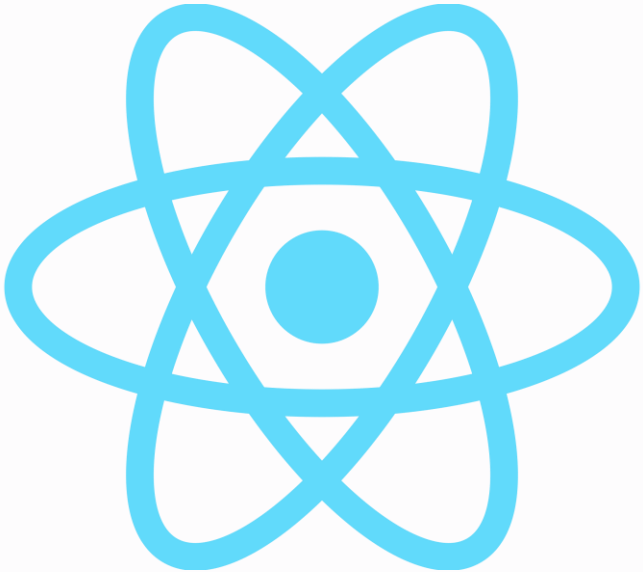
React JS



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 - Store management

- **Using Redux in your project**
- Redux: the basics
- Redux toolkit: slices & asyncthunk
- RTK Query

- **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

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

01

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));
```

Combining selectors with Reselect

Use the Reselect library to create memorized selectors that can combine several selectors. This allows you to reduce the number of useSelectors in a single call.

```
import { createSelector } from 'reselect';

const selectDataA = (state) => state.dataA;
const selectDataB = (state) => state.dataB;

const selectCombinedData = createSelector(
  [selectDataA, selectDataB],
  (dataA, dataB) => {
    // Logique pour combiner dataA et dataB
    return { ...dataA, ...dataB };
  }
);

// Dans votre composant
const combinedData = useSelector(selectCombinedData);
```

Creating custom hooks

Encapsulate selection logic in custom hooks to improve readability and reusability.

```
const useCombinedData = () => {
  const dataA = useSelector(selectDataA);
  const dataB = useSelector(selectDataB);
  // Logique supplémentaire si nécessaire
  return { dataA, dataB };
};

// Utilisation dans le composant
const { dataA, dataB } = useCombinedData();
```

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

```
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;
```

- createSlice groups reducers and 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.

Creating a service with RTK Query

API service with RTK Query

```
import { createApi, fetchBaseQuery } from '@reduxjs/toolkit/query/react';

const api = createApi({
  reducerPath: 'api',
  baseQuery: fetchBaseQuery({ baseUrl: '/api' }),
  endpoints: (builder) => ({
    getUser: builder.query({
      query: (id) => `user/${id}`,
    }),
  }),
});

export const { useGetUserQuery } = api;
```

Awning configuration with RTK Query

Integrating RTK Query into the store

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

const store = configureStore({
  reducer: {
    [api.reducerPath]: api.reducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(api.middleware),
});

export default store;
```

Using hooks generated by RTK Query

Using RTK Query hooks

```
const User = ({ id }) => {
  const { data, error, isLoading } = useGetUserQuery(id);

  if (isLoading) return <div>Loading...</div>;
  if (error) return <div>Error occurred</div>;

  return <div>User: {data.name}</div>;
};
```

Mutation with RTK Query

Mutation (POST, PUT, DELETE)

```
endpoints: (builder) => ({
  updateUser: builder.mutation({
    query: (user) => ({
      url: `user/${user.id}`,
      method: 'PUT',
      body: user,
    }),
  }),
});

export const { useUpdateUserMutation } = api;
```

Using a mutation in a component

Mutation hook in components

```
const UpdateUser = ({ user }) => {
  const [updateUser] = useUpdateUserMutation();

  return (
    <button onClick={() => updateUser(user)}>
      Update User
    </button>
  );
};
```

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

02

Introduction to XState

Introduction to XState

- XState is a JavaScript library for modeling finite state machines and hierarchical state machines. It helps to manage the states of an application in a clear, predictable and deterministic way. Unlike tools like Redux, which focus on data flows and actions, XState emphasizes **state transitions** and **state logic**.
- **Key points:**
- XState is based on the concept of **finite-state machines**.
- Each machine has an initial state and a set of possible transitions.
- Transitions are triggered by events.

Problems related to status management

- In many applications, Redux and Zustand are often used to manage global states. However, when it comes to complex flows (nested states, multiple transitions, management of asynchronous side effects), these tools can become difficult to maintain. Here are some common problems that XState can solve:
- **Unpredictability**: In Redux, actions can be triggered unpredictably, creating behaviors that are difficult to follow.

