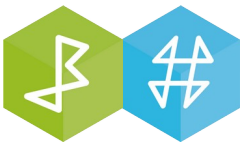


Iut Belfort 2022/2023

Rapport de projet S4.01

Services Web

Lilian Schott, Gabin Blondieau, Samson Dupuy

Arnaud Chevalme, Mael Turchetto, Hugo Foulon

Introduction

Le projet que nous avons utilisé est celui du FIMU réalisé au S3.

Il sert à gérer l'ensemble des informations relatives au festival. Pour ce faire une API REST avec Node.js et Express fournit les informations au front en VueJS.

État avant le cours R4.01

Nous pouvions déjà nous authentifier grâce à des JWT token. Ces tokens sont générés par l'API NodeJS/Express puis stockés dans le front en VueJS. Ils sont ensuite échangés dans chaque requête et permettent au serveur de vérifier qu'un utilisateur à le droit d'accéder à la ressource souhaitée.

Pour ce faire un rôle est attribué à chaque utilisateur et ce rôle décrit quelles manipulations sont permises (ajout/modification/suppression). C'est un middleware qui récupère les informations de l'utilisateur, obtient son rôle et l'autorise ou non à poursuivre.

Activité 1 :

Micro Service :

Cette tâche n'est pas réalisé car nous avons eu des difficultés à faire interagir le micro service d'authentification avec l'API principale et nous avons donc choisi de revenir en arrière et laisser le fonctionnement initiale, c'est à dire avoir toutes les routes et contrôleurs dans l'API principale.

Autorisation :

La stratégie d'autorisation de base n'a pas changée, elle fonctionne comme décrit dans l'introduction. En revanche, l'authentification via Github et Google a été ajouté. Pour cela, des Stratégies ont été utilisées avec passport.js dans les routes initiant les flows correspondants.

Les différentes clés ont été obtenus en créant des applications pour l'authentification via OAuth2 sur les services correspondants.

Lorsque que utilisateur initie une connexion grâce à ces réseaux sociaux, la vue nous redirige sur la page d'authentification dédiée. Après avoir accepté, une redirection se fait sur la vue qui va alors récupérer le code d'autorisation et effectuer une requête sur l'API pour demander l'échange de ce token avec les informations de l'utilisateur.

L'API décide alors si il faut créer un utilisateur ou simplement le récupérer. Pour Google, l'adresse email est utilisée en tant qu'identifiant pour garantir son unicité. Pour Github, on utilise le nom d'utilisateur pour la même raison.

Une fois l'utilisateur créé/récupéré, l'API crée un JWT token et le transmet à la Vue qui va le stocker dans le localStorage et l'envoyer à chaque requête pour permettre de vérifier les droits d'accès aux différentes ressources comme expliqué dans l'introduction.

Sauvegarde du token :

Côté Vue, le token est donc stocké dans le localStorage plutôt que dans un cookie. Ce choix est discutable mais nous avons choisis cette approche pour garder l'API aussi « stateless » que possible.

Fichiers concernés :

API :

- API/routes/auth.router.ts
- API/config/github-passport.config.ts
- API/config/google-passport.config.ts

Vue :

- vue/src/components/GithubRedirect.vue
- vue/src/components/GoogleRedirect.vue
- vue/src/router/index.js

Pour tester :

Pour tester l'ensemble de ces fonctionnalités, il faut suivre le readme pour lancer l'API ainsi que la Vue. Il faut se rendre sur la page de login à http://localhost:8080/login. On peut alors se connecter avec un des utilisateurs par défaut (admin:admin ou editor:editor) ou bien via Github ou Google.

Activité 2 : Sockets et émetteurs (Lilian Schott)

Pendant la période de travail sur se projet, nous avons amélioré un projet que nous avions commencés précédemment notamment en ajoutant des sockets et des émetteurs en les utilisant dans un cas précis.

Utilisation des sockets et des émetteurs:

La tâche est partiellement réalisée car il y a des problèmes avec l'affichage des messages dans le chat. Nous avons donc opté pour la création d'un chat afin que les personnes connectées à notre site puissent communiquer via le chat.

