ReactJS - Fonctionnalités avancées

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

ReactJS - Fonctionnalités avancées

Table of Contents

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Redux Toolkit

Problème initial en React

React permet à un composant de gérer son propre état (`useState`, `useReducer`), mais **quand l'application devient complexe**, on a besoin de :

- partager l'état entre plusieurs composants éloignés ;
- avoir un état global, modifiable de manière prévisible.

Redux (avant Redux Toolkit)

Redux a été inventé pour résoudre ce problème :

- Un **store global** unique qui contient tout l'état
- Des **actions** pour demander un changement
- Des **reducers** pour dire comment l'état évolue

Mais Redux « classique » est :

- très **verbeux** (on écrit beaucoup pour peu d'effet) ;
- **fragile** (erreurs faciles à introduire) ;
- **peu agréable à utiliser sans beaucoup de configuration** (immutabilité, middleware, devtools… à ajouter à la main).

Redux Toolkit

Redux Toolkit (RTK) est une **boîte à outils officielle** pour Redux, proposée par l'équipe Redux elle-même.

1. **Réduire le code répétitif**
2. **Gérer automatiquement les bonnes pratiques (immutabilité, devtools, middleware)**
3. **Rendre Redux agréable à utiliser avec TypeScript**
4. **Guider les développeurs vers une architecture claire et maintenable**

Redux Toolkit (Vue d'ensemble des APIs)

API	Rôle
`configureStore`	Crée le store Redux moderne
`createSlice`	Crée une portion d'état (slice), avec ses actions et reducers
`createAsyncThunk`	Crée des actions pour des appels asynchrones (API, délais...)
`createAction`	Crée manuellement une action Redux
`createReducer`	Crée manuellement un reducer (si on n'utilise pas `createSlice`)
`createEntityAdapter`	Gère les collections normalisées (CRUD)
`createListenerMiddleware`	Réagit à des actions (effets secondaires façon Redux-Saga)

1. `configureStore`

`configureStore` est une fonction de Redux Toolkit qui **remplace** `createStore` de Redux classique.

- créer un store global ;
- l'initialiser avec un ou plusieurs `reducer`s ;
- activer automatiquement les devtools ;
- intégrer `redux-thunk` par défaut pour les appels asynchrones.

```
const store = configureStore({
  reducer: {
    counter: counterReducer,
    users: userReducer
  },
```

1. `configureStore`

- **Store** : objet global contenant l'état de l'application.
- **Reducer** : fonction pure qui calcule un nouvel état à partir de l'ancien + action.
- **Devtools** : Outils de développement Redux pour visualiser les actions, le state et le time-travel.

2. `createSlice`

Un **slice** (traduction : "tranche") est une **portion du state global** avec

- un nom ;
- un état initial ;
- une série d'**actions** ;
- des **reducers** associés à ces actions.

2. createSlice

```
const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment(state) {
      state.value += 1;
    }
  }
});
```

2. `createSlice`

- `counterSlice.reducer` → à mettre dans le `store`
- `counterSlice.actions` → à utiliser avec `dispatch`

> Il regroupe ce qui était séparé : types, action creators, reducers. C'est un pattern **cohérent et modulaire**.

Redux Toolkit utilise **Immer**.
Immer est une bibliothèque qui permet de **travailler avec des objets immuables de façon mutative**.

```
state.value += 1;
```

3. `createAsyncThunk`

Redux classique ne gère **pas** les actions asynchrones nativement.
Il fallait :

- créer 3 actions : `pending`, `fulfilled`, `rejected` ;
- les appeler manuellement dans un middleware (`redux-thunk`).

3. `createAsyncThunk`

`createAsyncThunk` automatise tout ça.

```
export const fetchUsers = createAsyncThunk('users/fetch', async () => {
  const response = await fetch('/api/users');
  return await response.json();
});
```

RTK crée automatiquement :

- `users/fetch/pending` → quand l'appel commence
- `users/fetch/fulfilled` → quand la réponse est reçue
- `users/fetch/rejected` → si une erreur survient

Dans `createSlice`, on réagit à ces actions dans `extraReducers`.

4. `createAction` et `createReducer`

- `createAction` permet de créer manuellement une action (rare si on utilise `createSlice`)
- `createReducer` permet de créer un reducer avec une syntaxe déclarative.

