

1. React Server Component

클라이언트 사이드 렌더링

- HTML을 다운로드한다
- 앱을 실행하는데 필요한 모든 것이 포함된 자바스크립트 파일(bundle)을 다운로드한다
- 리액트가 작동해서 DOM을 불러온다
- 비어있는 `<div id="root"></div>` 에 렌더링한다.

```
<!doctype html>
<html>
  <body>
    <div id="root"></div>
    <script src="/static/js/bundle.js"></script>
  </body>
</html>
```

1. React Server Component

클라이언트 사이드 렌더링 단점

- 작업 수행시간이 걸린다.
- 작업 진행 중 사용자는 흰 화면을 본다.
- 자바스크립트 버들 크기가 증가하면 작업 시간이 비례해서 오래 걸린다.

1. React Server Component

서버 사이드 렌더링

- 서버가 첫 번째 렌더링을 수행한다.
- 비어있는 흰색 페이지 대신 기본 정적 레이아웃 정도는 보인다.
- But. 사용자가 원하는건 내용이다.

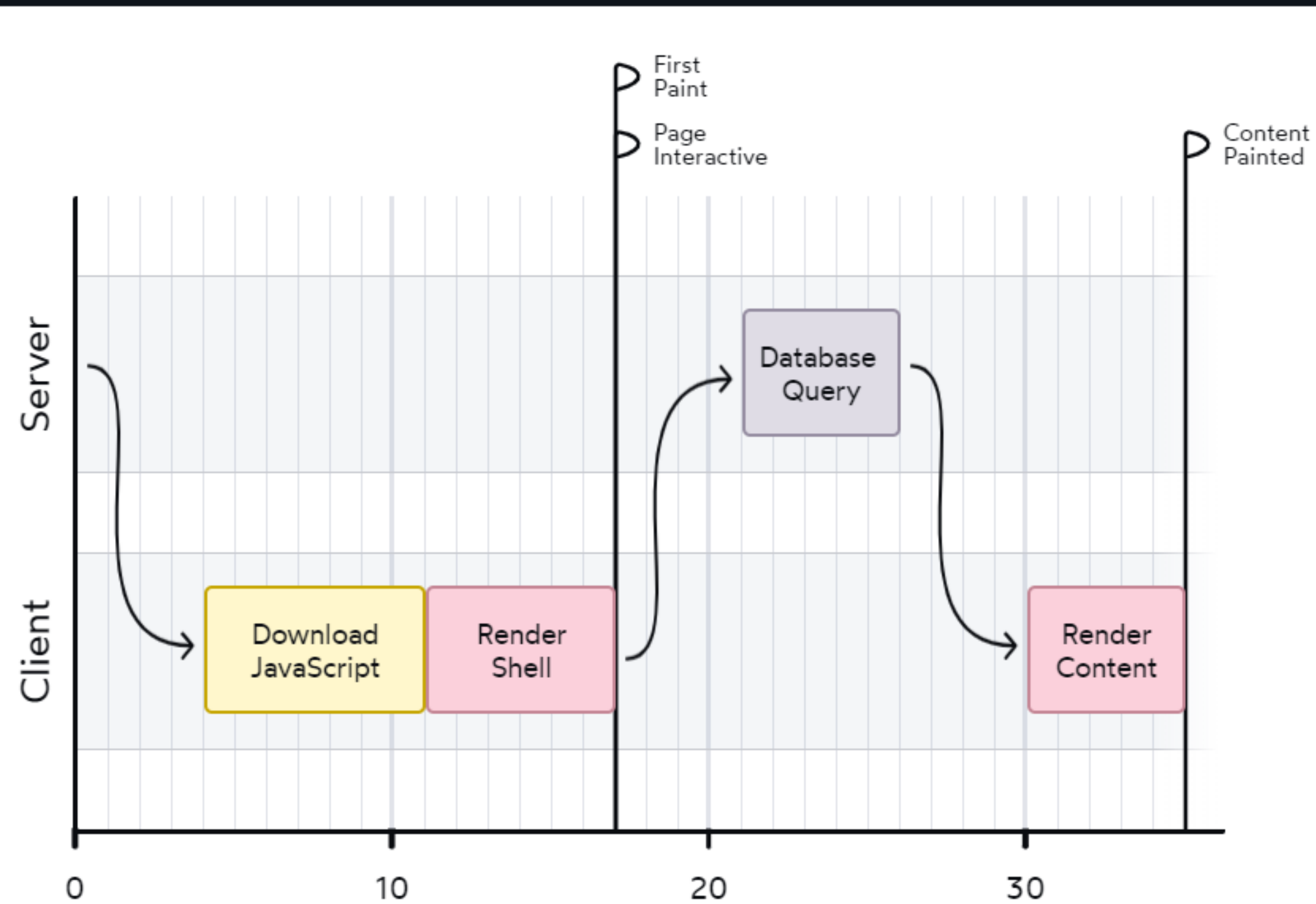
1. React Server Component

클라이언트 사이드 렌더링 vs 서버 사이드 렌더링

Client Side Rendering

CSR

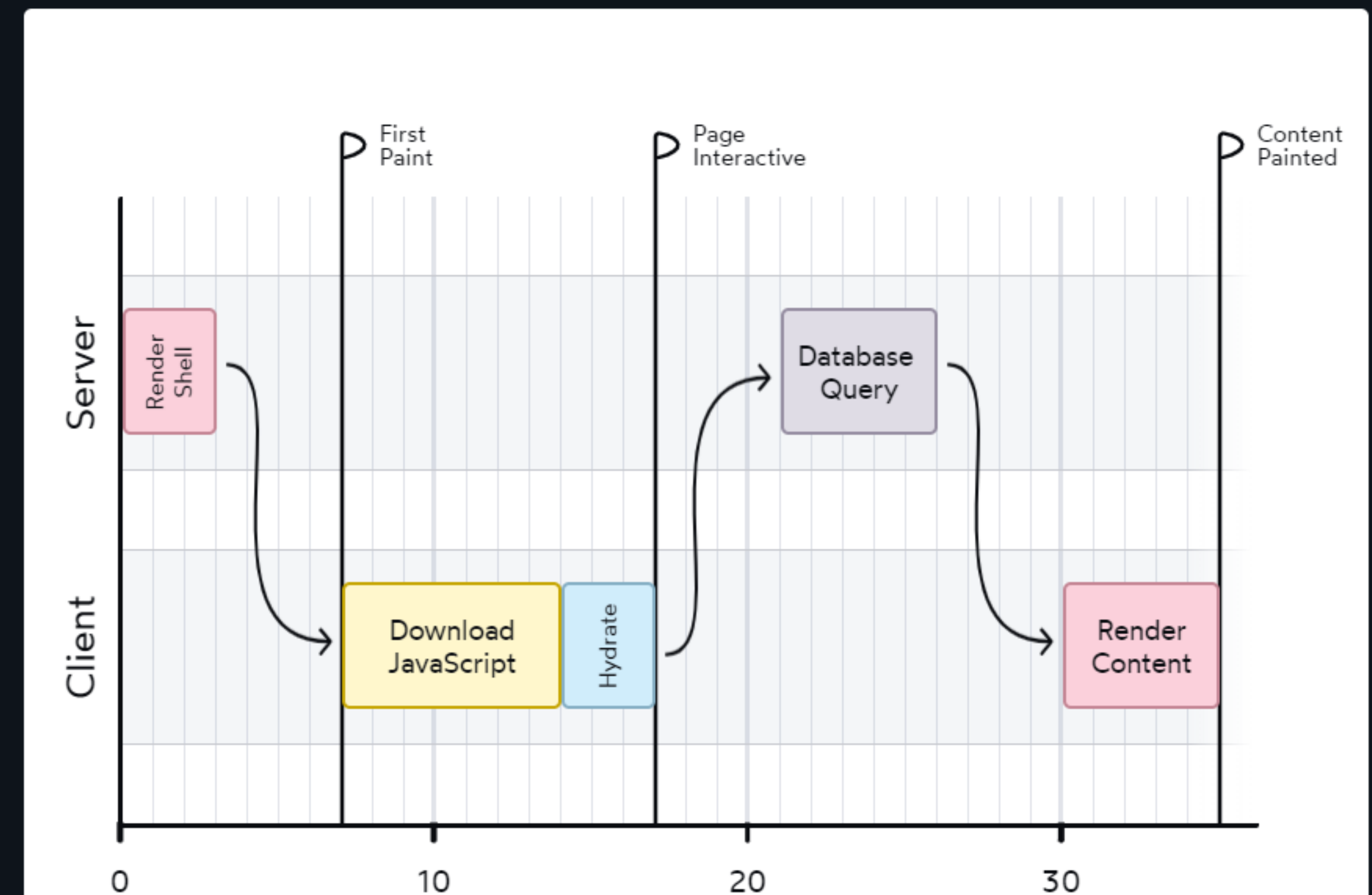
SSR



Server Side Rendering

CSR

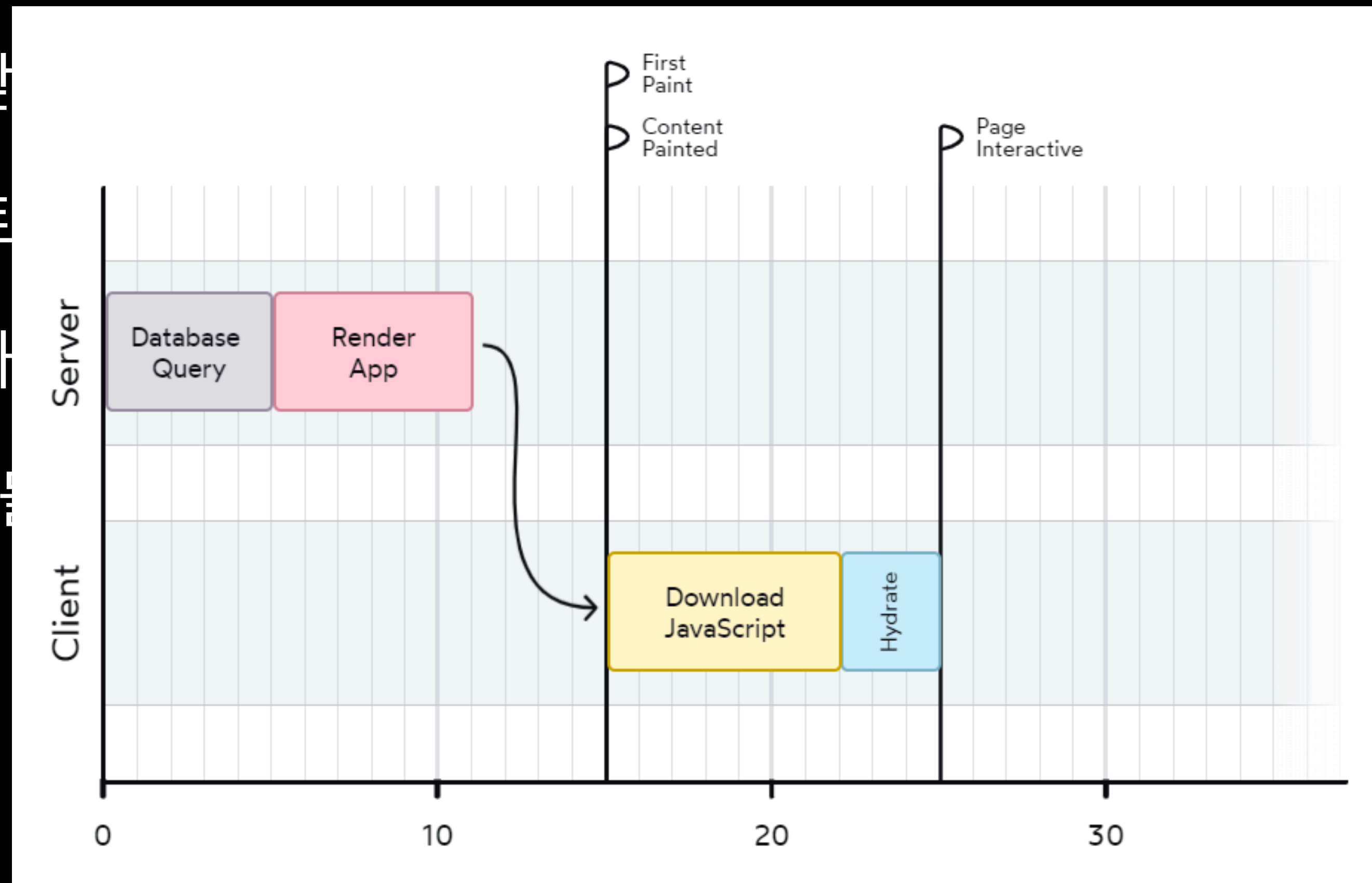
SSR



1. React Server Component

서버 사이드 렌더링 (Next.js)

