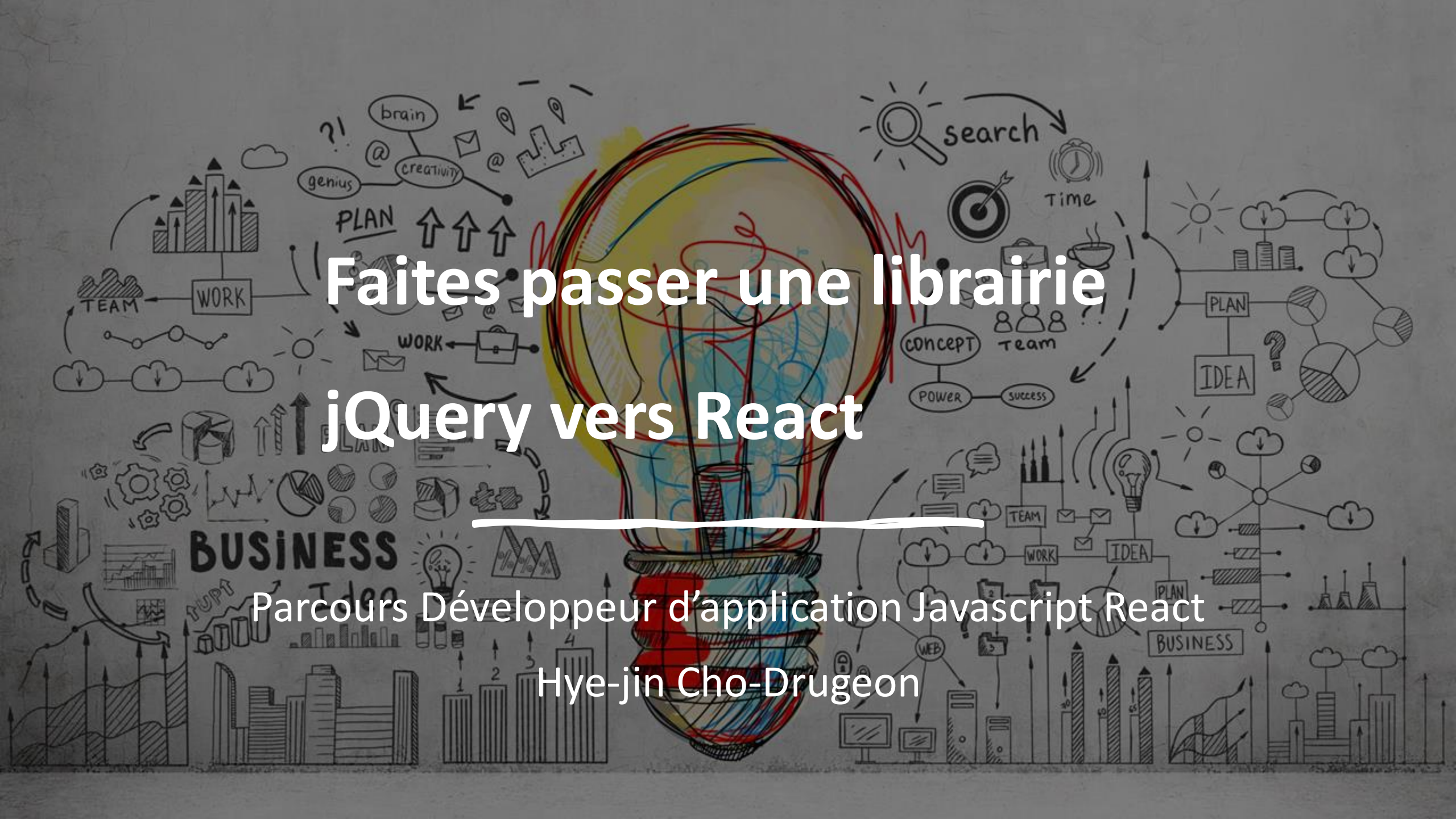


Faites passer une librairie jQuery vers React

Parcours Développeur d'application Javascript React

Hye-jin Cho-Drugeon

Ce que vous allez apprendre

- **Objectif du projet:**

- **HRnet:** Application web interne pour la gestion des employés

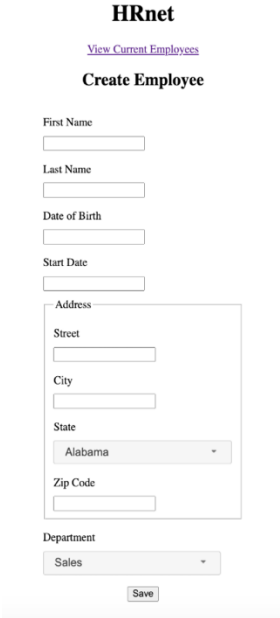
- **Principales fonctionnalités:**
- Création de nouveaux dossiers d'employés
- Affichage de la liste actuelle des employés