- **Complex transition logic**: State transitions can become difficult to model and understand with nested reducers.

- **Lack of visualization**: It's difficult to visualize states and transitions in Redux or Zustand, making it tedious to manage complex applications.

Finite-state machines

- The components of a finite state machine :
    - **States**: A set of defined states (e.g. idle, loading, success, failure).
    - **Events**: Actions that cause transitions from one state to another.
    - **Transitions**: The rules that dictate how to move from one state to another in response to an event.
    - **Actions**: Side effects that occur during transitions.

```
import { Machine } from 'xstate';

const toggleMachine = Machine({
  id: 'toggle',
  initial: 'inactive',
  states: {
    inactive: {
      on: { TOGGLE: 'active' }
    },
    active: {
      on: { TOGGLE: 'inactive' }
    }
  }
});
```

XState and React

How to integrate XState into a React application using the useMachine hook.

```
import { useMachine } from '@xstate/react';
import { Machine } from 'xstate';

const toggleMachine = Machine({
  id: 'toggle',
  initial: 'inactive',
  states: {
    inactive: { on: { TOGGLE: 'active' } },
    active: { on: { TOGGLE: 'inactive' } }
  }
});

function ToggleButton() {
  const [state, send] = useMachine(toggleMachine);

  return (
    <button onClick={() => send('TOGGLE')}>
      {state.matches('inactive') ? 'Off' : 'On'}
    </button>
  );
}
```

Context in XState

Use context to store persistent data in an XState machine.

```
const toggleMachine = Machine({
  id: 'toggle',
  initial: 'inactive',
  context: { count: 0 },
  states: {
    inactive: {
      on: {
        TOGGLE: {
          target: 'active',
          actions: assign({ count: (context) => context.count + 1 })
        }
      }
    },
    active: { on: { TOGGLE: 'inactive' } }
  }
});
```

Asynchronous actions

Use XState to manage asynchronous actions, such as API calls.

```
const fetchMachine = Machine({
  id: 'fetch',
  initial: 'idle',
  states: {
    idle: {
      on: { FETCH: 'loading' }
    },
    loading: {
      invoke: {
        src: 'fetchData',
        onDone: { target: 'success', actions: assign({ data: (_, event) => event.data }) }
        onError: { target: 'failure' }
      }
    },
    success: { type: 'final' },
    failure: { on: { RETRY: 'loading' } }
  }
});
```

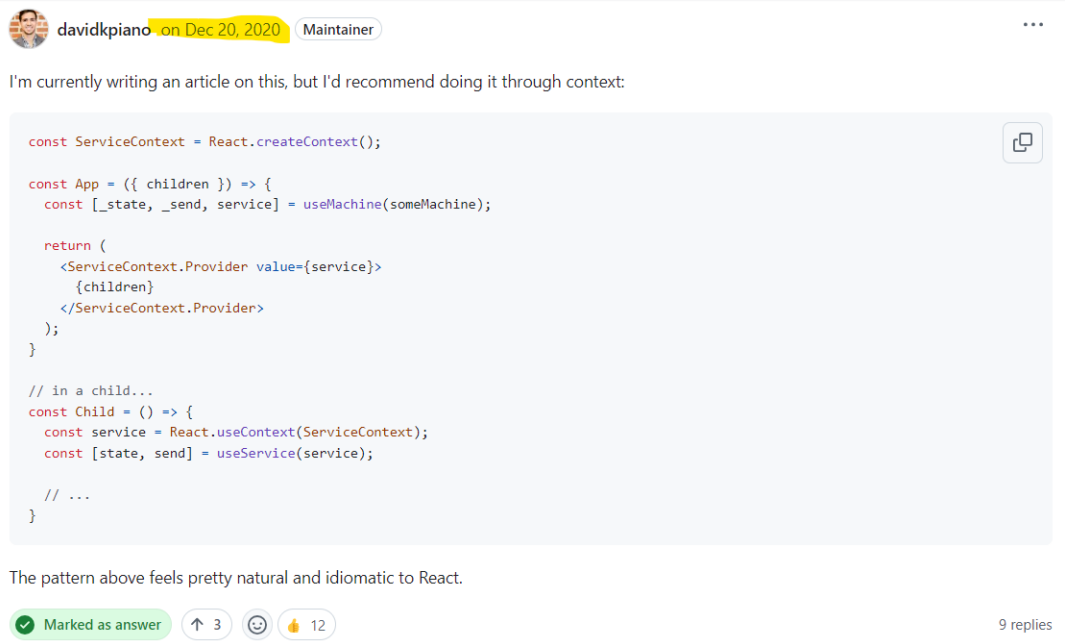
Guard conditions

Use **guards** to condition transitions according to the current state.

```
const toggleMachine = Machine({
  initial: 'inactive',
  context: { attempts: 0 },
  states: {
    inactive: {
      on: {
        TOGGLE: {
          target: 'active',
          cond: (context) => context.attempts < 5
        }
      }
    },
    active: { on: { TOGGLE: 'inactive' } }
  }
});
```

