React
JS

Support by Pierrick Hauguel

01

Training presentation

First Day

- **Introduction and Review of Best Practices in React**
- - Source Organization
- - Use of hooks
- - Optimizations: memoization, virtual DOM
- - ErrorBoundary
- - ESLint rules
- - Strict mode

- **Advanced Patterns in React**
- - Combining hooks
- - Using useEffect and useContext to trigger actions
- - Pattern of functions as children

- **Introduction to TanStack Query**
- - Fundamentals of TanStack Query
- - Management of asynchronous data states
- - Caching, refetching, and optimizations
- - Integration with React

Second Day

- **Introduction to Zustand**
- - Basic principles of Zustand
- - Creation and management of simplified global states
- - Using Zustand in React components
- - Optimizations and best practices

- **Improving Application Performance**
- - Using React Dev Tools
- - Concurrent mode and Server Side Rendering (introduction)
- - Splitting code

Third Day

- **Advanced Testing in React**
- - Testing hooks
- - Testing components using hooks
- - Asynchronous tests
- - Advanced mocks

- **New Features in React 19**
- - New hooks
- - Changes and breaking changes
- - Review of new features

More topics ?

- Complex forms ?
- Good practices with types ?
- How do you manage complex local data ? useState ? Something else ?
- How to override the TS namespace of React? Which usecase ?

02

The Virtual DOM

How many object allocations for each rendering?

```jsx
import React, { useState } from 'react';

function MyComponent() {
  const [user, setUser] = useState({
    name: 'Alice',
    age: 25,
    address: {
      street: '123 Main Street',
      city: 'Paris',
      country: 'France'
    }
  });

  return (
    <div>
      <h1>User Information</h1>
      <p>Name: {user.name}</p>
      <p>Age: {user.age}</p>
      <p>Address:</p>
      <ul>
        <li>Street: {user.address.street}</li>
        <li>City: {user.address.city}</li>
        <li>Country: {user.address.country}</li>
      </ul>
    </div>
  );
}

export default MyComponent;
```

How React creates objects with every rendering

Each element (such as <div>, <h1>, <p>, etc.) in the JSX code is transformed into an object that React uses to display the content.

In our example :
<div>, <h1>, <p>, <ul>, <li>
→ These tags create objects.

This makes a total of 8 objects created when this component is displayed.

React.createElement - What React does behind JSX

When you write JSX (like <h1>Hello World</h1>), React automatically translates it into :

```
React.createElement('h1', null, 'Hello World');
```

How React creates objects with every rendering

Each element (such as <div>, <h1>, <p>, etc.) in the JSX code is transformed into an object that React uses to display the content.

In our example :
<div>, <h1>, <p>, <ul>, <li>
→ These tags create objects.

This makes a total of 8 objects created when this component is displayed.

Corresponding React.createElement tree :

```
React.createElement(
  'div',
  null,
  React.createElement('h1', null, 'User Information'),
  React.createElement('p', null, `Name: ${user.name}`),
  React.createElement('p', null, `Age: ${user.age}`),
  React.createElement('p', null, 'Address:'),
  React.createElement(
    'ul',
    null,
    React.createElement('li', null, `Street: ${user.address.street}`),
    React.createElement('li', null, `City: ${user.address.city}`),
    React.createElement('li', null, `Country: ${user.address.country}`)
  )
);
```

Diffing algorithm

- The **diffing algorithm** is the method used by React to compare two versions of the Virtual DOM.
- React determines the differences between the old and new Virtual DOM.
- Thanks to this algorithm, React identifies the minimum modifications to be made to the actual DOM.

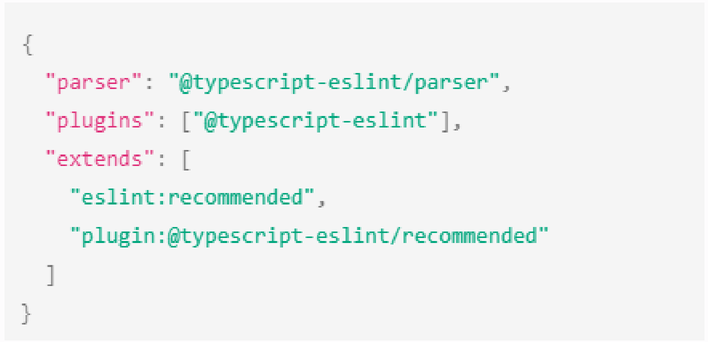
- Step 1: Compare elements on the same level (tree comparison).
- Step 2: If the elements have the same type, React keeps the old element and updates its attributes.
- Step 3: If the elements are of different types, React deletes the old one and creates a new one.
- Step 4: For children, React recursively scans the structure for changes.

03

ESLINT rules to know

Basic ESLint configuration for TypeScript

Basic configuration in .eslintrc.json :



```json
{
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended"
  ]
}
```

Rule @typescript-eslint/explicit-function-return-type

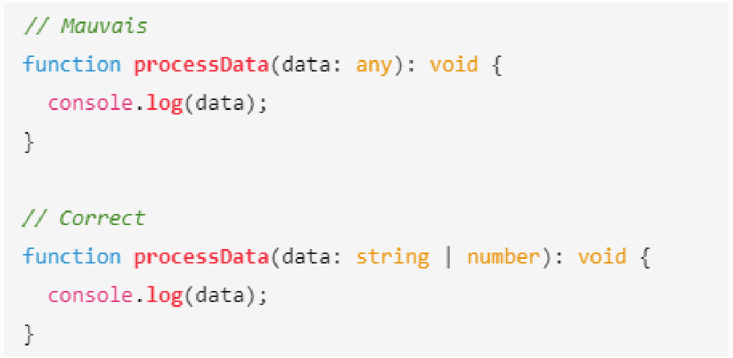
Description: Requires function return type to be defined.
Benefit: Clarifies expectations and avoids unforeseen mistakes.

```
// Mauvais
function add(a: number, b: number) {
  return a + b;
}

// Correct
function add(a: number, b: number): number {
  return a + b;
}
```

Rule @typescript-eslint/no-explicit-any

Description: forbids the use of the any type.

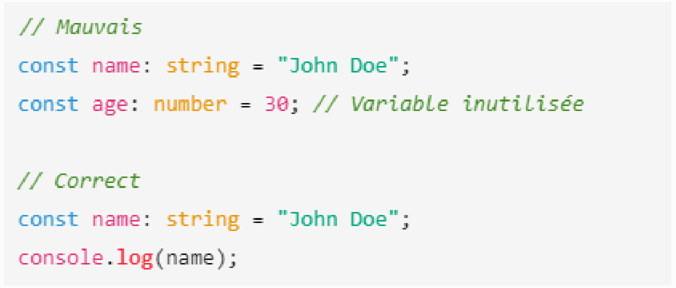
Advantage: encourages the use of specific types to avoid errors at runtime.



```
// Mauvais
function processData(data: any): void {
  console.log(data);
}

// Correct
function processData(data: string | number): void {
  console.log(data);
}
```

Rule @typescript-eslint/no-unused-vars

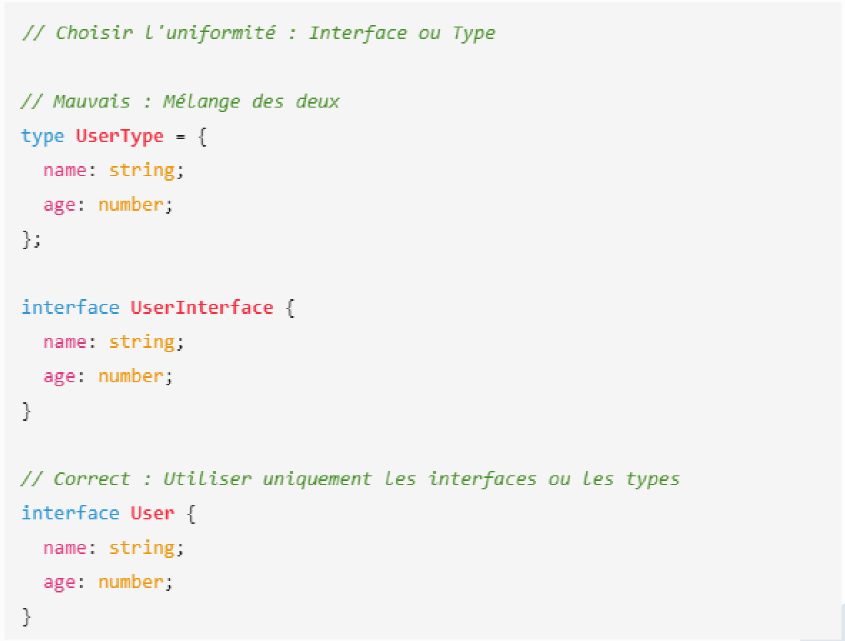
Description: prevents the declaration of unused variables.
Benefit: Helps keep code clean and avoids confusion.



```typescript
// Mauvais
const name: string = "John Doe";
const age: number = 30; // Variable inutilisée

// Correct
const name: string = "John Doe";
console.log(name);
```

Rule @typescript-eslint/consistent-type-definitions

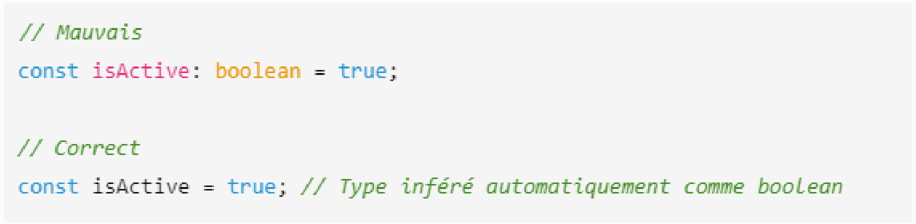
Description: Forces the use of either interface or type for object types.
Advantage: Ensures consistency in type definition.



```
// Choisir l'uniformité : Interface ou Type

// Mauvais : Mélange des deux
type UserType = {
  name: string;
  age: number;
};

interface UserInterface {
  name: string;
  age: number;
}

// Correct : Utiliser uniquement les interfaces ou les types
interface User {
  name: string;
  age: number;
}
```

Rule @typescript-eslint/no-inferrable-types

Description: avoids redundancy in declarations of types that can be inferred.
Benefit: Simplifies code and improves readability.



```
// Mauvais
const isActive: boolean = true;

// Correct
const isActive = true; // Type inféré automatiquement comme boolean
```

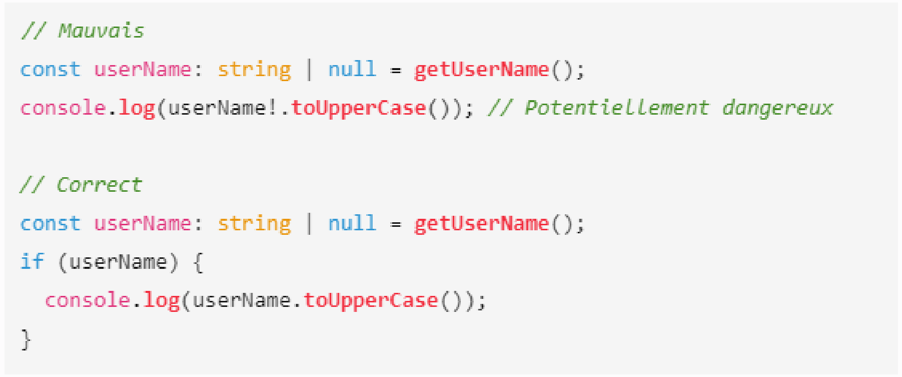
Rule @typescript-eslint/explicit-module-boundary-types

Description: Requires specification of argument and return types for exported functions.
Benefit: Improves maintainability of public APIs.

```
// Mauvais
export function fetchData(url) {
  return fetch(url).then(res => res.json());
}

// Correct
export function fetchData(url: string): Promise<any> {
  return fetch(url).then(res => res.json());
}
```

Rule @typescript-eslint/no-non-null-assertion

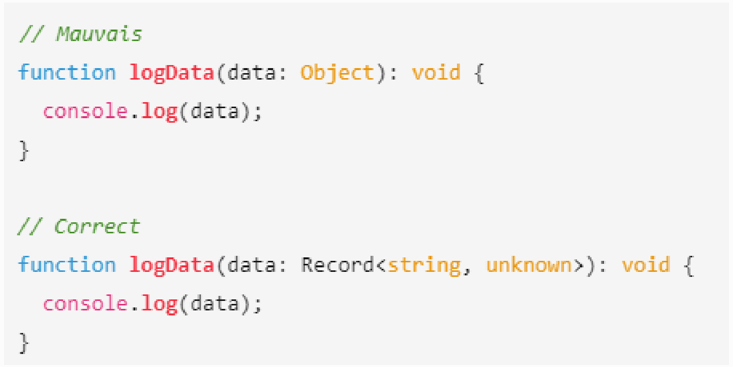
Description: prohibits the use of the ! operator for non-null assertions.
Advantage: Avoids errors associated with zero or undefined values.



```
// Mauvais
const userName: string | null = getUserName();
console.log(userName!.toUpperCase()); // Potentiellement dangereux

// Correct
const userName: string | null = getUserName();
if (userName) {
  console.log(userName.toUpperCase());
}
```

