Authentification des comptes utilisateurs à l'aide des sessions

Objectifs

Paramétrage des sessions et des cookies

Création de messages flash dans les actions de votre contrôleur

Ajout de sessions et de messages flash

Jusqu'au dernier TP, les messages d'erreur sont affichés dans la console et les utilisateurs de l'application n'ont aucun moyen de savoir ce qu'ils pourraient faire différemment. Vous utilisez des sessions et des cookies avec le package **connect-flash** pour transmettre ces messages à vos vues. À la fin de ce codelab vous aurez une application qui vous donnera une description visuelle du succès ou de l'échec des opérations.

Les messages flash sont des données semi-permanentes utilisées pour afficher des informations aux utilisateurs d'une application. Ces messages proviennent de votre serveur d'applications et passent vers les navigateurs de vos utilisateurs dans le cadre d'une session. Les sessions contiennent des données sur l'interaction la plus récente entre un utilisateur et l'application, telles que l'utilisateur actuellement connecté, la durée avant l'expiration d'une page ou les messages destinés à être affichés une seule fois.

Vous avez plusieurs façons d'incorporer des messages flash dans votre application. Dans ce codelab, vous utilisez le module middleware **connect-flash**. Téléchargez **users-sessAuth**, exécutez la commande "**npm install**", puis installez les modules suivants:

```
$ npm i connect-flash -S
```

```
$ npm i cookie-parser express-session -S
```

Vous avez besoin du module **express-session** pour transmettre des messages entre votre application et le client. Ces messages persistent sur le navigateur de l'utilisateur mais sont finalement stockés sur le serveur. **express-session** vous permet de stocker vos messages de plusieurs manières sur le navigateur de l'utilisateur. Les cookies sont une forme de stockage de session, vous avez donc besoin du package **cookie-parser** pour indiquer que vous souhaitez utiliser des cookies et que vous souhaitez que vos sessions puissent analyser (ou décoder) les données des cookies renvoyées au serveur par le navigateur.

Question : que sont les sessions et les cookies et en quoi diffèrent-ils ?

1. Session :

Une session est utilisée pour enregistrer momentanément des informations sur le serveur afin qu'elles puissent être utilisées sur différentes pages du site Web. C'est le temps total consacré à une activité. La session utilisateur commence lorsque l'utilisateur se connecte à une application réseau spécifique et se termine lorsque l'utilisateur se déconnecte du programme ou arrête la machine.

Les valeurs de session sont beaucoup plus sécurisées car elles sont enregistrées sous forme binaire ou cryptée et ne peuvent être décodées que sur le serveur. Lorsque l'utilisateur arrête la machine ou se déconnecte du programme, les valeurs de session sont automatiquement supprimées. Nous devons enregistrer les valeurs dans la base de données pour les conserver indéfiniment.

2. Cookie :

Un cookie est un petit fichier texte qui est enregistré sur l'ordinateur de l'utilisateur. La taille de fichier maximale pour un cookie est de 4 Ko. Il est également appelé cookie HTTP, cookie Web ou cookie Internet. Lorsqu'un utilisateur visite un site Web pour la première fois, le site envoie des paquets de données à l'ordinateur de l'utilisateur sous la forme d'un cookie. Les informations stockées dans les cookies ne sont pas sécurisées car elles sont conservées côté client dans un format texte que tout le monde peut voir. Nous pouvons activer ou désactiver les cookies en fonction de nos besoins.

Étape 1: importer les modules dans server.js

Importer ces trois modules — **connect-flash, cookie-parser, et express-session** — dans votre fichier **server.js**.

```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
const port = 3000;

// sessions & cookies
const expressSession = require("express-session"),
    cookieParser = require("cookie-parser"),
    connectFlash = require("connect-flash");

// Static Files
app.use(express.static('public'))
app.use('/img', express.static(__dirname + 'public/img'))
// Templating Engine
app.set('views', './views')
app.set('view engine', 'ejs')


app.use(bodyParser.json());
app.use(bodyParser.urlencoded())


app.listen(port,()=>{
    console.log(`Le serveur ecoute sur le port ${port}`);
});
```

