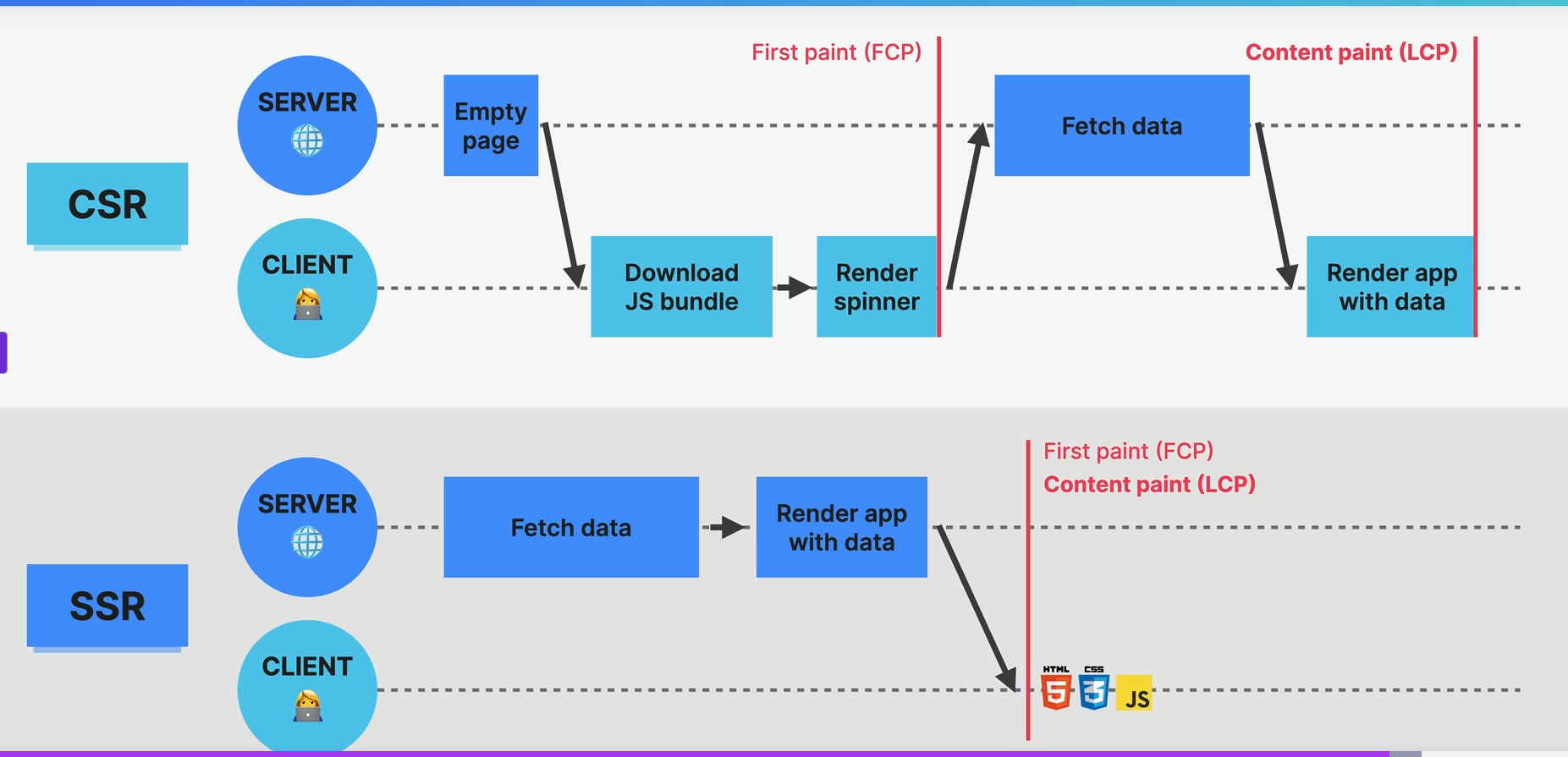
NEXT JS