On a donc utilisé les sockets afin de créer des instances de socket afin d'écouter les événements « init » et « message » envoyé par le serveur, mais aussi afin d'émettre des événements « newMessage » lorsque l'utilisateur envoie un nouveau message. Les messages sont ensuite stockés dans un tableau « messages », lui même affiché dans le template.

De plus, nous avons crée un serveur HTTP et une instance de socket.IO sur le port 3000 sur lequel nous écoutons l'événement « connexion » pour gérer la connexion de nouveaux clients. Une fois qu'un client est connecté, nous écoutons l'événement « chat message » pour recevoir des messages du client et émet- tons l'événement « chat message » à tous les clients connectés pour diffuser le message. Nous écoutons également l'événement « disconnect » pour gérer la déconnexion des clients.

Pour tout cela, nous avons utilisé le package « socket.io-client ».

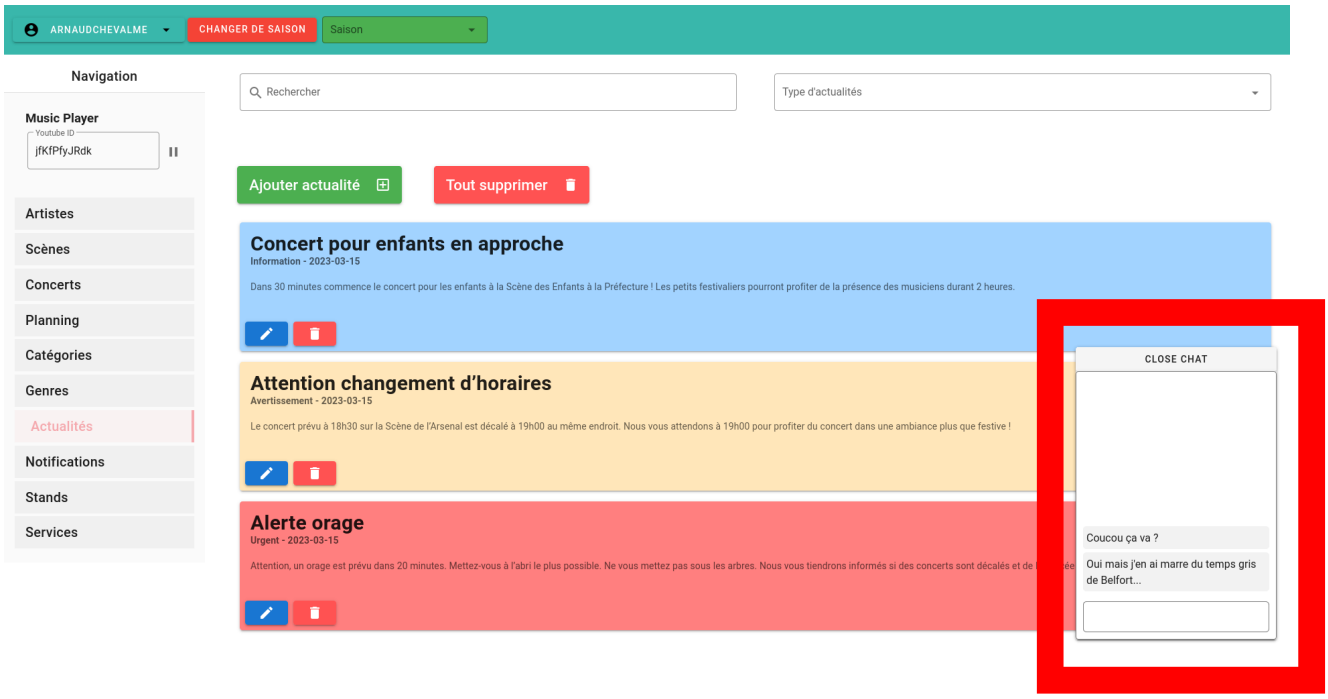
Fichiers concernés:

Sur Github, les principaux fichiers implémentant cette fonctionnalité de chat sont le fichier server.ts ainsi que le fichier ChatMenu.vue.

Pour tester :

Pour tester cette fonctionnalité, vous pouvez exécuter l'application en local et ouvrir plusieurs instances dans différents navigateurs ou terminaux pour simuler plusieurs clients. Vous pouvez également utiliser des outils de test automatisé pour tester l'application.