Étape 2: Ajout de la messagerie flash et d'un middleware de session dans server.js

Dans cet exemple, la clé secrète est affichée en texte brut dans votre fichier de serveur d'applications pour plus de simplicité. Cependant, je ne recommande pas d'afficher votre clé secrète ici, car cela ouvre votre application à des failles de sécurité. Au lieu de cela, vous pouvez stocker votre clé secrète dans une variable d'environnement et accédez à cette variable avec **process.env.**

À chaque requête et réponse effectuée entre le serveur et le client, un en-tête HTTP est regroupé avec les données envoyées sur Internet. Cet en-tête contient de nombreuses informations utiles sur les données transférées, telles que la taille des données, le type de données et le navigateur à partir duquel les données sont envoyées.

Les cookies sont un autre élément important de l'en-tête de la requête. Les cookies sont de petits fichiers de données envoyés du serveur au navigateur de l'utilisateur, contenant des informations sur l'interaction entre l'utilisateur et l'application. Un cookie peut indiquer quel utilisateur a accédé à l'application en dernier, si l'utilisateur s'est connecté avec succès, et même quelles demandes l'utilisateur a faites, par exemple s'il a réussi à créer un compte ou fait plusieurs tentatives infructueuses.

Dans cette application, vous utilisez des cookies cryptés avec une clé de cryptage de code secret pour stocker des informations sur l'activité de chaque utilisateur sur l'application et si l'utilisateur est toujours connecté, ainsi que des messages courts à afficher dans le navigateur de l'utilisateur pour lui faire savoir le cas échéant des erreurs se sont produites sur leur requête la plus récente.

server.js

```
// sessions & cookies
const expressSession = require("express-session"),
    cookieParser = require("cookie-parser"),
    connectFlash = require("connect-flash");
const serverRouter = express.Router();

// Static Files
app.use(express.static('public'))
app.use('/img', express.static(__dirname + 'public/img'))
// Templating Engine
app.set('views', './views')
app.set('view engine', 'ejs')

app.use(bodyParser.json());
app.use(bodyParser.urlencoded())

serverRouter.use(cookieParser( secret: "secret_passcode"));
// création de 24 heures à partir de millisecondes
const unJour = 1000 * 60 * 60 * 24;
serverRouter.use(
    expressSession( options: {
        secret: "secret_passcode",
        cookie: {
            maxAge: unJour
        },
        resave: false,
        saveUninitialized: false
    })
);
serverRouter.use(connectFlash());
serverRouter.use((req, res, next) => {
    res.locals.flashMessages = req.flash();
    next();
});
app.use("/", serverRouter);
```

connect-flash: c'est quoi?

Utilisez le package **connect-flash** pour créer vos messages flash. Ce package dépend des sessions et des cookies pour transmettre des messages flash entre les requêtes. Vous dites à votre application Express.js d'utiliser l'analyseur de cookies comme middleware et d'utiliser le code secret de votre choix. **cookie-parser** utilise ce code pour chiffrer vos données dans les cookies envoyés au navigateur. Ensuite, votre application utilise des sessions en indiquant à **express-session** d'utiliser l'analyseur de cookies comme méthode de stockage et d'expirer les cookies après environ un jour. Vous devez également fournir une clé secrète pour chiffrer vos données de session. Enfin, demandez à l'application d'utiliser **connect-flash** comme middleware. Pour que les messages flash fonctionnent, vous devez les joindre à la requête faite avant de rendre une vue à l'utilisateur. Généralement, lorsqu'un utilisateur fait une requête GET pour une page, par exemple pour charger la page d'accueil, vous n'avez pas besoin d'envoyer de message flash.

Les messages Flash sont particulièrement utiles lorsque vous souhaitez informer l'utilisateur d'une requête réussie ou échouée, impliquant généralement la base de données. Sur ces requêtes, comme pour la création d'utilisateurs, vous redirigez généralement vers une autre page, en fonction du résultat. Si un utilisateur est créé, vous redirigez vers la route **/users**; sinon, vous pouvez rediriger vers **/users/add** . Un message flash n'est pas différent d'une variable locale rendue disponible pour la vue. Pour cette raison, vous devez configurer une autre configuration middleware pour qu'express traite vos messages connectFlash comme une variable locale sur la réponse.

