Concevoir une application dorsale

Derrière le concept d'application dorsale (*back-end* en anglais), on entend un ensemble d'outils conçus pour traiter les commandes en provenance de l'application frontale (*front-end* en anglais). Durant ce TD, vous serez amené·es à construire un serveur *Node.js* qui ne se chargera que de la gestion des tâches commandées par l'application *AngularJS* à venir, en vous reposant sur une base de données relationnelles MySQL (ou MariaDB).

L'idée de départ est simple : votre laboratoire a décidé d'attribuer un plafond annuel pour les missions de ses membres en fonction de leur statut et, pour faciliter le travail de votre service de gestion, il vous est demandé de monter une petite application qui permette, en plus des opérations courantes (saisie, modification ou suppression de missions), de suivre l'état actualisé du solde à disposition pour chacun·e.

Présentation du code de départ

Dans le dossier *debut*, vous trouverez l'ensemble des fichiers sur lesquels vous travaillerez :

- *private/config.js*, les données de configuration pour la connexion à la base de données ;
- *private/missions.sql*, le code SQL de la base de données avec des enregistrements de départ ;
- *services/db.js*, le module chargé de se connecter à la base de données.

Conception de l'application

Étape 1 : mise en place de la base de données

Assurez-vous de disposer d'un système de gestion de bases de données relationnelles comme MySQL ou MariaDB puis appelez l'utilitaire en ligne de commandes, qui vous demandera de saisir votre mot de passe de connexion :

```
mysql -u {user} -p
```

Créez une nouvelle base de données `missions` :

```
CREATE DATABASE `missions` CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

Puis saisissez les instructions contenues dans le fichier *private/missions.sql* ou sortez du *shell* MySQL et transmettez directement le fichier à l'utilitaire `mysql` (en supposant que vous êtes dans le répertoire *private*) :

```
mysql -u {user} -p missions < missions.sql
```

Étape 2 : démarrage de l'application

Créez un répertoire de travail *missions* puis, *backend* et démarrez un nouveau projet *Express.js* avec `express-generator` :

```
$ mkdir missions && mkdir missions/backend
$ cd missions/backend
```

```
$ npx express-generator
```

Vous aurez également besoin du module *mysql2* pour interagir avec votre base de données :

```
npm install mysql2 --save
```

Configurez ensuite le script de démarrage dans le fichier *package.json* pour que la commande `npm start` lance l'application *app.js* avec la commande `nodemon` :

```
"scripts": {
    "start": "nodemon ./bin/www"
}
```

Pour finir de tout mettre en place, collez les répertoires *private* et *services* fournis au début du TD à la racine de votre application *backend* et vérifiez que l'application se lance bien avec la commande :

```
npm start
```

Étape 3 : un service pour se connecter à la base de données

La prochaine étape consiste à connecter la base de données à votre application. Pour cela, vous définirez la configuration de la connexion dans le fichier *config.js* du répertoire *private* que vous importerez dans le module *db.js* du répertoire *services*. Notez au passage que ce TD ne s'intéresse pas à la sécurisation effective du répertoire.

Configuration de la connexion

Ajoutez, dans le fichier *config.js*, un objet `config` avec les clés `host`, `user`, `password` et `database`. En toute logique, votre hôte devrait s'appeler *localhost* et votre base de données *missions* :

```
const config = {
  host: 'localhost',
  user: '{login}',          // your login
  password: '{password}',   // your password
  database: 'missions'
};
```

Et terminez en exportant l'objet sous forme de module :

```
module.exports = config;
```

Mise en place du service

Le service *db.js* repose sur deux modules :

- *config*, qui prend en charge les paramètres de configuration ;
- *mysql2*, qui sert d'interface avec le système de gestion de bases de données.

Importez ces modules dans le fichier *db.js* :

```
const mysql = require('mysql2');
const config = require('../private/config');
```

Le module *mysql2* expose une méthode `createConnection()` pour établir la connexion avec la base de données. Elle prend en paramètre un objet de configuration :

```
const connection = mysql.createConnection(config);
```

Il ne reste plus qu'à exporter ce nouveau module :

