TP - Système d'Authentification avec Sessions et Cookies

Comprendre et Implémenter l'Authentification Web

IUT Nord Franche-Comté - BUT2 S3 INFO

Objectif: Créer un système d'authentification complet avec sessions, cookies et SQLite

---

📋 Contexte

Vous êtes développeur dans une startup et devez créer un système d'authentification pour une application web. Le système doit permettre aux utilisateurs de :

- S'inscrire avec un email et mot de passe
- Se connecter et rester connecté grâce aux sessions
- Accéder à une page d'accueil protégée
- Se déconnecter proprement

Technologies utilisées:

- ✅ Node.js + Express
- ✅ Sessions avec `express-session`
- ✅ Cookies avec `cookie-parser`
- ✅ Base de données SQLite
- ✅ Hachage de mots de passe avec bcrypt
- ✅ EJS pour les vues

🎯 Objectifs d'Apprentissage

À la fin de ce TP, vous saurez:

- ✅ Comprendre la différence entre cookies et sessions
- ✅ Configurer `express-session` dans Express
- ✅ Créer et gérer des sessions utilisateur
- ✅ Protéger des routes avec un middleware d'authentification
- ✅ Hasher et vérifier des mots de passe avec bcrypt

- ✅ Utiliser SQLite pour stocker des utilisateurs
- ✅ Implémenter une architecture MVC propre
- ✅ Créer des vues avec EJS

📦 Prérequis

- ✅ Node.js installé (v18+)
- ✅ Connaissances de base en JavaScript
- ✅ Connaissances de base en Express
- ✅ Avoir suivi le cours sur les Cookies et Sessions

Partie 1 : Installation et Configuration

Exercice 1.1 : Setup Initial du Projet

📝 **Task 1:** Créez le projet et installez les dépendances

Bash

```
# Créer le dossier du projet
mkdir auth-system
cd auth-system

# Initialiser npm
npm init -y

# Installer les dépendances
npm install express express-session ejs bcrypt better-sqlite3 cookie-parser

# Installer nodemon pour le développement
npm install --save-dev nodemon
```

💡 **Explication des packages:**

- `express` : Framework web
- `express-session` : Gestion des sessions
- `ejs` : Moteur de templates pour les vues
- `bcrypt` : Hachage sécurisé des mots de passe
- `better-sqlite3` : Base de données SQLite (plus simple que PostgreSQL pour ce TP)

- `cookie-parser` : Parse les cookies (optionnel mais utile)

📝 **Task 2:** Modifiez `package.json` pour ajouter les scripts

JSON

```
{
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js"
  }
}
```

Exercice 1.2 : Structure du Projet

📝 **Task 3:** Créez la structure de dossiers suivante

Bash

```
mkdir models controllers services routes views public
mkdir public/css
```

Structure finale:

Plain Text

```
auth-system/
├── app.js                  # Point d'entrée
├── package.json
├── models/
│   ├── database.js         # Configuration BD
│   └── userModel.js        # Modèle User
├── services/
│   └── authService.js      # Logique métier
├── controllers/
│   └── authController.js   # Contrôleurs
├── routes/
│   └── authRoutes.js       # Routes
├── views/
│   ├── register.ejs        # Page inscription
│   ├── login.ejs           # Page connexion
│   └── home.ejs            # Page d'accueil
└── public/
```

```
    └── css/
        └── style.css       # Styles
```

💡 **Architecture MVC:**

- **Models** : Gèrent les données (base de données)
- **Controllers** : Traitent les requêtes HTTP
- **Services** : Contiennent la logique métier
- **Routes** : Définissent les endpoints
- **Views** : Templates EJS (interface utilisateur)

Partie 2 : Base de Données SQLite

Exercice 2.1 : Configuration de la Base de Données

📝 **Task 4:** Créez `models/database.js`

Instructions: Ce fichier doit :

1. Importer `better-sqlite3`
2. Créer/ouvrir une base de données SQLite nommée `database.sqlite`
3. Activer les foreign keys (important pour SQLite)
4. Créer une fonction `initDatabase()` qui crée les tables

Code de départ:

