



Tests unitaires

Animé par Mazen Gharbi

React & test

- ▶ En utilisant « create-react-app », React intègre un outil de testing :



- ▶ Chaque fichier de test est représenté par une extension « test.js »

A snippet from a file explorer showing two files: "Link.js" with a yellow JavaScript icon and "Link.test.js" with a yellow test icon. The "Link.test.js" file is highlighted with a blue underline.

- ▶ Et une ligne de commande simple pour enclencher les tests :

```
Test Suites: 1 failed, 1 total
Tests:      0 total
Snapshots:  0 total
Time:       2.464s
```



Watch Usage

- › Press a to run all tests.
- › Press f to run only failed tests.
- › Press q to quit watch mode.
- › Press p to filter by a filename regex pattern.
- › Press t to filter by a test name regex pattern.
- › Press Enter to trigger a test run.

App.test.js

▷ Voici le contenu généré par défaut :

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import App from './App';
```

App.test.js

```
it('renders without crashing', () => {  
  const div = document.createElement('div');  
  ReactDOM.render(<App />, div);  
  ReactDOM.unmountComponentAtNode(div);  
});
```

On crée une div dynamiquement

Retire un composant React monté du DOM et nettoie ses gestionnaires d'événements et son état local.

- ▷ « it » permet de déclarer un test unitaire
 - › Fournit par JEST

Plusieurs fonctions fournies

[Liste des fonctions](#)

Intitulé du test

```
it('Test de la fonction du meilleur cours', () => {  
  expect(quelEstLeMeilleurCours()).toBe('ReactJS');  
});
```

▷ **expect** prend une valeur, et en association à **toBe**, il permet de vérifier que la valeur est bien égale au résultat attendu

```
test('valeurs numériques', () => {  
  expect(100).toBeWithinRange(90, 110);  
  expect(101).not.toBeWithinRange(0, 100);  
  expect({ apples: 6, bananas: 3 }).toEqual({  
    apples: expect.toBeWithinRange(1, 10),  
    bananas: expect.not.toBeWithinRange(11, 20),  
  });  
});
```

Thématique de test

▷ Il est possible de définir des thématiques de test

```
describe('Les différentes synthaxes en JEST', () => {  
  it('Premier test simple', () => {  
    expect(1 + 2).toEqual(3);  
    expect(2 + 2).toEqual(4);  
  });  
  
  it('Simple boolean', () => {  
    expect([1]).toBeTruthy();  
    expect(0).toBeFalsy();  
  });  
  
  it('Manipulation sur objet', () => {  
    const houseForSale = {  
      bath: true,  
      bedrooms: 4  
    };  
  
    expect(houseForSale).toHaveProperty('bath');  
    expect(houseForSale).toHaveProperty('bedrooms', 4);  
    expect(houseForSale).not.toHaveProperty('pool');  
  })  
});
```

Gestion des tests asynchrones

► Soit le code suivant :

```
export default function asynchronousRequest() {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve(true);  
    }, 2000);  
  });  
}
```

Test.js

Test.spec.js

```
it('Tester le retour asynchrone', async () => {  
  expect.assertions(1); // S'attend à 1 appel asynchrone durant ce test  
  await expect(asynchronousRequest()).resolves.toBeTruthy();  
});
```

Problème

✓ Tester le retour asynchrone (2004ms)

► Un test doit être **F.I.R.S.T**

- ✓ Tester le retour asynchrone (1ms)

Mock

► Un appel au serveur étant coûteux en terme de temps, nous allons **bypasser** le comportement natif d'axios pour **simuler** l'appel

