

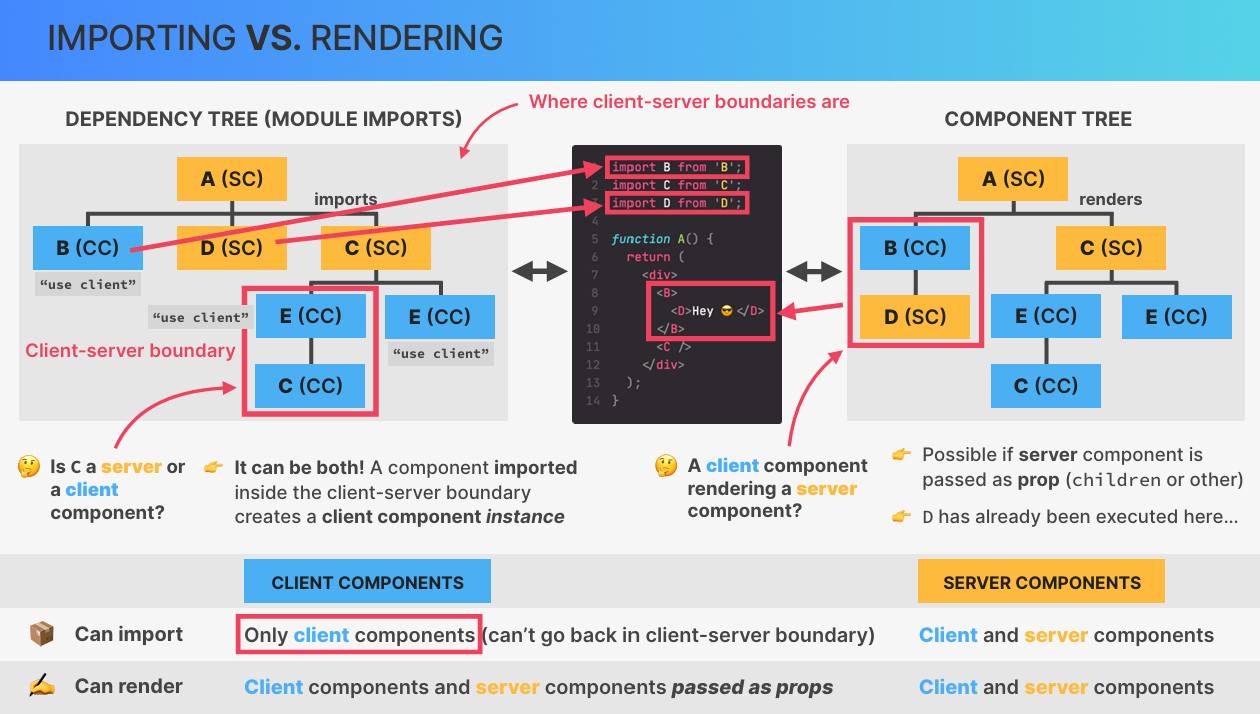
* Focus on interactions between **client** and **server** parts in applications using **React server components** (RSC) architecture.
* New paradigm blurs the traditional separation between back-end and front-end.

**Traditional Model**

* **Back-end**:
  + Runs services (e.g., a Node.js API).
  + Typically has its own code base and hosting environment.
* **Front-end**:
  + Runs applications (e.g., a React app) on the user's device.
  + Communicates with the back-end exclusively through an **API**.
* **Communication Flow**:
  + **Data Fetching**: Front-end makes GET requests to fetch JSON data.
  + **Data Mutations**: Front-end sends POST, PUT, DELETE requests to update back-end data.
* Clear, rigid boundary exists between back-end and front-end.

**React Server Components (RSC) Paradigm**

* **Component Tree Composition**:
  + Mixes **server components** (e.g., rendered in yellow) and **client components** (e.g., rendered in blue) within the same tree.
  + This interweaving pattern is termed **knitting**.
* **Flexible Boundaries**:
  + No strict separation; server and client code coexist within one code base.
  + Enables seamless switching between server and client domains.
* **Data Operations Without an API**:
  + **Fetching Data**:
    - Server components can read directly from a database and render data.
    - Alternatively, data can be passed as **props** to client components.
  + **Mutating Data**:
    - Utilizes **server actions** to perform mutations directly from client components, replacing traditional API requests



**Dependency Tree**:

* Represents the module import relationships (e.g., parent modules importing child modules).
* Establishes the client-server boundaries—client components cannot directly import server components.

