Exploration sécurisation Node.JS

NodeJS, ExpressJS, EJS, Mongoose

Rédigé par

David ROUMANET
Professeur BTS SIO

Changement

Date	Révision
Août 2021	Création
Mars 2024	Ajout des outils d'analyse des vulnérabilités et gestion des rôles
Avril 2024	Ajout sécurisation des données sensibles dans le code (dotenv)

Sommaire

A Introduction

Dans le cadre du bloc 3 Cybersécurité, il est intéressant de créer une application sécurisée. Il y a plusieurs axes de sécurisation :

- Au niveau réseau, utiliser un protocole chiffré entre le serveur et les clients
- Au niveau application, mettre en œuvre un mécanisme d'authentification
- Au niveau application, gérer la qualité du code et un développement non régressif
- Au niveau SGBD, sécuriser les échanges entre le serveur et l'application

A.1 Pré-requis

Pour suivre correctement ce tutoriel, il vous faut :

- Le projet Template-Express ou le projet Pharmacie Sautheuz opérationel
- Connaître le fonctionnement des certificats SSL
- Connaître les cookies de session

D'un point de vue technique :

- Si projet Template-Express, Avoir MongoDB installé sur le poste (port 27017)

A.2 Objectifs

Pour sécuriser ce code, nous allons modifier l'application existante, étudiée en Bloc 2 spécialisation.

À l'issue de cette activité, vous serez capable :

- de sécuriser les communications avec HTTPS
- de sécuriser l'application en ajoutant un système d'authentification JWT
- de sécuriser et améliorer la qualité du développement d'une application

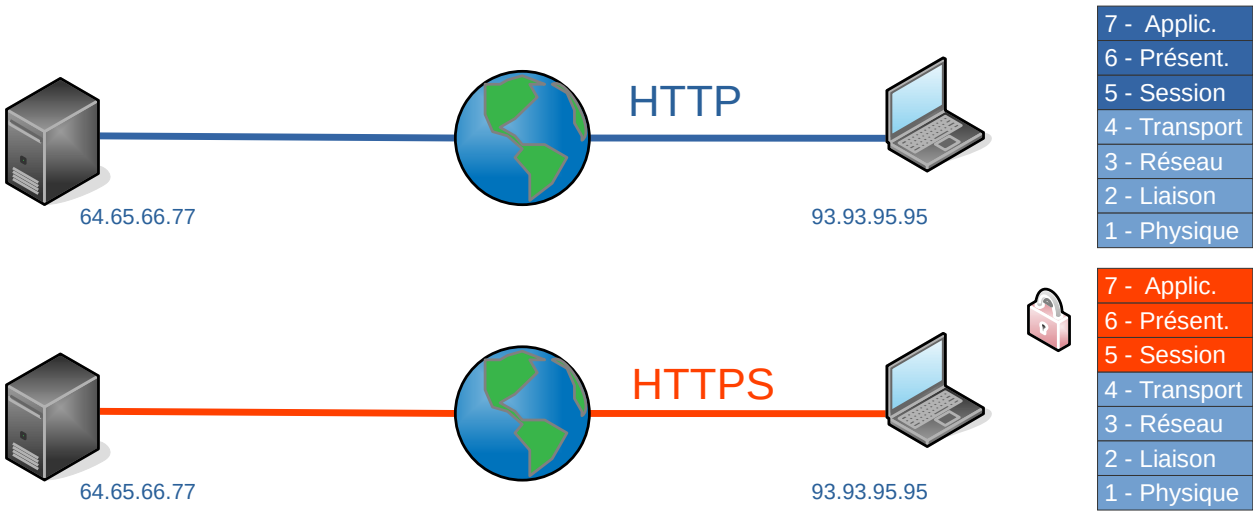
B Sécurisation de la communication

La première sécurisation consiste à utiliser le protocole HTTPS pour que les données transitant entre le serveur et les clients ne soient pas interceptées.

B.1 Rappel

Le protocole HTTPS s'appuie sur le protocole TCP (et notamment les échanges "3 way handshake") et utilise les primitives HTTP : simplement, on insère une couche intermédiaire de chiffrement, rendant la communication impossible à déchiffrer entre le serveur et le client. C'est un chiffrement de bout en bout.

Cependant, si votre conversation est indéchiffrable, on peut déterminer le destinataire, par son adresse IP. La connaissance des couches du modèle OSI de l'ISO montre que le chiffrement se fait dans les couches hautes (contrairement au fonctionnement d'un VPN IPSec ou GRE qui va chiffrer jusqu'à la couche 3).



Les protocoles sécurisés par TLS/SSL ne protègent donc pas les trames IP mais uniquement leur contenu.

- IMAPS
- FTPS
- SMTPS
- HTTPS
- LDAPS

B.2 NodeJS et HTTPS

Pour commencer, il faut ajouter un répertoire **certificate** à la racine du projet.

B.2.1 Génération des certificats

Il faut ensuite générer un certificat avec l'outil openssl :


```
openssl req -nodes -new -x509 -keyout server.key -out server.cert
Generating a RSA private key
.............+++++
.........................................+++++
writing new private key to 'server.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:Isere
Locality Name (eg, city) []:Grenoble
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Lycee A. Berges
Organizational Unit Name (eg, section) []:BTS-SIO
Common Name (e.g. server FQDN or YOUR name) []:localhost
Email Address []:void@free.fr
```

Cela permet de générer la clé privée (.key) et le certificat avec la clé publique (.cert).

