORM (Object-Relational-Mapping) de django. Cependant, l'intégration de mongoDB à l'ORM reste encore limitée, et la plupart de nos requêtes étant déjà écrite, on se passera des models dans le contexte de notre application. On utilisera à la place des services personnalisés permettant à notre application de communiquer avec nos bases de données.
- Les "views" correspondent à la logique métier de l'application. Elles sont chacune associées à une ou plusieurs urls et sont appelées à chaque accès à cette url. Elles seront chargés dans notre application de récupérer les données depuis nos bases de données à l'aide de nos services, puis de les injecter dans les templates avant de les envoyer au client.
- Les "templates" sont l'interface de notre application. Ils prennent la forme de fichier html dans lesquels on peut injecter des données et de la logique python légère afin de modifier dynamiquement l'affichage en fonction des données passées par les views.

D'un point de vue conceptuel, l'architecture de notre application prend la forme suivante :

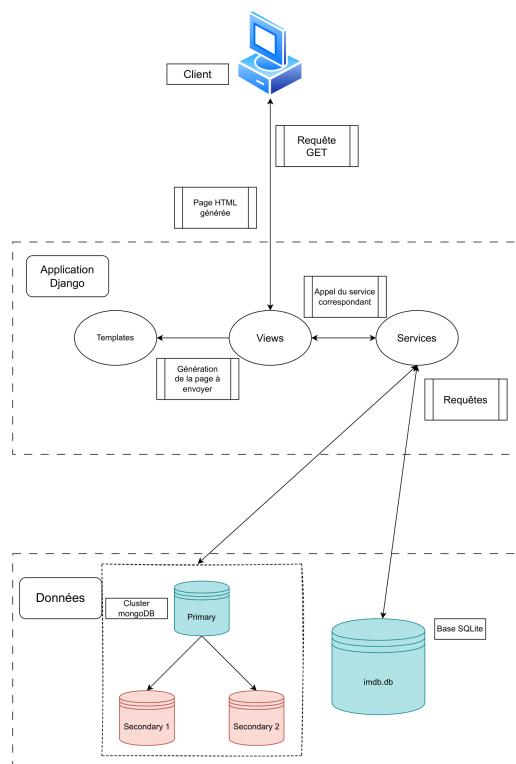


FIGURE 1. Architecture de l'application

B. Navigation

- Notre application sera composée de 5 pages différentes :
- '/' la page d'accueil.
 - '/movies' une page contenant une liste de films filtrable sous la forme de pages de 20 films.
 - '/movies/<id>' une description précise d'un film.
 - '/search' une page de recherche de film par nom ou par personne.
 - '/stats' une page contenant des statistiques sur nos données.

On associe d'ores et déjà ces urls à des views dans le fichier urls.py de notre application :

```

urlpatterns = [
    path('', views.home_view),
    path('movies/<str:movie_id>',
         views.movie_complete_view),
    path('movies', views.pages_view),
    path('search', views.search_view),
    path('stats', views.stats_view),
]
  
```

La navigation s'effectuera par le biais d'une barre de navigation permettant d'accéder aux pages 'home', 'movies', 'stats', et 'search' en effectuant une recherche directement. Pour accéder au détail d'un film, l'utilisateur pourra cliquer sur l'image du film n'importe où sur le site pour être redirigé vers l'url '/movies/<id>' correspondante.

IV. SERVICES

Dans le dossier de notre application movies, on crée un dossier "services" destiné à contenir nos requêtes SQL et mongoDB. Afin de garantir une séparation des responsabilité claire, ce sera lui et lui seul qui sera chargé de communiquer avec nos bases de données. Les autres parties de l'application devront faire appel à ce service pour récupérer les données dont elles ont besoin.

A. Service SQLite

La plupart de nos requêtes s'effectueront envers notre base SQLite, en effet c'est la plus efficace pour gérer les nombreuses requêtes relationnelles dont a besoin notre application.

