

# P\_Sec

---



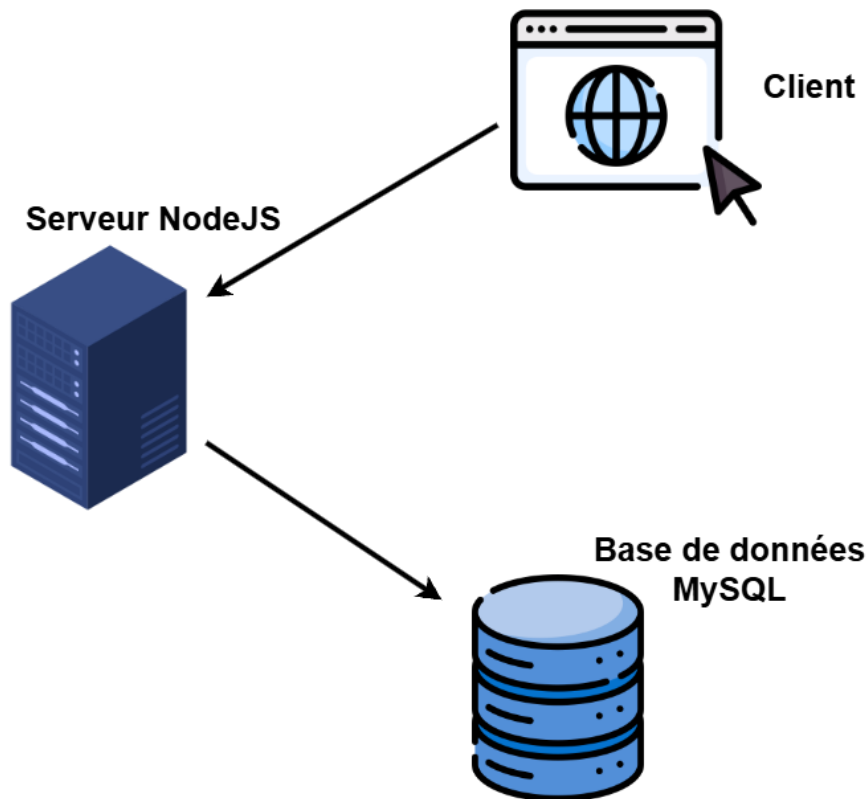
Emma Blanchoud – MID2A  
Vennes  
24 périodes  
Gaël Sonney

# Table des matières

<b>1</b>	<b>CONCEPTUALISATION .....</b>	<b>3</b>
1.1	SCHÉMA .....	3
<b>2</b>	<b>RÉALISATION .....</b>	<b>3</b>
2.1	HTTPS.....	3
2.2	AUTHENTIFICATION PAR MOT DE PASSE .....	4
2.3	VÉRIFICATION DU TOKEN JWT.....	5
2.4	ADMINISTRATION .....	6
2.5	PROTECTION CONTRE LES INJECTIONS SQL.....	6
2.6	UTILISATION DE BCrypt .....	7
2.7	UTILISATION DE L'IA .....	7
<b>3</b>	<b>CONCLUSION.....</b>	<b>7</b>
3.1	BILAN PERSONNEL .....	7

# 1 CONCEPTUALISATION

## 1.1 Schéma



# 2 RÉALISATION

## 2.1 HTTPS

Changer le port :

```
// Démarrage du serveur
app.listen(443, () => {
  console.log('Server running on port 443');
});
```

Générer une clé privée :

- openssl genpkey -algorithm RSA -out privkey.key

Créer une demande de signature de certificat (CSR) :

- openssl req -new -key privkey.key -out request.csr

Signer la demande pour produire un certificat auto-signé :

- openssl x509 -req -in request.csr -signkey privkey.key -out certificate.crt -days 365

```
const fs = require("fs");

// Charger les clés SSL
const privateKey = fs.readFileSync("keys/privkey.key", "utf8");
const certificate = fs.readFileSync("keys/certificate.crt", "utf8");
```

```
const credentials = { key: privateKey, cert: certificate };
```

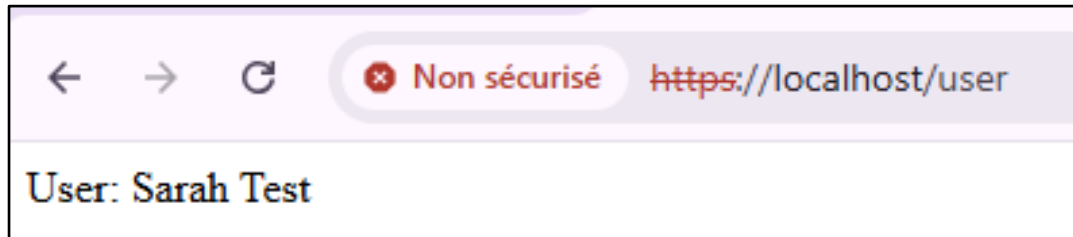
Ensuite il faut Créer le serveur HTTPS

```
const httpsServer = https.createServer(credentials, app);
```

Et ajouter

```
const fs = require("https");
```

Le certificat est auto-signé alors pour chrome il n'est pas reconnu c'est pourquoi il ne fonctionne pas.



