TP n°2: Persister les données de son API

Dans ce TP, vous allez apprendre à:

- Persister les données de votre API grâce à une base de données MongoDB
- Structurer votre projet en utilisant l'objet router d'express
- Utiliser une source de données externe

Le but de ce TP sera de mettre en place une API permettant de gérer des watchlists.

1 Initialisation du projet

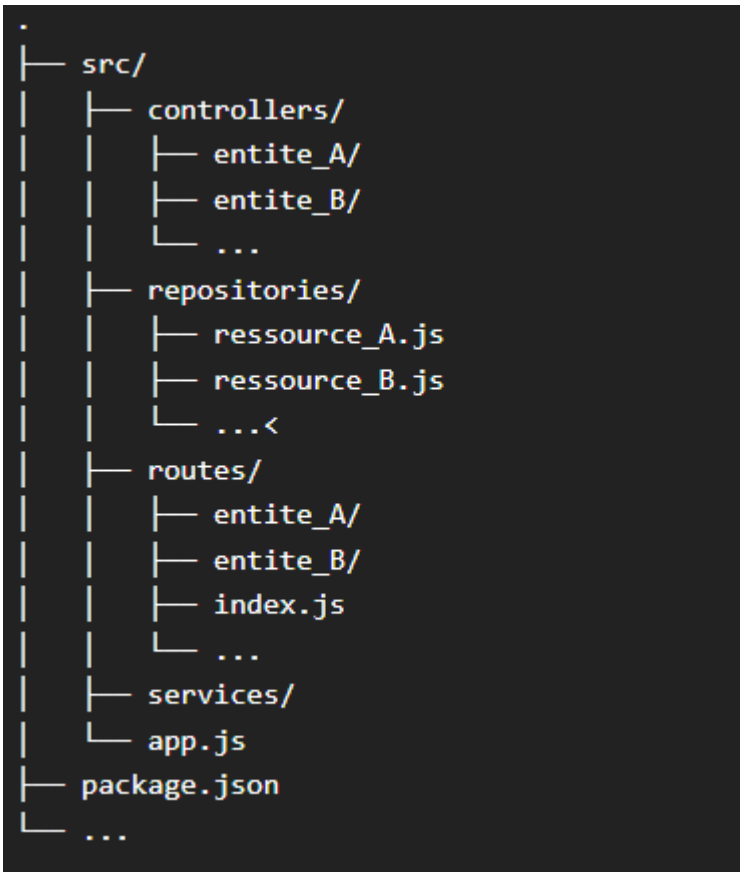
Créez un nouveau dépot git.
Appelez le NOM_PRENOM_TP2_BUT
Pour le TD1: Invitez l'utilisateur `NxiMidway` à votre repo
Pour le TD2: Inviter l'utilisateur antoineoffroy62@gmail.com

Conseil

Pour vous faciliter la vie, je vous conseille d'installer l'extension "Prettier ESLint" sur vscode.
Suivez le guide d'installation, cela vous permettera d'avoir un système d'indentation automatique ainsi que d'un linter(outil d'analyse de code relevant les mauvaises pratiques).

Structure du projet

Pour ce TP, nous utiliserons la structure suivante:



L'ensemble de notre code sera dans un répertoire **src/** (cela permet de séparer le code des fichiers de configurations que l'on place généralement à la racine d'un projet).

Le fichier **app.js** nous permettra de lancer notre API.

Le dossier **routes/** contiendra les différentes routes de notre api. Une bonne pratique est de faire une route par type d'entité. Par exemple, les actions liés aux utilisateurs commenceront tous par `/users`.
On trouve dans ce répertoire le fichier **index.js** qui va relier nos différentes routes et gérer tous ce qu'il se passe au niveau de la racine de notre api.

Le dossier **repositories/** contiendra l'ensemble des appels à des ressources externes (APIs, services ou encore bases de données).

Le dossier **controllers/** contiendra nos middlewares qu'on regroupera par type d'entité.

Enfin, le dossier **services** contiendra toutes les fonctions utilitaires ou avec de la logique avancée qui ne sont pas propres à un type entité en particulier. On peut aussi y mettre les fonctions qui manipulent plusieurs types entités en même temps.

Faciliter le lancement des APIs

Ajoutez les scripts suivants ans votre package.json, :
`"start": "node ./src/app.js"`

Vous pouvez maintenant lancer votre en tapant la commande `npm run start`.

Installation de mongodb

Téléchargez et installez mongodb community edition.