Dans le fichier sqlite_service.py, on définit les fonctions suivantes :

- `get_top_N_movies(N : int)` renvoyant les N films ayant les meilleures notes et plus de 100000 votants.
- `get_basic_stats()` renvoyant le total de films, acteurs, et directeurs sur l'ensemble de notre base de données.
- `get_random_movies(N : int)` renvoyant N films au hasard.
- `get_film_list(page : int,filters : list,sort_on : Literal["title", "year", "note"],order_by : Literal["ASC", "DESC"])` filtrant les films selon filters, et les triant selon sort_on et order_by, avant de renvoyer les résultats correspondant à la page passée en argument.
- `get_film_list_size(filters : list)` renvoyant le nombre de résultats obtenus en appliquant les filtres.

- get_genre_list() renvoyant une liste de tous les genres présents dans notre base de données.
- search_movies_from_title(title : str, page : int) renvoyant les résultats de la recherche par titre avec LIKE sous forme de page.
- get_list_from_title_size(title : str) renvoyant le nombre de résultats obtenus en cherchant avec le titre.
- search_movies_from_person(person_name : str, page : int) renvoyant les résultats de la recherche par nom de personne avec LIKE sous forme de page.
- get_list_from_person_size(person_name : str) renvoyant le nombre de résultats obtenus en cherchant avec le nom de personne.
- get_movies_count_by_genre() renvoyant le nombre de film pour chaque genre.
- get_movies_count_by_decade() renvoyant le nombre de films par décennie.
- get_ratings_distribution(bin_size : int) renvoyant la distribution de notes, avec un découpage selon bin_size pour pouvoir les formater en histogramme.
- get_top_N_prolific_actors(N : int) renvoyant les N acteurs les plus prolifiques.

Afin de pouvoir passer plus simplement les données aux templates dans les views, on les convertit à l'intérieur des différentes fonctions en dicts avant de les renvoyer.

B. Services mongoDB

Nous avons besoin de moins de fonction mongoDB que SQLite, en effet cette base de données nous servira surtout à récupérer les documents pré-aggrégés de la collection movies_complete, en effet, si l'on tenait à récupérer des documents des autres collections, alors la base SQLite serait plus efficace, puisque cela impliquerait d'effectuer des requêtes relationnelles.

Dans le fichier mongo_service.py, on définit les fonctions suivantes :

- get_movie_complete(movie_id : str) renvoyant le document de movies_complete ayant pour identifiant celui passé en argument.
- get_movie_and_title(movie_id : str) renvoyant uniquement l'identifiant et le titre du film.
- get_rd_movies_from_directors(directors : list, N : int, original_id : str) renvoyant N films au hasard dirigé par les directeurs passés en argument, et omettant le film passé en argument.
- get_rd_movies_from_genres(genres : list, N : int, original_id : str) renvoyant N films au hasard des genres passés en argument, et omettant le film passé en argument.

V. VIEWS ET LOGIQUE MÉTIER

À présent que nous avons défini les services mongoDB et SQLite dont a besoin notre application, nous pouvons définir les views qui utiliseront ces services. Dans la partie III-B Navigation, nous avons déjà mappé une view à chaque url, il nous reste donc seulement à les remplir.

Pour cela, on utilise la fonction render de django.shortcuts pour renvoyer dans chaque view la page html correspondant comme suit :

```
return render(request,
    → "movies/template.html", context)
```

où request correspond à l'url qui a amené à cette view, "template.html" le template qui a servi à la génération de la page, et context les données que l'on passe au template.

Dans les views qui nécessitent de lire des variables depuis l'url (ex : q=movie), on peut simplement utiliser :

```
request.GET.get('variable_name',
    → default_value)
```

pour récupérer la valeur de la variable et lui donner une valeur par défaut.

Il nous suffit ensuite de passer cette valeur à nos services pour récupérer les données correspondant à la requête.

Pour injecter les données dans nos templates, il nous faut juste nous assurer que le contexte que l'on leur passe ait bien la forme d'un dict, puisque notre template verra chacun des couples clé-valeur du dict comme des variables.

VI. INTERFACE UTILISATEUR

A. base.html

Un template django correspond à un fichier html dans lequel on peut écrire de la logique python simple, comme des structures de contrôle : if, for etc... et où l'on peut injecter la valeur des variables passées pendant le render.

On commence par définir notre template de base, dont tous les autres hériteront.