Rule @typescript-eslint/ban-types

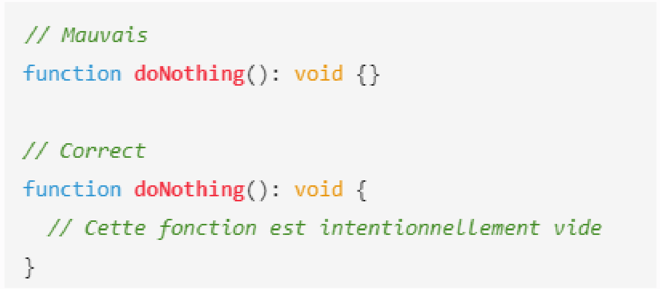
Description: Prohibits the use of certain types such as Object, Function, etc.
Advantage: Encourages the use of more specific, safer types.



```
// Mauvais
function logData(data: Object): void {
  console.log(data);
}

// Correct
function logData(data: Record<string, unknown>): void {
  console.log(data);
}
```

@typescript-eslint/no-empty-function

Description: forbids empty functions without comments explaining why they are empty.
Benefit: Encourages more expressive and useful code.



```
// Mauvais
function doNothing(): void {}

// Correct
function doNothing(): void {
  // Cette fonction est intentionnellement vide
}
```

List of all rules

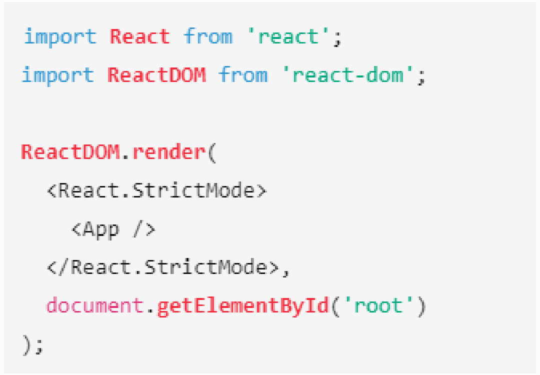
```
// @typescript-eslint/explicit-function-return-type
// @typescript-eslint/no-explicit-any
// @typescript-eslint/no-unused-vars
// @typescript-eslint/consistent-type-definitions
// @typescript-eslint/no-inferrable-types
// @typescript-eslint/explicit-module-boundary-types
// @typescript-eslint/no-non-null-assertion
// @typescript-eslint/ban-types
// @typescript-eslint/no-empty-function
```

04

Le Strict Mode

What is Strict Mode?

Definition: Strict Mode is a tool for detecting potential problems in a React application.
Activation: It is activated via the <React.StrictMode> component.
Objective: Help developers identify performance problems, insecure usage and obsolete practices.



```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Duplicate returns: Why?

Side-effect detection: Double rendering helps identify unexpected side-effects. For example, if a mutative operation is performed in a component or hook, double rendering can reveal unexpected behavior.

Idempotency test: Ensure that components have no visible side effects and are "idempotent" (produce the same result every time they are run).

Debugging status updates: Duplicate rendering helps ensure that status updates are correctly handled without introducing bugs invisible in production mode.

Warnings and Depreciation

Deprecated APIs: Strict Mode issues warnings for React methods that will soon be deprecated. This helps developers anticipate future API changes.

Legacy String Refs and findDOMNode: It displays warnings when these old practices are used, as they will be removed in future versions of React.

How does Strict Mode help detect bugs?

Warning on unsafe effects: Strict Mode can reveal errors such as unsafe effects (e.g. changing state after an edge effect) that might not be obvious without double rendering.

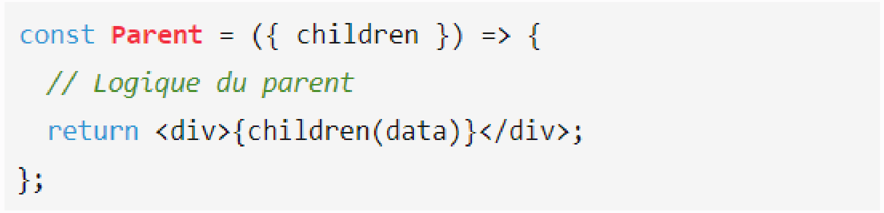
Example of a subtle bug: For example, an asynchronous function that is not properly cleaned up can lead to errors that Strict Mode will highlight.

05
functions as children

How does it work?

The parent component :
Accepts a function as a child via props.children.

The child component :
The child function is invoked with parameters from the parent.



```
const Parent = ({ children }) => {
  // Logique du parent
  return <div>{children(data)}</div>;
};
```

Code example - Parent component

The parent defines data and calls children with this data.

```
const ParentComponent = ({ children }) => {
  const data = "Parent's data";

  return (
    <div>
      {children(data)}
    </div>
  );
};
```

```
// App.jsx
<ParentComponent>
  {(dataFromParent) => (
    <div>Data received: {dataFromParent}</div>
  )}
</ParentComponent>
```

Example 1: Asynchronous loading with data feedback

In an application that interacts with an API or makes asynchronous network calls, this model can be used to manage loading status and condition data display flexibly.

```jsx
// DataFetcher.jsx
const DataFetcher = ({ children }) => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    fetch("https://api.example.com/data")
      .then((response) => response.json())
      .then((result) => {
        setData(result);
        setLoading(false);
      });
  }, []);

  return <div>{children({ data, loading })}</div>;
};
```

```jsx
// App.jsx
<DataFetcher>
  {({ data, loading }) => (
    <div>
      {loading ? (
        <p>Loading...</p>
      ) : (
        <div>Data fetched: {JSON.stringify(data)}</div>
      )}
    </div>
  )}
</DataFetcher>
```

Example 2: Managing user permissions

Use case: Display action buttons only if the user has the right permissions.

```jsx
// PermissionHandler.jsx
const PermissionHandler = ({ children, userPermissions }) => {
  const hasPermission = (permission) => userPermissions.includes(permission);

  return <div>{children(hasPermission)}</div>;
};
```

```jsx
const userPermissions = ["edit", "view"]; // Permissions de l'utilisateur actuel

<PermissionHandler userPermissions={userPermissions}>
  {(hasPermission) => (
    <div>
      {hasPermission("edit") && <button>Edit</button>}
      {hasPermission("delete") && <button>Delete</button>}
      {!hasPermission("delete") && <p>No permission to delete.</p>}
    </div>
  )}
</PermissionHandler>
```

06 Visit Advanced hooks

The Hook useReducer

useReducer is an alternative to useState, ideal for managing more complex states with action-based local state logic.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

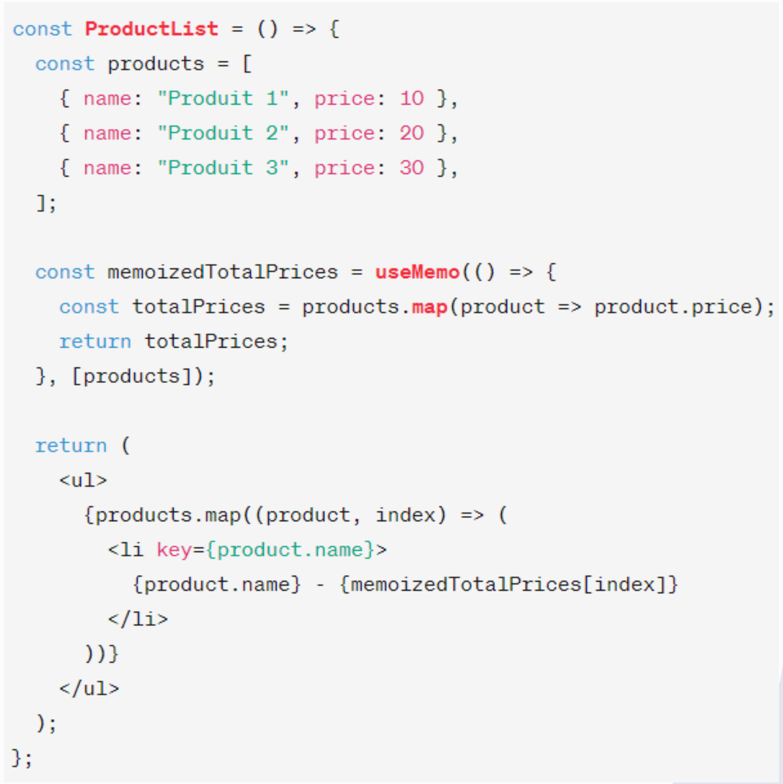
Accessing DOM elements with useRef

useRef keeps a reference mutable across the component's re-renders, often used to access a DOM element directly.

```
const myRef = useRef(initialValue);
```

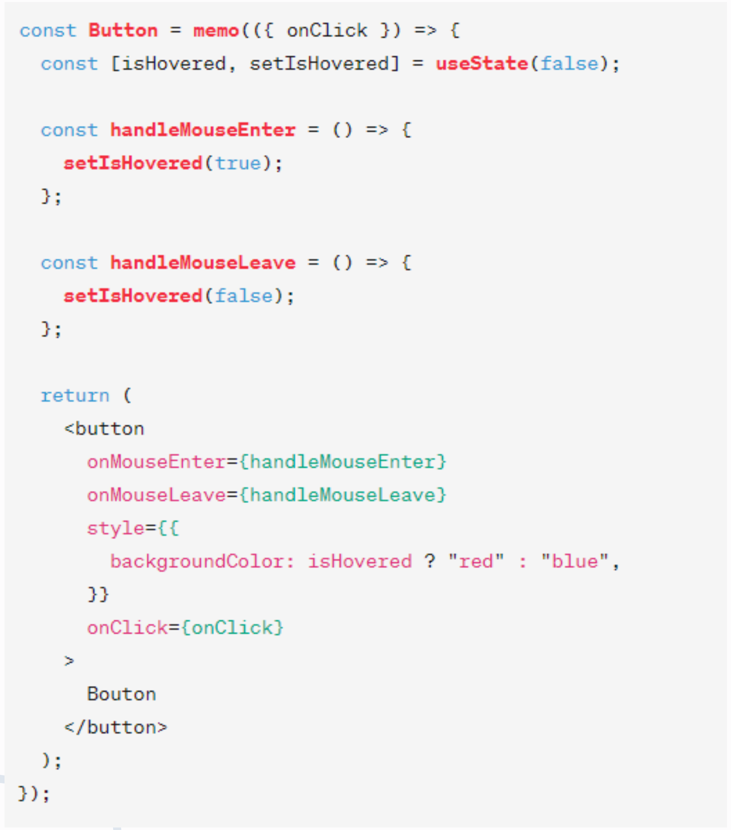
useMemo

useMemo is a hook that stores a calculated value and prevents it from being recalculated each time the component is rendered. This can be useful for values that are expensive to calculate, such as complex functions or API calls.



```
const ProductList = () => {
  const products = [
    { name: "Produit 1", price: 10 },
    { name: "Produit 2", price: 20 },
    { name: "Produit 3", price: 30 },
  ];

  const memoizedTotalPrices = useMemo(() => {
    const totalPrices = products.map(product => product.price);
    return totalPrices;
  }, [products]);

  return (
    <ul>
      {products.map((product, index) => (
        <li key={product.name}>
          {product.name} - {memoizedTotalPrices[index]}
        </li>
      ))}
    </ul>
  );
};
```

memo

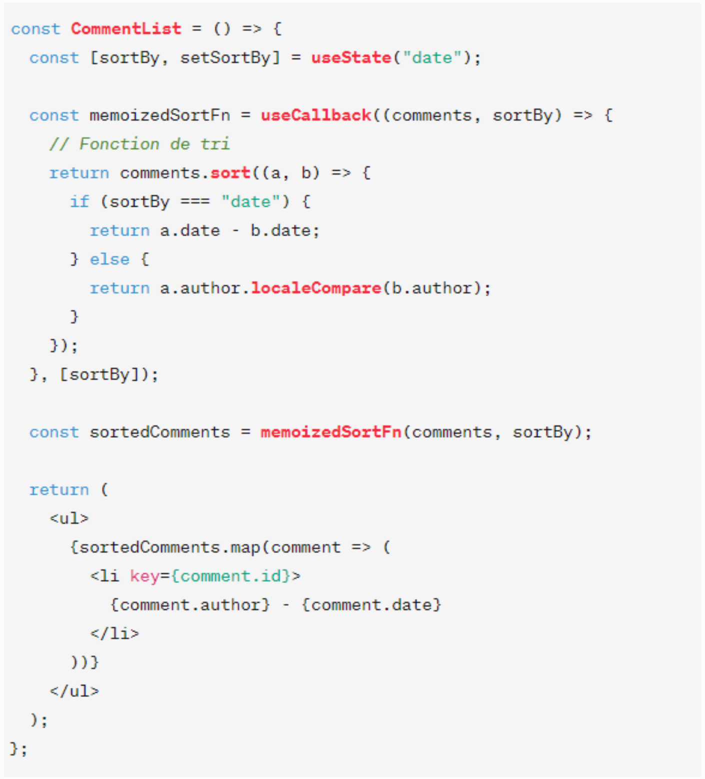


```
const Button = memo(({ onClick }) => {
  const [isHovered, setIsHovered] = useState(false);

  const handleMouseEnter = () => {
    setIsHovered(true);
  };

  const handleMouseLeave = () => {
    setIsHovered(false);
  };

  return (
    <button
      onMouseEnter={handleMouseEnter}
      onMouseLeave={handleMouseLeave}
      style={{
        backgroundColor: isHovered ? "red" : "blue",
      }}
      onClick={onClick}
    >
      Bouton
    </button>
  );
});
```

memo is a function that encapsulates a functional component and prevents it from rendering unnecessarily. This is useful for pure components, which should only surrender if their props or state change.

useCallback

useCallback is a hook that stores a function and prevents it from being recreated each time the component is rendered. This can be useful for callbacks that are passed on to child components, as it avoids recreating them each time the parent component renders.