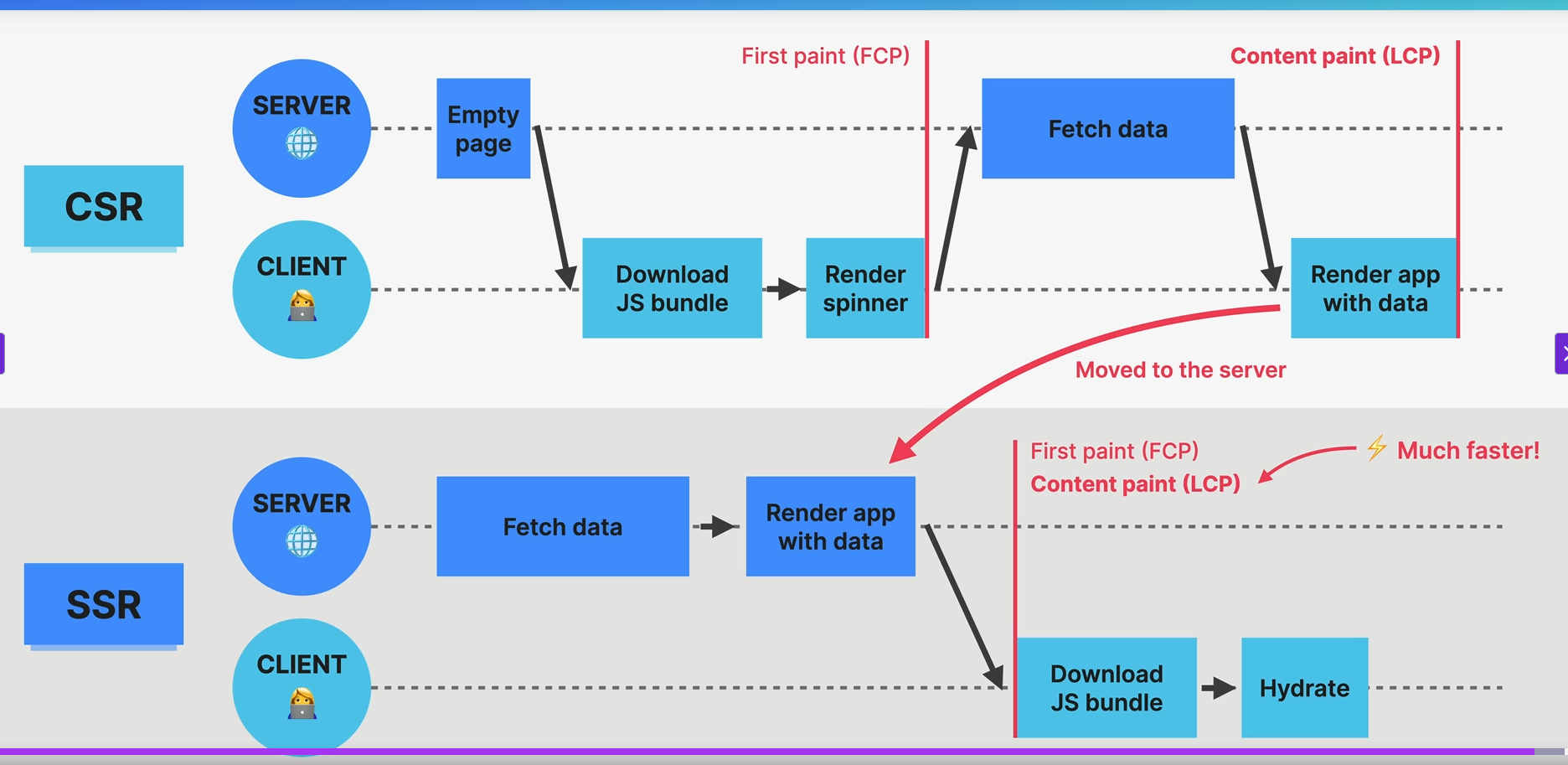
# Server-Side Rendering (SSR)

