use() and data fetching

The `use()` hook significantly improves the data fetching workflow in React applications.

Let’s see how it changes and enhances the way we handle data fetching.

Traditionally, data fetching in React components, if not using a separate library like React Query or SWR or others, often involved fetching data in a `useEffect`

```
import React, { useState, useEffect } from 'react'

interface Todo {
  id: number
  text: string
}

function App() {
  const [data, setData] = useState<{ todos: Todo[] }>({ todos: [] })

  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('/api/todos')
        if (!response.ok) {
          throw new Error('Network response was not ok')
        }
        const result = await response.json()
        setData(result)
      } catch (error) {
        console.error('Error fetching data:', error)
      }
    }

    fetchData()
  }, [])
```

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

export default App
```

This traditional approach to data fetching in React components has several drawbacks when compared to React Server Components: the data fetching occurs on the client side, which can lead to slower initial page loads and poorer performance, especially on slower devices or networks. If multiple components need to fetch data, it can result in a series of dependent requests, creating a “waterfall” effect that further slows down the application. The code for data fetching and state management is included in the JavaScript bundle sent to the client, increasing the overall bundle size. And as you can see, managing loading states, errors, and data synchronization adds complexity to the client-side code.

React Server Components address many of these issues by moving data fetching to the server, reducing client-side JavaScript, and allowing for more efficient rendering and caching strategies.

With React 19 and React Server Components, in a Next.js 15 context, here’s what we can do.

We define the type externally since it will be used by 2 components

```
export interface Todo {
  id: number
  text: string
  // Add other properties if needed
}
```

The server component, which could be in `src/app/todos/page.tsx`, loads a client component in a suspense, so we can fallback to a loading state until the data is streamed to the client component from the RSC:

```
import { Suspense, use } from 'react'
import { Todos } from './todos'
import { sql } from '@vercel/postgres'
import type { Todo } from '../types.ts'

async function fetchData(): Promise<Todo[]> {
  const { rows } = await sql`SELECT * FROM todos`
  // test wait 3 seconds, to see the fallback working locally
  // await new Promise((resolve) => setTimeout(resolve, 3000))
  return rows as Todo[]
}

export default async function Home() {
  return (
    <div className='text-6xl p-20'>
      <Suspense fallback={<div>...</div>}>
        <Todos promise={fetchData()} />
      </Suspense>
    </div>
  )
}
```

```
'use client'

import { use } from 'react'
import type { Todo } from '../types.ts'

export function Todos({ promise }: { promise: Promise<Todo[]> }) {
  const rows = use(promise)

  return (
    <div>
      {rows.map((row) => (
        <li key={row.id}>{row.text}</li>
      ))}
```

```
      </div>
    )
  }
```

Advantages of this approach:

1. data fetching starts immediately when the server component is rendered on the server, you don’t have to wait for the browser to render the client-side component and then fetch the data as we did before
2. the client-side JavaScript bundle is smaller
3. we don’t have to implement (and protect) an API route just to fetch data
4. the client component is just concerned with visualization, not data fetching
5. we can display a shell instantly while we wait for the data to arrive

Written on Sep 18, 2024