# TME 7 – Implémentation des services

TME 7 – Implémentation des services

## 7.1 Serveur basique et axios

### 7.1.1 Création du serveur node

Avec Express, créer un serveur écoutant le port 8000 qui :
- renvoie au client la chaîne de caractères "Message reçu" à une requête à l'URL /
- affiche dans sa console le contenu de la propriété texte du `body` de la requête en JSON envoyée en POST à l'URL /

Tester les deux URL avec Postman.

```
const express = require('express');
const port = 8000;

const app=express();

app.use(express.json());

app.get('/', (req, res) =>{
        res.setHeader('Content-type', 'text/plain;charset=UTF-8');
        res.send('Message reçu');
})
.post('/', (req, res) =>{
        console.log("Requête POST reçue : "+req.body.texte);
        res.end();
})

app.listen(port);
```

### 7.1.2 Création du client React

Créer un composant React :
- possédant un champ de formulaire et un bouton d'envoi
- un état `champ` synchronisé avec la valeur du champ de formulaire
- au click sur le bouton, une requête POST est envoyée au serveur contenant le body `texte: champ`

À l'envoi, vous devriez obtenir une erreur CORS. Pour éviter cela, installez sur le serveur le module cors puis ajoutez dans le code les lignes

```
const cors = require('cors');
app.use(cors({origin:"*"}));
```

```
import axios from 'axios';

import { useState } from 'react';

function TestAxios() {
```

```
        const [champ, setChamp] = useState("");

        const envoi = (evt) => {
                evt.preventDefault();
                axios.post('http://localhost:8000/', {texte: champ})
                        .then(console.log("envoi requête POST"));
        }

        const handleInput = (evt) => {
                setChamp(evt.target.value);
        }

  return (
                <form>
                        <label htmlFor="chp">Valeur</label><input id="chp" onChange={handleInput} />
                        <button onClick={envoi}>OK</button>
                </form>
  )
}

export default TestAxios
```

## 7.2 Premier service : Création d'utilisateurs

On va utiliser pour tous les services les codes de statut HTTP :
https://www.restapitutorial.com/httpstatuscodes.html.
Un message d'erreur plus détaillé peut être donné dans la réponse.
Pour rappel, voici la description du service de création d'utilisateur :

| | |
|---|---|
| Nom du web service | CreateUser |
| URL du web service | /user/ avec PUT |
| Description du service | Permet la création d'un nouvel utilisateur |
| Paramètres en entrée | login ;password ;password2 ;nom ;prenom ;... |
| Format de sortie | ok, ou erreur |
| Exemple de sortie | `{status : XXX, message : "xxxxx", details: "yyyyy" }`, par exemple `{status : 201, message : "Utilisateur créé" }` |
| Erreurs possibles | requête mal formulée ; paramètres manquants ; password et password2 différents ; serveur base de données inaccessible ; utilisateur déjà existant |
| Avancement du Service | À faire |
| Classes/Fichiers JavaScript | Users.exists, Users.create |
| Informations additionnelles | xxxx |

### 7.2.1 Donnez le JSON retourné en cas de succès et d'erreur, ainsi que le statut HTTP de la réponse

Code 201 lorsque tout est OK, code 409 lorsque le login existe déjà (*Conflict*), 400 (*bad request*, champs manquants) et 500 si jamais une autre erreur a eu lieu (accès BD, *etc.*).
À noter que le code erreur est au niveau de notre application, alors que le statut HTTP est plus générique (mais on pourrait se passer du code d'erreur au niveau application). On pourrait définir un code d'erreur tout à fait arbitraire renvoyé dans le JSON (par exemple `ERR001`), mais il est plus simple de reprendre le code du statut HTTP.
Pour les JSON

```
// Status 400 (Bad request)
{
```

```
        "status": 400,
        "message": "Champs manquants"
}

// Status 409 (conflict)
{
        "status": 409,
        "message": "utilisateur déjà existant"
}

// Status 201 (Created)
{
        "status": 201,
        "message":  "login ... enregistré"
}
```

#### 7.2.2 Écrire l'algorithme pour réaliser le service `createUser`, en considérant les connexions à la base de données.