```
const CommentList = () => {
  const [sortBy, setSortBy] = useState("date");

  const memoizedSortFn = useCallback((comments, sortBy) => {
    // Fonction de tri
    return comments.sort((a, b) => {
      if (sortBy === "date") {
        return a.date - b.date;
      } else {
        return a.author.localeCompare(b.author);
      }
    });
  }, [sortBy]);

  const sortedComments = memoizedSortFn(comments, sortBy);

  return (
    <ul>
      {sortedComments.map(comment => (
        <li key={comment.id}>
          {comment.author} - {comment.date}
        </li>
      ))}
    </ul>
  );
};
```

07

The Context

Introduction to Context in React

Context makes it easy to share values between several components, without having to explicitly pass a prop to each level of the component tree. It is ideal for so-called "global" data such as theme, user preferences, etc.

Context creation and use

Setting up a context begins with React.createContext(), which returns an object containing a Provider and Consumer component. The Provider is used to supply a context value to all child components encapsulating it.

```
const MyContext = React.createContext(defaultValue);
```

Providing Values with the Provider

```
<MyContext.Provider value={/* some value */}>
  {/* child components */}
</MyContext.Provider>
```

The Provider component is used to provide a context value to the component tree, enabling child components to access this data without passing through a prop.

Access Context Values

Context values are accessible to child components via the Consumer component, the useContext hook in functional components, or MyContext.Consumer and static contextType in class components.

```
const value = useContext(MyContext);
```

Best Practices with Context

- **Usage Restriction:** Use context for global data that rarely changes.
- **Context Update:** Optimize updates to avoid unnecessary rendering.
- **Context composition:** Use multiple contexts to separate concerns without excessive prop drilling.

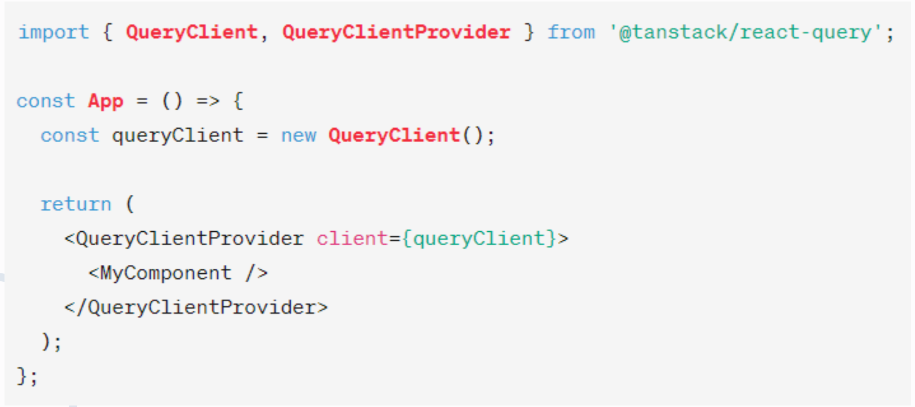
Example

```
const ThemeContext = createContext({
  theme: "light",
  toggleTheme: () => {},
});

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

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === "light" ? "dark" : "light"));
  };

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

const MyComponent = () => {
  const { theme } = useContext(ThemeContext);

  return (
    <div>
      The current theme is: {theme}
    </div>
  );
};
```

08.1

Using Tanstack Query

Getting started with Tanstack Quer

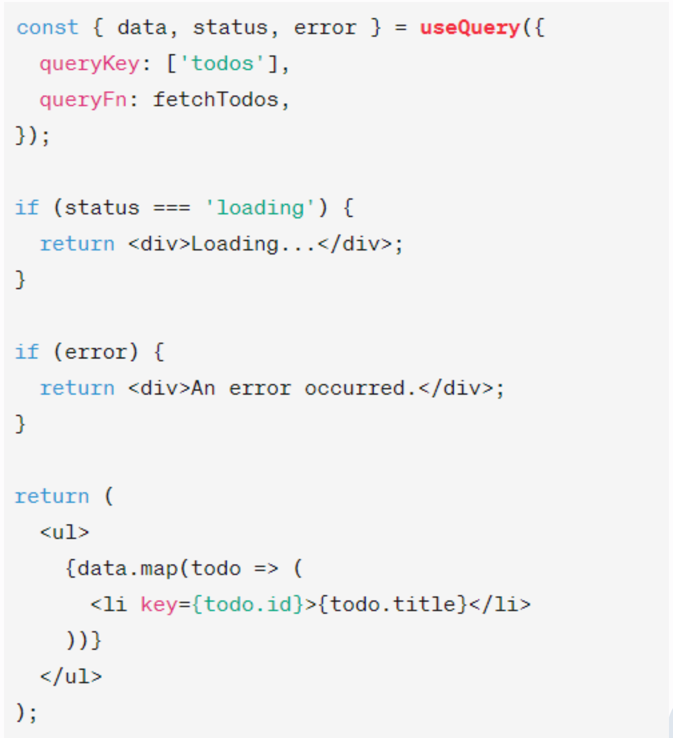


```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const App = () => {
  const queryClient = new QueryClient();

  return (
    <QueryClientProvider client={queryClient}>
      <MyComponent />
    </QueryClientProvider>
  );
};
```

This module guides you through the installation and configuration of Tanstack Query v5 in your React project. You'll learn how to install the necessary packages, configure the library and understand the fundamental concepts.

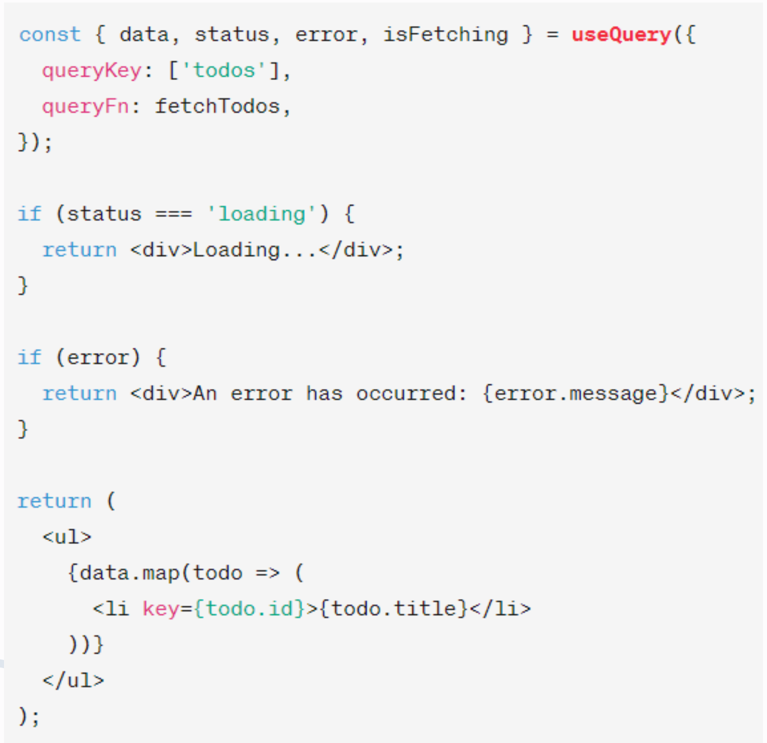
Understanding the key concepts of Tanstack Query

This module explores the fundamental concepts of Tanstack Query, such as queries, mutations, data state, cache invalidation and basic hooks.



```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error occurred.</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    ))}
  </ul>
);
```

Error and load management with Tanstack Query

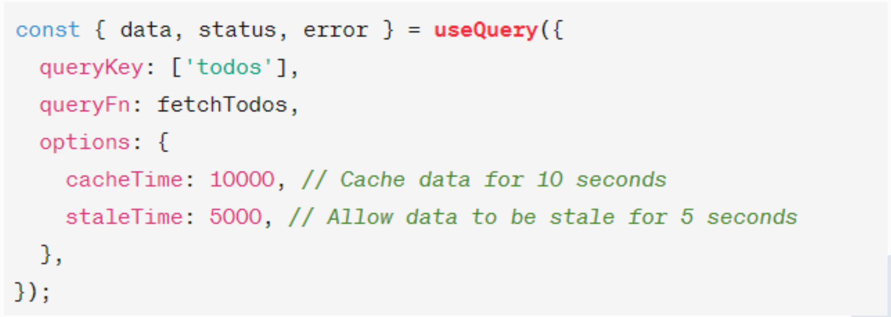


```
const { data, status, error, isFetching } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
});

if (status === 'loading') {
  return <div>Loading...</div>;
}

if (error) {
  return <div>An error has occurred: {error.message}</div>;
}

return (
  <ul>
    {data.map(todo => (
      <li key={todo.id}>{todo.title}</li>
    ))}
  </ul>
);
```

This module shows you how to handle errors and loads in your Tanstack Query applications. You'll learn how to display error messages, manage retries and use load statuses.

Optimize performance with Tanstack Query

This module shows you how to optimize the performance of your Tanstack Query applications. You'll learn how to use the data cache, perform selective queries and set up pagination.



```
const { data, status, error } = useQuery({
  queryKey: ['todos'],
  queryFn: fetchTodos,
  options: {
    cacheTime: 10000, // Cache data for 10 seconds
    staleTime: 5000, // Allow data to be stale for 5 seconds
  },
});
```

Further information

TkDodo: https://twitter.com/TkDodo
Competitor's account: (SWR)
https://twitter.com/huozhi

08.2

Advanced use of Tanstack Query

Introduction to Mutations with Tanstack Query

Mutations are used in Tanstack Query to perform data modifications such as insertions, updates or deletions.

```
import { useMutation } from '@tanstack/react-query';

const createItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItem() {
  const mutation = useMutation(createItem);
  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add Item</button>;
}
```

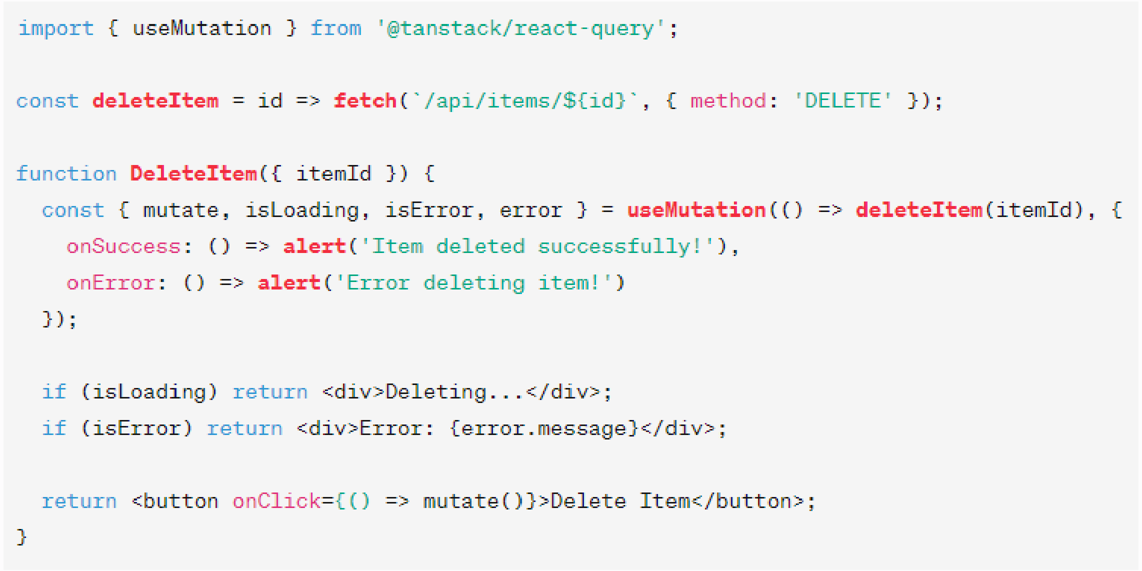
Mutations database

Learn how to useMutation to create, update or delete data.

