In a Frontend System Design interview, this is a common point of confusion. The short answer is: **The App Router is the first and currently only "production-ready" way to use RSCs in the React ecosystem.**

Technically, RSC is a **React feature**, not a Next.js feature. However, because RSCs require a deep integration between the "Server" (Node.js) and the "Bundler" (Webpack/Turbopack/Vite), you cannot simply "enable" them in a standard Vite or CRA project yet.

---

1. Why the App Router is required

To implement RSCs, you need a "React Server Protocol" implementation. This protocol handles:

- **The Component Split:** Deciding which code stays on the server and which goes to the client.
- **The Serialized Stream:** Taking the server-rendered components and turning them into the **RSC Payload** (a special JSON-like stream).
- **Client-Side Stitching:** Taking that stream in the browser and updating the DOM without refreshing the page.

Next.js built the **App Router** specifically to be the reference implementation for this architecture. The older **Pages Router** cannot support RSCs because its architecture is fundamentally built on the "Server-Side Rendering (SSR) then Hydrate Everything" model.

---

2. RSC Implementation: Basic Pattern

In the App Router, you don't "import" RSCs differently. You simply write async components.

The Server Component (The Default)

```
// app/users/page.tsx
// This is an RSC. No 'use client' directive.

import { db } from '@/lib/db'; // Direct DB access!

export default async function UsersPage() {
  // This code runs ONLY on your server.
  // It never reaches the user's browser.
  const users = await db.user.findMany();

  return (
    <div>
      <h1>User Directory</h1>
      <ul>
        {users.map(user => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
      {/* We nest a Client Component for interactivity */}
      <FilterButton />
    </div>
  );
}
```

The Client Component (The "Island")

```
// components/FilterButton.tsx
'use client'; // This creates the "Boundary"

import { useState } from 'react';

export default function FilterButton() {
  const [isOpen, setIsOpen] = useState(false); // Hooks allowed here!
```

```
  return (
    <button onClick={() => setIsOpen(!isOpen)}>
      {isOpen ? 'Close Filters' : 'Open Filters'}
    </button>
  );
}
```

---

3. The "Children" Pattern (Interview Must-Know)

A common interview question is: *"How do I put a Server Component inside a Client Component if I can't import it?"*

If you import an RSC into a Client Component, it is treated as a Client Component and loses its RSC benefits (it will hydrate). To keep the inner component "Server-only," you must pass it as a **prop** (usually children).

```
// ClientWrapper.tsx ('use client')
export default function ClientWrapper({ children }) {
  return <div className="interactive-shell">{children}</div>;
}

// page.tsx (Server Component)
export default function Page() {
  return (
    <ClientWrapper>
      <ServerComponent /> {/* This stays a Server Component! */}
    </ClientWrapper>
  );
}
```

---

4. The RSC Lifecycle (Behind the Scenes)

When a user navigates to an RSC page in the App Router:

1. **Request:** The browser asks for the page.
2. **Render:** The server renders the RSC tree.
3. **Serialization:** Instead of HTML, the server generates the **RSC Payload**.
4. **Stitching:** The browser receives the payload. React's client-side runtime updates the DOM, preserving the state of existing Client Components (like an open dropdown) while the new Server content arrives.

---

5. Interview Q&A: RSC Implementation

Q: Can I use RSC without Next.js?
A: Theoretically, yes. Frameworks like Waku or Hydrogen (Shopify) also implement RSC. You could also build your own "Flight" implementation using React's low-level server APIs, but it is extremely complex and not recommended for most teams.

Q: Why doesn't the Pages Router support RSC?
A: The Pages Router is based on the _app.js and _document.js lifecycle, which assumes the entire component tree will be hydrated in the browser. RSC requires a "Server-First" routing system that can handle partial updates, which is what the App Router provides.

Q: Is RSC just another way of writing APIs?
A: No. In the CSR/SSR world, you fetch data (JSON) and turn it into UI (HTML) on the client. In RSC, you fetch UI directly. The "data" stays on the server; the client only receives the "result" of the UI logic.

Summary Checklist for your Interview:

- **App Router** = The primary implementation of RSC.
- **Default** = All components are RSCs unless marked 'use client'.
- **Benefits** = Zero bundle size for static parts, no hydration for server parts, no "waterfalls."
- **The Catch** = No hooks (useState, useEffect) in RSCs.