Vérifiez que le certificat est bien valide avec la commande suivante :

```
openssl x509 -in server.cert -text -noout
```

Le résultat doit être similaire à celui-ci (en jaune, les informations saisies, en orange, la date de validité) :

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            1c:1e:b8:df:26:60:83:a7:29:cf:ae:60:6f:4c:c1:c9:ce:8f:67:8a
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = FR, ST = Isere, L = Grenoble, O = Lycee A. Berges, OU = BTS-SIO, CN
= localhost, emailAddress = void@free.fr
        Validity
            Not Before: Nov 23 22:20:31 2022 GMT
            Not After : Dec 23 22:20:31 2022 GMT
        Subject: C = FR, ST = Isere, L = Grenoble, O = Lycee A. Berges, OU = BTS-SIO, CN
= localhost, emailAddress = void@free.fr
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
```

Pour aller plus loin, vous pouvez consulter la ressource Openssl Tutorial (Phoenixnap).

B.2.2 Modification du code

Clonez le projet git droumanet/ExpressTemplate pour travailler cette exploration.

Dans un premier temps, il faut ajouter au début du fichier, les modules '**fs**' (pour filesystem) et '**https**'. S'il n'est pas déjà présent, il faut aussi ajouter le module '**path**' :

app.js

```
const fs = require('fs')
const https = require('https')
const path     = require('path')
```

À la fin du fichier, avant le démarrage de l'application, il faut lire les certificats (placer les deux fichiers `server.key` et `server.cert` dans le dossier `certificate`) et créer une variable **options** :

```
const key = fs.readFileSync(path.join(__dirname, 'certificate', 'server.key'));
const cert = fs.readFileSync(path.join(__dirname, 'certificate', 'server.cert'));
const options = { key, cert };
```

Enfin, il faut modifier l'écoute du serveur en créant un serveur utilisant l'option pour HTTPS, en remplaçant complètement le code http.listen() par :

```
https.createServer(options, app).listen(port, () => {
    console.log(`server running HTTPS. Go to https://localhost:${port}`);
  });
```

Votre serveur doit maintenant démarrer avec la commande nodemon app.js ou node app.js

```
nodemon app.js
[nodemon] starting `node app.js`
server running HTTPS. Go to https://localhost:3000
connected to database Hopital
```

C Mécanismes d'authentification

On différencie l'identification (fournir un identifiant personnel) de l'authentification (la preuve que l'identifiant fournit est celui de la personne devant l'application).

Un **identifiant** (comme en base de données) et un identifiant unique qui identifie une seule personne dans un système. En France, le numéro de sécurité sociale est un bon exemple, car il ne peut y avoir deux personnes vivantes, avec le même numéro.

Une **authentification** ajoute une preuve, avec un code ou un mot de passe que seule, la personne concernée peut connaître.

Il existe plusieurs mécanismes d'authentification et ils répondent chacun à un besoin particulier :

- Authentification basique (un identifiant et un mot de passe)
- Authentification par session (cookie)
- Authentification par Token (jeton)
- Authentification unique (SSO - Single Sign On)
- Oauth 2.0
- Authentification multi-facteur (2FA)
- Sans mot de passe (ex : envoi d'un lien par courriel)

