IUT de Paris - Rives de Seine
Université Paris Cité

BUT 3 INFORMATIQUE

Parcours A FI

Réalisations d'applications

RAPPORT TP4

Authentification et Autorisation

Prénom et Nom de l'étudiant : Akash Selvaratnam

Groupe : 303

Promotion : 2023-2024

Sommaire

Table des matières

Étape 1 – « De base… »

Au départ, j'ai pu prendre connaissance du code présent avec le server.js qui peut être lancé sur le port 3000 en localhost avec Fastify qui propose deux différentes requêtes GET http://localhost:3000/dmz et http://localhost:3000/secu et qui valide seulement les personnes ayant pour nom d'utilisateur Tyrion et mot de passe wine.

Pour l'étape 1, j'ai pu tester via Postman deux différentes requêtes Get http://localhost:3000/dmz et http://localhost:3000/secu et observez le résultat retourné.

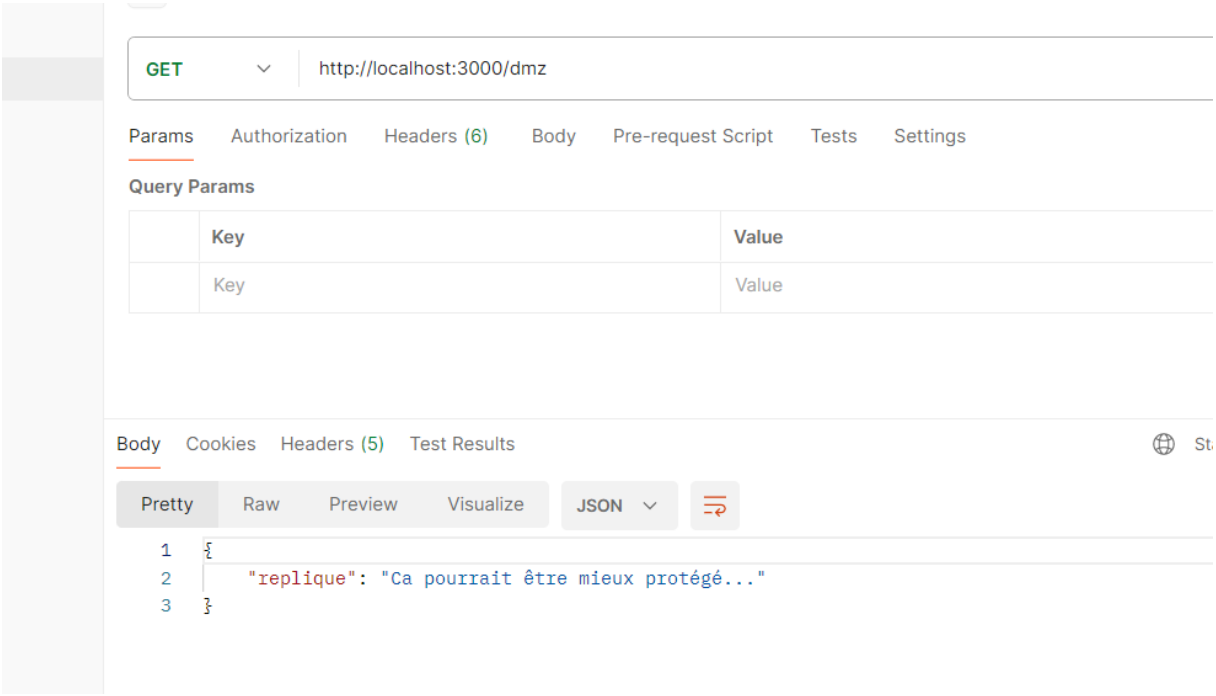

Figure 1 : Requête http://localhost:3000/dmz

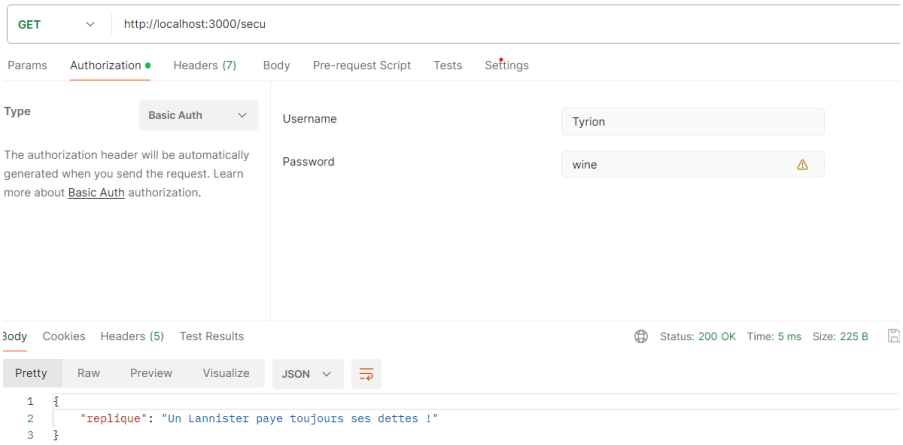

Figure 2 : Requête http://localhost:3000/secu

Ensuite, j'ai pu réaliser une authentification pour la requête http://localhost:3000/secu sur Postman en indiquant le nom ainsi que le mot de passe que nous avons indiqué dans la fonction validate du fichier server.js afin que le retour « Un Lannister paye toujours ses dettes ! » puisse être envoyé, car il est envoyé seulement s'il respecte les authentifications.