JavaScript

```
const Database = require('better-sqlite3');
const path = require('path');

// Créer ou ouvrir la base de données SQLite
const db = new Database(path.join(__dirname, '../database.sqlite'), {
  verbose: console.log // Affiche les requêtes SQL
});

// À COMPLÉTER: Activer les foreign keys
db.pragma(/* ... */);

function initDatabase() {
  // À COMPLÉTER: Créer la table utilisateurs
  db.exec(`
    CREATE TABLE IF NOT EXISTS utilisateurs (
```

```
      -- À COMPLÉTER
      -- id: INTEGER PRIMARY KEY AUTOINCREMENT
      -- email: TEXT UNIQUE NOT NULL
      -- password_hash: TEXT NOT NULL
      -- nom: TEXT
      -- prenom: TEXT
      -- date_creation: DATETIME DEFAULT CURRENT_TIMESTAMP
    )
  `);

  console.log('✅ Base de données initialisée');
}

module.exports = { db, initDatabase };
```

💡 **Indice:**

- Pour activer les foreign keys: `db.pragma('foreign_keys = ON');`
- SQLite utilise `INTEGER PRIMARY KEY AUTOINCREMENT` pour l'auto-incrémentation
- `TEXT` pour les chaînes, `DATETIME` pour les dates

Exercice 2.2 : Modèle User (CHALLENGE)

📝 **Task 5 (CHALLENGE):** Créez `models/userModel.js`

Mission: Créez une classe `UserModel` avec les méthodes suivantes :

1. `create(email, passwordHash, nom, prenom)` - Créer un utilisateur
2. `findByEmail(email)` - Trouver par email
3. `findById(id)` - Trouver par ID
4. `emailExists(email)` - Vérifier si email existe

Code de départ:

JavaScript

```
const { db } = require('./database');

class UserModel {
  /**
   * Créer un nouvel utilisateur
   */
  static create(email, passwordHash, nom, prenom) {
    // À COMPLÉTER
    // 1. Préparer la requête INSERT avec db.prepare()
    // 2. Exécuter avec .run()
```

```
    // 3. Retourner l'objet utilisateur créé avec result.lastInsertRowid
  }

  /**
   * Trouver un utilisateur par email
   */
  static findByEmail(email) {
    // À COMPLÉTER
    // Utilisez db.prepare() et .get()
    // Retourne NULL si pas trouvé
  }

  /**
   * Vérifier si un email existe
   */
  static emailExists(email) {
    // À COMPLÉTER
    // SELECT COUNT(*) as count FROM utilisateurs WHERE email = ?
    // Retourner true si count > 0
  }
}

module.exports = UserModel;
```

💡 **Aide - Syntaxe SQLite avec better-sqlite3:**

JavaScript

```
// Préparer une requête
const stmt = db.prepare('SELECT * FROM utilisateurs WHERE email = ?');

// Récupérer UN résultat
const user = stmt.get(email); // Retourne null si pas trouvé

// Récupérer TOUS les résultats
const users = stmt.all();

// Insérer des données
const insert = db.prepare('INSERT INTO utilisateurs (email, nom) VALUES (?,
?)');
const result = insert.run(email, nom);
console.log(result.lastInsertRowid); // ID de la ligne insérée
```

Partie 3 : Service d'Authentification

Exercice 3.1 : Comprendre bcrypt

Question 1: Pourquoi ne JAMAIS stocker les mots de passe en clair ?

Dangers:

- 🚨 Si la base de données est compromise, tous les mots de passe sont exposés
- 🚨 Les utilisateurs réutilisent souvent le même mot de passe partout
- 🚨 Les employés ont accès aux mots de passe
- 🚨 Violation du RGPD et lois sur la protection des données

📝 **Task 6:** Testez bcrypt dans la console Node

JavaScript

```
// Ouvrez node dans le terminal
node

// Dans la console Node:
const bcrypt = require('bcrypt');

// Hasher un mot de passe
const password = 'monMotDePasse123';
const hash = await bcrypt.hash(password, 10);
console.log('Hash:', hash);

// Vérifier un mot de passe
const isValid = await bcrypt.compare('monMotDePasse123', hash);
console.log('Valide?', isValid); // true

const isInvalid = await bcrypt.compare('mauvaisMotDePasse', hash);
console.log('Invalide?', isInvalid); // false
```

