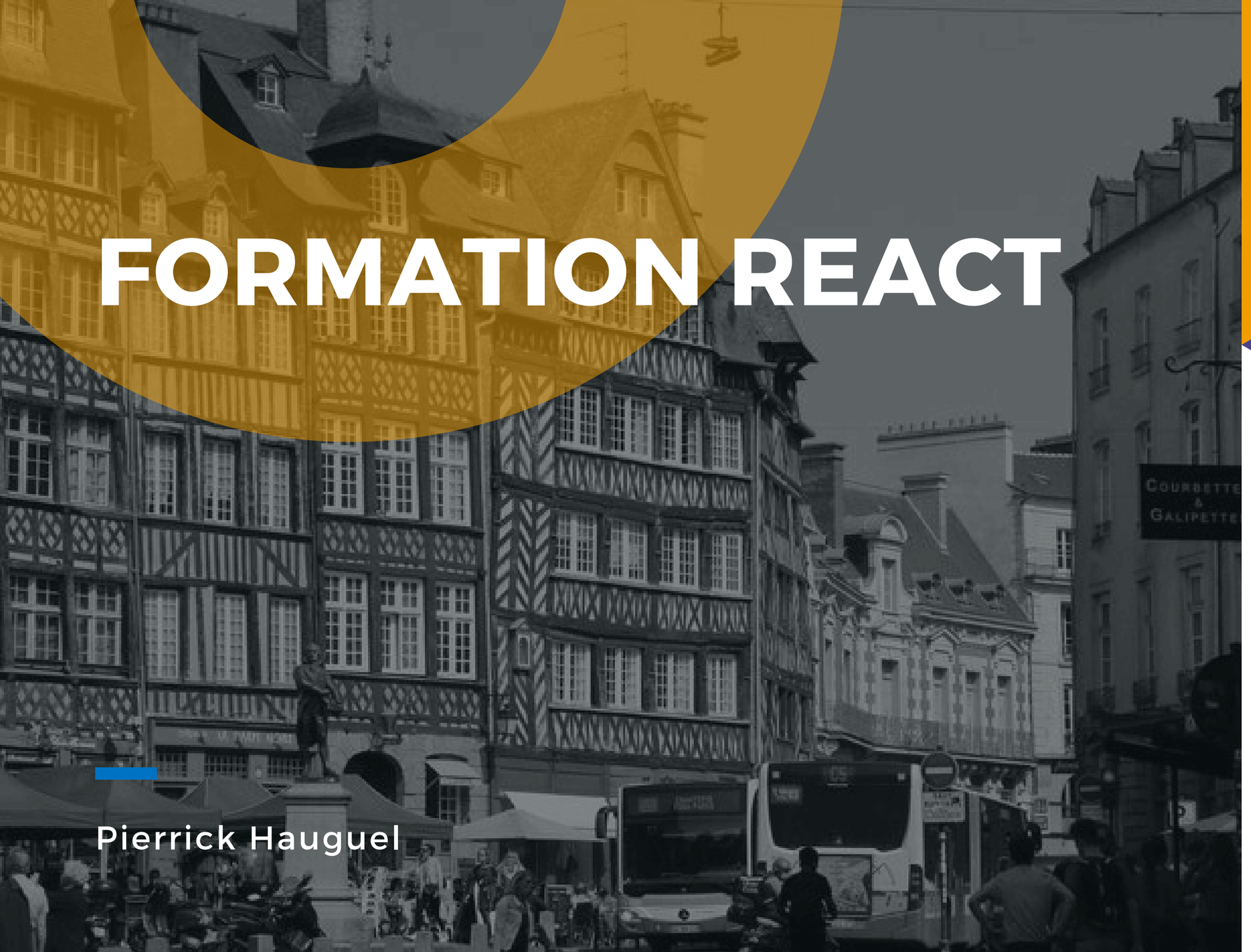
# FORMATION REACT




Pierrick Hauguel

# React Redux Toolkit

```
// store.js
import { configureStore } from "@reduxjs/toolkit";
import rootReducer from "./rootReducer";

const store = configureStore({
  reducer: rootReducer,
});

export default store;
```

# Création d'un Slice

- Organisation du code
  - Chaque slice gère un morceau spécifique de l'état global de l'application
  - Permet de diviser les fonctionnalités en morceaux logiques et maintenables
- Réduction de la complexité
  - Regroupe les actions et les réducteurs en une seule entité
  - Automatise la création d'actions et de réducteurs basés sur les méthodes fournies
- Encapsulation
  - Garde les actions et les réducteurs liés à un domaine spécifique ensemble
  - Facilite la compréhension du code et améliore la lisibilité

