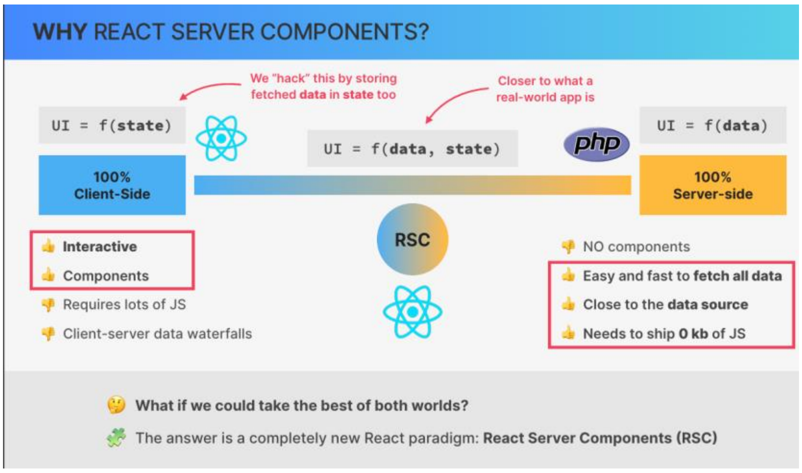


WHY REACT SERVER COMPONENTS?

We "hack" this by storing **fetched data** in **state** too

Closer to what a real-world app is

`UI = f(state)`

`UI = f(data, state)`

`UI = f(data)`

100% Client-Side

100% Server-side

RSC

👍 **Interactive**

👍 Components

👎 Requires lots of JS

👎 Client-server data waterfalls

👎 NO components

👍 Easy and fast to **fetch all data**

👍 Close to the **data source**

👍 Needs to ship **0 kb** of JS

🤔 **What if we could take the best of both worlds?**

🧩 The answer is a completely new React paradigm: **React Server Components (RSC)**

Notes:

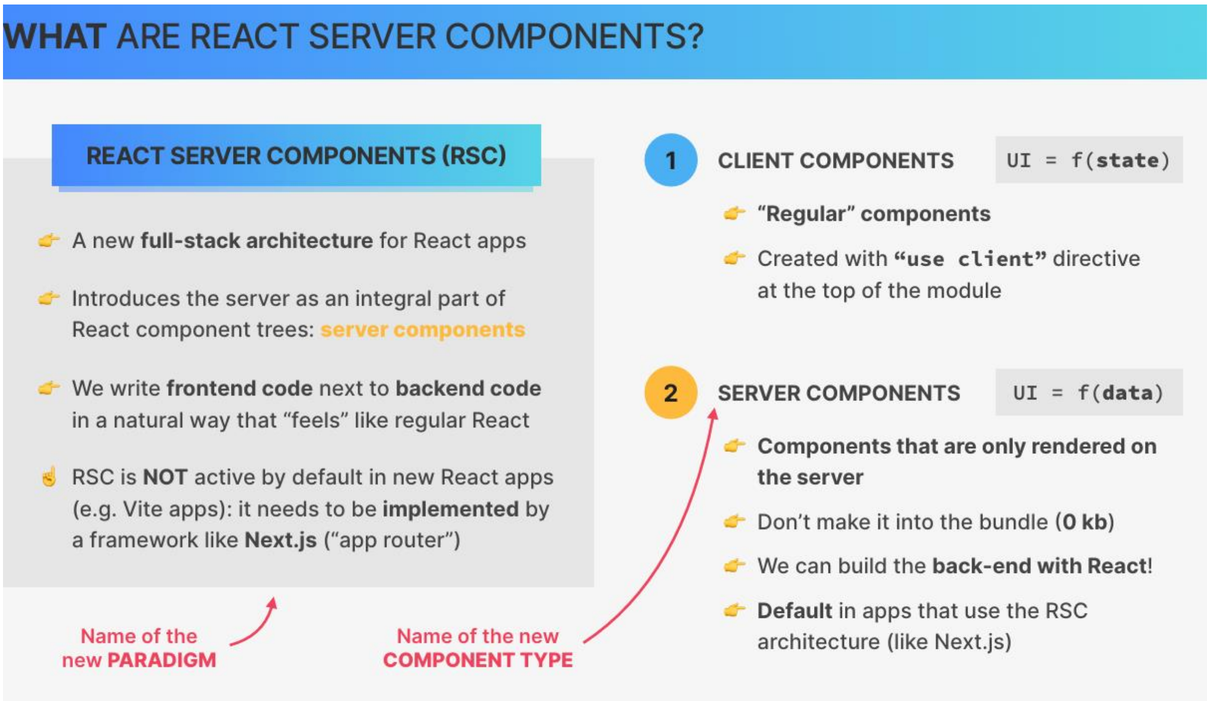
React Server Components (RSC) are a new paradigm in React, which combines server-side and client-side rendering

- **Traditional React (100% Client-Side Rendering):**
  - UI is a **function of state**.
  - Pros:
    - Highly interactive.
    - Reusable, composable components.
  - Cons:
    - Large JavaScript bundles impact performance.
    - **Client-server waterfalls**: **Sequential data fetching in components leads** to delays.

- **Traditional Server-Side Rendering (e.g., PHP):**
  - UI is a **function of data**.
  - Pros:
    - **Fast data fetching directly** from the server (e.g., database).
    - **No JavaScript needed** for initial render.

- Cons:
    - Lack of interactivity.
    - No component-based architecture.