💡 **Points clés:**

- `bcrypt.hash(password, saltRounds)` : Hashe le mot de passe
- `bcrypt.compare(password, hash)` : Vérifie si le mot de passe correspond au hash
- Le hash est **différent à chaque fois** (grâce au salt)
- `saltRounds = 10` : Bon équilibre sécurité/performance

Exercice 3.2 : Service d'Authentification

📝 **Task 7:** Créez `services/authService.js`

Mission: Créez une classe `AuthService` avec :

1. Méthode `register(email, password, nom, prenom)` - Inscription
2. Méthode `login(email, password)` - Connexion

Code à compléter:

JavaScript

```
const bcrypt = require('bcrypt');
const UserModel = require('../models/userModel');

class AuthService {
  /**
   * Enregistrer un nouvel utilisateur
   */
  static async register(email, password, nom, prenom) {
    // ÉTAPE 1: Vérifier si l'email existe déjà
    if (UserModel.emailExists(email)) {
      throw new Error('EMAIL_EXISTS');
    }

    // ÉTAPE 2: Hasher le mot de passe
    // À COMPLÉTER: Utilisez bcrypt.hash() avec 10 rounds
    const passwordHash = /* ... */;

    // ÉTAPE 3: Créer l'utilisateur
    const user = UserModel.create(email, passwordHash, nom, prenom);

    return {
      success: true,
      message: 'Utilisateur créé avec succès',
      user: {
        id: user.id,
        email: user.email,
        nom: user.nom,
        prenom: user.prenom
      }
    };
  }

  /**
   * Authentifier un utilisateur
   */
  static async login(email, password) {
    // ÉTAPE 1: Trouver l'utilisateur
    const user = UserModel.findByEmail(email);
```

```
    if (!user) {
      throw new Error('INVALID_CREDENTIALS');
    }

    // ÉTAPE 2: Vérifier le mot de passe
    // À COMPLÉTER: Utilisez bcrypt.compare()
    const isPasswordValid = /* ... */;

    if (!isPasswordValid) {
      throw new Error('INVALID_CREDENTIALS');
    }

    // ÉTAPE 3: Retourner les infos utilisateur
    return {
      success: true,
      user: {
        id: user.id,
        email: user.email,
        nom: user.nom,
        prenom: user.prenom
      }
    };
  }
}

module.exports = AuthService;
```

💡 **Aide:**

- `await bcrypt.hash(password, 10)` : Hashe le mot de passe
- `await bcrypt.compare(password, user.password_hash)` : Vérifie le mot de passe

Partie 4 : Contrôleurs et Routes

Exercice 4.1 : Comprendre req.session

Question importante: D'où vient `req.session` ? Est-ce que c'est automatique ?

💡 Réponse

Non, ce n'est PAS automatique !

`req.session` est créé par le middleware `express-session` :

JavaScript

```
app.use(session({
  secret: 'ma-clé-secrète',
  resave: false,
  saveUninitialized: false
}));

// Maintenant req.session est disponible !
```

Ce qui est fourni par express-session (PAR DÉFAUT):

- `req.session.id` - ID de session unique
- `req.session.cookie` - Informations du cookie
- `req.session.destroy()` - Détruire la session
- `req.session.regenerate()` - Régénérer l'ID
- `req.session.save()` - Sauvegarder manuellement

Ce que VOUS ajoutez manuellement:

- `req.session.user` ✍️ VOUS
- `req.session.isAuthenticated` ✍️ VOUS
- `req.session.cart` ✍️ VOUS
- N'importe quelle propriété !

Exemple:

JavaScript

```
// Login
req.session.user = {
  id: 1,
  email: 'alice@example.com'
};

// Plus tard, dans une autre route
console.log(req.session.user.email); // alice@example.com
```

Exercice 4.2 : Contrôleur d'Authentification

📝 **Task 8:** Créez `controllers/authController.js`

Code fourni (LISEZ-LE ATTENTIVEMENT):