# Création d'un Slice

```
// counterSlice.js
import { createSlice } from "@reduxjs/toolkit";

const counterSlice = createSlice({
  name: "counter",
  initialState: 0,
  reducers: {
    increment: (state) => state + 1,
    decrement: (state) => state - 1,
  },
});

export const { increment, decrement } = counterSlice.actions;

export default counterSlice.reducer;
```

# Création du rootReducer

```
// rootReducer.js
import { combineReducers } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";

const rootReducer = combineReducers({
  counter: counterReducer,
});

export default rootReducer;
```

# createAsyncThunk

```
// asyncThunks.js
import { createAsyncThunk } from "@reduxjs/toolkit";
import api from "./api";

export const fetchPosts = createAsyncThunk("posts/fetchPosts", async () => {
  const response = await api.getPosts();
  return response.data;
});
```

# Utilisation de createAsyncThunk avec un slice

```
// postsSlice.js
import { createSlice } from "@reduxjs/toolkit";
import { fetchPosts } from "./asyncThunks";

const postsSlice = createSlice({
  name: "posts",
  initialState: { posts: [], status: "idle", error: null },
  extraReducers: (builder) => {
    builder
      .addCase(fetchPosts.pending, (state) => {
        state.status = "loading";
      })
      .addCase(fetchPosts.fulfilled, (state, action) => {
        state.status = "succeeded";
        state.posts = action.payload;
      })
      .addCase(fetchPosts.rejected, (state, action) => {
        state.status = "failed";
        state.error = action.error.message;
      });
  },
});

export default postsSlice.reducer;
```

# createAsyncThunk

- Gestion simplifiée des actions asynchrones
- Crée automatiquement trois actions pour chaque étape d'une action asynchrone : pending, fulfilled et rejected
- Gère automatiquement l'état de chargement, de réussite et d'échec
- Intégration facile avec Redux Toolkit
- Peut être utilisé directement avec extraReducers dans un slice
- Gère les erreurs et propage les informations d'erreur dans l'état du store
- Réduction de la complexité du code
- Encapsule la logique asynchrone et la gestion des erreurs
- Permet de se concentrer sur la logique métier plutôt que sur la gestion des états de chargement et des erreurs

# Introduction à Material-UI v5 (Mui)

# Les Grids

```jsx
import React from "react";
import { Grid, Paper } from "@mui/material";

const GridExample = () => {
  return (
    <Grid container spacing={2}>
      <Grid item xs={12} sm={6} md={4}>
        <Paper>Item 1</Paper>
      </Grid>
      <Grid item xs={12} sm={6} md={4}>
        <Paper>Item 2</Paper>
      </Grid>
      <Grid item xs={12} sm={6} md={4}>
        <Paper>Item 3</Paper>
      </Grid>
    </Grid>
  );
};

export default GridExample;
```

# Création d'un thème custom

```
// theme.js
import { createTheme } from "@mui/material";

const theme = createTheme({
  palette: {
    primary: {
      main: "#1976d2",
    },
    secondary: {
      main: "#dc004e",
    },
  },
  typography: {
    fontFamily: "'Roboto', 'Helvetica', 'Arial', sans-serif",
    fontSize: 14,
  },
  // autres options de personnalisation...
});

export default theme;
```

# Création d'un thème custom

```jsx
// App.js
import React from "react";
import { ThemeProvider } from "@mui/material";
import theme from "./theme";

const App = () => {
  return (
    <ThemeProvider theme={theme}>
      {/* Votre application */}
    </ThemeProvider>
  );
};

export default App;
```

# Création d'un thème custom

```jsx
// App.js
import React, { useState } from "react";
import { ThemeProvider } from "@mui/material";
import { lightTheme, darkTheme } from "./theme";
import MyComponent from "./MyComponent";

const App = () => {
  const [isDarkMode, setIsDarkMode] = useState(false);

  const handleThemeToggle = () => {
    setIsDarkMode(!isDarkMode);
  };

  return (
    <ThemeProvider theme={isDarkMode ? darkTheme : lightTheme}>
      <MyComponent onThemeToggle={handleThemeToggle} />
    </ThemeProvider>
  );
};

export default App;
```