- 서버가 요청을 받
- 그 후 클라이언트
- 서버 로직이 클라
- 자바스크립트 번



1. React Server Component

서버 사이드 렌더링 (Next.js) 단점

- 최상위 컴포넌트에서만 작동하기 때문에 각각 컴포넌트에 적용할 수 없다.
- 리액트 라이브러리가 아닌 프레임워크의 기능이다.
- 클라이언트에서 하이드레이션할 필요가 없어도 모든 컴포넌트가 하이드레이션 되어야 한다.

1. React Server Component

리액트 서버 컴포넌트란?

- 서버에서만 실행되는 컴포넌트
- 함수 컴포넌트 안에 직접적으로 `async` 키워드 사용 불가능
- 렌더 함수 내에서 Side Effect를 사용 중이다.

```
const ScreeningLayout = async ({ children, title }: Props) : Promise<Element> => {
  const nextCookies : ReadonlyRequestCookies = cookies();
  const queryClient : QueryClient = getQueryClient();

  const cookieScreening : string | undefined = nextCookies.get(COOKIE_KEY.SCREENING)?.value;
  if (cookieScreening) {
    const screening : Screening = JSON.parse(decodeURI(atob(cookieScreening))) as Screening;
    await queryClient.prefetchQuery(FETCH_SCREENING(), () => screening);
  }

  const attendeeRole: ATTENDEE_ROLE = getAttendeeRoleFromCookies(nextCookies);
  await queryClient.prefetchQuery(FETCH_ATTENDEE_ROLE(), () => attendeeRole);

  const dehydratedState : DehydratedState = dehydrate(queryClient);

  return (
    <Hydrate state={dehydratedState}>
      {title}
      {children}
    </Hydrate>
  );
};
```

1. React Server Component

리액트의 클라이언트 컴포넌트와 서버 컴포넌트

- 기존의 컴포넌트는 클라이언트 컴포넌트로 리브랜딩 된 것이다.
- 서버 컴포넌트라는 패러다임이 추가 됐다.
- 클라이언트 컴포넌트는 서버, 클라이언트 양쪽에서 렌더링 된다.
- 서버 컴포넌트는 서버에서만 렌더링 된다.

1. React Server Component

서버 컴포넌트 특징

- 서버 컴포넌트는 다시 렌더링하지 않아도 된다.
- 서버에서 한번 렌더링 된 값은 클라이언트로 전송되어 제자리에 고정 된다.
- 서버 컴포넌트는 JS 번들에 포함되는 것이 아닌 서버에서 문자열로 내려와 렌더링 된다.
- 하이드레이션 되거나 리렌더링 되지 않는다.

1. React Server Component

RSC와 SSR

- 리액트 서버 컴포넌트는 서버 사이드 렌더링을 대체하는 기술이 아니다.
- 서버 사이드 렌더링을 지원하기 위해 나온 새로운 컴포넌트 패러다임이다.
- 현재 리액트 서버 컴포넌트를 공식 지원하는 프로젝트는 Next.js 13.4 이상에서 새롭게 재설계된 App Router를 사용하는 방법 밖에 없다.