- **Défis:** Performances lentes avec les plugins jQuery
- Difficulté de maintenance et d'extension

source
https://github.com/chatlapin/P12_Front-end-HR-Net

2 pages (create, list), Figma X

Page1: create



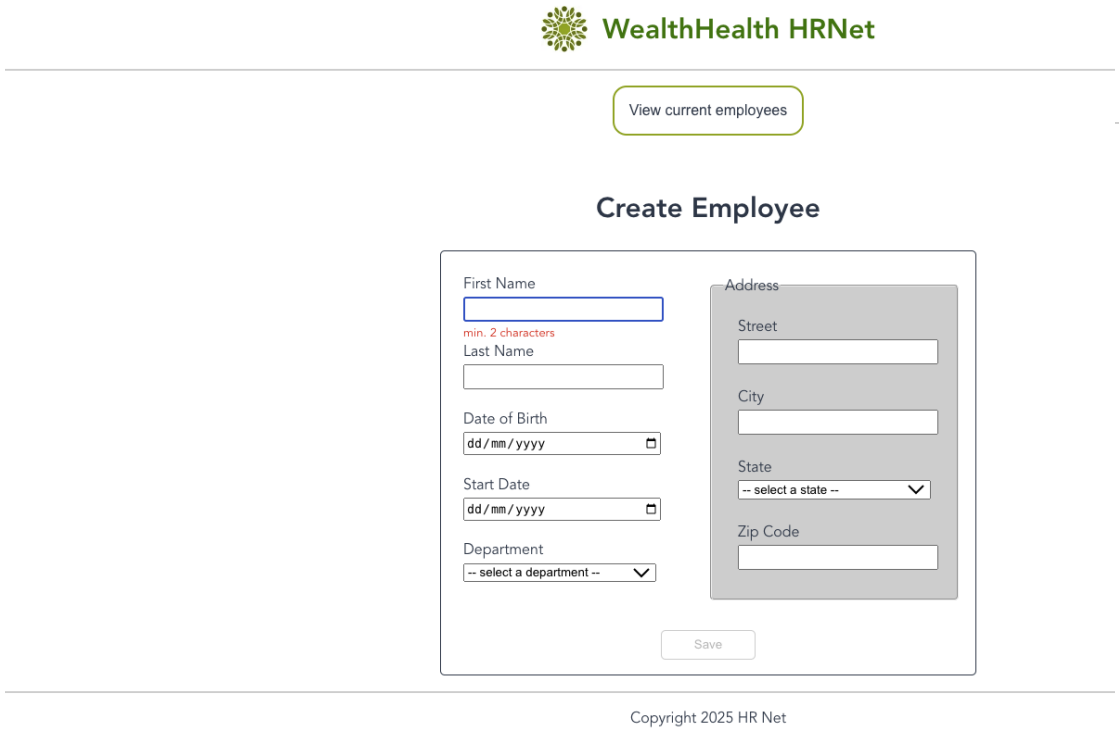
Page2: list

HRnet: De jQuery à React

Amélioration des performances: React est généralement plus performant que jQuery, ce qui devrait conduire à une application plus rapide et plus réactive pour les utilisateurs.

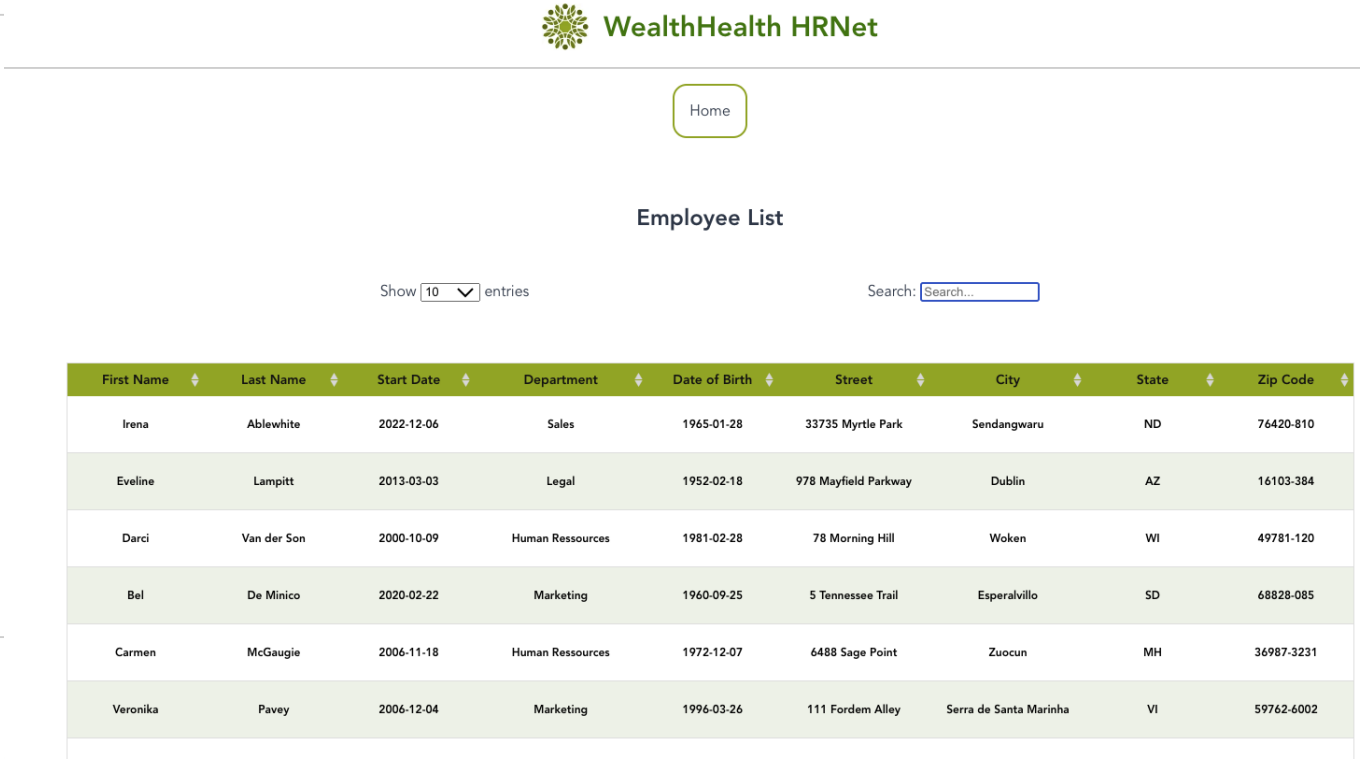
- **Meilleure maintenabilité:** Le code React est souvent considéré comme plus facile à maintenir et à modifier en raison de sa structure composant-basée et de ses outils de développement.
- **Modernisation:** La migration vers React permet d'utiliser les dernières technologies et de profiter des avantages d'un framework moderne.