Capture d'écran:



Activité 3 : Ajouter 8 mesures de sécurité / protection à cette application (Gabin Blondieau)

Première protection : Attaque DDoS

Les attaques DDoS (Distributed Denial of Service) sont des attaques informatiques visant à rendre un service ou un site web inaccessible en surchargeant ses serveurs avec un grand nombre de demandes de connexions simultanées.

Nous utilisons donc le module “express-rate-limit” qui va justement permettre à notre API de limiter le nombre de requêtes par seconde.

Nous trouvons alors les paramètres de limitation dans le fichier “rate-limiter.ts” dans le dossier “middleware” de l'API. Nous avons donc pensé que pour un projet comme le FIMU, 300 requêtes par secondes étaient correctes. L'utilisation du limiteur est donc appelé dans le fichier “server.ts” à la ligne 89.

Deuxième protection: Injection de commande

Dans notre code, nous n'utilisons à aucun moment la fonction “eval()” qui permet d'exécuter du code JavaScript à partir d'une chaîne de caractères. Cette fonction peut donc être utilisé pour exécuter du code malveillant.

Troisième protection: Injection SQL

Sequelize est une bibliothèque ORM (Object-Relational Mapping) pour Node.js qui permet d'interagir avec une base de données relationnelle (comme MySQL, PostgreSQL, SQLite, etc.) en utilisant des objets JavaScript au lieu d'écrire des requêtes SQL manuellement.

L'une des fonctionnalités importantes de Sequelize est qu'elle empêche les injections SQL grâce à l'utilisation de requêtes paramétrées et de l'encodage des caractères spéciaux. Les requêtes paramétrées permettent de séparer les données des instructions SQL, ce qui empêche les attaquants d'insérer des commandes SQL malveillantes dans les données d'entrée.

Nous utilisons ainsi Sequelize pour toutes les requêtes SQL du projet.

Quatrième protection: Mot de passe crypté

Le cryptage de mot de passe est une sécurité de base. Si ces informations sont stockées en texte clair dans la base de données du site, elles peuvent être facilement compromises en cas de violation de données ou de piratage. Nous pouvons voir un exemple de cryptage dans le fichier “PasswordEditView.vue” dans le dossier “vue/views/utilisateur”.

Cinquième protection: Faiblesse des mots de passe

Il également important que ces mots de passe aient une longueur suffisante ou alors qu’ils contiennent des caractères spéciaux pour être plus difficilement usurpable. Vous pouvez trouver une vérification de longueur dans le même fichier que la protection précédente.

Sixième protection : Séparation des permissions

Une application peut comporter plusieurs fonctionnalités, chacune nécessitant des autorisations spécifiques pour être utilisée. Chacune de ces fonctionnalités nécessite un niveau d'accès différent, et il est important de s'assurer que chaque utilisateur n'a accès qu'aux fonctionnalités pour lesquelles il est autorisé. Dans notre projet, toute la partie vue est régie par la séparation des rôles, par exemple un éditeur ne pourra pas supprimer un artiste.

Septième protection : JSON WEB TOKEN

Les JSON Web Tokens (JWT) sont un format ouvert de jeton d'authentification utilisé pour transmettre des informations entre les parties de manière sécurisée et compacte. Les JWT sont souvent utilisés pour les applications web et les API, en particulier pour l'authentification et l'autorisation des utilisateurs.

Les avantages des JWT sont les suivants :

- Sécurité : Les JWT sont signés numériquement et cryptés pour garantir l'intégrité des données transmises. Les informations stockées dans le JWT sont vérifiées à chaque demande, ce qui permet de s'assurer que les données n'ont pas été altérées pendant la transmission.
- Compact : Les JWT sont plus compacts que les cookies et les sessions, ce qui les rend plus efficaces pour les applications mobiles et les API. Les JWT sont également facilement transférables et peuvent être utilisés dans des environnements distribués.
- Autonomie : Les JWT stockent toutes les informations d'authentification nécessaires dans le jeton lui-même, ce qui permet aux serveurs de vérifier l'authenticité d'un utilisateur sans avoir besoin de consulter une base de données. Cela peut rendre l'application plus rapide et plus évolutive, en réduisant la charge sur les serveurs.