- **Goal of RSC:**
    - Combine the best of both worlds:
        - Interactivity (client-side).
        - Proximity to data sources (server-side).
        - Reduce JavaScript bundle size.



WHAT ARE REACT SERVER COMPONENTS?

REACT SERVER COMPONENTS (RSC)

- A new **full-stack architecture** for React apps
- Introduces the server as an integral part of React component trees: **server components**
- We write **frontend code** next to **backend code** in a natural way that "feels" like regular React
- ☝ RSC is **NOT** active by default in new React apps (e.g. Vite apps): it needs to be **implemented** by a framework like **Next.js** ("app router")

Name of the new **PARADIGM**

Name of the new **COMPONENT TYPE**

1 CLIENT COMPONENTS `UI = f(state)`

- **"Regular" components**
- Created with **"`use client`"** directive at the top of the module

2 SERVER COMPONENTS `UI = f(data)`

- **Components that are only rendered on the server**
- Don't make it into the bundle (**0 kb**)
- We can build the **back-end with React**!
- **Default** in apps that use the RSC architecture (like Next.js)

- **Server Components**:
    - Rendered **only on the server**.
    - Fetch data directly from the server (e.g., databases).
    - **No interactivity** (no state or hooks).
    - **Zero JavaScript** shipped to the browser.

- Default in RSC architecture.

  Fetch data using async/await directly in the component.

  Cannot use hooks or state.

  Re-render when the URL changes (tied to routes)

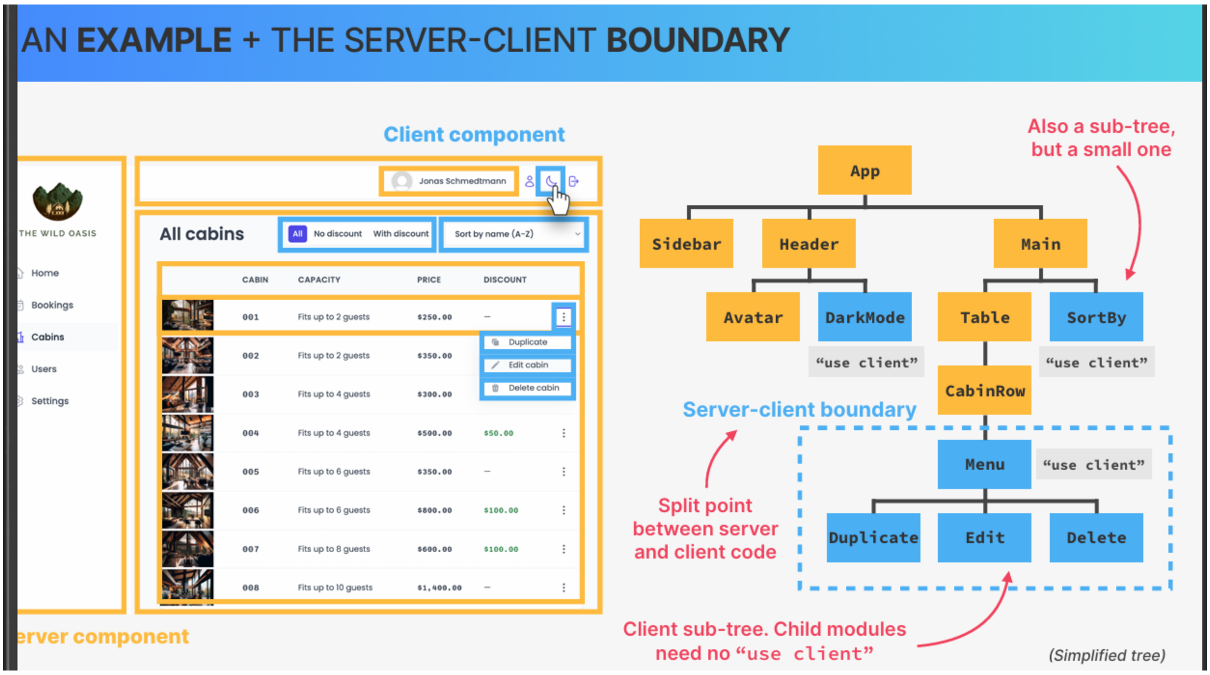
Client Components:

- Require JavaScript in the browser.

  Opt-in using the use client directive.

  Handle interactivity, state, and hooks.

  Can render server components passed as props



AN **EXAMPLE** + THE SERVER-CLIENT **BOUNDARY**

	CABIN	CAPACITY	PRICE	DISCOUNT
	001	Fits up to 2 guests	$250.00	–
	002	Fits up to 2 guests	$350.00	
	003	Fits up to 4 guests	$300.00	
	004	Fits up to 4 guests	$500.00	$50.00
	005	Fits up to 6 guests	$350.00	–
	006	Fits up to 6 guests	$800.00	$100.00
	007	Fits up to 8 guests	$600.00	$100.00
	008	Fits up to 10 guests	$1,400.00	–

Also a sub-tree, but a small one

Server-client boundary

Split point between server and client code

Client sub-tree. Child modules need no "use client"

(Simplified tree)

Client-Server Boundary:

Marks the split between server and client code.

Created using the use client directive.

Child components of client components inherit the boundary.

