localhost:3000 -> app/page.tsx -> routes
localhost:3000/about -> app/about/page.tsx -> nested routes
localhost:3000/blogs/[id] -> app/blogs/page.tsx -> dynamic routes

```
const id = await params();
id -> what we get from the url id
```

```
                                                   -> nested dynamic routes
```

localhost:3000/products/[id]/reviews/[id] -> app/products/page.tsx
app/products/reviews/page.tsx

```
                                                         catch all segments
```

localhost:3000/docs/feature1/concept1 -> app/docs/[...slug]/page.tsx
-> here slug will catch all the route params after docs
-> here if i go to the /docs -> no page is there

```
                                                  optional catch all segments
```

localhost:3000/docs/feature1/concept1 -> app/docs/[...slug]/page.tsx
-> here slug will catch all the route params from docs
-> here if i go to the /docs ->there will be a /docs page also

```
                                                           NotFound Page
```

in the app folder create a not-found.tsx page & design it accordingly.
there is notfound function also -> to use it

```
                                                           private folders
```

if there is any folder inside src/app & we dont want to make it a url route, then make it _folderName

```
                                                              route groups
```

if there are multiple folders like -> login, register, forgot password -> then to keep all the folders in place, we can group them together , if we do something like this ->
localhost:3000/auth/login
localhost:3000/auth/register
localhost:3000/auth/forgotpassword -> it can work, but we don't need the auth route,
to keep the auth route, outside -> make it ->
(auth)
src/app/(auth)/login
src/app/(auth)/register
src/app/(auth)/forgotpassword

```
                                                  Layouts
```

layout.tsx ->

in the Root Layout,
children becomes the page.tsx and all the localhost:3000/... routes

if we add the Header component and Footer component like this ->

{children}
all the routes will get the header and footer in the whole project

```
                                          multiple root layout
```

make route groups, change the root layout function name to routeGroup name, and move the root
page.tsx to the same route group , this will changed page.tsx will act as root page.tsx as localhost:3000/ -> route

```
                                             generate Metadata
```

```
    import {Metadata}  from 'next';
    type Props={
        params:Promise<{productId:string}>;
    }
    export const generateMetaData = async({
    params
    }:Props):Promise<Metadata>{
        const id = (await params).productId;
        const title = await new Promise((res)=>{
            setTimeout(()=>{
                resolve(`iphone ${id}`);
            },100);
        });

        return {
            title:``Product ${title}
        };
    };
```

this is the way to fetch some data and use that to generate metadata;

```
about metadata
```

the metadata const can't be in a client component.

if we need to use that and decompose the client component into server component.

```
    export const metadata={
        title:"Counter",
    };

    export default function CounterPage(){
        return <Counter/>;
    }
```

```
title MetaData
```

```

export const metadata={
        title:{
            default:"Next.js Tutorial",
            template:"%s | Coder",
            absolute: -> this we will directly give into the child components
        },
        description:"Generated by me"
    };
```

