




Node.js

Authentication



Authentification ?

- Processus permettant à un système de s'assurer de la légitimité de la demande d'accès faite par une entité (être humain ou autre système) afin d'autoriser l'accès de cette entité à des ressources du système ([wikipédia](#))
- Facteurs d'authentification:
 - ce que l'entité connaît (exemple: nom d'utilisateur et mot de passe)
 - ce que l'entité possède (exemple: token physique ou token logiciel)
 - ce que l'entité est (exemple: les empreintes digitales)
 - ce que l'entité sait faire (exemple: signature)
- Authentification multi-facteurs (MFA): utiliser au moins deux facteurs d'authentification différents
- Nous allons mettre en place une authentification du premier type (ce que l'entité connaît) pour notre serveur




Authentification par nom d'utilisateur et mot de passe

- C'est la méthode d'authentification la plus classique et la plus utilisée
- Pour s'authentifier sur un site, l'utilisateur doit fournir son nom d'utilisateur et son mot de passe
- La sûreté de ce système d'authentification est directement dépendante de la capacité de l'utilisateur à créer un mot de passe fort
- Les échanges entre le client et le serveur doivent être chiffrés:
 - Dans le cas d'un serveur web, cela implique l'utilisation du **HTTPS**
 - Si les échanges ne sont pas chiffrés entre le client et le serveur, alors le mot de passe de l'utilisateur est potentiellement lisible par des tierces parties. Il est compromis
- Il est difficile d'implémenter proprement un système d'authentification qui ne comporte pas de failles. Il faut passer par des bibliothèques externes voir des services dédiés




À propos du chiffrement

- Le chiffrement consiste à rendre le contenu d'un message incompréhensible pour qui ne possède pas la clé de déchiffrement
- Il permet d'échanger un message entre un émetteur et un récepteur, et de garantir que:
 - seuls l'émetteur et le récepteur soient en mesure de le lire
 - le message ne soit pas altéré durant son transit
 - l'émetteur du message est bien celui qu'il annonce être
- Dans le cas du protocole HTTP utilisé par les navigateurs et les serveurs web, il s'agit du protocole HTTPS



Authentification par nom d'utilisateur et mot de passe : processus

1. On suppose que l'utilisateur possède déjà un compte
2. L'utilisateur non authentifié arrive sur une page de login sur laquelle il peut saisir son nom d'utilisateur et son mot de passe
3. Une requête HTTPS contenant ces informations est envoyée au serveur
4. Le serveur confronte le nom d'utilisateur et le mot de passe reçus dans la requête HTTPS avec le contenu de sa base de données:
 - a. Si le serveur trouve un utilisateur avec ce nom et ce mot de passe, il renvoie une réponse HTTPS contenant un identifiant de session. L'utilisateur s'est authentifié avec succès
 - b. Si le serveur ne trouve pas d'utilisateur avec ce nom et ce mot de passe, l'utilisateur n'est pas authentifié



Authentification par nom d'utilisateur et mot de passe : stockage du mot de passe

- **Le mot de passe d'un utilisateur ne doit jamais être stocké en clair dans la base de données du système**
- Même en temps que concepteur du système, je ne dois jamais être en mesure de connaître le mot de passe d'un utilisateur
- Cela permet de:
 - s'assurer que personne d'autre que l'utilisateur ne connaît son mot de passe
 - s'assurer, en cas de fuite de la base de données, que les mots de passe ne soient pas directement utilisables par les attaquants
- Comment peut-on à la fois avoir la responsabilité de vérifier le mot de passe d'un utilisateur sans pour autant être capable de connaître ce mot de passe ? Grâce aux fonctions de hachage



Les fonctions de hachage cryptographique

- Ce sont des fonctions qui, pour une donnée fournie en entrée, produisent une empreinte numérique (le *hash*)
- Elles doivent être irréversibles: il ne doit pas être possible, à partir d'une empreinte numérique donnée, d'obtenir la donnée d'origine
- Il ne doit pas exister un ensemble de données différentes pour lesquelles l'empreinte numérique est la même (*collision*)
- Elles doivent pouvoir calculer le hash d'une donnée rapidement
- Une petite variation entre deux données d'entrée doit produire deux hash complètement différents