```
import axios from 'axios';
```

user.service.js

```
class Users {  
  static getAllUsers() {  
    return axios.get('/users').then(resp => resp.data);  
  }  
}
```

user.service.test.js

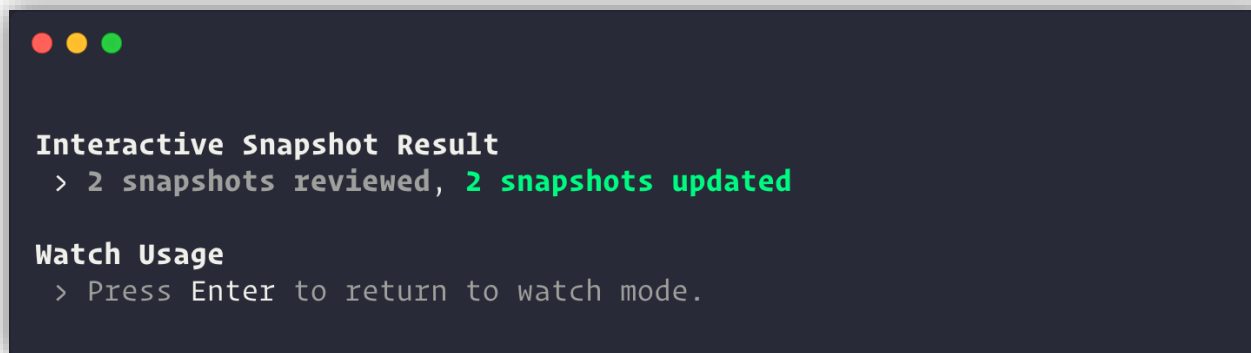
```
jest.mock('axios'); // SURCHARGE !
```

```
it('Récupère les utilisateurs du site', () => {  
  const users = [{name: 'Bob'}];  
  const resp = {data: users};  
  axios.get.mockResolvedValue(resp);
```

```
  return Users.getAllUsers().then(data => expect(data).toEqual(users));  
});
```


Snapshots

- ▶ Jest a introduit une notion de tests « **Snapshot** »
 - › Capture
- ▶ Permet de **figer l'état** d'un composant à un moment **t** et de comparer les différents « snapshot »
 - › Permet de repérer les éventuels différences

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The text inside the terminal is white, with some words in green for emphasis. It shows the 'Interactive Snapshot Result' and 'Watch Usage' sections.

```
Interactive Snapshot Result
> 2 snapshots reviewed, 2 snapshots updated

Watch Usage
> Press Enter to return to watch mode.
```

Snapshots

- ▷ Si le snapshot change et que l'on compare les différentes version, le « **test snapshot** » échouera ;
 - › Ce n'est pas forcément une mauvaise chose !
- ▷ Une fois le « Snapshot test » échoué, **vous décidez** si la différence doit provoquer un échec du test ou non.
 - › Très utile pour nous assurer que l'interface utilisateur ne change pas de manière inattendue
- ▷ Avec les snapshot, nos tests seront bien plus légers, mais pour l'utiliser il va être nécessaire d'installer « **react-test-renderer** »

```
› npm install react-test-renderer
```

Test d'un composant

Link.js

```
export default class Link extends React.Component {
  constructor(props) {
    super(props);

    this._onMouseEnter = this._onMouseEnter.bind(this);
    this._onMouseLeave = this._onMouseLeave.bind(this);

    this.state = {
      class: STATUS.NORMAL,
    };
  }

  _onMouseEnter() {
    this.setState({class: STATUS.HOVERED});
  }
}
```

Link.js (suite)

```
  _onMouseLeave() {
    this.setState({class: STATUS.NORMAL});
  }

  render() {
    return (
      <a
        className={this.state.class}
        href={this.props.page || '#'}
        onMouseEnter={this._onMouseEnter}
        onMouseLeave={this._onMouseLeave}
      >
        {this.props.children}
      </a>
    );
  }
}
```

Test d'un composant

```
import Link from './Link';
import renderer from 'react-test-renderer';

it('Link changes the class when hovered', () => {
  const component = renderer.create(
    <Link page="http://www.facebook.com">Facebook</Link>,
  );
  let tree = component.toJSON(); // Génère arbre
  expect(tree).toMatchSnapshot(); // Vérifie si le snapshot match la dernière version

  // Trigger manuellement la méthode onMouseEnter()
  tree.props.onMouseEnter();
  // Ré-affichage
  tree = component.toJSON();
  expect(tree).toMatchSnapshot();

