

Rapport

Étape 1

Quand je fais la requête get sur `http://localhost:3000/secu` j'obtiens la réponse suivante :

```
{
  "replique": "Tu ne sais rien, John Snow.."
}
```

Après lecture du code, on voit que cette réponse est envoyée si l'erreur 401 est renvoyé. Il semble avoir eu un problème lors de la requête.

```
fastify.setErrorHandler( handler: function (err, req, reply) {

  if (err.statusCode === 401) {
    console.log(err)
    reply.code( statusCode: 401 ).send( payload: {replique: 'Tu ne sais rien, John Snow..'})
  }
  reply.send(err)
})
```


Quand je fais la requête get sur `http://localhost:3000/dmz` j'obtiens la réponse suivante :

```
{
  "replique": "Ca pourrait être mieux protégé..."
}
```

Après inspection du code on voit bien que c'est la seule réponse que la page renvoie, ce n'est pas surprenant d'y recevoir cette réponse.

```
fastify.get('/dmz', {}, (req, res) => {
  res.send( payload: {replique: "Ca pourrait être mieux protégé..."} )
})
```

Après avoir essayé d'ajouter une Authorization dans le header et demande au professeur de l'aide pour cette partie, j'ai trouvé l'onglet Authorization dont j'ai changé en basic Auth afin de pouvoir y entrer les informations corrects qu'on peut trouver dans la fonction valide du code.

Username	<input type="text" value="Tyrion"/>
Password	<input type="password" value="wine"/> 

Après alors la réalisation sur la requête avec une erreur soit, <http://localhost:3000/secu>, je reçois maintenant une réponse différente qui correspond au résultat attendu dans le code.

```
"replique": "Un Lannister paye toujours ses dettes !"
```

En regardant plus en détails l'header de la requête on voit que postman a complété automatiquement la valeur Authorization que je n'avais pas trouvée, j'imagine que le chiffrement correspond au chiffrement du nom d'utilisateur ainsi que du mot de passe.

```
--header 'Authorization: Basic  
VHlyYW9uOndpbmU=' \
```

Après décodage en UTF-8, j'ai compris qu'il fallait encoder le nom d'utilisateur suivi du mot de passe séparé par ':'.

Dans le code, on voit que l'erreur obtenue avant vient sûrement du `fastify.basicAuth` qui renvoie à la fonction `validate` qui renvoie une erreur si le username ou le mot de passe est incorrect.

```
fastify.after(() => {  
  fastify.route({ opts: {  
    method: 'GET',  
    url: '/secu',  
    onRequest: fastify.basicAuth,  
    handler: async (req, reply) => {  
      return {  
        replique: 'Un Lannister paye toujours ses dettes !'  
      }  
    }  
  })  
})  
})
```

Après des recherches sur internet, je me suis retrouvé sur le lien suivant:

<https://fastify.dev/docs/latest/Reference/Server/#after>

after

Invoked when the current plugin and all the plugins that have been registered within it have finished loading. It is always executed before the method `fastify.ready`.

La partie d'exécution avant `fastify.ready` ne nous intéresse pas, cependant le fait qu'il est exécuté après que les plugins soient chargés permet alors de s'assurer que les plugins nécessaires au bon fonctionnement de l'authentification sont présents.

Après la copie de la route `secu`, j'ai simplement changé la route en `autre` ainsi que retirer entièrement le `onRequest` qui faisait à l'authentification.

```
fastify.route({
  method: 'GET',
  url: '/autre',
  handler: async (req, reply) => {
    return {
      replique: 'Bravo !'
    }
  }
})
```

Étape 2

Pour la réalisation du certificat je me suis aidé de la page 35 et 36 du cours.

J'ai exécuté les commandes affichées dans le cours dans un fichier `keys` dans le projet j'obtiens alors le résultat suivant :

```

v  keys
  server.crt
  server.key
  server.req
```

Après l'exécution des commandes demandées dans le tp ainsi que la requête postman, je reçois le code html suivant dont je n'y comprends rien :

```
<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
TLSv1.0      :ECDHE-RSA-AES128-SHA      SSLv3       :DHE-RSA-AES128-SHA
TLSv1.2      :RSA-PSK-AES256-GCM-SHA384 TLSv1.2      :DHE-PSK-AES256-GCM-SHA384
TLSv1.2      :RSA-PSK-CHACHA20-POLY1305 TLSv1.2      :DHE-PSK-CHACHA20-POLY1305
```