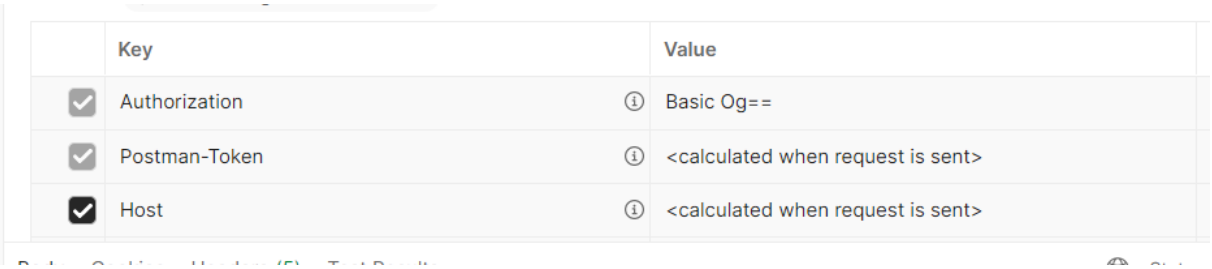

Figure 3 : Configuration de l'Authorization

Le rôle de la fonction after() est d'être exécuté lorqu'e l'ensemble des plugins ont terminé de charger, il est toujours exécuté avant la fonction ready().

Dans la fonction after(), j'ai ajouté une nouvelle route /autre du type GET, mais à la différence de la route /secu cela doit être accessible sans les authentifications donc sans le onRequest.

```
fastify.after(() : void  => {
    fastify.route( opts: {
        method: 'GET',
        url: '/secu',
        onRequest: fastify.basicAuth,
        handler: async (req : FastifyRequest<RouteGeneric, http.Server, http.IncomingMessage
            return {
                replique: 'Un Lannister paye toujours ses dettes !'
            }
        }
    })
    fastify.route( opts: {
        method: 'GET',
        url: '/autre',
        handler: async (req : FastifyRequest<RouteGeneric, http.Server, http.IncomingMessage
            return {
                replique: 'Un Ecureil paye toujours ses dettes !'
            }
        }
    })
})
```

Figure 4 : Fonction after()