1. React Server Component

Client Side Component in App dir

- 리액트 서버 컴포넌트 패러다임에서는 모든 컴포넌트가 서버 컴포넌트라고 가정한다.
- 클라이언트 컴포넌트를 사용하기 위해서는 지시문을 사용해야 한다.
- “use server” 지시문은 서버 액션에 사용된다.

```
'use client';

import ...

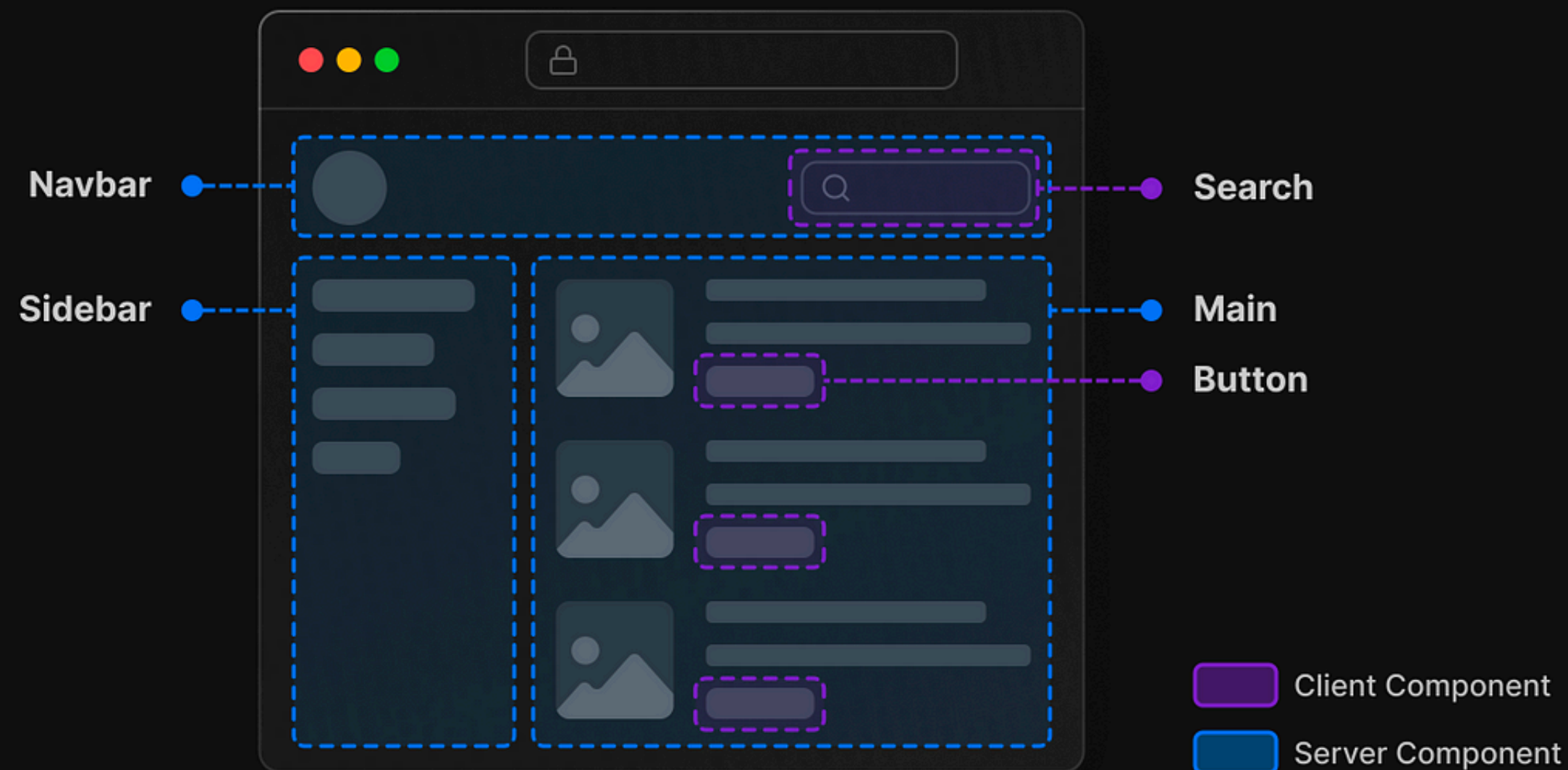
interface Props { ... }

const ReverseCounterTemplate = ({ targetDay, screeningId }: Props) => { ... };
💡
export default ReverseCounterTemplate;
```