1. **Props in RSC:**
   - **Can pass props from server to client components.**
   - **Props must be serializable (no functions or classes).**

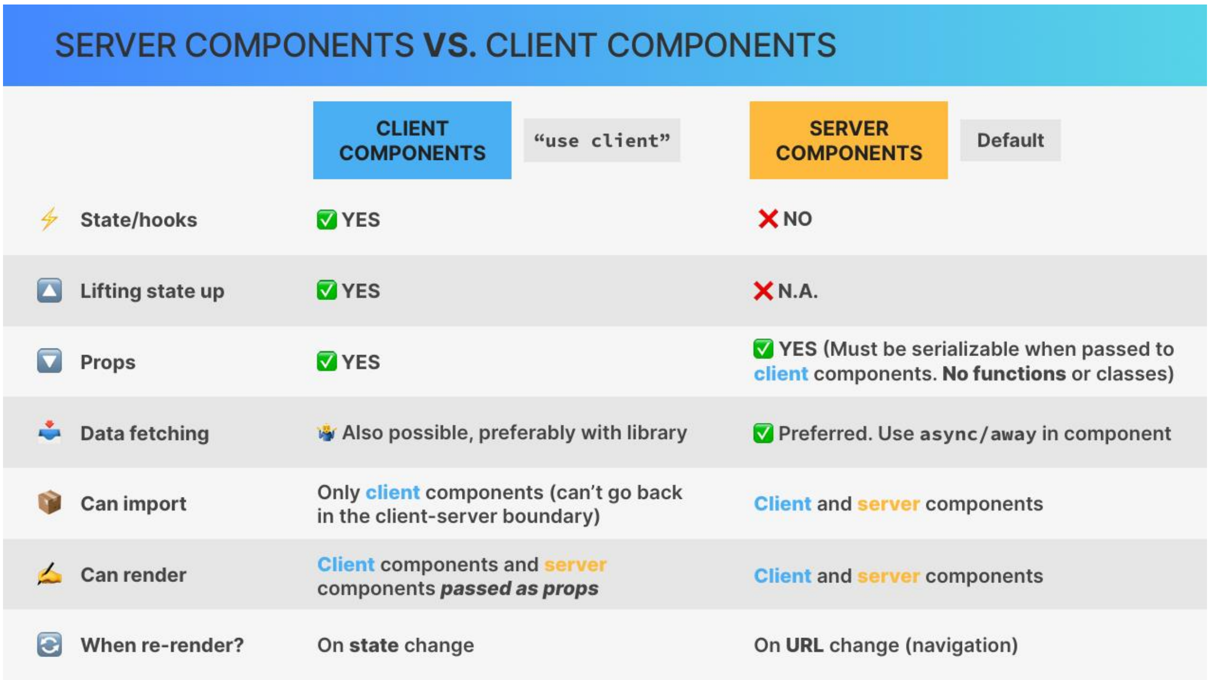
Data Fetching:

- **Preferred in server components.**
- **Avoids client-server waterfalls.**

- **Client components can still fetch data (e.g., using React Query)**

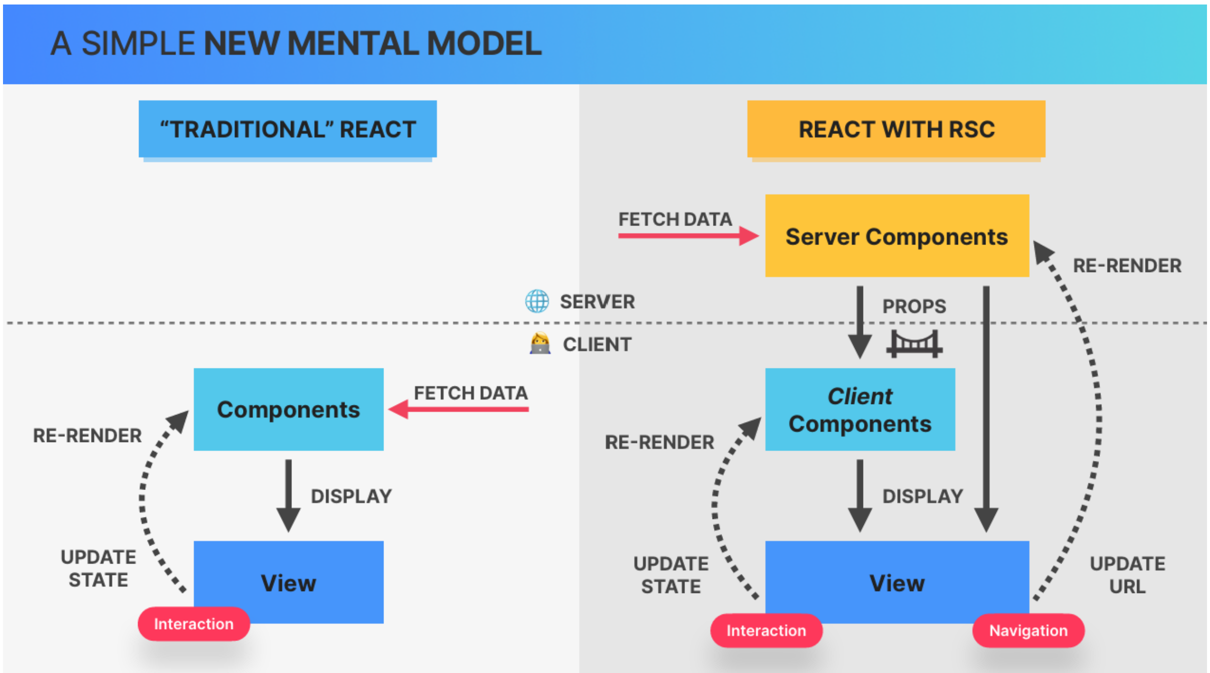
Comparison: Server vs. Client Components

Aspect	**Server Components**	**Client Components**
Default/Opt-in	Default in RSC.	Opt-in using use client.
State & Hooks	No state or hooks.	Can use state and hooks.
Interactivity	Non-interactive.	Interactive.
Data Fetching	Fetch data directly on the server.	Fetch data on the client (e.g., React Query).
JavaScript Bundle	Zero JavaScript shipped to the browser.	Requires JavaScript in the browser.
Re-rendering	Re-renders on URL changes.	Re-renders on state or parent state changes.