## 2.2 Authentification par mot de passe

Ajouter ces lignes dans userController.js afin d'afficher l'html de la page login  
sendFile qui permet d'envoyer le fichier html au serveur ?

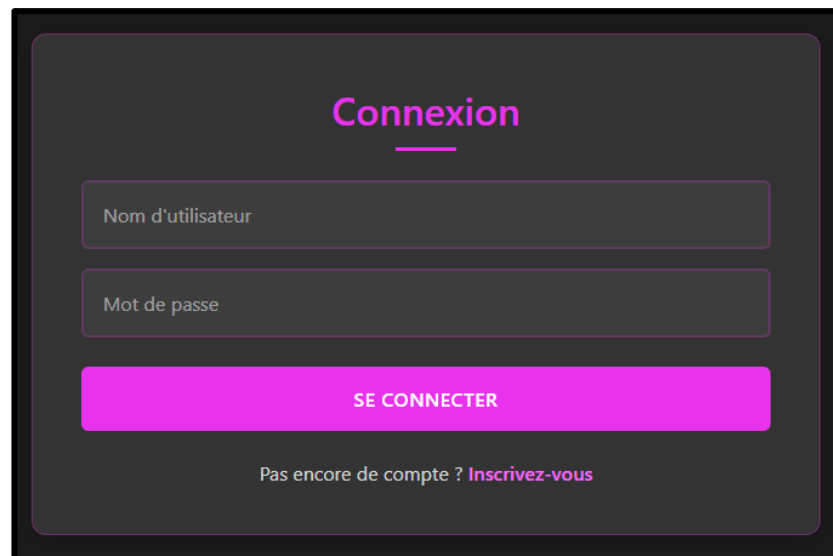
```
login: (req, res) => {  
  //Afficher l'html  
  res.sendFile("login.html", { root: "./view" });  
},
```

Créer un fichier login.html et y ajouter l'html nécessaire

A screenshot of a login form. At the top, the word 'Login' is written in a large, bold, black serif font. Below it, there are two input fields. The first field is preceded by a user icon and has the placeholder text 'Username'. The second field is preceded by a lock icon and has the placeholder text 'Password'. To the right of the 'Password' field is a button labeled 'Login'.

Figure 1 Page de connexion

Finalement y ajouter du style dans l'html proposé par chatgpt pour améliorer le visuel.



Connexion

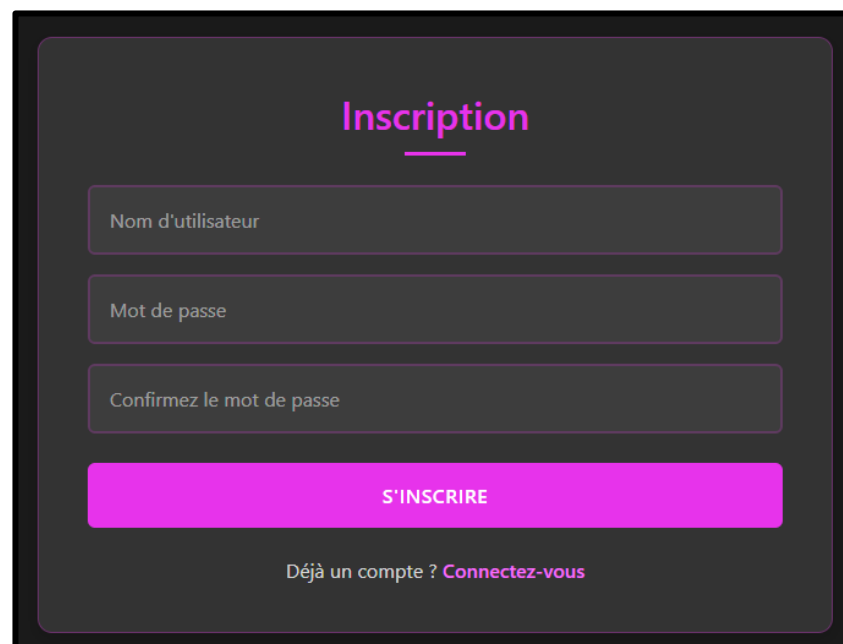
Nom d'utilisateur

Mot de passe

SE CONNECTER

Pas encore de compte ? [Inscrivez-vous](#)

Figure 2 Page de connexion avec css



Inscription

Nom d'utilisateur

Mot de passe

Confirmez le mot de passe

S'INSCRIRE

Déjà un compte ? [Connectez-vous](#)

Figure 3 Page d'inscription avec css

## 2.3 Vérification du token JWT

Dans ce projet afin de s'assurer que l'utilisateur effectuant une requête est bien authentifié et autorisé à accéder aux ressources demandées, la vérification du token JWT est importante.