```
module.exports = connection;
```

Étape 4 : exposer la liste des agents

Pour cette étape, vous vous concentrerez sur les opérations liées au R (*read* ou *retrieve*) de l'acronyme CRUD pour ne faire qu'exposer la liste des agents du laboratoire et les missions qu'ils ont effectuées.

Liste des routes nécessaires

Vous recensez les routes suivantes :

- `/agents`, pour lister les agents du laboratoire ;
- `/agents/{id_agent}`, pour obtenir les nom et prénom d'un agent, son statut (chercheur·se, ingénieur·e…) ainsi que le plafond de dépenses qui lui est de fait attribué ;
- `/agents/{id_agent}/missions`, pour listes les missions effectuées par un agent ;
- `/agents/{id_agent}/missions/{id_mission}`, pour obtenir le détail d'une mission effectuée par un agent.

Routage des requêtes concernant les agents

Dans le répertoire *routes*, créez un fichier `agents.js` et commencez par importer les modules nécessaires à la configuration d'un routeur :

```
const express = require('express');
const router = express.Router();
```

Directement à la suite, importez le service de connexion à la base de données. Ce service vous permettra d'exécuter des requêtes SQL et d'en traiter les résultats :

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

Occupez-vous à présent de la route qui permet de lister tous les agents du laboratoire : `/agents`. Il s'agit d'une route qui utilise le verbe `GET` :

```
/* GET agents */
router.get('/', (req, res) => {
    // query
});
```

L'objet `db` met à disposition un ensemble de méthodes pour exécuter des requêtes. La plus simple d'entre elles est la méthode `query()` qui prend deux paramètres :

- une requête SQL ;

- une fonction de *callback* qui prend en charge une erreur éventuelle ou la liste des résulats.

Pour faire les choses simplement, la requête SQL suivante récupère les nom et prénom de chacun·e :

```
SELECT firstname, lastname
FROM agents
ORDER BY lastname ASC;
```

Intégrez-la à la méthode `query()` :

```
/* GET agents */
router.get('/', (req, res) => {
    // query
    db.query(
        `SELECT firstname, lastname
        FROM agents
        ORDER BY lastname ASC;`,
        (err, results) => {
            if (err) res.send(err);
            else res.send(results);
        }
    );
});
```

Exportez le routeur :

```
module.exports = router;
```

Et importez-le dans le fichier *app.js* qui gère le cœur de votre application :

```
const agentsRouter = require('./routes/agents');
```

Plus loin dans ce même fichier, déclarez un *middleware* pour la route `/agents` chargé de mettre en œuvre le routeur `agentsRouter` :

```
app.use('/agents', agentsRouter);
```

Essayez à présent d'effectuer, en modifiant la requête SQL, une jointure avec la table `status` afin de récupérer le statut de chaque agent ainsi que le plafond des dépenses accordé par le conseil de laboratoire.

Testez le résultat en consultant la page http://localhost:3000/agents.

Obtenir le détail des informations d'un agent particulier

Maintenant que vous disposez d'une route pour lister les agents du laboratoire, vous pouvez en tracer une pour obtenir les informations détaillées de chacun. De manière très classique, on s'attend à trouver, à la fin de la route `/agents` un chiffre qui identifie à coup sûr un agent. Par exemple, la route `/agents/5` doit récupérer les informations de l'agent dont l'identifiant `id_agent` vaut 5 dans la table `agents` de la base de données.

Comme il s'agit d'un paramètre variable, il doit être préfixé par le caractère `:`. Vous pouvez en plus le combiner à la route précédente en le suffixant du point d'interrogation `?` qui souligne son caractère facultatif. Transformez ainsi la route écrite précédemment :

```
router.get('/:id_agent?', (req, res) => {
    // query
    …
});
```

Comme vous avez déjà écrit la requête pour le cas où l'identifiant n'est pas présent, il reste à définir l'autre sans oublier de les intégrer à une structure conditionnelle qui teste sa présence dans l'objet `req.params` :

```
router.get('/:id_agent?', (req, res) => {
  if (req.params.id_agent) {
    // query
  } else {
    // query
    db.query(
    `SELECT firstname, lastname
    FROM agents
    ORDER BY lastname ASC`,
    (err, results) => {
      if (err) res.send(err);
      else res.send(results);
    });
  }
});
```

Écrivez à présent la requête SQL, dérivée de la première, qui permet de récupérer les informations détaillées d'un agent :