# styled

La fonction styled est une fonction d'ordre supérieur (HOF) qui prend un argument et retourne une nouvelle fonction. L'argument peut être une chaîne représentant un élément HTML (comme "button" dans cet exemple) ou un composant React existant. La fonction retournée attend un objet de style ou une fonction qui retourne un objet de style.

```
import { styled } from "@mui/system";

const CustomButton = styled("button")(({ theme }) => ({
  backgroundColor: theme.palette.primary.main,
  color: theme.palette.primary.contrastText,
  padding: theme.spacing(1, 3),
  borderRadius: theme.shape.borderRadius,
  "&:hover": {
    backgroundColor: theme.palette.primary.dark,
  },
}));
```

# styled

```
// CustomCard.js
import React from "react";

const CustomCard = ({ children, className }) => {
  return <div className={`custom-card ${className}`}>{children}</div>;
};

export default CustomCard;

// StyledCustomCard.js
import { styled } from "@mui/system";
import CustomCard from "./CustomCard";

const StyledCustomCard = styled(CustomCard)(({ theme }) => ({
  border: `1px solid ${theme.palette.primary.main}`,
  borderRadius: theme.shape.borderRadius,
  padding: theme.spacing(2),
  backgroundColor: theme.palette.background.paper,
  boxShadow: theme.shadows[1],
  "&:hover": {
    boxShadow: theme.shadows[3],
    cursor: "pointer",
  },
}));

export default StyledCustomCard;
```

# styled

```
// App.js
import React from "react";
import StyledCustomCard from "./StyledCustomCard";

const App = () => {
  return (
    <div>
      <StyledCustomCard>
        <h3>Titre de la carte</h3>
        <p>Contenu de la carte</p>
      </StyledCustomCard>
    </div>
  );
};

export default App;
```

# Utilisation de la prop sx

```jsx
import React from "react";
import { Button } from "@mui/material";

const MyComponent = () => {
  return (
    <div>
      <Button
        sx={{
          backgroundColor: "primary.main",
          color: "primary.contrastText",
          padding: (theme) => theme.spacing(1, 3),
          borderRadius: (theme) => theme.shape.borderRadius,
          "&:hover": {
            backgroundColor: "primary.dark",
          },
        }}
      >
        Cliquez-moi
      </Button>
    </div>
  );
};

export default MyComponent;
```

# useStyles

```
// useStyles.js
import { makeStyles } from "@mui/styles";

const useStyles = makeStyles((theme) => ({
  customButton: {
    borderRadius: "50%",
    padding: theme.spacing(1),
    "&:hover": {
      backgroundColor: theme.palette.secondary.main,
    },
  },
}));

export default useStyles;
```

# useStyles

```
// useStyles.js
import { makeStyles } from "@mui/styles";

const useStyles = makeStyles((theme) => ({
  customButton: {
    borderRadius: "50%",
    padding: theme.spacing(1),
    "&:hover": {
      backgroundColor: theme.palette.secondary.main,
    },
  },
}));

export default useStyles;
```

# useStyles

```jsx
// MyComponent.js
import React from "react";
import { Button } from "@mui/material";
import useStyles from "./useStyles";

const MyComponent = () => {
  const classes = useStyles();

  return (
    <div>
      <Button className={classes.customButton}>Cliquez-moi</Button>
    </div>
  );
};

export default MyComponent;
```

# Utiliser Typescript dans React

# Types de base (string, number, boolean, etc.)

```
let myString: string = 'Hello World';
let myNumber: number = 42;
let myBoolean: boolean = true;
```

# Type any et unknown

```
let myAny: any = 'Hello';
myAny = 42;
myAny = true;

let myUnknown: unknown = 'Hello';
// myUnknown = 42; // Error: Type 'number' is not assignable to type 'string'
```

# Interfaces et déclaration de types d'objets

```typescript
interface User {
  id: number;
  name: string;
  age?: number; // Optional property
}

const user: User = {
  id: 1,
  name: 'John Doe',
};
```

# Interfaces et déclaration de types d'objets

```tsx
import React from 'react';

interface Props {
  message: string;
}

// Functional component
const MyComponent: React.FC<Props> = ({ message }) => {
  return <div>{message}</div>;
};
```

# Gestion des événements avec TypeScript

```tsx
const handleClick = (event: React.MouseEvent<HTMLButtonElement>) => {
  // ...
};

<button onClick={handleClick}>Click me</button>
```

