TEST DRIVEN DEVELOPMENT

Avant de commencer les exercices, un petit rappel sur le test driven development

1) Introduction

Le test driven development (ou TDD) est un concept où l'on écrit les tests avant de s'attaquer au code.

Nos tests ratent donc tous au début, le but est donc d'agencer son code de manière à ce que tous nos tests réussissent.

2) Quels sont les avantages ?

Faire du TDD permet de mettre en place les contraintes et attente de notre projet avant meme de commencer à coder.

Ça permet de mettre en place simplement le cahier des charges de notre projet, ça permet d'éviter les erreurs et oubli, ça facilite également l'élaboration du projet, pour résumer simplement le TDD nous permet de savoir ce qu'on veut.

MISE EN PRATIQUE DU TDD

Votre mission, si vous l'acceptez (vous n'avez pas le choix soyons clair), est d'écrire les tests de notre futur API qui aura les spécificités suivantes:

Votre mission sera d'écrire des tests unitaires ansi que des tests d'intégration.

L'API sera hébergé en local, son adresse sera http://localhost:8080

Elle dispose d'une route /book, qui autorise les requêtes GET et POST

Elle dispose d'une route /book/:id qui autorise les requêtes GET et PUT et DELETE

———

Route /book :

———
———

La requête GET permet de récupérer tous les livres stockés dans notre base de donnée, le format de la réponse données est le suivant :

```
{
  "books": [
    {
      "id": "55b7d315-1a5f-4b13-a665-c382a6c71756",
      "title": "Oui-Oui contre Dominique Strauss-Kahn",
      "years": "2015",
      "pages": "650"
    }
  ]
}
```

On récupère un objet avec une clé books, qui est un tableau d'objet.

En cas de succès l’api retourne un status 200

En cas d’échec, l’api retourne un status 400 ainsi que l’objet suivant:

{ “message”: “error fetching books”}

———————————————————————————————————————

La requête POST permet d’ajouter un livre a notre base de données,

La data envoyé dans le body est de format x-www-form-urlencoded

Le body de notre requête aura la forme suivante

```
{
    title: ‘Oui-Oui contre Elizabeth II',
    years: 1990,
    pages: 400
}
```

En cas de succès l’api retourne un status 200 ainsi que l’objet suivant:

{ “message”: “book successfully added”}

En cas d’échec, l’api retourne un status 400 ainsi que l’objet suivant:

{ “message”: “error adding the book”}

Route /book/:id :

———————————————————————————————————————
———————————————————————————————————————

La requête GET récupère le livre dont l’id est renseigné, l’objet retourné par notre base de donnée est dans le format suivant :

```
{
   "message": "book fetched",
   "book": {
      "id": "55b7d315-1a5f-4b13-a665-c382a6c71756",
      "title": "Oui-Oui contre l’abbé Pierre",
      "years": "2012",
      "pages": "335"
   }
}
```

L’api retourne également un statut 200.

Si l’id n’existe pas l’api retourne un status 400 ainsi que l’objet suivant :

{ "message": "book does not exist"}

En cas d’échec, l’api retourne un status 400 ainsi que l’objet suivant:

{ “message”: “an Error occured”}

———————————————————————————————————————

La requete /PUT modifie le livre dont l'id est renseigné,
En cas de succès l'api renvoi un status 200 ainsi que l'objet suivant :

{ message: 'book succesfully updated' }

Si l'id n'existe pas l'api retourne un status 400 ainsi que l'objet suivant :

{ "message": "book does not exist"}

Le body de notre requête aura la forme suivante

```
{
    title: 'Oui-Oui contre Elizabeth II',
    years: 1990,
    pages: 400
}
```

En cas d'erreur l'api renvoie l'objet suivant:

{ message: 'an Error occured' }

———

La requête /DELETE supprime le livre dont l'id est renseigné,

En cas de succès l'api renvoi un status 200 ainsi que l'objet suivant :

{ message: 'book succesfully deleted'}

Si l'id n'existe pas l'api retourne un status 400 ainsi que l'objet suivant :

{ "message": "book does not exist"}

En cas d'erreur l'api renvoie l'objet suivant:

{ message: 'an Error occured' }

———

MISE EN PLACE DES TESTS D'INTEGRATION

On devra créer une batterie de tests où l'on s'assure que l'api répond correctement,

On utilisera les modules chai et chai-http, ainsi que la fonction resetDatabase pour effectuer la réinitialisation de la base de données entre chaque test.