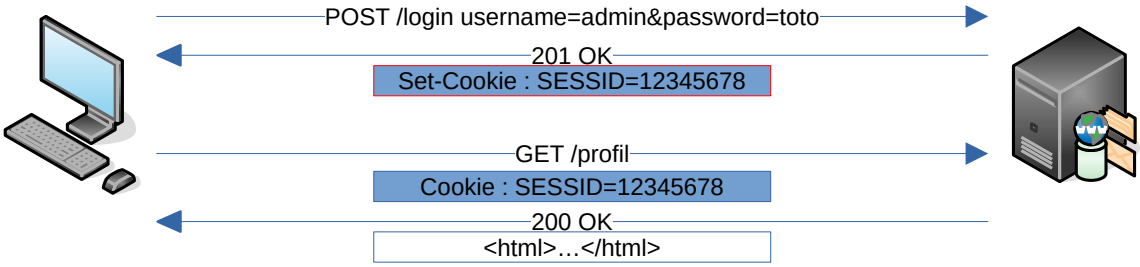
Dans le cadre du BTS SIO, les deux technologies intéressantes sont le cookie de session et le jeton web (JWT). La différence est subtile, mais importante, pour une personne ayant à se connecter sur plusieurs terminaux :

- Avec le **cookie de session**, l'authentification est stockée sur le poste client, dans un cookie créé par le serveur. Il est unique et ne peut donc pas être réutilisé sur un autre terminal. Résultat : l'ouverture d'une session sur un autre terminal, coupe automatiquement la session du premier appareil.
- Avec le **jeton web**, l'authentification est une clé signée, valable pour une durée déterminée. Si l'utilisateur se connecte sur plusieurs postes, il aura le même jeton, qu'il transmettra à chaque requête (d'où l'importance de l'utiliser avec un protocole sécurisé, comme HTTPS). Ce jeton sera reconnu, indépendamment de la session.

C.1 Cookie de session versus jeton web

C.1.1 Cookie-session

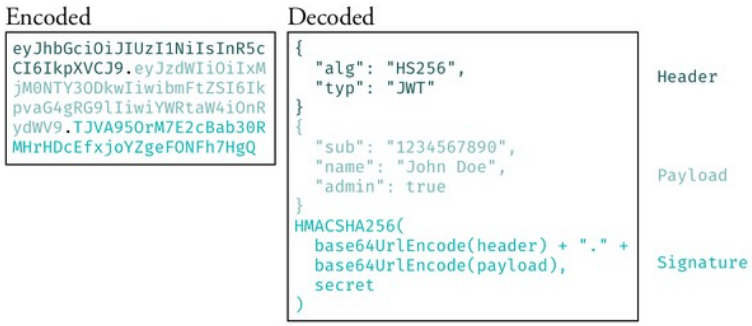
Lorsqu'un utilisateur se connecte à l'application, un cookie de session est généré dans la mémoire du serveur et est envoyé à l'utilisateur via une instruction du 'header' (set-Cookie). Ensuite, le navigateur du client renvoie ce cookie à chaque requête vers le serveur, prouvant qu'il est toujours connecté.



Ce système est tout aussi fiable que les tokens mais a un inconvénient : il ne permet pas la gestion d'un même utilisateur sur plusieurs terminaux simultanément[1]. En effet, le cookie étant mémorisé coté serveur, il est écrasé lors d'une nouvelle authentification.

C.1.2 JWT

Le principe de fonctionnement est très similaire, mais le serveur ne stocke pas l'état de la connexion (pas de cookie). Pour valider l'authentification, on utilise un token, qui contient un header, un payload et une signature[2].



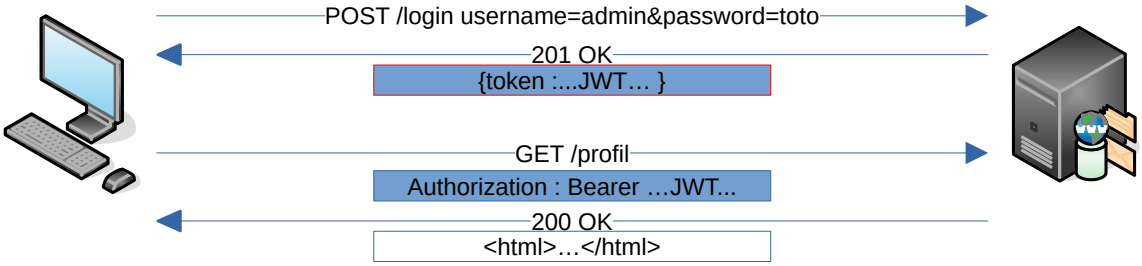
Il s'agit donc d'une chaîne de caractères encodés.

1 Voir https://www.vaadata.com/blog/fr/jetons-jwt-et-securite-principes-et-cas-dutilisation/
2 Voir https://jwt.io/

La partie algorithme propose HS256 (pour le chiffrement asymétrique RSA et empreinte SHA-256) ou HS256 (chiffrement symétrique HMAC et empreinte SHA-256). Contrairement à un cookie :

- la quantité de données n'est pas limitée,
- un token peut-être utilisé sur plusieurs applications (SSO) ou plusieurs équipements, notamment des API REST,
- le contenu est signé (garantie d'intégrité)

Le mode de fonctionnement de JWT est très similaire à celui du cookie de session.



L'inconvénient du jeton (token) est que les données ne sont pas obligatoirement chiffrées, il est donc impératif d'utiliser HTTPS.

Le site PrimeFX.com JWT contient des informations très bien détaillées.

C.2 Authentification JWT dans un projet NodeJS

C.2.1 Modules nécessaires

Dans ce projet, les modules suivants ont été ajoutés avec `npm` :

- installation de `jsonwebtoken` (le module de gestion pour JWT)
- installation de `cors` (permet de gérer la provenance des requêtes, voir Cross-Scripting)
- installation de `morgan` (permet d'afficher des informations plus lisibles dans les logs)

Les modifications de code touchent le code principal (`app.js`), ajouter un fichier de route et contrôleur.

app.js