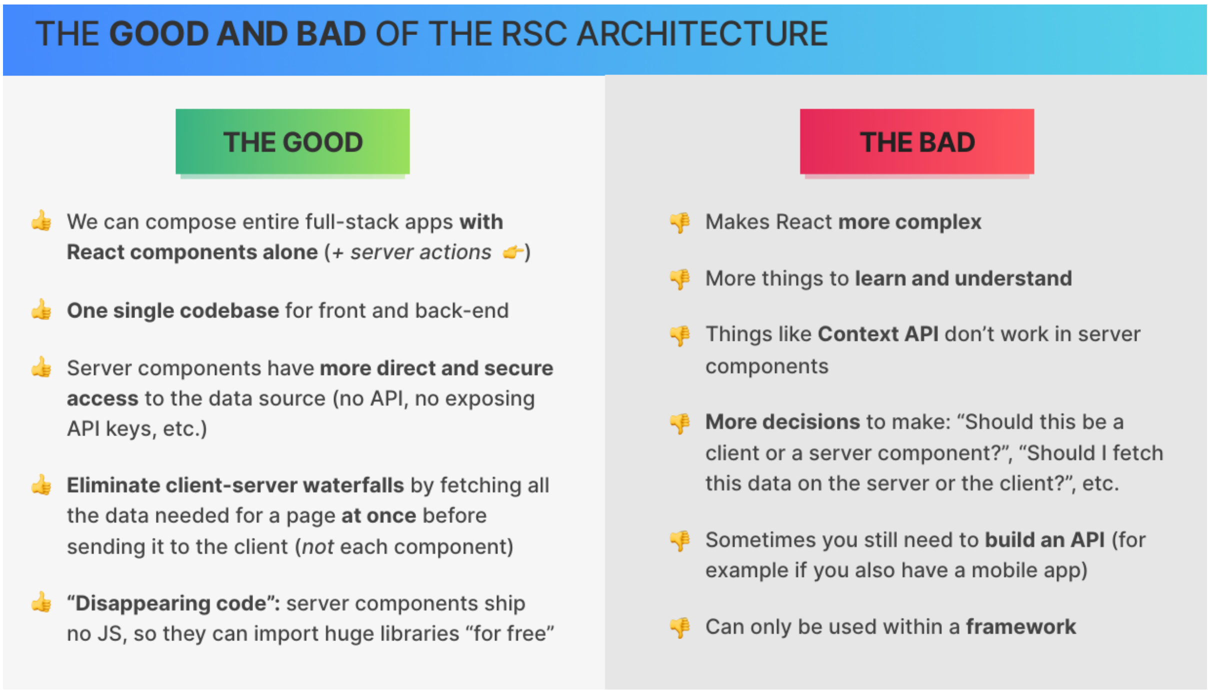
SERVER COMPONENTS **VS.** CLIENT COMPONENTS

	CLIENT COMPONENTS “use client”	SERVER COMPONENTS Default
State/hooks	✓ YES	✗ NO
Lifting state up	✓ YES	✗ N.A.
Props	✓ YES	✓ YES (Must be serializable when passed to client components. **No functions** or classes)
Data fetching	Also possible, preferably with library	✓ Preferred. Use `async/away` in component
Can import	Only client components (can't go back in the client-server boundary)	Client and server components
Can render	Client components and server components ***passed as props***	Client and server components
When re-render?	On **state** change	On **URL** change (navigation)



Mental Model of RSC

- **Traditional React**:
  - Components display a view.
  - State updates trigger re-renders.
  - Data fetching stored in state.
- **RSC Enhanced Model**:
  - **Server Components**:
    - Fetch data and render views on the server.
    - Pass data to client components via props.
  - **Client Components**:
    - Handle interactivity and state.
    - Re-render based on state changes.
  - Both types contribute to the same view.



THE **GOOD AND BAD** OF THE RSC ARCHITECTURE

THE GOOD

- 👍 We can compose entire full-stack apps **with React components alone** (+ *server actions* 👉)
- 👍 **One single codebase** for front and back-end
- 👍 Server components have **more direct and secure access** to the data source (no API, no exposing API keys, etc.)
- 👍 **Eliminate client-server waterfalls** by fetching all the data needed for a page **at once** before sending it to the client (*not* each component)
- 👍 **"Disappearing code":** server components ship no JS, so they can import huge libraries "for free"