Utile dans des cas avancés ou pour des usages spécifiques (lib partagée, traitement global).

```
const reset = createAction('counter/reset');
const reducer = createReducer(initialState, builder =>
  builder.addCase(reset, () => ({ value: 0 }))
);
```

5. `useSelector` et `useDispatch`

Ces deux hooks sont fournis par la bibliothèque `react-redux`, pas Redux Toolkit directement.

`useSelector`

Permet à un composant React de **lire une portion du store**.

```
const counter = useSelector((state: RootState) => state.counter.value);
```

> Il est **réactif** : si la valeur change dans le store, le composant se met à jour.

`useDispatch`

Permet à un composant React **d'envoyer une action au store**.

```
const dispatch = useDispatch<AppDispatch>();
dispatch(increment());
dispatch(fetchUsers());
```

> Il peut envoyer une action synchrone (`increment()`) ou une action asynchrone (`fetchUsers()`).

React Query

React Query (maintenant appelé TanStack Query) est une bibliothèque de gestion d'état serveur qui révolutionne la façon dont nous gérons les données distantes dans les applications React. Contrairement aux solutions de gestion d'état traditionnelles comme Redux qui se concentrent sur l'état local, React Query se spécialise dans la gestion des données provenant du serveur.

React Query

Simplification du code : React Query élimine des centaines de lignes de code boilerplate nécessaires pour gérer les requêtes, le cache, les états de chargement et les erreurs. Là où vous auriez besoin de plusieurs hooks useState, useEffect et de logique complexe, React Query fait tout automatiquement.

Cache intelligent : Le système de cache intégré stocke automatiquement les réponses des API et les réutilise intelligemment. Si vous naviguez entre des pages qui utilisent les mêmes données, elles s'affichent instantanément depuis le cache.

React Query

Synchronisation automatique : React Query maintient vos données synchronisées avec le serveur en arrière-plan. Il peut refetch automatiquement quand l'utilisateur revient sur l'onglet, quand la connexion réseau est rétablie, ou selon des intervalles définis.

Optimisations de performance : La bibliothèque implémente automatiquement des optimisations comme la déduplication des requêtes (si deux composants demandent les mêmes données simultanément, une seule requête est faite), le prefetching intelligent, et la gestion efficace du re-rendering.

React Query

L'installation se fait via npm ou yarn :

```
npm install @tanstack/react-query
# ou
yarn add @tanstack/react-query
```

React Query

staleTime : Détermine combien de temps les données sont considérées comme "fraîches". Pendant cette période, React Query ne fera pas de nouvelles requêtes automatiquement.

gcTime : Durée pendant laquelle les données inactives restent en cache. Après ce délai, elles sont supprimées de la mémoire.

retry : Nombre de tentatives automatiques en cas d'échec de requête.

ReactQueryDevtools : Outil de développement qui permet de visualiser l'état du cache, les requêtes actives, et déboguer facilement.

React Query

- **Fetching, caching et synchronisation automatique des données**

Le cœur de React Query repose sur le hook `useQuery` pour récupérer les données :**Concepts clés du fetching :**

Query Key : La clé de requête est un identifiant unique qui permet à React Query de mettre en cache et récupérer les données. Les clés hiérarchiques (`['users', userId]`) permettent une gestion fine du cache.

États de requête : React Query fournit plusieurs états utiles : `isLoading` (première charge), `isFetching` (requête en cours), `isError`, `isSuccess`, etc.

React Query

Synchronisation automatique : React Query refetch automatiquement les données dans plusieurs situations : retour sur l'onglet, reconnexion réseau, montage d'un nouveau composant utilisant des données périmées.

Prefetching : Vous pouvez précharger des données avant qu'elles soient nécessaires, améliorant l'expérience utilisateur.

React Query

- **Pagination et chargement infini**

React Query excelle dans la gestion de la pagination avec `useInfiniteQuery` :**Concepts clés de la pagination :**

keepPreviousData : Maintient les données précédentes affichées pendant le chargement de la nouvelle page, évitant l'effet de clignotement.

useInfiniteQuery : Hook spécialisé pour le chargement infini qui accumule les données de toutes les pages.

getNextPageParam : Fonction qui détermine le paramètre de la page suivante à partir de la dernière page récupérée.

