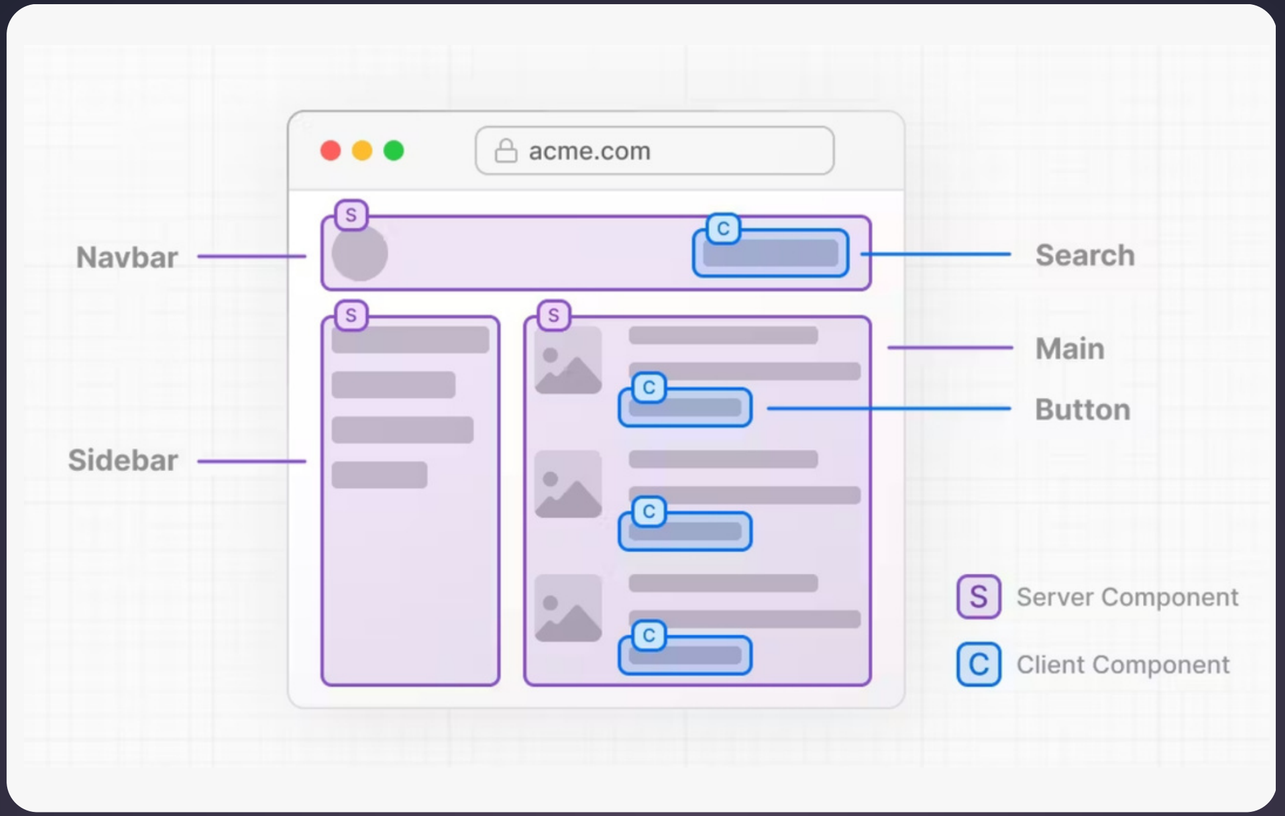
Learn with Rabbil

NEXT JS

FOR NEXT GENERATION

Thinking in next js

Instead of rendering your whole application **client-side** or whole in **server-side**. Go for a better **combination** based on components purpose.



WHY ?

Server Components

Secure data fetching to the server.
Calling SMS/Payment/B2B or other server depended API, that can't call form client-side.

To manage **database operation**. Keep large dependencies that can impact user .

Minimize client JavaScript **bundle size**. Faster initial page load & Better SEO

Server Components make writing a application **feel similar** to PHP/Laravel, C#/ASP or Ruby on Rails.

WHY ?

Client Components

To add **client-side interactivity** to your application line onClick, onChange events, useState,useRef,useEffect hooks, transition, animations and other's.