active link search in nextjs -> usePathName();

```
params and searchParam
```

for client comp

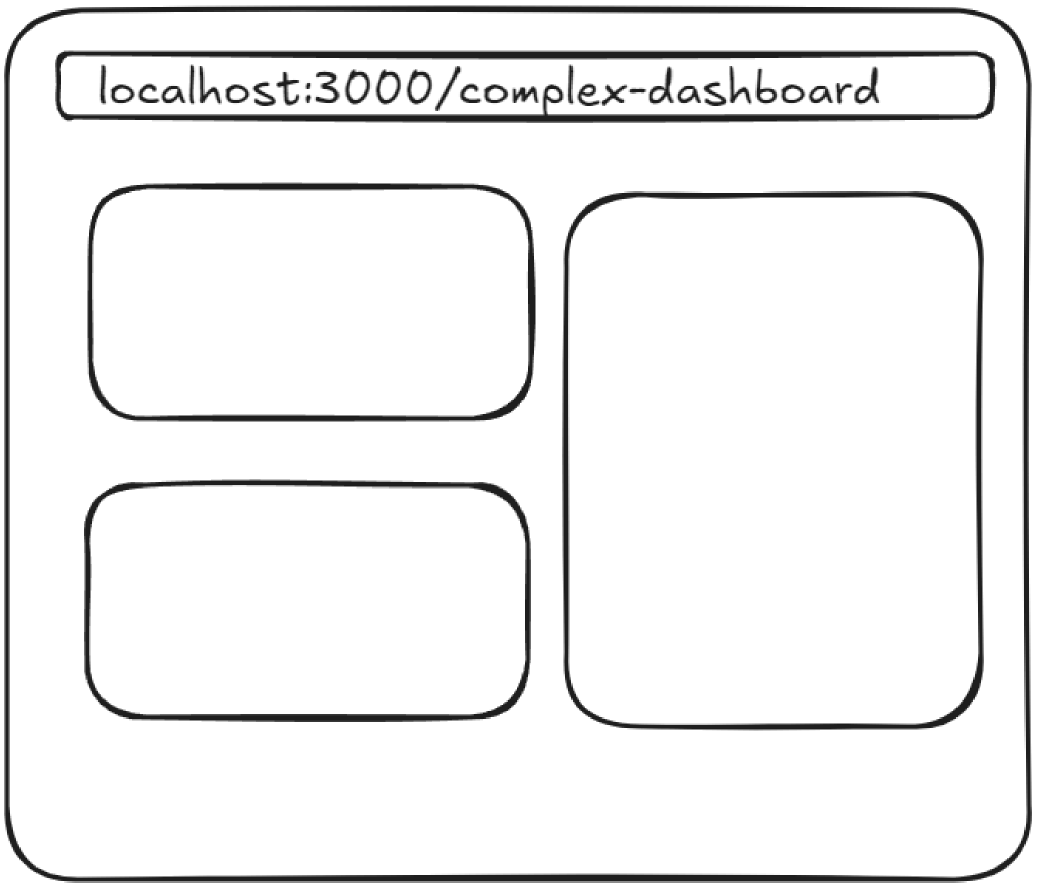
export default fn newsArticle({params,searchParams}:{params:Promise<{articleId:string}>;}
searchParams:Promise<{lang?:"en"|"es"}>;
){
const {articleId} = use(params);
const {lang = "en"} = use(searchParams);
}

for server components

export default async fn newsArticle({params,searchParams}:
{params:Promise<{articleId:string}>;}
searchParams:Promise<{lang?:"en"|"es"}>;
){
const {articleId} = await params;
const {lang = "en"} = await searchParams;
}

page.tsx -> params & searchParams;
layout.tsx -> params, (no searchParams)

```
parralel routes
```



loading 3-4 separate components in the same page.
file structure ->
app/dashboard/@notifications/page.tsx
app/dashboard/@revenue/page.tsx
app/dashboard/@users/page.tsx
@filename is the slots
these page.tsx components are not imported in the /dashboard/page.tsx, these are passed as props.