Context API and XState

React's **Context API** can be used with XState to share machine state across the component tree. This makes it possible to **manage a global state** without having to manually pass props to each component level.



davidkpiano on Dec 20, 2020 Maintainer

I'm currently writing an article on this, but I'd recommend doing it through context:

```
const ServiceContext = React.createContext();

const App = ({ children }) => {
  const [_state, _send, service] = useMachine(someMachine);

  return (
    <ServiceContext.Provider value={service}>
      {children}
    </ServiceContext.Provider>
  );
}

// in a child...
const Child = () => {
  const service = React.useContext(ServiceContext);
  const [state, send] = useService(service);

  // ...
}
```

The pattern above feels pretty natural and idiomatic to React.

Marked as answer 3 12 9 replies

No news?

siman on Jun 6, 2021

> I tried looking up this article, but did not seem to find it. Do you have a link?
>
> Not finished yet.

Just wondering if the article is finished by now

davidkpiano on Jun 6, 2021 **Maintainer**

> I tried looking up this article, but did not seem to find it. Do you have a link?
>
> Not finished yet.
>
> Just wondering if the article is finished by now

Sorry, I got tied up with other things that are higher priority. Will get to this sometime this year.

Simple way without context API :

davidkpiano on Aug 31, 2023 Maintainer

edited

There's also a much simpler way if you don't want to use context:

```
import { createActor } from 'xstate'; // xstate@beta

export const actor = createActor(someMachine).start();

// Use anywhere
import { useSelector } from '@xstate/react'; // @xstate/react@beta
import { actor } from '../path/to/actor';

const Component = () => {
  const count = useSelector(actor, s => s.context.count);

  // ...
  // can directly call `actor.send(...)`
}
```

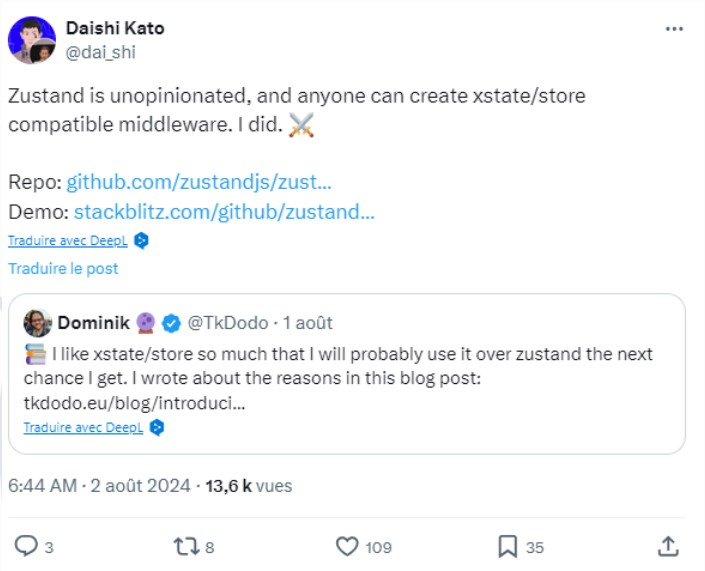
1 1

More about xState ? -> follow david!

- https://www.reactiflux.com/transcripts/david-k-piano

Zustand x Xstate, possible?

Yup.



03

Introduction to Code Splitting

Introduction to Code Splitting

•Code splitting is a technique used to divide an application into several smaller bundles, loaded on demand. It improves performance by reducing the initial weight of the bundle.

•It is particularly useful in React applications with many components and dependencies.

Why use Code Splitting?

- Key benefits :
- Improved initial loading times.
- Reducing the impact of heavy dependency.
- Load resources only when needed.
- Used with tools such as Webpack or Vite.