Dans celui-ci, on défini :

- Le head de base de notre application : balises meta, lien vers la feuille de style générée par tailwind etc... et un bloc pour que chaque héritier puisse rajouter des informations dans le head
- La barre de navigation de notre application, commune à toutes les pages de notre site.
- Un block "content" où les pages filles écriront leur contenu.
- Un block "extra_js" destiné à accueillir les scripts supplémentaires.

Pour notre barre de navigation, on la rend responsive en profitant des fonctionnalités de tailwind, qui est une librairie "mobile-first", en exploitant les classes dépendantes de la taille de l'écran comme :

- sm :
- md :
- lg :

pour appliquer certains styles conditionnellement au type d'appareil visitant le site.



FIGURE 2. Nav-bar sous grand écran.

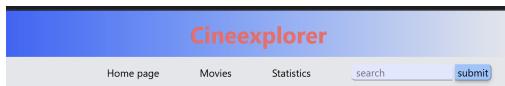


FIGURE 3. Nav-bar sous écran moyen.

B. Composants



FIGURE 4. Nav-bar sous petit écran.

1) Movie card: Dans le dossier movies/templates/movies/components, on définit deux composants que l'on réutilisera tout au long de notre application :

"movie_card.html", correspondant à l'affichage d'un film sur notre site avec : son titre, et un poster représentant le film.

Cependant, notre base de données n'incluant pas d'images, il nous faudra récupérer les images correspondant à chaque film depuis internet. Fort heureusement, nous utilisons des données issues d'imdb, et il existe de nombreuses api permettant d'effectuer des requêtes vers des bases de données imdb et notamment utilisant les mêmes identifiants pour les mêmes films que nous.

Dans notre cas, nous utiliserons l'api "https://api.imdbapi.dev" puisqu'il s'agit non seulement d'une api gratuite, mais qu'elle ne nécessite pas de clé privée pour pouvoir effectuer des requêtes. Cela nous permettra d'effectuer les requêtes directement côté client sans exposer un quelconque secret de développement et ainsi réduire la latence du serveur pour améliorer l'expérience utilisateur.

Pour récupérer les images depuis cette api, chaque page incluant des movie_cards devra également inclure la page statique "fetch_movie_thumbnails.js" qui contient le script effectuant l'appel à l'api et chargeant les images dans chacune des movies cards. Afin d'éviter de dépasser le nombre de requêtes qu'autorise l'api, on pense à ajouter un petit délai entre chaque requête dans notre script. Afin d'éviter de frustrer l'utilisateur pendant qu'il attend le chargement des images, un placeholder simple constitué du titre du film sur un fond gris s'affichera en attendant la réponse de l'api.

Notre composant prendra comme arguments :

- L'identifiant du film, pour pouvoir le chercher dans l'api imdb.
- Le titre du film pour pouvoir l'afficher.

Maintenant que nous avons défini notre composant, nous pourrons l'utiliser dans le reste de notre site avec la ligne :

```
{% include
  "movies/components/movie_card.html" with
  mid=movie.mid title=movie.title %}
```

2) Movie page: Dans notre site, deux pages seront amenées à utiliser un système de pagination des résultats : list et search. Pour éviter d'avoir à répéter du code, on va donc modulariser notre application et séparer le système de pagination en le mettant sous la forme d'un composant.

Chaque page affiche 20 films sous la forme de movie_cards, notre composant les affichera donc sous la forme de grille avec un nombre de colonnes dépendant de la taille de l'écran pour garantir un bon affichage sous tout appareil.

Notre composant affichera également la page actuelle, le nombre de pages total et un système de navigation entre les pages par numéro.

C'est pourquoi il aura besoin comme arguments de :

- La liste des 20 films.
- Le numéro de page.
- Le nombre de pages totales.
- Une liste des pages où peut naviguer l'utilisateur (+- 3 pages autour de la page courante).
- La requête de base pour pouvoir y changer le numéro de page pour faire naviguer l'utilisateur.

C. Templates

1) Home: La page d'accueil de notre site affiche :

- Le nom et slogan du site.
- Des statistiques basiques en bandeau.
- Les 10 meilleurs films sous forme de grille(responsive).
- Des recommandations de films aléatoires.