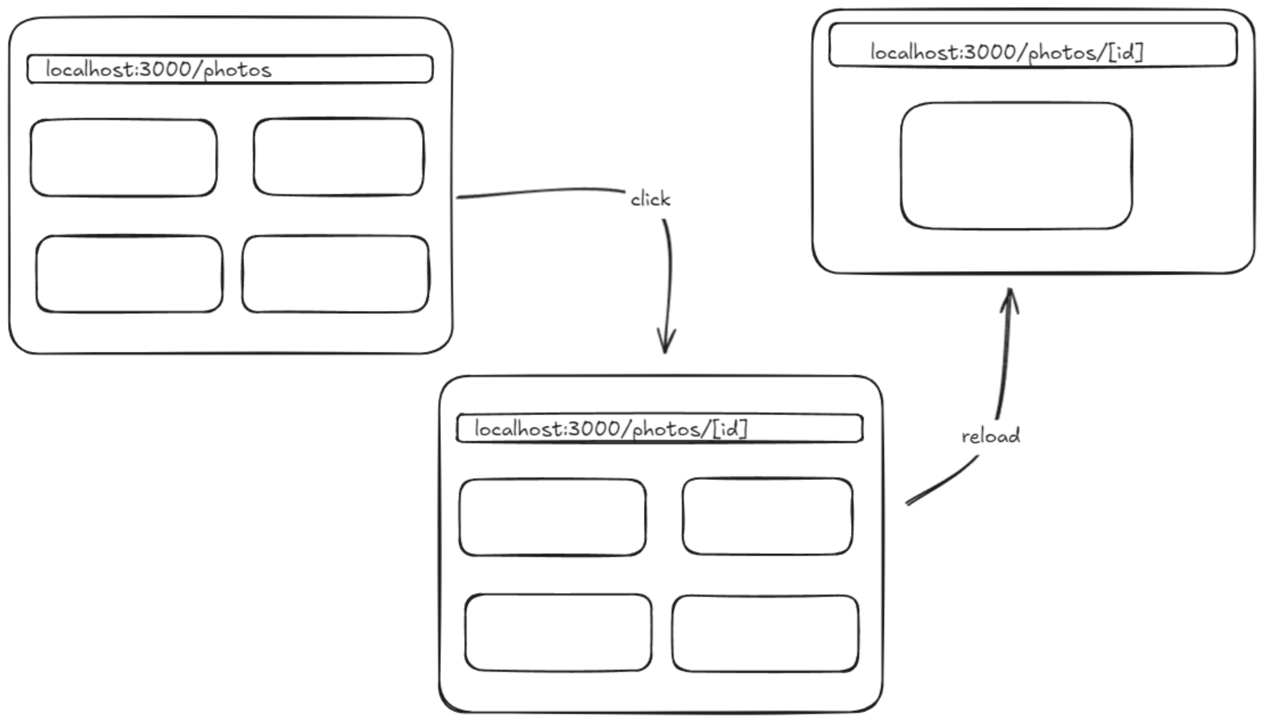
```
export default function ComplexDashboardLayout({
children,
users,
revenue,
notifications
}:{
children:React.ReactNode;
users:React.ReactNode;
revenue:React.ReactNode;
notifications:React.ReactNode;
}){
return (
    <>
        <div>{children}</div>
        <div>{users}</div>
        <div>{revenue}</div>
        <div>{notifications}</div>
    </>
)

}
```

```
                                        unmatched routes
```

now if i go to the complex dashboard route, at the time of reloading ,it shows some errors.
so to prevent this we have a default.tsx page.
add it to each slot and with root page.tsx
because of this when the page reloads it will show the default view.

```
                                    intercepting routes
```



if we want to go from routes a to b, and if there is a middle intercepting page, that is intercepting route, and if the page is reloaded , it goes back to org page.

conventions
(.) to match segments on the same level
(..) to match segments one level above
(..)(..) to match segments two level above
(...) to match segments from the root app dir

folder structure

intercepting f1 --> f2 (source to dest)
-> src/app/f1/page.tsx
-> src/app/f1/f2/page.tsx

create a folder to the same level of dest
folder name -> (.f2) -> one level up

intercepting f1 --> f3
-> src/app/f1/page.tsx
-> src/app/f3/page.tsx
creating folder under f1 dir, gap is now 2 level up

so folder name -> (..f3)

intercepting f2 --> f4
-> src/app/f1/f2/page.tsx
-> src/app/f4/page.tsx

create folder under f2, gap is now 3
-> (..)(..) f4

```
                                        Route Handlers
```

route handlers -> api routes
next supports get,post,patch,put,delete,head & options
if an unsupported method is called Nextjs will return a 405 method not allowed response.

```
                                 Headers in route handlers
```