Mutation et gestion des erreurs

Mise à jour optimiste : Les données sont immédiatement mises à jour dans l'interface avant la confirmation du serveur, offrant une expérience utilisateur plus fluide.

Rollback automatique : En cas d'erreur, React Query restaure automatiquement l'état précédent grâce au contexte sauvegardé dans `onMutate`.

Mutation et gestion des erreurs

Cycles de vie des mutations :

- `onMutate` : Avant l'appel API (mise à jour optimiste)
- `onSuccess` : Après succès de l'API
- `onError` : En cas d'erreur (rollback)
- `onSettled` : Dans tous les cas (nettoyage)

Mutation et gestion des erreurs

Invalidation du cache : `invalidateQueries` force React Query à considérer certaines données comme périmées et les recharger.

Gestion d'erreurs avancée : React Query permet une gestion fine des erreurs avec retry personnalisé, différents messages selon le type d'erreur, et récupération gracieuse.

Synchronisation réseau : React Query détecte automatiquement les changements de connectivité et peut relancer les mutations échouées quand la connexion revient.

Qu'est-ce qu'une PWA ?

Une Progressive Web App est une application web qui utilise des technologies modernes pour offrir une expérience utilisateur similaire à une application native. Elle combine le meilleur des applications web et mobiles.

PWA Caractéristiques principales

- **Progressive** : Fonctionne pour tous les utilisateurs, quel que soit le navigateur
- **Responsive** : S'adapte à tous les formats d'écran
- **Connectivité indépendante** : Fonctionne hors ligne grâce aux Service Workers
- **App-like** : Interface et navigation similaires aux apps natives
- **Fraîche** : Toujours à jour grâce aux Service Workers
- **Sûre** : Servie via HTTPS pour éviter les intrusions
- **Re-engageante** : Push notifications et écran d'accueil

PWA - Avantages

- **Performance** : Chargement rapide même sur des connexions lentes
- **Engagement** : Taux de rétention supérieur aux sites web classiques
- **Coût** : Une seule base de code pour toutes les plateformes
- **SEO** : Indexable par les moteurs de recherche
- **Pas de store** : Distribution directe sans validation des stores

Comprendre la mémoïsation et son importance

> La **mémoïsation** est une technique d'optimisation qui permet de **mémoriser** le résultat d'une fonction ou d'un rendu pour ne pas le recalculer inutilement.

React ré-exécute tous les composants à chaque rendu (re-render), même si les props n'ont pas changé. C'est ici qu'intervient la mémoïsation.

mémoïsation

- Évite les **recalculs coûteux** (ex : tri, calculs, rendu SVG, graphique, etc.)
- Améliore les **performances perçues**
- Réduit les **rendus inutiles** de composants enfants

Utilisation de `React.memo` pour les composants fonctionnels

C'est un **Higher-Order Component** (HOC) qui va mémoriser le rendu d'un composant si ses `props` **n'ont pas changé**.

```
const MyComponent = React.memo(function MyComponent({ name }) {
  console.log("Rendered!");
  return <p>Hello {name}</p>;
});
```

`useMemo` et `useCallback` pour les hooks

- `useMemo(fn, deps)` → mémorise une **valeur de retour**

```
const sortedList = useMemo(() => sortList(list), [list]);
```

> Si `list` ne change pas, `sortList()` **n'est pas réexécuté**. On récupère l'ancien résultat.

`useMemo` et `useCallback` pour les hooks

- `useCallback(fn, deps)` → mémorise une **fonction**

```
const handleClick = useCallback(() => {
  console.log("Clicked");
}, []);
```

> La fonction **reste identique** entre les renders, utile quand on passe des fonctions à des enfants mémoïsés avec `React.memo`.

Code Splitting et lazy loading avec `React.lazy` et `Suspense`

Découper ton code en **morceaux chargés dynamiquement**, pour **accélérer le chargement initial**.

- **React.lazy()**

Permet de **charger un composant à la demande**, comme un import dynamique :

```
const About = React.lazy(() => import('./About'));
```

Code Splitting et lazy loading avec `React.lazy` et `Suspense`

- **Suspense**

Permet d'afficher un **fallback** pendant le chargement :

```
<Suspense fallback={<div>Chargement...</div>}>
  <About />
</Suspense>
```

Tests avancés

Quand on utilise **Redux Toolkit** ou **React Query**, nos composants dépendent :

- soit du **store Redux** (useSelector, useDispatch),
- soit du **cache React Query** et du comportement asynchrone (useQuery, useMutation),
- soit de **hooks personnalisés** qui encapsulent cette logique.

