Formation sur mesure

React avancé

Introduction

Plan

---

1) Rappels JS et React
2) Architecture des composants
3) Qualité de code
4) Etude de code
5) Travaux pratiques

Rappels JS et React

- Gestion du state
- Props particulière
- Cycle de rerendering
- Traitement asynchrone
- Curryfication

Plan

---

1) **Rappels JS et React**
2) Architecture des composants
3) Qualité de code
4) Etude de code
5) Travaux pratiques

Immutabilité du state (1/2)

Faire :

```
const [value, setValue] = useState();

const handleValueChange = (v) => {
    setValue(v);
}
```

Ne pas faire :

```
let [value] = useState();

const handleValueChange = (v) => {
    value = v
}
```

Immutabilité du state (2/2)

- Ne pas utiliser les fonctions “impures” (‘*.push*’, ‘*.pop*’, ‘*delete*’, etc…)
- Manipuler le state au travers de copie manuelle :
  - Utiliser le spread operator (‘...’)
  - Utiliser des fonctions pures (‘*.map*’, ‘*.filter*’, ‘*Object.values*’, etc…)
- Manipuler le state au travers de copie générée (‘*immutable.js*’, etc…)

Uplifting du state

Une donnée partagée entre deux composants “*siblings*” doit être remontée dans le *state* parent et transmises aux enfant au travers des *props*.

La modification de ce *state* par les enfants se fera au travers de callback définie dans le parent et transmise aux enfants par les *props*.

Props particulière

- ***children***

Représente le JSX se trouvant entre la balise ouvrante et la balise fermante du composant.

- ***ref***

Permet au parent de récupérer le noeud DOM de la balise dans *ref.current*

- ***key***

Permet d’assurer l’unicité d’un élément dans le render d’une liste.

L’absence d’unicité mène à une inconsistance du render (ex. : mise à jour partielle du rendu).

Cycle de rerendering

- **Mount** : Déclenche tous les *useEffect*

```
useEffect(() => {/*...*/})
useEffect(() => {/*...*/}, [])
useEffect(() => {/*...*/}, [/*...*/])
```

- **Update** : Déclenche les *useEffect* sans dépendances et avec les dépendances concernées

```
useEffect(() => {/*...*/})
useEffect(() => {/*...*/}, [/*...*/])
```

- **Unmount** : Déclenche les callbacks *return* par les *useEffect*

```
useEffect(() => {
    return () => {/* cleanup */}
})
```

Traitement asynchrone

```
new Promise((resolve, reject) => {
    /* async process */
    if (success) {
        resolve(data)
    } else {
        reject(error)
    }
})
    .then((data) => {/* succes handling */})
    .catch((error) => {/* error handling */})
```

```
const asyncFunc = async () => {
    /* sync process */
    const asyncResult = await otherAsyncFunc();
    /* sync process */
}
```

```
asyncFunc()
    .then((data) => {/* success handling */})
    .catch((err) => {/* error handling */})
```

A voir: [Jake Archibald : In the loop - JSConf.Asia]

Curryfication

Non curryfié :

```
const myFct = (a, b, c) => {
    /* use a, b & c */
}

const result = myFct(1, 2, 3);
```

Curryfié :

```
const myFct = (a) => (b) => (c) => {
    /* use a, b & c */
}

const myFctWithADefined = myFct(1);
const myFctWithAandBDefined = myFctWithADefined(2);
const result = myFctWithAandBDefined(3);

const sameResult = myFct(1)(2)(3);
```

