

AlloMovie



ASSIE Alexandre
CLEMENT Hugo
MORIN Céline

WebX



1. User Stories

Nous avons respecté au maximum les exigences extra-fonctionnelles du projet :

- Le serveur correspondant à la partie back-office est implanté en suivant le style d'architecture REST à l'aide du framework Jersey
- Le client de la partie front est réalisé avec VueJS 2.0 et exploite la librairie Axios
- Nous avons utilisé l'API OMDb afin de récupérer les informations sur les films
- La base de données sélectionnée pour conserver les Avis est H2
- Apache Maven a été utilisé pour structurer notre projet, gérer les dépendances et faciliter le lancement des tests (clean test site)
- Git est notre gestionnaire de version

Nous avons essayé de respecter les exigences RESTful au maximum :

- Les URL suivent le principe RESTful des accès aux ressources ; on peut trouver une ressource spécifique avec un appel de type `/ressource/{id}`, et récupérer toutes les ressources de ce type avec `/ressource`
- Nous avons utilisé des path params dès que possible afin d'éviter au maximum les query params
- L'API peut fonctionner indépendamment du front
- Lors de la création d'une ressource, sa location est renvoyée dans le header
- Des statuts HTTP sont renvoyés avec chaque réponse

a. Partie back-end

Le JSON est le plus simple pour envoyer et utiliser des données rapidement. Il est d'ailleurs plus léger que XML et permet d'économiser des ressources. XML convient mieux quand il s'agit de présenter les données ; or les réponses de notre API sont plutôt destinées à être interprétées et affichées par un client (avec VueJS dans notre cas) et non d'être affichées directement en brut. De plus la librairie Axios se base sur du JSON ; c'est pourquoi nous l'avons choisi.

Le back-end est organisé en couches : *domains* pour les objets utilisés, *dao* pour les appels à la base de données, *services* pour la logique métier et *controllers* pour gérer la réception des requêtes HTTP. Ces couches sont dédoublées : on trouvera une classe de chaque pour Avis (path `/avis`) et Film (path `/film`).

Nous avons utilisé Postman afin de tester des requêtes vers notre API et de voir les réponses obtenues en détail. Des fonctions qui ne sont pas relatives aux US ont été créées, soit par nécessité soit pour respecter les exigences RESTful :

- findAvisByFilm : GET `/avis/film/{filmID}`
- findAvisById : GET `/avis/{id}`
- findAllAvis : GET `/avis`

US1a - Récupérer la liste des films

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/1>

Cette US est réalisée par deux requêtes de type GET :

- Récupère la liste des films par son titre via le path `/film/liste/{titre}`.
Exemple : <http://localhost:8081/allomovie/film/liste/star+wars>
- Récupère la liste des films par son titre et son année de sortie via le path `/film/liste/{titre}/{année}`.
Exemple : <http://localhost:8081/allomovie/film/liste/star+wars/2005>

Un titre est valide si sa taille est supérieure ou égale à 4 caractères.

Une année est valide si c'est un entier de taille 4.

Les codes de retour et réponses qui peuvent être renvoyés sont :

- OK (200) : Si tout se passe bien. Accompagné de la liste des films.
- BAD_REQUEST (400) : Si il est renseigné dans le path un titre et/ou une année invalide. Accompagné d'un message d'erreur indiquant si c'est le titre ou l'année qui est invalide.
- NOT_FOUND (404) : Si aucun film n'est trouvé avec les critères fournis. Accompagné d'un message d'erreur indiquant qu'aucun film n'a été trouvé.

Le code permettant la réalisation de l'US se trouve dans les classes :

```
back/src/main/java/webx/huceal/controllers/FilmController.java
back/src/main/java/webx/huceal/services/FilmService.java
back/src/main/java/webx/huceal/dao/FilmDAO.java
back/src/main/java/webx/huceal/domains/Film.java
```

Nous estimons à 95% le niveau d'aboutissement de la solution proposée. En effet nous n'avons pas réalisé de pagination pour les résultats, mais tout le reste est fonctionnel.

US1b - Donner son avis sur un film

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/2>

Cette US est réalisée par une requête de type POST :

- Ajoute un avis via le path `/avis` au travers d'un objet JSON comprenant les paramètres filmID (String), note (int) et commentaire (String). Renvoie la location de la ressource si créée.
Exemple : <http://localhost:8081/allomovie/avis>

```
CREATE TABLE IF NOT EXISTS Avis
(ID INTEGER AUTO_INCREMENT PRIMARY KEY,
FilmID VARCHAR(10) CHECK(REGEXP_LIKE(FilmID, '^tt[0-9][0-9][0-9][0-9][0-9][0-9][0-9]$')),
Note INTEGER(1) CHECK (Note >= -1) AND (Note <= 5),
Commentaire VARCHAR(500),
CONSTRAINT avis_not_empty CHECK (note != -1 OR Commentaire != ''));
```

Le FilmID est valide s'il correspond au regex ci-dessus.