Tests avancés

1. Tests des composants Redux & React Query

```
// helpers/test-utils.tsx
import { render } from "@testing-library/react"
import { Provider } from "react-redux"
import { store } from "../redux/store"
import { QueryClient, QueryClientProvider } from "@tanstack/react-query"
const createTestQueryClient = () =>
  new QueryClient({
    defaultOptions: {
      queries: { retry: false }, // éviter les retries pendant les tests
    },
  })

export function renderWithProviders(ui) {
  const queryClient = createTestQueryClient()
  return render(
    <Provider store={store}>
      <QueryClientProvider client={queryClient}>
        {ui}
      </QueryClientProvider>
```

Tests avancés

2. Mocking des API et du store

```
// __tests__/handlers.js
import { rest } from 'msw'

export const handlers = [
  rest.get('/api/users/:id', (req, res, ctx) => {
    return res(ctx.json({ id: 1, name: 'Ihab' }))
  }),
]
```

```
// __tests__/setupTests.ts
import { server } from './server' // configure MSW
beforeAll(() => server.listen())
afterEach(() => server.resetHandlers())
afterAll(() => server.close())
```

Tests avancés

- **Mock Redux store**

```
import { configureStore } from "@reduxjs/toolkit"
import { Provider } from "react-redux"
import userReducer from "../redux/userSlice"

const mockStore = configureStore({
  reducer: { user: userReducer },
  preloadedState: {
    user: { name: "Marine", isLoggedIn: true },
  },
})
```

Tests avancés

2. Tester les hooks personnalisés

- Exemple de hook custom

```
export function useUserName() {
  const user = useSelector((state) => state.user)
  return user.name.toUpperCase()
}
```

- Test avec `renderHook`

```
import { renderHook } from "@testing-library/react"
import { Provider } from "react-redux"
import { store } from "../redux/store"
import { useUserName } from "../hooks/useUserName"

test("returns uppercased user name", () => {
  const wrapper = ({ children }) => (
    <Provider store={store}>{children}</Provider>
  )
  const { result } = renderHook(() => useUserName(), { wrapper })
  expect(result.current).toBe("MARINE")
})
```

Tests avancés

3. Test d’intégration (Redux + React Query)

```
function UserProfile({ id }) {
```

```
test("renders user profile", async () => {
```

Introduction à GraphQL et ses avantages

GraphQL est un **langage de requête** pour les API, inventé par Facebook. Contrairement à REST, où chaque ressource est exposée via une URL différente, GraphQL propose **une seule URL** et **des requêtes dynamiques**.

REST	GraphQL
Plusieurs endpoints	1 seul endpoint `/graphql`
Sur-fetching (trop)	Juste ce dont le client a besoin
Under-fetching (pas assez)	Possibilité d'inclure tout en une seule requête
Rigidité	Requêtes personnalisables

- REST : `/users/1/posts/3/comments` → 3 appels HTTP
- GraphQL : Une seule requête avec les champs `user`, `posts`, `comments` à la volée

Introduction à GraphQL et ses avantages

1. Installation :

```
npm install @apollo/client graphql
```

2. Configuration :

```
// apolloClient.js
import { ApolloClient, InMemoryCache } from '@apollo/client';
const client = new ApolloClient({
  uri: 'http://localhost:4000/graphql', // endpoint unique
  cache: new InMemoryCache(),
});
export default client;
```

Introduction à GraphQL et ses avantages

3. **Fournir à React :**

```
// main.jsx ou App.jsx
import { ApolloProvider } from '@apollo/client';
import client from './apolloClient';

<ApolloProvider client={client}>
  <App />
</ApolloProvider>
```

Requêtes et mutations avec GraphQL

```
import { gql, useQuery } from '@apollo/client';

const GET_USERS = gql`
  query {
    users {
      id
      name
    }
  }
`;

function UserList() {
  const { data, loading, error } = useQuery(GET_USERS);
  if (loading) return <p>Loading...</p>;
  return data.users.map(u => <div key={u.id}>{u.name}</div>);
}
```