```
const cors = require('cors')            // Cross Origin Resource Sharing
const morgan = require('morgan')        // logs pour authentification par token
const loginRoutes = require('./routes/loginRoutes')
...
// Utilisation des midlewares pour l'authentification
app.use(cors())
app.use(morgan('tiny'))
... au niveau des routes...
app.use('/login', loginRoutes)
```

Le contrôleur associé à ce code est le suivant :

loginControllers.js

```
// Contient la logique nécessaire à l'authentification
const jwt = require('jsonwebtoken')     // ajout token sécurisé
const SECRET = 'appKey'                 // normal. dans un fichier de  conf.

var loginController={
    login(req,res){
        if (!req.body.username || !req.body.password) {
            return res.status(400).json({ message: `Erreur authentif.` })
        }

        // Simulation d'un utilisateur géré dans une BDD
        if (req.body.username == "admin" && req.body.password =="azerty123") {
            const token = jwt.sign(
                { id: 0, username: req.body.username},
                SECRET,
                { expiresIn: '1 hours'}
            )
            return res.json({ access_token: token})
        } else {
            return res.status(400).json({ message: `Erreur D'authentif.` })
        }
    },

    Error(req, res) {
         res.render('404')
    }
}

module.exports = loginController;
```

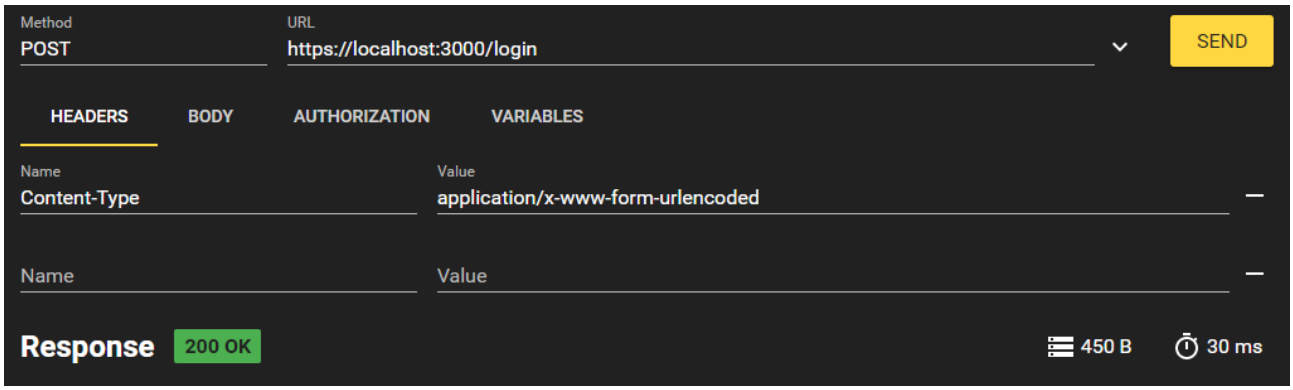
Et enfin le routeur associé au login.

loginRoutes.js

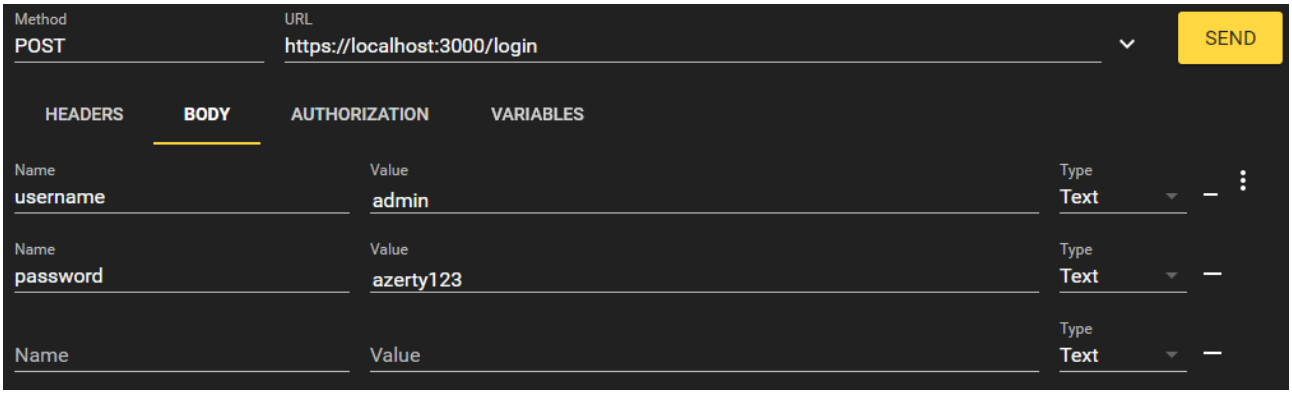
```
const loginCtrl = require('../controllers/loginControllers')
const express =  require('express')
// Instantiation du router pour permettre la gestion des requêtes HTTP (get, post, etc.)
const router = express.Router()

// Une authentification se passe par méthode POST
router.post('/',loginCtrl.login)


// exportation du module (pour le rendre utilisable dans un autre fichier)
module.exports = router
```

Désormais, il est possible de tester le code en créant une requête simulant un formulaire, avec Postman ou RESTer.

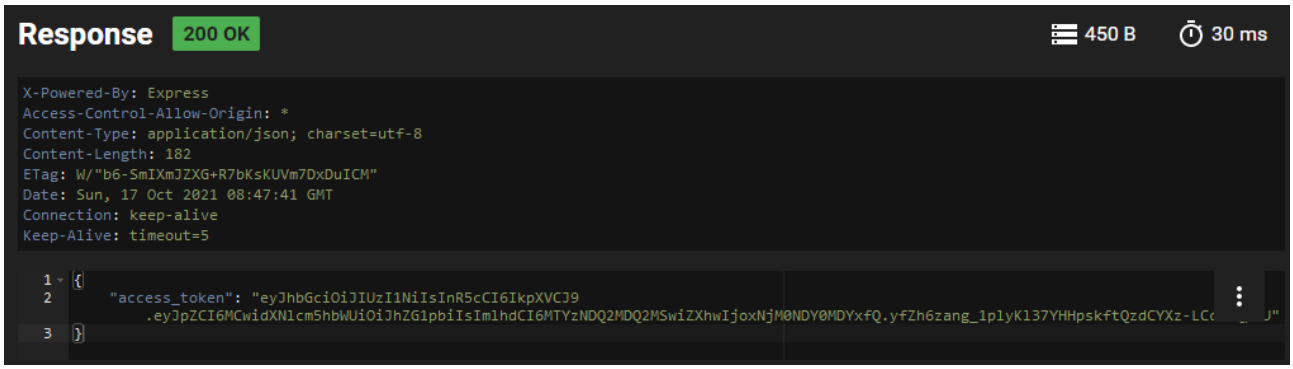
Dans la partie HEADER :



Dans la partie BODY :



Si tout va bien, la requête affiche une information similaire :



La suite consiste à faire les choses suivantes :

1. Créer un middleware qui vérifiera si l'utilisateur est bien authentifié avant d'afficher une page
2. Mettre en place le formulaire d'authentification (page EJS)

C.2.2 Middleware

Dans `app.js`, l'appel à la route générale `/medecins/` doit désormais être sécurisé. La ligne doit être modifiée comme suit :

app.js

```
app.use('/medecins/', middlewares.checkCookieJWT, medecinsRoutes)
```

Le routage pourra être effectif, seulement si la vérification du JWT fonctionne et reconnaît le jeton toujours actif.

Le code du middleware lui-même (à la racine du projet) :

middleware.js

```
const dotenv = require('dotenv')
const jwt = require("jsonwebtoken")

dotenv.config()

module.exports = {
    // vérification du cookie contenant le jeton JWT
    checkCookieJWT: (req, res, next) => {
        let token = req.cookies.access_token
        // Si le token n'existe pas
        if (!token) {
            return res.render("main", {message: "erreur : le token n'existe pas."})
        }
        try {
            // Cas de l'authentification réussie : on passe au middleware suivant
            let data = jwt.verify(token, process.env.TOKEN_SECRET)
            console.log(data)
            return next()
        } catch{
            // Cas du cookie dépassé ou du token dépassé
            console.log("Token ou cookie dépassé")
            return res.render("main", {message: {type:"error", msg: "erreur : Veuillez vous
authentifier."}})
        }
    }
}
```

Le JWT (prononcer 'djot' en anglais) doit être stocké dans un cookie : si le cookie disparaît ou n'est plus valide, il doit y avoir une gestion des erreurs. La durée d'existence du jeton doit être supérieur à la durée du jeton.

C.2.3 Formulaire d'authentification

Sous la ligne d'appel du middleware pour les routes vers /medecins/ on ajoute le middleware d'authentification :

app.js

