ÉNONCE TRAVAUX PRATIQUE – SUJET 4

Authentification et Autorisation

Thèmes

- Authentification
- https
- Certificats
- JWT

Présentation

Les services Web que vous avez développés jusqu'à présent, bien que fonctionnel, ne proposait pas un grand niveau de sécurité. La première étape consisterait à effectuer une authentification basique des utilisateurs. **Fastify** propose un plugin permettant cette authentification, l'exemple que je propose vous permettra appréhender le mécanisme plugin/register d'authentification de **Fastify**.
La deuxième étape sera de passer du protocole **http** à **https** pour tous les échanges relatifs à vos serveurs. Il faudra pour cela générer des certificats que vous signerez vous-même. Évidemment, les navigateurs modernes verront cela d'un mauvais œil, mais nous prendrons garde à ignorer ces avertissements uniquement lorsque nous sommes sûr de la connexion, et c'est le cas avec nos serveurs.
Et enfin, nous utiliserons les **JWT** pour effectuer une authentification plus robuste pour notre service web. Là aussi, l'usage des plugins, qui est au cœur du fonctionnement de **Fastify**, permet de lui ajouter un composant supplémentaire simplement. Le dispositif **decorate** , que vous allez découvrir en fin de sujet, permet d'ajouter, sur l'instance **Fastify** en cours, une « particularité ». Ainsi, il est facile de proposer dans certains cas une authentification ou pas.

Ressources indispensables :

- **https://jwt.io**
- **https://wiki.openssl.org/index.php/Main_Page**
- **https://fastify.dev/docs/latest/**

Étape 1 – « De base… »

Cette première étape consiste à récupérer le dépôt d'un service web utilisant le mécanisme **Basic Authentication**. Vous pouvez récupérer le projet avec la commande **git clone https://github.com/laurentgiustignano/R6.A.05-TP-Secu1** dans votre répertoire de travail. Ou bien depuis **PhpStorm**, en choisissant : **Get From Version Control** avec la même URL.

Le projet contient :

- **src/server.js** qui permet la création d'un service web sur le port **3000** et la requête **GET /**.
- **Package.json** qui décrit le projet avec l'inclusion du module **fastify** et **@fastify/basic-auth**

En analysant le fichier source src/server.js, vous pouvez répondre aux questionnements suivants et comprendre le comportement des différents appels vers ce service web. Pour cela, démarrer la configuration dev.

- Avec le logiciel **Postman**, sans préciser de paramètre supplémentaire, essayez d'accéder aux end-points **http://localhost:3000/secu** et **http://localhost:3000/dmz**.
- A l'aide du site **https://www.base64encode.org/fr/**, effectuer l'encodage des identifiants de connexion tels qu'ils sont attendus. Dans l'onglet **Headers** de **Postman**, ajouter la clé

Authorization et placer la valeur obtenue en **base64**, précédé de la mention **Basic**. Retester ensuite les end-points.

- Pour simplifier l'utilisation, décocher cette clé nouvellement créée. Dans l'onglet **Authorization**, choisissez **Basic Auth** et remplissez les valeurs en claires. Vous observerez que le **Header** est automatiquement complété avec la bonne valeur. Retester ensuite les end-points et confirmer aussi qu'une erreur dans la saisie du **username/password** rejette l'authentification.
- Déterminer maintenant le rôle de la fonction **after()** pour la déclaration de la route **'/secu'**.
- Dupliquer le code de route **'/secu'** pour créer la route **'/autre'** à l'intérieur de **after()**, mais elle doit être accessible sans authentification.

Étape 2 – Prouves qui tu es !

Pour cette partie, nous allons mettre en œuvre le protocole **https** pour notre service web. **Fastify** passe de **http** à **https** par le biais des options lors de la création de l'objet **fastify**. Par contre il faut entre autre lui fournir une clé privée RSA et le certificat signé correspondant. Ces informations sont dans des fichiers, nous aurons besoin du module **node :fs** qui contient les méthodes de lecture dans les fichiers.
Mais avant de programmer, il nous faut créer votre **Autorité de Certification** (CA), générer un **certificat** valide pour votre serveur que nous testerons avec **openssl**.

- Créer une nouvelle clé RSA de 2048 bits appelé **server.key**.
- Créer un fichier de **Certificate Signing Request** et effectuer la signature avec votre clé privée. Pour tester le certificat généré, vous pouvez utiliser la commande suivante : **openssl s_server -accept 4567 -cert server.crt -key server.key -www -state**. Et avec **Postman**, tester contacter **https://localhost:4567**. Commentez l'affichage retourné dans **Postman**.
- En vous inspirant de la documentation sur le site de **Fasify** (**https://fastify.dev/docs/latest/Reference/HTTP2**), faites évoluer le service web pour que l'accès s'effectue en **https** avec la clé privée et le certificat que vous venez de générer.