Requêtes et mutations avec GraphQL

```
import { gql, useMutation } from '@apollo/client';

const ADD_USER = gql`
  mutation($name: String!) {
    addUser(name: $name) {
      id
      name
    }
  }
`;

function AddUser() {
  const [addUser] = useMutation(ADD_USER);
  return <button onClick={() => addUser({ variables: { name: 'Jean' } })}>Ajouter</button>;
}
```

Gestion du cache avec Apollo Client

Apollo utilise un cache **normalisé** avec `InMemoryCache`. Il reconnaît les objets par leur `id`.

Après une mutation, Apollo peut mettre à jour le cache **automatiquement**, ou manuellement :

```
const [addUser] = useMutation(ADD_USER, {
  update(cache, { data: { addUser } }) {
    cache.modify({
      fields: {
        users(existingUsers = []) {
          return [...existingUsers, addUser];
        }
      }
    });
  }
});
```

Pagination et chargement infini

1. **Offset-based pagination** :

```
query getPosts($offset: Int!, $limit: Int!) {
  posts(offset: $offset, limit: $limit) {
    id
    title
  }
}
```

Pagination et chargement infini

2. **Cursor-based pagination** (meilleure pour les grands jeux de données) :

```
query getPosts($cursor: ID) {
  posts(after: $cursor) {
    edges {
      node {
        id
        title
      }
    }
    pageInfo {
      endCursor
      hasNextPage
    }
  }
}
```

Pagination et chargement infini

```
const { data, fetchMore } = useQuery(GET_POSTS);

<button onClick={() => fetchMore({ variables: { offset: data.posts.length } })}>
  Charger plus
</button>
```

Gestion des contextes et des providers

React propose le **Context API**, qui permet de créer un **contexte global** accessible par n'importe quel composant de l'arbre, peu importe sa profondeur.

Un **contexte** est composé de deux éléments :

- Un `Provider` : il fournit une valeur à ses composants enfants.
- Un `Consumer` (ou le hook `useContext`) : il permet d'accéder à la valeur du contexte.

Gestion des contextes et des providers

```
// 1. Création du contexte
import { createContext, useState, useContext } from "react";

const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState("light");

  const toggleTheme = () =>
    setTheme((prev) => (prev === "light" ? "dark" : "light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

Gestion des contextes et des providers

```
// 2. Utilisation dans un composant
const ThemeToggler = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);
  return (
    <button onClick={toggleTheme}>
      Current theme: {theme}
    </button>
  );
};

// 3. Wrapper dans l'arbre racine
export default function App() {
  return (
    <ThemeProvider>
      <ThemeToggler />
    </ThemeProvider>
  );
}
```

Gestion des contextes et des providers

- **Cas d’usage :**
- Authentification (UserContext)
- Préférences utilisateur (Theme, langue)
- Données globales (panier, paramètres, notifications)

Utilisation des Portals pour les modales et tooltips

- ReactDOM fournit une méthode appelée `createPortal` :

```
ReactDOM.createPortal(child, container)
```

Cela permet d'injecter un composant **dans un autre élément du DOM**, en dehors de la hiérarchie classique.

Utilisation des Portals pour les modales et tooltips

```
import ReactDOM from "react-dom";

const Modal = ({ children, onClose }) => {
  return ReactDOM.createPortal(
    <div className="modal">
      <div className="overlay" onClick={onClose}></div>
      <div className="content">{children}</div>
    </div>,
    document.getElementById("modal-root") // En dehors de #root
  );
};
```

Utilisation des Portals pour les modales et tooltips