The **"use client"** directive must be defined at the top of a file before any imports.

In Next.js client-component **pre-rendered** on the server and **hydrated** on the client.

Pre-rendering on the server and hydrated on the client

Learn with Rabbil

Traditional React applications

Client download all of the
JavaScript code and assets

Entire application is
rendered on the client side

Pre-rendering on the server and hydrated on the client

Render the React application on the server
before it is sent to the client

Client only has to download the initial
HTML and CSS for the application

Hydration is the process of taking the pre-rendered
HTML and attaching event listeners and state
to it on the client side.

Server Component Nesting

- Server Components can be nested within other Server Components.
- Server Components can contain Client Components, but these Client Components would be re hydrated on the client side after the initial render.
- Server Components can also contain regular HTML and other content.

Client Component Nesting

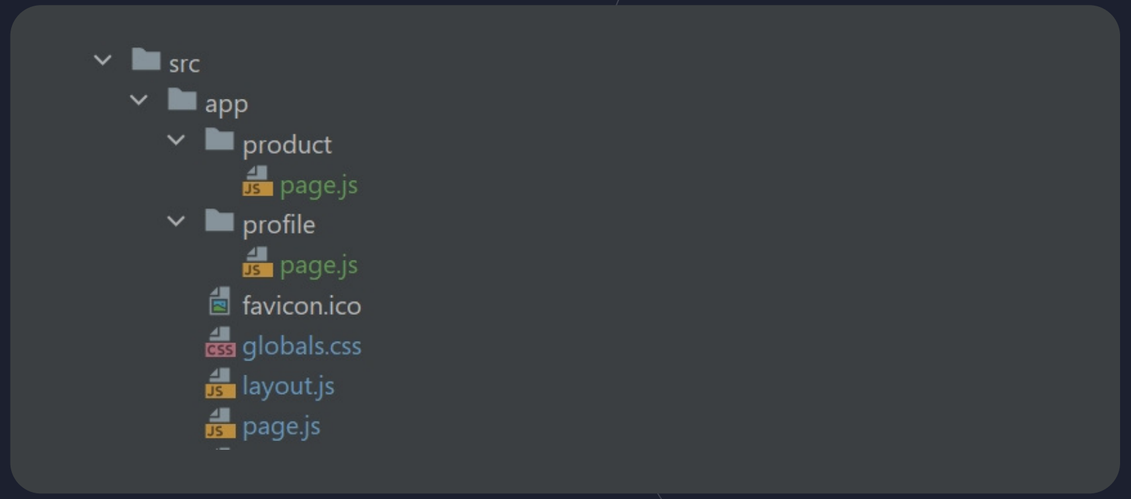
- Client Components can be nested within other Client Components.
- Client Components can contain Server Components, but the server-rendered output of these components would be "hydrated" on the client side, meaning the JavaScript code would take over and make them interactive.

Learn with Rabbil

- RootLayout is the **top level of the app** directory and applies to all routes
- The app directory **must include** a root layout.
- The root layout must define **<html> and <body>** tags
- Any component or assets inside RootLayout will **available** entire the application
- RootLayout is only one and **priority first**

layout.js

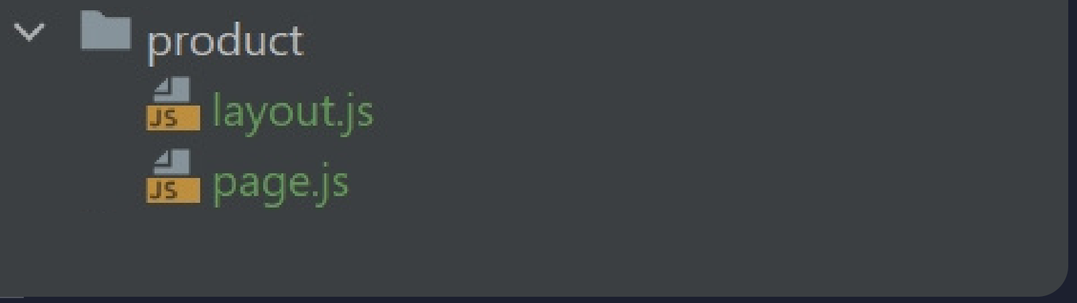
```
import './globals.css'
import AppNavBar from "@/component/AppNavBar";
import AppFooter from "@/component/AppFooter";
export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
      <AppNavBar/>
        {children}
      <AppFooter/>
      </body>
    </html>
  )
}
```