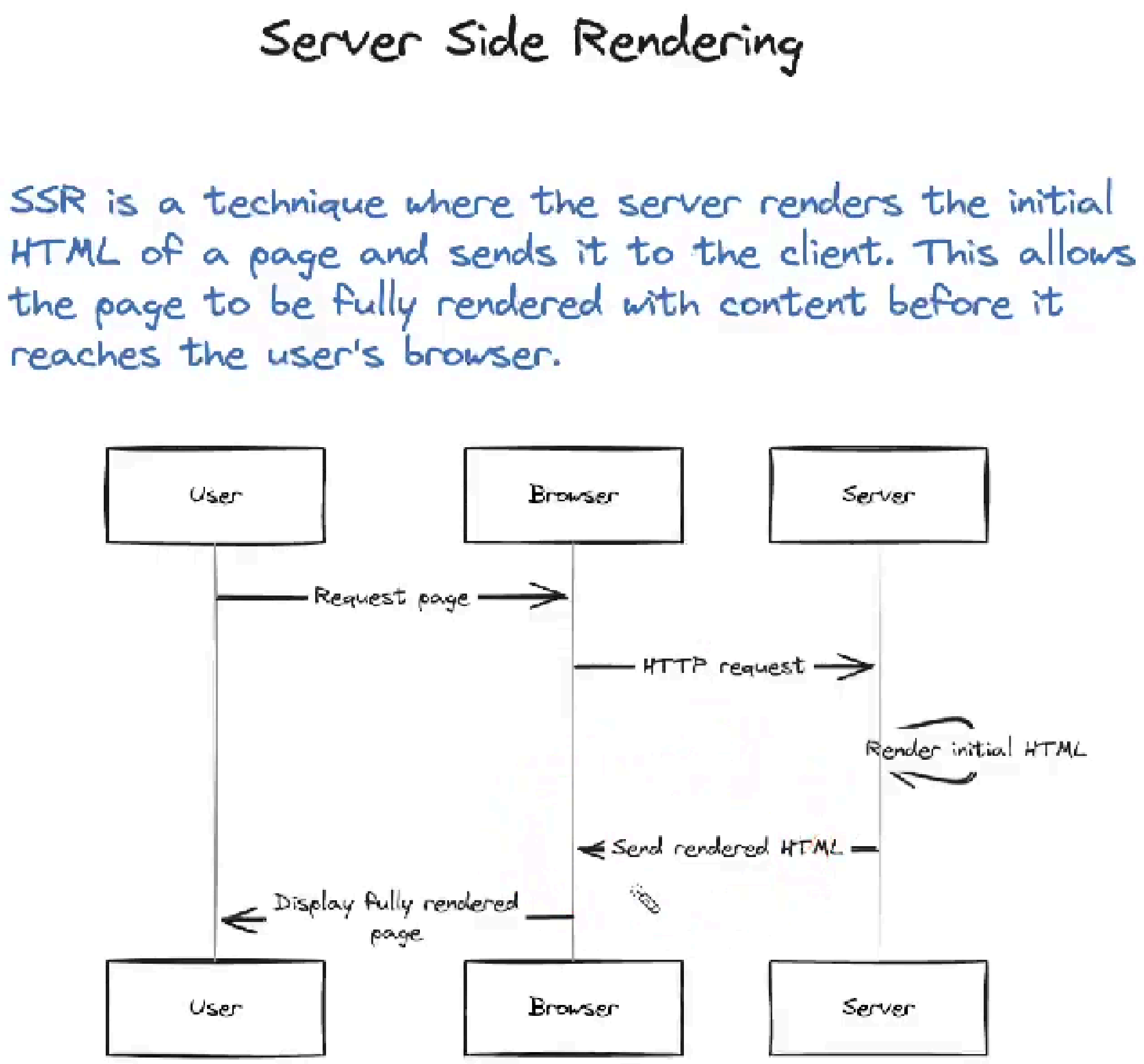
Http headers represent the metadata associated with an API request and response.

Request Headers
These are sent by the client,such as a web browser to the server.They contain essential information about the request,which helps the server understand and process it correctly.
"User-Agent" which identifies the browser and os to the server.

