React

New Hooks

19

Introduction

React 19 introduces new hooks that simplify common challenges in React development, especially around state management, form handling, and asynchronous updates.

Here are the new additions that comes with this latest update:

- **useFormState ( )**
- **useFormStatus ( )**
- **useActionState ( )**
- **useOptimistic ( )**
- **use ( )**

We will be learning how the procecesses for these hooks were handled before and how these hooks make all the differences.

useFormState ( )

Before React 19, managing form data usually required setting up a **useState** hook for each form field or a single state object with multiple fields. Custom onChange are usually set up with manual update of the state for each input field, which added complexity to the code.

With the **useFormstate()** hook in React 19, form states can be managed more cleanly, as it keeps track of all form inputs and updates them automatically when fields change.
It simplifies the tracking and updating of form field values by consolidating form data into a **single state object**.

This hook allows for real-time tracking of input values without needing separate **useState** handlers for each field. It also helps keep the form state in sync, especially when dealing with complex forms.

For example: older way of handling form field states

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

function ProfileForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  const handleSubmit = async (e) => {
    e.preventDefault();
    await fetch('/api/profile', { method: 'POST',
    body: JSON.stringify({ name, email }) });
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        name="name"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Name"
      />
      <input
        name="email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
        placeholder="Email"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

The form's name and email fields are tracked separately, requiring unique useState calls.

For example: with useFormState() hook

```jsx
import { useFormState } from 'react';

function ProfileForm() {
  const [formData, setFormData] = useFormState({
    name: '',
    email: '',
  });

  const handleChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const handleSubmit = () => {
    // Handle form submission with the entire formData object
  };

  return (
    <form onSubmit={handleSubmit}>
      <input name="name" value={formData.name} onChange={handleChange} />
      <input name="email" value={formData.email} onChange={handleChange} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

All form fields managed in one state object.

useFormStatus ( )

Before React 19, form states are usually managed by setting up multiple state variables (e.g., loading, error, success, submitting, submitted, idle) to handle form status and submission states.

This often led to complex code to track and render the form state, such as disabling the submit button during submission or showing loading indicators.

In React 19, the useFormStatus() hook provides a more streamlined way to manage these statuses. It lets easy tracking of whether a form is submitting, submitted, idle, or has encountered an error.

This makes managing the visual feedback of form interactions (e.g., disabling buttons or showing loading spinners) more straightforward.

For example: older way of handling form submission

```jsx
const UserProfile = () => {
  const [name, setName] = useState("");
  const [pending, setPending] = useState(false);

  const handleSubmit = async (event) => {
    event.preventDefault();
    setPending(true);
    await fetch('/api/profile', { method: 'POST',
    body: JSON.stringify(formData) });
    setPending(false);
    setName("");
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        placeholder="Enter your name"
        onChange={(e) => setName(e.target.value)}
      />
      <button type="submit">Submit</button>
      {pending && <p>Submitting {name} ... </p>}
    </form>
  );
};
```

Form submission status being handled using the useState hook.

Status update in the UI

Example: with useFormStatus() hook

When you use **useFormStatus**, you receive an object with properties like **pending**, **data**, **method**, and **action**. Each of these properties helps manage and respond to the form's state, improving the UX by allowing you to show loading indicators, success messages, or error handling.

- **pending**: A boolean that is true when the form is in the process of submitting and false otherwise.
- **data**: Contains any response data from the form submission.
- **method**: Indicates the HTTP method used for the form (like POST or GET).
- **action**: The URL or endpoint to which the form is being submitted.

In this example, we will use **useFormStatus** to disable the submit button while the form is being submitted, show a loading indicator during submission, and display a success or error message once the submission is complete.