```
app.use('/medecins/', middlewares.checkCookieJWT, medecinsRoutes)
app.use('/login', loginRoutes)
```

Puis une route dans le répertoire des routes :

loginRoutes.js

```
const loginCtrl = require('../controllers/loginControllers')
const express =  require('express')
// Instantiation du router pour permettre la gestion des requêtes HTTP (get, post, etc.)
const router = express.Router()

// Une authentification se passe par méthode POST
router.post('/',loginCtrl.login)
router.get('/', loginCtrl.formLogin)


// exportation du module (pour le rendre utilisable dans un autre fichier)
module.exports = router
```

Le contrôleur correspondant :

loginControllers.js

```
// Contient la logique nécessaire à l'authentification
const jwt = require('jsonwebtoken')     // ajout token sécurisé
const dotenv = require('dotenv')        // le secret y est stocké
dotenv.config()

let loginController={
    // lorsque l'utilisateur doit saisir les identifiants pour s'authentifier
    formLogin(req, res) {
        console.log("Affiche authentification")
        res.render('authentification')
    },

    // vérification des identifiants et création éventuelle du token
    login(req,res){
        if (!req.body.username || !req.body.password) {
            return res.status(400).json({ message: `Erreur D'authentification.` })
        }

        // Simulation d'un utilisateur géré dans une BDD (prod: utiliser une requête SQL)
        if (req.body.username == "admin" && req.body.password =="azerty123") {
            const token = jwt.sign(
                { id: 0, username: req.body.username},
                process.env.TOKEN_SECRET,
                { expiresIn: '120s'}     // 120 secondes pour pouvoir tester. En prod : 600s
            )
            // renvoie vers la page authSuccess et transmet  la "variable" access_token
            res.cookie("access_token", token, {httpOnly: true, secure: true})
            .render('main', {message: {type:"success", msg: "Authentification réussie
("+token+")"}})
        } else {
            return res.status(400).json({ message: `Erreur D'authentification.` })
        }
    },

    Error(req, res) {
         res.render('404')
    }
}

module.exports = loginController;
```

Après un contrôleur, viennent les vues :

authentification.ejs

```
<%- include('header'); %>
    <h1>Authentification</h1>

    <form action="/login" method="POST">
        <div>
           <div class="col-sm-12 col-md-6">
               <div class="input-group fluid">
                   <span class="tooltip" aria-label="Saisir identifiant"><label
for="frmUsername">Identifiant</label></span>
                   <input type="text" name="username" id="frmUsername"
placeholder="identifiant"">
               </div>
               <div class="input-group fluid">
                   <span class="tooltip" aria-label="Saisir le mot de passe à l'abri des
regards"><label for="frmPass">Mot de passe</label></span>
                   <input type="password" name="password" id="frmPass" placeholder="surprise">
               </div>

               <div class="input-group fluid">
                   <input type="submit" value="Valider">
               </div>
            </div>
        </div>
    </form>
    <%- include('footer'); %>
```

Enfin, à la racine du projet, placer un fichier `.env` qui contiendra le jeton secret :

.env

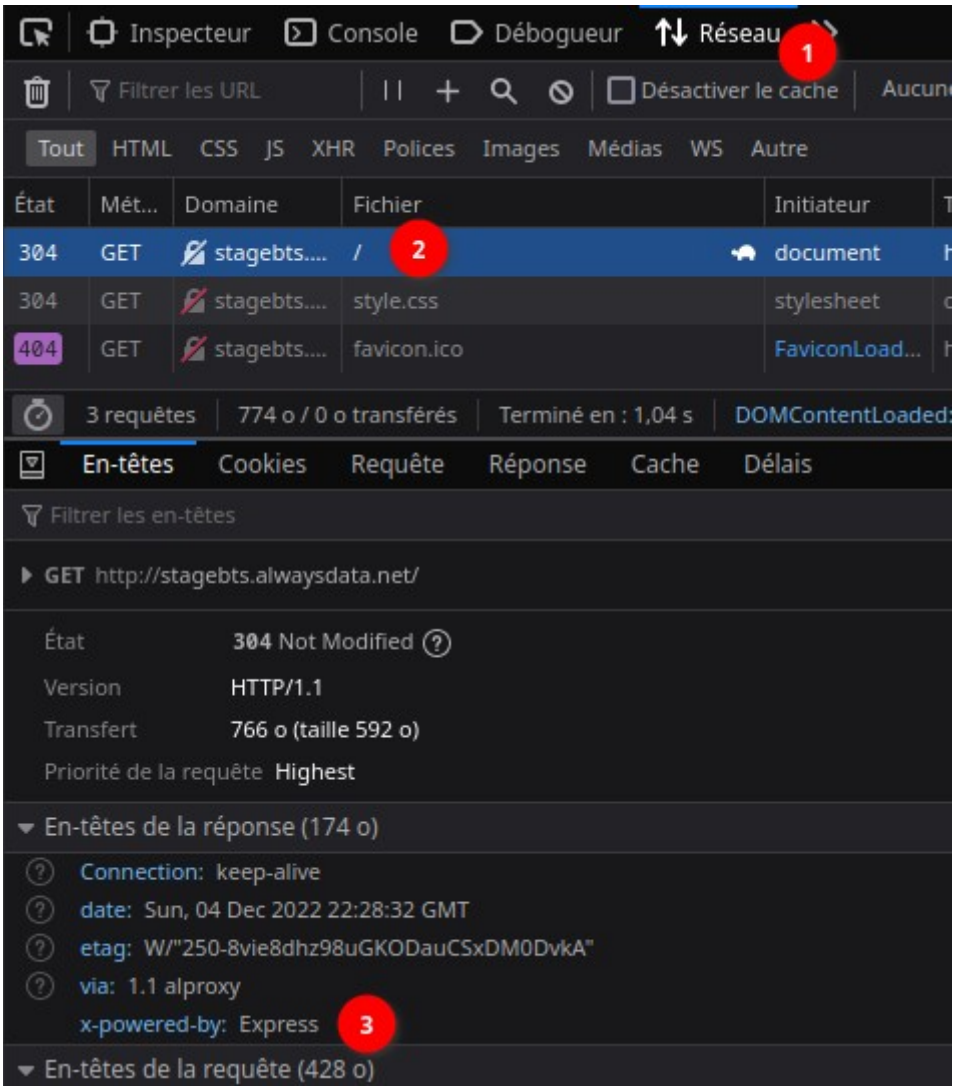
```
TOKEN_SECRET=ceci-est-un-jeton-secret-devant-avoir-32-caracteres-minimum-pour-le-chiffrement-
HSA256
```

Sauvegardez bien tous les fichiers. Relancez l'application : normalement, tout doit fonctionner correctement.

En cas de problème, vous pouvez basculer sur la branche 'Securisation' du projet GIT qui contient l'ensemble des modifications. Mais ne cédez pas à la tentation de ne pas corriger vos erreurs (ou les miennes). Appuyez-vous sur le code disponible pour comprendre d'où vient le problème. Normalement, le module 'morgan' doit fournir des informations pratiques.

C.3 Autres sécurisations

Il s'agit peut-être d'un détail, mais lorsque le serveur Express.JS est interrogé par un client, il peut renvoyer des informations le concernant :



Pour éviter cela, le plus simple est de bloquer ce comportement, avec la commande suivante, dans le fichier principal :