**Rendering Server Components in Client Components**:

* Achieved by passing a server component as a **prop** (e.g., via the children prop).
* Example:
  + Component A imports both **client component B** **and server component D**.
  + **Server component D** **is executed on the server** **and then passed to client component B, avoiding a direct client-to-server import**.

**Dynamic Nature of Component Instances**:

* Components serve as blueprints; their instance type (server or client) depends on the context of their use.
* Without the **use client** directive, a component defaults to a **server component**.
* **When imported by a client component, an instance becomes a** **client component**.
* Example:
  + Component C, when used in a server context (imported by Component A), behaves as a server component.
  + The same Component C, when used within a client component (imported by Component E), behaves as a client component.

**1. Create TextExpander.jsx (Client Component)**

This component manages interactivity by toggling between showing a preview (first 40 words) and the full text.

javascript

CopyEdit

// src/components/TextExpander.jsx

"use client";

import { useState } from "react";

export default function TextExpander({ children }) {

const [isExpanded, setIsExpanded] = useState(false);

// Show first 40 words if not expanded

const displayText = isExpanded

? children

: children.split(" ").slice(0, 40).join(" ") + "...";

return (

<div>

<p>{displayText}</p>

<button

onClick={() => setIsExpanded(!isExpanded)}

className="px-4 py-2 bg-blue-500 text-white rounded"

>

{isExpanded ? "Show Less" : "Show More"}

</button>

</div>

);

}

**2. Use TextExpander in CabinsPage.jsx (Server Component)**

This component passes the cabin description to TextExpander.

javascript

CopyEdit

// src/app/cabins/page.jsx

import TextExpander from "@/components/TextExpander";

export default function CabinsPage() {

const description = `

Welcome to our beautiful cabins! Nestled in nature with serene views,

our cabins offer a perfect escape from the hustle and bustle. Enjoy

cozy interiors, modern amenities, and personalized service.

Experience tranquility and relaxation at its finest. Book now to

make unforgettable memories with us.

`;

return (

<div className="p-4">

<h1 className="text-3xl font-bold mb-4">Cabins</h1>

{/\* Using TextExpander Client Component \*/}

<TextExpander>

{description}

</TextExpander>

</div>

);

}

**3. Explanation of Behavior**

* **TextExpander.jsx** is a **Client Component** because it uses "use client".
* **CabinsPage.jsx** is a **Server Component** that passes data to TextExpander.
* Clicking the button toggles between showing a **preview (first 40 words)** and the **full text**.

 **Goal:** Highlight the currently active navigation link based on the URL.

 **Using usePathname Hook:**

* The **usePathname** hook, provided by **Next.js**, retrieves the current URL path.
* The hook only works in **client components**, as it is a React hook.
* **'use client' directive** is required to ensure the component is a client component.

 **Console Logging the Pathname:**

* Logging usePathname returns URL paths (e.g., /account/reservations).
* These paths match the **links' href attributes** in the navigation array.

 **Conditional Styling for Active Links:**

* Use **template literals** for conditional className styling with **TailwindCSS**.
* Example conditional styling:

javascript

CopyEdit

className={`${pathname === link.href ? 'bg-primary-900' : ''}`}

* **bg-primary-900** class highlights the active link.

 **Reason for Client-Side Hook Usage:**

* Despite being a seemingly static feature, highlighting the active link requires **React hooks**, which can only be used in client components.

**Steps to Implement Client-Server State Sharing:**

1. **Filter Component Setup**:
   * Create a **client component** for the filter (e.g., filtering cabins by guest capacity: small, medium, large).
   * Use buttons to allow users to select filter options.
2. **State in the URL**:
   * Store the filter state in the URL using **search parameters** (e.g., ?capacity=small).
   * Update the URL dynamically when a user clicks a filter button.
3. **Reading URL State on the Server**:
   * Use the searchParams prop in the **page component** to read the filter term from the URL.
   * Pass the filter term as a prop to the server component that renders the filtered data.
4. **Filtering Data**:
   * Fetch all cabins from the server.
   * Filter the cabins based on the searchParams value (e.g., small, medium, large).
   * Use conditional logic to display the filtered cabins.
5. **Dynamic Rendering**:
   * Using searchParams makes the page **dynamically rendered** (not statically generated).
   * Dynamic pages cannot use static revalidation (revalidate).
6. **Client-Side Navigation**:
   * Use the useRouter hook from next/navigation to programmatically update the URL.
   * Use router.replace to update the URL without a full page reload.
7. **Loading Feedback**:
   * Use **Suspense** to show a loading spinner while data is being fetched.
   * Add a unique key to the Suspense component to ensure the fallback re-renders during transitions.
8. **Active Filter Styling**:
   * Highlight the currently selected filter button by comparing the activeFilter (from searchParams) with the button’s filter term.
   * Abstract the filter button into a reusable component to avoid repetitive code.