* 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

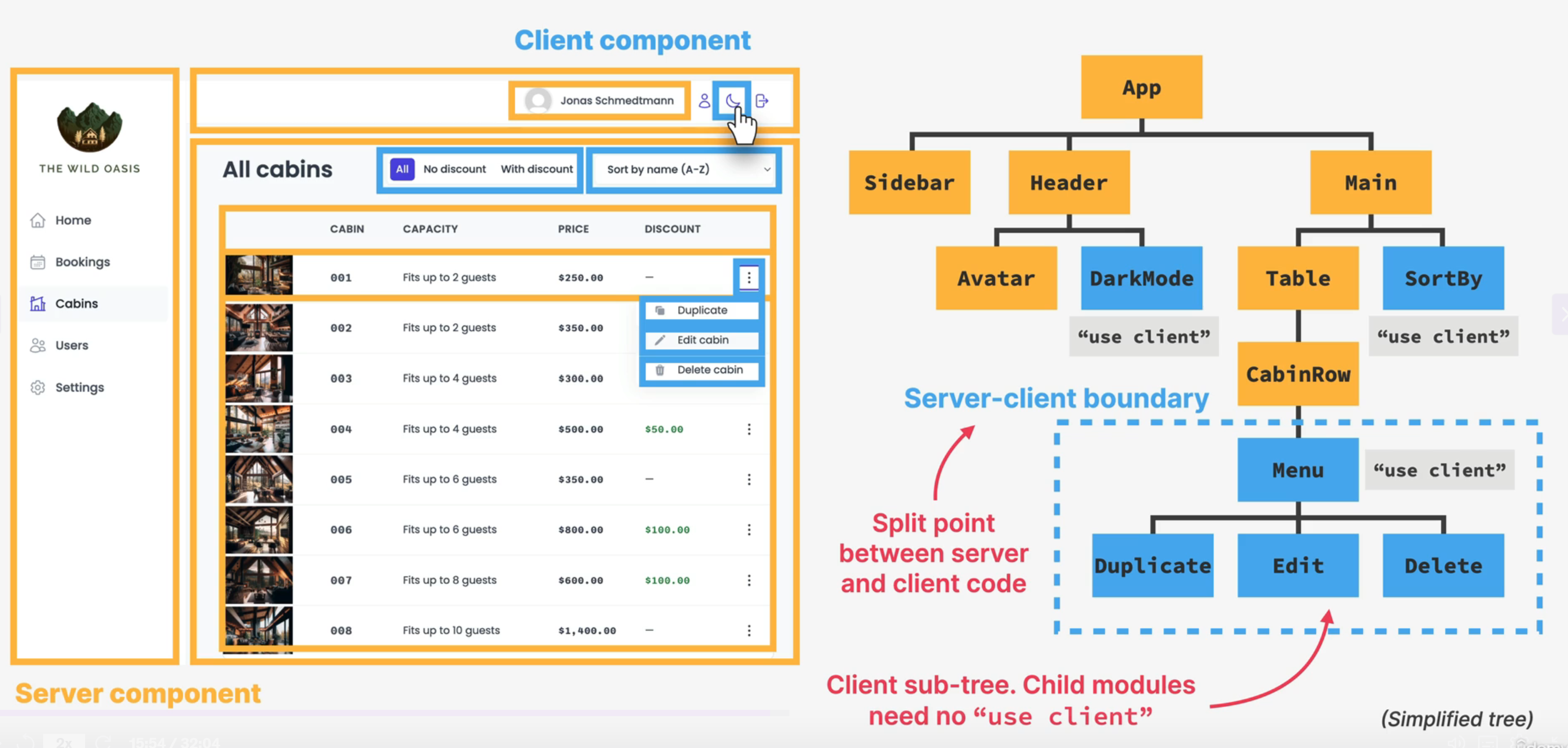
* We can export some page metadata from the **Layout Page**
  + We simply to
    - Export const metadata = {title:”what ever title we want”})