```
app.disable('x-powered-by');
```

Ainsi, un pirate aura moins de facilités pour déterminer les composants du site.

C.4 Gestion des rôles

Une application se doit de gérer les rôles des utilisateurs, pour autoriser ou non, l'accès à certaines pages.

Il faut donc prévoir un système de permission par vue. Ainsi, chaque route utilisera un middleware capable de déterminer si l'utilisateur est autorisé ou non.

Par exemple :

```
const express = require('express');
const app = express();

// Middleware pour vérifier les rôles de l'utilisateur
function checkRoles(roleRequired, userRoles) {
  return (req, res, next) => {
    // Vérifier si l'utilisateur a l'un des rôles requis
    if (userRoles.some(role => role === roleRequired)) {
      next();
    } else {
      // Code potentiel pour journaliser la tentative d'accès
      res.status(403).render('accessDenied');
    }
  };
}

// Exemple d'utilisateur avec une liste de rôles
const user = {
  username: 'jean',
  roles: ['infirmière', 'médecin']
};

// Route pour lire les données médicales (accessible aux médecins et infirmières)
app.get('/donnees-medicales', checkRoles('médecin', user.roles), checkRoles('infirmière',
user.roles), (req, res) => {
  // Code pour récupérer les données médicales
  res.render('viewMedicalInfo');
});

// Route pour ajouter des données médicales (accessible uniquement aux médecins)
app.post('/donnees-medicales', checkRoles('médecin', user.roles), (req, res) => {
  // Code pour ajouter des données médicales
  res.render('viewMedicalInfo');
});

// Démarrer le serveur
app.listen(3000, () => {
  console.log('Serveur démarré sur le port 3000');
});
```

Le middleware va comparer un rôle requis avec la liste des rôles de l'utilisateur et permettre l'accès à la vue ou renvoyer une erreur 403 (en ayant éventuellement enregistré la tentative d'accès).

Comme il est possible de chaîner les middlewares, on vérifiera d'abord que le token est toujours valide, puis les rôles.

```
app.get('/donnees-medicales', checkCookieJWT, checkRoles('médecin', user.roles), (req, res) =>
{
  // Code pour récupérer les données médicales
  res.render('viewMedicalInfo');
});
```

D Forge et informations sensibles

Durant le processus de développement, nous avons favorisé l'usage de forges logicielles (GIT) pour le travail en équipe, pour la capacité à historiser les changements ainsi que la facilité de travail sur des branches de codes.

Cependant, la publication du code sur une forge publique créé un risque de sécurité, car la plupart des développeurs saisissent les informations de connexions aux différents services "en dur" dans les fichiers de code.

D.1 Exemple de mauvais code

Le cas le plus fréquent, est l'accès à une base de données. En NodeJS, cela peut ressembler à ceci :

```
const mongoose = require('mongoose')
...

mongoose.connect('mongodb+srv://sioadmin:M0t!v@tionSIO@siodb.yl12m10.mongodb.net/?
retryWrites=true&w=majority', {useNewUrlParser:true, useUnifiedTopology: true})
.then(()=>console.log('connected to database Hopital')).catch(error=>console.log('error occured
while onnecting to local DB',error))
```

Un attaquant identifie immédiatement comment accéder à la base distante, car il trouve l'identifiant, le mot de passe et la base utilisée.

Un autre exemple plus lisible :

```
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'hopital'
});

connection.connect((err) => {
  if (err) throw err;
  console.log('Connexion à la base de données réussie.');
});
```

Un cadeau bénit des hackers.

D.2 Contourner la transmission par la forge

L'idée est ici de placer les données dans un fichier avec une extension précise et d'exclure ce fichier de la réplication GIT, en utilisant le fichier `.gitignore`.

D.2.1 Déclaration du fichier .env dans .gitignore

L'instruction est simple, puisqu'il suffit de créer une nouvelle ligne contenant

```
.env
```

D.2.2 Installation du module dotenv

Dans un projet NodeJS, il est facile d'installer un petit framework appelé dotenv : ce dernier permet de stocker des informations sensibles dans un fichier .env.