Une note est valide si c'est un entier entre -1 et 5 : -1 si elle n'est pas entrée, de 0 à 5 sinon.

Un commentaire est valide s'il fait moins de 500 caractères (il peut être vide).

Un avis est valide s'il contient au moins un commentaire ou une note.

Les codes de retour et réponses qui peuvent être renvoyés sont :

- CREATED (201) : Si tout se passe bien. Accompagné de la location de la ressource créée dans le header sous la forme `/avis/{id}`.
- BAD_REQUEST (400) : Si il est renseigné dans l'objet JSON un filmID, une note, un commentaire ou un avis invalide. Accompagné d'un message d'erreur indiquant ce qui ne va pas.
- INTERNAL_SERVER_ERROR (500) : Si la base de données a un souci de connexion lors de l'ajout de l'avis.

Le code permettant la réalisation de l'US se trouve dans les classes :

```
back/src/main/java/webx/huceal/controllers/AvisController.java
back/src/main/java/webx/huceal/services/AvisService.java
back/src/main/java/webx/huceal/dao/AvisDAO.java
back/src/main/java/webx/huceal/domains/Avis.java
```

Nous estimons à 100% le niveau d'aboutissement de la solution proposée ; tout est fonctionnel.

US1c - Rechercher des films

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/3>

Cette US est réalisée par une requête de type GET :

- Récupère la liste des films ayant une moyenne des avis reçu supérieur ou égale à la note demandé et/ou ayant un commentaire contenant le mot demandé via le path `/film/avis` avec en paramètre (query params) une note et un mot.

Exemple : <http://localhost:8081/allomovie/film/avis?note=3&commentaire=bien>

Une note est valide si c'est un entier compris entre 0 et 5.

Les codes de retour et réponses qui peuvent être renvoyés sont :

- OK (200) : Si tout se passe bien. Accompagné de la liste des films respectant les critères de recherche.
- BAD_REQUEST (400) : Si il est renseigné en paramètre une note invalide. Accompagné d'un message d'erreur indiquant que la note est invalide.
- NOT_FOUND (404) : Si aucun film n'est trouvé avec les critères fournis. Accompagné d'un message d'erreur indiquant qu'aucun film n'a été trouvé.

Le code permettant la réalisation de l'US se trouve dans les mêmes classes que US1a.

Nous estimons à 100% le niveau d'aboutissement de la solution proposée ; tout est fonctionnel.

US1d - Modération des commentaires

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/4>

Cette US est réalisée par une requête de type DELETE :

- Supprime tous les avis contenant un mot donné via le path `/avis/{mot-clé}` et renvoie en JSON le nombre de commentaires supprimés.

Exemple : <http://localhost:8081/allomovie/avis/bien>

Le mot-clé est valide s'il ne contient pas d'espace (un seul mot-clé) et n'est pas vide.

Les codes de retour et réponses qui peuvent être renvoyés sont :

- OK (200) : Si tout se passe bien. Accompagné du nombre de commentaires supprimés en JSON (> 0).
- BAD_REQUEST (400) : Si il est renseigné dans le path un mot-clé invalide. Accompagné d'un message d'erreur indiquant ce qui ne va pas.
- NOT_FOUND (404) : Si aucun commentaire comportant le mot-clé n'a été trouvé.

Le code permettant la réalisation de l'US se trouve dans les mêmes classes que l'US1b.

Nous estimons à 100% le niveau d'aboutissement de la solution proposée ; tout est fonctionnel.

b. Partie front-end

Pour la partie front nous avons opté pour une solution complète et facile de mise en place. Pour cela nous nous basés sur webpack, un incontournable pour le front-end qui nous permet d'utiliser un routeur et une architecture propre permettant un travail de groupe efficace ainsi qu'une meilleur appropriation du projet. Pour ce faire vous avons utilisé les commandes suivantes :

```
$ npm install -g vue-cli
$ vue init webpack allomovie
$ cd allomovie
$ npm install
$ npm run dev
```

Pour la partie visuelle, nous n'avons décidé de nous baser sur du bootstrap pour la mise en place d'un graphisme simple, élégant et responsive. Une surcouche CSS a quand même été mise en place afin d'avoir un design personnalisé et intuitif.