```jsx
import { useFormStatus } from 'react';

function LoginForm() {
  const { pending, data, method, action } = useFormStatus();

  const handleSubmit = async (event) => {
    event.preventDefault();
    // Simulate form submission with fetch or other method
  };

  return (
    <form method="POST" action="/login" onSubmit={handleSubmit}>
      <h3>Login</h3>
      <label>
        Username:
        <input name="username" type="text" required />
      </label>
      <label>
        Password:
        <input name="password" type="password" required />
      </label>
      <button type="submit" disabled={pending}>
        {pending ? 'Logging in...' : 'Login'}
      </button>

      {/* Display success message if data is available */}
      {data && <p>Welcome back, {data.username}!</p>}

      {/* Display form metadata for debugging purposes */}
      <p>Form Method: {method}</p>
      <p>Form Action: {action}</p>
    </form>
  );
}
```

Form submission ststaus handled using the useFormStatus hook

In this example:

- The **pending** property is used to **disable** the submit button while the form is in the middle of submission. When **pending** is **true**, it shows a loading message, providing feedback to the user that the form is being processed. Once the form has
- submitted successfully, **data** will hold any response information returned from the submission. Here, if **data** is populated (e.g., it has a username field), we display a welcome message to the user. **method** and **action** provide
- useful **metadata** about the form. In this example, they are shown as debugging information to indicate the **HTTP** method and action **URL** being used.

useActionState ( )

In earlier versions of React, handling **asynchronous action states** often required using multiple **useState** variables to track the **status** of each operation:

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

function SubmitButton() {
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [error, setError] = useState(null);
  const [success, setSuccess] = useState(false);

  const handleSubmit = async () => {
    setIsSubmitting(true);
    setError(null);
    setSuccess(false);
    try {
      await fetch('/api/submit', { method: 'POST' });
      setSuccess(true);
    } catch (err) {
      setError('Submission failed');
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <div>
      <button onClick={handleSubmit} disabled={isSubmitting}>
        {isSubmitting ? 'Submitting ... ' : 'Submit'}
      </button>
      {error && <p>{error}</p>}
      {success && <p>Success!</p>}
    </div>
  );
}
```

You need separate state variables (isSubmitting, error, success) to represent each possible state of the action.

The code could becomes verbose and harder to manage, especially as more action states are added or if multiple actions are performed in the component.

The **useActionState** is a generalized hook designed to manage the lifecycle of these **actions** or **tasks** in a more streamlined way, by providing a standardized way to manage and respond to each stage of the operations.

It helps maintain a clear, controlled flow in UI updates, especially when actions need to track multiple statuses like "**idle**," "**loading**," "**success**," or "**error**."

Using our example above, the **useActionState** hook will help simplify this pattern by encapsulating the various states of an action into a **single hook**, allowing you to handle the loading, success, and error states more concisely.

Updating the former example to use **useActionState** hook:

```jsx
import { useActionState } from 'react';

function SubmitButton() {
  const [submitStatus, performSubmit] = useActionState(async () => {
    await fetch('/api/submit', { method: 'POST' });
  });

  return (
    <div>
      <button onClick={performSubmit} disabled={submitStatus.isLoading}>
        {submitStatus.isLoading ? 'Submitting ... ' : 'Submit'}
      </button>
      {submitStatus.error && <p>{submitStatus.error.message}</p>}
      {submitStatus.isSuccess && <p>Success!</p>}
    </div>
  );
}
```

Form submission statuses encapsulated into a single hook

With useActionState:

- **submitStatus** is an object that contains properties like **isLoading**, **isSuccess**, and **error**, which you can use to conditionally render UI based on the action’s current state.
- **performSubmit** is the function to invoke the action, which automatically updates submitStatus. There’s no need for additional state variables or manual error handling in the UI logic, reducing code complexity and making it more readable.

useOptimistic ( )

Before React 19, to create optimistic UI updates, where changes appear instantly **before** server confirmation, we often used a mix of state variables and asynchronous functions.

For instance, we'd set a state variable to indicate an immediate UI update, then send the API request and update the state again based on the response.

This approach was functional but required a lot of boilerplate code and error handling to ensure consistency between UI and server state.

in React 19, the **useOptimistic()** hook simplifies this by providing an easy way to handle optimistic updates. This hook allows you to update the UI **immediately**, even as the actual server confirmation is **pending**.
If the server request fails, **useOptimistic()** can be configured to **revert** the state, thus providing a smooth experience with minimal code.