1. React Server Component

서버 컴포넌트와 클라이언트 컴포넌트를 나누는 기준

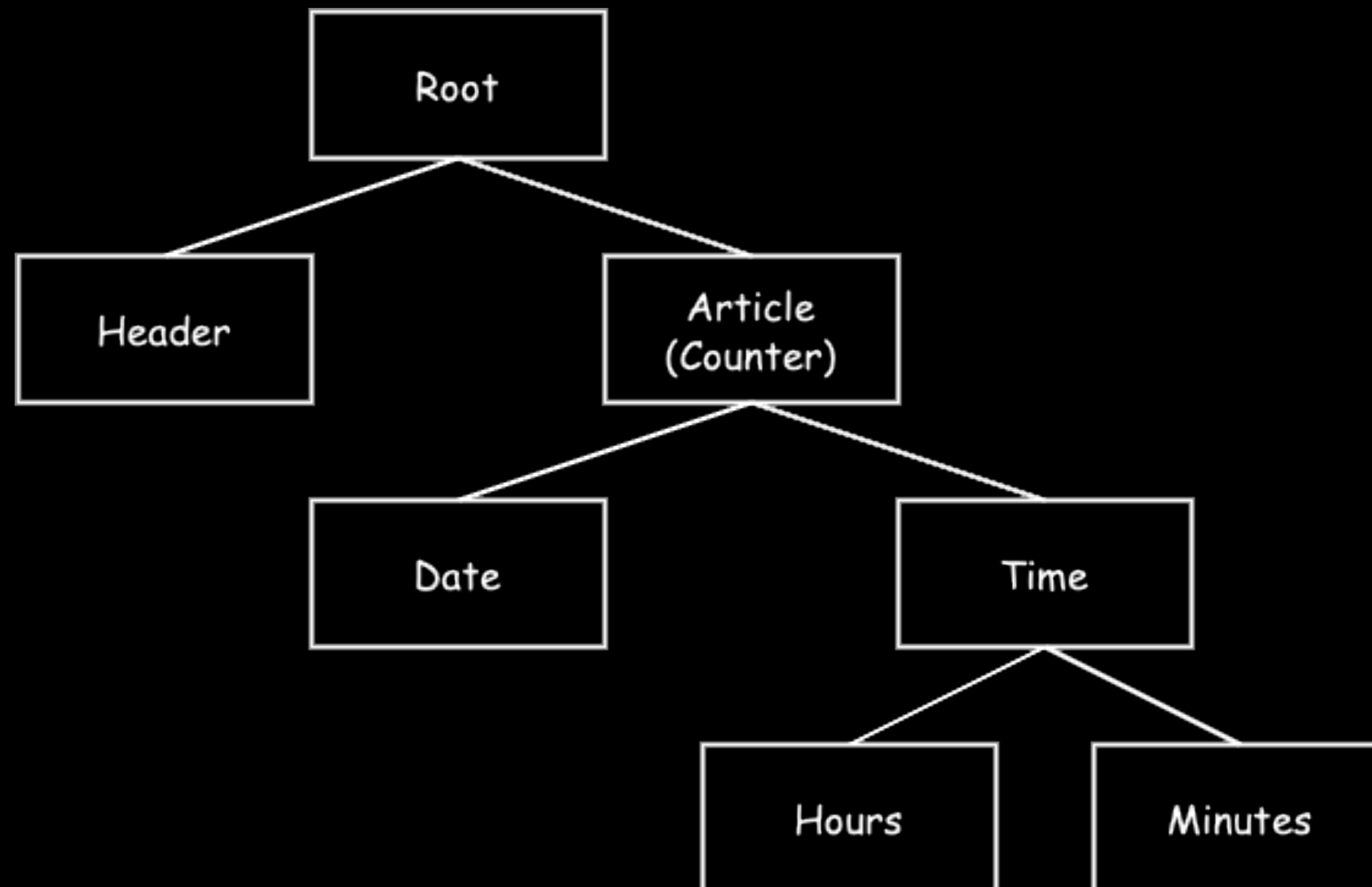
- 서버 컴포넌트로 만들 수 있다면 그렇게 하는 것이 좋다.
- 사용자 인터렉션이 있냐? 없냐? 로 나누면 좋다(경험).