Architecture des composants

- Domain Driven Development
- Atomic Design
- Stateless vs Stateful

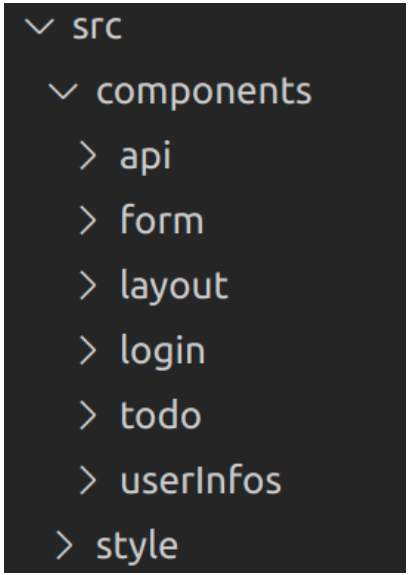
Plan

---

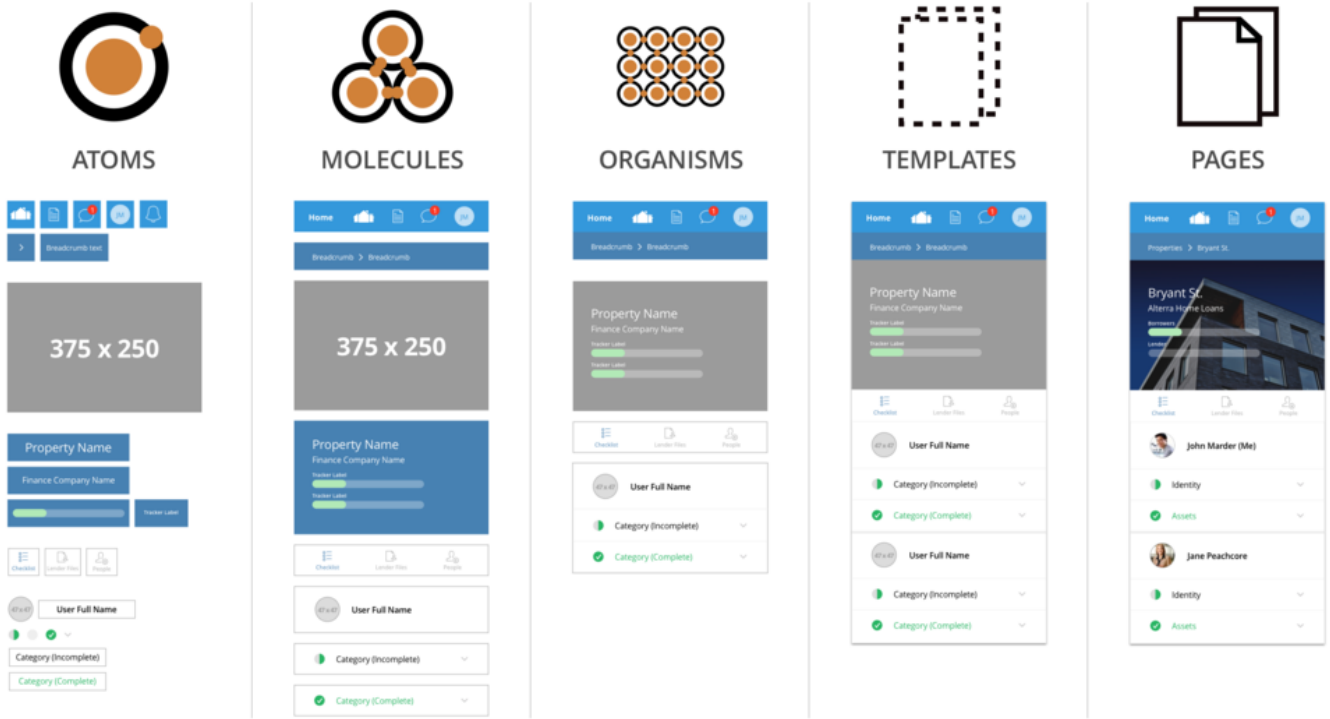
1) Rappels JS et React
2) **Architecture des composants**
3) Qualité de code
4) Etude de code
5) Travaux pratiques

Domain Driven Development

Regrouper les fichiers par domaine métier. Les morceaux de code / fichier concernant le même sujet doivent être regroupés dans un même dossier.



Atomic Design



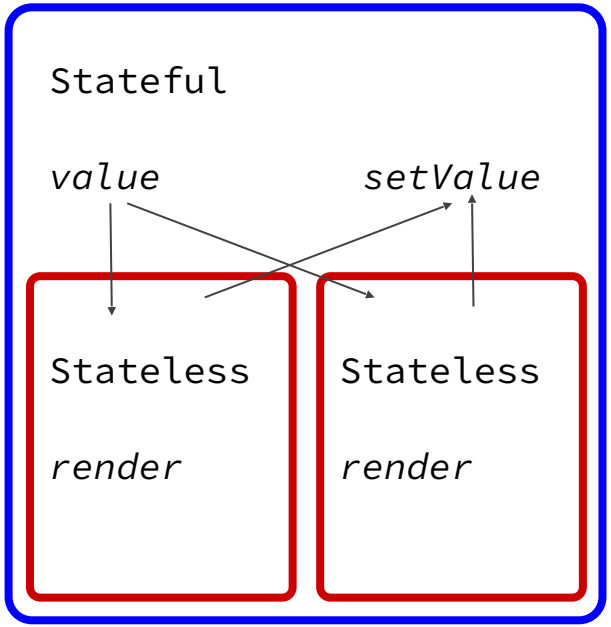
Stateless vs Stateful

- **Stateful**

Ces composants détiennent la donnée et a pour rôle d'implémenter la logique métier.

- **Stateless**

Ces composants reçoivent la donnée et les callbacks au travers des props. Il a pour rôle d'afficher la donnée et déclencher les comportements.



Qualité de code

- Formatage de code
- Typage de données
- Tests automatisés

Plan

---

1) Rappels JS et React
2) Architecture des composants
3) **Qualité de code**
4) Etude de code
5) Travaux pratiques

Formatage de code

Typage de données

- **Proptypes**
  - Typage faible
  - Explicite le contrat d’interface des composants
  - Facilite la gestion d’erreur au runtime
  - Conserve le Javascript “standard”
  - Propre à React
- **Typescript**
  - Typage fort
  - Explicite le type de l’ensemble des variables
  - Permet d’anticiper les erreurs de typage au moment de l’écriture
  - Surcouche à Javascript demandant une configuration spécifique
  - Non spécifique à React

React et TypeScript

- **Les types de base de React**
  - `React.ReactNode`
  - `JSX.Element`
  - `JSX.Element[]`
  - `React.ReactChildren`
  - `React.ReactChild[]`
  - `React.ReactFragment`

- **A lire:** React TypeScript Cheatsheet

Tests automatisés (1/2)

Objectif :

S’assurer que le comportement d’un point de vue utilisateur est correct. Il faudra donc interagir directement avec le DOM comme le ferait un utilisateur.

Sauf cas particulier, les tests ne doivent pas dépendre de l’implémentation et les assertion doivent vérifier l’état du DOM.

Source : https://testingjavascript.com/

Test automatisés (2/2)

Utilisation de *Jest* et de *react-testing-library*.

- **Jest :**

Test runner + assertion library

- **react-testing-library :**

Permet de rendre les composants dans le JS DOM et d'interagir avec ce rendu.

```
describe("My Component", () => {
    let wrapper;
    let props;

    beforeEach(() => {
        props = {/* ... */}
        wrapper = render(<MyComponent {...props} />)
    });

    it("should do behavior A when action A", () => {
        /* ... */
        expect(result).toBe(expected)
    })

    it("should do behavior B when action B", () => {
        /* ... */
        expect(result).toBe(expected)
    })
})
```

Etude de code

- Multiples rerendering
- Hooks et custom hooks
- Context API
- Gestion des formulaires
- Gestion du style (CSS)

Plan

---

1) Rappels JS et React
2) Architecture des composants
3) Qualité de code
4) **Etude de code**
5) Travaux pratiques

Multiples rerendering

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout multi-state-equal-multi-rendering*
- *npm install*
- *npm start*

Objectif :

Comprendre l’impact d’une mauvaise gestion de state.

(state calculé, multiple state, etc…)

Hooks et custom hooks

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout optimization-and-custom-hooks*
- *npm install*
- *npm start*

Objectif :

Comprendre l’utilité de certain hooks d’optimisation.