JavaScript

```
const AuthService = require('../services/authService');

class AuthController {
  /**
   * Afficher la page d'inscription
   */
  static showRegisterPage(req, res) {
    res.render('register', {
      error: null,
      success: null
    });
  }

  /**
   * Traiter l'inscription
   */
  static async handleRegister(req, res) {
    const { email, password, confirmPassword, nom, prenom } = req.body;

    // Validation
    if (!email || !password || !confirmPassword) {
      return res.render('register', {
        error: 'Tous les champs obligatoires doivent être remplis',
        success: null
      });
    }

    if (password !== confirmPassword) {
      return res.render('register', {
        error: 'Les mots de passe ne correspondent pas',
        success: null
      });
    }

    try {
      await AuthService.register(email, password, nom, prenom);

      res.render('register', {
        error: null,
        success: 'Compte créé avec succès ! Vous pouvez maintenant vous
connecter.'
      });
    } catch (error) {
      if (error.message === 'EMAIL_EXISTS') {
        return res.render('register', {
          error: 'Cet email est déjà utilisé',
          success: null
```

```
        });
      }

      console.error('Erreur inscription:', error);
      res.render('register', {
        error: 'Erreur lors de la création du compte',
        success: null
      });
    }
  }

  /**
   * Traiter la connexion (À COMPLÉTER)
   */
  static async handleLogin(req, res) {
    const { email, password } = req.body;

    // Validation
    if (!email || !password) {
      return res.render('login', {
        error: 'Email et mot de passe requis'
      });
    }

    try {
      const result = await AuthService.login(email, password);

      // À COMPLÉTER: Créer la session utilisateur
      // ÉTAPE 1: Stocker les infos utilisateur dans req.session.user
      req.session.user = {
        // À COMPLÉTER
      };

      // ÉTAPE 2: Rediriger vers /home
      res.redirect(/* ... */);
    } catch (error) {
      if (error.message === 'INVALID_CREDENTIALS') {
        return res.render('login', {
          error: 'Email ou mot de passe incorrect'
        });
      }

      console.error('Erreur connexion:', error);
      res.render('login', {
        error: 'Erreur lors de la connexion'
      });
    }
  }
```

```
  /**
   * Déconnexion (À COMPLÉTER)
   */
  static handleLogout(req, res) {
    // À COMPLÉTER: Détruire la session avec req.session.destroy()
    req.session.destroy((err) => {
      if (err) {
        console.error('Erreur déconnexion:', err);
        return res.redirect('/home');
      }

      // À COMPLÉTER: Rediriger vers /login
      res.redirect(/* ... */);
    });
  }
}

module.exports = AuthController;
```

Questions de compréhension:

1. Pourquoi utilise-t-on `res.render()` au lieu de `res.json()` ?
2. Que fait `req.body` et d'où vient-il ?
3. Pourquoi utilise-t-on `async/await` ?

Exercice 4.3 : Routes et Middleware d'Authentification

📝 **Task 9:** Créez `routes/authRoutes.js`

Mission: Définir les routes et créer un middleware `requireAuth`

Code à compléter:

JavaScript

```
const express = require('express');
const router = express.Router();
const AuthController = require('../controllers/authController');

// À COMPLÉTER: Middleware pour vérifier l'authentification
function requireAuth(req, res, next) {
  // ÉTAPE 1: Vérifier si req.session.user existe
  if (!req.session.user) {
    // ÉTAPE 2: Si pas connecté, rediriger vers /login
    return res.redirect(/* ... */);
  }
```

```
  // ÉTAPE 3: Si connecté, passer à la route suivante
  next();
}

// Routes publiques (pas besoin d'être connecté)
router.get('/register', AuthController.showRegisterPage);
router.post('/register', AuthController.handleRegister);

router.get('/login', AuthController.showLoginPage);
router.post('/login', AuthController.handleLogin);

// Routes protégées (nécessitent d'être connecté)
router.get('/home', requireAuth, AuthController.showHomePage);
router.get('/logout', requireAuth, AuthController.handleLogout);

module.exports = router;
```

💡 **Comprendre le middleware:**

JavaScript

```
// Sans middleware
router.get('/home', (req, res) => {
  // Route accessible à tous
});

// Avec middleware
router.get('/home', requireAuth, (req, res) => {
  // Route accessible seulement si requireAuth() appelle next()
});

// requireAuth s'exécute EN PREMIER
function requireAuth(req, res, next) {
  if (/* condition */) {
    next(); // Continue vers la route
  } else {
    res.redirect('/login'); // Stoppe ici
  }
}
```

Partie 5 : Vues EJS

Les vues sont **déjà fournies** pour gagner du temps. Vous n'avez qu'à les copier !