Les fonctions de hachage cryptographique, exemple

- [Outil en ligne](#) pour calculer des hashes md5
- Exemple de résultats:
 - a => 0cc175b9c0f1b6a831c399e269772661
 - b => 92eb5ffee6ae2fec3ad71c777531578f
 - superMotDePasse => eff68ccb03c8bcf69592e80d5076926d
 - superMotDePasse2 => 7ce0ef475b7a5909a4084588eeb9f9dd
- Attention, le MD5 est obsolète pour le stockage des mots de passe



Les fonctions de hachage cryptographique

- Quelques fonctions de hachage (obsolètes pour le stockage de mot de passe):
 - md5
 - SHA-1
 - SHA-256
 - ...
- Quelques fonctions de hachage dédiées:
 - PBKDF2
 - bcrypt
 - scrypt
 - argon2
 - ...

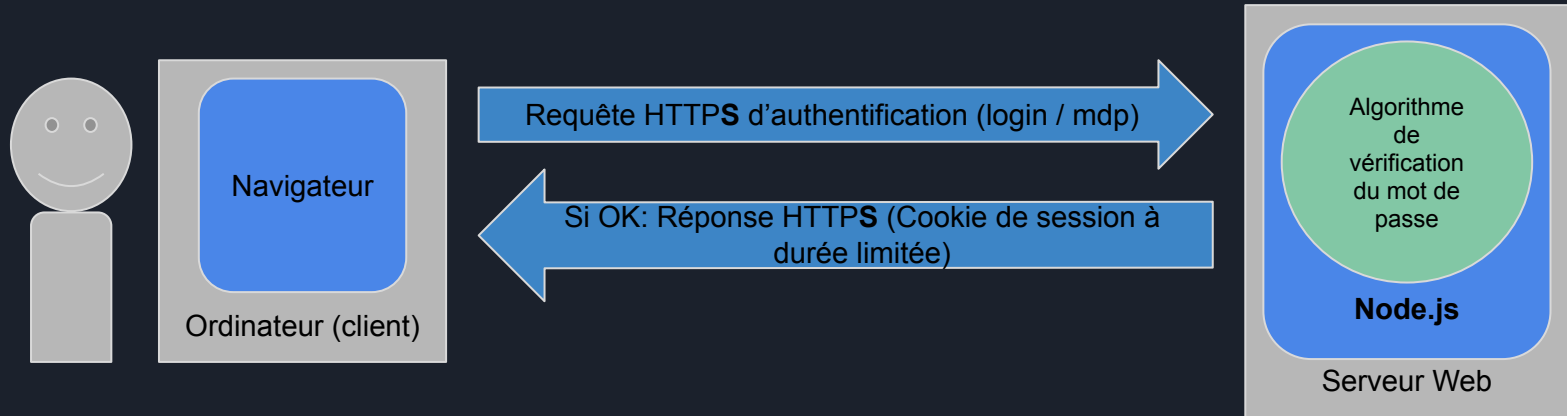


Authentification par nom d'utilisateur et mot de passe : exemples d'attaques

- Attaque par force brute:
 - Tester toutes les combinaisons de mots de passe possibles
 - a, b, c, ..., z, aa, ab, ac, ..., az, ba, bb, bc, ..., bz ...
- Attaque par dictionnaire:
 - Tester un ensemble de mots de passe prédéfinis
 - azerty, 123456789, kevin31, darkSasuke
- Ces attaques sont rendues largement inefficaces par la limitation du nombre d'essais dans un laps de temps donné pour un utilisateur (si le mot de passe de l'utilisateur est suffisamment complexe)
- Un mot de passe complexe doit:
 - être assez long (une dizaine de caractères ?)
 - comporter des majuscules, des minuscules, des chiffres, des caractères spéciaux
 - éviter les mots existants dans une langue

Authentification par nom d'utilisateur et mot de passe

- Nous allons mettre en place ce système d'authentification pour notre serveur
- Pour des raisons de simplicité, nous n'utiliserons pas le protocole HTTPS, mais il faudrait obligatoirement le faire pour un véritable site