# hooks

```tsx
const [count, setCount] = useState<number>(0);
```

```tsx
interface ThemeContext {
  theme: string;
  setTheme: (theme: string) => void;
}

const MyComponent = () => {
  const { theme, setTheme } = useContext<ThemeContext>(ThemeContext);
  // ...
};
```

```tsx
interface State {
  count: number;
}

interface Action {
  type: 'increment' | 'decrement';
}

const reducer = (state: State, action: Action): State => {
  // ...
};

const [state, dispatch] = useReducer(reducer, { count: 0 });
```

# Configuration de Redux avec TypeScript

```ts
// store.ts
import { createStore } from 'redux';
import rootReducer from './reducers';

export type RootState = ReturnType<typeof rootReducer>;

const store = createStore(rootReducer);

export default store;
```

# Typage des actions et des reducers

```ts
// actions.ts
interface IncrementAction {
  type: 'INCREMENT';
}

export const increment = (): IncrementAction => {
  return { type: 'INCREMENT' };
};

// reducer.ts
type ActionTypes = IncrementAction; // Add more action types as needed

const counterReducer = (state = 0, action: ActionTypes): number => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    default:
      return state;
  }
};
```

# Utilisation de Redux Toolkit avec TypeScript

```ts
// slice.ts
import { createSlice, PayloadAction } from '@reduxjs/toolkit';

interface CounterState {
  value: number;
}

const initialState: CounterState = {
  value: 0,
};

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    incrementByAmount: (state, action: PayloadAction<number>) => {
      state.value += action.payload;
    },
  },
});

export const { increment, incrementByAmount } = counterSlice.actions;

export default counterSlice.reducer;
```

# Typage de lib

```
npm install --save-dev @types/react-router-dom
npm install --save-dev @types/material-ui
```

```ts
// custom.d.ts
declare module 'my-custom-library' {
  export function myCustomFunction(): string;
}
```

# Type guards et discriminated unions

Les type guards permettent de vérifier dynamiquement le type d'une variable à l'exécution. Les discriminated unions combinent plusieurs types en un seul type, en utilisant une propriété discriminante pour distinguer entre les différents types.

```ts
interface Dog {
  type: 'dog';
  bark: () => void;
}

interface Cat {
  type: 'cat';
  meow: () => void;
}

type Animal = Dog | Cat;

const makeSound = (animal: Animal): void => {
  if (animal.type === 'dog') {
    animal.bark();
  } else {
    animal.meow();
  }
};
```

# Jest

Jest est un framework de tests JavaScript populaire qui facilite l'écriture de tests simples, efficaces et faciles à comprendre. Pour écrire un test Jest, vous devez créer un fichier avec une extension .test.js ou .spec.js. Voici un exemple de test Jest :

```
// exemple.test.js
const sum = require('./exemple.js');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

# Jest

La fonction test est utilisée pour définir un test. Elle prend deux arguments : une chaîne de caractères qui décrit le test et une fonction qui contient le code du test. Dans cet exemple, le test vérifie que la fonction sum renvoie le résultat attendu pour deux nombres donnés. La fonction expect est utilisée pour définir l'expression à tester et les matchers Jest comme toBe sont utilisés pour vérifier que le résultat est correct.

# Jest

Jest fournit une variété de matchers pour vérifier les expressions et les résultats de test. Voici quelques exemples de matchers courants :

- toBe : vérifie que deux valeurs sont strictement égales.
- toEqual : vérifie que deux valeurs sont égales en valeur, même si elles sont de types différents.
- toContain : vérifie que l'élément cible est inclus dans un tableau ou une chaîne de caractères.
- toThrow : vérifie qu'une fonction lance une erreur.

# Jest

```
// exemple.test.js
const list = ['apple', 'orange', 'banana'];

test('the list contains banana', () => {
  expect(list).toContain('banana');
});

test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

# Gestion des erreurs et débogage des tests

Lorsque vous exécutez des tests avec Jest, vous pouvez rencontrer des erreurs ou des échecs de test. Pour déboguer un test qui échoue, vous pouvez utiliser la commande --watch pour exécuter les tests en mode interactif et voir les résultats en direct. Vous pouvez également ajouter des instructions de débogage à votre code en utilisant la méthode debug de React Testing Library ou des outils de débogage standard tels que console.log.