```
SELECT firstname, lastname
FROM agents
WHERE id_agent = ?;
```

La partie variable de la requête, présente dans l'objet `req.params`, peut-être transmise à la méthode `db.query()` en second paramètre, à l'intérieur d'un tableau :

```
db.query(
    `SELECT firstname, lastname
    FROM agents
    WHERE id_agent = ?`,
    [req.params.id_agent],
    (err, results) => {
      if (err) res.send(err);
      else res.send(results);
    });
```

N'oulbiez pas d'adapter la requête SQL afin de récupérer également le statut de l'agent et le plafond qui lui est accordé !

Routage des requêtes pour les missions des agents

Sur le même principe que la route `/:id_agent?`, tracez une route pour obtenir toutes les missions d'un agent ou le détail de l'une d'elles :

```
/* GET missions by agent */
router.get('/:id_agent/missions/:id_mission?', (req, res) => {
    if (req.params.id_mission) {
        // an id_mission is given
    } else {
        // all missions for a specific agent
    }
});
```

Écrivez le code pour le cas où un identifiant de mission est transmis, en sachant que les informations à récupérer sont :

- les nom et prénom d'un agent à puiser dans la table `agents` ;
- le statut et le plafond des dépenses de l'agent depuis la table `status` ;
- le pays de destination, le coût de la mission ainsi que les dates de départ et de retour à prendre dans la table `missions`.

Pour le second cas, vous souhaitez exposer un objet un peu plus complexe où, pour un agent, vous affichez la liste des missions qu'il a effectuées. La procédure se passe en deux temps, avec deux requêtes successives : la première pour obtenir les informations sur un agent particulier et la seconde pour la liste de ses missions.

Concevez d'abord la structure de l'objet à exposer :

```
const agent = {
    'firstname': '',
    'lastname': '',
    'status': '',
    'cap': 0,
    'missions': []
};
```

Puis écrivez la première requête dont le but est de remplir les quatre premiers champs de l'objet `agent` :

```
/* which agent ? */
db.query(
    `SELECT firstname, lastname, status, cap
    FROM agents, status
    WHERE id_agent = ?
    AND ref_status = id_status`,
    [req.params.id_agent],
    (err, results) => {
        if (err) res.send(err);
        else {
            agent.firstname = results[0]['firstname']
            agent.lastname = results[0]['lastname'];
            agent.status = results[0]['status'];
            agent.cap = results[0]['cap'];
```

```
        }
    }
);
```

Enfin, écrivez la requête qui sélectionne les missions d'un agent particulier :

```
db.query(
    `SELECT country, cost, date_from, date_to
    FROM missions
    WHERE ref_agent = ?`,
    [req.params.id_agent],
    (err, results) => {
        if (err) res.send(err);
        else {
            agent.missions = results;
            res.send(agent);
        }
    }
);
```

Étape 5 : une application CRUD pour les missions

La dernière étape, et non des moindres, consiste à créer des routes pour toutes les opérations de gestion des missions, à savoir :

- lister les missions ;
- afficher le détail d'une mission particulière ;
- créer une nouvelle mission ;
- modifier les informations d'une mission ;
- supprimer une mission.

Liste des routes nécessaires

Un tableau des routes en fonction des opérations peut être dressé à partir des besoins recensés plus haut :

Opération	Verbe HTTP	Route
Lister les missions	GET	/missions
Afficher le détail d'une mission	GET	/missions/:id_mission
Créer une mission	POST	/missions/add
Modifier une mission	PUT	/missions/update/:id_mission
Supprimer une mission	DELETE	/missions/del/:id_mission

Écriture du code

Répétez les étapes suivies pour le routage des requêtes concernant les agents afin de compléter votre application !

Remarques :

- si les variables transmises via l'URL sont accessibles dans un objet `req.params`, les variables transmises avec la méthode `POST` sont quant à elles

disponibles dans un objet `req.body` ;
- en l'absence de frontale, utilisez l'application *Postman* pour tester vos routes autres que `GET`.

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