```
import { useMutation } from '@tanstack/react-query';

const updateItem = async (item) => {
  return fetch(`/api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  }).then(res => res.json());
};

function EditItem({ item }) {
  const { mutate } = useMutation(updateItem, {
    onSuccess: () => alert('Item updated successfully!')
  });

  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
}
```

Transfer status management

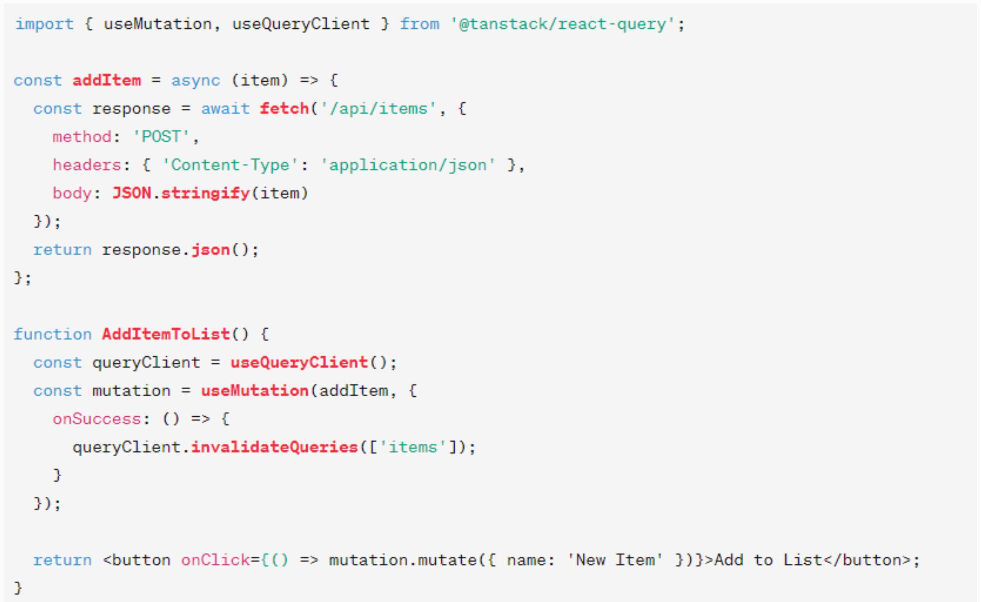
Mutation status management to display appropriate visual feedback such as loads, successes and errors.



```
import { useMutation } from '@tanstack/react-query';

const deleteItem = id => fetch(`/api/items/${id}`, { method: 'DELETE' });

function DeleteItem({ itemId }) {
  const { mutate, isLoading, isError, error } = useMutation(() => deleteItem(itemId), {
    onSuccess: () => alert('Item deleted successfully!'),
    onError: () => alert('Error deleting item!')
  });

  if (isLoading) return <div>Deleting...</div>;
  if (isError) return <div>Error: {error.message}</div>;

  return <button onClick={() => mutate()}>Delete Item</button>;
}
```

Synchronization with the Cache after a Mutation

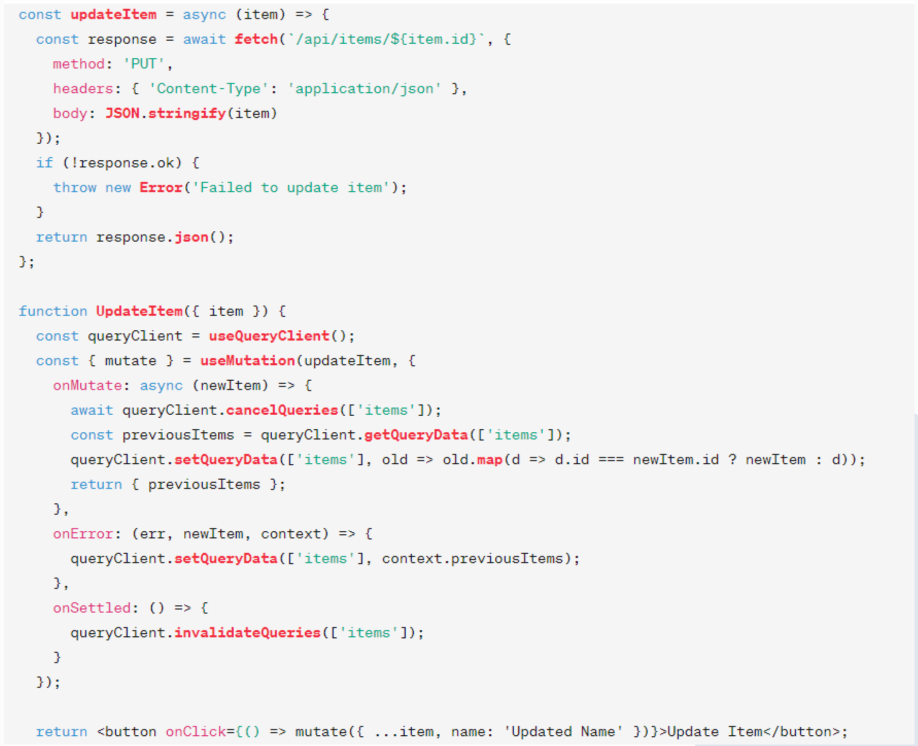
How to use mutations not only to modify data, but also to synchronize these modifications with the local cache.



```javascript
import { useMutation, useQueryClient } from '@tanstack/react-query';

const addItem = async (item) => {
  const response = await fetch('/api/items', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  return response.json();
};

function AddItemToList() {
  const queryClient = useQueryClient();
  const mutation = useMutation(addItem, {
    onSuccess: () => {
      queryClient.invalidateQueries(['items']);
    }
  });

  return <button onClick={() => mutation.mutate({ name: 'New Item' })}>Add to List</button>;
}
```

Using Rollbacks in the event of Failure

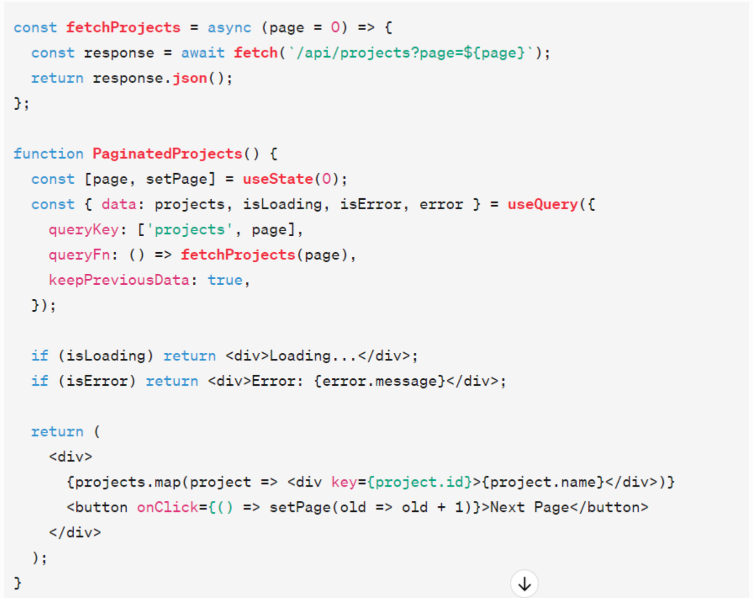
Implement rollbacks to restore the previous state in the event of mutation failure.



```
const updateItem = async (item) => {
  const response = await fetch(`/api/items/${item.id}`, {
    method: 'PUT',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify(item)
  });
  if (!response.ok) {
    throw new Error('Failed to update item');
  }
  return response.json();
};

function UpdateItem({ item }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateItem, {
    onMutate: async (newItem) => {
      await queryClient.cancelQueries(['items']);
      const previousItems = queryClient.getQueryData(['items']);
      queryClient.setQueryData(['items'], old => old.map(d => d.id === newItem.id ? newItem : d));
      return { previousItems };
    },
    onError: (err, newItem, context) => {
      queryClient.setQueryData(['items'], context.previousItems);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['items']);
    }
  });

  return <button onClick={() => mutate({ ...item, name: 'Updated Name' })}>Update Item</button>;
```

Paged queries

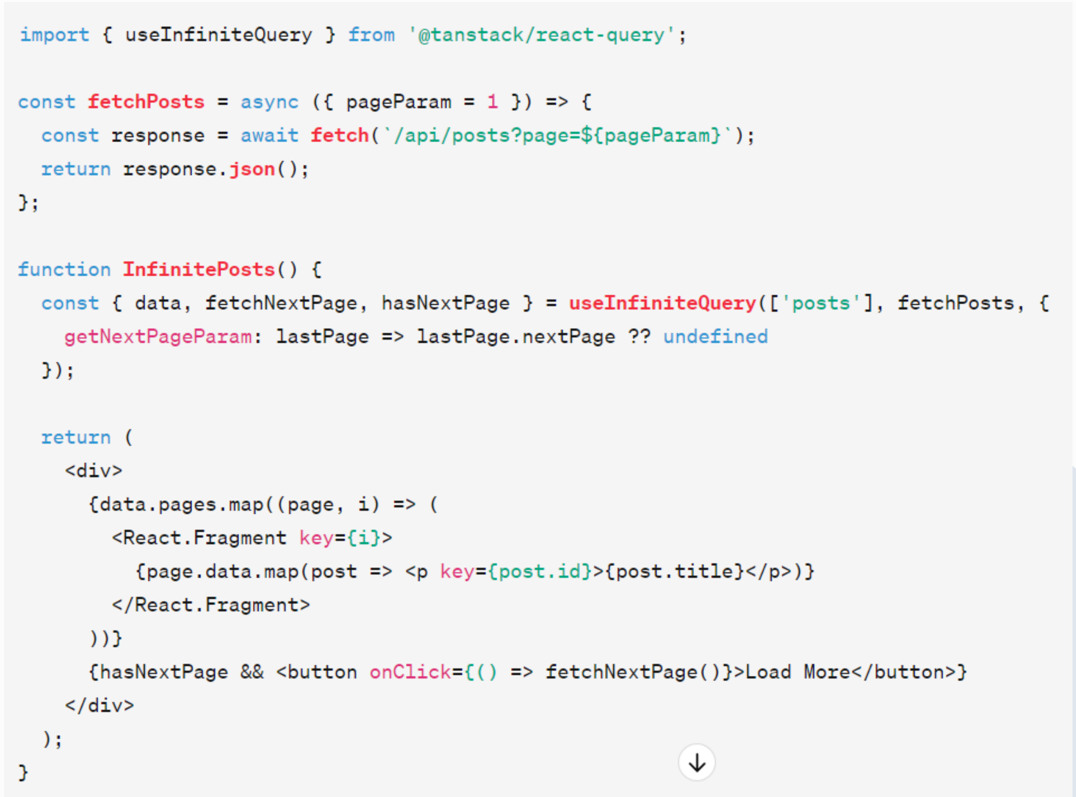
Paged requests are used to load data per page, reducing the load on the server and improving the user experience by loading data progressively.



```
const fetchProjects = async (page = 0) => {
  const response = await fetch(`/api/projects?page=${page}`);
  return response.json();
};

function PaginatedProjects() {
  const [page, setPage] = useState(0);
  const { data: projects, isLoading, isError, error } = useQuery({
    queryKey: ['projects', page],
    queryFn: () => fetchProjects(page),
    keepPreviousData: true,
  });

  if (isLoading) return <div>Loading...</div>;
  if (isError) return <div>Error: {error.message}</div>;

  return (
    <div>
      {projects.map(project => <div key={project.id}>{project.name}</div>)}
      <button onClick={() => setPage(old => old + 1)}>Next Page</button>
    </div>
  );
}
```

Introduction to useInfiniteQuery

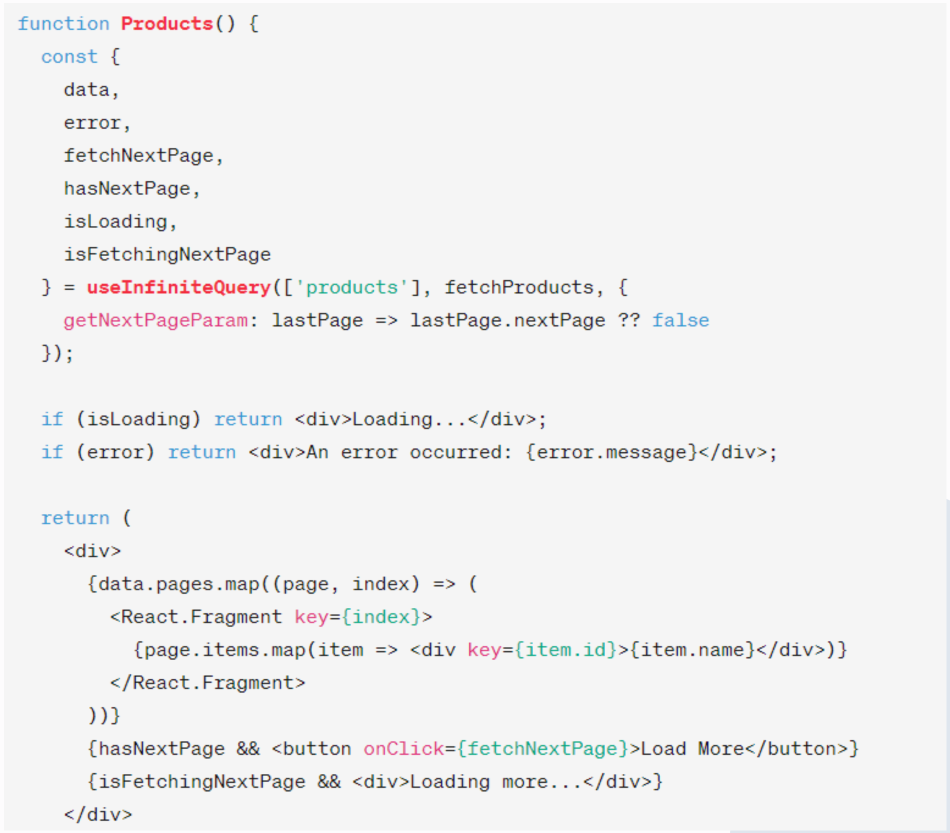
Configuring useInfiniteQuery involves specifying how to retrieve data and how to identify the next page to load.



```
import { useInfiniteQuery } from '@tanstack/react-query';

const fetchPosts = async ({ pageParam = 1 }) => {
  const response = await fetch(`/api/posts?page=${pageParam}`);
  return response.json();
};

function InfinitePosts() {
  const { data, fetchNextPage, hasNextPage } = useInfiniteQuery(['posts'], fetchPosts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? undefined
  });

  return (
    <div>
      {data.pages.map((page, i) => (
        <React.Fragment key={i}>
          {page.data.map(post => <p key={post.id}>{post.title}</p>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={() => fetchNextPage()}>Load More</button>}
    </div>
  );
}
```

Configuring useInfiniteQuery

Configuring useInfiniteQuery involves specifying how to retrieve data and how to identify the next page to load.



```
function Products() {
  const {
    data,
    error,
    fetchNextPage,
    hasNextPage,
    isLoading,
    isFetchingNextPage
  } = useInfiniteQuery(['products'], fetchProducts, {
    getNextPageParam: lastPage => lastPage.nextPage ?? false
  });

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>An error occurred: {error.message}</div>;

  return (
    <div>
      {data.pages.map((page, index) => (
        <React.Fragment key={index}>
          {page.items.map(item => <div key={item.id}>{item.name}</div>)}
        </React.Fragment>
      ))}
      {hasNextPage && <button onClick={fetchNextPage}>Load More</button>}
      {isFetchingNextPage && <div>Loading more...</div>}
    </div>
```

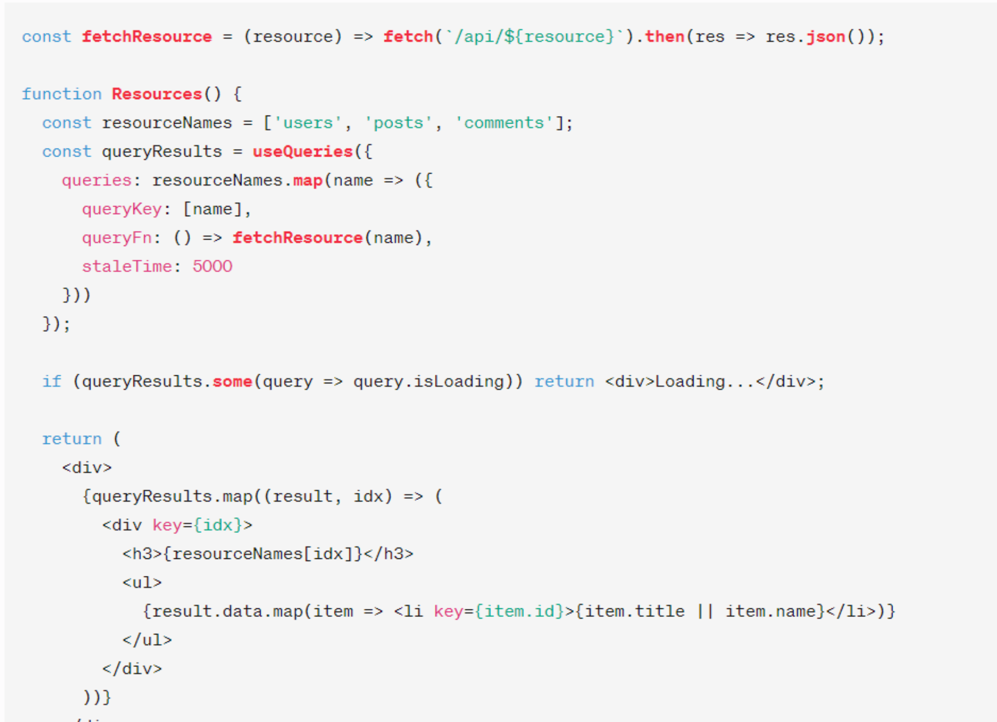
Simultaneous requests

Use the useQueries hook to execute multiple queries simultaneously. This is useful for loading several independent data items at the same time.

```
import { useQueries } from '@tanstack/react-query';

function FetchMultipleData() {
  const results = useQueries({
    queries: [
      { queryKey: ['user', 1], queryFn: () => fetch('/api/user/1').then(res => res.json()) },
      { queryKey: ['post', 1], queryFn: () => fetch('/api/post/1').then(res => res.json()) }
    ]
  });

  return (
    <div>
      <h1>User: {results[0].data?.name}</h1>
      <h2>First Post: {results[1].data?.title}</h2>
    </div>
  );
}
```

Configuring useQueries for multiple queries

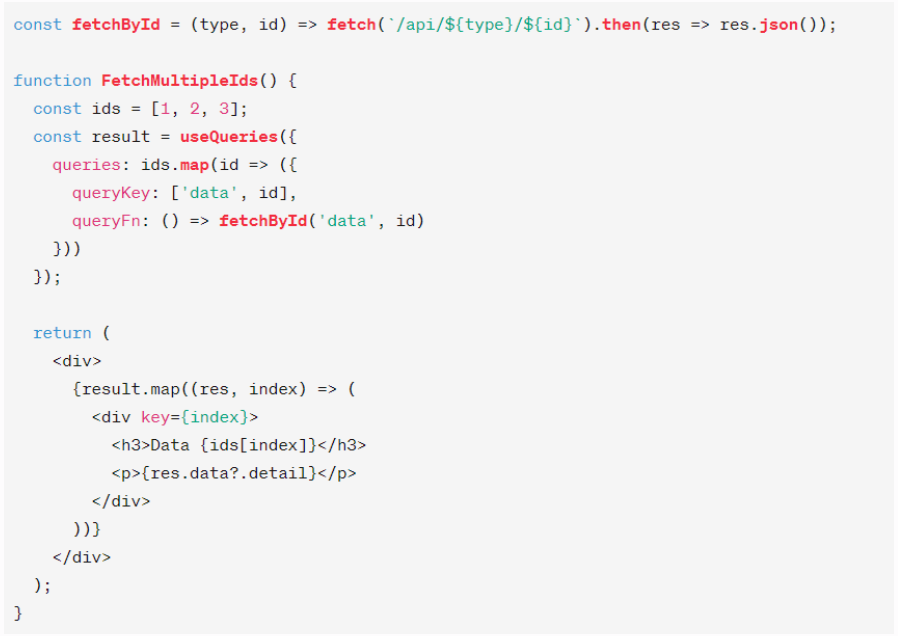
useQueries can be used to launch several simultaneous queries. Each query is independent but configured in a single hook call, simplifying the management of multiple data sources.



```
const fetchResource = (resource) => fetch(`/api/${resource}`).then(res => res.json());

function Resources() {
  const resourceNames = ['users', 'posts', 'comments'];
  const queryResults = useQueries({
    queries: resourceNames.map(name => ({
      queryKey: [name],
      queryFn: () => fetchResource(name),
      staleTime: 5000
    }))
  });

  if (queryResults.some(query => query.isLoading)) return <div>Loading...</div>;

  return (
    <div>
      {queryResults.map((result, idx) => (
        <div key={idx}>
          <h3>{resourceNames[idx]}</h3>
          <ul>
            {result.data.map(item => <li key={item.id}>{item.title || item.name}</li>)}
          </ul>
        </div>
      ))}
```

A simple example with useQueries

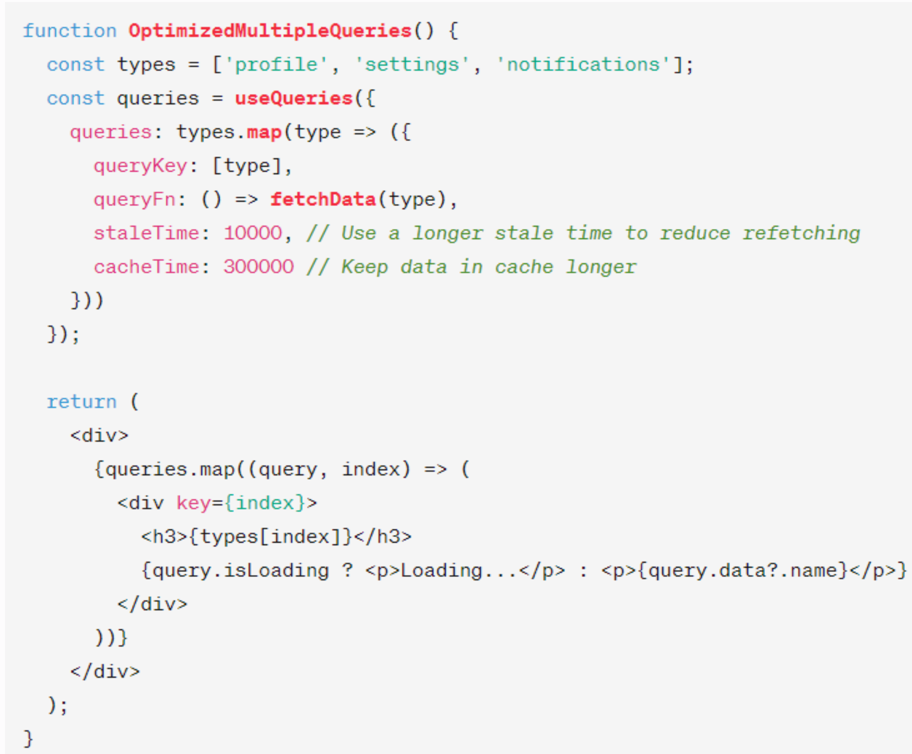
UseQueries to load data from different API endpoints simultaneously.



```
const fetchById = (type, id) => fetch(`/api/${type}/${id}`).then(res => res.json());

function FetchMultipleIds() {
  const ids = [1, 2, 3];
  const result = useQueries({
    queries: ids.map(id => ({
      queryKey: ['data', id],
      queryFn: () => fetchById('data', id)
    }))
  });

  return (
    <div>
      {result.map((res, index) => (
        <div key={index}>
          <h3>Data {ids[index]}</h3>
          <p>{res.data?.detail}</p>
        </div>
      ))}
    </div>
  );
}
```

Optimizing simultaneous requests

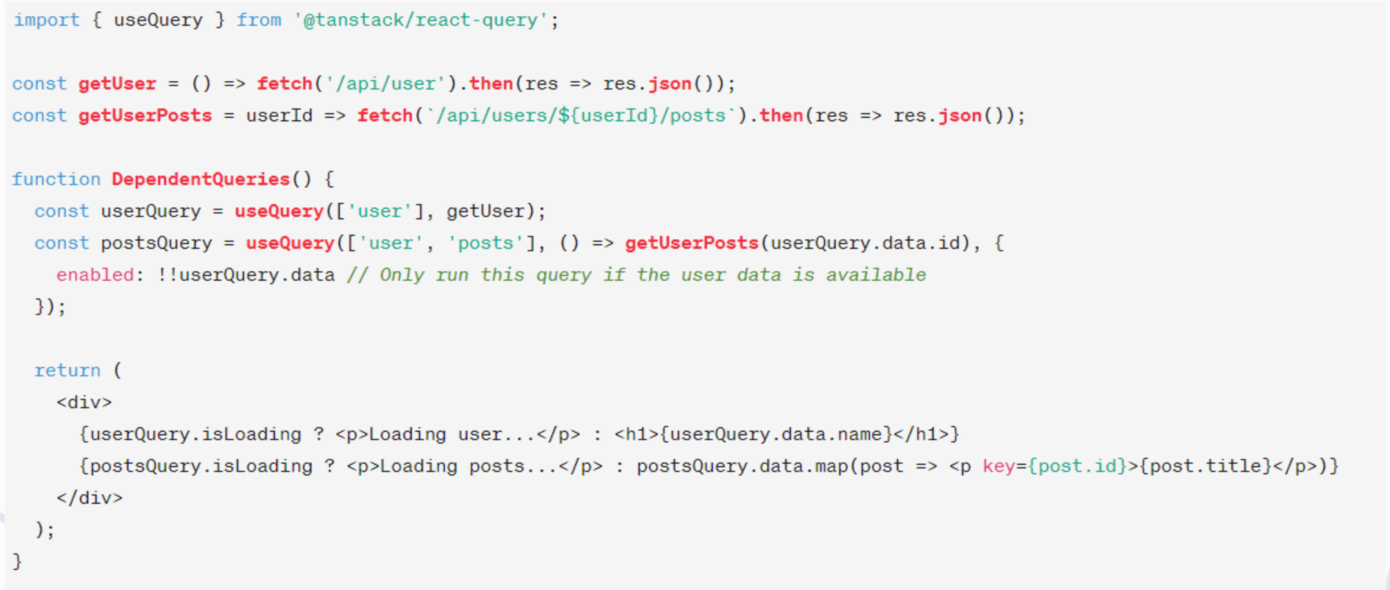
Strategies for optimizing the performance and efficiency of simultaneous requests, such as cache sharing and reducing network calls.



```
function OptimizedMultipleQueries() {
  const types = ['profile', 'settings', 'notifications'];
  const queries = useQueries({
    queries: types.map(type => ({
      queryKey: [type],
      queryFn: () => fetchData(type),
      staleTime: 10000, // Use a longer stale time to reduce refetching
      cacheTime: 300000 // Keep data in cache longer
    }))
  });

  return (
    <div>
      {queries.map((query, index) => (
        <div key={index}>
          <h3>{types[index]}</h3>
          {query.isLoading ? <p>Loading...</p> : <p>{query.data?.name}</p>}
        </div>
      ))}
    </div>
  );
}
```

Managing dependencies between queries

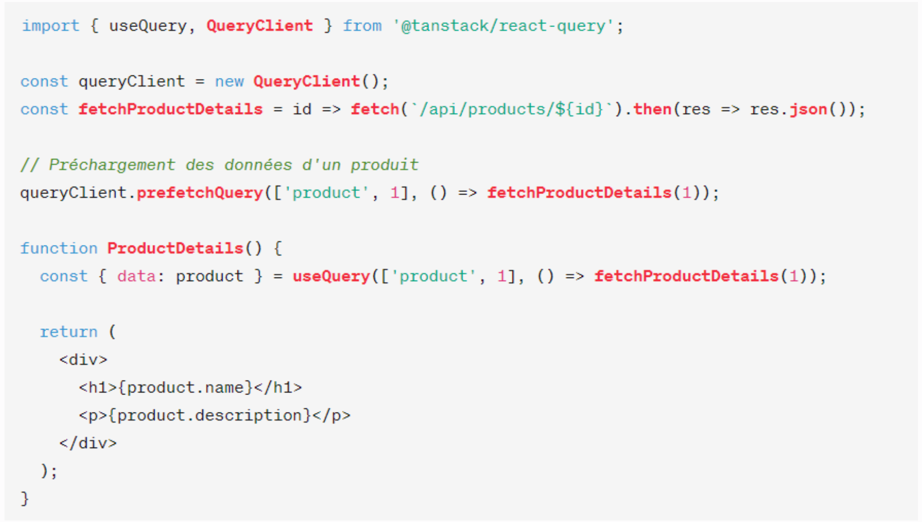
Manage dependencies between queries with useQuery to ensure that some data is loaded before others.



```
import { useQuery } from '@tanstack/react-query';

const getUser = () => fetch('/api/user').then(res => res.json());
const getUserPosts = userId => fetch(`/api/users/${userId}/posts`).then(res => res.json());

function DependentQueries() {
  const userQuery = useQuery(['user'], getUser);
  const postsQuery = useQuery(['user', 'posts'], () => getUserPosts(userQuery.data.id), {
    enabled: !!userQuery.data // Only run this query if the user data is available
  });

  return (
    <div>
      {userQuery.isLoading ? <p>Loading user...</p> : <h1>{userQuery.data.name}</h1>}
      {postsQuery.isLoading ? <p>Loading posts...</p> : postsQuery.data.map(post => <p key={post.id}>{post.title}</p>)}
    </div>
  );
}
```

Data preloading

Introducing Tanstack Query's preloading functionality to enhance the user experience by loading data before it is needed.



```
import { useQuery, QueryClient } from '@tanstack/react-query';

const queryClient = new QueryClient();
const fetchProductDetails = id => fetch(`/api/products/${id}`).then(res => res.json());

// Préchargement des données d'un produit
queryClient.prefetchQuery(['product', 1], () => fetchProductDetails(1));

function ProductDetails() {
  const { data: product } = useQuery(['product', 1], () => fetchProductDetails(1));

  return (
    <div>
      <h1>{product.name}</h1>
      <p>{product.description}</p>
    </div>
  );
}
```

Data preloading

Introducing Tanstack Query's preloading functionality to enhance the user experience by loading data before it's needed.

Selective cache disabling

Selective invalidation makes it possible to refresh some data in the cache without affecting others, which is crucial for keeping data synchronized with servers.

```
import { useQueryClient, useQuery } from '@tanstack/react-query';

const fetchUser = id => fetch(`/api/users/${id}`).then(res => res.json());

function UserProfile({ userId }) {
  const queryClient = useQueryClient();
  useQuery(['user', userId], () => fetchUser(userId), {
    onSuccess: () => {
      // Invalider et rafraîchir les dépendances
      queryClient.invalidateQueries(['posts', userId]);
    }
  });

  return <div>User Profile</div>;
}
```

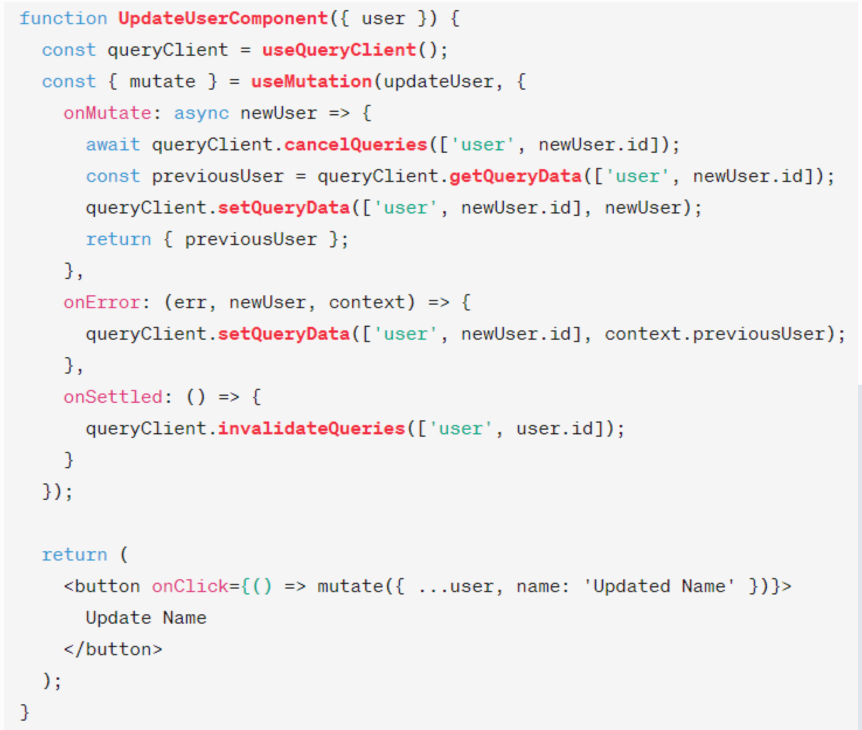
Selective cache disabling

Selective invalidation makes it possible to refresh some data in the cache without affecting others, which is crucial for keeping data synchronized with servers.

```
import { useMutation, useQueryClient } from '@tanstack/react-query';

const updateUser = userData => fetch('/api/user', {
  method: 'PUT',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(userData)
});

function UpdateUser() {
  const queryClient = useQueryClient();
  const mutation = useMutation(updateUser, {
    onSuccess: () => {
      // Invalider uniquement les requêtes liées à l'utilisateur mis à jour
      queryClient.invalidateQueries(['userProfile']);
    }
  });

  return <button onClick={() => mutation.mutate({ id: 1, name: 'Jane Doe' })}>Update User</button>;
}
```

Introduction to Optimistic Updates

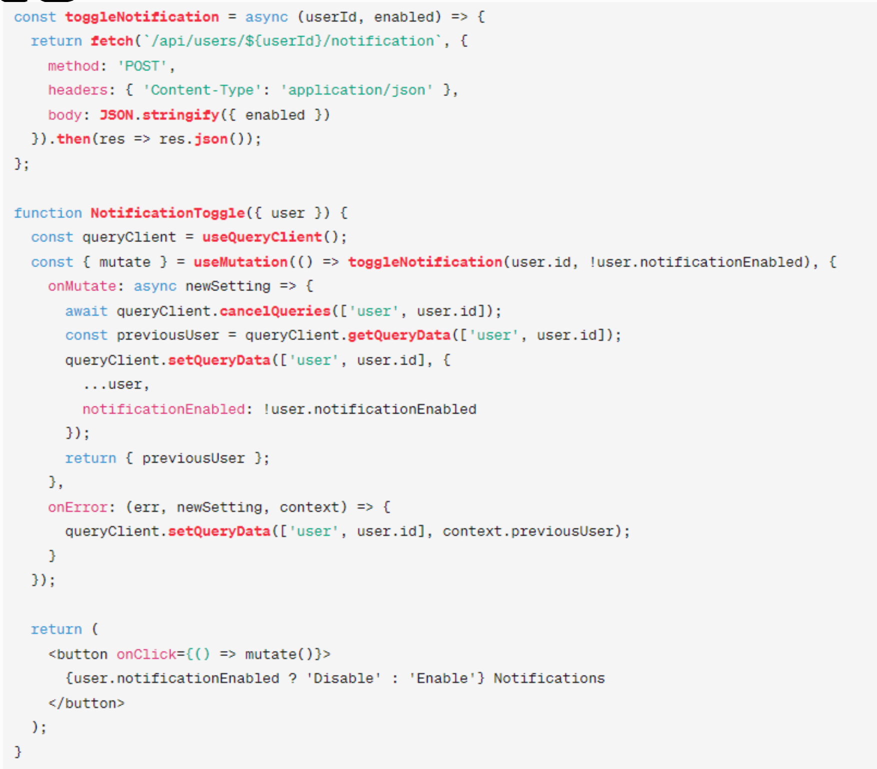
Optimistic Updates are a strategy for improving the user experience in interactive web applications, by immediately applying presumed changes without waiting for server confirmation.



```
function UpdateUserComponent({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(updateUser, {
    onMutate: async newUser => {
      await queryClient.cancelQueries(['user', newUser.id]);
      const previousUser = queryClient.getQueryData(['user', newUser.id]);
      queryClient.setQueryData(['user', newUser.id], newUser);
      return { previousUser };
    },
    onError: (err, newUser, context) => {
      queryClient.setQueryData(['user', newUser.id], context.previousUser);
    },
    onSettled: () => {
      queryClient.invalidateQueries(['user', user.id]);
    }
  });

  return (
    <button onClick={() => mutate({ ...user, name: 'Updated Name' })}>
      Update Name
    </button>
  );
}
```

A practical example of Optimistic Update

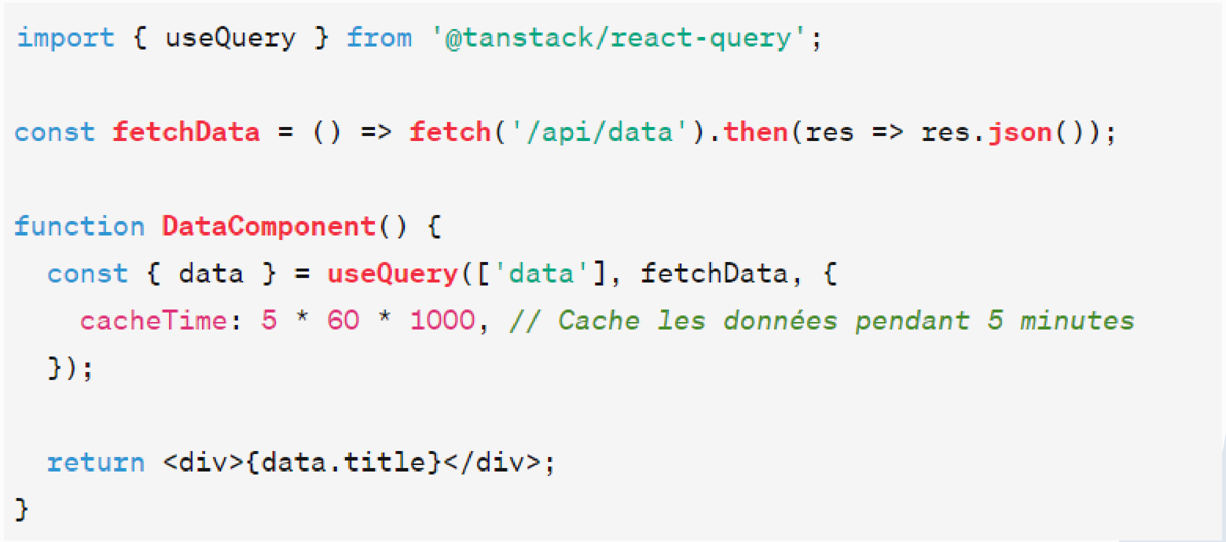
Demonstration of an Optimistic Update with an example where a user can activate or deactivate a notification feature instantly.



```
const toggleNotification = async (userId, enabled) => {
  return fetch(`/api/users/${userId}/notification`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ enabled })
  }).then(res => res.json());
};

function NotificationToggle({ user }) {
  const queryClient = useQueryClient();
  const { mutate } = useMutation(() => toggleNotification(user.id, !user.notificationEnabled), {
    onMutate: async newSetting => {
      await queryClient.cancelQueries(['user', user.id]);
      const previousUser = queryClient.getQueryData(['user', user.id]);
      queryClient.setQueryData(['user', user.id], {
        ...user,
        notificationEnabled: !user.notificationEnabled
      });
      return { previousUser };
    },
    onError: (err, newSetting, context) => {
      queryClient.setQueryData(['user', user.id], context.previousUser);
    }
  });

  return (
    <button onClick={() => mutate()}>
      {user.notificationEnabled ? 'Disable' : 'Enable'} Notifications
    </button>
  );
}
```

Caching basics

Caching with Tanstack Query enables query results to be stored and reused to improve the speed of data loading and reduce the load on backend servers.



```
import { useQuery } from '@tanstack/react-query';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataComponent() {
  const { data } = useQuery(['data'], fetchData, {
    cacheTime: 5 * 60 * 1000, // Cache les données pendant 5 minutes
  });

  return <div>{data.title}</div>;
}
```

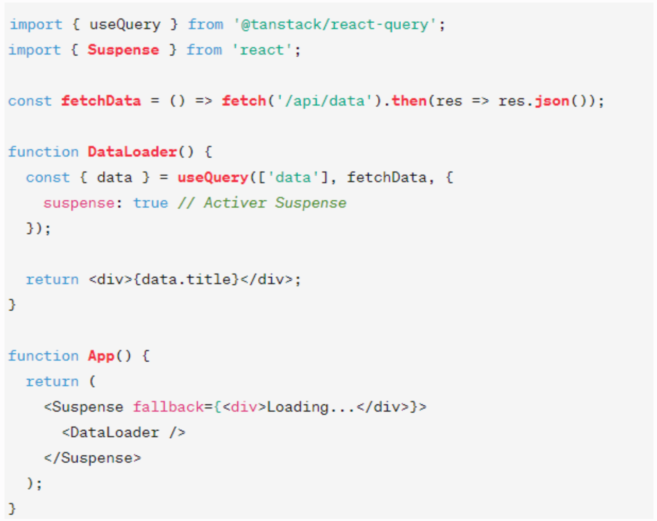
Configuring caching for optimized performance

Configure cache to maximize performance, by defining cache lifetime policies and customizing data recovery strategies.

```
import { useQuery, useQueryClient } from '@tanstack/react-query';

const fetchProduct = id => fetch(`/api/products/${id}`).then(res => res.json());

function Product({ productId }) {
  const queryClient = useQueryClient();
  const { data: product } = useQuery(['product', productId], () => fetchProduct(productId), {
    onSuccess: () => {
      // Précharger les données relatives dès que le produit est chargé
      queryClient.prefetchQuery(['reviews', productId], () => fetch(`/api/products/${productId}/reviews`));
    }
  });

  return <div>{product.name}</div>;
}
```

Using Suspense with React Query

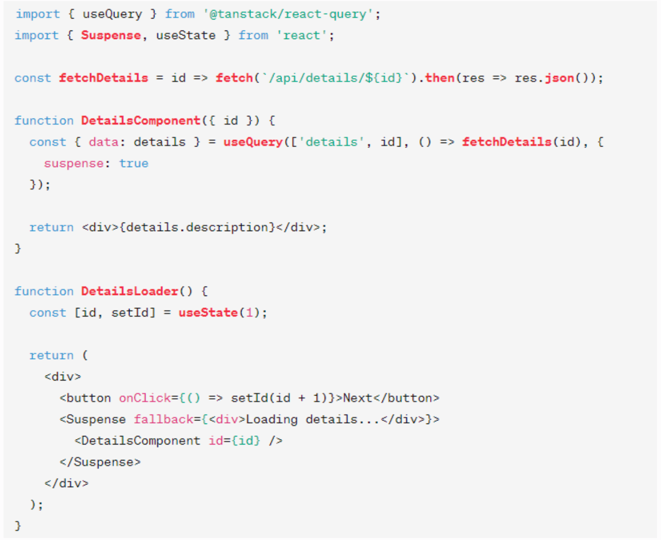
Explanation of the integration of React Suspense with React Query to manage loading states in a more declarative and homogeneous way.



```
import { useQuery } from '@tanstack/react-query';
import { Suspense } from 'react';

const fetchData = () => fetch('/api/data').then(res => res.json());

function DataLoader() {
  const { data } = useQuery(['data'], fetchData, {
    suspense: true // Activer Suspense
  });

  return <div>{data.title}</div>;
}

function App() {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <DataLoader />
    </Suspense>
  );
}
```

Suspense integration with React components

Techniques for integrating Suspense with React components, facilitating the management of data dependencies and the manipulation of loading states.



```
import { useQuery } from '@tanstack/react-query';
import { Suspense, useState } from 'react';

const fetchDetails = id => fetch(`/api/details/${id}`).then(res => res.json());

function DetailsComponent({ id }) {
  const { data: details } = useQuery(['details', id], () => fetchDetails(id), {
    suspense: true
  });

  return <div>{details.description}</div>;
}

function DetailsLoader() {
  const [id, setId] = useState(1);

  return (
    <div>
      <button onClick={() => setId(id + 1)}>Next</button>
      <Suspense fallback={<div>Loading details...</div>}>
        <DetailsComponent id={id} />
      </Suspense>
    </div>
  );
}
```

09 Redux and Zustand

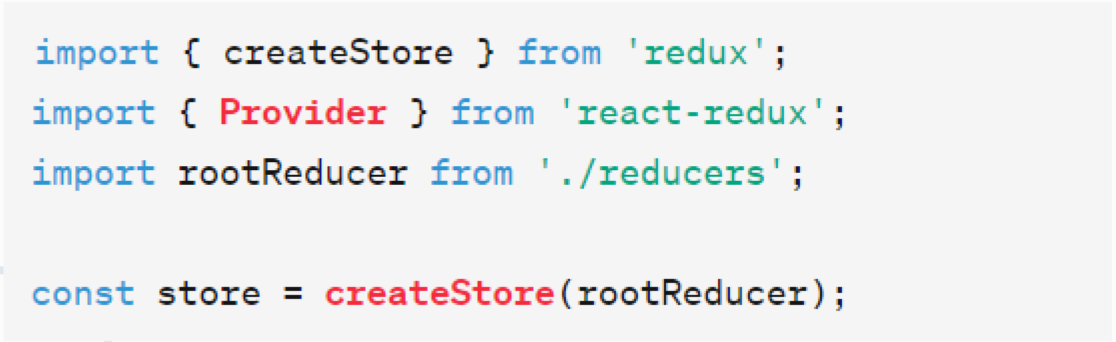
Introduction to Redux

Redux is a predictable state management library for JavaScript applications. It helps you write applications that behave consistently, run in different environments (client, server and native), and are easy to test.

Redux fundamentals

Redux is based on a few key principles: the global state of the application is stored in a single tree object within a single store. State is read-only. State changes are effected by sending actions. Reducers are pure functions that take the previous state and an action, and return the new state.

Initial Redux configuration



```
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from './reducers';

const store = createStore(rootReducer);
```

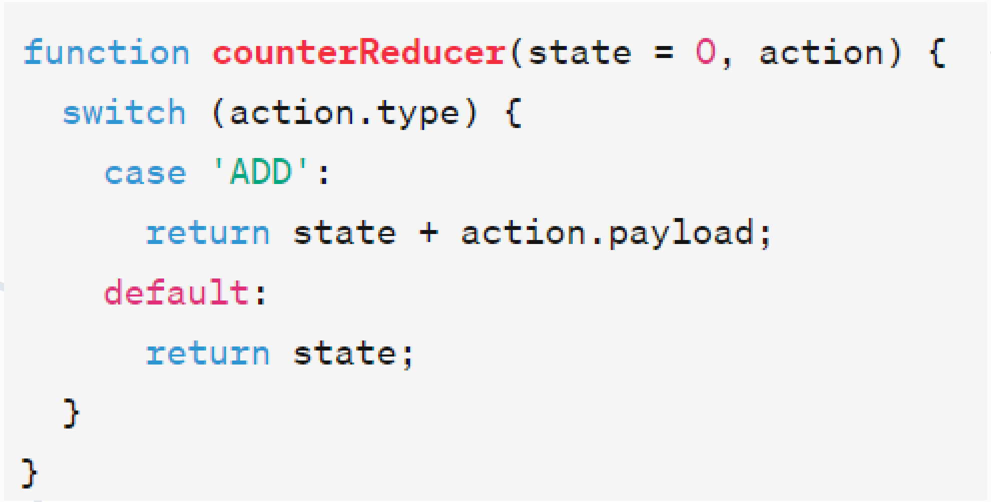
To get started with Redux, install redux and react-redux. Then create a Redux store and provide it to your application via the React-Redux Provider.

Actions

Actions are JavaScript objects that send data from your application to your store. They are the only source of information for the store.

```
const addAction = { type: 'ADD', payload: 1 };
```

Reducers



```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case 'ADD':
      return state + action.payload;
    default:
      return state;
  }
}
```

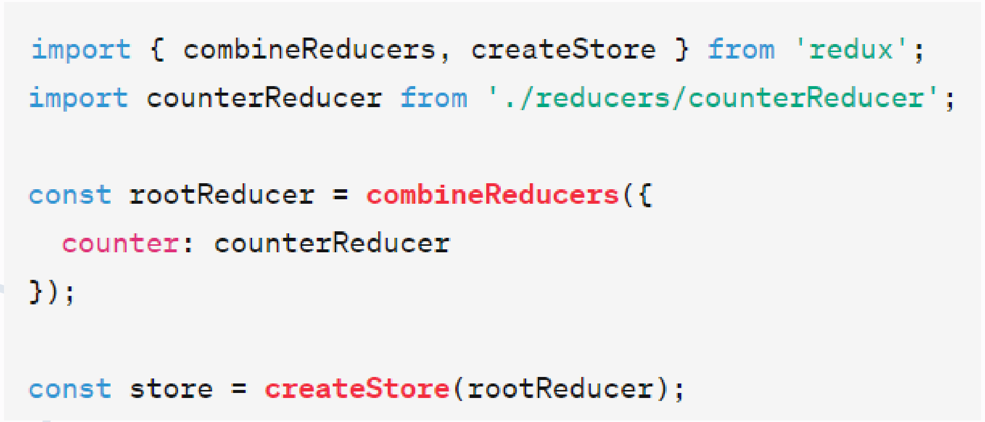
- Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions describe the fact that something has happened, but do not specify how the application's state changes.

useSelector and useDispatch

useSelector gives you access to the store state, while useDispatch gives you access t the dispatch function for sending actions.

```
import { useSelector, useDispatch } from 'react-redux';

function Counter() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <button onClick={() => dispatch({ type: 'ADD', payload: 1 })}>
        Increment
      </button>
      <span>{count}</span>
    </div>
  );
}
```

Store and condition management



```
import { combineReducers, createStore } from 'redux';
import counterReducer from './reducers/counterReducer';

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

const store = createStore(rootReducer);
```

- The Redux store acts as a container for the global state of your application. Use combineReducers to split the state and reduction logic into several functions managing independent parts of the state.

Redux middleware

Middleware provides an extension point between the time an action is sent and the time it reaches the reducer. Use redux-thunk to handle asynchronous logic.