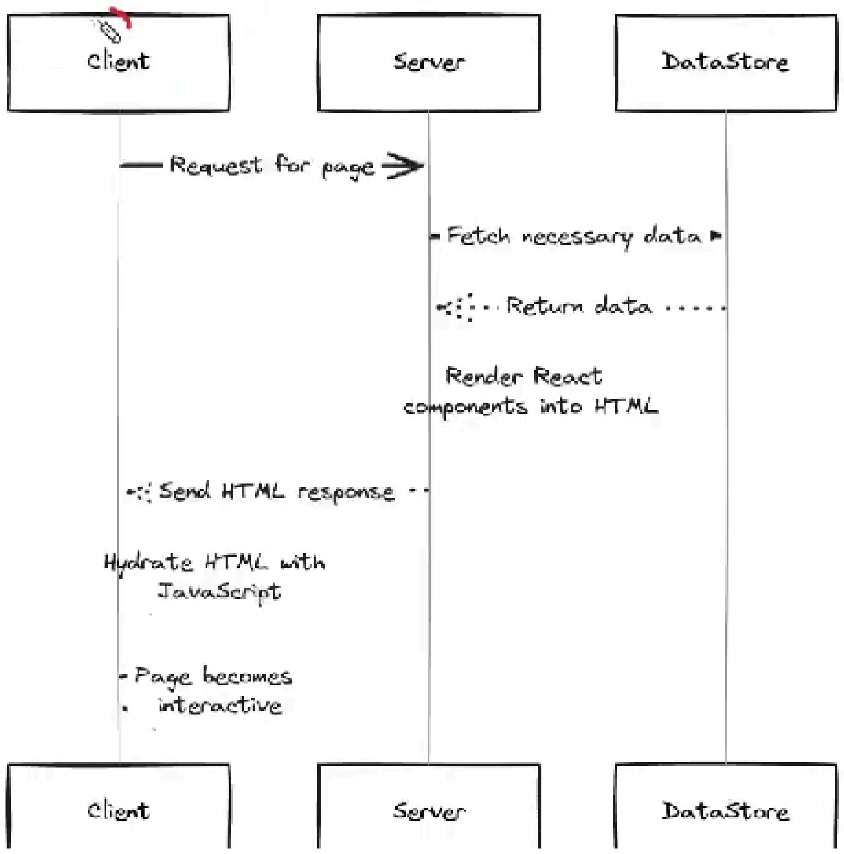
"Accept" which indicates the content types like text,video or image formats that the client can process.
"Authorization" header used by the client to authenticate itself to the server.

Response Headers
These are sent back from the server to the client. They provide information about the serveer and the data being sent in the response.
'Content-Type' header which indicates the media type of the response.It tells the client what the data type of the returned content is,such as text/html for HTML doc,application/json for JSON data,etc.

```
                    Rendering
```


Server Side Rendering
SSR is a technique where the server renders the initial HTML of a page and sends it to the client. This allows the page to be fully rendered with content before it reaches the user's browser.
User
Browser
Server
Request page
HTTP request
Render initial HTML
Send rendered HTML
Display fully rendered page
User
Browser
Server

When a request is made to the server, the server fetches the necessary data, renders the React components into HTML, and sends this HTML to the client. The client then hydrates this HTML with JavaScript, making it interactive.



Hydration

Hydration is a process of taking your server side rendered HTML & making it interactive by attaching necessary JS event handlers and states to the existing HTML elements.

- SSR : The server generates the HTML for a page and send it back to browser. This HTML has all the relevant data and markup.

$\downarrow$

- Intial page load: the downloaded HTML is rendered immediately.

