Server and Client Composition Patterns

When building React applications, you will need to consider what parts of your application should be rendered on the server or the client. This page covers some recommended composition patterns when using Server and Client Components.

Here's a quick summary of the different use cases for Server and Client Components:

What do you need to do?	Server Component	Client Component
Fetch data	✓	✗
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	✗	✓
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use React Class components ↗	✗	✓

Server Component Patterns

Before opting into client-side rendering, you may wish to do some work on the server like fetching data, or accessing your database or backend services.

Here are some common patterns when working with Server Components:

Sharing data between components

When fetching data on the server, there may be cases where you need to share data across different components. For example, you may have a layout and a page that depend on the same data.

Instead of using React Context↗ (which is not available on the server) or passing data as props, you can use `fetch` or React's `cache` function to fetch the same data in the components that need it, without worrying about making duplicate requests for the same data. This is because React extends `fetch` to automatically memoize data requests, and the `cache` function can be used when `fetch` is not available.

Learn more about memoization in React.

Keeping Server-only Code out of the Client Environment

Since JavaScript modules can be shared between both Server and Client Components modules, it's possible for code that was only ever intended to be run on the server to sneak its way into the client.

For example, take the following data-fetching function:

lib/data.ts — TypeScript

```ts
export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```

At first glance, it appears that `getData` works on both the server and the client. However, this function contains an `API_KEY`, written with the intention that it would only ever be executed on the server.

Since the environment variable `API_KEY` is not prefixed with `NEXT_PUBLIC`, it's a private variable that can only be accessed on the server. To prevent your environment variables from being leaked to the client, Next.js replaces private environment variables with an empty string.

As a result, even though `getData()` can be imported and executed on the client, it won't work as expected. And while making the variable public would make the function work on the client, you may not want to expose sensitive information to the client.

To prevent this sort of unintended client usage of server code, we can use the `server-only` package to give other developers a build-time error if they ever accidentally import one of these modules into a Client Component.

To use `server-only`, first install the package:

>_ Terminal

```
npm install server-only
```

Then import the package into any module that contains server-only code:

lib/data.js

```
import 'server-only'

export async function getData() {
  const res = await fetch('https://external-service.com/data', {
    headers: {
      authorization: process.env.API_KEY,
    },
  })

  return res.json()
}
```

Now, any Client Component that imports `getData()` will receive a build-time error explaining that this module can only be used on the server.

The corresponding package `client-only` can be used to mark modules that contain client-only code – for example, code that accesses the `window` object.

Using Third-party Packages and Providers

Since Server Components are a new React feature, third-party packages and providers in the ecosystem are just beginning to add the `"use client"` directive to components that use client-only features like `useState`, `useEffect`, and `createContext`.

Today, many components from `npm` packages that use client-only features do not yet have the directive. These third-party components will work as expected within Client Components since they have the `"use client"` directive, but they won't work within Server Components.

For example, let's say you've installed the hypothetical `acme-carousel` package which has a `<Carousel />` component. This component uses `useState`, but it doesn't yet have the `"use client"` directive.

If you use `<Carousel />` within a Client Component, it will work as expected:

app/gallery.tsx — TypeScript

```tsx
'use client'

import { useState } from 'react'
import { Carousel } from 'acme-carousel'

export default function Gallery() {
  let [isOpen, setIsOpen] = useState(false)

  return (
    <div>
      <button onClick={() => setIsOpen(true)}>View pictures</button>

      {/* Works, since Carousel is used within a Client Component */}
      {isOpen && <Carousel />}
    </div>
  )
}
```

However, if you try to use it directly within a Server Component, you'll see an error:

app/page.tsx — TypeScript

```tsx
import { Carousel } from 'acme-carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/* Error: `useState` can not be used within Server Components */}
      <Carousel />
    </div>
  )
}
```

This is because Next.js doesn't know `<Carousel />` is using client-only features.

To fix this, you can wrap third-party components that rely on client-only features in your own Client Components:

app/carousel.tsx — TypeScript

```tsx
'use client'

import { Carousel } from 'acme-carousel'

export default Carousel
```

Now, you can use `<Carousel />` directly within a Server Component:

app/page.tsx — TypeScript

```tsx
import Carousel from './carousel'

export default function Page() {
  return (
    <div>
      <p>View pictures</p>

      {/*  Works, since Carousel is a Client Component */}
      <Carousel />
    </div>
  )
}
```

We don't expect you to need to wrap most third-party components since it's likely you'll be using them within Client Components. However, one exception is providers, since they rely on React state and context, and are typically needed at the root of an application. Learn more about third-party context providers below.

Using Context Providers

Context providers are typically rendered near the root of an application to share global concerns, like the current theme. Since React context ↗ is not supported in Server Components, trying to create a context at the root of your application will cause an error:

app/layout.tsx — TypeScript

```tsx
import { createContext } from 'react'

//  createContext is not supported in Server Components
export const ThemeContext = createContext({})

export default function RootLayout({ children }) {
  return (
    <html>
      <body>
        <ThemeContext.Provider value="dark">{children}</ThemeContext.Provider>
      </body>
    </html>
  )
}
```

To fix this, create your context and render its provider inside of a Client Component:

app/theme-provider.tsx — TypeScript

```tsx
'use client'

import { createContext } from 'react'

export const ThemeContext = createContext({})

export default function ThemeProvider({
  children,
}: {
  children: React.ReactNode
}) {
  return <ThemeContext.Provider value="dark">{children}</ThemeContext.Provider>
}
```

Your Server Component will now be able to directly render your provider since it's been marked as a Client Component:

app/layout.tsx TypeScript