THE BAD

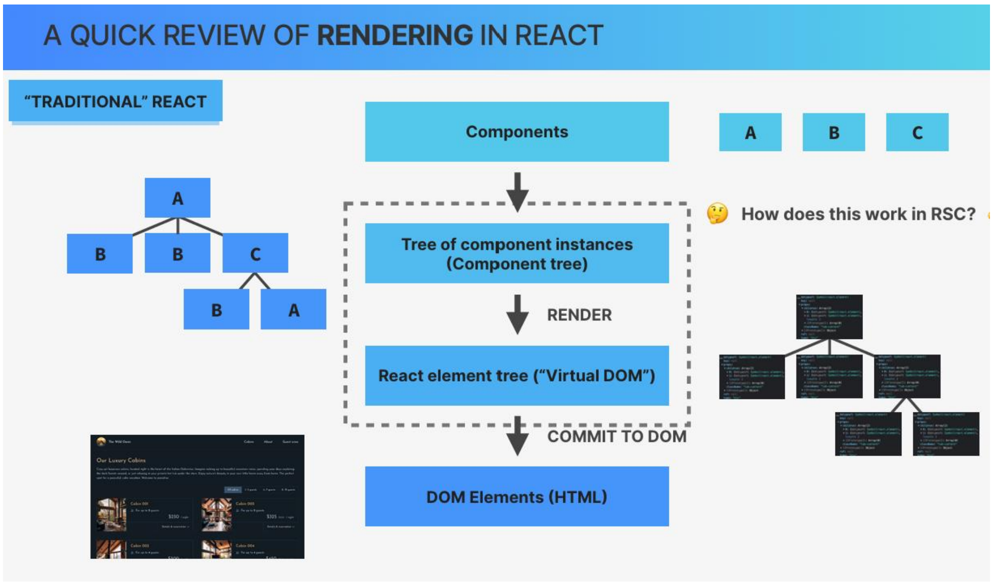
- 👎 Makes React **more complex**
- 👎 More things to **learn and understand**
- 👎 Things like **Context API** don't work in server components
- 👎 **More decisions** to make: "Should this be a client or a server component?", "Should I fetch this data on the server or the client?", etc.
- 👎 Sometimes you still need to **build an API** (for example if you also have a mobile app)
- 👎 Can only be used within a **framework**

Pros of RSC

1. **Full-Stack React**:
   - Write frontend and backend code in React.
   - Encapsulate server-side concerns in components.
2. **Reduced JavaScript**:
   - Server components ship **zero JavaScript**.
   - Import large libraries (e.g., CMS, syntax highlighting) without increasing bundle size.
3. **Eliminate Client-Server Waterfalls**:
   - Fetch all data on the server at once.
4. **Direct Data Access**:
   - Access databases or APIs directly from server components.
   - No need for a separate API in many cases.
5. **Improved Security**:
   - API keys and sensitive data stay on the server.

Cons of RSC

1. **Increased Complexity**:
   - More concepts to learn (e.g., boundaries, serialization).
   - Decisions required (e.g., client vs. server components).
2. **Framework Dependency**:
   - RSC requires a framework (e.g., Next.js, Remix).
   - Cannot be implemented in a simple Vite app.
3. **Limited APIs**:
   - Context API and hooks don’t work in server components.
4. **Mobile App Considerations**:
   - May still need an API for mobile apps.



A QUICK REVIEW OF **RENDERING** IN REACT

“TRADITIONAL” REACT

Components

Tree of component instances (Component tree)

RENDER

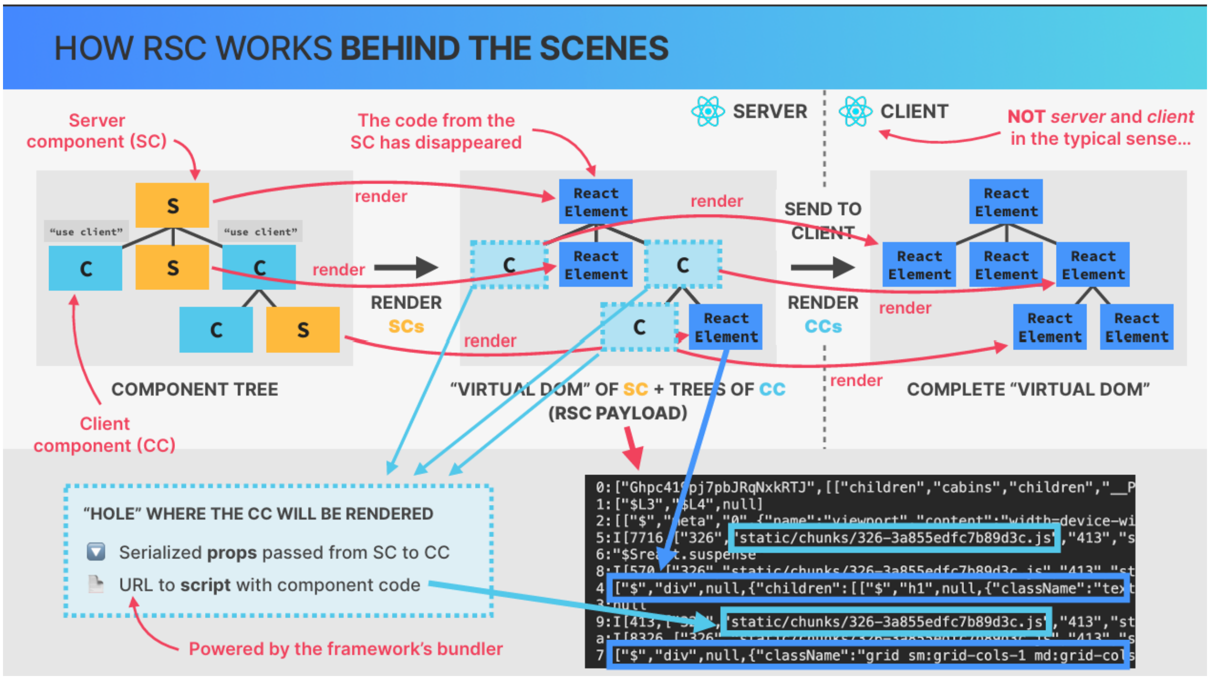
React element tree (“Virtual DOM”)

COMMIT TO DOM

DOM Elements (HTML)

🤔 How does this work in RSC?