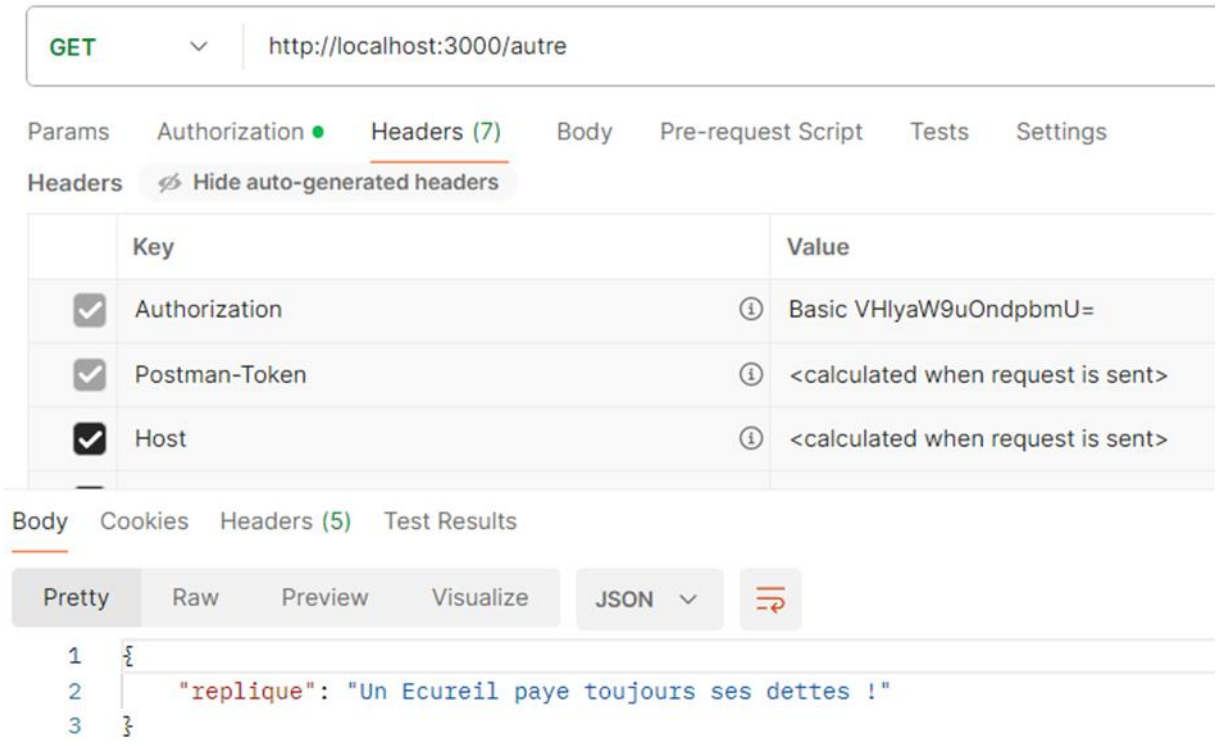

Figure 5 : Requête http://localhost:3000/autre

Étape 2 – Prouves qui tu es !

Pour l'étape 2, j'ai créé une nouvelle clé RSA de 2048 bits nommé server.key avec la commande suivante openssl genrsa -out server.key 2048 .

J'ai ensuite créé un nouveau Certificate Signing Request en signant mon certificat avec la clé privé créé précédemment à l'aide des commandes proposé sur les slides 35 et 36 et ensuite, j'ai pu tester le certificat généré via la commande proposé dans le cours permettant de vérifier le certificat sur le port 4567 avec l'application Postman.

```
<HTML><BODY BGCOLOR="#ffffff">
<pre>

s_server -accept 4567 -cert server.crt -key server.key -www -state
This TLS version forbids renegotiation.
Ciphers supported in s_server binary
TLSv1.3    :TLS_AES_256_GCM_SHA384    TLSv1.3    :TLS_CHACHA20_POLY1305_SHA256
TLSv1.3    :TLS_AES_128_GCM_SHA256    TLSv1.2    :ECDHE-ECDSA-AES256-GCM-SHA384
TLSv1.2    :ECDHE-RSA-AES256-GCM-SHA384 TLSv1.2    :DHE-RSA-AES256-GCM-SHA384
TLSv1.2    :ECDHE-ECDSA-CHACHA20-POLY1305 TLSv1.2    :ECDHE-RSA-CHACHA20-POLY1305
TLSv1.2    :DHE-RSA-CHACHA20-POLY1305 TLSv1.2    :ECDHE-ECDSA-AES128-GCM-SHA256
TLSv1.2    :ECDHE-RSA-AES128-GCM-SHA256 TLSv1.2    :DHE-RSA-AES128-GCM-SHA256
TLSv1.2    :ECDHE-ECDSA-AES256-SHA384 TLSv1.2    :ECDHE-RSA-AES256-SHA384
TLSv1.2    :DHE-RSA-AES256-SHA256     TLSv1.2    :ECDHE-ECDSA-AES128-SHA256
TLSv1.2    :ECDHE-RSA-AES128-SHA256   TLSv1.2    :DHE-RSA-AES128-SHA256
TLSv1.0    :ECDHE-ECDSA-AES256-SHA    TLSv1.0    :ECDHE-RSA-AES256-SHA
SSLv3      :DHE-RSA-AES256-SHA        TLSv1.0    :ECDHE-ECDSA-AES128-SHA
```

