# WEB3, Session 11

Server-Side Rendering

# The more things change … (1999)

- I can see a page, but when I interact…
- *First* send data to the server
- *Then* the server fetches data
- *Then* the server creates HTML
- *Then* the HTML is send over my dial-up connection
- *Time spent on an interaction* is too long

# … the more they stay the same (2024)

- I'm connected to the server, but …
- *First* the scripts are loaded
- *Then* the scripts are compiled
- *Then* the scripts are run
- *Then* the data is fetched
- *Then* the page is rendered
- *Time to interactive* is too long

# To SPA and back

- The reasons for single-page applications:
  - Immediate user interactions
  - Distributing the generation of HTML (or HTML DOM)
  - Lower bandwidth - the data is smaller than the HTML generated from it
- The reasons against single-page applications:
  - Difference in computation power of clients
  - Higher bandwidth once you factor in all the scripts
  - Delayed page loading (SEO)

# Where does that leave us?

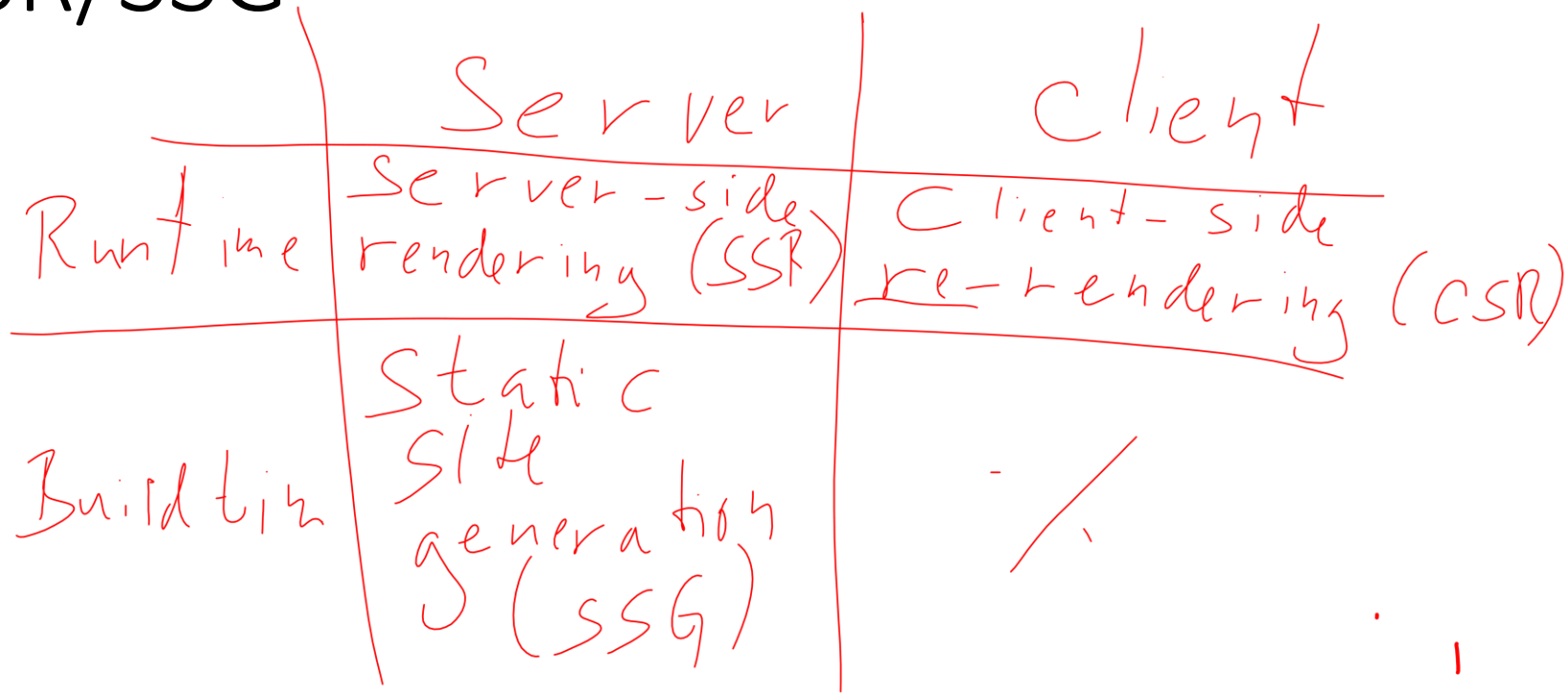
- Single-page applications
  - Actual *applications*
  - When most things are about user interactions
- Server-side rendering
  - When dynamic content is shown
  - User interaction is sparse
  - Can still allow for user interaction after rendering
- Static side generation
  - When (mostly) static generation is shown
  - Can still allow for user interaction after rendering