La première étape consiste à vérifier la validité du token reçu. Cette opération est effectuée à l'aide de la méthode `jwt.verifyToken(token)`. Si le token est valide, il est décodé et contient les informations associées à l'utilisateur, comme son nom d'utilisateur et son identifiant. Si la vérification échoue une erreur est levée et l'authentification est rejetée.

Une fois le token validé, l'identifiant de l'utilisateur est extrait et utilisé pour interroger la base de données afin de récupérer les informations associées (ID, nom d'utilisateur et statut administrateur).

## 2.4 Administration

L'administrateur a accès à une barre de recherche qui permet d'afficher tous les utilisateurs ayant tout ou une partie du nom recherché.

Avant d'exécuter la requête, le code s'assure que le paramètre de recherche (req.query.search) est bien défini. S'il n'y en a pas, il est remplacé par une chaîne vide, ce qui permet d'éviter les erreurs lors de l'exécution de la requête SQL.

```
if (!req.query.search) {  
    req.query.search = "";  
}
```

Ensuite une fonction verifyEntry est appelée pour valider l'entrée de l'utilisateur.

```
Await verifyEntry(req.query.search, res);
```

Une fois la requête validée, une recherche est effectuée dans la table t\_users pour récupérer les utilisateurs dont le username contient la valeur recherchée. L'utilisation de LIKE avec les caractères % permet de rechercher le nom en entier ou simplement une partie.

```
Const [users] = await db.db.query(  
    "SELECT id, username FROM t_users WHERE username LIKE ?"  
    [`%${req.query.search}%`]  
);
```

## 2.5 Protection contre les injections SQL

Pour la protection contre les injections SQL il est nécessaire de mettre en place un module de validation des entrées utilisateur dans le fichier secureEntry.js. Ce module bloque les caractères et mot-clés couramment utilisés pour les attaques SQL, tel que :

- Caractères spéciaux susceptibles de perturber une requête SQL (par exemple : « ; », « ' », « -- », « /\* », « \*/ »).
- Mots-clés SQL pouvant être exploités dans une injection (par exemple : SELECT, INSERT, DELETE, DROP, UNION, JOIN).

Le fichier secureEntry.js implémente la fonction verifyEntry, qui filtre les entrées en comparant leur contenu avec une liste de caractères interdits.

Cette fonction est utilisée dans différents modules, comme register.js, afin de vérifier la validité des entrées utilisateur avant leur traitement :

```
let error = await verifyEntry(username, res);  
if (error) {  
    return;  
}  
error += await verifyEntry(password, res);  
if (error) {
```

```
return;  
}
```

## 2.6 Utilisation de bcrypt

Dans le cadre de la sécurisation des mots de passe des utilisateurs, il est essentiel d'adopter une approche robuste qui empêche toute récupération aisée des mots de passe stockés en base de données. Une des solutions les plus couramment utilisées est l'utilisation de bcrypt.

### 1. Vérification et sécurisation des mots de passe

Avant d'enregistrer des mots de passes dans une base de données, certaines validations de base doivent être mises en place :

- Le mot de passe doit contenir au moins 8 caractères
- L'utilisateur doit confirmer son mot de passe

### 2. Hashage du mot de passe avec bcrypt

L'utilisation de bcrypt permet de hasher le mot de passe afin de ne pas le stocker en clair. Bcrypt permet la génération d'un sel unique à chaque hashage.

Dans l'implémentation un sel de 10 est utilisé pour déterminer la complexité du hashage.

```
// Définition du coût du hachage  
const saltRounds = 10;  
const hashedPassword = await bcrypt.hash(password, saltRounds);
```

Cela permettra ensuite de créer un utilisateur et d'enregistrer son nom ainsi que son mot de passe, mais uniquement sous forme hachée.

Lors de la connexion, le mot de passe saisi par l'utilisateur sera également haché afin de le comparer à celui stocké en effectuant une requête récupérant les informations de l'utilisateur à partir de son nom d'utilisateur, permettant ainsi de vérifier son existence.

## 2.7 Utilisation de l'IA

Dans ce projet, j'ai eu recours à l'intelligence artificielle pour m'aider à résoudre des erreurs complexes qui me faisaient perdre trop de temps. De plus, je l'ai utilisée pour reformuler certaines parties de ce rapport afin d'en améliorer la clarté.

## 3 CONCLUSION

### 3.1 Bilan personnel

Pendant ce projet, j'ai principalement appris à sécuriser une application web avec des outils comme JWT et bcrypt. L'authentification, la gestion des tokens et la protection contre les injections SQL ont été des points clés du développement.

Si c'était à refaire, je garderais l'utilisation de ces bonnes pratiques de sécurité et l'organisation du code, qui ont facilité la compréhension et la maintenance du projet.

Merci à mes camarades comme Théo Richard, Ethan Rotzetter et Eithan Sanchez Filipe pour leur aide précieuse et leurs conseils tout au long du projet.