Output: 2 pages, avec 4 issues



Page1: create

Page2: list



- 4 issues

1. **Date Picker:** issue sur les sélecteurs de date: Converting the date picker to a React component (components/NewEmployeeFrom)
2. Modal: issues de fenêtres modales: Creating a custom modal dialog component (pages/Home.tsx)
3. Dropdown: issue sur menus déroulants: Slow and inconsistent dropdown menus in the Department (components/NewEmployeeForm)
4. Tableaux: issue sur les tableaux: Creating a smart React component for the employee table (pages/Employees.tsx)

https://github.com/OpenClassrooms-Student-Center/P12_Front-end/issues

3 Composants Clés

NewEmployee, Table, Modal

// File: NewEmployeeForm.tsx

Utilise les hooks React (useState, useDispatch): Tout d'abord, observons l'utilisation des hooks useState et useDispatch. Le hook useState nous permet de gérer l'état local du composant, ici pour stocker les données saisies dans le formulaire. Le hook useDispatch, quant à lui, nous offre un moyen d'envoyer des actions à notre store Redux afin de mettre à jour l'état global de notre application.

Validation de formulaire avec des règles personnalisées: Pour assurer la qualité des données saisies, nous avons mis en place une validation personnalisée. Cette validation, définie dans formValidationRules, vérifie que tous les champs du formulaire sont correctement remplis avant d'envoyer les données.

Envoie les nouvelles données d'employé au store Redux

Une fois les données validées, nous utilisons useDispatch pour envoyer une action addNewEmployee à notre store Redux. Cette action va déclencher une mise à jour de notre état global, ajoutant ainsi le nouvel employé à la liste des employés.

```tsx
import stateList from '../../data/state.json'
import departmentList from '../../data/department.json'
import { useState } from 'react'
import { useDispatch } from 'react-redux'
import { addNewEmployee } from '../../app/employeesSlice'
import { formValidationRules } from '../../utils/formValidation'

type FormData = {…
}

interface IProps {…
}

const initialState: FormData = {…
}

function NewEmployeeForm({ setModalOpen }: IProps) {
    const [formData, setFormData] = useState<FormData>(initialState)
    const [currentDate] = useState(new Date().toISOString().slice(0, 10))
    const dispatch = useDispatch()
    const canSave: boolean = Object.values(formData).every(Boolean)

    const handleB
    }
    const handleC
    }

    const handleS
        e.prevent
        dispatch(addNewEmployee(formData))
        const form = document.querySelector('form')
        form?.reset()
        setFormData(initialState)
        setModalOpen(true)
    }

    return (…
    )
}

export default NewEmployeeForm
```

```
(alias) addNewEmployee(payload: any): {
    payload: any;
    type: "employees/addNewEmployee";
}
import addNewEmployee

Calling this redux#ActionCreator with an argument will return a PayloadAction of type T with a payload of P
```

3 Composants Clés

NewEmployee, Table, Modal

// File: Employees.tsx

Un tableau personnalisable Ce composant nous offre une grande flexibilité pour personnaliser l'affichage de nos données. Nous pouvons définir les colonnes que nous souhaitons afficher, ainsi que leurs titres et les propriétés à afficher. Par exemple, dans notre cas, nous avons des colonnes pour le prénom, le nom, la date de naissance, etc.

Le tri, un atout pour l'analyse Une des fonctionnalités les plus intéressantes de cette bibliothèque est la possibilité de trier les données en cliquant sur l'en-tête d'une colonne. Cela permet d'ordonner les employés par ordre alphabétique, par date d'embauche, ou par tout autre critère que nous jugeons pertinent.

La recherche, pour gagner du temps Pour faciliter la recherche d'un employé en particulier, nous avons également implémenté une fonction de recherche. En saisissant une chaîne de caractères dans le champ de recherche, le tableau se mettra à jour en temps réel pour n'afficher que les lignes contenant le texte recherché.

Intégration avec Redux Pour récupérer les données des employés, nous utilisons le hook useSelector de Redux. Ce hook nous permet de sélectionner une partie de l'état de notre application et de la rendre disponible dans notre composant. Dans notre cas, nous récupérons la liste complète des employés depuis notre store Redux.

En résumé Ce composant de table, combiné avec les fonctionnalités de tri et de recherche, nous permet de créer des interfaces utilisateur intuitives et efficaces pour afficher et manipuler de grandes quantités de données. C'est un outil précieux pour les développeurs React qui souhaitent construire des applications de gestion de données.