$\downarrow$

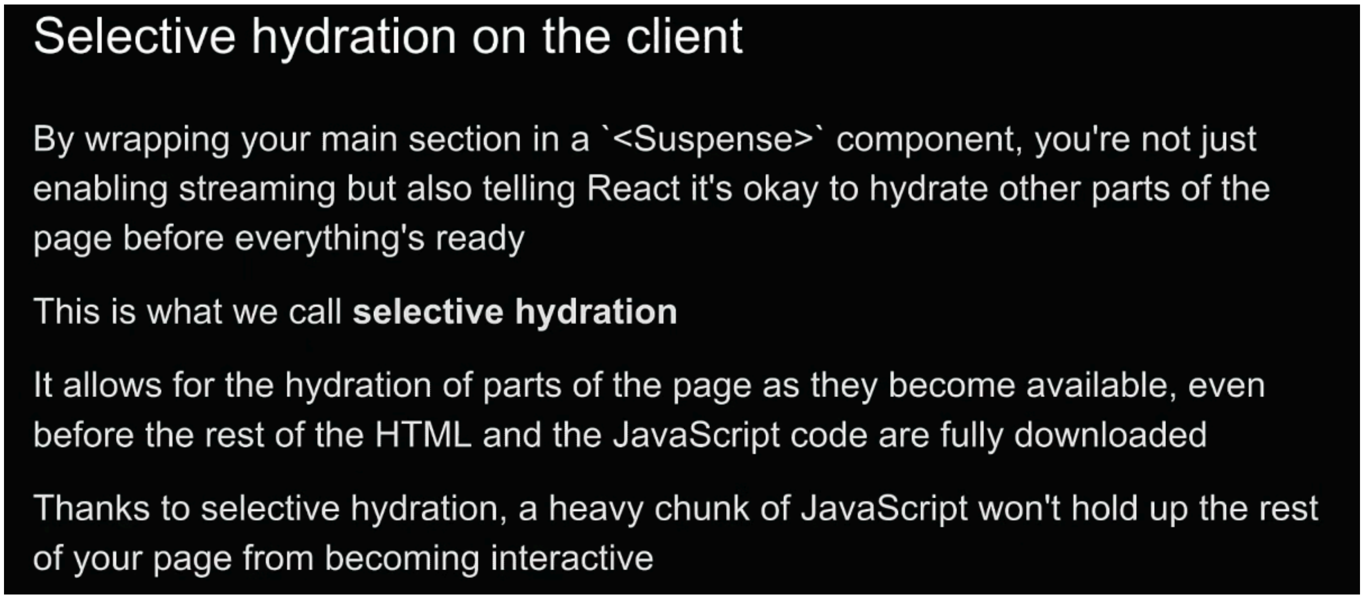
- Hydration: React client side JS is loaded on the client side, which then attahces states or event listeners., post this the page becomes interactive.

React Server Components

Definition: RSC is a new feature in React that allows developers to build components that run on the server. These components can fetch data and render to HTML on the server, but they don't include client-side JavaScript, resulting in smaller bundles and better performance.

How it Works: RSC allows for splitting the application into server components and client components. Server components are rendered on the server and sent to the client as static HTML, while client components are hydrated with JavaScript on the client side.

Benefits:
Performance: RSC can significantly reduce the amount of JavaScript sent to the client, leading to faster load times and smaller bundles.
Simplified Data Fetching: Since server components run on the server, they can directly fetch data and render HTML without requiring complex client-side data fetching logic.