*Remarque : l'exemple de code proposé dans la documentation utilise du **CommonJS** et pas du format **ESM**, vous ne pourrez pas utiliser **__dirname** dans la fonction **readFileSync()**. Néanmoins, en parcourant la documentation de **NodeJS**, cette fonction a plusieurs formes d'écriture pour le **path**. Un autre détail, l'exemple utilise une fonction synchrone. Vous réfléchirez à l'impact de cet usage sur le serveur, et adapter si nécessaire pour une meilleure optimisation.*

Étape 3 – Un jeton dans la machine

Cette dernière partie du TP utilisera le mécanisme d'authentification JWT. Nous allons produire deux services qui fonctionneront respectivement sur les ports **3000** et **4000** :

- Un premier service s'occupera de simuler la création d'un compte et son authentification. En complément de ce dernier point, le jeton **JWT** sera généré et renvoyé. Nous l'appellerons **auth**.
- Un second service vérifiera l'authenticité du Jeton pour l'accès à une ressource protégée. Nous le nommerons **data**.

Voici les **Endpoint** qui seront proposés avec leurs rôles :

- **/signup** : Disponible sur le serveur **auth** avec la méthode **POST**. Cette route permet de créer un compte utilisateur. Les paramètres envoyés au serveur sont l'email et un mot de passe. Si ce compte n'existe pas, le mot de passe haché avec **sha256** sera généré, et un rôle choisi au hasard entre **admin** et **utilisateur**. Ces trois informations seront enregistrées dans **users** sous forme d'objets. Si le compte est déjà enregistré, un retour **401** informera le client que ce compte utilisateur est déjà enregistré. Cette dernière partie est déjà fourni.
- **/signin** : Disponible sur le serveur **auth** avec la méthode **POST**. Cette route permet de se connecter avec un compte utilisateur créé précédemment. Les paramètres envoyés au serveur sont toujours l'email et un mot de passe. Si les informations correspondent, un jeton **JWT** devra

être généré avec comme **payload** : l'email et le rôle, puis renvoyé avec le code statu **200**. Sinon, un retour avec le code statu **401** et le message : « Utilisateur non-identifié » sera retourné. La signature s'effectuera avec l'algorithme **ES256**.

- **/home** : Disponible sur le serveur **data** avec la méthode **GET**. Cette route doit être accessible avec ou sans **JWT** valide, et renvoie le **Json {'hello' : 'world'}**. Elle doit toujours être fonctionnelle.
- **/auth** : Disponible sur le serveur **data** avec la méthode **GET**. Cette route doit utiliser l'authentification **JWT** et vérifier la validité du jeton avec la clé publique **ECDSA**. Si le rôle contenu dans le jeton est **admin**, alors une clé **message** de valeur « **Full access** » sera ajouté au retour. Mais si le rôle est **utilisateur**, alors la valeur de **message** sera « **Accès limité** ».

Pour réaliser cette partie, vous pouvez récupérer le début de projet du serveur **auth** qui est disponible sur Github à partir de cette URL (**https://github.com/laurentgiustignano/R6.A.05-TP-Secu1-auth**). En plus des fonctionnalités décrites qu'il faudra implémenter, il faut construire les clés de chiffrements **ECDSA** pour les token **JWT**. La norme spécifie que des paramètres particuliers, dans le cas de l'usage d'**ESDSA** comme algorithme de chiffrement, sont expliqués ici (**https://datatracker.ietf.org/doc/html/rfc7519#section-8**). Pour la partie vérification, Le dépôt **https://github.com/laurentgiustignano/R6.A.05-TP-Secu1-data** est à récupérer pour démarrer le travail sur le server **data**.
Vous devez :

- Générer les clés de chiffrements avec **openssl** en **ECDSA** compatible avec la norme **JWT**. Vous les placerez dans le dossier **.ssl** du projet.
- Compléter l'enregistrement du plugin **fasitfyJwt** dans le fichier **src/plugins/jwt.js**.
- Compléter le handler **addUser()** dans le cas où il s'agit d'un nouveau compte utilisateur dans le fichier **src/controllers/login.js**.
- Écrire le handler **loginUser()** dans le fichier **src/controllers/login.js** pour signer et renvoyer un jeton pour un utilisateur valide.
- A partir du dépôt, développer l'enregistrement du plugin **fasitfyJwt** dans le fichier **src/plugins/jwt.js**.
- Compléter dans le fichier **src/middleware/authenticate.js** le handler **getAuthenticate()** qui « décore » l'application pour la vérification du jeton.
- Pour terminer, compléter la fonction **getAuthHandler()** qui doit retourner les informations utilisateurs complété par un message attestant son accès.

Bon courage…