```
npm install dotenv --save
```

D.2.3 Création du contenu du fichier

Il suffit alors de déplacer les informations sensibles du code, dans le fichier `.env`, à la racine du projet, comme ceci :

```
DB_HOST=localhost
DB_USER=hopitaladm
DB_PASSWORD=F0urm!
DB_DATABASE=hopital
```

Vous noterez au passage, que nous avons créé un compte utilisateur pour la base hopital (voir commandes SQL) et que nous n'utilisons plus le compte **root**.

D.2.4 Utilisation du fichier .env

Il ne reste qu'à remplacer les données sensibles dans le code original :

```
import * as dotenv from 'dotenv';
dotenv.config();

const connection = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_DATABASE
});

connection.connect((err) => {
  if (err) throw err;
  console.log('Connexion à la base de données réussie.');
});
```

Les données ne seront plus accessibles sur la forge distante.

E Documentation

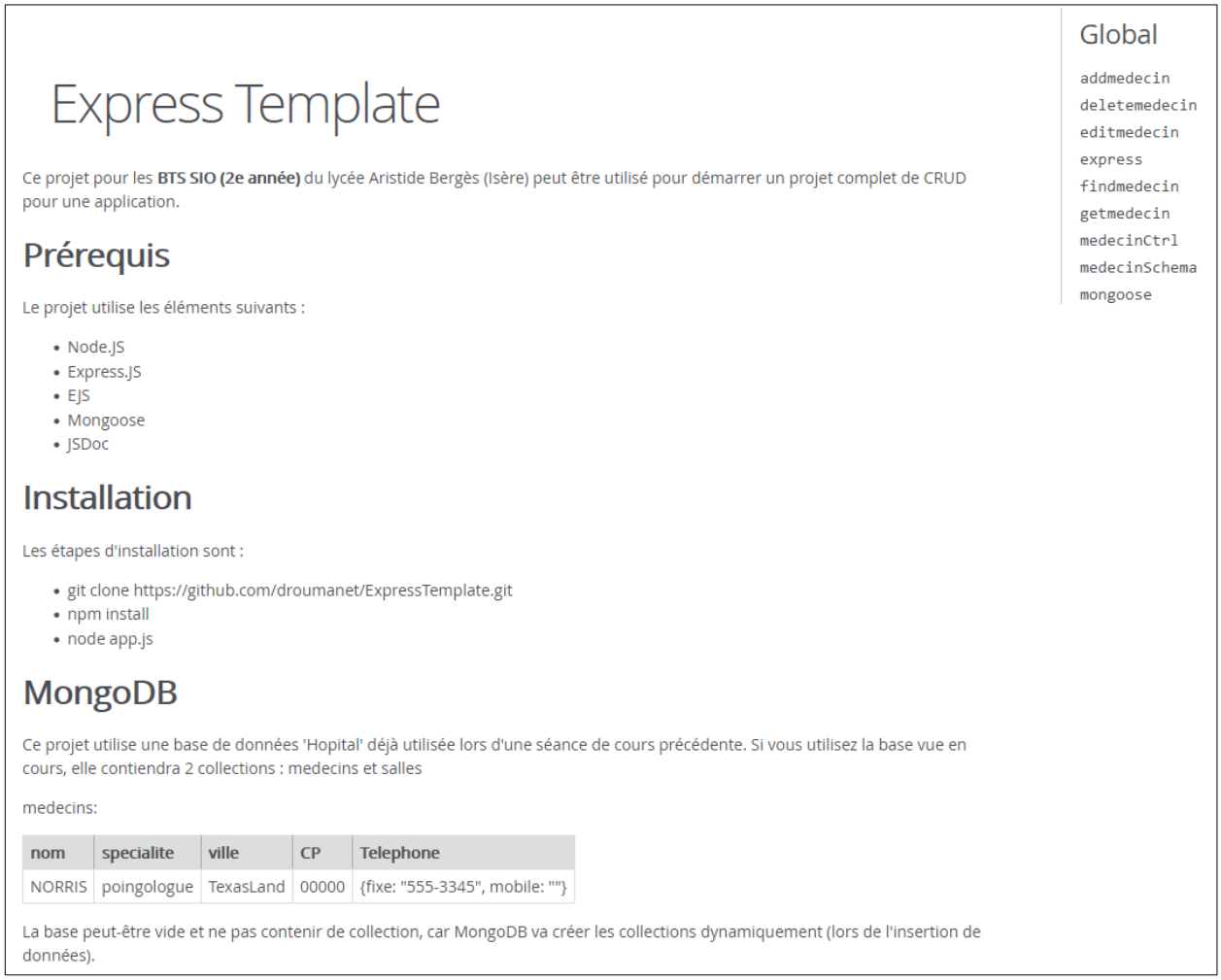
La documentation du projet est un élément de sécurisation, car il améliore la maintenabilité. De plus, de JSDoc dans le code, permet à Visual Studio Code de fournir une complétion automatique plus précise et efficace.

E.1 JSDoc

JSDoc est un standard de description de commentaires, mais aussi un exécutable capable de générer une documentation au format HTML

E.1.1 Exemple de documentation

La documentation générée comprend un sommaire et un formatage des commentaires, ainsi que la page README.md (JSDoc peut lire le Markdown).



Global

addmedecin
deletemedecin
editmedecin
express
findmedecin
getmedecin
medecinCtrl
medecinSchema
mongoose

Express Template

Ce projet pour les **BTS SIO (2e année)** du lycée Aristide Bergès (Isère) peut être utilisé pour démarrer un projet complet de CRUD pour une application.

Prérequis

Le projet utilise les éléments suivants :

- Node.JS
- Express.JS
- EJS
- Mongoose
- JSDoc

Installation

Les étapes d'installation sont :

- git clone https://github.com/droumanet/ExpressTemplate.git
- npm install
- node app.js

MongoDB

Ce projet utilise une base de données 'Hopital' déjà utilisée lors d'une séance de cours précédente. Si vous utilisez la base vue en cours, elle contiendra 2 collections : medecins et salles

medecins:

nom	specialite	ville	CP	Telephone
NORRIS	poingologue	TexasLand	00000	{fixe: "555-3345", mobile: ""}

La base peut-être vide et ne pas contenir de collection, car MongoDB va créer les collections dynamiquement (lors de l'insertion de données).

E.2 Mise en œuvre de la documentation JSDoc

Installez le module `jsdoc` dans le projet, uniquement pour le développement :

npm install --save-dev jsdoc

ensuite, créez un fichier `jsdoc.conf.json` à la racine du projet :

jsdoc.conf.json