● Projet Node.js : Authentification (0)

1. Via votre client MySQL, altérez la base de données avec la requête suivante:

```
ALTER TABLE `users`  
  MODIFY `name` varchar(128) COLLATE utf8mb4_bin UNIQUE NOT NULL,  
  ADD COLUMN `is_admin` TINYINT(1) DEFAULT 0;
```

2. Via l'interface web, créez un utilisateur avec pour nom "admin" et pour mot de passe "admin"
3. Via votre client MySQL, modifiez la valeur de la colonne "is_admin" avec la valeur 1 pour cet utilisateur



● Projet Node.js : Authentification (1)

1. Coupez votre serveur si il est lancé (Ctrl + C)
2. Installez les dépendances suivantes avec la commande `npm install --save`
 - a. passport ([Documentation](#))
 - b. passport-local
 - c. express-session ([Documentation](#))
 - d. express-mysql-session ([Documentation](#))
 - e. bcrypt ([Documentation](#))
3. Relancez le serveur



● Projet Node.js : Authentification (2)

Pour commencer, nous allons utiliser bcrypt pour ne plus stocker les mots de passe en clair

1. Dans le fichier "userController.js":
 - 1.1. Importez bcrypt: `import bcrypt from "bcryptjs";`
 - 1.2. Dans la fonction `putUser` n'utilisez plus directement le mot de passe pour l'insertion en base de donnée mais utilisez le résultat de la fonction `await bcrypt.hashSync(password, 12)`
 - 1.3. Dans la fonction `postUser` n'utilisez plus directement le mot de passe pour la modification en base de donnée mais utilisez le résultat de la fonction `await bcrypt.hashSync(newpassword, 12)`
 - 1.4. Mettez à jour le mot de passe de chaque utilisateur via l'interface de la page GET /users pour qu'ils ne soient plus stockés en clair. Faites en sorte de vous en souvenir ou de les noter quelque part



● Projet Node.js : Authentification (3)

1. Dans "userDb.js":
 - 1.1. Créez une fonction `getUserByUsername` qui prend en paramètre un nom d'utilisateur et retourne l'utilisateur ayant le nom d'utilisateur spécifié via une requête vers la base de données
 - 1.2. Exportez cette nouvelle fonction
2. Dans le dossier "src", créez un fichier "passport.js", et dans ce fichier:
 - 2.1. `import passport from "passport";`
 - 2.2. `import bcrypt from "bcryptjs";`
 - 2.3. `import Strategy from "passport-local/lib/index";`
 - 2.4. `import userDb from "../db/userDb";`
 - 2.5. Pour la suite du code de ce fichier, elle sera faite en direct sur le stream

● Projet Node.js : Authentification (4)

1. Dans "index.js", ajoutez les imports suivants:

```
import passport from "passport";  
import expressSession from "express-session";  
import MySQLStore from "express-mysql-session";  
import passportInitialization from  
"./passport";
```

2. puis le code suivant, après la ligne `const port = 3000;` (n'oubliez pas de renseigner vos valeurs pour la BDD)

```
app.use(expressSession({  
  key: "authentication_cookie",  
  secret: "ceciDoitResterSecret",  
  store: new MySQLStore({  
    host: 'localhost',  
    user: 'root',  
    password: 'root',  
    database: 'exercice_node_js'}),  
  resave: false,  
  saveUninitialized: false  
}));  
  
passportInitialization();  
  
app.use(passport.initialize());  
app.use(passport.session());
```




● Projet Node.js : Authentification (5)

1. Créez une nouvelle route GET /login qui retourne une page HTML contenant un formulaire avec:
 - 1.1. Un champ pour saisir le nom d'utilisateur
 - 1.2. Un champ pour saisir le mot de passe
 - 1.3. Un bouton pour envoyer une requête AJAX sur la route POST /login (que vous créerez au point 2) avec un corps de requête contenant:
 - 1.3.1. un attribut "username" avec la valeur du champ de nom d'utilisateur
 - 1.3.2. un attribut "password" avec la valeur du champ de mot de passe
 - 1.4. À la réception de la réponse de la requête AJAX, vous utiliserez l'instruction suivante, qui permettra de rediriger votre utilisateur après le login `window.location.assign(this.responseURL);`
2. Dans "routes.js", créez la route POST /login de la façon suivante:

```
routes.post('/login', passport.authenticate('local', {  
  successRedirect: '/users',  
  failureRedirect: '/login'  
}));
```