Front > src > pages > Employees.tsx > Employees

```tsx
import { useSelector } from 'react-redux'
import { getEmployees } from '../app/employeesSlice'
import Table from 'chatlapin-simpletable'
import { useNavigate } from 'react-router-dom'

export type EmployeeType = {…
}
export type Column = {…
}

function Employees() {
    const columns: Column[] = [
        { label: 'First Name', accessor: 'firstName', sortable: true },
        { label: 'Last Name', accessor: 'lastName', sortable: true },
        { label: 'Start Date', accessor: 'startDate', sortable: true },
        { label: 'Department', accessor: 'department', sortable: true },
        { label: 'Date of Birth', accessor: 'dateOfBirth', sortable: true },
        { label: 'Street', accessor: 'street', sortable: true },
        { label: 'City', accessor: 'city', sortable: true },
        { label: 'State', accessor: 'state', sortable: true },
        { label: 'Zip Code', accessor: 'zipCode', sortable: true },
    ]

    const data = useSelector(getEmployees)
    const navigate = useNavigate()
    // const data = []

    const handleClickHome = (e: React.MouseEvent<HTMLSpanElement>) => {…
    }

    return (
        <main>
            <span id="viewHome" onClick={(e) => handleClickHome(e)}>
                Home
            </span>
            <div className="tableContainer">
                <Table
                    caption={'Employee List'}
                    data={data}
                    columns={columns}
                    showEntries={true}
                    showSearch={true}
                />
            </div>
        </main>
    )
}

export default Employees
```

3 Composants Clés

NewEmployee, Table, Modal

// File: Modal.tsx

Un modal, qu'est-ce que c'est ? Un modal, c'est cette petite fenêtre qui s'ouvre par-dessus votre page web pour afficher un message important, demander une confirmation ou présenter des informations supplémentaires. Il est crucial qu'il soit bien conçu et accessible pour ne pas perturber l'expérience utilisateur.

La réutilisabilité, un gain de temps Notre composant modal est conçu pour être réutilisable dans toute notre application. Il prend en entrée un message, un identifiant unique pour les outils d'assistance (aria-labelledby) et une fonction pour fermer le modal. Grâce à cette approche, nous pouvons l'utiliser dans différentes situations sans avoir à le réécrire à chaque fois.

L'accessibilité, une priorité Pour rendre notre modal accessible, nous avons mis en place plusieurs mécanismes :

- Attribut ARIA : L'attribut aria-labelledby relie le modal à un élément contenant le titre du modal. Cela permet aux lecteurs d'écran de comprendre le contenu de la fenêtre.
- Gestion du clavier : Nous avons implémenté une gestion des événements clavier pour permettre à l'utilisateur de fermer le modal en appuyant sur la touche Échap. Cela est essentiel pour les utilisateurs qui ne peuvent pas utiliser la souris.
- Focus: Nous nous assurons que le focus est automatiquement placé sur le premier élément interactif du modal lorsque celui-ci s'ouvre, et qu'il reste confiné à l'intérieur du modal tant qu'il est ouvert.

Front > src > components > Modal > Modal.tsx > Modal

```
1  > import {…
7    } from 'react'
8
9  > type Modal = {…
13   }
14
15   function Modal({ setModalOpen, message, aria_labelledby }: Modal) {
16       useEffect(() => {
17           function keyListener(e: KeyboardEvent) {
18               handleKey(e)
19           }
20           document.addEventListener('keydown', keyListener)
21
22           return () => document.removeEventListener('keydown', keyListener)
23       })
24
25       const modalRef: RefObject<HTMLDivElement> = createRef()
26
27 >     const handleKey = (e: KeyboardEvent) => {…
37       }
38
39 >     return (…
59       )
60   }
61
62   export default Modal
63
```