Pour ce qui est de l'application front en elle-même nous avons déterminé deux pages primordiales avec d'une part une page d'accueil contenant une liste de films ainsi que le

moteur de recherche permettant de sélectionner par titre ou/et par date de parution. D'autre part, une seconde partie permet d'afficher un film en fonction de son identifiant et de récupérer l'ensemble des informations pertinentes de celui-ci ainsi que les avis relatifs. En plus de cela un formulaire est présent afin de pouvoir soumettre un avis.

US2a - Rechercher des films

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/5>

La recherche se situe sur la page d'accueil du site, celle-ci utilise le fameux axios avec une requête get vers `/film/liste` afin de récupérer et d'afficher de les afficher.

```
axios.get(api + request)
  .then(response => {
    this.filmList = response.data
  })
  .catch(e => {
    this.errors.push(e.response.data.message)
  })
```

Dans le cas de la réussite on affiche simplement la liste de film, sinon on affiche un message d'erreur dans un encadré en rouge en haut de la page.

La couverture de cette User Story est donc de **100%**.

US2b - Afficher un film

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/6>

Mise en place d'une route `/film/{filmID}` pour un film, elle affiche les informations utiles d'un film ainsi que l'ensemble des avis utilisateurs. Pour cela deux requêtes XHR doivent être effectuées afin d'obtenir d'une part le film et ses informations et d'autre part l'ensemble des avis utilisateurs.

La couverture de cette User Story est donc de **100%**.

US2c - Donner son avis sur un film

<https://github.com/DCLL-MDL/projet-webx-2017-2018-allomovie-webx-huceal/issues/7>

Etant donné un film, l'utilisateur a la possibilité de déposer un commentaire composé d'une note et/ou d'un commentaire. Les deux ne sont pas forcément requis mais l'un des deux champs doit être rempli. De plus, d'après les exigences fonctionnelles demandées nous avons décidé de mettre en place une différenciation entre la note "0" et la non notation "-1", ce qui dégrade un peu l'expérience utilisateur mais correspond à l'US.









La couverture de cette User Story est donc de **100%**.

2. Politique de tests

Pour ce qui est du front, la qualité du code est assurée par l'ensemble des recommandations données par le Style Guide de VueJS ; il y a 0 erreur.

Pour nos tests du back-end nous utilisons JUnit 4.12 et Mockito 1.9.5. Nous avons utilisé Jacoco afin de calculer la couverture de tests. Afin de tester la qualité du code nous avons choisi d'utiliser Checkstyle et Findbugs.

Nous avons réussi à avoir 0 erreur sur Findbugs, seulement 12 sur Checkstyle (mais ce sont des erreurs qui ne peuvent pas être résolues, ce qu'on expliquera dans la suite de cette partie). Nos 98 tests couvrent 93% du code (hors classes du package domains) et tous les tests passent.

Element	Missed Instructions	Cov.
 webx.huceal		60 %
 webx.huceal.dao		98 %
 webx.huceal.services		99 %
 webx.huceal.controllers		100 %
Total	108 of 1 560	93 %

Des tests unitaires ont été effectués sur la partie DAO et Services, mais pas sur la couche Controllers car cette dernière ne fait qu'appeler la couche Services pour récupérer la réponse à renvoyer. La couche Controllers étant la seule directement en contact avec le front-office, nous n'avons effectué de tests d'intégration que pour cette dernière.

Tests unitaires DAO

Nombre de tests : 32.

Pour ces tests unitaires, nous avons utilisé une base de données différente de celle utilisée pour l'application pour des raisons évidentes : les tests unitaires doivent être indépendants les uns des autres, l'environnement doit donc être mis en place puis détruit avec chaque test. Ici, nous avons testé tous les appels valides et invalides aux fonctions de la couche DAO afin de vérifier que les accès à la base de données fonctionnent bien : on teste le bon fonctionnement des queries.

On peut trouver ces tests dans les classes suivantes :

```
back/src/test/java/webx/huceal/dao/AvisDAOUnitTest.java
back/src/test/java/webx/huceal/dao/FilmDAOUnitTest.java
```

Tests unitaires Services

Nombre de tests : 40.

Pour ces tests unitaires, nous avons utilisé des mocks qui permettent de simuler le fonctionnement des DAO, ce qui permet donc réaliser les tests des services indépendamment du fonctionnement des classes DAO. Ici, on n'utilise donc pas de base de données de test puisqu'on accède jamais réellement à celle-ci.