Figure 6 : Test du Certificat sur Postman

J'ai ensuite configuré Fastify en HTTPS en indiquant le chemin vers ma clé privée ainsi que vers ma certification, j'ai du également importer des modules telles fs pour pouvoir lire les fichiers et le module path.

```
const fastify : FastifyInstance<…> & PromiseLike<…>  = Fastify( opts: {
    logger: true,
    http2: true,
    https: {
        key: fs.readFileSync(path.join(__dirname, '..', 'clés', 'server.key')),
        cert: fs.readFileSync(path.join(__dirname, '..', 'clés', 'server.crt')),
    }
})
```

Étape 3 – Un jeton dans la machine

Pour l'étape 3, j'ai créé une clé privée ainsi qu'une clé publique avec openssl en ECDSA qui est compatible avec la norme JWT. Pour Savoir, si ma clé est compatible, j'ai consulté la documentation (lien : RFC 7519 - JSON Web Token (JWT) (ietf.org) qui m'a indiqué que P-256 était compatible avec l'algorithme ECDSA. J'ai ensuite lancé la commande suivante : openssl ecparam -list_curves pour voir la clé correspondant à P-256 qui est prime256v1. J'ai pu ensuite créer ma clé privée et ma clé publique.

Pour pouvoir configuré mon fastifyJwt, j'ai tout simplement indiqué le chemin vers clé publique et le chemin vers ma clé privé.

```
app.register(fastifyJwt, {
    sign: {
        algorithm: 'ES256',
        issuer: 'info.iutparis.fr'
    },
    secret: {
        allowHTTP1 : true,
        private : fs.readFileSync(path.join(__dirname, '..','..','.ssl', 'private_key.pem')),
        public: fs.readFileSync(path.join(__dirname,  '..','..', '.ssl', 'public_key.pem')),
    },
})
```

Figure 7 : Configuration fastifyJwt

J'ai ensuite complété la fonction addUser() en créant un objet contenant le mail de l'utilisateur, son mot de passe haché ainsi que son rôle via la fonction random de la librairie Math. J'ai ajouté cet objet dans mon tableau d'utilisateur.

```
else{
  let obj :{…}  = {
      email : email,
      password : hashedPassword,
      type : Math.floor( x: Math.random() * 2)  === 0 ? "admin" : "utilisateur",
  }
  users.push(obj);
  res.status(200).send("Utilisateur bien enregistré")
}
```

Figure 8 : Fonction addUser()