Processus de Conversion et Améliorations

Redux, Hook, Typescript

a. Gestion d'état avec Redux

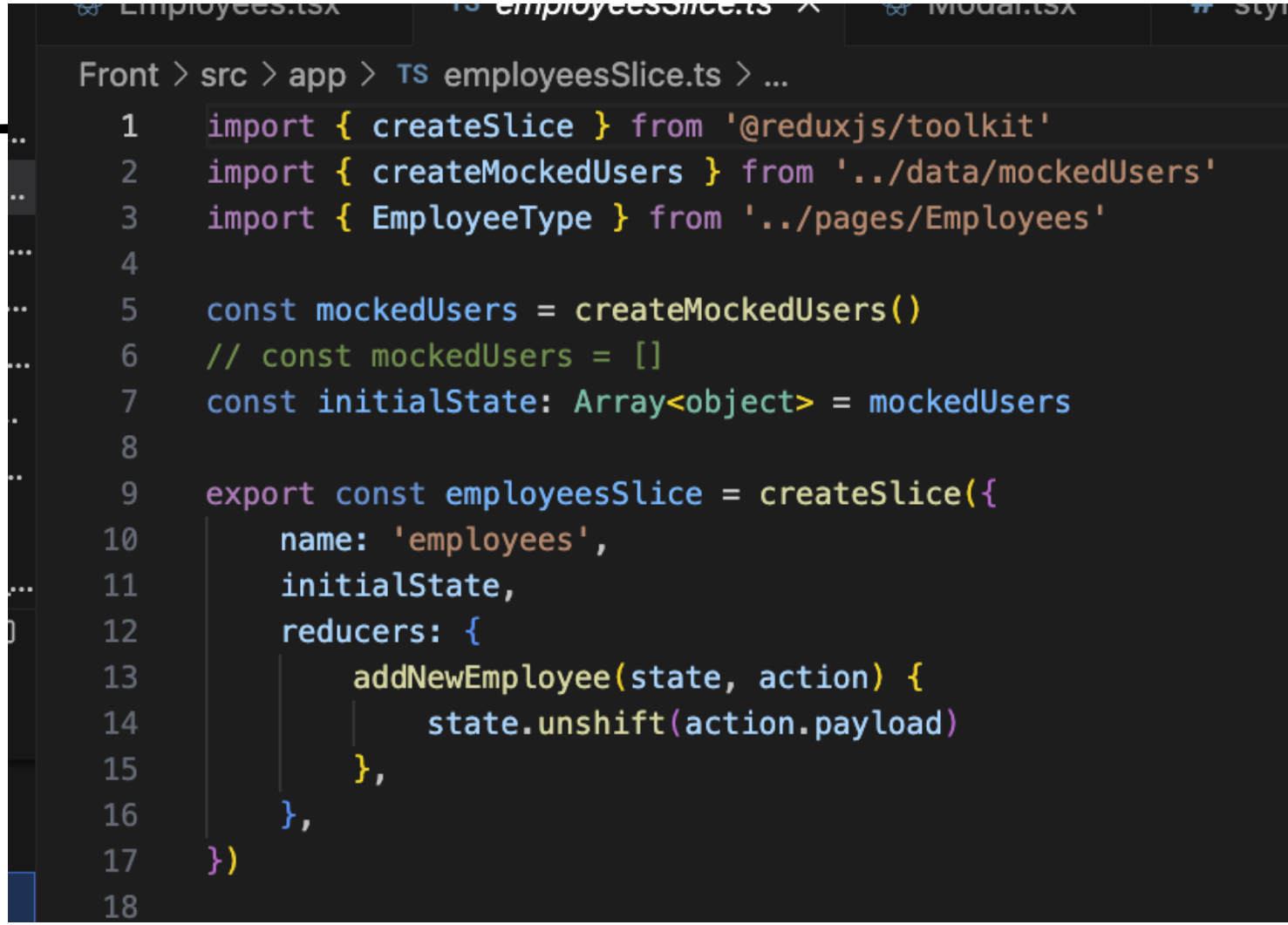
// File: employeesSlice.ts

Dans ce code, nous utilisons Redux pour gérer l'état de notre application.

- Nous définissons un slice nommé 'employees' avec un état initial contenant une liste d'employés.

- Nous créons un reducer 'addNewEmployee' qui permet d'ajouter un nouvel employé au début de la liste.

- Redux nous permet de centraliser la gestion de l'état de notre application, facilitant ainsi la maintenance et le débogage.



Processus de Conversion et Améliorations

Redux, Hook, Typescript

b. Hooks personnalisés pour une meilleure fonctionnalité

Hook personnalisé : Nous introduisons le concept de hook personnalisé et son utilité pour encapsuler de la logique réutilisable.
Fonctionnalité : Nous expliquons brièvement que ce hook sert à gérer le tri de données.
État local : Nous mentionnons l'utilisation de useState pour gérer l'état local du tableau et la fonction de tri.
Flexibilité : Nous soulignons que ce hook peut être réutilisé dans différents composants pour ajouter une fonctionnalité de tri.

// File: useSortableTable.ts

Découvrez la puissance des hooks personnalisés en React !

Ici, nous créons un hook nommé useSortableTable. Il nous permet de gérer le tri d'un tableau d'employés.

En utilisant useState, nous stockons les données du tableau et une fonction pour mettre à jour le tri. Cette fonction handleSorting prend en paramètres le champ à trier et l'ordre de tri (ascendant ou descendant).

Grâce à ce hook, nous pouvons facilement ajouter une fonctionnalité de tri à n'importe quel tableau dans notre application, rendant nos interfaces plus dynamiques et intuitives.

Front > src > utils > TS useSortableTable.ts > useSortableTable

```
1   import { useState } from 'react'
2   import { EmployeeType } from '../pages/Employees'
3
4   export function useSortableTable(
5       data: EmployeeType[]
6   ): [EmployeeType[], (sortField: string, sortOrder: string) => void] {
7       const [tableData, setTableData] = useState(data)
8
9   >   const handleSorting = (sortField: string, sortOrder: string) => {…
28      }
29      return [tableData, handleSorting]
30  }
31
```