1. **Component Tree**:
   - Composed of component instances (e.g., A, B, C).
2. **Rendering**:
   - Each component function is called, producing **React elements** (JavaScript objects).
   - React elements form the **virtual DOM**.
3. **Commit to DOM**:
   - Virtual DOM is committed to the actual DOM, creating visible UI



```
0:["Ghpc419pj7pbJRqNxkRTJ",[["children","cabins","children","__P
1:["$L3","$L4",null]
2:[["$","meta","0",{"name":"viewport","content":"width=device-wi
5:I[7716,["326","static/chunks/326-3a855edfc7b89d3c.js","413","s
6:"$Sreact.suspense
8:I[570,["326","static/chunks/326-3a855edfc7b89d3c.js","413","st
4:["$","div",null,{"children":[["$","h1",null,{"className":"text
3:null
9:I[413,["326","static/chunks/326-3a855edfc7b89d3c.js","413","st
a:I[8326,["326","static/chunks/326-3a855edfc7b89d3c.js","413","s
7:["$","div",null,{"className":"grid sm:grid-cols-1 md:grid-cols
```

Rendering in React Server Components (RSC)

- **Component Tree**:
  - Contains both **server components (SC)** and **client components (CC)**.
- **Step 1: Server-Side Rendering**:
  - **Server Components**:
    - Rendered on the server.
    - Produce **React elements** (virtual DOM).
    - Code disappears after rendering (no state or hooks).
    - Output is serializable (no functions or classes).
  - **Client Components**:
    - Not rendered on the server.
    - **Placeholders are created**:
      - Contain **serialized props** (from server components).(important)
      - Include a **URL** to the component's script (for client-side rendering).
  - **RSC Payload**:
    - A mix of rendered server components and placeholders for client components.
    - Sent to the client as a **streamable JSON-like data structure**.

Explanation of code disappearing: (TMI)

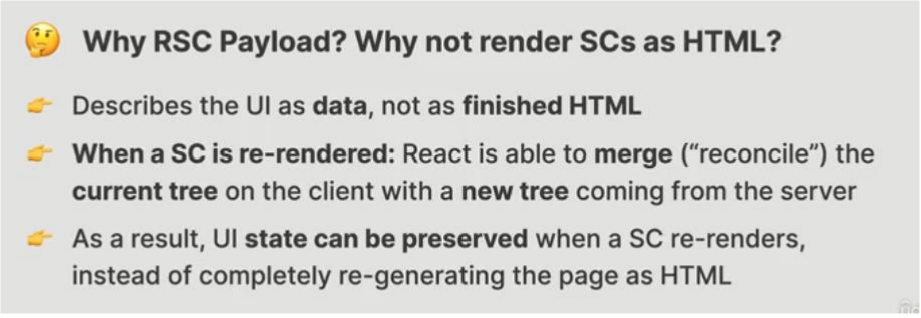
- Server components are **executed on the server**, not in the browser.
- When a server component is rendered**, React calls its function** and produces **React elements**
- After the server component is rendered, **the actual code of the component is not sent to the browser**.
    - For example, if you have a server component that fetches data from a database and renders a list, the logic for fetching data and rendering the list **stays on the server**.
    - Only the **output** (the React elements, i.e., the virtual DOM) is sent to the client.
- This is why server components **cannot use state or hooks**:
    - Hooks like useState or useEffect are JavaScript functions that need to run in the browser.
    - Since the server component code doesn't exist in the browser, these hooks would have nowhere to run.

Output is Serializable

- The output of server components (React elements) is **serialized** into a format that can be sent from the server to the client.
    - Serialization means converting data (like React elements) into a format that can be transmitted (e.g., JSON-like structure).
    - **Functions and classes cannot be serialized**, which is why they cannot be used in server components or passed as props to client components.

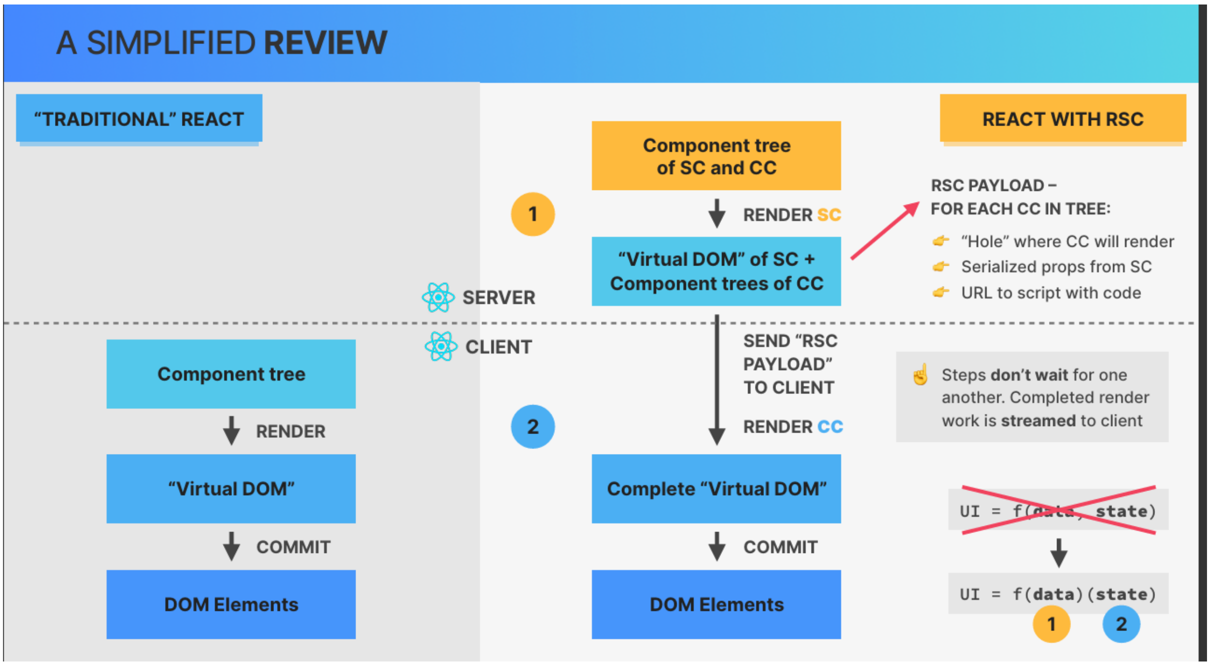
- **Step 2: Client-Side Rendering**:
    - **Client Components**:
        - Rendered on the client using the script URL and props from the RSC payload.
        - Produce React elements, completing the virtual DOM.
    - **Commit to DOM**:

        - Final virtual DOM is committed to the actual DOM (same as traditional React).



🤔 **Why RSC Payload? Why not render SCs as HTML?**

- Describes the UI as **data**, not as **finished HTML**
- **When a SC is re-rendered:** React is able to **merge** (“reconcile”) the **current tree** on the client with a **new tree** coming from the server
- As a result, UI **state can be preserved** when a SC re-renders, instead of completely re-generating the page as HTML



A SIMPLIFIED **REVIEW**

“TRADITIONAL” REACT

Component tree

RENDER

“Virtual DOM”

COMMIT

DOM Elements

SERVER

CLIENT

1

Component tree of SC and CC

RENDER SC

“Virtual DOM” of SC + Component trees of CC

SEND “RSC PAYLOAD” TO CLIENT

2

RENDER CC

Complete “Virtual DOM”

COMMIT

DOM Elements

REACT WITH RSC

RSC PAYLOAD – FOR EACH CC IN TREE:

- “Hole” where CC will render
- Serialized props from SC
- URL to script with code

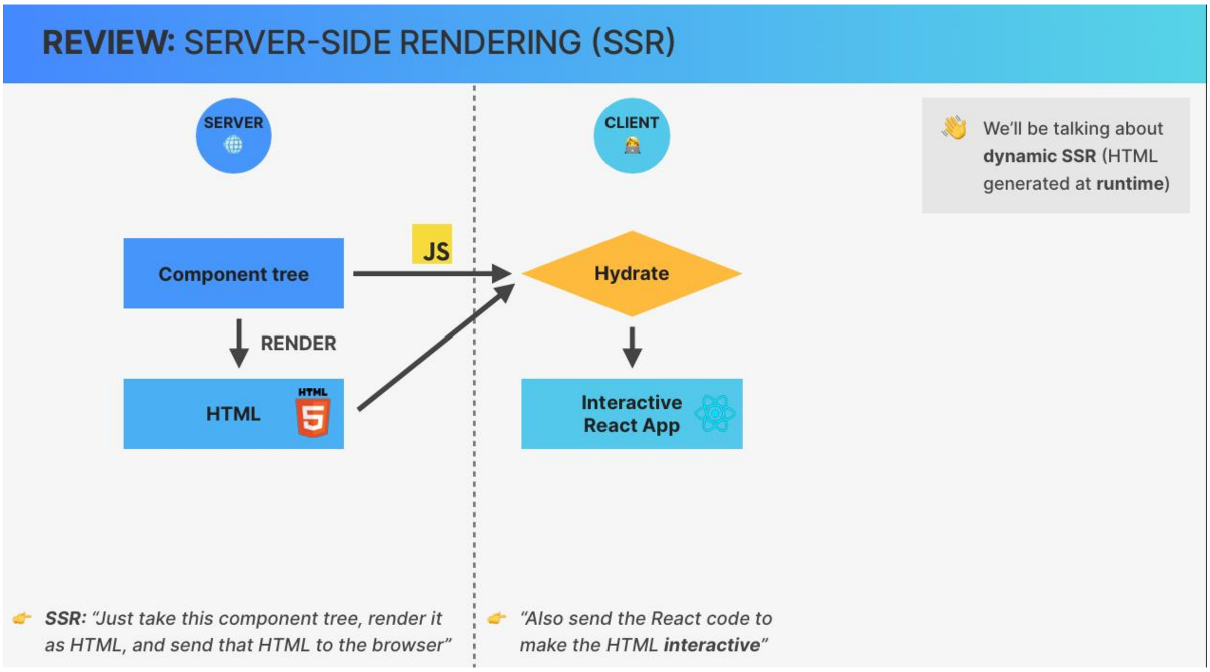
☝ Steps **don’t wait** for one another. Completed render work is **streamed** to client

~~`UI = f(data, state)`~~

`UI = f(data)(state)`

1 2

- RSC introduces a **two-step rendering process**:
    - Server components handle data fetching and initial rendering.
    - Client components handle interactivity and state.