**Important Code Snippets:**

1. **Reading searchParams in a Page Component**:

javascript

Copy

export default function Page({ searchParams }) {

const filter = searchParams.capacity || 'all';

console.log(filter); // Logs the current filter term

}

1. **Updating the URL on Button Click**:

javascript

Copy

const handleFilter = (filterTerm) => {

const params = new URLSearchParams(window.location.search);

params.set('capacity', filterTerm);

router.replace(`${pathname}?${params.toString()}`, { scroll: false });

};

1. **Filtering Data Based on searchParams**:

javascript

Copy

let displayedCabins;

if (filter === 'small') {

displayedCabins = cabins.filter(cabin => cabin.maxCapacity <= 3);

} else if (filter === 'medium') {

displayedCabins = cabins.filter(cabin => cabin.maxCapacity >= 4 && cabin.maxCapacity <= 7);

} else if (filter === 'large') {

displayedCabins = cabins.filter(cabin => cabin.maxCapacity >= 8);

} else {

displayedCabins = cabins;

}

1. **Suspense with Unique Key**:

javascript

Copy

<Suspense key={filter} fallback={<LoadingSpinner />}>

<CabinList cabins={displayedCabins} />

</Suspense>

1. **Reusable Filter Button Component**:

javascript

Copy

function FilterButton({ filter, handleFilter, activeFilter, children }) {

return (

<button

onClick={() => handleFilter(filter)}

style={{ backgroundColor: filter === activeFilter ? 'primary-700' : 'transparent' }}

>

{children}

</button>

);

}

**Key Takeaways:**

* **URL as State Storage**: The URL is a simple and effective way to share state between client and server, making pages shareable and bookmarkable.
* **Dynamic Rendering**: Using searchParams forces the page to be dynamically rendered, which impacts performance and caching.
* **Suspense and Transitions**: Next.js wraps page navigations in transitions, which affects how Suspense fallbacks are displayed. Use a unique key to ensure the fallback re-renders.
* **Client-Side Navigation**: Use router.replace to update the URL without a full page reload, enabling smooth client-side navigation.
* **Reusable Components**: Abstract repetitive UI elements (e.g., filter buttons) into reusable components for cleaner code.

[End of Notes]

**Notes on Reservation Section for Cabin Page**

* **Overview**: Implementing a reservation feature on the individual cabin page with data fetching strategies for optimal user experience.
* **Components**:
  + **DateSelector**: Displays a date picker for selecting booking dates.
  + **ReservationForm**: Collects user information, such as name and number of nights.
  + **use client directive** added to enable client-side rendering for the date picker.

**🗂 Data Requirements**

* **From Supabase API**:
  1. **Settings**: Minimum and maximum booking length.
  2. **Booked Dates**: To block unavailable dates.
* **From Cabin Data**:
  1. **Maximum Capacity** (for ReservationForm).
  2. **Prices** (for DateSelector).

**🛑 Why Not Fetch Data Inside Components?**

* **Components (DateSelector and ReservationForm) are client components.**
* Recommended to **fetch data server-side and pass as props** for better performance and SEO.

**🚀 Fetching Data Efficiently**

* **Initial Approach (Sequential Fetching):**
  + Used await getSettings() and await getBookedDatesByCabinId() **sequentially**, causing a **waterfall delay**.
  + Slow queries blocked each other, increasing page load time.
* **Improved Approach (Parallel Fetching with Promise.all):**

javascript

CopyEdit

const [settings, bookedDates] = await Promise.all([

getSettings(),

getBookedDatesByCabinId(params.cabinID)

]);

* + Fetches multiple data sources concurrently.
  + **Faster than sequential fetching**, but still limited by the slowest request.

**💡 Best Strategy: Component-Based Data Fetching with Streaming**

* **Create a parent Reservation.js component** to:
  + Fetch only necessary data for DateSelector and ReservationForm individually.
  + Stream components as soon as they are ready, **improving perceived load time**.
* **Refactoring Steps:**
  + Move DateSelector and ReservationForm into a new component (Reservation.js).
  + Each component fetches its required data independently.
  + Render components progressively using streaming.

