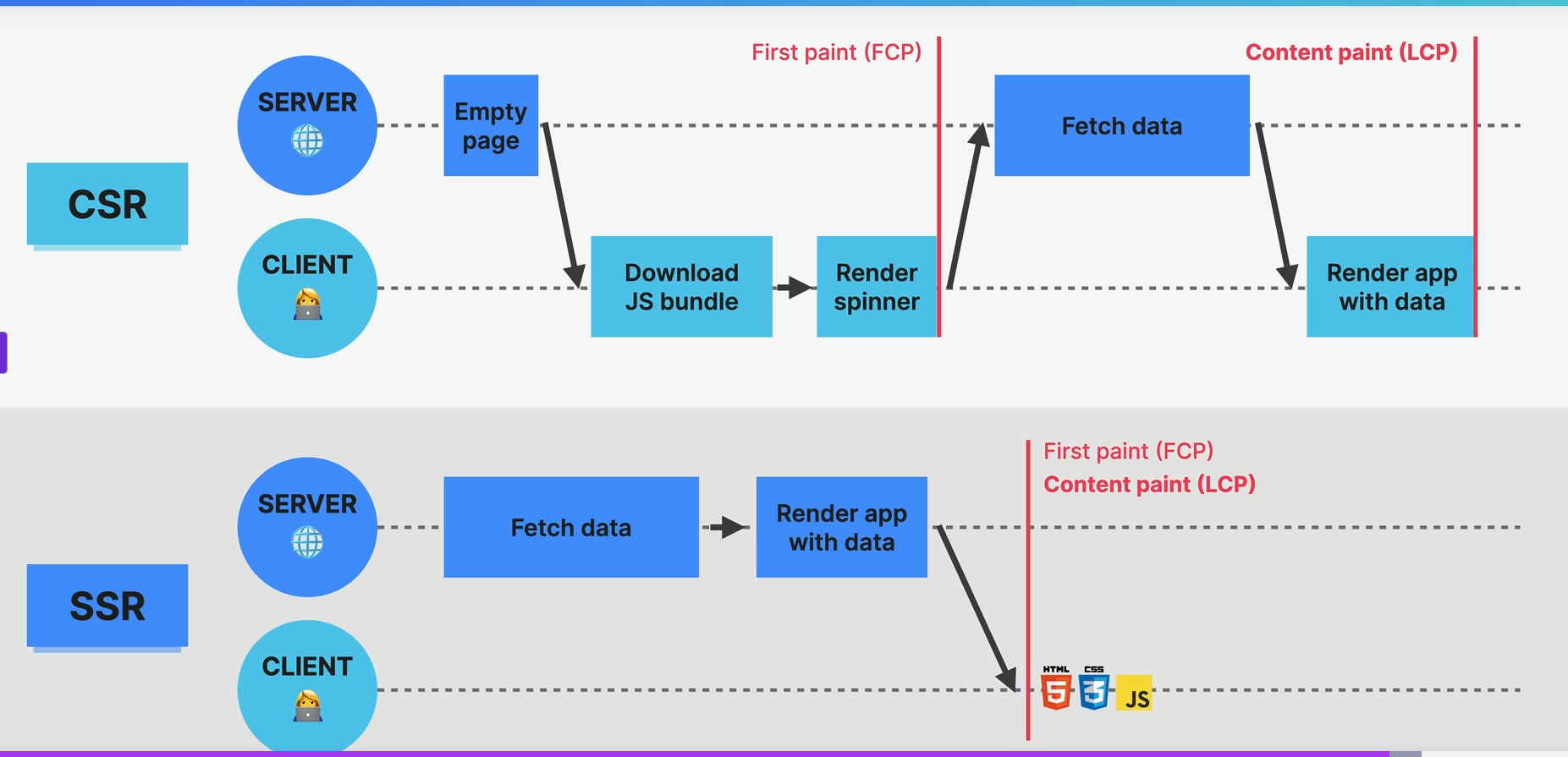
NEXT JS

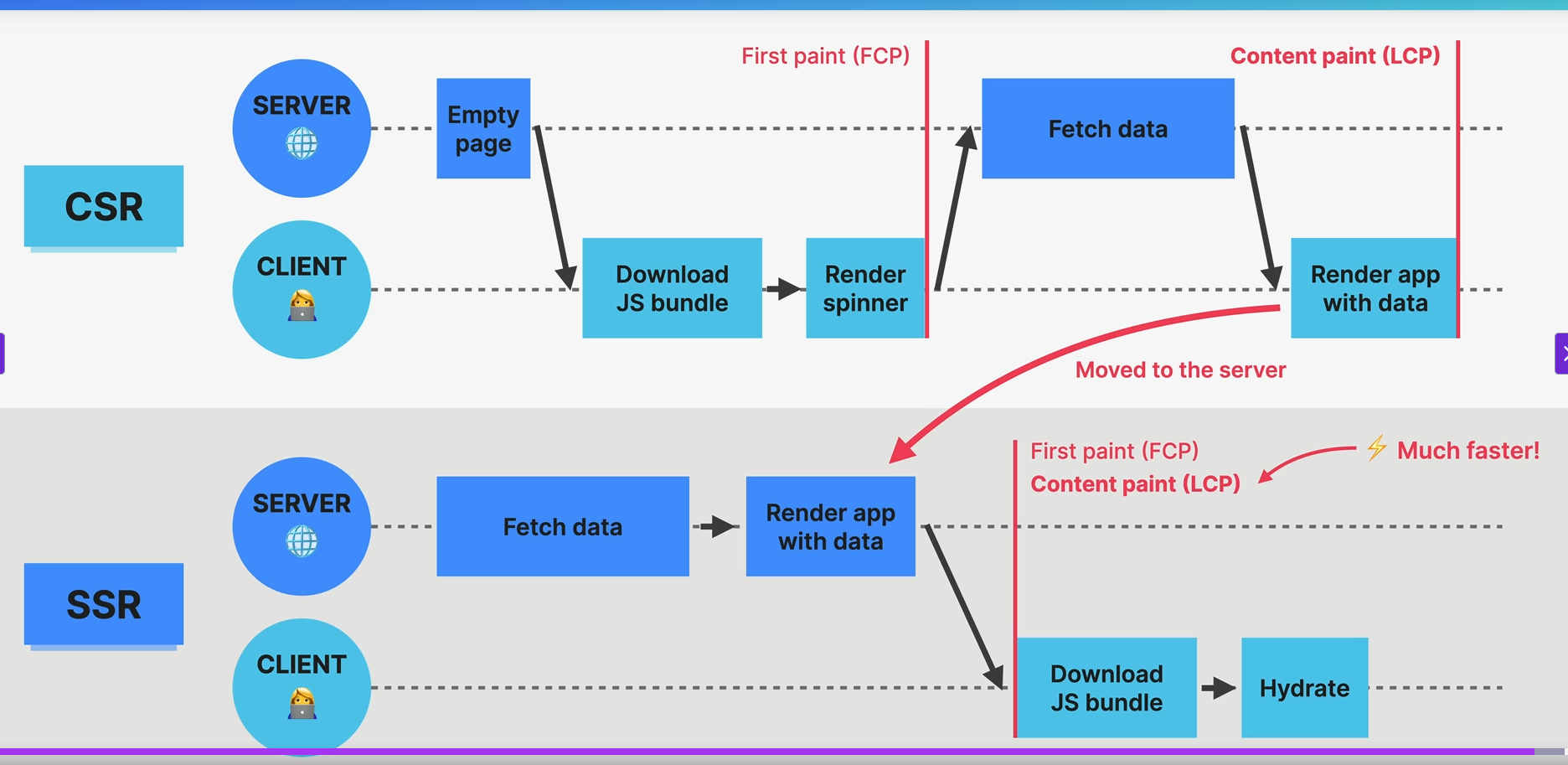
# Server-Side Rendering (SSR)

* The Old Way
  + The pages were rendered on the Server and send to the Browser
* The Modern Way
  + Client-side rendering
    - React, Angular, Vue, Svelte
* Back to Origin
  + Certain web apps shifted back to Server-Side Rendering
  + Next JS – Full stack Framework
* The HTML is generated on the server and sent to the client
* We shifted the work of rendering from the user to the server
* **SEO** is better on SSR since the data and HTML is already there when the bots are crawling the siter
  + With Client-Side Rendering, the bots might see an empty page, since the data is downloaded after the JS

# Typical timeline of CSR vs SSR



* SSR – much faster
  + Content Paint(LCP) is faster with less actions
  + Useful in heavy content sites
  + Static JS is sent to the client => **HYDARATIOn** happens where static JS is enhanced and transformed into Dynamic JS



# Manual SSR experiment

* We initialize a Node JS server
  + Npm init
* createServer from http module
* server.listen
* We copy/paste the JSX js code into the server
  + In order to run it, we have to install some BABEL libraries since JSX is not valid JS code
  + Npm I -D @babel/core @babel/preset-env @babel/preset-react @babel/register
  + Npm i react react-dom
* Start.js file
  + Some black magic
* To render React Components
  + renderToString from react-dom/server
  + Syntax
  + Const renderHtml = renderTOString(React component)
  + **We send this as the response. Like this we have SSR Rendered the content**
* We are sending the content inside the actual HTML file
  + New line
  + <div id=”root”>%%%%Content</div>
    - We will replace the %CONTENT with our actual **React component from renderToString**

Right now, there is no interactivity since we have displayed just simple static HTML. No click happens on the buttons.

This is the exact problem of the SSR – it’s generating static HTML and react logic can’t be used here. The answer is **HYDRATION**

# HYDRATION

* An essential concept to SSR
* It **adds back the interactivity and event handlers that were lost during the SSR rendering of HTML**
  + Because rendering HTML on the SSR will **remove all interactivity**
  + We need a way to add it back 🡺 **HYDRATION**

### How it works

* The Client will also download the JS bundle
* The Rendered HTML will use that JS bundle to add back interactivity
  + The process is called **HYDRATIOn**
* React will build back the component tree and compares it with the actual SSRd DOM. **They must be the same, so React will use it.**
* React simply adopts the existing DOM
  + In the end **we have the exact same React app**
  + (largest content paint) LCP is improved here

# Back to our experiment to implement HYDRATION manually

* We have the server and the HTML file
* We need the Client file ( the JS that will be attached to the DOM)
  + Create a new file – client.js
  + We add it as a **script** in the HTML
* There is a **GET** to /client.js so we also need to add it as a **Route**
  + We read the file with readFileSync
  + Serve it as a response
* We **must HYDRATE** the JS
  + We copy/paste the JS code into the Client.js
  + We have to add babel into the Front end using the **CDN**
    - <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    - We add it in the HTML – index.js
  + We also have to add React into the Front End using the **CDN**
* To **Hydrate**
  + ReactDOM (is a global variable from the CDN)
  + ReactDOM.hydrateRoot(document.getElementById(“root”),<Home/>);

# NEXT.jS

* A meta framework built on top of react
* We still get to use all the components , props and hooks
* Adds a **set of conventions and opinions** 
  + **Opinionated way of building React** apps
  + best practices regarding routing, data fetching
* Allows us to build complex FULL STACK apps and web sites

## SSR

* Supports both dynamic and static rendering
  + Can be selected for each route

## File-Base Routing

* Folders as Routes
* Special files for pages, layouts, loaders

## Data Mutation and Fetching

* Ability to fetch and mutate data directly in the **Server Components**
* Mutations in **Server Actions**

## Optimization

* Images
* Fonts
* SEO
* Preloading

# Next JS Router

1. Modern **APP router**
   1. From 2023 – Next.js 13.4
   2. Recommended way of starting new projects
   3. Implement the Full-stack architecture vision
      1. Server Components
      2. Server Actions
      3. Streaming
   4. Easy fetching with **fetch()**
   5. Easy to create layouts, loaders
   6. More advanced routing
   7. Better DX (developer Experience) and UX
2. Legacy **Pages router**
   1. Legacy router
   2. Still supported

# First NEXT.js Project

* Npx create-next-app@14
  + Npx create-next-app@latest – to get the latest version

# Routes and Pages

* We want to add 3 new routes
  + Simply create 3 new folders
    - Cabins
    - About
    - Account
  + Inside each folder we have to add a js file
    - A react component

# Page Navigation

* An anchor element with a href will **reload the whole page, download the whole app again**
* **Next** provides the **Link Component**
* <Link href='/cabins'> Explore Cabins with link</Link>

# Layouts

* We add a global layout to our application
  + Similar to the **outlet** in React Router
* Next JS creates the **Root Layout**
  + the RootLayout will wrap the whole application
  + it needs to contain the **HTML and Body** tags
  + we add the Navigation
* Using **children Props**, we also display the content of the page
  + RootLayout({children})
  + We use the {children}inside the RootLayout

## Page Metadata

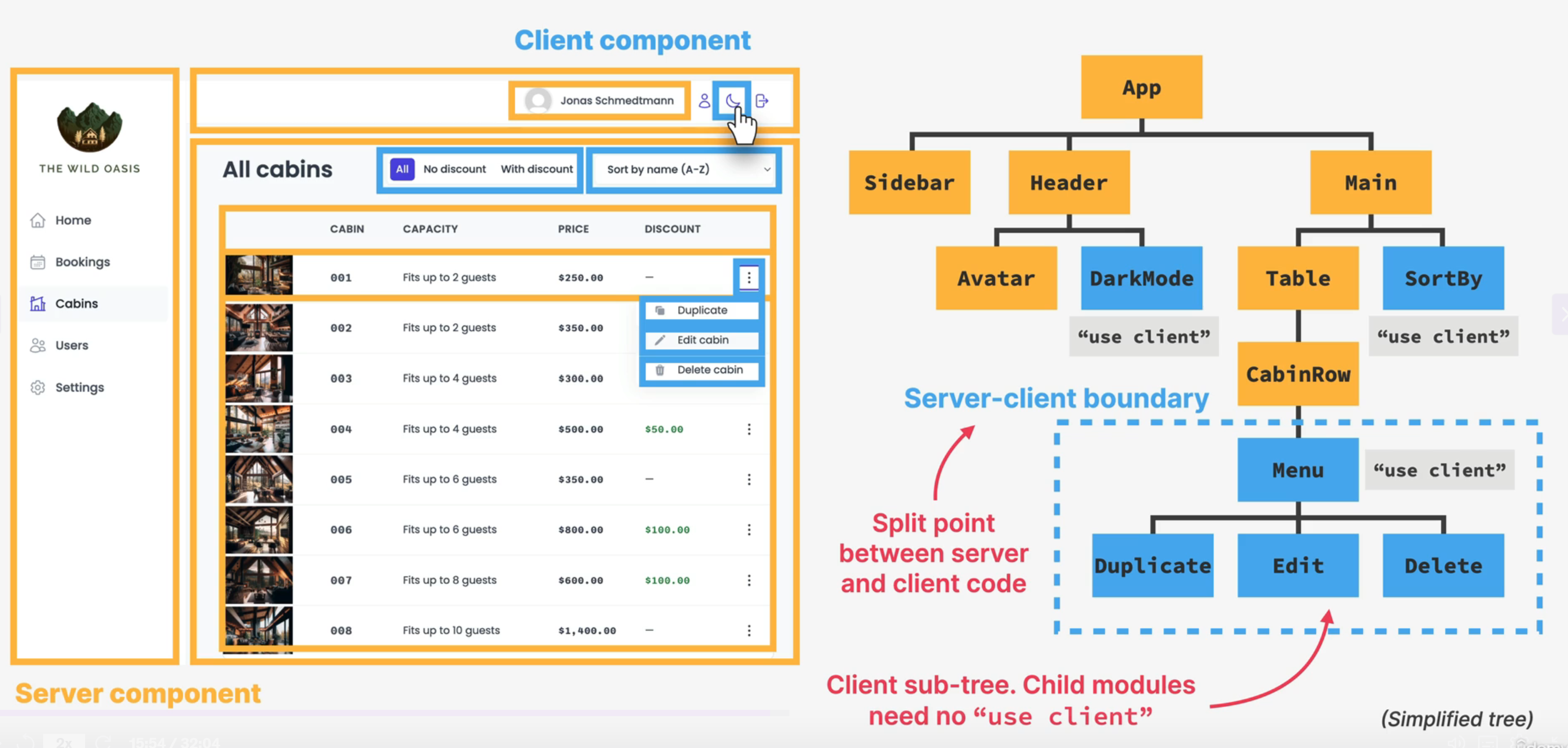
* We can export some page metadata from the **Layout Page**
  + We simply to
    - Export const metadata = {title:”what ever title we want”})

# Server Components – What are?

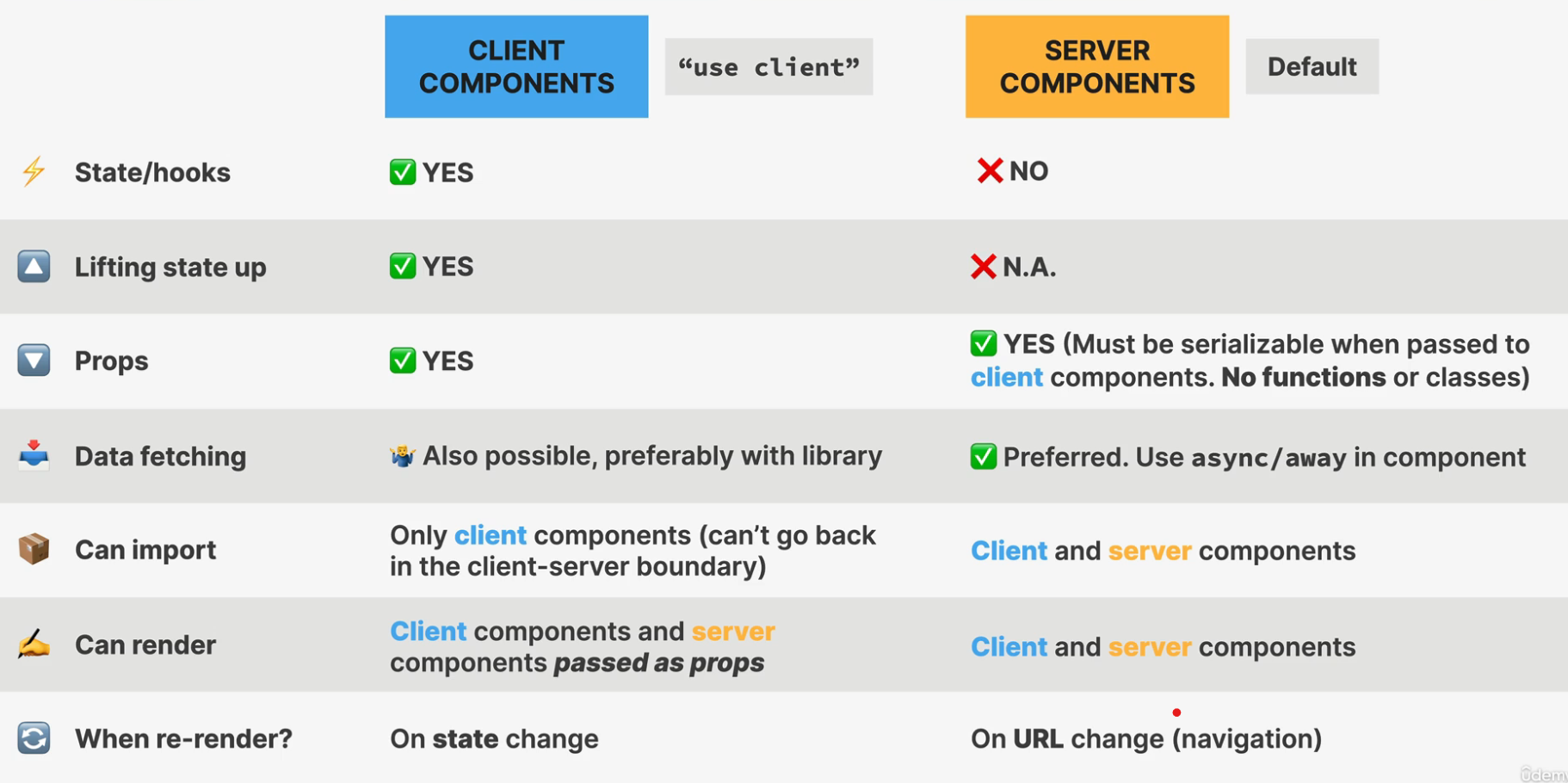
* A completely new Full Stack Architecture for React Apps
* Introduces the **server** as a **integral part of the component tree**
  + React Tree extends to the server
* **RSC =>** the name of the new architecture
* **Server Component =>** the new Component type
  + Used to fetch data from the server
  + Components that are only rendered on the server
  + No interactivity
  + No state
  + No JS for dynamic things
* **Client Components**
  + The “old” components that have state and interactivity
  + On the Client Side
* Server components are the new **default** for the component in a **Next.js app**
* We need to **specify if a component is client component** 
  + **Use client** at the top of the module

## Server-Client boundary

* An important principle
* Defines the boundary between **server** and **client** components
* A Client component is defined using the **use client** at the top
  + The **children of SC** don’t need use client **anymore**
  + They are already in the **client sub-tree**



## Client Side Comp vs Server Side Comp



# Fetching Data in a Page

* We are using the json placeholder for dummy data

### We use async/ await

1. We need to convert the **component** into a **ASYNC** function
2. We can await fetch(‘URL api’) inside the Component
3. const res = await fetch("https://jsonplaceholder.typicode.com/comments");
4. const dummyData = await res.json();
5. console.log("dummyData", dummyData);

# Client Component

* Just add ‘use client’ at the top and then the component can be used as Client Component => **INTERACTIVITY**

# Loading Spinner

* There is a **Convention** => **Loading.js**
  + It’s a global Loading File
  + It’s a component
* export default function Loader() {
* return <p>Loading Data....</p>;
* }

# How RSC works behind the scenes?