```
{
    "tags": {
      "allowUnknownTags": true,
      "dictionaries": ["jsdoc", "closure"]
    },
    "source": {
      "include": [".\\controllers", ".\\models", ".\\routes"],
      "includePattern": ".+\\.js(doc|x)?$",
      "excludePattern": "(^|\\/|\\\\)_"
    },
    "plugins": ["plugins/markdown"],
    "templates": {},
    "opts": {
      "destination": "docs",
      "recurse": true,
      "readme": "README.md"
    }
  }
```

Commentez vos fichiers et vos fonctions (normalement, vous pouvez utiliser les commentaires JSDoc déjà présents dans la branche 'Securisation' du projet).

Exemple :

Entête de fichiers

```
/**
 * @author David ROUMANET
 * @version 1.0.0
 * @description Le contrôleur de la partie Médecins. En fonction de la route choisie, le
contrôleur exécute les actions ci-dessous.
 */
```

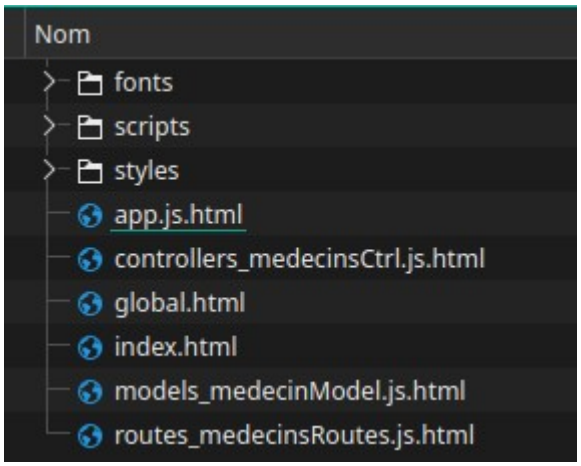
Description de fonctions (homeMedecins)

```
    /**
     * Fonction permettant d'afficher la liste des médecins
     * @param {*} req non utilisé
     * @param {*} res réponse des données lues avec getmedecin()
     */
```

Générez la documentation :

```
node node_modules/jsdoc/jsdoc.js -r -c jsdoc.conf.json app.js
```

La documentation ainsi générée au format HTML est présente dans le dossier `docs`.



Cette fonctionnalité, quoiqu'un peu longue à installer, permet à un développeur de se créer des commentaires, tout en décrivant son application et les paramètres des méthodes.

F Gestion des dépendances

La mise en œuvre des dépendances dans une application rend rapidement le code plus sensible aux vulnérabilités.

Connaître les packages employés dans son code n'est pas suffisant pour se protéger : il faut régulièrement vérifier les raisons des mises à jour et donc faire évoluer son projet, avec le risque d'utiliser des fonctions qui deviennent dépréciées.

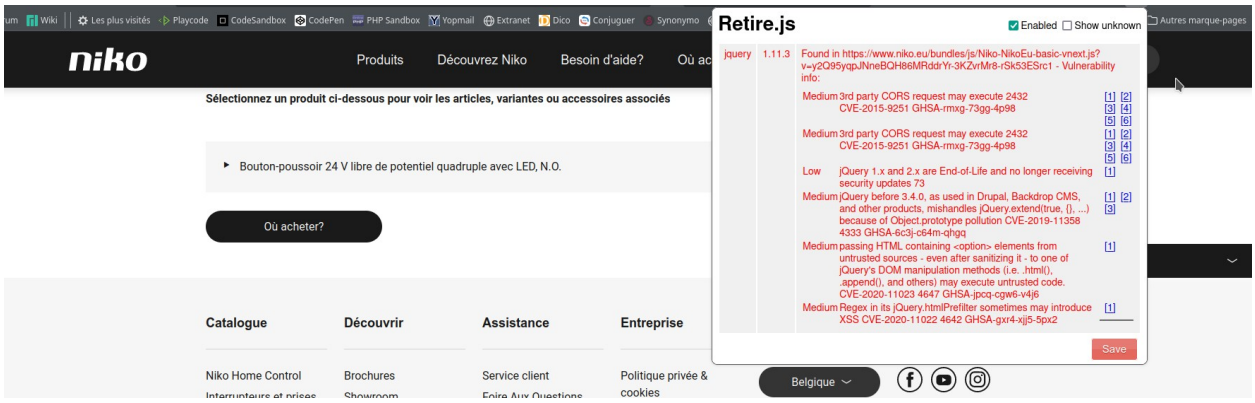
F.1 Outil de visualisation des risques

Il s'agit d'une extension pour navigateur, qui affiche les vulnérabilités connues pour les modules utilisés sur le site visité :

Le site officiel donne des informations : https://retirejs.github.io/retire.js/

Les extensions pour navigateurs sont accessibles pour Firefox et pour Chrome.

Le résultat est surprenant.



En indiquant la version du module utilisé et les vulnérabilités associées, RetireJS permet de faciliter la recherche des mises à jour nécessaire.

Attention : un module peut avoir une vulnérabilité sur une fonction qui n'est pas utilisée dans votre application. Il n'est peut-être pas nécessaire de mettre à jour le module, s'il y a des risques de casser la compatibilité de votre application, il faut toujours peser les pour et les contre.

F.2 Outil de visualisation des dépendances

Il existe un outil pratique qui – à partir d'un module NPM – retrace les sous-modules et donc les dépendances.

https://npmgraph.js.org/

Un exemple avec le module Express.JS