What Is the useOptimistic Hook, and How Does It Work?

freecodecamp.org/learn/front-end-development-libraries-v9/lecture-working-with-data-fetching-and-memoization-in-react/what-is-the-useoptimistic-hook-and-how-does-it-work

Recent versions of React introduced server components and server actions to shift some of the rendering and logic responsibilities to the server.

Along with those updates, React added a new hook called `useOptimistic` to keep UIs responsive while waiting for an async action to complete in the background.

While this is often used for fetching data from a server, it's not limited to that. The hook is generally useful for handling async operations, ensuring the UI remains smooth and interactive while the action runs.

Let's take a look at what the `useOptimistic` hook is and how it contributes to making snappy and responsive UIs.

The `useOptimistic` hook helps manage "optimistic updates" in the UI, a strategy in which you provide immediate updates to the UI based on the expected outcome of an action, like waiting for a server response.

Here's the basic syntax of the `useOptimistic` hook:

```
const [optimisticState, addOptimistic] = useOptimistic(actualState, updateFunction);
```

- `optimisticState` is the temporary state that updates right away for a better user experience.

- `addOptimistic` is the function that applies the optimistic update before the actual state changes.
- `actualState` is the real state value that comes from the result of an action, like fetching data from a server.
- `updateFunction` is the function that determines how the optimistic state should update when called.

At first glance, it might seem like the `useOptimistic` hook is just another way to handle loading states in React. But it's more than that.

A loading state controls whether you see a spinner, message, or some other indicator in the UI while something happens in the background.

However, the `useOptimistic` hook updates the UI instantaneously based on an expected outcome, even before you, say, make a call to an API. This hook gives you a chance to show a loading indicator or message, handle potential errors gracefully, and show instant feedback to make the UI feel snappy.

This will become clearer as we go through some examples showing how the `useOptimistic` hook works.

Here's an action that simulates saving a task to a server. It returns the task after a 1 second delay, as it could happen with a real-world API request:

```
export async function saveTask(task) {
  await new Promise((res) => setTimeout(res, 1000));

  return task;
}
```

Here's the code that sets up the `useOptimistic` hook by importing and initializing it, with an `handleSubmit` function that sends an input to the action:

```
"use client";

import { useOptimistic } from "react";

export default function TaskList({ tasks, addTask }) {
  const [optimisticTasks, addOptimisticTask] = useOptimistic(
    tasks,
    (state, newTask) => [...state, { text: newTask, pending: true }]
  );

  async function handleSubmit(e) {
    e.preventDefault();
    const formData = new FormData(e.target);

    addOptimisticTask(formData.get("task"));

    addTask(formData);
    e.target.reset();
  }

  return <>{/* UI */}</>;
}
```

In the code, the `useOptimistic` hook keeps a temporary list of tasks that updates immediately when you add a new task.

The line, `(state, newTask) => [...state, { text: newTask, pending: true }]` ensures that a new task appears with a pending status even before the server confirms something is coming from the form.

When the form is submitted, the `handleSubmit` function extracts the task and adds it "optimistically" with the `addOptimisticTask` parameter. Then `addTask` is passed as a prop which sends the task to the server. Finally, the form is reset by calling `e.target.reset()`.

Here's the `TaskList` component:

```
"use client";
import { useOptimistic, startTransition } from "react";

export default function TaskList({ tasks, addTask }) {
  const [optimisticTasks, addOptimisticTask] = useOptimistic(
    tasks,
    (state, newTask) => [...state, { text: newTask, pending: true }]
  );

  async function handleSubmit(e) {
    e.preventDefault();
    const formData = new FormData(e.target);

    startTransition(() => {
      addOptimisticTask(formData.get("task"));
    });

    addTask(formData);
    e.target.reset();
  }

  return (
    <div className="max-w-md mx-auto p-4">
      <h3 className="text-xl font-medium mb-3">Tasks</h3>

      <ul className="space-y-2 mb-4">
        {optimisticTasks.map((task, index) => (
          <li key={index} className="p-2 border-b">
            {task.text}
            {task.pending && (
              <small className="ml-2 text-gray-500 text-sm">
                Adding Task...
              </small>
            )}
          </li>
        ))}
      </ul>

      <form onSubmit={handleSubmit} className="flex gap-2">
        <input
          type="text"
          name="task"
          placeholder="Type in a task..."
          className="flex-1 p-2 border"
        />
        <button type="submit" className="bg-gray-200 px-3 py-2 cursor-pointer">
          Submit
        </button>
      </form>
    </div>
  );
}
```