En ajoutant cette fonction middleware, vous dites à Express de transmettre un objet local appelé ***flashMessages*** à la vue. La valeur de cet objet est égale aux messages flash que vous créez avec le module **connect-flash**. Dans ce processus, vous transférez les messages de l'objet de requête vers la réponse.

Avec ce middleware en place, vous pouvez ajouter des messages à req.flash au niveau du contrôleur et accéder aux messages dans la vue via **flashMessages**.

***express-session,* saveUninitialized, et resave**

Vous pouvez spécifier que vous ne souhaitez pas envoyer de cookie à l'utilisateur si aucun message n'est ajouté à la session en définissant **saveUninitialized** sur **false**. Vous pouvez indiquer également que vous ne souhaitez pas mettre à jour les données de session existantes sur le serveur si rien n'a changé dans la session existante.

196

Supposons que les sessions sont activées globalement (pour toutes les requêtes).

Lorsqu'un client fait une requête HTTP et que cette requête ne contient pas de cookie de session, une nouvelle session sera créée par `express-session`. La création d'une nouvelle session fait plusieurs choses :

- générer un identifiant de session unique
- stocker cet identifiant de session dans un cookie de session (afin que les demandes ultérieures faites par le client puissent être identifiées)
- créer un objet de session vide, comme `req.session`
- en fonction de la valeur de `saveUninitialized`, à la fin de la requête, l'objet de session sera stocké dans le magasin de session (qui est généralement une sorte de base de données)

Si pendant la durée de vie de la requête l'objet session n'est pas modifié alors, à la fin de la requête et lorsque `saveUninitialized` vaut **false**, l'objet session (toujours vide, car non modifié) ne sera pas stocké dans le magasin de session.

Le raisonnement derrière cela est que cela empêchera un grand nombre d'objets de session vides d'être stockés dans le magasin de session. Comme il n'y a rien d'utile à stocker, la session est "oubliée" à la fin de la requête.

Quand voulez-vous l'activer ? Lorsque vous souhaitez pouvoir identifier les visiteurs récurrents, par exemple. Vous seriez en mesure de reconnaître un tel visiteur car il envoie le cookie de session contenant l'identifiant unique.

À propos `resave` : cela peut devoir être activé pour les magasins de session qui ne prennent pas en charge la commande "touch". Cela indique au magasin de sessions qu'une session particulière est toujours active, ce qui est nécessaire car certains magasins supprimeront les sessions inactives (inutilisées) après un certain temps.

Si un pilote de magasin de session n'implémente pas la commande touch, vous devez l'activer `resave` pour que même lorsqu'une session n'a pas été modifiée lors d'une demande, elle soit toujours mise à jour dans le magasin (la marquant ainsi comme active).

Cela dépend donc entièrement du magasin de session que vous utilisez si vous devez activer cette option ou non.

Étape 3: Ajout de messages flash dans footer.ejs

Parce que vous voulez que chaque vue affiche les succès ou les échecs potentiels, ajoutez le code ci-dessous à **views/partials/footer.ejs**. Vous pouvez également ajouter des styles personnalisés dans votre dossier **public/css** afin que les messages puissent être différenciés du contenu de la vue normale.

Tout d'abord, vérifiez s'il existe des **flashMessages**. Si des messages de réussite existent, affichez les messages de réussite dans un **div**. Si des messages d'erreur existent, affichez ces messages avec une classe de style différent.