J'importe les modules nécessaires pour récupérer le chemin du dossier actuel soit le src.

```
import * as fs from "fs"
import * as path from "path"
import { fileURLToPath } from 'url'
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
```

Avec l'aide de la fonction path.join, je crée le chemin vers le dossier keys, puis le certificat ainsi que la clé privée qui sont demandés.

```
const fastify = Fastify({ opts: {
  logger: true,
  http2: true,
  https: {
    key: fs.readFileSync(path.join(__dirname, '..', 'keys', 'server.key')),
    cert: fs.readFileSync(path.join(__dirname, '..', 'keys', 'server.crt'))
  }
}
```

Après le lancement du nouveau programme, on obtient maintenant bien des liens en https.

Étape 3

Afin de générer les clés je me suis aidé du site suivant :

<https://www.scottbrady91.com/openssl/creating-elliptical-curve-keys-using-openssl>

J'ai ensuite utilisé les commandes suivantes pour créer la clé privée ainsi que publique.

```
# generate a private key for a curve
openssl ecparam -name prime256v1 -genkey -noout -out private-key.pem

# generate corresponding public key
openssl ec -in private-key.pem -pubout -out public-key.pem
```

J'ai rajouté à l'initialisation de fastify le paramètre suivant qui permet de localiser la clé privée et publique.

```
secret: {
  allowHTTP1 : true,
  private : fs.readFileSync(path.join(__dirname, "..", "..", ".ssl", "private-key.pem")),
  public : fs.readFileSync(path.join(__dirname, "..", "..", ".ssl", "public-key.pem")),
}
```

Pour l'ajout d'un utilisateur j'ai rajouté la condition else pour la partie où l'utilisateur n'est pas enregistré, j'ai utilisé le module Math pour générer un nombre aléatoire entre 0 et 1 afin de déterminer le rôle du compte avant d'enregistrer toutes les informations et de le rajouter dans le tableau d'utilisateurs, je renvoie ensuite un message de confirmation pour confirmer que l'ajout de l'utilisateur s'est bien réalisé.

```
else{
  let random = Math.random();
  let role = random > 0.5 ? "admin" : "utilisateur"

  user = {email, password: hashedPassword, role};
  users.push(user);
  res.status(200).send({
    message: "Utilisateur enregistré",
    user
  })
}
```

J'ai copié la première partie de AddUser pour vérifier qu'un utilisateur se trouve déjà dans le tableau ou non, si l'utilisateur existe il créer un token avant de l'envoyer sinon le programme ne fait rien, il aurait été possible d'ajouter un message de refus avec un code d'erreur.

```
export const loginUser = async function (req, res) {
  let {email,password} = req.body
  const hashedPassword = createHash({algorithm: "sha256"}).update(password).digest().toString({encoding: "hex"})
  let user = users.find((u) => u.email === email && u.password === hashedPassword)
  email = user.email
  let role = user.role
  if (user) {
    const token = await res.jwtSign({email, role});

    res.status(200).send({
      message: "Connexion réussie",
      token
    })
  }
}
```

Pour le fichier jwt du nouveau projet, j'ai simplement repris entièrement celui du projet précédent.

Je récupère le token dans le header avant de vérifier avec la fonction `jwtVerify` qui lancera une erreur si le token est invalide. J'envoie ensuite le role dans un header pour passer l'information et éviter que les futures doivent décoder le token à leur tour.

```
try {  
  const authHeader = req.headers['authorization']  
  const token = authHeader && authHeader.split(' ')[1]  
  
  if(token){  
    const decoded = await req.jwtVerify(token);  
    req.headers['role'] = decoded.role  
  }  
  else{  
    throw new Error("No token");  
  }  
} catch (err) {
```

Enfin dans `handler.js` je récupère le rôle dans le header et je vérifie sa valeur pour retourner le message correspondant.

```
const role = req.headers['role']  
  
console.log(role)  
  
if(role === "admin"){  
  res.send({message: "Full access"})  
}  
  
else if(role === "utilisateur"){  
  res.send({message: "Limited access"})  
}  
  
else {  
  res.send({message: "role invalide ??? comment t'as fait frère ???"})  
}
```

Conclusion

J'ai appris lors de ce tp, comment sécuriser un site que ce soit à l'aide de token pour des api ou des certificats pour un site. J'ai du m'aider de beaucoup de documentation en ligne n'étant pas familier avec ses modules.