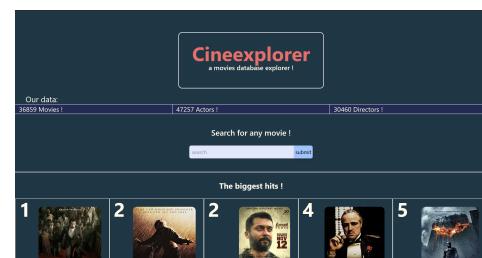


FIGURE 5. Page d'accueil sous grand écran.

2) movies: La page movies affiche des listes de films (en utilisant le composant movie_page) en fonction de filtre. Les filtres prennent la forme d'un form, effectuant une requête GET sur la page movies, mais avec les champs correspondant aux filtres remplis, pour qu'ils puissent être lus par la view.

Afin de rendre l'affichage des filtres responsive, on change la direction du display flex de flex-row à flex-col pour les petits écrans.

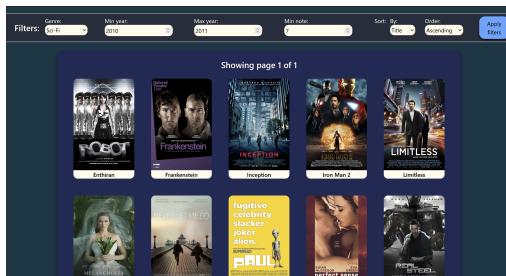


FIGURE 6. Page movies sous grand écran.

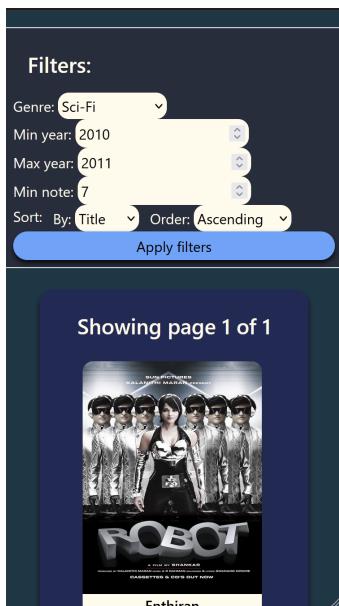


FIGURE 7. Page movies sous petit écran.

3) *movies/<id>*: La page *movies/<id>* affiche les détails du film correspondant à <id> à savoir :

- Un poster récupéré sur `api.imdbapi.dev` (en grand, pas une `movie_card`).
- Le titre
- Les genres
- Les directeurs si ils existent.
- Les auteurs si ils existent.
- Le casting (quel acteur joue quel personnage) sous forme de grille.
- Les titres alternatifs par région.
- Des films similaires par :
 - directeur(s)
 - genre(s)



FIGURE 8. Page movies/id sous grand écran

4) *search*: La page *search* est très similaire à *movies*, en cela qu'elle est elle aussi constituée d'un form est d'un `movie_page`, à la différence près qu'ici, le form est constitué d'un champ de recherche et d'un type de recherche pour alterner entre la recherche par personne et par titre à l'aide d'une balise `select`.

5) *stats*: La page *stats* contient des statistiques tirées de nos bases de données. Afin de les mettre en forme, on utilise la librairie frontend javascript `Chartjs`. Pour l'inclure, on ajoute la ligne :

```
<script
  → src="https://cdn.jsdelivr.net/npm/chart.js">
</script>
```

On crée des balises `canvas` dans notre `html`, que l'on vient ensuite récupérer à l'aide de scripts javascript pour y injecter les charts, comme dans l'exemple suivant :

```
new
→ Chart(document.getElementById("genres-canvas"),
→ {
  type: "bar",
  data: {
    labels: [% for g in movies_by_genre
    → %] "{{ g.genre }}", % endfor
    → %],
    datasets: [%{
      label: "number of movies",
      data: [% for g in
      → movies_by_genre %] {
        → g.movie_count }, % endfor
        → %]
    }]
  },
  options: {
    responsive: true,
    maintainAspectRatio: true
  }
}); // movies_by_genre
```