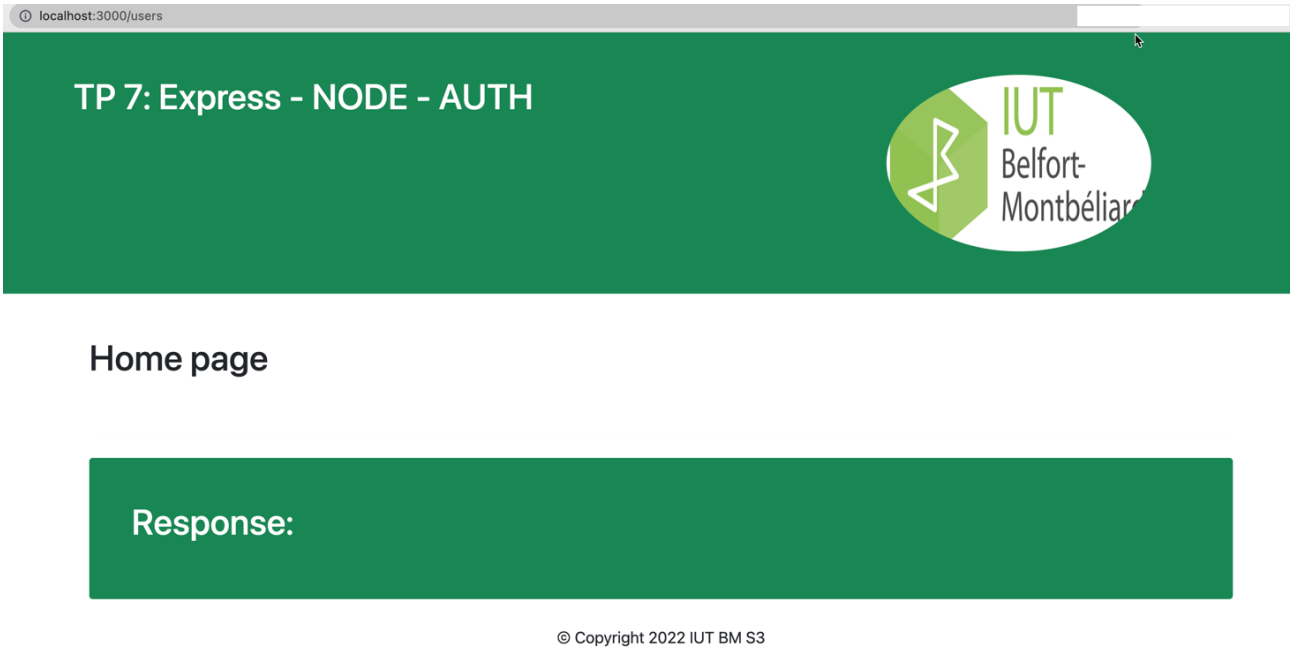
```
<div class="flashes">
    <% if (flashMessages) { %>
        <% if (flashMessages.success) { %>
            <div class="alert alert-success">
                <%= flashMessages.success %>
            </div>
        <% } else if (flashMessages.error) { %>
            <div class="alert alert-danger">
                <%= flashMessages.error %>
            </div>
        <% } %>
    <% } %>
</div>
```

Gérer la route de la page d'accueil dans le routeur et le contrôleur

```
1   const usersService = require("../services/users.service");
2
3   exports.home = (req,res)=> {
4       res.render("home");
5   }
```

users.controller.js

Vous pouvez lancer le serveur et tester la route **localhost:3000/users**.



Jusqu'à présent, il n'y a pas de validation pour la session. Ensuite, nous vérifierons s'il y a une session. Si c'est le cas, nous envoyons l'utilisateur à la page d'accueil, sinon, nous l'envoyons à la page de connexion (login).

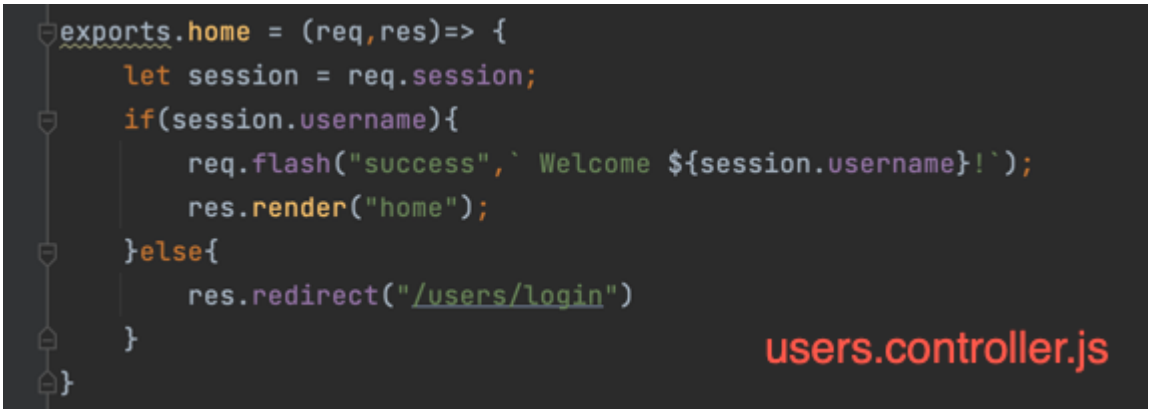
Définir une route pour la page de connexion

```
1   const usersController = require("../controllers/users.controller");
2   const express = require("express");
3   var router = express.Router();
4
5   router.get("/",usersController.home);
6   router.get("/login",usersController.login);
7
8   module.exports = router;
```