# Server Components – What are?

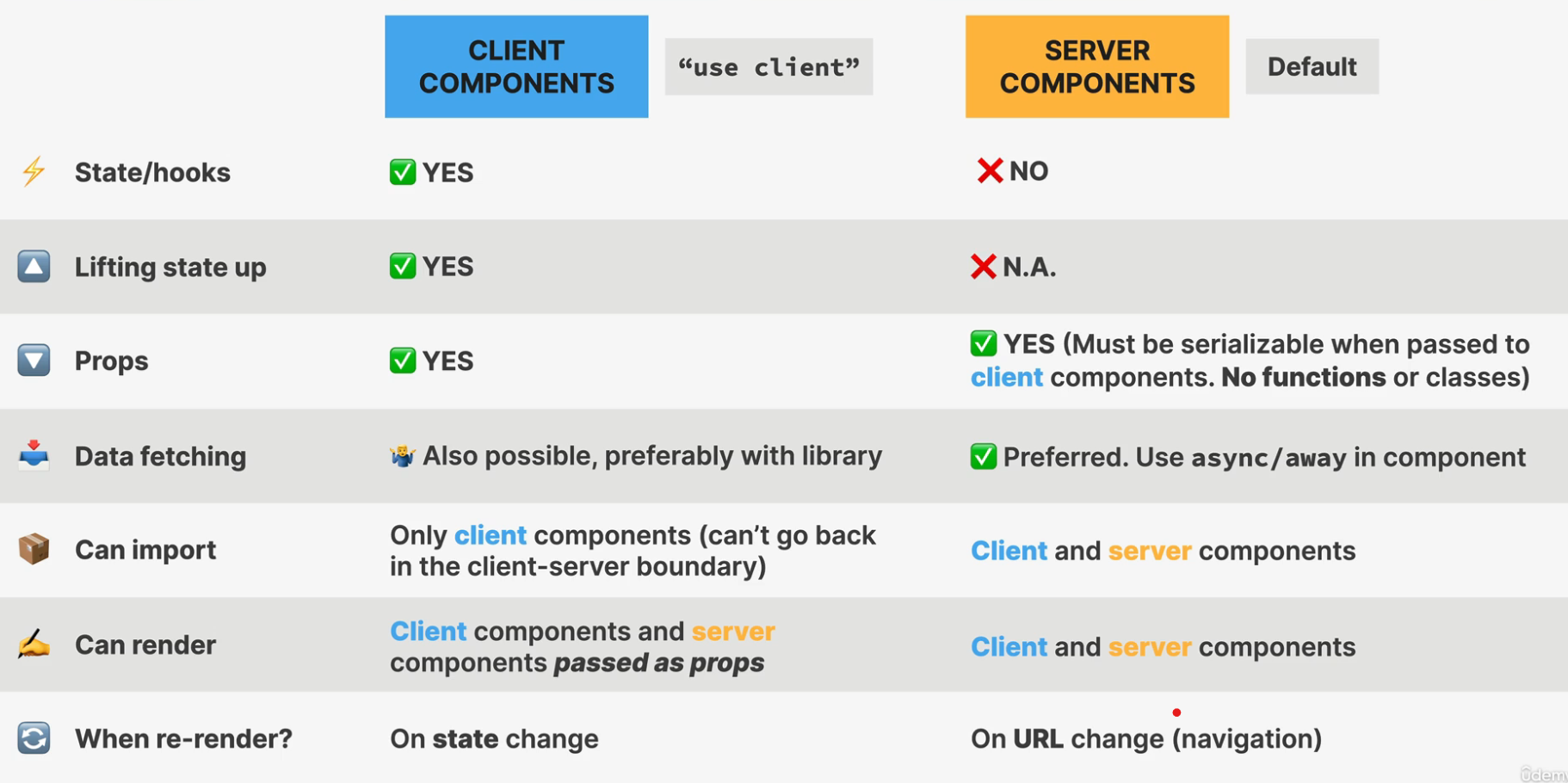
* A completely new Full Stack Architecture for React Apps
* Introduces the **server** as a **integral part of the component tree**
  + React Tree extends to the server
* **RSC =>** the name of the new architecture
* **Server Component =>** the new Component type
  + Used to fetch data from the server
  + 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);