  // Trigger manuel
  tree.props.onMouseLeave();
  // Ré-affichage
  tree = component.toJSON();
  expect(tree).toMatchSnapshot();
});
```

Simule l'affichage du composant

On reprend un snapshot

Fonctionnement

▷ A la première exécution du test précédent, Jest va créer le snapshot suivant :

```
exports[`Link changes the class when hovered 1`] = `<a className="normal"
  href= " http://www.facebook.com "
  onMouseEnter={{[Function]}}
  onMouseLeave={{[Function]}} >
  Facebook
</a> `;
```

▷ Aux tests suivants, Jest comparera les prochains snapshot avec ceux pris précédemment pour aller rapidement

- › S'il ne passe pas, le test échoue

Fichier snapshot généré

Link.test.js.snap

```
1 // Jest Snapshot v1, https://goo.gl/fbAQLP
2
3 exports[`Link changes the class when hovered 1`] = `
4 <a
5   className="normal"
6   href="http://www.facebook.com"
7   onMouseEnter=[[Function]]
8   onMouseLeave=[[Function]]
9 >
10 | Facebook
11 </a>
12 `;
13
14 exports[`Link changes the class when hovered 2`] = `
15 <a
16   className="hovered"
17   href="http://www.facebook.com"
18   onMouseEnter=[[Function]]
19   onMouseLeave=[[Function]]
20 >
21 | Facebook
22 </a>
23 `;
24
25 exports[`Link changes the class when hovered 3`] = `
26 <a
27   className="normal"
28   href="http://www.facebook.com"
29   onMouseEnter=[[Function]]
30   onMouseLeave=[[Function]]
31 >
32 | Facebook
33 </a>
34 `;
35 |
```

Mettons à jour notre test

▷ On fait le choix de modifier l'url :

```
const component = renderer.create(  
  <Link page="http://www.macademia.fr">Macademia</Link>,  
);
```

▷ Et puis l'erreur survient :

```
- Snapshot  
+ Received  
  
  <a  
    className="normal"  
-   href="http://www.facebook.com"  
+   href="http://www.macademia.fr"  
    onMouseEnter={[[Function]]}  
    onMouseLeave={[[Function]]}  
  >  
-   Facebook  
+   Macademia  
  </a>
```

Mettre à jour les snapshot

► Comme indiqué précédemment, c'est à nous développeur d'indiquer si la différence entre les 2 snapshots est un « *bug ou une feature* »

Watch Usage

- > Press a to run all tests.
- > Press f to run only failed tests.
- > Press p to filter by a filename regex pattern.
- > Press t to filter by a test name regex pattern.
- > Press u to update failing snapshots.
- > Press i to update failing snapshots interactively.
- > Press q to quit watch mode.
- > Press Enter to trigger a test run.

—

Tester vos composants avec Enzyme

- Les snapshots sont bien pratiques, mais ils ne permettent pas de tester la **logique** de nos composants

```
i> npm i --save-dev enzyme enzyme-adapter-react-16
```

- Nous allons donc utiliser **Enzyme** pour simuler la création d'un component



Enzyme

▷ Enzyme fournit une fonction « **mount** » qui nous permettra de simuler l’affichage d’un composant

```
import { configure, mount } from 'enzyme';  
import Adapter from 'enzyme-adapter-react-16';
```

```
configure({ adapter: new Adapter() });
```

Permet d’adapter la librairie à notre version de React

```
it('Devrait faire l\'affichage qu\'on lui demande', () => {  
  const wrapper = mount(  
    <Link>Mon texte</Link>  
  );  
  const p = wrapper.find('a');  
  expect(p.text()).toBe('Mon texte');  
});
```

Simuler un événement

▷ Il est possible de simuler un click très simplement avec Jest

```
it('Simuler le click', () => {  
  const wrapper = mount(  
    <Link>Mon texte</Link>  
  );  
  const p = wrapper.find('a');  
  p.simulate('click');  
});
```

TP

- ▷ Créez un composant simple permettant d'afficher une liste d'éléments
- ▷ Le composant doit contenir un **input** et un **bouton** pour valider
- ▷ **Testez** ce composant en vérifiant que la fonction permettant d'ajouter un élément fonctionne
- ▷ Testez ce composant avec **différents snapshots**

Questions ?