1. React Server Component

컴포넌트 경계

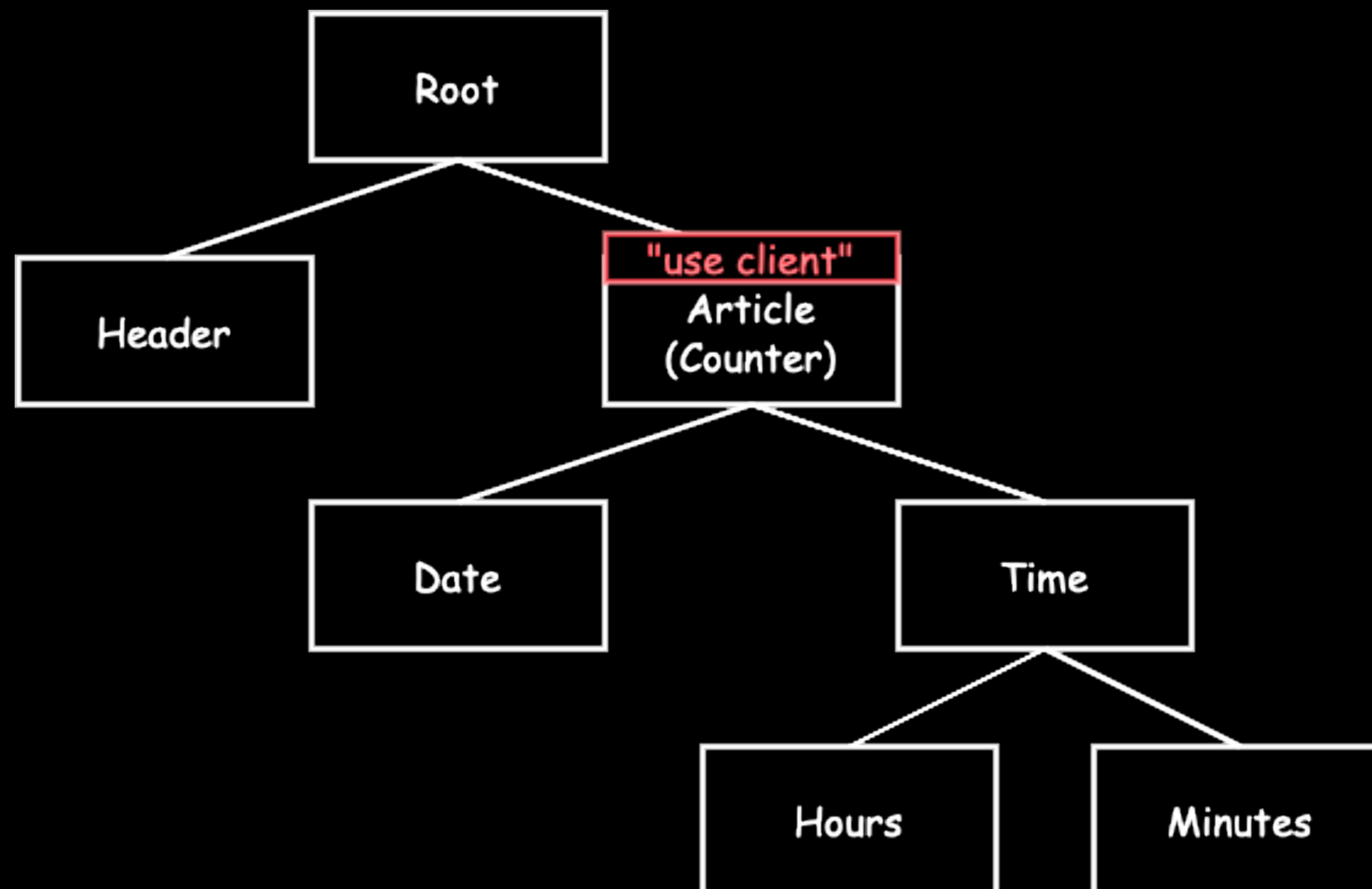
- 서버 컴포넌트의 속성(props)가 변해도 리렌더링 되지 않는다.