users.route.js

Modifiez la fonction du contrôleur “home” pour vérifier si une session existe

Ouvrez **users.controller.js** et modifiez la fonction **home** précédemment créée. Nous voulons maintenant vérifier la session username existe. Si c'est le cas, nous affichons la page d'accueil (**home.ejs**). Sinon, nous le redirigeons vers la page de connexion (**login.ejs**).



Authentification d'un utilisateur

Vous avez ajouté des messages flash aux actions et vues de votre contrôleur. Dans cet exemple, vous approfondissez le modèle User en validant un formulaire de login. Ensuite, vous enregistrez l'état de connexion de vos utilisateurs.

Tout d'abord, Ajoutez la route **/login** qui accepte une requête HTTP avec la méthode POST. Lorsque l'utilisateur appuie sur le bouton de type **submit** dans le formulaire HTML, les données arrivent sur cette route.

Dans le contrôleur, nous ajouterons deux méthodes : **authenticateUser** et **redirectView**.

La méthode **authenticateUser** obtiendra le nom d'utilisateur et le mot de passe à partir du corps de la requête HTTP et appellera une méthode d'authentification (**authenticate**) dans **users.service.js**. Si le service retourne que l'utilisateur existe et que le mot de passe est correct, nous ajouterons à la réponse une nouvelle information appelée "**redirect**" vers l'attribut "**locals**". Dans l'attribut "**redirect**", nous ajouterons la route vers laquelle nous voulons être redirigés. En cas de connexion réussie, nous voulons être redirigés vers **/users**. Sinon, à **/users/login**.

La méthode **authenticateUser** appellera un prochain middleware **redirectView** qui prendra le contenu de **res.locals.redirect** et appellera la fonction **res.redirect**.

Comparaison de mots de passe dans le service

Dans **users.service.js**, nous avons une méthode "**authenticate**" qui ouvrira le fichier **users.json**, obtiendra tous les utilisateurs et recherchera un utilisateur correspondant au nom d'utilisateur soumis.

users.json

```
{
  "users": [{
    "name": "admin",
    "username" : "admin",
    "passwordPlain": "admin",
    "password": "$2b$10$8NbApUVnL5l2V7QZLodU8.JO8fnX6LK6AOYM30YzMDH0YpcK4PNSC"
  }, {
    "name": "elon musk",
    "username" : "elonmusk",
    "passwordPlain": "elon123",
    "password": "$2b$10$AUysLrQJBkNn5lwLhWE4z.CYZs0JWD/KHb6c0clJzRSitUqx1kI3m"
  }],
  "note": "J'ajoute passwordPlain pour chaque utilisateur à des fins de démonstration dans le TP.
}
```

Le fichier **users.json** pourrait représenter une base de données dans ce cas. Les mots de passe sont généralement stockés dans un format crypté/haché dans la base de données et non sous forme de texte brut (plain text). Dans le fichier json, vous pouvez trouver le mot de passe en clair (utilisé uniquement pour la démonstration) et le mot de passe haché qui va être comparé au mot de passe soumis.

Qu'est-ce que le hachage de mot de passe ?