```tsx
import ThemeProvider from './theme-provider'

export default function RootLayout({
  children,
}: {
  children: React.ReactNode
}) {
  return (
    <html>
      <body>
        <ThemeProvider>{children}</ThemeProvider>
      </body>
    </html>
  )
}
```

With the provider rendered at the root, all other Client Components throughout your app will be able to consume this context.

Good to know: You should render providers as deep as possible in the tree – notice how `ThemeProvider` only wraps `{children}` instead of the entire `<html>` document. This makes it easier for Next.js to optimize the static parts of your Server Components.

Advice for Library Authors

In a similar fashion, library authors creating packages to be consumed by other developers can use the `"use client"` directive to mark client entry points of their package. This allows users of the package to import package components directly into their Server Components without having to create a wrapping boundary.

You can optimize your package by using 'use client' deeper in the tree, allowing the imported modules to be part of the Server Component module graph.

It's worth noting some bundlers might strip out `"use client"` directives. You can find an example of how to configure esbuild to include the `"use client"` directive in the React Wrap Balancer ↗ and Vercel Analytics ↗ repositories.

Client Components

Moving Client Components Down the Tree

To reduce the Client JavaScript bundle size, we recommend moving Client Components down your component tree.

For example, you may have a Layout that has static elements (e.g. logo, links, etc) and an interactive search bar that uses state.

Instead of making the whole layout a Client Component, move the interactive logic to a Client Component (e.g. `<SearchBar />`) and keep your layout as a Server Component. This means you don't have to send all the component Javascript of the layout to the client.

app/layout.tsx — TypeScript

```tsx
// SearchBar is a Client Component
import SearchBar from './searchbar'
// Logo is a Server Component
import Logo from './logo'

// Layout is a Server Component by default
export default function Layout({ children }: { children: React.ReactNode }) {
  return (
    <>
      <nav>
        <Logo />
```

```
          <SearchBar />
        </nav>
        <main>{children}</main>
      </>
    )
  }
```

Passing props from Server to Client Components (Serialization)

If you fetch data in a Server Component, you may want to pass data down as props to Client Components. Props passed from the Server to Client Components need to be serializable ↗ by React.

If your Client Components depend on data that is not serializable, you can fetch data on the client with a third party library or on the server via a Route Handler.

Interleaving Server and Client Components

When interleaving Client and Server Components, it may be helpful to visualize your UI as a tree of components. Starting with the root layout, which is a Server Component, you can then render certain subtrees of components on the client by adding the `"use client"` directive.

Within those client subtrees, you can still nest Server Components or call Server Actions, however there are some things to keep in mind:

- During a request-response lifecycle, your code moves from the server to the client. If you need to access data or resources on the server while on the client, you'll be making a **new** request to the server - not switching back and forth.
- When a new request is made to the server, all Server Components are rendered first, including those nested inside Client Components. The rendered result (RSC Payload) will contain references to the locations of Client Components. Then, on the client, React uses the RSC Payload to reconcile Server and Client Components into a single tree.
- Since Client Components are rendered after Server Components, you cannot import a Server Component into a Client Component module (since it would require a new request back to the server). Instead, you can pass a Server Component as `props` to a Client Component. See the unsupported pattern and supported pattern sections below.

Unsupported Pattern: Importing Server Components into Client Components

The following pattern is not supported. You cannot import a Server Component into a Client Component:

app/client-component.tsx TypeScript

```tsx
'use client'

// You cannot import a Server Component into a Client Component.
import ServerComponent from './Server-Component'

export default function ClientComponent({
  children,
}: {
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>

      <ServerComponent />
    </>
  )
}
```

Supported Pattern: Passing Server Components to Client Components as Props

The following pattern is supported. You can pass Server Components as a prop to a Client Component.

A common pattern is to use the React `children` prop to create a *"slot"* in your Client Component.

In the example below, `<ClientComponent>` accepts a `children` prop:

app/client-component.tsx TypeScript

```tsx
'use client'

```

```tsx
import { useState } from 'react'

export default function ClientComponent({
  children,
}: {
  children: React.ReactNode
}) {
  const [count, setCount] = useState(0)

  return (
    <>
      <button onClick={() => setCount(count + 1)}>{count}</button>
      {children}
    </>
  )
}
```

`<ClientComponent>` doesn't know that `children` will eventually be filled in by the result of a Server Component. The only responsibility `<ClientComponent>` has is to decide **where** `children` will eventually be placed.

In a parent Server Component, you can import both the `<ClientComponent>` and `<ServerComponent>` and pass `<ServerComponent>` as a child of `<ClientComponent>`:

app/page.tsx — TypeScript

```tsx
// This pattern works:
// You can pass a Server Component as a child or prop of a
// Client Component.
import ClientComponent from './client-component'
import ServerComponent from './server-component'

// Pages in Next.js are Server Components by default
export default function Page() {
  return (
    <ClientComponent>
      <ServerComponent />
    </ClientComponent>
  )
}
```

With this approach, `<ClientComponent>` and `<ServerComponent>` are decoupled and can be rendered independently. In this case, the child `<ServerComponent>` can be rendered on the server, well before `<ClientComponent>` is rendered on the client.

Good to know:

- The pattern of "lifting content up" has been used to avoid re-rendering a nested child component when a parent component re-renders.
- You're not limited to the `children` prop. You can use any prop to pass JSX.

Previous
Client Components

Next
Edge and Node.js Runtimes