For example: older way of handling optimistic update

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

function LikeButton({ postId }) {
  const [isLiked, setIsLiked] = useState(false);
  const [error, setError] = useState(null);

  const handleLike = async () => {
    // Update state immediately
    setIsLiked(!isLiked);

    try {
      // Make the API call
      await fetch(`/api/posts/${postId}/like`, { method: 'POST' });
    } catch (err) {
      // Revert state if an error occurs
      setIsLiked(isLiked);
      setError('Failed to update like status');
    }
  };

  return (
    <div>
      <button onClick={handleLike}>
        {isLiked ? 'Unlike' : 'Like'}
      </button>
      {error && <p>{error}</p>}
    </div>
  );
}
```

States used for setting optimistic updates

Optimistic state update

If an error occurs during the API call, it reverts isLiked to the original state.

The component immediately updates the isLiked state optimistically.

With **useOptimistic** hook: simplifies this pattern by automatically providing an optimistic state that can be **toggled** without additional error-handling code.

```jsx
import { useOptimistic } from 'react';

function LikeButton({ postId }) {
  const [isLiked, toggleLike] = useOptimistic(false);

  const handleLike = async () => {
    toggleLike(!isLiked);  // Toggle optimistic state
    await fetch(`/api/posts/${postId}/like`, { method: 'POST' });
  };

  return (
    <button onClick={handleLike}>
      {isLiked ? 'Unlike' : 'Like'}
    </button>
  );
}
```

- The **toggleLike** function immediately updates the UI, even while waiting for the server response.

useOptimistic makes optimistic updates less error-prone by allowing UI changes with minimal code, enhancing readability and **reducing boilerplate code for error handling**.

use ( )

In earlier versions of React, handling **asynchronous** code, such as data fetching, within components typically involved **useEffect** and **useState**.

This approach often led to complex, nested logic, especially if we are to manage loading and error states.

The new **use()** hook allows you to directly await a promise within components, reducing the need for additional **useEffect** and **useState** code.

This makes it easier to fetch data in a synchronous style, leading to cleaner and more readable code.

Example: asynchronous data fetching using useEffect and useState hooks

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

function UserProfile() {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const response = await fetch('/api/user');
        const data = await response.json();
        setUser(data);
      } finally {
        setLoading(false);
      }
    };
    fetchUser();
  }, []);

  if (loading) return <p>Loading...</p>;

  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.email}</p>
    </div>
  );
}
```

The **useEffect** hook is used for running the fetch request on component mount.

loading state management is necessary to handle the UI during data fetching.

Example: using the new use() hook for asynchronous fetching

```jsx
import { use } from 'react';

async function fetchUser() {
  const response = await fetch('/api/user');
  return response.json();
}

function UserProfile() {
  const user = use(fetchUser());  // Directly await the data

  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.email}</p>
    </div>
  );
}
```

With use():

- Async calls are handled directly in the component body, reducing the need for useEffect and additional state variables.
- This approach streamlines data fetching, reduces boilerplate code, and is especially useful for server components or Suspense-based apps.

Use Cases for these new hooks:

- **useFormState()**: Ideal for applications that require dynamic forms with multiple fields, such as registration or profile forms. useFormState() reduces the setup required to manage input data and lets developers focus more on validation and form submission.
- **useFormStatus()**: Perfect for user-intensive applications where forms are frequently submitted, such as contact forms, checkout pages, or surveys. This simplifies tracking the form's state and handling any potential submission errors.
- **useOptimistic()**: Ideal for applications with real-time interactions, such as social media apps where users expect immediate feedback on actions like likes, votes, or toggling a setting.

- **useActionState()**: A use case for useActionState is managing the status of actions like adding an item to a shopping cart in an e-commerce app or submitting a form. This hook can track whether the action is pending, resolved, or rejected, allowing for real-time feedback, such as showing a loading state, confirming a successful addition, or displaying an error message if the action fails.
- **use()**: Particularly useful for components that need data from asynchronous sources, like user profiles or content from an API. It eliminates the need for useEffect and separate loading states, streamlining asynchronous operations.

Follow_for_more

Somesh_Bhatnagar

Follow_for_more

Somesh_Bhatnagar