Processus de Conversion et Améliorations

Redux, Hook, Typescript

c. Typage sécurisé avec TypeScript

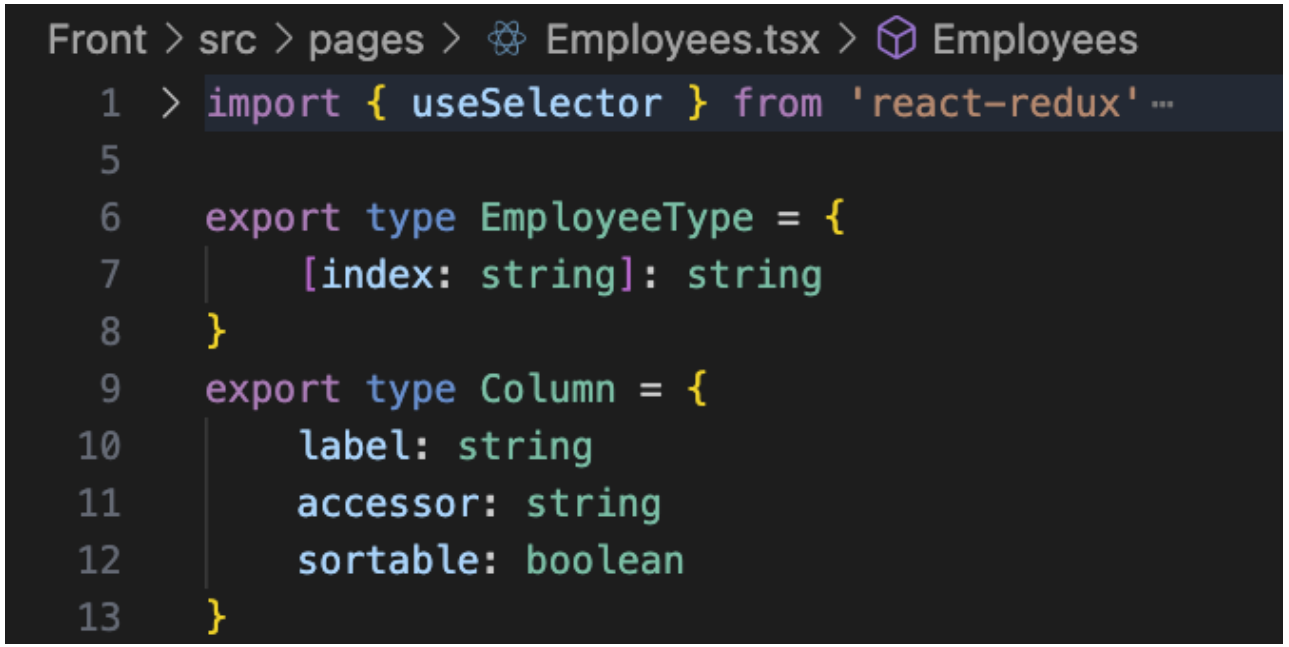
// File: employeesSlice.ts

Dans ce code, nous utilisons TypeScript pour assurer la sécurité de type dans notre application React.

- Nous définissons des types pour nos données, comme EmployeeType et Column.

- Cela permet d'éviter les erreurs de type au moment de la compilation, rendant notre code plus robuste et plus facile à maintenir.

- TypeScript nous aide à écrire du code plus propre et plus prévisible, améliorant ainsi la qualité de notre application.