Lors de l'installation, je vous conseille de cocher l'option **Install MongoD as a Service**. Cela permet de lancer un server mongo lors du démarrage de votre pc, vous pouvez choisir de ne pas cocher cette option mais il faudra lancer votre serveur mongo à la main.

Sélectionnez également l'option permettant d'installer mongodb compass, nous en aurons besoin.

Une fois l'installation terminée, ouvrez mongo compass. Si vous avez installé mongo as a service, vous pouvez simplement cliquer sur le bouton "Connect" et compass se connectera directement à votre serveur mongo local (l'url par défaut de mongo est `mongodb://localhost:27017`).

Créez une nouvelle base de données `watchlist` via compass.

2 Mise en place de la connexion avec mongo

Comme pour le tp1, créez votre fichier app.js:

```
const express = require('express')
const app = express()
const port = 3000

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

```
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Pour connectez votre API à votre mongdb, copiez le code ci dessous dans un fichier "connection.js" en respectant cette arborescence:


```
const { MongoClient } = require('mongodb');
const conf = require("../../../conf.json")
// Connection URI
const url = conf.databaseUrl;
const dbName = conf.databaseName;

// Create a new MongoClient
const client = new MongoClient(uri);

async function connectTodB() {
    try {
        console.log('Trying to access the db...');
        // Connect the client to the server (optional starting in v4.7)
        await client.connect();

        // Establish and verify connection
        await client.db('admin').command({ ping: 1 });
        console.log('Connected successfully to server');
    } catch (e) {
        // Ensures that the client will close when you finish/error
        console.log(JSON.stringify(err));
        await client.close();
        throw e;
    }
}

function getCollection(collectionName) {
    return client.db(dbName).collection(collectionName);
}

module.exports = {
```

```
    connectTodB,
    getCollection,
};
```

Créez un fichier conf.json à la racine de votre projet et complétez le afin de faire fonctionner la connexion au serveur mongo.

Lancez votre API, si le message 'Connected successfully to server' apparaît dans la console, vous êtes bien connecté à votre base mongodb.

3 Mise en place des opération CRUD

Les opérations CRUD (Create Read Update Delete) sont essentielles à la gestion de votre base de données.

Créez un fichier crud.js dans le répertoire `db` et ajoutez le code ci dessous:

```
const { getCollection } = require('./connection');

async  function  findOne(collectionName, query, options = {}) {
    try {
        const  collection = getCollection(collectionName);
        const  result = await  collection.findOne(query, options);
        return  result;
    } catch (e) {
        console.log(`Erreur lors de l execution de la fonction findOne avec les
parametres suivants: ${query}`);
        console.log(e);
        throw  e;
    }
}
```

En s'inspirant de la fonction findOne, compléter le fichier crud.js en implémentant les fonctions suivantes*:

- find
- insertOne
- insertMany
- updateOne
- updateMany
- replace
- deleteOne
- deleteMany

**N'hésitez pas à consulter la documentation de mongodb pour vous aider !*

4 Gestion de watchlist

Notre API permettra de gérer des watchlists pour différents utilisateurs.

Une watchlist est constituée de différents films et épisodes de séries qu'on appelera "item". Un item peut avoir 4 état différents quand il est dans une watchlist: "A voir", "En cours", "Terminé" et "Abandonné".

Avant de pouvoir être ajouté à une watchlist, l'utilisateur devra ajouter l'item souhaité à un registre commun à tous les utilisateurs. Une fois l'item ajouté au registre, il sera disponible pour tous les autres utilisateurs et pourra être ajouté dans n'importe quelle watchlist.

Nous utiliserons cette api pour rechercher des items et remplir le registre.

Vous trouverez ci dessous une liste de fonctionnalités. Elles sont volontairement peu détaillées, si vous avez des questions, je jouerai le rôle du client.

Avant de commencer à coder, identifiez les différentes entités à gérer et proposez un modèle de données pour chaque entités. *Vous pouvez utiliser le format json schema (voir section Validation des entités).*

Faites valider votre conception. Une fois cela fait, créer les collections mongodb associées à vos entités dans une base de données "watchlist" via compass.