- Flexibilité : Les JWT peuvent être utilisés pour l'authentification et l'autorisation des utilisateurs, ainsi que pour la transmission d'autres types d'informations. Les JWT peuvent également être configurés pour expirer automatiquement après une période de temps donnée, ce qui renforce la sécurité en limitant le temps d'exposition.

En résumé, les JSON Web Tokens sont un format pratique et sécurisé pour l'authentification et l'autorisation des utilisateurs dans les applications web et les API. Ils offrent une sécurité accrue, sont plus compacts que les cookies et les sessions, permettent une autonomie accrue des serveurs et sont flexibles et configurables pour répondre à une variété de besoins d'authentification et d'autorisation.

Pour plus d'informations sur leur implémentation dans notre projet je vous invite à lire la partie commune.

Huitième protection : Le CSRF

Le Cross-Site Request Forgery (CSRF) est une technique d'attaque informatique qui vise à tromper un utilisateur en lui faisant effectuer une action non désirée sur un site web ou une application. Cette attaque fonctionne en exploitant la confiance de l'utilisateur dans un site web ou une application, en utilisant une session active ou un cookie pour effectuer des actions malveillantes à son insu.

Les attaques CSRF peuvent avoir des conséquences graves, comme la modification des paramètres de compte d'un utilisateur, la suppression de données, ou la réalisation de transactions financières frauduleuses.

Les mesures de protection contre les attaques CSRF incluent l'utilisation de tokens de synchronisation de jeton (CSRF tokens), qui sont des jetons de sécurité générés par l'application et envoyés à l'utilisateur avec chaque requête HTTP. Ces jetons sont vérifiés par l'application pour s'assurer que la requête est légitime et qu'elle provient d'une source fiable.

Malheureusement je n’ai pas eu le temps d’implémenter cette protection dans notre projet.

Activité 5 : Traitement de grandes quantités de données avec Node.js (Arnaud CHEVALME)

Pour cette activité, la plus grande difficulté était de trouver quelque chose se rattachant au FIMU. Comme scrapper twitter pour récupérer les posts du FIMU c’est très vite avéré impossible, j’ai du trouver autre chose. Au final, quoi de mieux qu’un peu de musique pendant qu’on travaille sur le festival ! J’ai donc décider d’ajouter un petit composant dans la sidebar du site.

Ce composant est en fait un « lecteur de musique ». On place l’identifiant d’une vidéo ou d’un livestream YouTube puis on clique sur le bouton play. J’ai choisi de tester cette fonctionnalité avec un livestream diffusé 24h/24 (valeur par défaut du champ) pour bien illustrer l’intérêt des streams NodeJS que j’ai utilisé.

En effet, mettre en mémoire l’ensemble de l’audio du live serait impossible. En revanche, avec les streams node.js, on peut traiter les informations au fur et à mesure qu’elles arrivent et ainsi bénéficier d’un traitement plus rapide mais surtout nécessitant moins de mémoire.

Pour implémenter cette fonctionnalité, j’ai d’abord installé les modules ytdl-core et ffmpeg-fluent côté API.

Pour me concentrer sur la gestion des flux côté API, j’ai aussi installé le module Howler sur la Vue pour être capable de facilement lire le stream audio que va renvoyer l’API.

J’ai défini une nouvelle route prenant en paramètre l’id du live/de la vidéo YouTube. J’utilise ensuite le module ytdl pour sélectionner le bon format et créer un premier stream nodejs donnant accès à la vidéo.

J’utilise ensuite ffmpeg pour modifier les données au fur et à mesure qu’elles arrivent et ainsi récupérer uniquement l’audio. Pour finir, une réponse dans express étant un stream « Writable », on peut piper directement l’audio dedans et c’est cette réponse qui est utilisée par Howler pour lire le fichier audio et égayer le travail des admins sur notre site.

Fichier concernés :

API/routes/music-stream.router.ts et vue/src/components/MusicPlayer.vue.

Pour tester :

Allez sur la Vue et connectez vous. Sur la gauche se trouve une zone de texte préremplie. Remplacez l’id de la vidéo ou laissez celui ci et appuyez sur play.

Activité 6 : Optimisation des performances (Mael Turchetto)

Optimisation du serveur

Durant cette période, j'ai travaillé sur l'optimisation du serveur pour améliorer ses performances.