```
// Utilisation
const App = () => {
  const [show, setShow] = useState(false);
  return (
    <>
      <button onClick={() => setShow(true)}>Open Modal</button>
      {show && (
        <Modal onClose={() => setShow(false)}>
          <h2>Je suis une modale</h2>
        </Modal>
      )}
    </>
  );
};
```

Utilisation des Portals pour les modales et tooltips

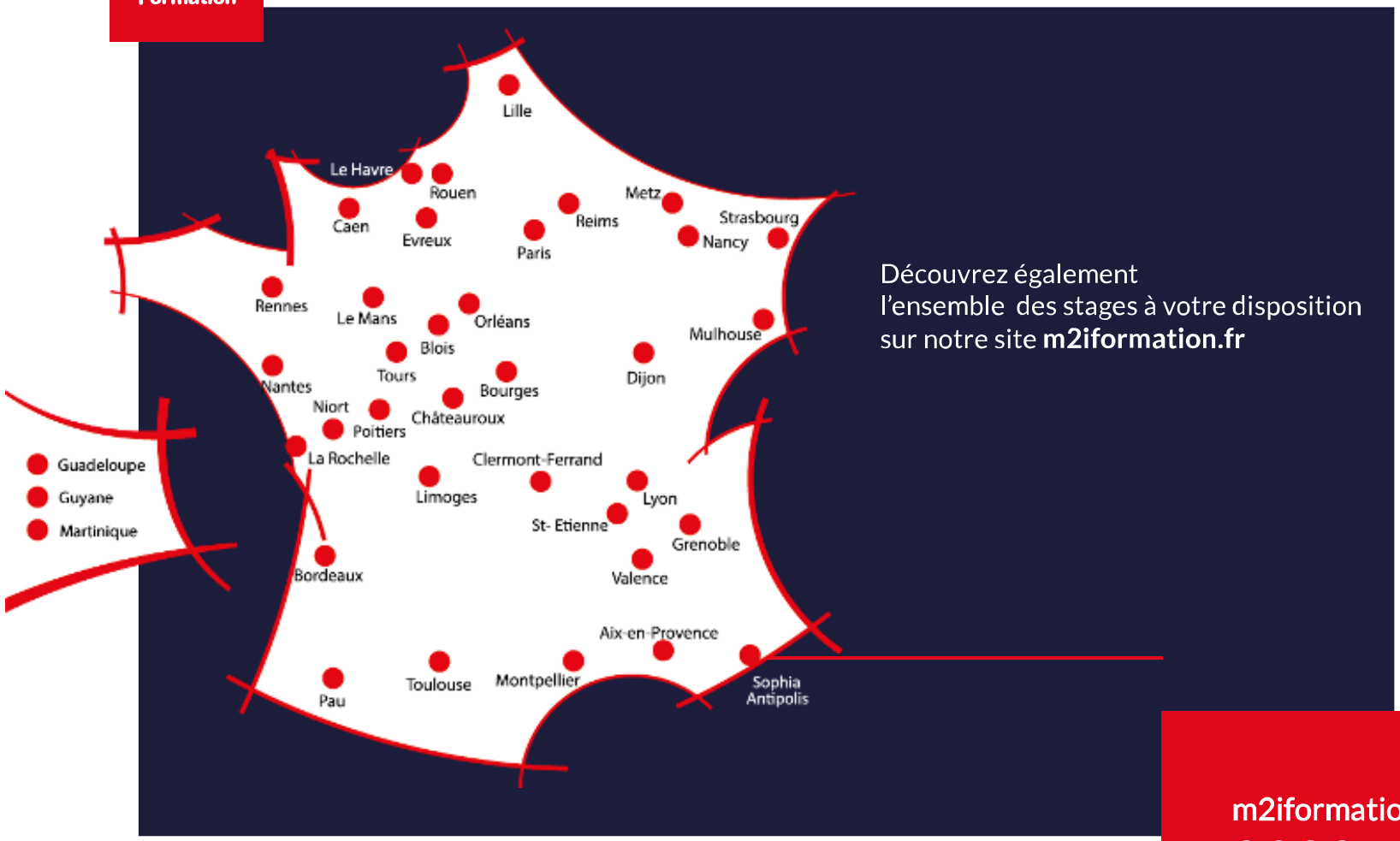
- Dans `index.html`, on aura :

```
<body>
  <div id="root"></div>
  <div id="modal-root"></div>
</body>
```

- **Avantages :**
- Pas d'interférence avec les styles parents
- Placement flexible (modales, tooltips, menus déroulants...)

Gérer les erreurs avec les composants "Error Boundary"

- Empêcher que **tout l'arbre React se casse** à cause d'une erreur JavaScript dans un composant enfant.
- Un **Error Boundary** est un composant de classe (et non une fonction) qui implémente `componentDidCatch` et `getDerivedStateFromError`.



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr