Etape 3 : SPA et react-router

Pré-requis

Vous devez maîtriser les étapes 0, 1 et 2 du workshop afin de pouvoir réaliser l'étape 3.

Le code disponible dans cette étape correspond au résultat attendu des étapes 0, 1 et 2. Vous pouvez partir de cette base pour développer l'étape 3.

Pour lancer l'application de l'étape 2, exécutez la commande `npm start` (après avoir fait un `npm install`). Ouvrez ensuite l'URL http://localhost:8080 dans votre navigateur.

Dans cette étape, vous allez avoir besoin de l'API. Pour l'exécuter, lancez la commande `npm start` dans le dossier `api`. La documentation de l'API est disponible à l'adresse http://localhost:3000

Vous avez également la possibilité de lancer les tests de cette étape (que nous avons rédigé pour vous) en utilisant la commande `npm test` afin de voir quelles parties de l'étape fonctionnent et quelles parties ne fonctionnent pas du tout. N'hésitez pas à lire le code des tests afin d'avoir quelques indications en plus sur la façon d'écrire votre application.

Objectif

Maintenant que notre application possède les fonctionnalités de base, nous allons commencer a nous attaquer à la navigation.

En effet pour le moment notre application regroupe toutes les fonctionnalités dans un seul et même écran, ce qui s'avère très pratique d'un point de vue technique mais qui n'est pas génial d'un point de vue utilisateur.

Nous allons donc découper notre application en 3 écrans successifs, permettant de choisir la région d'un vin,

puis le vin a consulter

et enfin la fiche de consultation du vin.

Ici chaque sélection entrainera la navigation jusqu'à l'écran suivant, etc ...

Mais comme nous sommes dans un contexte frontend et que nous ne sommes pas la pour coder du backend, nous allons développer tout ca sous forme d'une SPA (Single Page Application).

Une SPA est une application web (front) accessible via une page unique. Le but est d'éviter le chargement d'une nouvelle page à chaque action demandée et donc de fluidifier l'expérience utilisateur.

Il va donc nous falloir un moyen de router l'utilisateur à travers divers écrans.

Un moyen simple pourrait être d'avoir un composant technique au plus haut niveau de notre application pour gérer la navigation. Ce composant gérerait la pile d'appel et la vue courante dans son `state` et proposerait une API pour être piloté depuis les diverses vues.

Par exemple, nous pourrions définir un composant comme suivant :

```
const Navigator = React.createClass({
  propTypes: {
    initialRoute: React.PropTypes.shape({
      component: React.PropTypes.func.isRequired,
      title: React.PropTypes.string,
      props: React.PropTypes.object,
    }).isRequired,
  },
  getInitialState() {
    return {
      component: null,
      title: null,
      props: null,
    };
  },
  componentDidMount() {
    this.setState({
      component: this.props.initialRoute.component,
      title: this.props.initialRoute.title,
      props: this.props.initialRoute.props,
    });
  },
  navigateTo({ component, title, props }) {
    this.setState({ component, title, props });
  },
  render() {
    const Component = this.state.component;
    const { title, props } = this.state;
    return (
      <Component
        {...props}
        navTitle={title}
        navigator={{ navigateTo: this.navigateTo }} />
    );
  }
});

const Page2 = React.createClass({
  ...
});

const Page1 = React.createClass({
  gotoNext() {
    this.props.navigator.navigateTo({
      title: 'Page 2',
      component: Page2,
      props: {
        foo: 'bar',
      },
    });
  },
  render() {
    return (
      <div>
        <h2>Hello World!</h2>
        <button onClick={this.gotoNext}>Next</button>
      </div>
    );
```

```
  }
});

ReactDOM.render(
  <Navigator initialRoute= {{ title: 'Page 1', component: Page1 }} />,
  document.getElementById('main')
);
```

Cependant, ce genre d'approche a l'inconvénient de perdre la navigation courante lorsque l'on recharge la page. Du coup il existe de meilleures solutions, notamment, `react-router que nous allons utiliser pour gérer la navigation de notre application.

Commençons par ajouter une dépendance pour `react-router` dans l'application.

Dans le fichier `package.json` ajoutez la dépendance suivante :

```
"dependencies": {
    ...
    "react-router": "2.0.1",
    ...
}
```

vous pouvez évidemment l'ajouter via la ligne de commande :

```
npm install --save react-router@2.0.1
```

Maintenant nous pouvons commencer l'intégration du router (l'intégration de base est présente dans le projet mais vous pouvez tout de même lire les paragraphes suivant).

Pour ce faire, commençons par lire l'introduction à `react-router` puis importons les APIs dans `src/app.js`. Pour des raisons de testabilité, nous allons faire en sorte d'encapsuler toute l'application et son système de routage dans un composant dédié dans `src/app.js` et faire en sorte que ce composant puisse recevoir une API `history` dédiée (différente en environnement de test). La montage de ce composant dans le DOM sera effectué dans le fichier `src/index.js`

```
import { Router, Route, browserHistory, IndexRoute } from 'react-router';
```

l'initialisation du routeur se fera de la façon suivante :

```
import { Router, Route, browserHistory, IndexRoute } from 'react-router';
import NotFound from './components/not-found';

export const App = React.createClass({
  propTypes: {
    history: PropTypes.object,
  },
  render() {
    const history = this.props.history || browserHistory;
    return (
      <Router history={history}>
        <Route path="/" component={???}>
          <IndexRoute component={???} />
          ...
          <Route path="*" component={NotFound} />
        </Route>
      </Router>,
    );
```

```
  }
});
```

Ici nous configurons le routeur pour utiliser l'API `history`, tirée de HTML5, du navigateur comme URL de routage côté client. Cette API permet de faire varier l'URL du navigateur sans pour autant déclencher un rechargement de la page.

```
const history = this.props.history || browserHistory;
<Router history={history}> ... </Router>
```

puis nous spécifions un container qui aura le role d'afficher la vue courante du router et qui sera le point d'entrée de l'application.

```
<Route path="/" component={???}> ... </Route>
```

D'après le tutorial de `react-router`, ce genre de composant peut s'écrire de la façon suivante :

```
const MyApp = React.createClass({
  render() {
    return (
      <div>
        <h1>App</h1>
        {this.props.children}
      </div>
    )
  }
});
```

ensuite nous spécifions la vue à afficher pour une navigation vers `/`

```
<IndexRoute component={???} />
```

et enfin nous spécifions une route permettant d'attraper tous les appels n'ayant pu être routés

```
<Route path="*" component={NotFound} />
```

Pour notre application, nous vous proposons de respecter les schémas d'url suivants :

- `/` => vue des régions
- `/regions/:regionId` => vue des vins de la région
- `/regions/:regionId/wines/:wineId` => vue du vin selectionné

Ce routage est défini dans le routeur via l'utilisation du composant

```
<Route path="/mon/path" component={MonComponent} />
```

Pour passer des paramètres aux routes et les récupérer, vous pouvez déclarer vos routes comme ceci

```
<Route path="/mon/path/:monId" component={MonComponent} />

const MonComponent = React.createClass({
  render() {
    return (
      <div>Valeur de monId: {this.props.params.monId}</div>
    );
  }
})
```

enfin vous pouvez créer des liens en utilisant l'API

```
import { Link } from 'react-router';

...

<Link to="/mon/path/1234">Chose 1234</Link>
```

de `react-router` ou en utilisant directement l'API `history` disponible à travers le contexte `react`.

Pour celà, il est nécessaire de spécifier sur le composant routé, un `contextType` afin de valider le contenu du contexte

```
export const Page = React.createClass({

  contextTypes: {
    router: React.PropTypes.object
  },

  handleNavigationTo(path) {
    // ici on déclenche la navigation vers l'url /view/${path}/details
    this.context.router.push({
      pathname: `/view/${path}/details`
    });
  },

  render() {
    ...
  }
}
```

Un dernier petit conseil, vos composants existent déjà et sont idiots (Dumb Components). Ce qui veut dire qu'ils n'ont pas d'état propre, et fonctionnent uniquement via les propriétés qui leur sont passés. Autrement dit, ce sont des composants stateless.

Ce genre d'approche est plutôt intéressante car elle permet de bien séparer ce genre de composants des composants intelligents (Smart Components) qui eux sont souvent stateful et technique sans forcément produire des éléments graphiques. (voir cet article sur le sujet)

Dans le cadre de notre application, il serait intéressant de garder nos composants graphiques simple tel qu'ils sont, et les wrapper dans des composants intelligents qui se chargeront des appels HTTP et de la gestion de l'état

- `RegionsPage` => `Regions`
- `WinelistPage` => `Winelist`
- `WinePage` => `Wine`

Les composants `RegionsPage`, `WinelistPage` et `WinePage` sont mis a votre disposition dans les fichiers `src/components/regions.js`, `src/components/wine-list.js`, `src/components/wine.js`

A vous de jouer !

Surtout ne restez pas bloqués ! N'hésitez pas à demander de l'aide aux organisateurs du workshop ou bien à jeter un oeil au code disponible dans la version corrigée ;-)

Bonus

A priori vous avez créé un composant type `container` lié à l'URL `/` qui a pour seul rôle de contenir les différentes pages de l'application. Il pourrait être intéressant que ce composant affiche le titre courant de la vue (`Regions` -> `Wines from Bordeaux` -> `Cheval Noir`). Pour cela une petite astuce, le meilleur moyen à votre disposition est de rajouter une fonction de mise à jour du titre courant dans les props de la vue rendue à l'intérieur du container principal. Il vous est donc possible de cloner l'élément à render et de lui rajouter des propriétés de la façon suivante :

```
const WineApp = React.createClass({
  render () {
    return (
      <div className="grid">
          <div className="1/2 grid__cell">
            {this.props.children && React.cloneElement(this.props.children, {
              uneNouvelleProps: 'Hello World!'
            })}
          </div>
      </div>
    );
  }
});
```

Prochaine étape

Une fois cette étape terminée, vous pouvez aller consulter la version corrigée puis aller jusqu'à l'étape suivante