<u>Première série de tests (Empty Database)</u>

Dans un premier temps, on s'assure que la base de données soit initialisée avec l'objet suivant :

```
{
  books: []
};
```

Pour la route /book avec l'appel /GET :

Les tests devront avoir les spécificités suivantes :

Il devront s'assurer que le body de la réponse soit un objet
Que la réponse ait un status 200
Que la clé books soit un tableau
Que la taille du tableau books soit de zero
Pour la route

Pour la route avec l'appel /POST :

Les tests devront avoir les spécificités suivantes :

Que la réponse ait un status 200
Que la clé message dans l'objet body contiennent : "book successfully added"

Seconde serie de test (Mocked Database)

Pour la deuxième série de test, on s'assure que la base de données sois rempli avec l'objet suivant :

```
{
 books: [
   {
     id: '0db0b43e-dddb-47ad-9b4a-e5fe9ec7c2a9',
     title: 'Coco raconte Channel 2',
     years: 1990,
     pages: 400
   }
 ]
}
```

Pour la route /book/:id avec l'appel /PUT :

Les tests devront avoir les spécificités suivantes :

Que la réponse ait un status 200
Que la clé message dans l'objet body contiennent : "book successfully updated"

Pour la route /book/:id avec l'appel /DELETE :

Que la réponse ait un status 200
Que la clé message dans l'objet body contiennent : "book successfully deleted"

Pour la route /book/:id avec l'appel /GET :

Que la réponse ait un status 200
Que la clé message dans l'objet body contiennent : "book fetched"
Que la clé book dans l'objet body soit un objet
Que la clé title dans l'objet book soit une chaine de charactere
Que la clé title dans l'objet book soit égale a la clé title de la database mocké
Que la clé years dans l'objet book soit un nombre
Que la clé years dans l'objet book soit égale a la clé years de la database mocké

Que la clé pages dans l'objet book soit un nombre
Que la clé pages dans l'objet book soit égale a la clé pages de la database mocké

TESTS UNITAIRE

En ce qui concerne les tests unitaires, nous allons utiliser un package nommé nock, qui va permettre de mocker les appels à notre base de données.

Pour expliquer simplement, nock va override les méthodes http.request et http.ClientRequest de node.js.

Nock va intercepter les appels et simuler la réponse voulue, ce qui est parfait pour tester votre code en isolation.

On devra s'assurer également que les intercepteurs soient clean entre chaque test.

Premiere série de test (simulation de réponse ok)

Les tests devront avoir les spécificités suivantes :

/GET

Que la réponse ait un status 200
Que la clé books de la réponse simulé soit un array

/POST

Que la réponse ait un status 200
Que la clé message de la réponse simulé soit :

'book successfully added'

/PUT

Que la réponse ait un status 200
Que la clé message de la réponse simulé soit :

'book successfully updated'

/DELETE

Que la réponse ait un status 200
Que la clé message de la réponse simulé soit :

'book successfully deleted'

Seconde série de test (simulation de mauvaise réponse)

/GET

Que la réponse ait un status 400
Que la clé message de la réponse simulé soit :

‘error fetching books’

/POST

Que la réponse ait un status 400
Que la clé message de la réponse simulé soit :

‘error adding the book’

/PUT

Que la réponse ait un status 400
Que la clé message de la réponse simulé soit :

‘error updating the book’

/DELETE

Que la réponse ait un status 400
Que la clé message de la réponse simulé soit :

‘error deleting the book’