

ITS A NEW PARADIGM ,DONT CONFUSE THIS WITH Server components

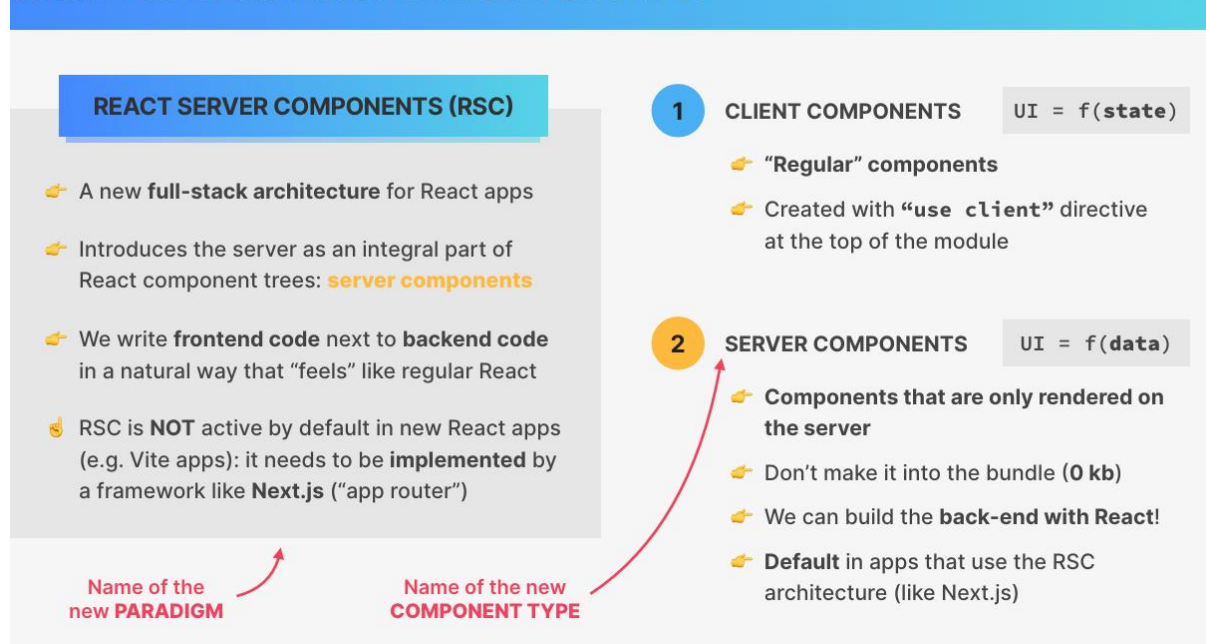
Notes:

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?



- 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:

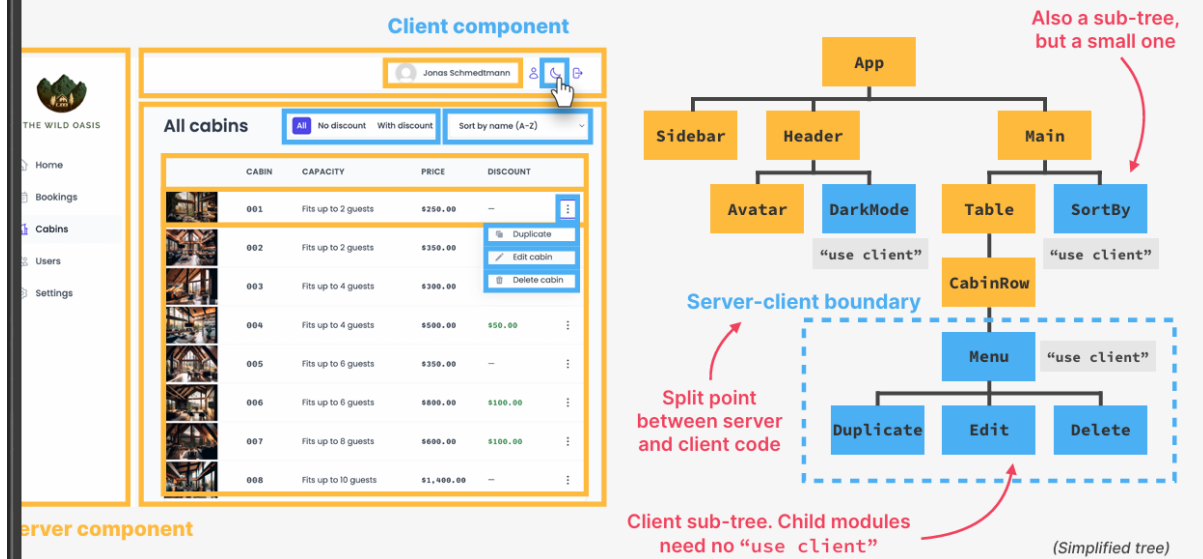
- Require JavaScript in the browser.