# Rendering and user interaction

- In this context, *rendering* refers to the generation of HTML or HTML DOM
  - Not what the browser does to display the DOM
- It can happen on the server or client side
- It can happen on build time or run time
- Note, we are not turning of JavaScript
  - Re-rendering in the browser can (usually will) still happen
  - User interaction can (usually will) still happen

# Rendering in SSR/SSG

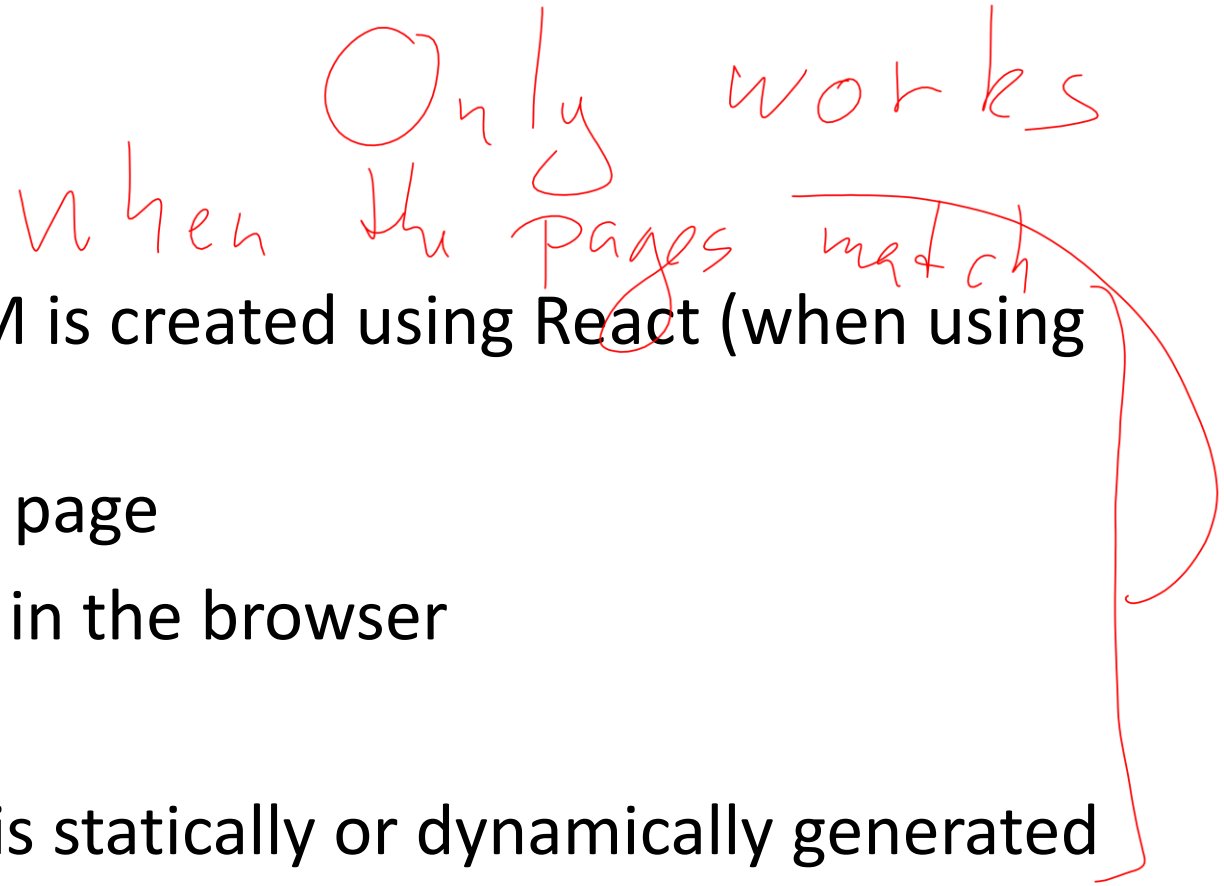
- When?
  - Build time
  - Run time
- Where?
  - Server
  - Client (browser)