Exercice 5.1 : Comprendre EJS

Question: Quelle est la différence entre `<%= %>` et `<% %>` en EJS ?

💡 Réponse

- `<%= variable %>` : **Affiche** la variable (échappe le HTML)
- `<% code %>` : **Exécute** du code JavaScript sans afficher
- `<%- html %>` : Affiche du HTML sans échapper (dangereux)

Exemples:

Plain Text

```
<h1>Bonjour <%= user.nom %></h1>
<!-- Affiche: Bonjour Alice -->

<% if (user.isAdmin) { %>
  <p>Vous êtes admin</p>
<% } %>
<!-- Affiche la balise p seulement si admin -->

<% users.forEach(user => { %>
  <li><%= user.nom %></li>
<% }); %>
<!-- Boucle sur les utilisateurs -->
```

📝 **Task 10:** Les vues EJS sont déjà créées dans le code fourni. Lisez-les pour comprendre :

- `views/register.ejs` : Formulaire d'inscription
- `views/login.ejs` : Formulaire de connexion
- `views/home.ejs` : Page d'accueil protégée

Points à observer:

- Comment les erreurs sont affichées : `<% if (error) { %>`
- Comment les variables sont passées depuis le contrôleur
- L'utilisation des formulaires HTML avec `method="POST"`

Partie 6 : Point d'Entrée de l'Application

Exercice 6.1 : Fichier app.js (IMPORTANT!)

📝 **Task 11:** Créez `app.js` - Le cœur de l'application

Code à compléter:

JavaScript

```
const express = require('express');
const session = require('express-session');
const cookieParser = require('cookie-parser');
const path = require('path');
const { initDatabase } = require('./models/database');
const authRoutes = require('./routes/authRoutes');

const app = express();
const PORT = 3000;

// ÉTAPE 1: Initialiser la base de données
initDatabase();

// ÉTAPE 2: Configuration du moteur de templates
app.set('view engine', 'ejs');
app.set('views', path.join(__dirname, 'views'));

// ÉTAPE 3: Middleware pour parser les formulaires
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// ÉTAPE 4: Middleware pour les fichiers statiques (CSS)
app.use(express.static(path.join(__dirname, 'public')));

// ÉTAPE 5: Middleware pour parser les cookies
app.use(cookieParser());

// ÉTAPE 6: Configuration des SESSIONS (TRÈS IMPORTANT!)
app.use(session({
  secret: 'votre-cle-secrete-super-securisee', // À CHANGER en production!
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: false,    // À COMPLÉTER: Mettre à true en production avec HTTPS
    httpOnly: true,   // À COMPLÉTER: Protection XSS
    maxAge: /* À COMPLÉTER: 24 heures en millisecondes */
  }
}  ));

// ÉTAPE 7: Routes
app.use('/', authRoutes);

// Route par défaut
app.get('/', (req, res) => {
```

```
  if (req.session.user) {
    res.redirect('/home');
  } else {
    res.redirect('/login');
  }
});

// ÉTAPE 8: Démarrage du serveur
app.listen(PORT, () => {
  console.log(`🚀 Serveur démarré sur http://localhost:${PORT}`  );
});
```

À COMPLÉTER:

1. `maxAge` du cookie : 24 heures en millisecondes

- 💡 Indice: 1000ms × 60s × 60min × 24h = ?

1. Qu'est-ce que `resave: false` signifie ?
2. Qu'est-ce que `saveUninitialized: false` signifie ?

Partie 7 : Tester l'Application

Exercice 7.1 : Premier Lancement

📝 **Task 12:** Démarrez l'application

Bash

```
# Installer les dépendances si pas déjà fait
npm install

# Démarrer le serveur
npm start

# Ou en mode dev avec auto-reload
npm run dev
```

✅ **Vérifiez:**

- Le serveur démarre sur http://localhost:3000
- La base de données `database.sqlite` est créée
- Pas d'erreurs dans la console

Exercice 7.2 : Test Complet du Système

📝 **Task 13 (CHALLENGE):** Testez toutes les fonctionnalités

Scénario de test:

1. **Inscription**

- Aller sur http://localhost:3000/register
    - Créer un compte avec :
        - Email: `test@example.com`
        - Mot de passe: `password123`
        - Nom: `Dupont`
        - Prénom: `Jean`
- ✅ Vérifier le message de succès