Pour ce faire, j'ai décidé d'utiliser la clusterisation de Node.js, qui permet de créer plusieurs processus enfants (workers) qui s'exécutent simultanément sur plusieurs cœurs de CPU. Cela permet d'améliorer les performances de l'application en répartissant les tâches sur plusieurs cœurs de CPU.

Si le processus en cours d'exécution est le processus principal (master), il crée un processus enfant (worker) pour chaque cœur de CPU disponible sur la machine.

Si un processus enfant (worker) se termine, le processus principal en crée un nouveau pour maintenir le nombre de processus enfants constant.

Si le processus en cours d'exécution est un processus enfant (worker), il démarre l'application en écoutant le port PORT et affiche un message indiquant que le serveur a démarré.

Fichiers concernés :

Le principal fichier où se déroulent toutes ces opérations est server.ts .

Difficultés rencontrée :

Problèmes rencontrés avec la génération d'un graphique de flamme Cependant, lors de mes tentatives pour générer un graphique de flamme, j'ai rencontré plusieurs problèmes. En effet, l'API est dans un docker, ce qui a compliqué la tâche. Les fichiers que me créais perf étaient illisibles pour perf script .

Évaluation des performances avec Autocannon

Afin d'évaluer les performances du serveur, j'ai utilisé l'outil Autocannon pour tester les temps de réponse de plusieurs routes. Après plusieurs tests, j'ai choisi de retenir la route concert car elle contient une quantité importante de données et est constituée de différents INNER JOIN.

Les résultats ont montré que la clusterisation a permis de diviser par deux le temps de réponse du serveur pour cette route en particulier. Ces résultats sont très encourageants et montrent que l'optimisation des performances peut avoir un impact significatif sur l'expérience utilisateur.

Résultats

```
→  API-FIMU-Docker git:(master) × autocannon -c 100 -d 10 http://localhost:3000/concert
Running 10s test @ http://localhost:3000/concert
100 connections


┌─────────┬──────┬──────┬───────┬──────┬──────────┬─────────┬────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg      │ Stdev   │ Max    │
├─────────┼──────┼──────┼───────┼──────┼──────────┼─────────┼────────┤
│ Latency │ 301 ms │ 322 ms │ 358 ms │ 370 ms │ 321.41 ms │ 21.77 ms │ 483 ms │
└─────────┴──────┴──────┴───────┴──────┴──────────┴─────────┴────────┘

┌───────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 97.5%   │ Avg     │ Stdev   │ Min     │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Req/Sec   │ 248     │ 248     │ 311     │ 328     │ 306.3   │ 20.75   │ 248     │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Bytes/Sec │ 4.33 MB │ 4.33 MB │ 5.43 MB │ 5.72 MB │ 5.34 MB │ 362 kB  │ 4.32 MB │
└───────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘

Req/Bytes counts sampled once per second.
# of samples: 10

3k requests in 10.02s, 53.4 MB read
```

Sans Cluster

```
→  API-FIMU-Docker git:(master) × autocannon -c 100 -d 10 http://localhost:3000/concert
Running 10s test @ http://localhost:3000/concert
100 connections


┌─────────┬──────┬──────┬───────┬──────┬──────────┬─────────┬────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg      │ Stdev   │ Max    │
├─────────┼──────┼──────┼───────┼──────┼──────────┼─────────┼────────┤
│ Latency │ 63 ms │ 121 ms │ 233 ms │ 276 ms │ 127.76 ms │ 44.51 ms │ 456 ms │
└─────────┴──────┴──────┴───────┴──────┴──────────┴─────────┴────────┘

┌───────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 97.5%   │ Avg     │ Stdev   │ Min     │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Req/Sec   │ 493     │ 493     │ 824     │ 902     │ 775.5   │ 126.35  │ 493     │
├───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────┤
│ Bytes/Sec │ 8.6 MB  │ 8.6 MB  │ 14.4 MB │ 15.7 MB │ 13.5 MB │ 2.2 MB  │ 8.6 MB  │
└───────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘

Req/Bytes counts sampled once per second.
# of samples: 10

8k requests in 10.03s, 135 MB read
```

Avec Cluster

Activité 7 : Créer un service de gestion d'images

(Hugo Foulon)

Cette partie est dans un pdf à part car elle était déjà en PDF avant la mise en commun.