comparaison des performances pour la version jQuery vs React de HRnet

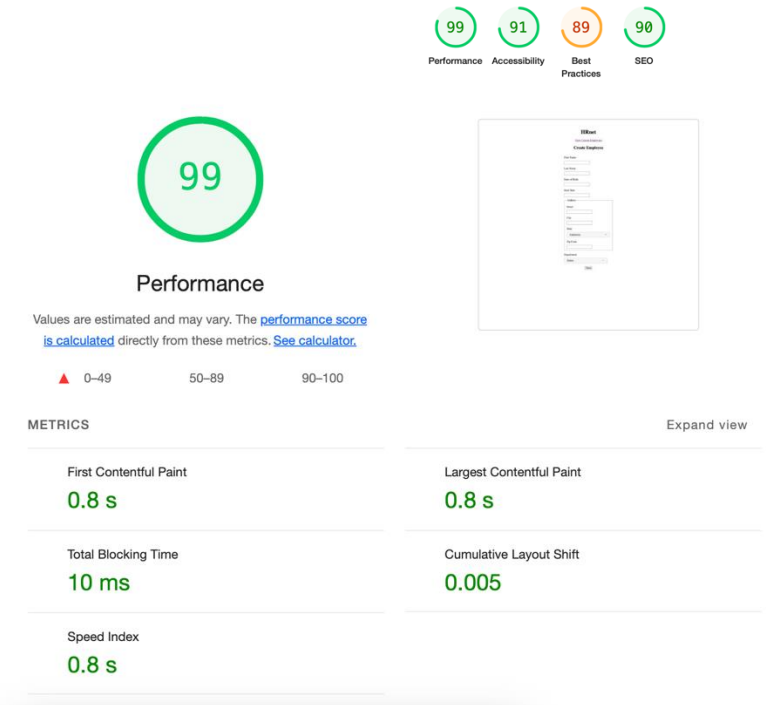
Google Lighthouse

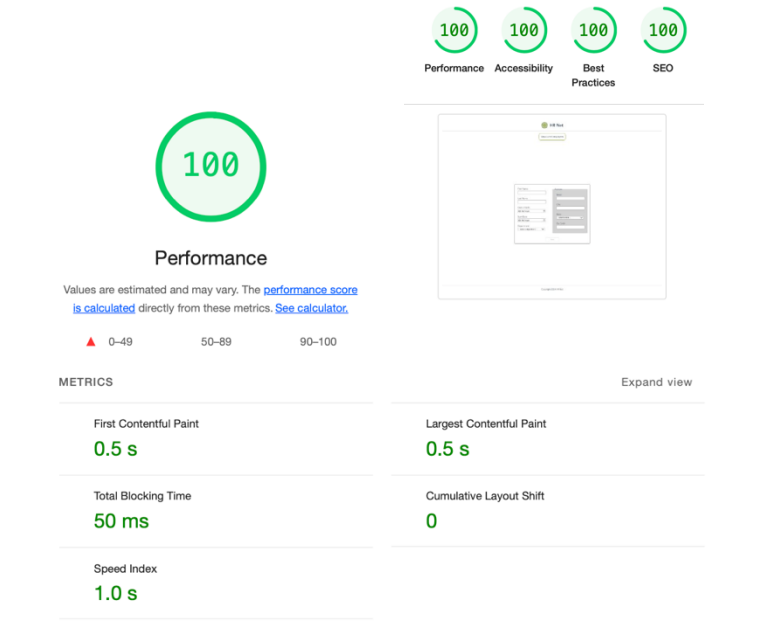
Version jQuery (Original)
- Performance : 99
- First Contentful Paint : 0.8s
- Largest Contentful Paint : 0.8s
- Total Blocking Time : 10ms
- Cumulative Layout Shift : 0.005
- Speed Index : 0.8s

Points critiques :
- Nombreuses requêtes bloquantes (jQuery + plugins)
- JavaScript non minifié (46 KiB d'économies potentielles)
- Pas de compression des textes (60 KiB d'économies potentielles)
- DOM excessif (606 éléments)

Version React (Nouvelle)
- Performance : 100
- First Contentful Paint : 0.5s
- Largest Contentful Paint : 0.5s
- Total Blocking Time : 20ms
- Cumulative Layout Shift : 0.02
- Speed Index : 1.1s

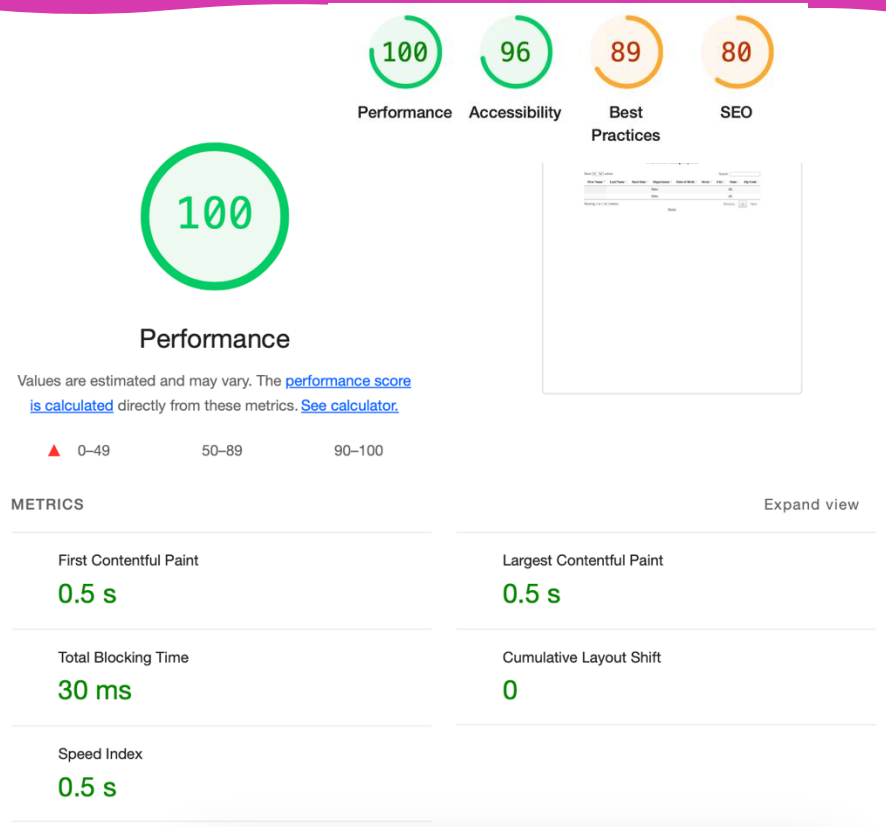
Améliorations notables :
- Meilleure performance globale (score 100 vs 99)
- Temps de chargement initial plus rapide (FCP 0.5s vs 0.8s)
- DOM plus léger (150 éléments vs 606)
- Meilleure accessibilité (score 100 vs 91)
- Meilleures pratiques (score 100 vs 89)