1. **Tentative d'inscription avec même email**

- Essayer de recréer un compte avec le même email
- ✅ Vérifier le message d'erreur "Email déjà utilisé"

1. **Connexion avec mauvais mot de passe**

- Aller sur http://localhost:3000/login
- Email: `test@example.com`
- Mot de passe: `mauvais`
- ✅ Vérifier le message d'erreur

1. **Connexion réussie**

- Email: `test@example.com`
- Mot de passe: `password123`
- ✅ Vérifier la redirection vers /home
- ✅ Vérifier que vos informations s'affichent

1. **Protection de route**

- Se déconnecter
- Essayer d'accéder à http://localhost:3000/home directement
- ✅ Vérifier la redirection vers /login

1. **Persistance de session**

- Se connecter

- Rafraîchir la page /home (F5)
- ✅ Vérifier que vous restez connecté

1. **Cookies dans le navigateur**

- Ouvrir les DevTools (F12)
- Aller dans Application > Cookies
- ✅ Vérifier la présence du cookie `connect.sid`

1. **Déconnexion**

- Cliquer sur "Déconnexion"
- ✅ Vérifier la redirection vers /login
- ✅ Vérifier que le cookie est supprimé

Exercice 7.3 : Inspecter les Sessions

📝 **Task 14:** Comprendre ce qui se passe côté serveur

Ajoutez ce middleware temporaire dans `app.js` (AVANT les routes):

JavaScript

```
// Middleware de debug (à retirer en production)
app.use((req, res, next) => {
  console.log('='.repeat(50));
  console.log('📍 Route:', req.method, req.path);
  console.log('🍪 Session ID:', req.sessionID);
  console.log('👤 Utilisateur:', req.session.user || 'Non connecté');
  console.log('='.repeat(50));
  next();
});
```

Observez la console lors de:

1. Première visite (pas de session)
2. Après connexion (session créée)
3. Rafraîchissement de page (même session ID)
4. Après déconnexion (session détruite)

Partie 8 : Sécurité et Bonnes Pratiques

Exercice 8.1 : Améliorations de Sécurité (CHALLENGES)

📝 **Task 15:** Ajoutez une validation de mot de passe forte

Modifiez `authController.js` pour refuser les mots de passe :

- Moins de 8 caractères
- Sans majuscule
- Sans chiffre

📝 **Task 16:** Limitez les tentatives de connexion

Ajoutez un système qui bloque après 5 tentatives échouées :

💡 Indice

1. Ajoutez `tentatives_echec` dans la table `utilisateurs`
2. Incrémentez à chaque échec
3. Réinitialisez à 0 lors d'un succès
4. Bloquez si > 5

📝 **Task 17:** Variables d'environnement

Créez un fichier `.env` pour stocker le secret de session :

Bash

```
npm install dotenv
```

Fichier `.env` :

Plain Text

```
SESSION_SECRET=votre-cle-super-secrete-a-generer
```

Dans `app.js` :

JavaScript

```
require('dotenv').config();

app.use(session({
  secret: process.env.SESSION_SECRET,
  // ...
}));
```

Partie 9 : Extensions

Challenge 1 : Page "Mot de passe oublié"

Ajoutez une fonctionnalité de réinitialisation de mot de passe :

1. Génération d'un token unique
2. Stockage dans la BD avec expiration
3. Envoi par email (simulé dans la console)

Challenge 2 : Remember Me

Ajoutez une case "Se souvenir de moi" qui prolonge la session à 30 jours.

Challenge 3 : Liste des Sessions Actives

Créez une table `sessions` dans SQLite et affichez toutes les sessions actives d'un utilisateur.

Challenge 4 : Rôles et Permissions

Ajoutez :

- Table `roles` (admin, user)
- Middleware `requireRole('admin')`
- Page d'administration

🎯 Points Clés à Retenir

1. **HTTP est stateless** : Les sessions permettent de "se souvenir" de l'utilisateur
2. **Cookie de session** : Contient UNIQUEMENT l'ID de session, pas les données
3. **req.session** : Créé par `express-session`, vous y ajoutez vos propres propriétés
4. **Middleware** : Fonction qui s'exécute avant la route finale
5. **bcrypt** : Hasher = impossible de retrouver le mot de passe original