● Projet Node.js : Authentification (6)

Il ne reste plus qu'à définir les routes qui nécessitent un login de l'utilisateur, à savoir toutes, sauf GET et POST /login

1. Pour protéger vos routes, modifiez "routes.js":
 - 1.1. Importez passport: `import passport from "passport";`
 - 1.2. Utilisez l'instruction `passport.authenticationMiddleware()` de la façon suivante sur GET /users:
 - `routes.get('/users', passport.authenticationMiddleware(), userController.getUsers);`
 - 1.3. Protégez toutes les routes (sauf les /login) de cette manière
2. Via votre navigateur, essayez d'accéder à votre route GET /users. Comme vous n'êtes pas logués, vous devriez être renvoyé sur le formulaire de connexion que vous avez créé
3. Connectez-vous avec un utilisateur existant dans votre base de donnée. Vous devriez être redirigé vers la liste des utilisateurs
4. Supprimez le cookie de cet utilisateur de la nouvelle table "sessions", puis sur le navigateur logué avec cet utilisateur, rafraichissez la liste des utilisateurs. Vous devriez être redirigé sur le login



● Projet Node.js : Authentification (7)

1. Maintenant que vous avez un système de login propre:
 - 1.1. Côté HTML et javascript client:
 - 1.1.1. Vous n'avez plus besoin du champ de mot de passe d'administrateur
 - 1.1.2. Vous n'avez plus besoin des paramètres de mot de passe dans vos requêtes AJAX (sauf le champ "newpassword" de la requête POST `/user/:userId` et le champ "password" de la requête PUT `/user`).
 - 1.2. Côté serveur:
 - 1.2.1. Vous ne devez plus vérifier l'égalité des mots de passe (utilisateur ou admin) au niveau de vos contrôleurs, vous vous servirez du cookie d'authentification pour les contrôles d'accès par la suite



● Projet Node.js : Authentification (8)

1. Côté serveur, vous allez implémenter les règles suivantes:
 - Tous les utilisateurs connectés peuvent aller sur GET `/users`
 - Seul un admin peut créer un nouvel utilisateur (PUT `/user`)
 - Un utilisateur ne peut-être modifié que par lui-même ou par un administrateur (POST `/user/:userId`)
 - Un utilisateur ne peut lister que ses propres messages, sauf si il est admin (GET `/user/:userId/messages`)
 - Un utilisateur ne peut envoyer un message qu'en son nom (PUT `/user/:userId/message`)
 - Seul un admin peut supprimer un utilisateur (DELETE `/user/:userId`)
 - Seul un admin peut supprimer un message (DELETE `/message/:messageId`)
2. Pour implémenter ces règles, vous aurez besoin de récupérer l'identifiant de l'utilisateur authentifié effectuant la requête en utilisant: `req.session.passport.user`. À partir de là, vous pouvez:
 - Comparer cet identifiant avec le paramètre `userId` pour vérifier que l'utilisateur est bien légitime pour effectuer l'opération
 - Charger l'utilisateur depuis la base de données pour voir si il a les droits d'administration
3. Si un utilisateur n'a pas les droits pour effectuer une requête, le serveur doit renvoyer l'erreur 403.



● Projet Node.js : Authentification (9)

1. Modifiez l'interface web (les templates ejs) pour que l'utilisateur ne puisse voir que les boutons pour les actions qu'il est effectivement autorisé à faire, par exemple:
 - Un user ne peut pas voir les boutons de suppression, seul un admin peut
 - Un user ne peut voir le bouton d'édition que pour son propre profil, l'admin peut voir tous les boutons d'édition de profil
 - Seul l'admin peut voir le formulaire de création d'utilisateur
 -