Fonctionnalités minimales de la watchlist

- Créer un utilisateur (fait)
- Ajouter un item au registre (fait)
- Créer une watchlist pour un utilisateur (fait)
- Ajouter un item dans une watchlist (fait)
- Modifier le statut d'un item dans une watchlist (fait)
- Afficher les items du registre (avec possibilité de filtrer)
- Récupérer la liste des utilisateurs (fait)
- Récupérer la liste des watchlists d'un utilisateur (fait)
- Récupérer le contenu d'une watchlist (fait)

Fonctionnalités supplémentaires:

- Supprimer un item d'une watchlist
- Modifier les informations personnelles d'un utilisateur
- Supprimer une watchlist
- Ajouter une watchlist en favori
- Partager sa watchlist avec un autre utilisateur
- Donner la possibilité d'écrire une note personnelle sur une watchlist ou un item d'une watchlist.
- Mettre en place une page permettant de tester les routes de notre api*

**Voir section Server side rendering*

Vous pouvez également implémenter vos propres fonctionnalités si vous avez des idées.

Appel à l'API omdbapi

Omdbapi est une api qui permet de récupérer des informations sur des films ou des séries. Afin de pouvoir utiliser cette api, il vous faudra une apiKey.
Pour cela, il suffit de faire une demande via le site de l'API.
Il est important de noter que cette apiKey vous permet d'effectuer **1 000 appels par jour**.

C'est suffisant pour l'usage de ce tp, cependant je vous conseille d'éviter de faire des appels dans des boucles. et de faire attention à vos appels d'une manière générale.

Nous allons avoir besoin de faire des appels HTTP depuis notre propre API.
Node possède nativement un module `http` qui permet de faire ce genre d'appel.
Cependant, je vous conseille d'utiliser le package axios qui est plus pratique.

Pour suivre la structure du projet, le code relatif aux appels omdbapi devra être dans le répertoire `repositories`

Validation des entités

Une bonne pratique est de valider le format des données avant une insertion en bdd. Cela permet de s'assurer d'avoir un format uniforme pour chaque entité de la collection.

Un moyen simple de faire cela est d'uiliser des jsonschema.

Dans un premier temps, il faudra créer un jsonchema pour chaque type d'entité de votre API.
Voici un exemple de jsonschema: .

```
{
    "title": "Ecran",
    "description": "Description d un ecran",
    "type": "object",
    "properties": {
        "reference": {
        "type": "string"
    },
    "numero_serie": {
        "type": "number"
    },
    "fabriquant": {
        "type": "object",
        "properties": {
            "nom": {
                "type": "string"
            },
            "mail_support": {
                "type": "string",
                "format": "email"
            }
        }
    },
    "composants": {
        "type": "array",
        "items": {
            "type": "string"
        }
    }
    },
    "required": ["reference"],
```

```
    "additionalProperties": false
}
```

Je vous conseille si besoin de regarder également cette page qui détaille les différents keywords.

Voici également deux mots clés utiles (utilisable dans la description d'objets):

- required: permet de préciser des champs obligatoires (sans la présence de ces champs, la validation échoue)
- additionalProperties: Booleen permettant ou non la présence de champs non décrits dans votre jsonschema.

Pour notre API, je vous propose d'utiliser le package npm jsonschema qui permet de valider simplement des objets en fonction d'un jsonschema.

Server side rendering

Express permet de faire du rendu server. C'est à dire qu'il est capable de retourner des pages web grâce à une vue et à un moteur de modèle.

Je vous propose d'utiliser pug afin de créer une page de présentation permettant d'utiliser les différentes routes de votre api à l'image de celle d'omdbapi. Vous trouverez un exemple d'implémentation sur le repository contenant les cours.

Voir la documentation de pug et res.render pour plus d'informations

Rendu du TP

Il suffit de me transmettre votre repository github. Je téléchargerai vos repos le **9 Mars 2023 vers 20h**. J'invite ceux qui ne m'ont pas encore envoyé leur repository de le faire **immédiatement**. En cas de repository absent lors de la deadline, une **grosse** pénalité sera appliquée à la note.

Une attention toute particulière sera apportée à la propreté de votre repository ainsi qu'a la qualité de votre code (notamment au niveau du respect de la structure vue pendant le cours).

N'hésitez pas à me contacter via discord si vous avez des questions après la fin du cours: Midway#1115

Ressources

Voici une liste de ressources utiles pour le développement de votre API:

- W3 school, site contenant des ressources utiles pour apprendre Node.
- Lodash, librairie qui contient de nombreuses fonctions utilitaires.
- Documentation du driver node mongodb
- Documentation d'express
- stack overflow en cas de soucis (car un bon développeur est un développeur qui sait comment trouver le bon topic stack overflow)