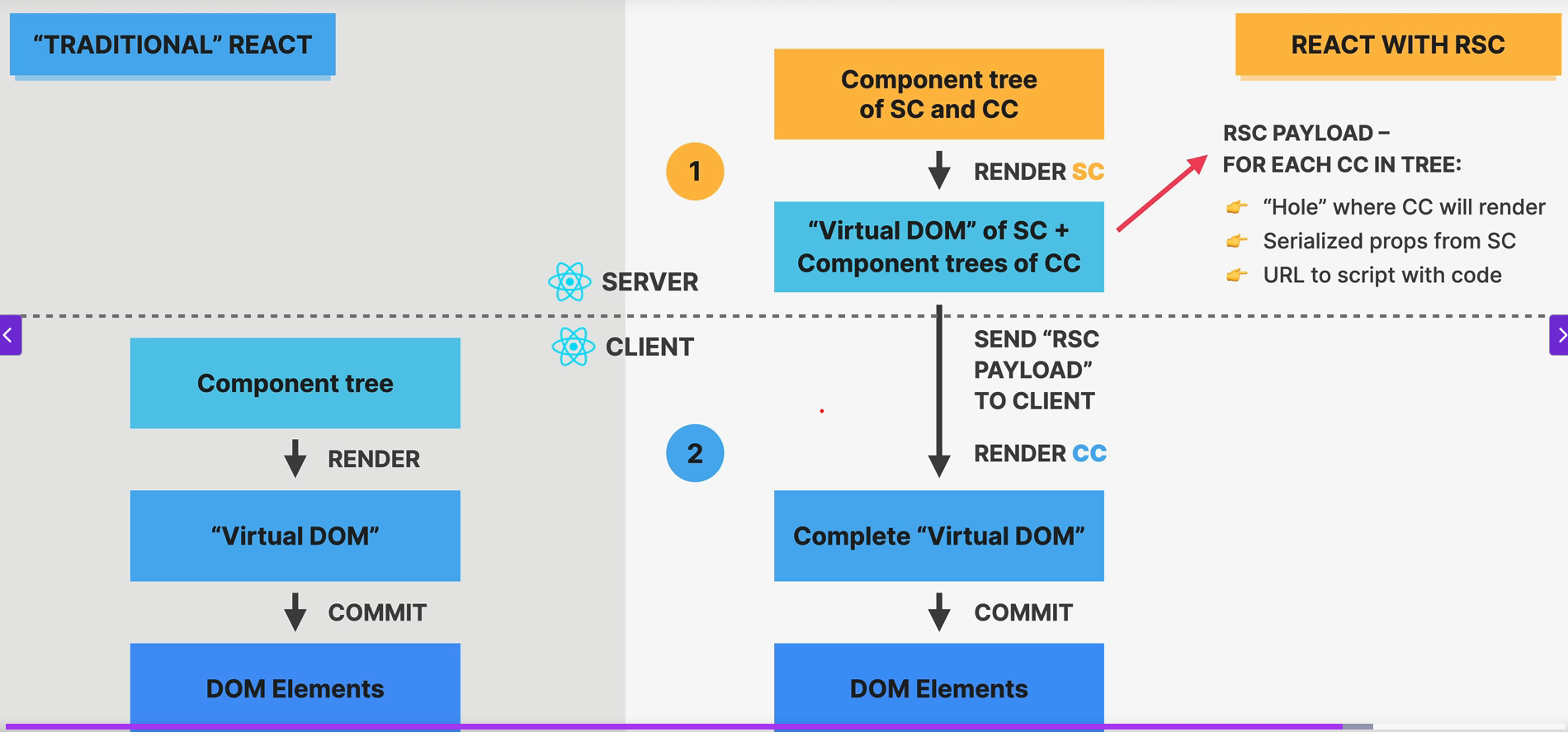
# Client Component

* 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?



# The Wild Oasis Customer Website

## Project Requirements

1. Users of the app are potential guests or actual guests
2. Guests should be able to learn all about the Wild Oasis Hotel
3. Guests should be able to get information about each cabin and see booked dates
4. Guests should be able to filter the cabins by their maximum capacity
5. Guests should be able to reserve for certain date range
6. No Payments Online
   1. Made at the property
   2. New reservations should be set to ‘unconfirmed’
7. Guest should be able to see the past or future reservation
8. Update or delete reservation
9. Gests need to be able to sign up and log in before they can reserve a cabin
10. On sign up, each guest get a profile in the DB
11. Guests should be able to set up and update basin data about their account

## Split into Features

### About

1. Guests should be able to learn all about the Wild Oasis Hotel

### Cabins

1. Guests should be able to get information about each cabin and see booked dates
2. Guests should be able to filter the cabins by their maximum capacity

### Reservations

1. Guests should be able to reserve for certain date range
2. No Payments Online
   1. Made at the property
   2. New reservations should be set to ‘unconfirmed’
3. Guest should be able to see the past or future reservation
4. Update or delete reservation

### Authentication

1. Gests need to be able to sign up and log in before they can reserve a cabin
2. On sign up, each guest get a profile in the DB

### Profile

1. Guests should be able to set up and update basin data about their account

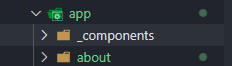
## Pages

1. Homepage
   1. /
2. About page
   1. /about
3. All Cabins
   1. /cabins
4. Single Cabin
   1. /cabins/:cabinId
5. Login page
   1. /login
6. Account
   1. /account

## Tech Stack

1. Next JS
2. Global UI state
   1. Context API
   2. Or Redux or Zustand
3. Supabase
   1. DB/API
4. Tailwind CSS
   1. Styling

## File Structure

* If we **start a folder name with an underscore**
  + Next JS will **not** account the folder **as a route**
* 

## Tailwind CSS Styling

* Import the file into the Layout or into the Application Wrapper

## Page Metadata

* All the text should be defined in the metadata tag
* export const metadata = {
* title: "The Wild Oasis",
* description: "Generated by Next.js",
* };
* We can export the metadata from the layout (wrapper)
* We can also export the metadata from each page
  + Export const metadata={
  + Title:”Cabins”}
* To use what we are exporting form each page we can use
* export const metadata = {
* // title: "The Wild Oasis",
* title: {
* template: "%s - The Wild Oasis",
* default: "Welcome - The Wild Oasis",
* },
* description: "Generated by Next.js",
* };
  + The %s will get replaced with **what we export from each page**

## Fonts

* We can host our own fonts straight in the Next js
* We import any font in the layout

1. Import the desired font
   1. This returns a function
2. Call the function and set the options
   1. Import the className from the function into the Wrapper

## Image Optimization in Next.js

* A huge advantage of this framework
* A new **Image component from Next.js**
  + Image from ‘next/image’
* Prevents layout shifs

### Another way of using Image

* Import the actual image at the beginning of the module
  + Import logo from “@/public/logo.pn”
* Like this we can remove the height and width
* Next.js is analyzing the image before it’s rendered
  + By this, we can use the **Quality** property

## Nested layout

* The **RootLayout** applies to the whole application
* If we want another **layout**  to apply just to a single route, we create a **new layout inside that route folder**
  + This layout will be applied to the **page** files that are inside that folder

# Supabase Installation

* Npm I @supabase/supaabase-js
* All keys stored into the .env are on the server, so they are not leaking to the public
  + We can set a key to be available to the public
    - **NEXT\_PUBLIC\_VArName**
* We are using the **Service\_role**  key since it’s overriding the Row Level Security

# Loading item

* We can add the global Loader alongside the global page and layout
* We can also add a **Loader**  for distinct routes, just like adding a layout
  + We add the loader file into the route

## Adding a loader activates the **STREAMING**

* Now the website will work only if JavaScript is available

# React Suspense

* A modern a react feature
* A built-in React component that we can use to catch/isolate componets that are **not ready to be rendered (“suspending”)**
* Like a catch block
  + It catches components that are suspending

### What causes a component to be suspending?

1. Fetching data
   1. We need a data fetching library that supports suspense
2. Loading code with React Lazy Loading

### How to use it

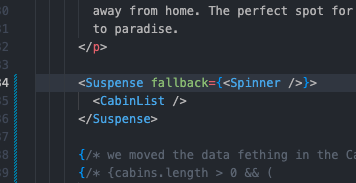
* We wrap the whole SubTree, above the **suspending component in a** Suspense component
* **This is a declarative way** – no more isLoading states and render logic

## How it works?

* While rendering, **suspending** component is found
* The flow goes back to the **closest Suspense Parent (‘**boundary’)
  + All already rendered children are discarded
    - But the state is **preserved** in the **Fiber Tree**
  + Display **fallback** component while ASYNC operation happens
    - just a spinner usually
* After Async work is done – the component is no longer suspending
  + **React will RERENDER the subtree** with the fetched data
* **Suspense works OUT OF THE BOX** with Next JS.

## How to use it?

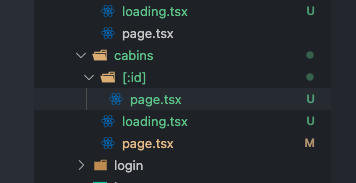
* We mode the data fetching into it’s own component
  + Cabin list
* We add the new component to the old component and **wrap it into Suspense**
* **Suspense** takes in a prop for the **fallback**



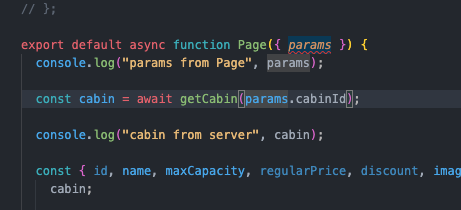
* Now, the text it loading and it’s now waiting for the cabins data

# Dynamic Route Segments

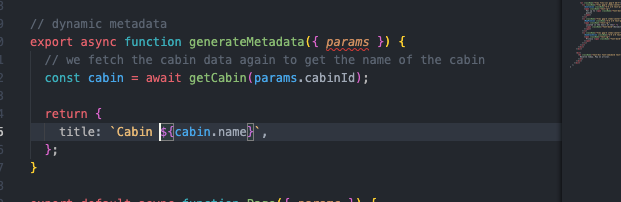
* It’s about the URL segment
* We create a new folder inside the parent (Cabins)
  + This are the childrens of the Cabins, a single **cabin**
  + We use **square brackets**



* Any page that is associated to the **Dynamic Route Segment**  has access to **PARAMS**



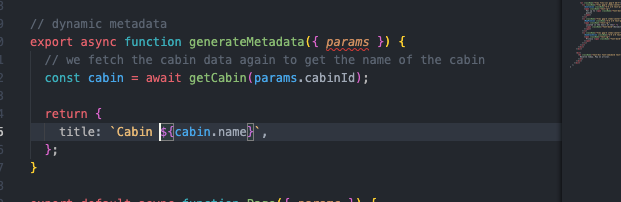
# Dynamic MetaData



* We create a generateMetadata function
  + We fetch the cabin data again

# Error Handling - Global Error Boundary

* Will automatically Wrap the **WHOLE app**
* Another convention
  + **In the root folder => error.js**
* The error file is **always a client component**
  + **‘**use client’



# Server-Side Rendering

* Next.js is a React Frameworks, so rendering is done by React, following the rules we learned earlier
  + Both Server and Client components are rendered on the server
* Next. Js splits the rendering work **by routes**
* Each route can be rendered
  + **Static** (pre-rendered)
  + **Dynamic**

## Static Rendering

* The HTML is generated at built time
* Or periodically in the background
* Rendered just once at built time
* When data doesn’t change often and it not personalized
* **DEFAULT RENDERING in next.js**
  + Way faster

## Dynamic Rendering

* HTML is generated at **Request Time**
* For each new request reaches the server
* The data changes frequently
* The data is personalized to the user
* Rendering depends on the request(params)

## When next.js switches from STATIC to DYNAMIC

* All is done automatically

1. The route has some dynamic segment (page uses **params)**
2. **searchParams** are used in the page component
3. **headers()** or **cookies()** are used in any of the route’s server components
4. An **uncached data request** is made in any of the route’s server

## Terminology

* CDN
  + **Content delivery network**
  + A network of servers located around the globe
  + They store the **static content of a website**
  + As close as possible to the user
* Serverless Functions
  + **Serverless computing**
  + Run single functions on a cloud provider
  + The server is initialized **only when the function is running**
  + Each dynamic route becomes a **serverless function**
  + Hosted on the CDN
* Edge
  + As close as possibe to the user
  + The CDN is part of the **edge network**
  + **Serverless computing is also part of the edge network**
  + Edge Computing
* ICR
  + **Incremental Static Regeneration**
  + Process in Next.js to allow devs to update the content of a **static page**, in the background

## Npm run Build

* We can see which routes are Dynamic or Static

A screenshot of a computer

AI-generated content may be incorrect.

## Making Dynamic Pages with static with GENERATEPARAMSVALUES

* We can actually let Next.js know what params will a Dynamic page have, so it will be rendered as a Static page
* When we have a small base of params, we can let Next.Js know about this
* So that we can generate the whole site as **static**  so it can be **deployed anywhere**

### generateStaticParams();

* We export this function from the page where we have the dynamic content
* We fetch all the data in this function
* We get all the id’s
* A screen shot of a computer program

  AI-generated content may be incorrect. A screenshot of a computer program

  AI-generated content may be incorrect.
* Now, by running npm run build we get a new Static Tree

A screenshot of a computer program

AI-generated content may be incorrect.

## Since all Routes are Static – we can do STATIC SITE GENERATION (SSG)

* For easy deployment to any Hosting provider

### Steps:

1. Next.config.mjs
   1. Add: output:”export”
2. Run npm run build
   1. It will throw an error while attempting to do this if not all **routes are static**
3. Take the **OUT** folder and use it to deploy to a host

# Partial Pre-Rendering

* Most pages don’t need to be 100% static or 100% dynamic
* **Solution => Partial Pre-Rendering**
  + A new rendering method
  + A middle ground between static and dynamic

## How it works

1. A **static** – pre-rendered – shell is served immediately from the CDN, making it super fast
   1. A shell
   2. It leaves holes for the dynamic content
2. In the meantime, the server starts rendering the **dynamic** parts, and **it’s streaming them as they are rendered on the server**

Using this new method, means that the pages can be fast even if there are small dynamic parts

## How to use

1. Highly experimental
2. The dynamic component needs to be set placed into a **Suspense Boundary**
3. This tells Next.JS that what is in the Suspense in dynamic

# Next JS Caching

* Everything that can be cached, will be cached
* Aggressive caching
* Always on by default
  + Can lead to strange behavior
* Very confusing
  + Many different Next.js APIs affect the caching

## Caching Mechanism

1. Request Memoization – Server
   1. Data is cached and reused during a page rendering
   2. Allows us to fetch cached data in multiple routes
   3. Fewer requests
   4. Works with native fetch function
2. Data Cache – Server
   1. Data stays there forever
   2. Every user will get the exact same data
   3. This data is used to statically render routes
3. Full Route Cache – Server
   1. Stores entire HTML pages
   2. Static routes and storing them as HTML and RSC payload
   3. Is persistent until the Data Cache is invalidated
   4. Does not survives redeployes
4. Router cache – Client
   1. Stores in the browser all the prefetched pages and visited pages
   2. Static and Dynamic routes
   3. Allows for instant navigation
   4. Static pages
      1. Stored for **5 minutes**
   5. Dynamic pages
      1. Stored for **30 seconds**
   6. **MIGHT SERVE STALE DATA**

## How to configure cache

* Request Memoization
  + Revalidation
    - No way of revalidation
    - Happens only on the life span of a single page
  + Opt Out
    - AbortController
* Data Cache
  + Revalidation
    - Time based revalidation for **all data on page**
      * Export const revalidate =<time>; (page.js)
      * We export a revalidate const
    - Time based revalidation for **one data request:**
      * fetch(‘…’, {next: {revalidate: <time?}})
    - On-demand (manual)
      * **revalidatePatch** or **revalidateTag**
      * we can call these 2 on a certain path or tag
  + Opt Out
    - Entire page
      * Export const revalidate = 0 (page.js)
      * We export the revalidate set to 0 seconds, it will always revalidate
    - Individual request
      * Fetch(‘…’, {cache: ‘no-store’})
    - Individual server component
      * noStore()

## Implement the caching mechanism

* caching is disabled in **dev**
* To simulate a prod environment we use the
  + **Build script**
    - Builds the application
  + **Start script**
    - Starts a production server

1. Opt Out of the data cache / full route cache
   1. We expor const revalidate =0 to the cabins route
   2. The value cannot be computed
      1. **It needs to be a REAL VALUE**

A screen shot of a computer program

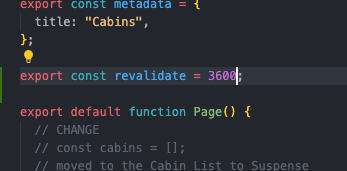
AI-generated content may be incorrect.

* 1. Now we can see that the /**cabins** is rendered Dynamically, and if we change something in the Database, the price is updating

A screenshot of a computer

AI-generated content may be incorrect.

# Incremental Static Regenration

* Export const revalidate = “some seconds”
* A middle ground between full static and full dynamic
* 

## Opt Out for Single Component

* We opt out the CabinList
  + We can’t use the Fetch(‘…’, {cache: ‘no-store’}) since we are not using **fetch**, but Supabase
* We can use the ‘noStore()’
* import { unstable\_noStore as noStore } from "next/cache";