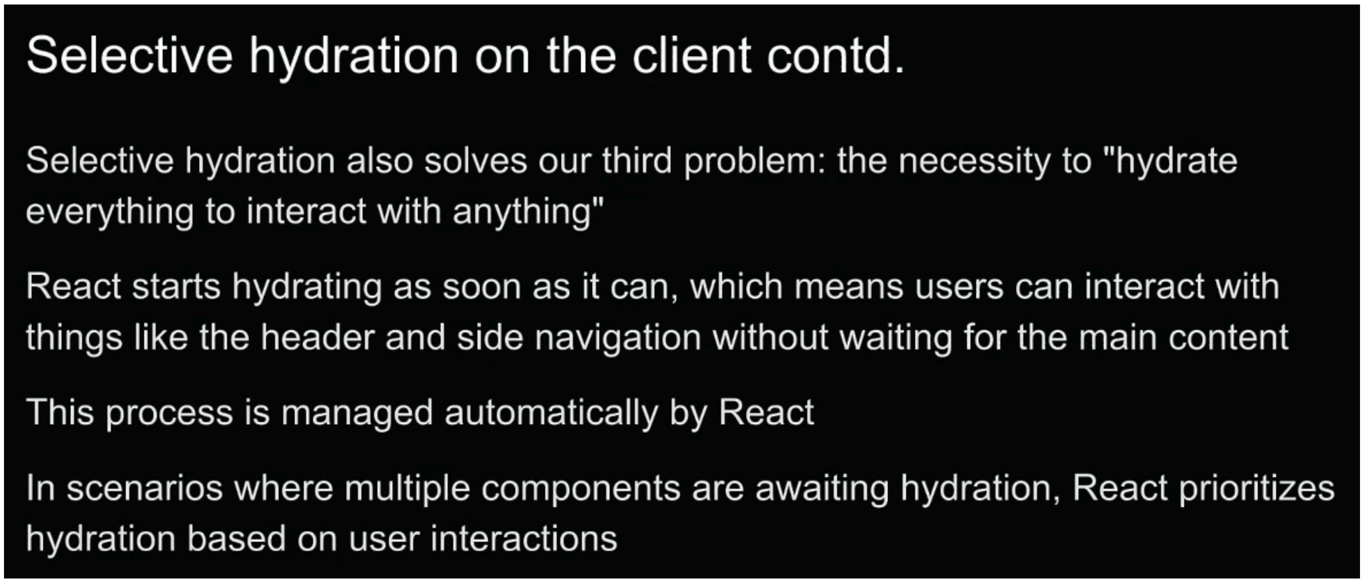
Selective hydration on the client

By wrapping your main section in a `<Suspense>` component, you're not just enabling streaming but also telling React it's okay to hydrate other parts of the page before everything's ready

This is what we call **selective hydration**

It allows for the hydration of parts of the page as they become available, even before the rest of the HTML and the JavaScript code are fully downloaded

Thanks to selective hydration, a heavy chunk of JavaScript won't hold up the rest of your page from becoming interactive



Selective hydration on the client contd.

Selective hydration also solves our third problem: the necessity to "hydrate everything to interact with anything"

React starts hydrating as soon as it can, which means users can interact with things like the header and side navigation without waiting for the main content

This process is managed automatically by React

In scenarios where multiple components are awaiting hydration, React prioritizes hydration based on user interactions

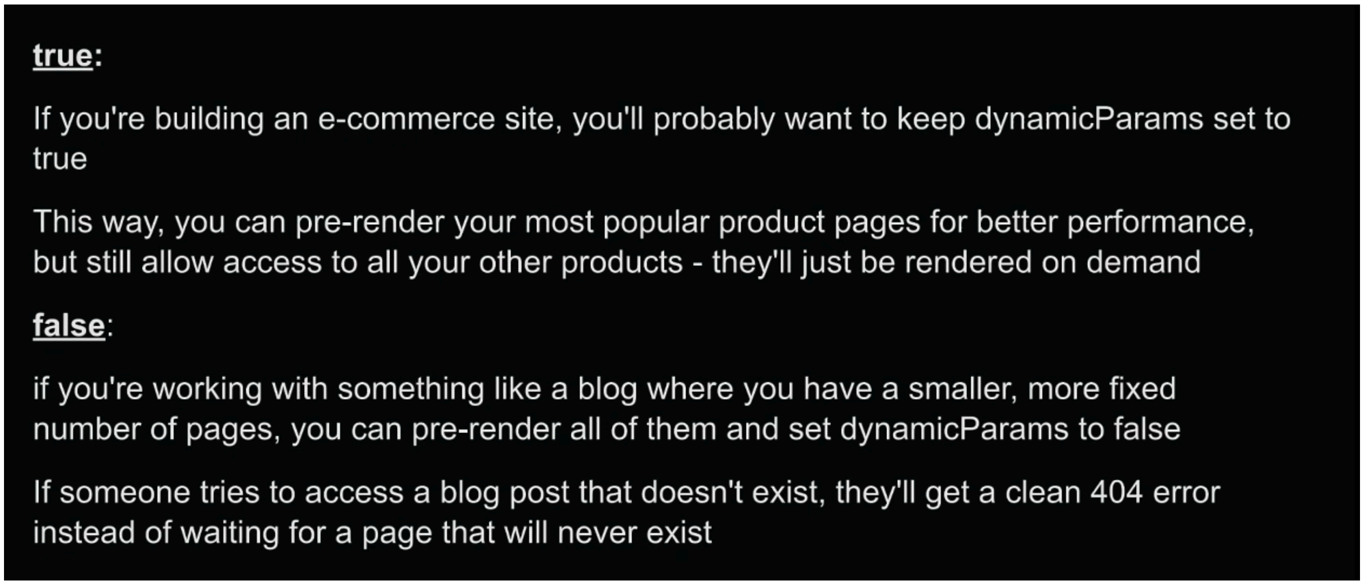
```
generateStaticParams
```

```
export const dynamicParams=true;

export async function generateStaticParams(){
    return [{id:'1'},{id:'2'},{id:'3'}];
}

export default async function ProductPage({
    params
}:{
    params:Promise<{id:string}>;
}){
    const {id} = await params;
    return (
        <h1>
            Product {id} details rendered at {new Date().LocaleTimeString()}
        </h1>

    )
}
```