# Page generation and data fetching

- Static site generation
  - Data is fetched at build time
- Server-side rendering
  - Data is fetched when the page is requested
- Client-side (re-)rendering
  - Data is fetched when user interacts
  - Can also fetch data when page is mounted

# Hydration




- In the browser, the Virtual DOM is created using React (when using Next.js)
- This is then used to update the page
- Your page is now fully dynamic in the browser
- It is *hydrated*
- This works whether your page is statically or dynamically generated

# Next.js

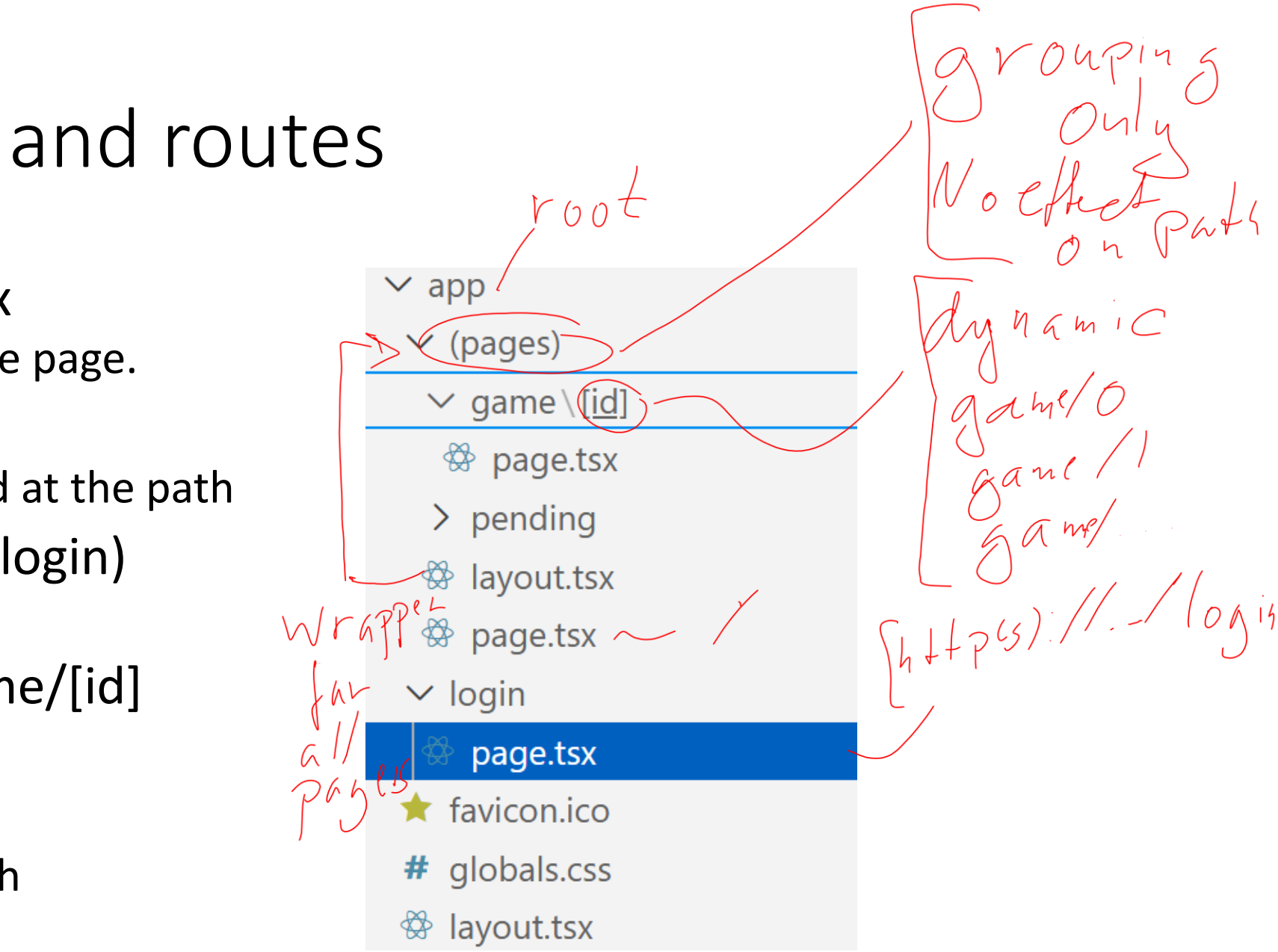
- Probably the most used library for SSR
  - (But these things change all the time)
- Build on top of React
- Uses React's server-side components
- Allows both SSR and SSG