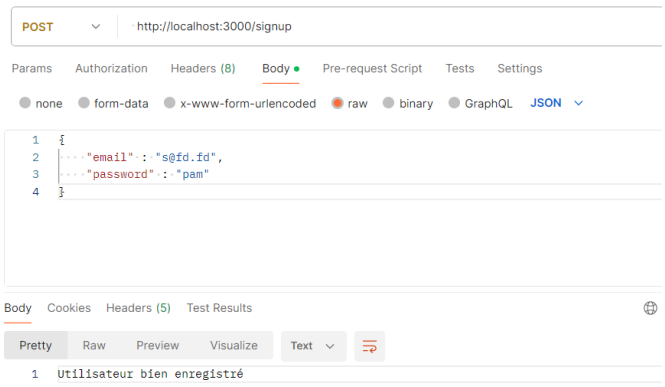

Figure 9 : Résultat Ajout d'un utilisateur

J'ai créé la fonction loginUser() en récupérant le mail ainsi que le mot de passe, puis en hachant le mot de passe et en vérifiant que l'utilisateur existe, si l'utilisateur existe alors je crée un jeton avec la fonction jwtSign proposé par Fastify qui contient le mail ainsi que le rôle de l'utilisateur en tant que payload et je renvoie cette information avec un statu 200 sinon je renvoie le message « utilisateur non-identifié » avec le code statu 401.

```
1+ usages   Akashos23
export const loginUser = async function (req, res) : Promise<void>  {
    const {email, password} = req.body
    const hashedPassword : string  = createHash( algorithm: "sha256").update(password).digest().toString( encoding: "hex")

    let user = users.find((u) => u.email === email && u.password === hashedPassword)
    if(user){
        const jeton = await res.jwtSign({
            email : email,
            role : user.type
        })

        res.status(200).send({jeton})
    }
    else{
        res.status(401).send("Utilisateur non-identifié")
    }
}
```

Figure 10 : Fonction loginUser()

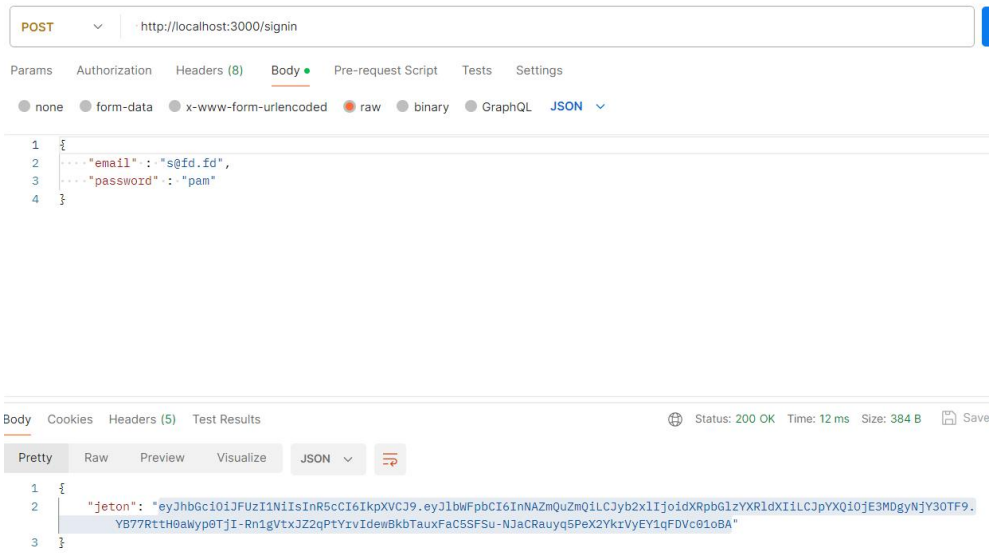

Figure 11 : Résultat connexion d'un utilisateur

J'ai ensuite complété le fastifyJwt dans le deuxième service de la même façon que le premier service en créant un répertoire .ssl qui contient mes clé privé et publique.

J'ai complété la fonction getAuthentificate() en indiquant sur Postman, dans authorization, OAuth 2.0 et en complétant le champ token avec le résultat que j'ai obtenue lors de la connexion.

Dans ma fonction getAuthenticate(), j'ai récupéré mon token puis j'exécute la fonction jwtVerify de la librairie Fastify qui prend en argument le token et qui permet de vérifier que le token est bien valide et existant.

```
export async function getAuthenticate(req, res) : Promise<void>  {
    try {
        const authHeader = req.headers['authorization']
        const token = authHeader && authHeader.split(' ')[1]

        const decode = await req.jwtVerify(token)


    } catch (err) {
        res.code(401).send({...err, message: "Vous ne passerez pas !"})
    }
}
```