**1. Create Reservation Context (ReservationContext.js)**

javascript

CopyEdit

"use client";

import { createContext, useContext, useState } from "react";

// Create context

const ReservationContext = createContext();

// Custom hook to use the context

export const useReservation = () => {

const context = useContext(ReservationContext);

if (!context) {

throw new Error("useReservation must be used within a ReservationProvider");

}

return context;

};

// Context provider component

export const ReservationProvider = ({ children }) => {

const [range, setRange] = useState({ from: undefined, to: undefined });

return (

<ReservationContext.Provider value={{ range, setRange }}>

{children}

</ReservationContext.Provider>

);

};

**2. Wrap the App with ReservationProvider (e.g., in layout.js or page.js)**

javascript

CopyEdit

import { ReservationProvider } from "@/context/ReservationContext";

export default function RootLayout({ children }) {

return (

<ReservationProvider>

{children}

</ReservationProvider>

);

}

**3. Implement Day Picker (DayPickerComponent.js)**

javascript

CopyEdit

"use client";

import { useReservation } from "@/context/ReservationContext";

import { DateRange } from "react-day-picker";

export default function DayPickerComponent() {

const { range, setRange } = useReservation();

return (

<div>

<DayPicker

mode="range"

selected={range}

onSelect={setRange}

numberOfMonths={2}

disabled={{ before: new Date() }} // Prevents selecting past dates

/>

<p>Selected from: {range.from?.toLocaleDateString()}</p>

<p>Selected to: {range.to?.toLocaleDateString()}</p>

</div>

);

}

**4. Access Dates in Another Component (e.g., ReservationForm.js)**

javascript

CopyEdit

"use client";

import { useReservation } from "@/context/ReservationContext";

export default function ReservationForm() {

const { range } = useReservation();

return (

<div>

<h2>Reservation Details</h2>

<p>Check-in: {range.from?.toLocaleDateString() || "Not selected"}</p>

<p>Check-out: {range.to?.toLocaleDateString() || "Not selected"}</p>

<button disabled={!range.from || !range.to}>Confirm Booking</button>

</div>

);

}

**How It Works**

* ReservationContext.js provides global state using **React Context API**.
* ReservationProvider wraps the entire app to make the date selection accessible across components.
* DayPickerComponent allows users to pick a date range and updates state via useReservation.
* ReservationForm reads the selected dates from the global context.

This setup ensures that the date selection state is **shared across all components** without prop drilling

**Notes on Route Handlers in Next.js**

* **Route Handlers** allow creating API endpoints directly in Next.js.
* **API Endpoints** were more crucial in the **Pages Router** but are now less important due to **Server Actions** in the **React Server Components architecture**.
* Still useful for handling custom API logic, such as aggregating data securely.

**Creating a Route Handler**

* Place a route.js file inside any folder **without a page.js file**.
* When a request is made to the route, it executes the handler **instead of returning HTML**.
* Example:
  + /api → create api/route.js.
  + Define **HTTP methods** (GET, POST, PUT, etc.) as exported functions.

javascript

CopyEdit

export async function GET() {

return Response.json({ test: "test" });

}

* Navigating to /api returns { "test": "test" } in JSON format.

**Handling Dynamic Routes**

* Dynamic routes can be created using **folder naming conventions**.
* Example: /api/cabins/[cabinId]
  + Create api/cabins/[cabinId]/route.js.

javascript

CopyEdit

export async function GET(request, { params }) {

const { cabinId } = params;

return Response.json({ cabinId });

}

* params.cabinId extracts the ID from the URL (/api/cabins/90).

**Fetching Data in API Endpoints**

* Fetching cabin details and booked dates for an **affiliate API endpoint**.
* Use **Promise.all** to fetch both data points in parallel.
* **Error handling** is required since **Next.js error boundaries don’t work in APIs**.

javascript

CopyEdit

export async function GET(request, { params }) {

try {

const { cabinId } = params;

const [cabin, bookedDates] = await Promise.all([

getCabin(cabinId),

getBookedDatesByCabin(cabinId),

]);

return Response.json({ cabin, bookedDates });

} catch (error) {

return Response.json({ message: "Cabin not found" }, { status: 404 });

}

}

* Returns cabin details and **future booked dates**.
* If the **cabin doesn’t exist**, responds with { "message": "Cabin not found" }.

**Why Use API Endpoints in Next.js?**

* Avoid exposing **Supabase API endpoints** directly.
* Aggregate data from **multiple sources** securely.
* Keep **API keys hidden** while providing structured access.