On peut trouver ces tests dans les classes suivantes :

```
back/src/test/java/webx/huceal/services/AvisServiceUnitTest.java
back/src/test/java/webx/huceal/services/FilmServiceUnitTest.java
```

Tests d'intégration Controllers

Nombre de tests : 26.

Pour ces tests d'intégration, nous avons créé un client Jersey afin de tester les réponses renvoyées par l'API : on mime des appels à la couche Controllers puis on vérifie le statut, le type et le contenu de la réponse. Une base de données de test a été utilisée afin de ne pas impacter celle de l'application.

On peut trouver ces tests dans les classes suivantes :

```
back/src/test/.../controllers/AvisControllerIntegrationTest.java
back/src/test/.../controllers/FilmControllerIntegrationTest.java
```


3. Difficultés rencontrées

Back-end

Titre du problème : Gestion de la DataSource

Source : <https://bit.ly/2H4tDi0>

Description : Nous avons eu un peu de mal à gérer l'accès à la base de données sans utiliser Spring en premier lieu ; c'est ce lien qui a pu nous permettre d'avancer. Il propose une connexion en singleton à une base de données H2.

Titre du problème : RESTful et QueryParam

Source : <https://www.javacodegeeks.com/2012/01/simplifying-restful-search.html>

Description : Afin de respecter au mieux RESTful nous avons utilisé le plus possible PathParam. Mais sur la recherche de film par avis, nous avons dû utiliser des QueryParam car les deux paramètres (la note et le commentaire) sont facultatifs. Malgré ça, notre API reste RESTful.

Titre du problème : Changements données API OMDb

Source : <http://www.omdbapi.com/>

Description : Durant le développement nous avons eu un problème de changement de données de l'API OMDb. Certains films ont vu le lien de leur image être modifiés. Cela a impacté nos tests qui contenaient encore l'ancien lien. Nous avons réglé ce problème en mettant à jour nos données de tests, mais si dans le futur d'autres films doivent être modifiés (surtout ceux que nous utilisons) cela pourrait provoquer un échec de nos tests malgré le bon fonctionnement de notre code.

Titre du problème : Cross-Origin Resource Sharing

Source : <https://bit.ly/2JQr1lJ>

Description : Nous avons eu une erreur de type CORS très tôt lorsque nous avons essayé de tester le front avec l'API du back ; cette classe CORSFilter a permis d'ajouter automatiquement les headers nécessaires à toutes les réponses renvoyées.

Titre du problème : Regex SQL pour le FilmID

Source : <https://www.guru99.com/regular-expressions.html>

Description : Nous souhaitions avoir un regex au niveau de la table SQL afin de bloquer les filmID qui ne correspondent pas au format. En effet, étant une String et non un long, ce type de vérification permet d'apporter de la sécurité. Ce lien décrit les possibilités de regex SQL.

Titre du problème : Mockito

Source : <http://www.baeldung.com/mockito-behavior>

Description : Pour les tests unitaires de la couche Services, nous devons nous abstraire de la couche DAO afin de ne tester que les services ; Mockito est la solution que nous avons choisie. Ce lien montre toutes les possibilités des mocks.

Front-end

Titre du problème : Responsive

Source : <http://getbootstrap.com/docs/4.1/examples/>

Description : Mise en place d'un design propre et responsive, on utilise donc un template pré-fait, ce qui a permis un gain de temps énorme.

Titre du problème : Manque d'images

Source : <https://placeholder.com/>

Description : Pour palier au problème des images non présentes, on a cherché une solution rapide afin de générer des images fictives pour garder un template cohérent. Cette API permet de générer des images en fonction de la hauteur et la largeur, comme cet exemple qui génère une image de 350 * 150 -> <https://via.placeholder.com/350x150>.

Titre du problème : Affichage d'icônes

Source : <https://fontawesome.com/>

Description : Recherche d'un système permettant d'afficher des icônes de façon rapide et simple. Cette librairie est plutôt simple d'utilisation et permet de mettre facilement l'affichage des étoiles pour la notation. Nous l'avons essentiellement utilisé pour l'affichage des étoiles grâce aux éléments `<i class="fas fa-star"></i>`.

Titre du problème : Système de notation

Source : <https://github.com/craigh411/vue-star-rating/>

Description : Pour ne pas avoir à re-développer la solution de notation, on a donc utilisé un composant de VueJS. Pour cela il suffit ensuite d'utiliser le composant `<star-rating v-model="variable"></star-rating>` afin de l'utiliser.