Le hachage de mot de passe consiste à transmettre un mot de passe en texte brut à un algorithme de hachage pour générer une valeur unique. Quelques exemples d'algorithmes de hachage sont bcrypt, scrypt et SHA. L'inconvénient du hachage est qu'il est prévisible.

Chaque fois que vous transmettez la même entrée à un algorithme de hachage, il générera la même sortie. Un pirate ayant accès au mot de passe haché peut désosser le cryptage pour obtenir le mot de passe d'origine. Ils peuvent utiliser des techniques telles que des attaques par force brute ou des tables arc-en-ciel. C'est là que le salage entre en jeu.

Qu'est-ce que le salage de mots de passe (password salting)?

Le salage de mot de passe ajoute une chaîne aléatoire (le sel) à un mot de passe avant de le hacher. De cette façon, le hachage généré sera toujours différent à chaque fois.

Même si un pirate obtient le mot de passe haché, il lui est impossible de découvrir le mot de passe d'origine qui l'a généré.

Générer le sel et le hachage à l'aide de bcrypt

bcrypt est un module npm qui simplifie le salage et le hachage des mots de passe.
Vous pouvez installer bcrypt en utilisant "**npm install bcrypt**". Il est déjà installé dans ce projet.

Dans ce projet, nous n'allons pas hacher et saler les mots de passe des nouveaux utilisateurs car nous considérons que la base de données (**users.json** dans notre cas) est déjà remplie avec les utilisateurs. Cependant, ci-dessous est un exemple de la façon de le faire :

```
const bcrypt = require("bcrypt")
// Le deuxième argument (10) est le sel
bcrypt.hash(plaintextPassword, 10, function(err, hash) {
  // stocke le hachage dans la base de données
  // ....
  console.log(hash);
});
```

Comparer les mots de passe à l'aide de bcrypt

Pour authentifier les utilisateurs, vous devrez comparer le mot de passe qu'ils fournissent avec celui de la base de données. **bcrypt.compare()** accepte le mot de passe en texte brut et le hachage que vous avez stocké, ainsi qu'une fonction de rappel. Ce rappel fournit un objet contenant toutes les erreurs survenues et le résultat global de la comparaison. Si le mot de passe correspond au hachage, le résultat est vrai.
Ouvrez **users.service.js** et mettez à jour la fonction "**authenticate**" pour ajouter la comparaison des mots de passe comme suit :

Ajouter une route de déconnexion

Ajoutez la route **/users/logout** dans **users.route.js:**

```
router.get("/logout",usersController.logout);
```

Lorsque nous accédons à l'URL **/users/logout**, nous voulons détruire la session. Ajoutez l'action de déconnexion à **users.controller.js**:

```
exports.logout = (req,res)⇒{
    req.session.destroy();
    res.redirect("/users/login");
}
```

L'action de déconnexion détruira la session et redirigera l'utilisateur vers la page de login.

L'idée ici est que s'il y a une session active avec l'attribut **username**, il n'est pas nécessaire de saisir à nouveau le username et le mot de passe. Dans ce cas, nous redirigeons l'utilisateur directement vers la page d'accueil **/users**.

Lorsque l'utilisateur se connecte pour la première fois, nous initialisons une session avec l'attribut **username** et le redirigeons vers la page d'accueil **/users**. La prochaine fois qu'il accédera à **/users/login**, il sera automatiquement redirigé vers la page d'accueil car la session est toujours active. Pour détruire la session, l'utilisateur doit se déconnecter en accédant à **/users/logout** dans son navigateur (généralement on lui crée un bouton ou un hyperlien dans un menu) ou en attendant que la durée de vie de la session expire.

Ouvrez la page de connexion dans votre navigateur (**/users/login**) et testez l'application. Essayez d'abord en ajoutant un utilisateur qui n'existe pas. Essayez ensuite avec un utilisateur qui existe (admin ou elonmusk) mais avec un mot de passe erroné. Essayez ensuite avec les valeurs correctes :

utilisateur 1 :

username : admin

password : admin

utilisateur 2 :

username: elonmusk

password : elon123

Lorsque la connexion est réussie et que vous êtes redirigé vers la page d'accueil, essayez la fonctionnalité de déconnexion en accédant à la route **/users/logout** dans votre navigateur.