# Next.js scripts

- npx create-next-app@latest
  - Creates new next project
- npm run build
  - Builds the project
- npm start
  - Runs the project
  - Has to be built
- npm run dev
  - Starts project in developer mode — hot deploy
  - Doesn't have to be built
  - No SSG - only SSR

# Next.js pages and routes

- layout.tsx/layout.jsx
  - Wrapers around the page.
- page.tsx/page.jsx
  - The page you'll find at the path
- Static path: / (and /login)
  - Default: SSG
- Dynamic path: /game/[id]
  - Default: SSR
- Grouping: (pages)
  - Not part of the path



# Layout

layout.tsx

```
export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <Page>{children}</Page>
  )
}
```

# Page

page.tsx

Server Component

game/[id]

```
import Game from "@/components/Game"

export default async ({params}: {
    params: Promise<{id: string}>
}) => {
  const {id} = await params

  return <Game game_id={id}></Game>
}
```

# Server and client component

- Server components are rendered only on the server
  - You **can** still manipulate the DOM in pure JavaScript, but not React
  - You cannot use any hooks (use…)
  - You can use async/await
  - Default, but can also be marked with 'use server' in the first line
- Client components are rendered on both server and client
  - You can use all hooks
  - You **cannot** use async/await (except in local functions - like useEffect)
  - Marked with 'use client' in the first line
  - **Does not trigger dynamic rendering**

# A Client Component (excerpts)

```
'use client'

export default function Login({proceedTo}: {proceedTo: string}) {
  const [player, setPlayer] = useState('')

  function login() {…}
  return <>
    Username: <input value={player}
                     onChange={e => setPlayer(e.target.value)}/>
    <button disabled={!enabled} onClick={login} >Login</button>
  </>
}
```