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



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)

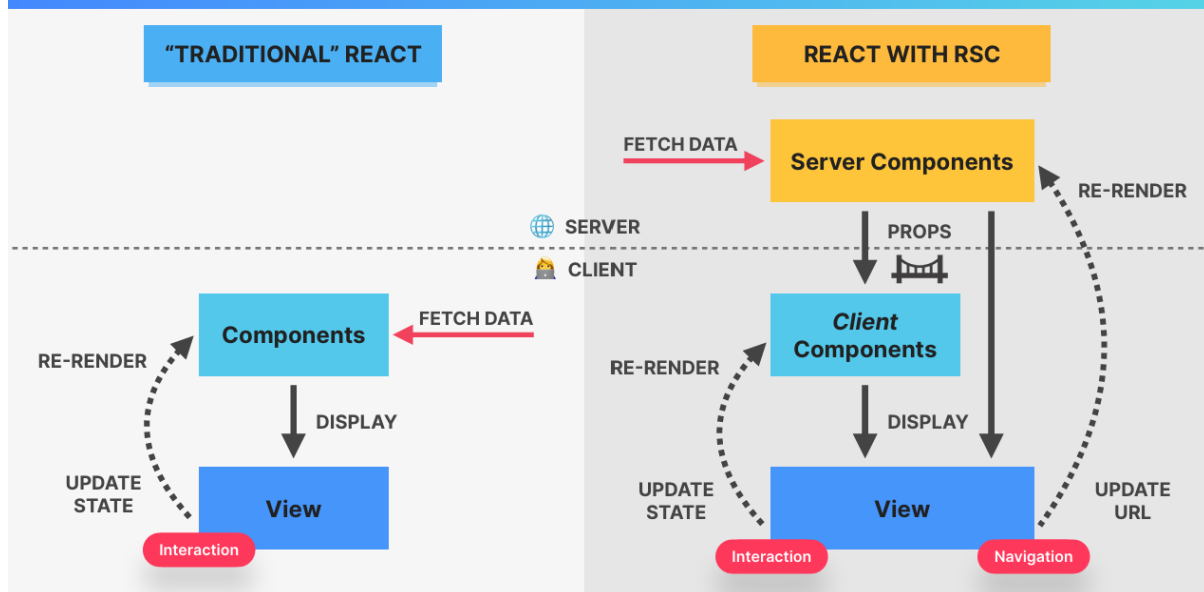
Comparison: Server vs. Client Components

Aspect	Server Components	Client Components
Default/Opt-in	Default in RSC.	Opt-in using <code>use client</code> .
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 <code>async/await</code> 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 <i>passed as props</i>	Client and server components
🔄 When re-render?	On state change	On URL change (navigation)

A SIMPLE NEW MENTAL MODEL



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:

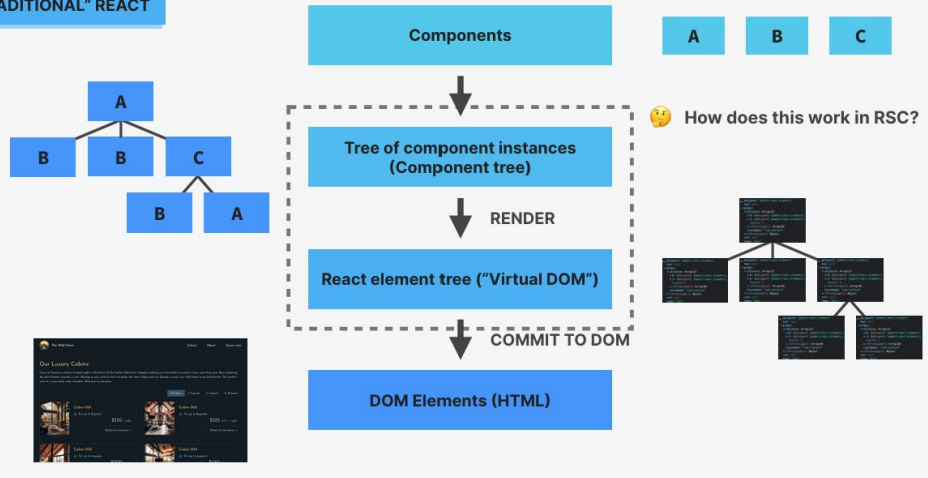
- Context API and hooks don't work in server components.