React.lazy for Code Splitting

React.lazy() is used to dynamically load components. It allows components to be split on demand.

```
// Lazy load of MyComponent
const MyComponent = React.lazy(() => import('./MyComponent'));

// Rendering
function App() {
  return (
    <React.Suspense fallback={<div>Loading...</div>}>
      <MyComponent />
    </React.Suspense>
  );
}

// React.Suspense is used to handle the loading state
```

React.Suspense for load management

Suspense is used to display a fallback while the component is loaded. It's essential to add a fallback to avoid UX problems during loading.

```
<React.Suspense fallback={<div>Loading...</div>}>
  <MyComponent />
</React.Suspense>
```

Dynamic Imports

Dynamic imports allow modules to be loaded on demand. This is a key feature of modern JavaScript.

```
import('./math').then(math => {
  console.log(math.add(16, 26));
});
```

Code Splitting and Webpack

Webpack supports splitting code natively.
It generates separate bundles when import() or React.lazy() is used.

```
// Webpack configuration for code splitting
module.exports = {
  entry: './src/index.js',
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
  },
};
```

04 Introduction to React DevTools

Exploring components with React DevTools

- **Tree view**: visualizes component structure.
- **Practical tools**: Click on a component to view its details (props, state).
- **Quick search**: Use the search bar to find a component by name.

Props inspection and live State

- **Props**: Displays properties passed to a component from its parent.
- **State**: Shows the component's internal state.
- **Live modification**: Change props or state values
for live behavior testing without modifying the source code.

Editing values for dynamic rendering

- **Practical example:** Change a value in a component's state to see immediately how it affects rendering.
- **Usefulness:** Practical for testing specific conditions in components (e.g. validation tests).

Monitor unnecessary Renders

•**Highlighted components**: When a component is re-rendered, it is highlighted.

•**Optimization**: Check for unnecessary re-rendering, which can be optimized via React.memo or shouldComponentUpdate.

Profiler: Performance overview

•**What is the Profiler?** A tool for measuring performance component rendering.

•**Register a session**: Click on "Start profiling". for starters.

•**Results display**: Shows rendering times for each component.

Cascading effects (useEffect, useMemo)

•**Observing cascades**: When one effect triggers another, uses DevTools to keep track of events.

•**Troubleshooting**: Reduce cascading renderings by adjusting hook dependencies.

Force component re-render

•**Force a re-render**: Use DevTools to manually trigger a **re-render**. re-render by modifying a prop or state.

•**Manual testing**: Practical for testing behaviours that are not only trigger after a re-render.

Working with Portals

•**Viewing portals in DevTools**: Even if portals are rendered outside the DOM tree, you can still view and interact with them in DevTools.

•**Event management**: See how events pass through portals and modify behavior in real time.

05

React Profiler

What is the React Profiler?

The React Profiler is an API for measuring the rendering performance of a React application. It helps you identify which parts of the application take the longest to render and why.

- The Profiler is useful for :
- Optimizing performance
- Analyze components that come back too often
- Detect performance problems in complex applications

Basic Profiler syntax

The Profiler component surrounds the components you want to monitor, with two main props: id and onRender.

```
<Profiler id="MyComponent" onRender={callbackFunction}>
  <MyComponent />
</Profiler>
```

onRender callback function

The onRender function receives several arguments:
id: Profiler identifier
phase: Shows whether the component is in the "mount" or "update" phase.
actualDuration: Actual time taken for the component to reach its destination.

```
function onRenderCallback(
  id, // id du Profiler
  phase, // "mount" ou "update"
  actualDuration // Durée réelle du rendu
) {
  console.log(`${id} phase: ${phase}, duration: ${actualDuration}ms`);
}
```

06

Overloading

React's TS namespace

Adding custom JSX elements

Enable TypeScript to recognize custom HTML elements in JSX.

```ts
// custom-elements.d.ts
import * as React from 'react';

declare module 'react' {
  namespace JSX {
    interface IntrinsicElements {
      'my-custom-element': React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>, HTMLEl
    }
  }
}
```

```tsx
const MyComponent = () => {
  return (
    <div>
      <my-custom-element>Contenu ici</my-custom-element>
    </div>
  );
};
```

Extending HTML Attributes with Custom Props

Add custom attributes to all HTML elements in JSX.

```tsx
import * as React from 'react';

declare module 'react' {
  namespace JSX {
    interface HTMLAttributes<T> extends AriaAttributes, DOMAttributes<T> {
      customAttribute?: string;
    }
  }
}
```

```tsx
const MyComponent = () => {
  return <div customAttribute="customValue">Hello, World!</div>;
};
```

Enhancing CSS properties with Custom Styles

Include custom CSS properties in inline styles.

```ts
// custom-css-properties.d.ts
import * as React from 'react';

declare module 'react' {
  interface CSSProperties {
    '--custom-color'?: string;
  }
}
```

```tsx
const MyComponent = () => {
  return (
    <div
      style={{
        '--custom-color': '#ff0000',
        color: 'var(--custom-color)',
      }}
    >
      Colored Text
    </div>
  );
};
```