Here, we are looping through the `optimisticTask` parameter to display the task. When `task.pending` is `true`, the text `Adding Task...` is displayed next to the task, confirming that the task has been added optimistically before the server confirms it.

Here's the `Task` component that manages the state for the form. It calls the `saveTask` function from the action so it can add the task, and appends the new task once it is received by the server:

```
"use client";

import { useState } from "react";
import TaskList from "./TaskList";
import { saveTask } from "./actions";

export default function Tasks() {
  const [tasks, setTasks] = useState([]);

  async function addTask(formData) {
    const newTaskText = formData.get("task");

    const savedTask = await saveTask(newTaskText);
    setTasks((prev) => [...prev, { text: savedTask, pending: false }]);
  }

  return <TaskList tasks={tasks} addTask={addTask} />;
}
```

This ensures snappy UI updates by showing instant feedback instead of waiting for a response. Once the task is saved, the `pending` property is removed, and the final task list updates accordingly.

In the UI, there are two things happening that are not supposed to happen. First, you can't see the `Adding Task...` text since it appears and disappears too quickly. Next, there's an error occurring after adding the task.

There are two things we need to do to address those issues.

First, we need to import `startTransition` from React and use it to wrap the line `addOptimisticTask(formData.get('task'))`:

```
startTransition(() => {
  addOptimisticTask(formData.get("task"));
});
```

Second, we need to make the `Adding Task...` text visible for some time before it goes away.

To do this, we can modify the `addTask` function with a pending state and simulate a delay of a few seconds before marking the task as completed. `setTimeout()` is ideal for this:

```
async function addTask(formData) {
  const newTaskText = formData.get("task");

  // Add an optimistic task with a pending state
  const tempTask = { id: Date.now(), text: newTaskText, pending: true };
  setTasks((prev) => [...prev, tempTask]);

  // Simulate a 3 seconds delay before marking the task as completed
  setTimeout(async () => {
    const savedTask = await saveTask(newTaskText);

    setTasks((prev) =>
      prev.map((task) =>
        task.id === tempTask.id
          ? { ...task, text: savedTask, pending: false }
          : task
      )
    );
  }, 3000);
}
```

And once you do that, everything works fine.

Questions

What is the purpose of the `useOptimistic` hook?

It allows components to fetch data from the server before rendering the UI.

It helps manage optimistic updates by updating the UI immediately while waiting for an async operation, like a server response.

It enables automatic error handling and rollback for failed API requests in React applications.

It optimizes state updates by batching them together to improve performance.

How is the `useOptimistic` hook different from a loading state?

A loading state shows UI feedback while waiting for a response, whereas `useOptimistic` updates the UI immediately based on an expected outcome.

A loading state modifies server data instantly while `useOptimistic` only updates the client UI.

The `useOptimistic` hook is used for handling errors, whereas a loading state is only for showing spinners.

Both are the same, but `useOptimistic` provides automatic retries for failed requests.

What does `addOptimistic` do in the `useOptimistic` hook syntax below?

```
const [optimisticState, addOptimistic] =
useOptimistic(actualState, updateFunction);
```

It applies the optimistic update before the actual state changes, providing a smoother user experience.

It fetches the real state from the server and updates the UI accordingly.

It replaces the actual state with a temporary state after receiving a server response.

It validates server data before applying the optimistic update to the UI.