true:

If you're building an e-commerce site, you'll probably want to keep dynamicParams set to true

This way, you can pre-render your most popular product pages for better performance, but still allow access to all your other products - they'll just be rendered on demand

false:

if you're working with something like a blog where you have a smaller, more fixed number of pages, you can pre-render all of them and set dynamicParams to false

If someone tries to access a blog post that doesn't exist, they'll get a clean 404 error instead of waiting for a page that will never exist

```
        Server and Client Composition Pattern
```

Server Components

1. fetching data
2. accessing backend resources directly
3. keeping sensitive info(access tokens and api keys) secure on the server.
4. handling large dependencies server-side --which means less js for users to download

Client Components

1. adding interactivity
2. handling event listeners(onClick,OnChange)
3. managing state(usestate,useEffect)
4. working with browser specific apis
5. implementing custom hooks
6. using react class components

```
export const serverSideFunction = ()=>{
    console.log(
    `use multiple libraries,
    interact with a database
    `
    );
    return 'server result';

}
```

if this server side function gets exposed to a client component in the browser , then it is a security issue , so to this function accessible to server components only, use ->

```
npm i server-only
```

```
import 'server-only'
export const serverSideFunction = ()=>{
    console.log(
    `use multiple libraries,
    interact with a database
    `
    )
    return 'server result';

}
```

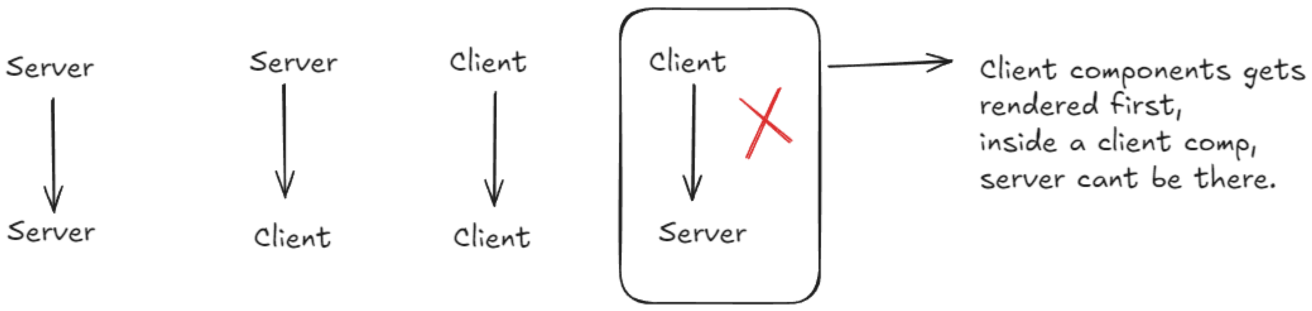
if this function used in client component , then at the build time this will throw error.

Client only code works with browser-specific features - DOM manipulations,window object interactions, or localStorage operations.

```
npm i client-only
```

```
import 'client-only'
export const serverSideFunction = ()=>{
    console.log(
    `use window object,
    use localStorage
    `
    )
    return 'client result';

}
```



Client components gets rendered first,
inside a client comp,
server cant be there.