

NodeJS API - A5 - Glados demande des tests

Vous n'avez pas la ref du titre ? Allez jouer à Portal 2 !

Notre objectif de ce TP est d'apprendre à mettre en place des tests pour notre back-end.

01. Test unitaires

Explication

Un test unitaire vérifie le bon fonctionnement d'une unité de code isolée, comme une fonction ou un middleware. L'objectif est de s'assurer que chaque composant fonctionne correctement en dehors de tout contexte global.

J'insiste sur la valeur du mot isolé !!

Mise en place.

Pour faire des tests unitaires, nous allons avoir besoin de bibliothèques externe. Il en existe une multitude dans le domaine de JS. La majorité utilise le BDD (Behavior Driven Development) comme base. Nous allons partir sur `Jest`.

Comme toujours un petit npm install en dev cette fois-ci

```
npm install --save-dev jest
```

Je vous invite à avoir la [doc](#) ouverte pour vous aider.

Dans notre projet, nous allons créer un dossier test. Comme nous n'avons pas beaucoup de code métier, nous allons créer un fichier `Person.test.js` dans ce dossier. Si nous avons d'autre partie de code à tester, nous pourrions créer d'autre fichiers.

Votre mission, à l'aide de la doc, réaliser un code qui test la création d'une `Person` et test bien que les getter retourne la bonne info.

Pour exécuter notre test, il nous faudra modifier notre package.json comme suit :

```
{
  "scripts": {
    "test": "jest"
  }
}
```

Il est possible que test existe déjà dans vos scripts donc remplacez juste le contenu !

Maintenant, plus qu'à `npm test` et vérifier que tout est bon. Essayez maintenant de casser le test en changeant l'un des getter (genre `getAge` retourne l'âge +1). Le tests ne devrait donc plus fonctionner ! Et c'est là tout l'intérêt. Votre code est maintenant un peu plus sûr et qualitatif !

Mission 01

Réalisez une classe `MinorPerson`. Celle-ci héritera de `Person`. En cas d'âge supérieur ou égal à 18, votre classe devra lever une exception. Votre classe devra aussi présenter un `setAge(int)` qui garantie que la contrainte métier est respecté. Evitez la répétition de code et réalisez les tests qui s'impose (testez les cas valide et les erreurs)

Dans l'optique où le test unitaire se résume sur une unité isolé de code, je vous invite à lire la partie Mock Functions de la documentation. Puis nous allons nous focaliser pour réaliser un mock de axios sur l'api [Random User](#) . Il est possible que l'api ne retourne que des personne majeures. N'hésitez pas à utiliser l'âge d'inscription si besoin.

Mission 02

Réalisez une classe `PersonDAO` avec une methode `get50Random()`. Celle-ci utilisera axios (qu'il faudra installer) pour appeler l'api et utiliser la réponse pour retourner un array de `Person`. (Evitez l'async/await pour vous faciliter la vie)

Puis, réalisez le test de cette méthode en mockant le module axios.

Et enfin, ajustez votre test pour que toute personne de moins de 18 ans soit de type `MinorPerson` (et ajustez le code en fonction)

Si tout c'est bien déroulé, nous avons testé uniquement la partie transformation de la donnée et non la partie réception (qui est le rôle d' `axios` que nous n'avons pas à tester).

`Jest` vous offre plein d'autre possibilités mais nous n'avons pas le temps de tout explorer. (Vous reste-t-il seulement du temps encore ?)

02. Test d'intégration

Explication

Un test d'intégration vérifie que plusieurs composants de l'application fonctionnent correctement ensemble, y compris l'intégration avec une base de données.

Mise en place

Nous allons mettre en place un test qui vérifie le fonctionnement d'un endpoint complet (On contrôle les input pour vérifier les output). On ne vérifie donc pas ce qu'il se passe pour chaque système, mais que la somme de ceux-ci ont bien le comportement attendu.

Pour cela, nous allons ajouter une nouvelle dépendance :

```
npm install supertest
```

Mission 03

Réalisez les tests pour vérifier que notre endpoint `POST /hello/person` est valide. Il y a au moins 3 cas à tester =>

- On envoi un payload complet
- On envoi un payload incomplet
- On envoi aucun payload

Aucun de ces choix ne devrait planter et retourner un code 200 et un body

N'hésitez pas à explorer en changeant le comportement de la route si besoin.

Pour finaliser cette partie, nous allons tester le processus de création et de récupération d'info depuis notre base de données ! Créez donc un fichier `user.test.js` . Je vous

donne un squelette de base pour vous aider à réaliser votre test. Chaque endroit commenté devrait avoir du code.

```
// Vos imports

describe('insert', () => {
  beforeAll(async () => {
    //Gère la connexion via mongoose
  });
  afterAll(async () => {
    //Permet de clean les collections créées avec cette connexion

    // Se déconnecter
  });

  it('should create a user, then get it by email and check ID match',
  async () => {
    //Instancie notre service

    // Utilise le service pour créer un nouvel user

    // Utilise le service pour récupérer l'user par mail

    //Vérifie que les 2 id sont les même (donc on a bien récupéré celui
    que l'on vient de créer)
  });
});
```

Mission 04

Réalisez le test de création et récupération d'utilisateur.

03. Autres types de tests

Il existe plein d'autres test automatisable. Je vous invite à vous renseigner dessus (internet, IA, collègues) :

- Tests de bout en bout (End-to-End)
- Tests d'acceptation
- Tests de performance
- Smoke tests

- PenTest (Test de pénétration)
- Test de Charge
- ...

[Plein d'infos](#)