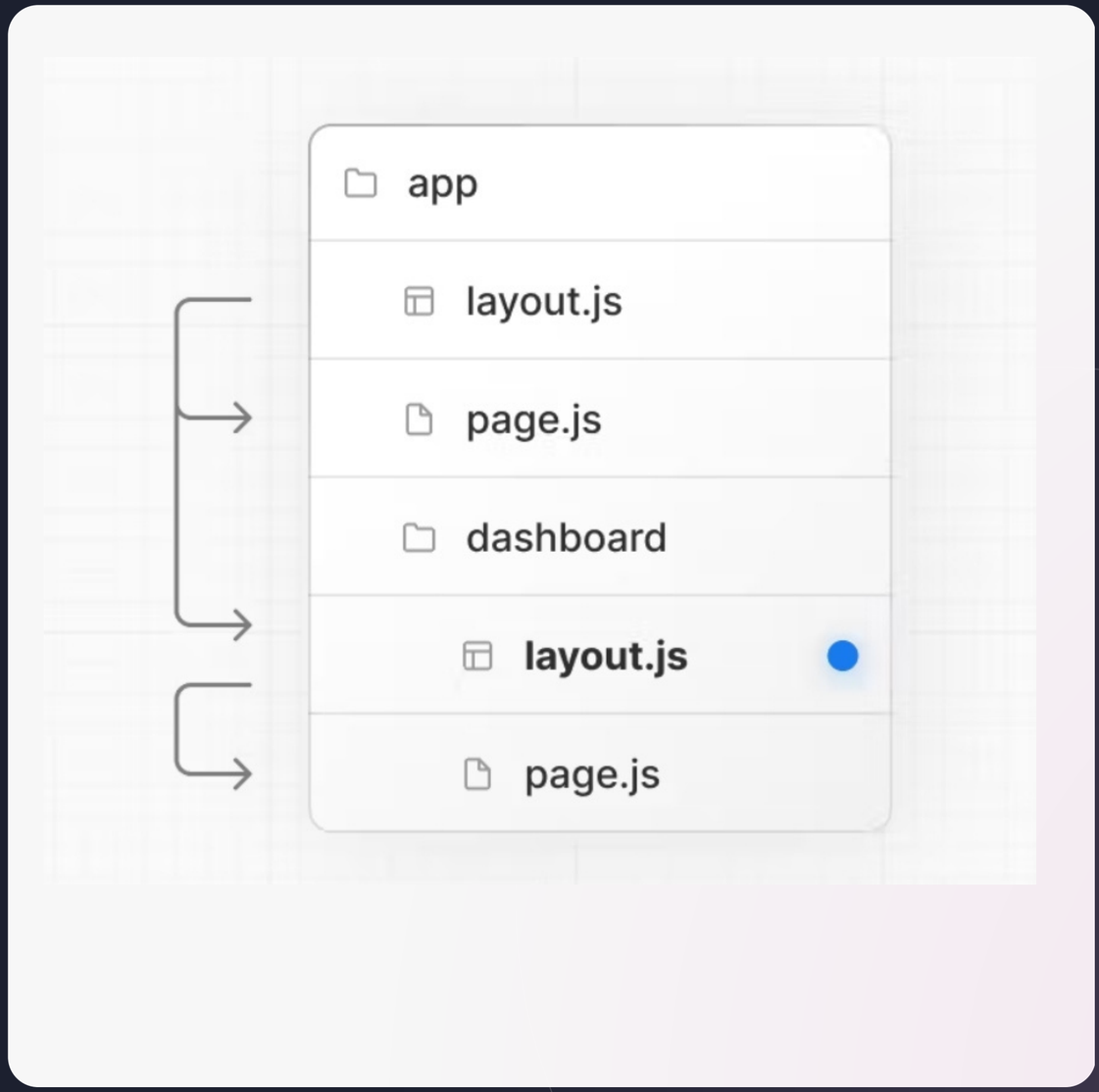


Learn with Rabbil

layout.js

```
1   const Layout=({ children }) => {
2       return (
3           <div>
4               <h1>Product Layout </h1>
5               {children}
6           </div>
7       );
8   };
9   export default Layout;
10
```