# Tester les composants React

Pour tester un composant fonctionnel avec Jest, vous pouvez utiliser l'outil React Testing Library. Voici un exemple de test pour un composant fonctionnel :

```
// ExempleComponent.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import ExempleComponent from './ExempleComponent';

test('renders ExempleComponent', () => {
  render(<ExempleComponent />);
  const component = screen.getByTestId('exemple');
  expect(component).toBeInTheDocument();
});
```

# Tester les composants React

Pour tester un composant de classe avec Jest, vous pouvez utiliser l'outil Enzyme. Voici un exemple de test pour un composant de classe :

```
// ExempleComponent.test.js
import React from 'react';
import { shallow } from 'enzyme';
import ExempleComponent from './ExempleComponent';

describe('<ExempleComponent />', () => {
  it('renders a div with the class "exemple"', () => {
    const wrapper = shallow(<ExempleComponent />);
    expect(wrapper.find('div.exemple')).toHaveLength(1);
  });
});
```

# Simulation des événements (click, change, etc.)

Les tests d'événements sont courants dans les tests de composants React. Pour simuler un événement, vous pouvez utiliser la méthode fireEvent de React Testing Library. Voici un exemple de test qui simule un clic sur un bouton :

```
// ExempleComponent.test.js
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import ExempleComponent from './ExempleComponent';

test('clicking the button increments the count', () => {
  const { getByText } = render(<ExempleComponent />);
  const button = getByText('Increment');
  const count = getByText('0');
  fireEvent.click(button);
  expect(count).toHaveTextContent('1');
});
```

# Tests des formulaires et de la validation

```
// ExempleForm.test.js
import React from 'react';
import { render, fireEvent } from '@testing-library/react';
import ExempleForm from './ExempleForm';

test('submitting the form', () => {
  const { getByLabelText, getByText } = render(<ExempleForm />);
  const input = getByLabelText('Name');
  fireEvent.change(input, { target: { value: 'John Doe' } });
  fireEvent.click(getByText('Submit'));
  expect(getByText('Thanks John Doe!')).toBeInTheDocument();
});
```

# Tests de navigation et de routage avec React Router

```
// ExempleRouter.test.js
import React from 'react';
import { render, screen } from '@testing-library/react';
import { Router } from 'react-router-dom';
import { createMemoryHistory } from 'history';
import ExempleRouter from './ExempleRouter';

test('renders correct component for route', () => {
  const history = createMemoryHistory();
  history.push('/example');
  render(
    <Router history={history}>
      <ExempleRouter />
    </Router>
  );
  expect(screen.getByTestId('exemple')).toBeInTheDocument();
});
```

# Tester les Hooks React

```
// useCounter.test.js
import { renderHook, act } from '@testing-library/react-hooks';
import useCounter from './useCounter';

test('should increment counter', () => {
  const { result } = renderHook(() => useCounter());
  act(() => {
    result.current.increment();
  });
  expect(result.current.count).toBe(1);
});
```

# Tester les Hooks React

```
// useFormInput.test.js
import { renderHook, act } from '@testing-library/react-hooks';
import useFormInput from './useFormInput';

test('should update input value', () => {
  const { result } = renderHook(() => useFormInput(''));
  act(() => {
    result.current.onChange({ target: { value: 'test' } });
  });
  expect(result.current.value).toBe('test');
});
```

# Tester les actions et les reducers

```
// todoActions.test.js
import { addTodo } from './todoActions';

test('should create an action to add a todo', () => {
  const text = 'test';
  const expectedAction = {
    type: 'ADD_TODO',
    payload: {
      id: expect.any(Number),
      text,
      completed: false,
    },
  };
  expect(addTodo(text)).toEqual(expectedAction);
});
```

# Tester les composants connectés

```
// TodoList.test.js
import React from 'react';
import { shallow } from 'enzyme';
import configureStore from 'redux-mock-store';
import TodoList from './TodoList';

const mockStore = configureStore([]);

test('should render TodoList component', () => {
  const store = mockStore({
    todos: [
      { id: 1, text: 'test 1', completed: false },
      { id: 2, text: 'test 2', completed: true },
    ],
  });
  const wrapper = shallow(<TodoList store={store} />);
  expect(wrapper.exists()).toBe(true);
});
```

# Tester les sélecteurs et les middlewares

```
// todoSelectors.test.js
import { getVisibleTodos } from './todoSelectors';

test('should return visible todos', () => {
  const todos = [
    { id: 1, text: 'test 1', completed: false },
    { id: 2, text: 'test 2', completed: true },
  ];
  const filter = 'SHOW_COMPLETED';
  const expectedVisibleTodos = [{ id: 2, text: 'test 2', completed: true }];
  expect(getVisibleTodos(todos, filter)).toEqual(expectedVisibleTodos);
});
```