4. Mobile App Considerations:

- May still need an API for mobile apps.

A QUICK REVIEW OF RENDERING IN REACT

"TRADITIONAL" REACT



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

[illegible]

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:**

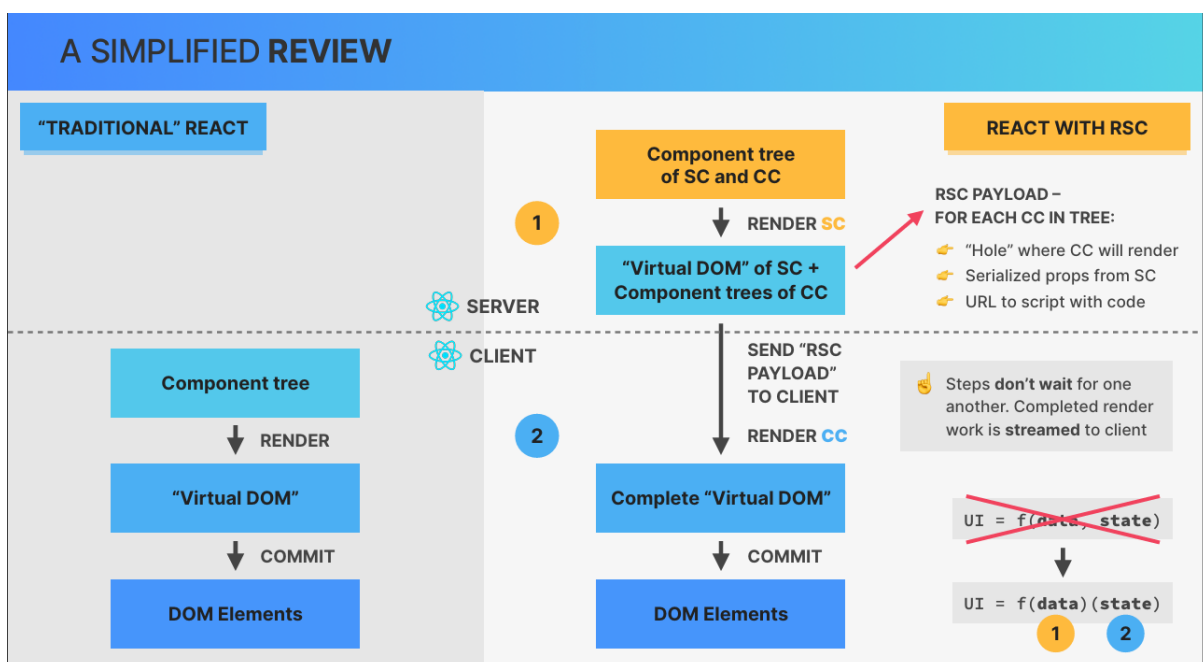
- Rendered on the client using the script URL and props from the RSC payload.
- Produce React elements, completing the virtual DOM.