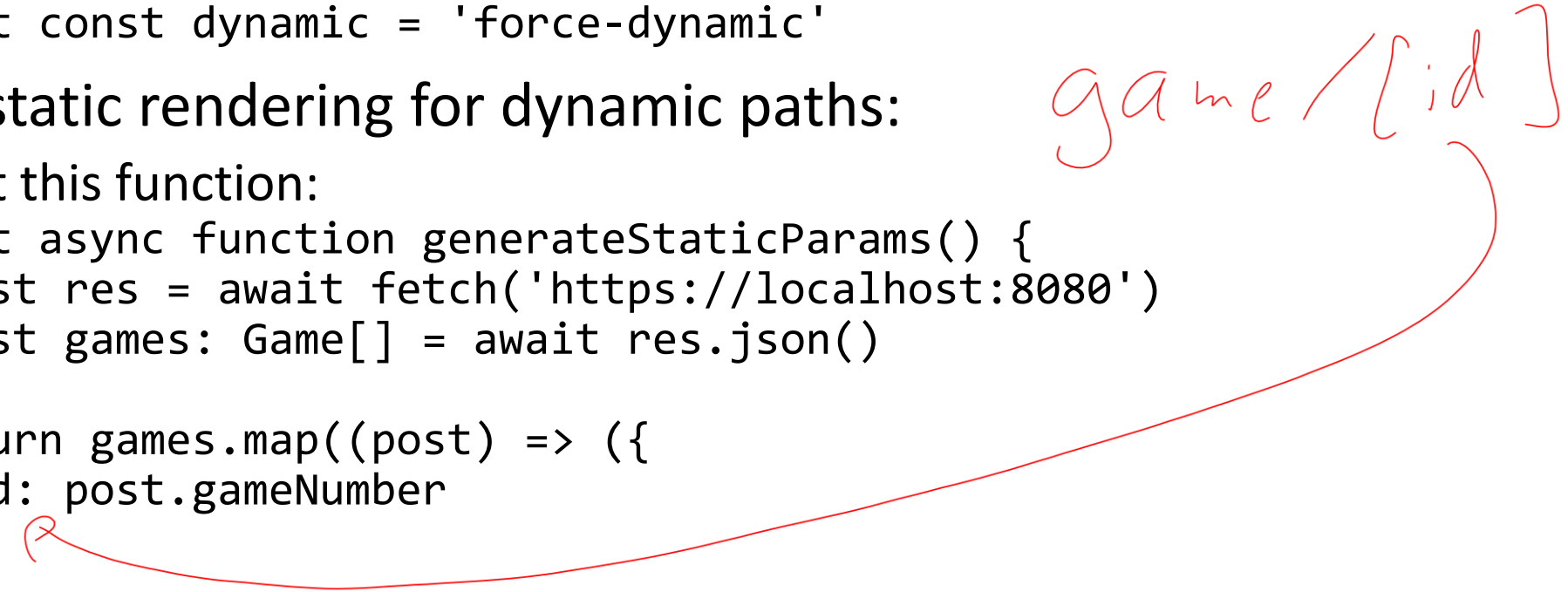
# Dynamic and static rendering

- Dynamic rendering (SSR)
  - Takes place at runtime
  - Generate the page with fresh data
  - Triggered by
    - A dynamic route: /game/[id]
    - `Using run-time APIs: cookies, headers, searchParams, params`
- Static rendering (SSG)
  - Take place at build-time
  - Generate data-independent pages (or with stale data)
    - But can be re-validated
  - Triggered by static route and no run-time APIs

# Forcing dynamic/static rendering

- Forcing dynamic rendering (for non-dynamic paths):
  - Somewhere in the file:
    ```
    export const dynamic = 'force-dynamic'
    ```
- Forcing static rendering for dynamic paths:
  - Export this function:
    ```
    export async function generateStaticParams() {
      const res = await fetch('https://localhost:8080')
      const games: Game[] = await res.json()

      return games.map((post) => ({
        id: post.gameNumber
      }))
    }
    ```



# Using Web Sockets

- All web socket code *has* to be in client components
- The best way is to create the web socket connection in useEffect
- Close the connection in the clean-up function
- **Important:** If you update a state (from useState) in the web socket, you *have* to use the functional version of the setter (see code later).