Directory File Conventions

Learn with Rabbil

page.js

- A page is UI that is **unique** to a route.
- Special file that is used to **define a page** in your application.
- The page.js file must **export a component.**
- The component **name must match** the file name.
- The component can be **any kind of React component**, including functional components, class components, and custom hooks.
- The component **can access** the props object, which contains information about the route, such as the route path, the route parameters, and the route query parameters.
- The component can be rendered **server-side or client-side**

Directory File Conventions

Learn with Rabbil

layout.js

- A layout is UI that is **shared** between routes.
- It can be used to render **common components** or functionality across **multiple pages** in your application
- For example, you could use a layout to render a **header and footer on every page** in your application

not-found.js

- The not-found file is used to render UI when the **notFound function** is thrown within a route segment.
- Along with serving a custom UI, Next.js will also return a 404 HTTP status code.
- The root app/not-found.js file handles any unmatched URLs for your whole application.
- not-found.js components **do not accept any props.**

Directory File Conventions

Learn with Rabbil

error.js

- An error file defines an **error UI boundary** for a route segment
- It is useful for **catching unexpected errors** that occur in Server Components and Client Components and displaying a fallback UI
- error.js boundaries must be **Client Components**.
- In Production builds, errors forwarded from Server Components will be stripped of specific error details **to avoid leaking** sensitive information.

loading.js

- A loading file can create instant **loading states built**
- By default, this file is a **Server Component**
- Can also be used **as a Client Component** through the "use client" directive
- Loading UI components **do not accept any parameters**.

Directory File Conventions

Learn with Rabbil

route.js

- Route Handlers allow you to create custom **request handlers** for a given route using the Web Request and Response
- HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS
- Use to manage **back-end**

Data facing

Learn with Rabbil

page.js

```
async function getData() {
    const res = await fetch('https://dummyjson.com/products/1')
    return res.json()
}
const Page = async () => {
    const data = await getData()
    return (
        <div>
            <h1>{JSON.stringify(data)}</h1>
        </div>
    );
};
export default Page;
```

On the server

page.js

```
'use client'
import {useEffect, useState} from "react";
const Page = async () => {
    const [data,setData]=useState([])
    useEffect(()=>{
        (async ()=>{
            const res = await fetch('https://dummyjson.com/products')
            const json =  await res.json()
            setData(json);
        })()
    },[])
    return (
        <div>
            <h1>{JSON.stringify(data)}</h1>
        </div>
    );
};
export default Page;
```

On the client

Caching & Revalidating

```
const res1 = await fetch( input: 'https://dummyjson.com/products', init: { cache: 'force-cache' })
const res2 = await fetch( input: 'https://dummyjson.com/products', init: { cache: 'no-store' })
const res4 = await fetch( input: 'https://dummyjson.com/products', init: { next: { revalidate: 3600 } })
```

Error Handling While Data Fetching

Learn with Rabbil

page.js

```
async function getData() {
    const res = await fetch('https://dummyjson.com/products')
    const json =  await res.json()
    if (!res.ok) {
        // This will activate the closest `error.js` Error Boundary
        throw new Error('Failed to fetch data')
    }
    return res.json()
}
```

Multiple Data Fetching

Learn with Rabbil

page.js

```
async function getData() {
    try {
        const res1 = await fetch('https://dummyjson.com/products/1')
        const res2 = await fetch('https://dummyjson.com/products/2')
        const json1 =  await res1.json()
        const json2 =  await res2.json()
        return {json1:json1,json2:json2}
    }
    catch (e) {
       throw new Error('Failed to fetch data')
    }
}
```

Data Fetching Read Header/Body/Status

page.js

```
async function getData() {
    try {
        const res = await fetch('https://crud.teamrabbil.com/api/v1/ReadProduct')
        let resBody=await  res.json();
        let resHeader=res.headers.get('x-ratelimit-limit');
        let resStatus=res.status;
        return {Body:resBody,Header:resHeader,Status:resStatus}
    }
    catch (e) {
       throw new Error('Failed to fetch data')
    }
}
```