Fonction : `createUser`

1. Vérification des paramètres (il ne faut pas que les paramètres soient nuls), sinon *bad request* (400)
2. Vérification que le mot de passe et la confirmation de mot de passe sont identiques, sinon erreur 400
3. Ouvrir la connexion à la base de données
4. Vérifier si un utilisateur avec le même login existe déjà. Si oui erreur 409
5. Ajouter l'utilisateur à la base de données (avec `Users.create`)
6. Fermer la connexion à la base de données
7. Renvoyer 201 et message si tout s'est bien passé

## 7.3 Second service : Login

Vous allez utiliser le squelette fourni (fichier `server.zip`).
Nous allons considérer le système d'authentification suivant :

- Lors du *login*, une clef sera générée automatiquement par le serveur, et transmise sous forme de cookie. Si vous utilisez le *middleware* `express-session`, cela est fait automatiquement
- La clef aura une durée de vie déterminée

#### 7.3.1 Spécifier le service *login*.

| | |
|---|---|
| Nom du web service | Login |
| URL du web service | authentification - en POST |
| Description du service | Permet de récupérer une clef de connexion valide pendant un certain temps |
| Paramètres en entrée | login ;password |
| Exemple de sortie | xxxx |
| Erreurs possibles | utilisateur inconnu (401) ; password incorrect (401) |
| Avancement du Service | A faire |
| Classes JavaScript en rapport avec le Web service | xxxx |
| Informations additionnelles | xxxx |

### 7.3.2 Écrire l'algorithme correspondant

Fonction : `login`

1. Vérification des paramètres (non null), sinon erreur 400
2. Ouvrir la connexion à la base de données
3. Si l'utilisateur n'existe pas : Erreur 401
4. Si le mot de passe est incorrect : Erreur 401
5. Fermer la connexion à la base de données
6. Modification de l'objet de session pour stocker l'identifiant de l'utilisateur
7. retour JSON

### 7.3.3 Quelles sont les fonctions d'interrogation de la base de données dont vous allez avoir besoin ?

```
userExists(login)
checkPassword(login, password)
```

### 7.3.4 Écrire le service correspondant en JavaScript

En s'appuyant sur le middleware `express-session` (on peut aussi le faire à la main, en utilisant des cookies, mais c'est plus fastidieux) avec dans `src/app.js` :

```
const api = require('./api.js');
// (...)
const session = require("express-session");

app.use(session({
        secret: "technoweb rocks",
        resave: true,
        saveUninitialized: false
}));
```

Dans le code de `src/api.js`, on aura

```
const express = require("express");
const Users = require("./entities/users.js");

function init(db) {
        const router = express.Router();
        // On utilise JSON
        router.use(express.json());
        // Pour garder une trace de toutes les requêtes
        router.use((req, res, next) => {
                console.log('API: method %s, path %s', req.method, req.path);
                console.log('Body:', req.body);
                next();
        });
        const users = new Users.default(db);
        router.post("/user/login", async (req, res) => {
                try {
                        const { login, password } = req.body;
                        // Paramètres manquants dans la requête HTTP
                        if (!login || !password) {
                                res.status(400).json({
                                        status: 400,
                                        "message": "Requête invalide : login et password nécessaires"
```

```
                        });
                        return;
                }
                if(! await users.exists(login)) {
                        res.status(401).json({
                                status: 401,
                                message: "Utilisateur inconnu"
                        });
                        return;
                }
                let userid = await users.checkpassword(login, password);
                if (userid) {
                        // Avec middleware express-session
                        req.session.regenerate(function (erreur) {
                                if (erreur) {
                                        res.status(500).json({
                                                status: 500,
                                                message: "Erreur interne"
                                        });
                                }
                                else {
                                        // C'est bon, nouvelle session créée
                                        req.session.userid = userid;
                                        res.status(200).json({
                                                status: 200,
                                                message: "Login et mot de passe acceptés"
                                        });
                                }
                        });
                        return;
                }
                // Faux login : destruction de la session et erreur
                req.session.destroy((err) => { });
                res.status(403).json({
                        status: 403,
                        message: "login et/ou mot de passe invalide(s)"
                });
                return;
        }
        catch (e) {
                // Toute autre erreur
                res.status(500).json({
                        status: 500,
                        message: "erreur interne",
                        details: (e || "Erreur inconnue").toString()
                });
        }
});

return router;
}
exports.default = init;
```

## 7.4 Autres services

**7.4.1 Faire la liste des services de base qui devront être implémentées dans votre projet. Les services seront regroupés par familles :**

- authentification
- utilisateurs
- messages

— demandes

Pour chaque service, vous spécifierez les entrées et sorties. La spécification complète et documentée des services sera à faire chez vous.