```
import { applyMiddleware, createStore } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));
```

Introduction to Redux Toolkit

The Redux Toolkit (RTK) is a set of tools designed to simplify Redux code, encourage best practice and improve Redux developability. It offers utilities to simplify store configuration, reducer definition, asynchronous logic management and more.

Advantages over standard Redux

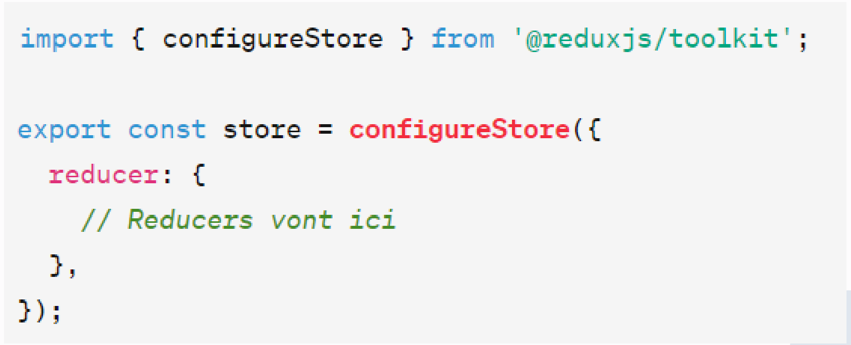
RTK reduces the amount of boilerplate code needed to set up a Redux store, simplifies the management of actions and reducers with createSlice, automates the creation of actions, and facilitates the management of asynchronous logic with createAsyncThunk.

Installation and configuration

- To get started with RTK, install the @reduxjs/toolkit package, as well as react-redux if you're working with React.

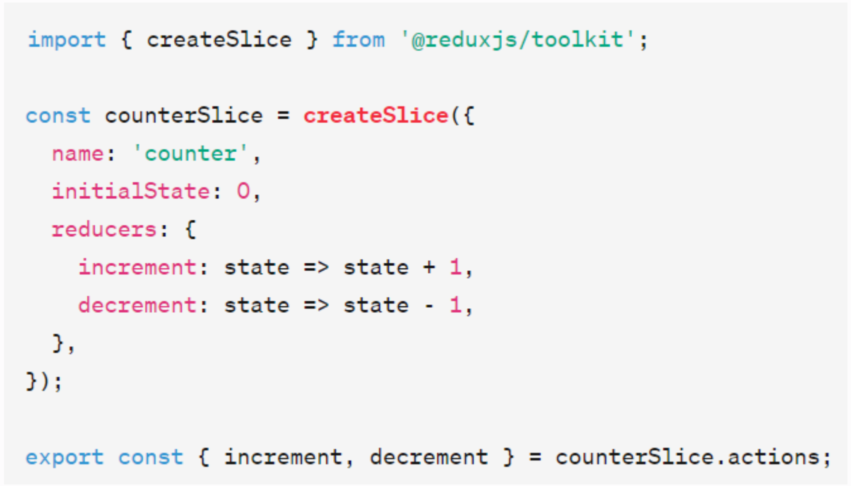
Configure the Store with configureStore

configureStore simplifies store configuration by automatically including middleware such as Redux Thunk and enabling Redux development tools.



```
import { configureStore } from '@reduxjs/toolkit';

export const store = configureStore({
  reducer: {
    // Reducers vont ici
  },
});
```

Slice creation with createSlice

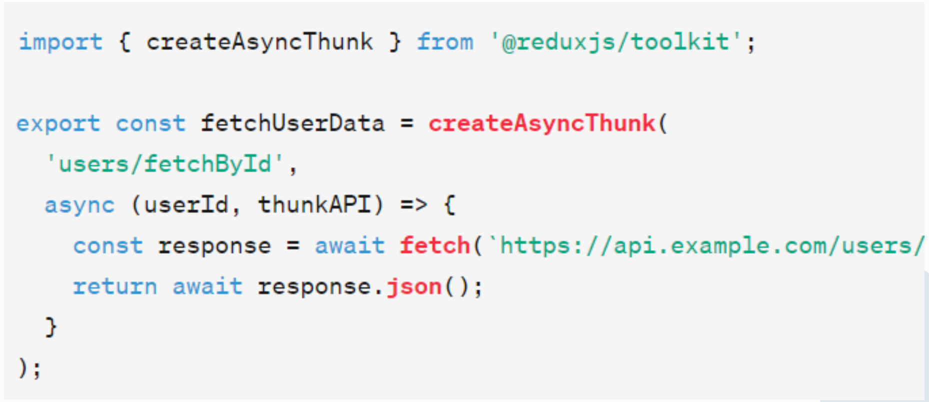


```javascript
import { createSlice } from '@reduxjs/toolkit';

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

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

- createSlice groups reducers and their corresponding actions in a single object, simplifying state management.

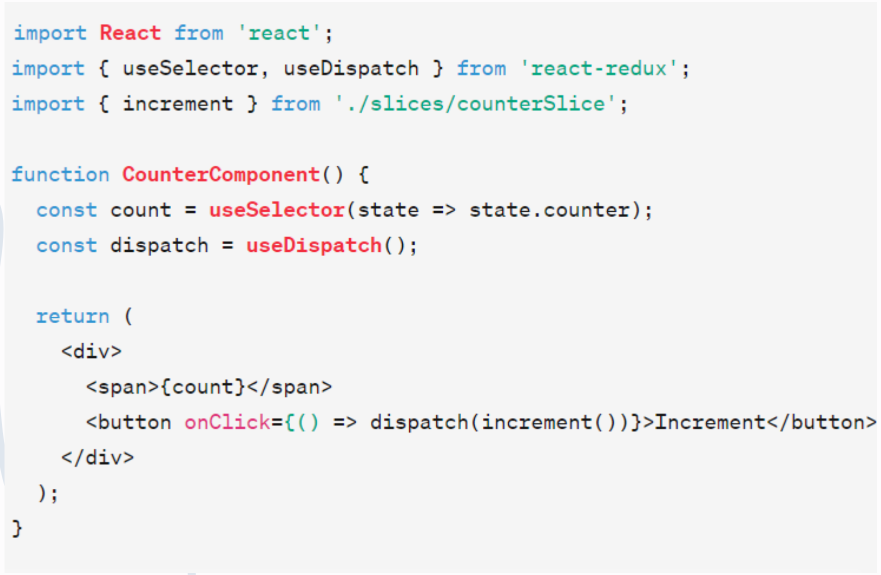
Managing side effects with createAsyncThunk

createAsyncThunk simplifies the management of asynchronous operations by encapsulating asynchronous logic and request state processing (loading, success, error) in a single function.



```
import { createAsyncThunk } from '@reduxjs/toolkit';

export const fetchUserData = createAsyncThunk(
  'users/fetchById',
  async (userId, thunkAPI) => {
    const response = await fetch(`https://api.example.com/users/
    return await response.json();
  }
);
```

Using useDispatch and useSelector with Redux Toolkit and React



```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './slices/counterSlice';

function CounterComponent() {
  const count = useSelector(state => state.counter);
  const dispatch = useDispatch();

  return (
    <div>
      <span>{count}</span>
      <button onClick={() => dispatch(increment())}>Increment</button>
    </div>
  );
}
```

- RTK works seamlessly with react-redux's useDispatch and useSelector hooks, making it easy to access status and dispatch actions in React components.

Introduction to Zustand

Zustand is a small state management library for React that offers a simpler, more straightforward approach than Redux. With Zustand, creating global stores to manage state is simplified and requires no boilerplate or additional middleware.

Zustand philosophy and benefits

- Zustand focuses on a minimalist API and a custom hook to access the store. Benefits include easy configuration, no dependency on Redux or Context API, and natural integration with React hooks.

Zustand installation

Installing Zustand is straightforward, using npm or yarn. This installs the latest version of Zustand in your project.

```
npm install zustand
// ou
yarn add zustand
```

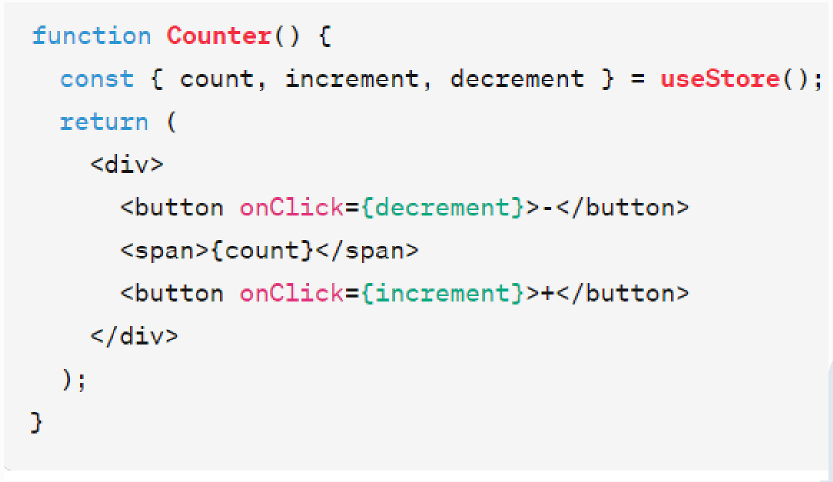
Creating an awning with Zustand

```
import create from 'zustand';

const useStore = create(set => ({
  count: 0,
  increment: () => set(state => ({ count: state.count + 1 })),
  decrement: () => set(state => ({ count: state.count - 1 })),
}));
```

To create a blind in Zustand, use the create function. Here you can define the initial state of the blind and the actions that will manipulate this state. Zustand lets you create blinds without the overhead traditionally associated with Redux.

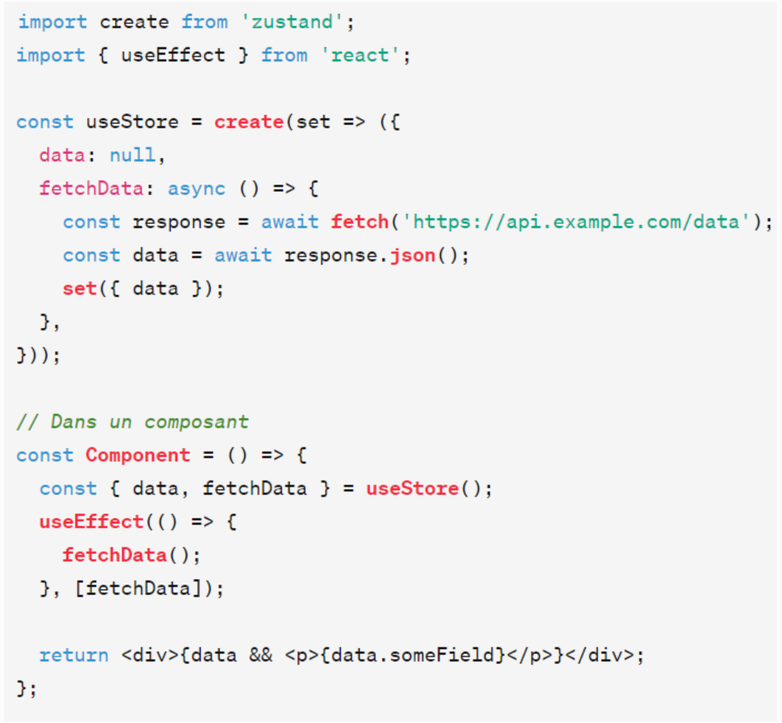
State management with hooks

Zustand uses hooks to access and manipulate the state of the store. This makes integration with React functional components natural and efficient.



```
function Counter() {
  const { count, increment, decrement } = useStore();
  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  );
}
```

Managing side effects



```
import create from 'zustand';
import { useEffect } from 'react';

const useStore = create(set => ({
  data: null,
  fetchData: async () => {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    set({ data });
  },
}));

// Dans un composant
const Component = () => {
  const { data, fetchData } = useStore();
  useEffect(() => {
    fetchData();
  }, [fetchData]);

  return <div>{data && <p>{data.someField}</p>}</div>;
};
```

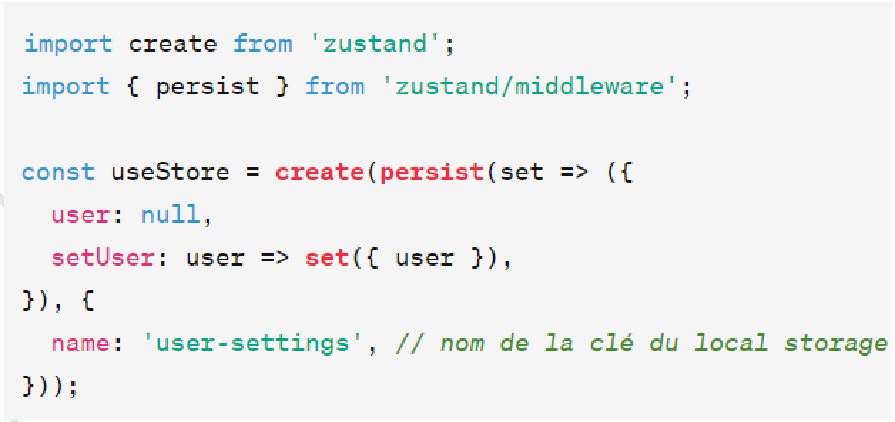
Zustand lets you manage secondary effects by using actions or reacting to specific changes in state using middleware or the native React.useEffect API.

Selectors and subscriptions

Zustand allows you to select part of the state when using the store hook, reducing unnecessary re-renders. Subscriptions to state changes are also possible for fine-tuned update management.

```
const count = useStore(state => state.count);
```

Examples of advanced Zustand use



```
import create from 'zustand';
import { persist } from 'zustand/middleware';

const useStore = create(persist(set => ({
  user: null,
  setUser: user => set({ user }),
}), {
  name: 'user-settings', // nom de la clé du local storage
}));
```

Zustand supports advanced use cases such as state sharing between different components, state persistence in local storage, and integration with debugging tools.

To find out more…

Daishi Kato: https://twitter.com/dai_shi

Video from Jack Herrington: https://www.youtube.com/watch?v=sqTPGMipjHk