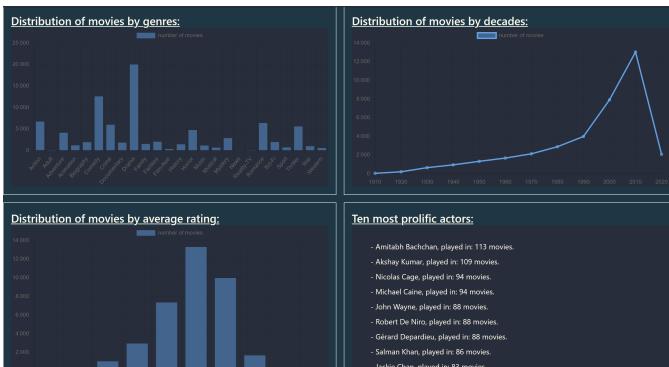


FIGURE 9. Page stats sous grand écran.

VII. DIFFICULTÉS RENCONTRÉES

Avec la version de django utilisée pour cette application (6.0) les requêtes paramétrées des curseurs SQLite déclenchant un bug qui empêche entièrement notre application de fonctionner en déclenchant l'erreur suivante :

FIGURE 10. Erreur déclenchée par django

pourtant, en se penchant un peu plus sur la cause de l'erreur, on s'aperçoit que la requête formulée est correcte :

```
return sql % params
```

qui une fois modifiée en :

```
return sql
```

nous laisse exécuter nos requêtes normalement.

Si ce genre d'erreur peut paraître surprenante pour un framework aussi populaire que django, elle pourrait s'expliquer par la philosophie du framework qui vise à utiliser l'ORM plutôt que des requêtes SQL pures. Un bug qui a première vue peut paraître évident, pourrait donc ne pas impacter la plupart des utilisateurs.

VIII. CONCLUSION

Partir de bases de données correctement normalisées, optimisées à l'aide d'index et configurées permet de fluidifier grandement le développement et de se concentrer sur le flux des données à l'intérieur de l'application. Il en reste néanmoins que si cela permet de se rendre compte à quel point passer le temps nécessaire pour modéliser correctement ses données est primordial pour toute application, afin de la rendre réactive, performante et fiable, créer la partie métier et l'interface nécessite tout de même un effort et une attention particulière, et si des frameworks comme django permettent de simplifier l'architecture de l'application et la création d'interfaces dynamiques, ils peuvent être parfois source de nouveaux problèmes, en particulier lorsque l'on s'éloigne de leurs fonctionnalités privilégiées (comme l'ORM django) pour gagner plus de contrôle. Cependant il s'agit d'étapes nécessaires si l'on veut profiter pleinement des données que l'on a disposition et les rendre navigables et exploitables pour nos utilisateurs.

```
C:\Users\guilh\Desktop\Cours\Semestre_7\BDD2\cineexplorer\.venv\Lib\site-packages\django\db\backends\sqlite3\operations.py:181:     return sql % params
                                                ^^^^^^^^^^

▼ Local vars
Variable Value
cursor <django.db.backends.sqlite3.base.SQLiteCursorWrapper object at 0x000001A2E00EA450>
params ("", '0', "9000", "0.0", "0")
self <django.db.backends.sqlite3.Operations object at 0x000001A2E011C7D0>
sql   ('```
        SELECT m.movie_id, t.title_name, m.year, r.average_rating
        FROM movies m```
        JOIN movie_titles mt ON m.movie_id = mt.movie_id```
        JOIN titles t ON mt.title_id = t.title_id ```
        JOIN ratings r ON m.movie_id = r.movie_id```
        JOIN movie_genres mg ON m.movie_id = mg.movie_id```
        JOIN genres g ON mg.genre_id = g.genre_id```
        WHERE g.genre_name LIKE ?```
        AND m.year <=?```
        AND m.year >=?```
        AND r.average_rating >=?```
        ORDER BY t.title_name ASC```
        LIMIT 20 OFFSET ?```
        ')
```

FIGURE 11. Trace de l'erreur

Cette erreur est en effet aisément reproductible, en effet, elle apparaît lorsque l'on effectue n'importe quelle requête paramétrée avec SQLite. La cause de l'erreur ne vient pas de la requête en elle-même mais bien du backend SQLite de django, plus particulièrement de cette ligne du fichier de l'installation de django : `django/db/sqlite3/operations.py` :