REVIEW: SERVER-SIDE RENDERING (SSR)

SERVER

CLIENT

Component tree

JS

RENDER

HTML

Hydrate

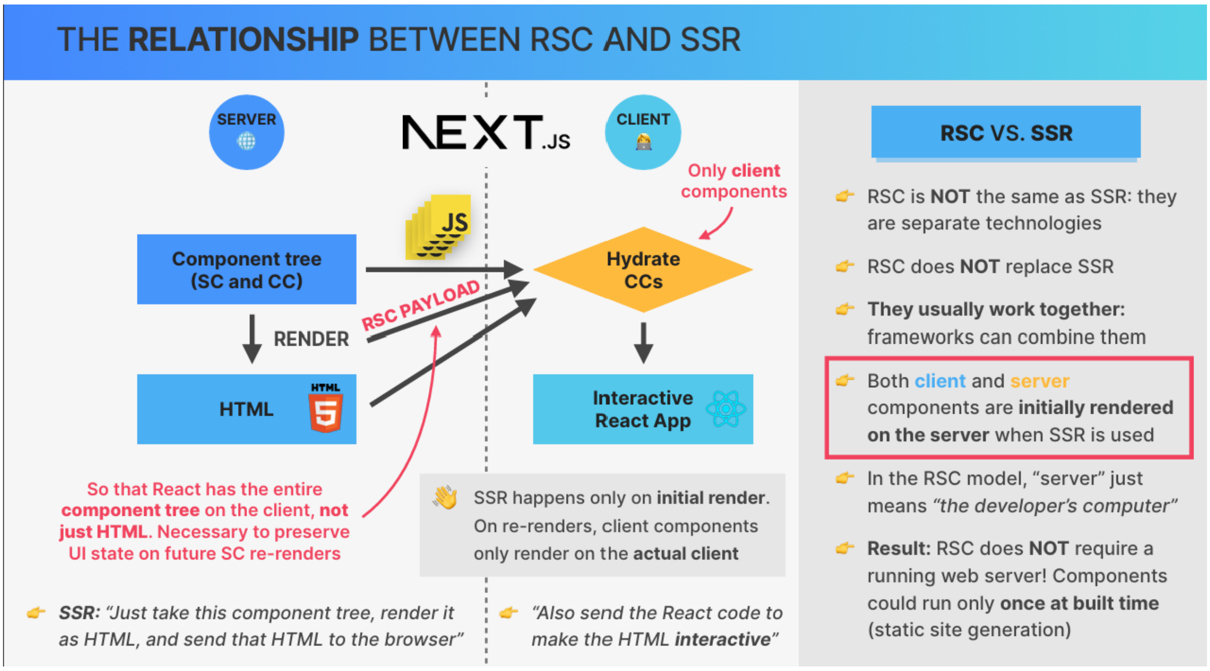
Interactive React App

👋 We'll be talking about **dynamic SSR** (HTML generated at **runtime**)

👉 ***SSR:*** *"Just take this component tree, render it as HTML, and send that HTML to the browser"*

👉 *"Also send the React code to make the HTML **interactive**"*

1. Notes: **What is SSR?**
   - **Dynamic SSR**: HTML is generated on the server for each incoming request.
   - **Process**:
     - Start with a **component tree**.
     - Render the tree to the **virtual DOM**.
     - Convert the virtual DOM to **HTML** and send it to the client (browser).
   - **Hydration**:
     - The React bundle (React + component code) is sent to the browser.
     - The HTML is **hydrated** (made interactive) in the browser.



THE **RELATIONSHIP** BETWEEN RSC AND SSR

So that React has the entire **component tree** on the client, **not just HTML**. Necessary to preserve UI state on future SC re-renders

👋 SSR happens only on **initial render**. On re-renders, client components only render on the **actual client**

👉 ***SSR:*** *"Just take this component tree, render it as HTML, and send that HTML to the browser"*

👉 *"Also send the React code to make the HTML* ***interactive****"*

RSC VS. SSR

👉 RSC is **NOT** the same as SSR: they are separate technologies

👉 RSC does **NOT** replace SSR

👉 **They usually work together:** frameworks can combine them

👉 Both client and server components are **initially rendered on the server** when SSR is used

👉 In the RSC model, "server" just means *"the developer's computer"*

👉 **Result:** RSC does **NOT** require a running web server! Components could run only **once at built time** (static site generation)

Key Concepts: React Server vs. React Client

1. **React Server**:
   - Not necessarily a traditional web server.
   - Can be any environment where code is executed (e.g., build-time static site generation).
   - Responsible for rendering **server components** and generating the **RSC payload**.
2. **React Client**:
   - Not necessarily a traditional web browser.
   - Consumes the rendered React app (e.g., as HTML during SSR).
   - In SSR, the React client runs on the server to produce HTML.

Rendering Process with SSR and RSC

1. **Initial Render (SSR)**:
   - **Component Tree**:
     - Contains both **server components** and **client components**.
   - **Rendering**:

        - Server components are rendered on the **React server**.
        - Client components are rendered on the **React client** (which runs on the server during SSR).
    - **Output**:
        - **HTML** is generated and sent to the browser.
        - **React Bundle** (chunks of code) is sent to the browser for hydration.
        - **RSC Payload** is sent to the browser (contains rendered server components and client component placeholders).

Additional note:

RSC Payload (Sent from Server)

- It contains **fully rendered HTML for server components**.
- It includes **placeholders for client components**, but these are not rendered on the server.
- These placeholders contain **links to JavaScript files** (which are part of the React Bundle).

2. **Hydration**:
    - The HTML is hydrated in the browser using the React bundle and RSC payload.
    - Only **client components** are hydrated (made interactive).
3. **Subsequent Renders**:
    - After the initial render, **server components** run on the actual web server.
    - **Client components** run in the browser.
    - New RSC payloads are generated and sent to the browser when server components re-render.