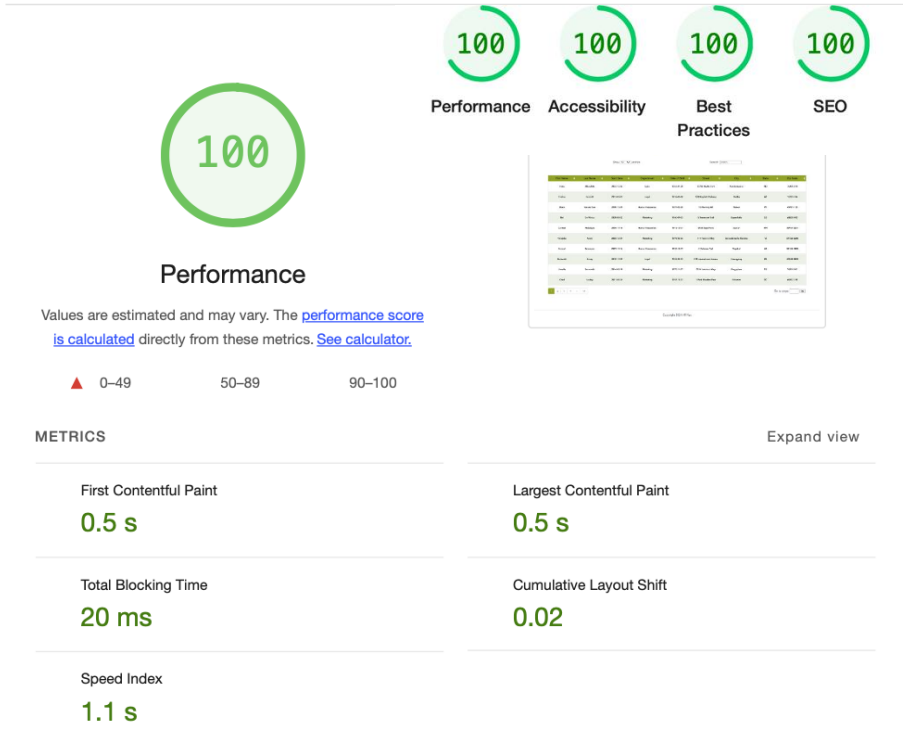


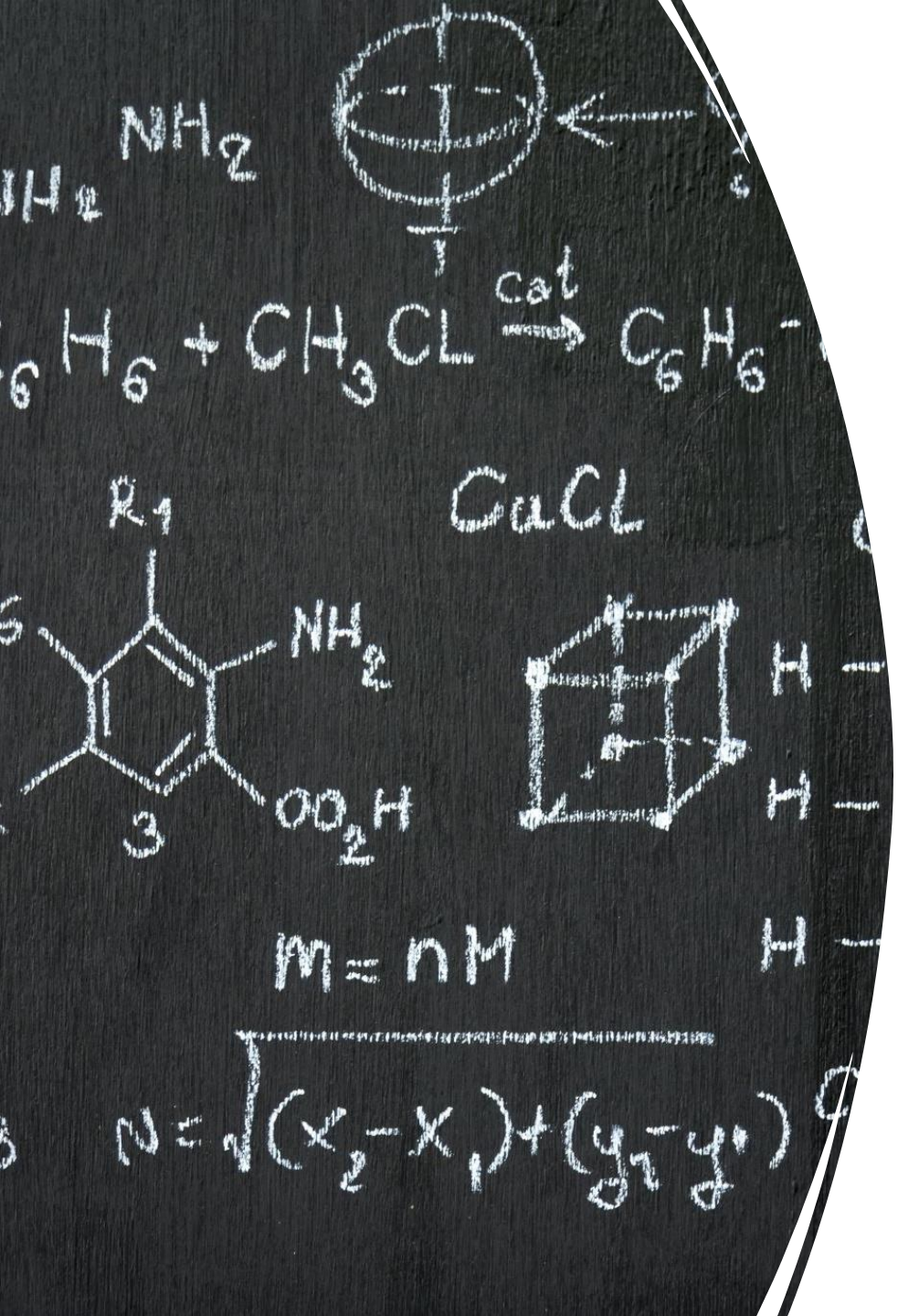


La migration vers React a permis d'améliorer significativement les performances, particulièrement au niveau du chargement initial et de la structure du DOM, tout en améliorant l'accessibilité et les bonnes pratiques de développement.

comparaison des performances pour la version jQuery vs React de HRnet







conclusion,

- Conversion réussie d'HRnet de jQuery à React
- Amélioration des performances et de la maintenabilité
- Amélioration de l'expérience utilisateur avec des composants d'interface utilisateur modernes

Demo