Être en mesure d’écrire ses propres hooks.

Context API

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout context-with-custom-hooks*
- *npm install*
- *npm start*

Objectif :

Savoir utiliser le contexte React.

Simplifier l'utilisation du contexte grâce à un hook custom.

Context API vs Redux

- Cloner le dépot https://github.com/Zenika/react-context-vs-redux
- *npm install*
- *npm start & nom run api*
- *git checkout with-context* (To see the use of Context API)
- *git checkout with-redux* (To see the use of Redux)

Objectif :

Comprendre les differences entre Redux et les Context React.

Comprendre comment synchroniser un front-end avec une API REST

Gestion des formulaires

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout forms*
- *npm install*
- *npm start*

Objectif :

Comparer la gestion manuelle d'un formulaire à la gestion déléguée à une bibliothèque.

Savoir utiliser Formik.

Gestion du style (CSS)

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout own-style-vs-materiel-ui*
- *npm install*
- *npm start*

Objectif :

Comparer la gestion du style avec CSS à la main, avec Sass à la main et avec une bibliothèque.

Comprendre le fonctionnement du Sass.

Travaux pratiques

- Mise en place du Sass
- Gestion de l’authentification
- Gestion du rerendering d’une liste
- Gestion d’un formulaire “create” et “update”
- Tests automatisés

Plan

---

1) Rappels JS et React
2) Architecture des composants
3) Qualité de code
4) Etude de code
5) **Travaux pratiques**

Prérequis

- Cloner le dépot https://github.com/Zenika/formation-react-sur-mesure
- *git checkout tp-0*
- *npm install*
- *npm install --prefix back*
- *npm run back*
- *npm start*

Le *npm run back* et le *npm start* doivent être lancer en parallèle pour que le front puisse se connecter au backend.

TP 0

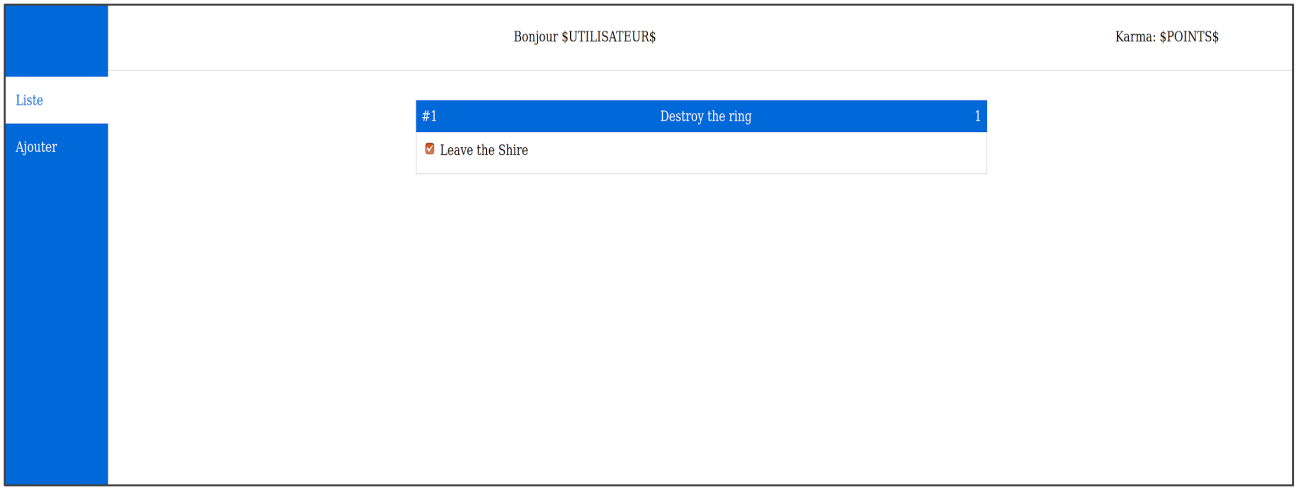
Mise en place du Sass

Départ : *git checkout tp-0*
Fin : *git checkout tp-1*

Objectifs :

- Mettre en place le Sass
- Utiliser les variables Sass
- Comprendre le fonctionnement du composant *Layout*

TP 0 - Instructions



Couleur principale : #0069D9
Couleur de hover : #005ab9

TP 1

Mise en place du login et des requêtes authentifiées

Départ : *git checkout tp-1*
Fin : *git checkout tp-2*

Objectifs :

- Gérer un formulaire simpliste à la main
- Faire des requêtes asynchrone
- Mettre en place le contexte et l'utiliser
- Mettre en place un hooks custom pour faire des requêtes authentifier en gérant les erreurs 401

TP 1 - Instructions (1/2)

- Installer axios (*npm install axios*)
- Faire un *POST /login* avec un body tel que :

  { username, password }

- Créer le contexte *UserInfosContext* contenant un objet
- Y stocker la réponse de succès dans la clé *token* puis déplacer l'utilisateur sur */*
- Ignorer les échecs

Pour tester, aller voir la liste des utilisateurs dans le fichier

back/index.js (login = *firstName*, mot de passe = *password*)

TP 1 - Instructions (2/2)

- Utiliser les informations de *UserInfosContext* pour faire un hook mettant à disposition un axios faisant des requêtes authentifiées et interceptant les 401 pour rediriger vers la page de login
- Au *mount* du *Header*, faire un appel *GET /users/current* et rajouter au *UserInfosContext* la réponse
- Utiliser le contenu de *UserInfosContext* pour afficher correctement le *Header*
- ***BONUS :*** Lors d'un 401 sur le site, en plus de la redirection, ajouter une étape dans l'historique contenant l'url de la page actuelle. Cette étape sera utilisée pour la redirection au moment de la connexion.

TP 2

Gestion de la liste des todos

Départ : *git checkout tp-2*
Fin : *git checkout tp-3*

Objectifs :

- Gérer l'affichage d'une liste
- Manipuler la transmission de props et de callback pour déclencher des comportements
- Utiliser la curryfication pour assurer la séparation des responsabilités

TP 2 - Instructions

- Récupérer la liste des todos au mount du composant *TodoList* avec *GET /tasks*
- Afficher la liste des tasks et subtasks (attention à la props *key*)
- Au *check* d'un subtask, identifier la task et subtask concernée, la mettre à jour et transmettre l'info au back sur *PUT /tasks/:id*. Utiliser la task en réponse pour mettre à jour le render
- A la validation d'un subtask, mettre à jour les userInfos affichée dans le header (le karma peut changer)

TP 3

Ajout et modification d’un Todo

Départ : *git checkout tp-3*
Fin : *git checkout master*

Objectifs :

- Gérer un formulaire avec Formik
- Gérer un tableau d’input dynamique (gestion basique)
- Mettre en place la validation du formulaire

TP 3 - Instructions (1/2)

- Utiliser *Formik* pour gérer le formulaire d'ajout. La valeur initiale est *{ name: "", priority: 1, subtasks: [{ name: "", done: false }]*
- Faire un *POST /tasks* avec les données du formulaire à la validation
- Rajouter un bouton de modification sur les items de la *Todolist* menant vers */update/:id*
- Récupérer la todo et l'injecter dans le formulaire s'il y a un *id* dans l'url
- Mettre en place une mécanique de loader pour n'afficher le formulaire qu'une fois la todo chargée

TP 3 - Instructions (2/2)

- A la validation, faire un *PUT /tasks/:id* à la place du *POST* si un *id* est présent dans l'url
- Rajouter des règles de validation telles que :
  - Le titre ne peut pas être vide
  - Le titre doit faire entre 5 et 50 caractères (inclus)
  - La priorité ne peut pas être vide
  - La priorité doit être entre 1 et 3 (inclus)
  - Les subtasks ne peuvent pas être vide
- Si les règles ne sont pas respectées, afficher un message d'erreur et bloquer la validation du formulaire