Figure 12 : Fonction getAuthenticate

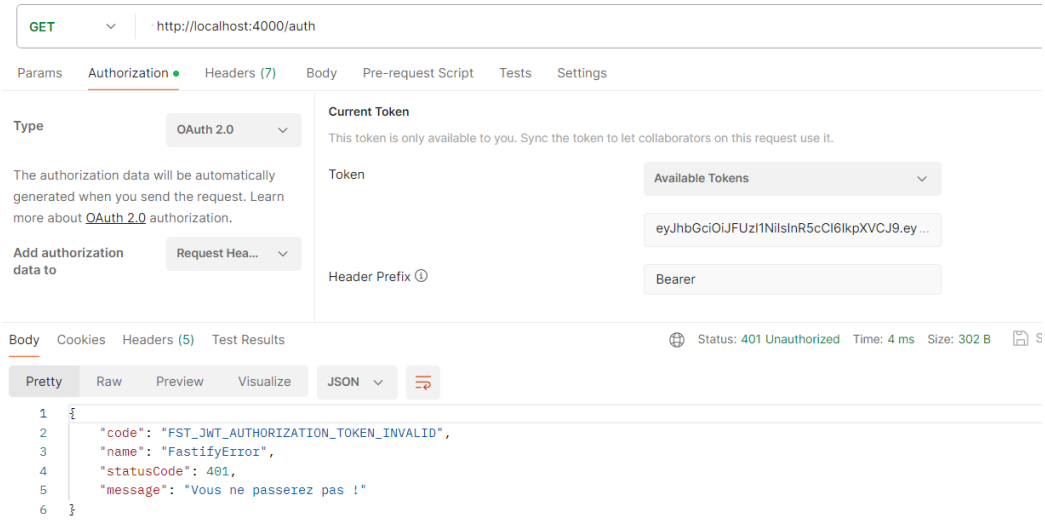

Figure 13 : Résultat d'un token non valide

Pour finir, j'ai complété la fonction getAuthHandler() en récupérant le token puis en utilisant la fonction jwtVerify qui permet également de récupérer le contenu (mail, rôle) du token, je vérifie le rôle de l'utilisateur, si l'utilisateur est un admin alors j'envoie le message suivant « Full Access » sinon « Accès limité ».

```
export const getAuthHandler = async function (req, rep) : Promise<void>  {
    const authHeader = req.headers['authorization']
    const token = authHeader && authHeader.split(' ')[1]

    const decode = await req.jwtVerify(token)

    if(decode.role === "admin"){
        rep.send("Full Access")
    }
    if(decode.role === "utilisateur"){
        rep.send("Accès limité")
    }
}
```

Figure 14 : Fonction getAuthHandler

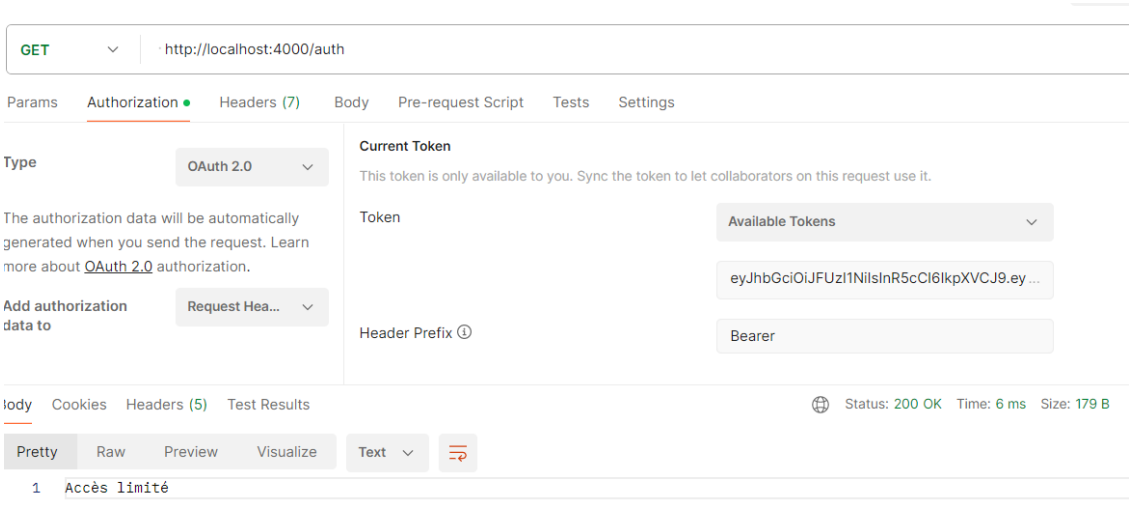

Figure 15 : Résultat d'une personne ayant le rôle d'utilisateur