- **Commit to DOM:**

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

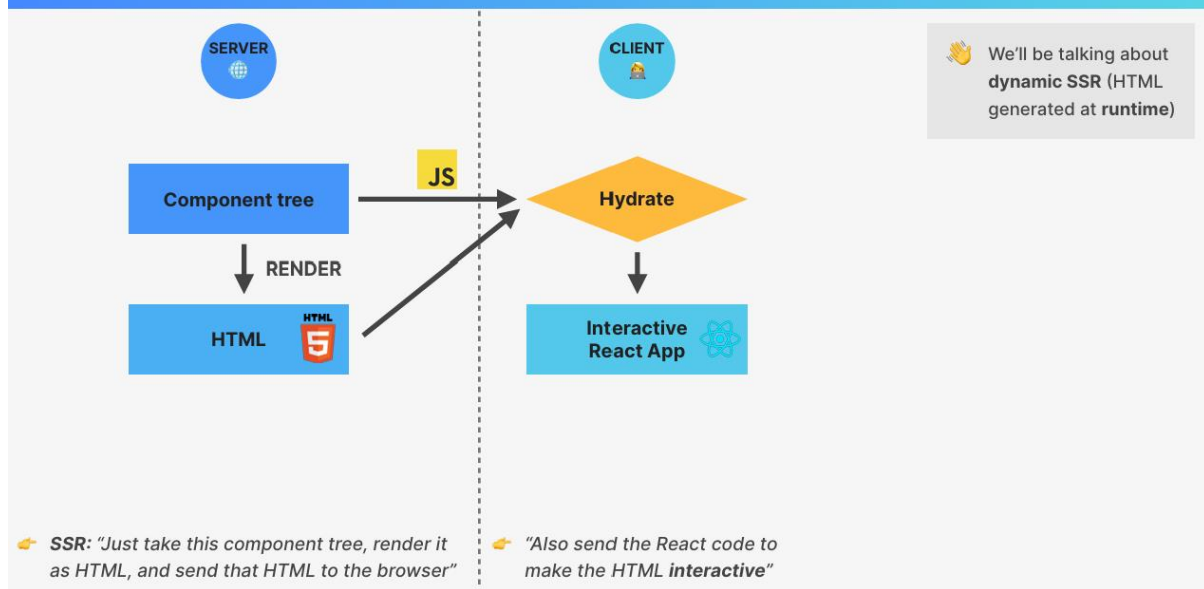
🤔 **Why RSC Payload? Why not render SCs as HTML?**

- 👉 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



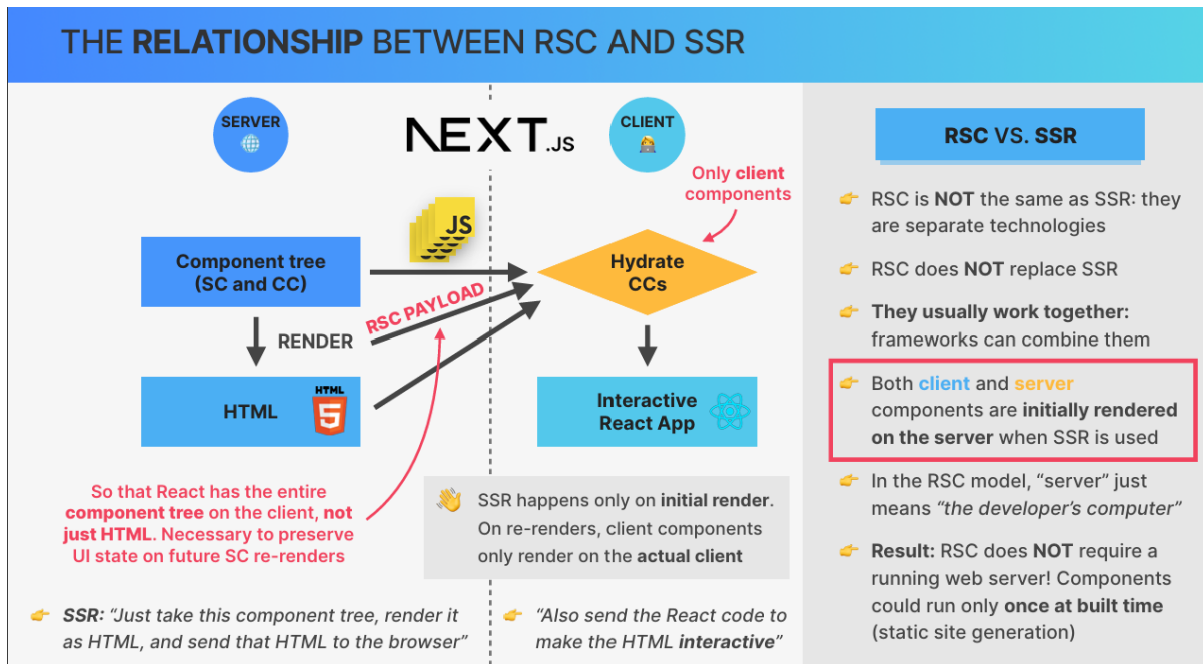
- RSC introduces a **two-step rendering process**:
 - Server components handle data fetching and initial rendering.
 - Client components handle interactivity and state.

REVIEW: SERVER-SIDE RENDERING (SSR)



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.



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.

