# Une approche de la sécurisation des routes

Si certaines routes se contentent d'exposer des données inoffensives, d'autres agissent directement sur la base de données. Il s'agit notamment de celles qui utilisent les verbes HTTP *POST*, *PUT* et *DELETE*.

Au cours de ce TD, vous mettrez en place un système très simple qui vérifiera la conformité d'un mot de passe envoyé dans l'objet `req` de Express avec l'information contenue dans la base de données.

## Étape 1 : ajouter une table des utilisateurs

Dans le répertoire *private*, vous trouverez un fichier *missions.sql* augmenté des instructions pour ajouter une table `users` et insérer un utilisateur `admin@envol2021.fr`. Importez de nouveau le fichier SQL dans votre système de gestion de bases de données :

```
mysql -u root -p missions < missions.sql
```

## Étape 2 : définir un modèle pour les utilisateurs

Créez, dans le répertoire *models*, un fichier *users.js* qui sera chargé des opérations sur les utilisateurs. Pour le moment, ne définissez qu'une méthode pour récupérer les informations d'un utilisateur particulier :

```
const db = require('../services/db');

/* GET one user */
async function getUserById(id_user) {

    const results = await db.query(
        `SELECT email, password
        FROM users
        WHERE id_user = ?;`,
        [id_user]
    );

    return results;
};

module.exports = {
    getUserById
};
```

Rien de bien nouveau dans cette méthode : elle affiche l'email et le mot de passe d'un utilisateur pour un `id_user` donné.

## Étape 3 : authentifier un utilisateur

À présent, créez un contrôleur *auth.js* dans le répertoire *controllers* et importez le modèle précédent :

```
const users = require('../models/users');
```

Définissez maintenant une méthode asynchrone `checkAuth()` qui se limite pour l'instant de mobiliser la méthode `getUserById()` du modèle *users* :

```
const checkAuth = async (req, res, next) => {
    try {
        const user = await users.getUserById(req.body.id_user)[0];
        // check
    } catch (err) {
        next(err);
    }
};
```

Dans le bloc `try`, vérifier qu'un paramètre `password` envoyé via formulaire correspond exactement à ce qui est enregistré dans la base de données pour un utilisateur donné :

```
const checkAuth = async (req, res, next) => {
    try {
        // a user
        const user = await users.getUserById(req.body.id_user);
        // if given password id equal to the one in database…
        if (req.body.password == user.password) {
            // … continue!
            next();
        }
        // in other cases, send a message
        else {
            res.status(401).send('Utilisateur non autorisé.');
        }
    } catch (err) {
        next(err);
    }
};
```

Il ne vous reste plus qu'à exporter votre méthode :

```
module.exports = {
    checkAuth
};
```

## Étape 4 : assigner la demande d'authentification à certaines routes

Rien de plus simple pour assujettir une route au contrôle d'accès. Importez tout d'abord le contrôleur *auth.js* dans le fichier de routage des requêtes concernant les missions :

```
const auth = require('../controllers/auth');
```

Ajoutez enfin comme deuxième paramètre des routes *POST*, *PUT* et *DELETE* la méthode *auth.checkAuth* :

```
router.post('/add', auth.checkAuth, missions.addMission);
router.put('/update/:id_mission', auth.checkAuth, missions.updateMission);
router.delete('/del/:id_mission', auth.checkAuth, missions.deleteMission);
```

Utilisez à présent le service *Postman* pour vérifier le bon fonctionnement de l'authentification en paramétrant par exemple une requête de type `POST` avec deux paramètres : `id_user` défini à `1` et `password` à `$2y$10$lgj2e81t8h/8.r/6MIHEVus8jVVr/Hw1IEZMX1bwya5LsF.g66Bvi`.

**Vous trouverez le code final des documents HTML et JavaScript dans le dossier *fin* de ce premier TD.**