# Optimisations

# Éviter les fonctions anonymes dans les props

- Les fonctions anonymes créent de nouvelles références à chaque rendu
- Provoquent des rendus inutiles pour les composants enfants

```
// Mauvaise pratique
<button onClick={() => handleClick(id)}>Cliquez-moi</button>

// Bonne pratique
<button onClick={handleClick}>Cliquez-moi</button>
```

# Utiliser React.memo pour les composants fonctionnels

```
const MyComponent = React.memo(function MyComponent(props) {
  // Le composant
});
```

# useMemo

Le but du useMemo est de mémoriser un résultat de calcul entre les appels d'une fonction et entre les rendus

# Mise en place du useMemo

```
const plusFive = (num: number) => {
    console.log('i was called!');
    return num + 5;
};

const Memo = () => {
    const [num, setNum] = useState(0);
    const [light, setLight] = useState(true);
    const numPlusFive = useMemo(() => plusFive(num), [num]);
    return (
        <div className={light ? 'light' : 'dark'}>
            <div>
                <h1>With useMemo</h1>
                <h2>
                    Current number: {num}, Plus five: {numPlusFive}
                </h2>
                <div className="button-container">
                    <button
                        onClick={() => {
                            setNum(num + 1);
                        }}
                    >
                        Update Number{' '}
                    </button>
                    <button
                        onClick={() => {
                            setLight(!light);
                        }}
                    >
                        {' '}
                        Toggle the light{' '}
                    </button>
                </div>
            </div>
        </div>
    );
};
```

# useCallback

```tsx
type DefaultFunction = { someFunc: () => number };

const SomeComp = ({ someFunc }: DefaultFunction) => {
    const [calcNum, setCalcNum] = useState(0);
    useEffect(() => {
        setCalcNum(someFunc());
    }, [someFunc]);

    return <span> Plus five: {calcNum}</span>;
};

const Callback = () => {
    const [num, setNum] = useState(0);
    const [light, setLight] = useState(true);
    const plusFive = useCallback(() => {
        console.log('I was called!');
        return num + 5;
    }, [num]);
    return (
        <div className={light ? 'light' : 'dark'}>
            <div>
                <h1>With useCallback</h1>
                <h2>
                    Current number: {num},
                    <SomeComp someFunc={plusFive} />
                </h2>
                <div className="button-container">
                    <button
                        onClick={() => {
                            setNum(num + 1);
                        }}
                    >
                        Update Number{' '}
                    </button>
                    <button
                        onClick={() => {
                            setLight(!light);
                        }}
                    >
                        {' '}
                        Toggle the light{' '}
                    </button>
                </div>
            </div>
        </div>
    );
```

# Rendre de larges listes :

```
import { List } from 'react-virtualized';

function MyList({ items }) {
  const rowRenderer = ({ index, key, style }) => {
    return (
      <div key={key} style={style}>
        {items[index]}
      </div>
    );
  };

  return (
    <List
      width={300}
      height={400}
      rowCount={items.length}
      rowHeight={30}
      rowRenderer={rowRenderer}
    />
  );
}
```

# Rendre de larges listes :

```
import { FixedSizeList as List } from 'react-window';

function MyList({ items }) {
  const rowRenderer = ({ index, style }) => {
    return (
      <div style={style}>
        {items[index]}
      </div>
    );
  };

  return (
    <List
      height={400}
      itemCount={items.length}
      itemSize={30}
      width={300}
    >
      {rowRenderer}
    </List>
  );
}
```

# Les Portals

Les Portals sont une fonctionnalité de React permettant de rendre un composant dans un emplacement différent de l'arbre de composants parent. Cela peut être utile pour afficher des éléments tels que des modales ou des menus contextuels en dehors de la zone où le composant parent est rendu.

# Modal Portal

```jsx
import React, { useState } from "react";
import ReactDOM from "react-dom";

const Modal = ({ isOpen, onClose, children }) => {
  if (!isOpen) return null;
  return ReactDOM.createPortal(
    <div className="modal">
      <div className="modal-overlay" onClick={onClose} />
      <div className="modal-body">{children}</div>
    </div>,
    document.body
  );
};

const App = () => {
  const [isModalOpen, setIsModalOpen] = useState(false);
  const openModal = () => setIsModalOpen(true);
  const closeModal = () => setIsModalOpen(false);
  return (
    <div>
      <h1>My App</h1>
      <button onClick={openModal}>Open Modal</button>
      <Modal isOpen={isModalOpen} onClose={closeModal}>
        <h2>Modal Content</h2>
        <p>This is the content of the modal.</p>
      </Modal>
    </div>
  );
};

ReactDOM.render(<App />, document.getElementById("root"));
```