# useEffect

- A React hook - is given a function
- Called the first time the component is rendered
- Called again based on an array of values:
  - If no array is given (undefined): Called on every re-render
  - If an empty array is given: Only called the first time
  - If a non-empty array is given: Called when any of the values have changed
- Clean-up function: Returned from the useEffect function

# Web socket in Next.js client component

const [game, setGame] = useState(...)

```
useEffect(() => {
  const ws = new WebSocket('ws://localhost:9090/publish')
  ws.onopen = () => {
    ws.send(JSON.stringify({type: 'subscribe', key: 'move_' + game.gameNumber}))
    ws.onmessage = ({data}) => {
      const {message}: {message: MoveMessage} = JSON.parse(data)
      setGame((game: Game) => applyMoveMessage(message, game))
    }
  }
  return () => { if (ws.readyState === WebSocket.OPEN) ws.close() }
}, [game.gameNumber])
```

L functional

# Advice and pitfalls

# Be aware of both server and client

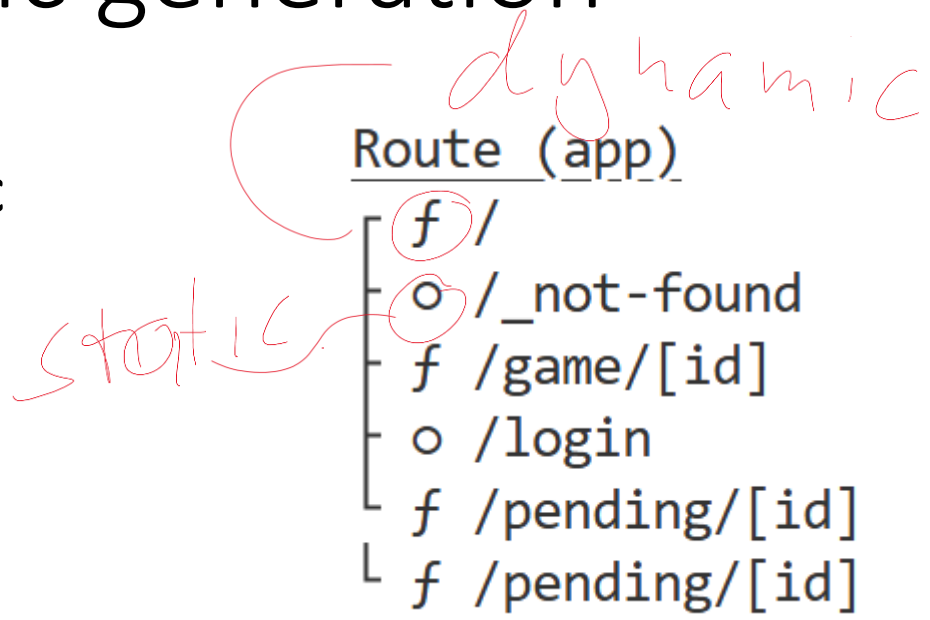
- Which data is available on the server?
  - How is it used to render the page?
- Which data is available on the client?
  - Especially: Does the client and server agree?
- Remember that a client component is rendered on the server first
- Disagreement between server and client may result in
  - At best: Hydration errors, triggering a re-render
  - At worst: Infinite re-renders

# Use cookies to share data between server and client

- For login tokens, session ids and similar
- On a dynamic page:
  - Use the next.js cookies() API
- On a static page:
  - Use whatever you would use in JavaScript
  - `js-cookie` is popular

# Be aware of static vs dynamic generation

- Plan for which pages are static or dynamic
- Use `npm run build` to check
- Force dynamic if you need to
- Failure might have you server stale data
- Pitfall:
  - `npm run dev` is always dynamic
  - You need to use `npm run build` + `npm start` to test



# State management

- Global management is incredibly hard
  - Since this runs on both server and all clients (see this)
- My approach:
  - In a server component:
    - Get initial data
    - Provide the data to the client component wrapper
  - In the client component: (React)
    - Create a state using one or more uses of setState()
    - Create a context with both the state values and the necessary setter(s)
    - Use useContext to get the state
    - Use and set the state as needed

# Conclusion

- Use Next.js for SSR and/or SSG
  - (Alternatives: Nuxt.js, Quasar, ...)
- Be aware of which parts are dynamic and which parts are static
- Test in *both* developer and production mode