1. React Server Component

컴포넌트 경계

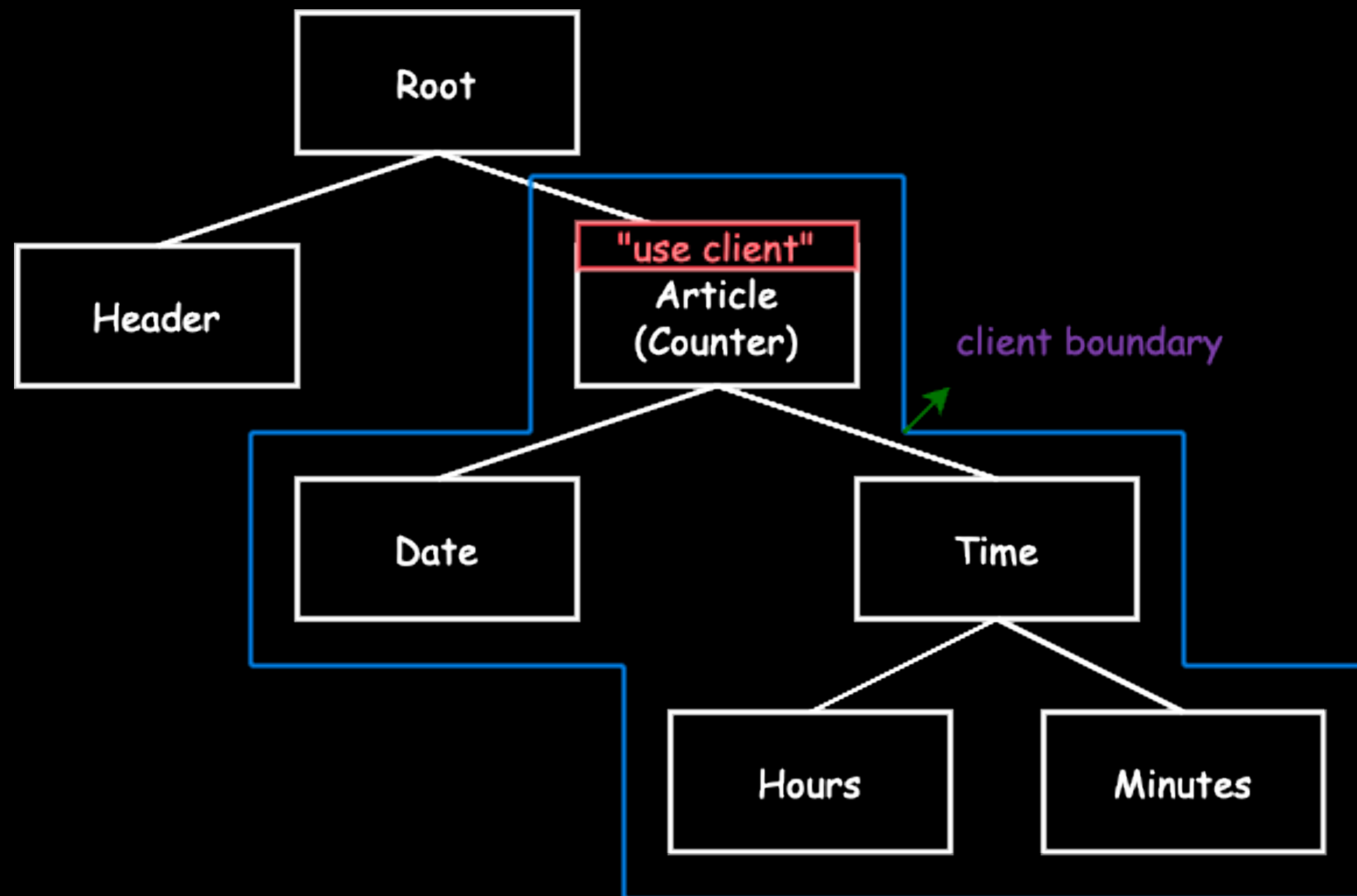
- 이를 해결하기 위해서 클라이언트 컴포넌트로 변환해야 한다.



1. React Server Component

컴포넌트 경계

- 클라이언트 컴포넌트로 변환하면 서버 컴포넌트도 재렌더링 되는 경계가 생성된다.



1. React Server Component

컴포넌트 경계 패턴

- 이때 컴포넌트 전역에 use client 키워드를 사용하기 싫다면 Provider 컴포넌트를 감싸면 된다.

```
export default function RootLayout({ children, header, footer }: Props) {
  return (
    <html lang="ko">
      <body className="select-none bg-black">
        <Providers>
          <GlobalHooks>
            <div className="flex flex-col justify-start w-full min-h-[var(--vh)] px-8 pc:px-0">
              {header}
              <div className="flex flex-col items-center">{children}</div>
              {footer}
            </div>
          </GlobalHooks>
        </Providers>
      </body>
    </html>
  );
}
```

```
'use client';

import { useState } from 'react';

import { QueryClientProvider } from '@tanstack/react-query';
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';

// import ToastProvider from '@components/organisms/ToastProvider';
import use100vh from '@hooks/use100vh';
import getQueryClient from '@utils/getQueryClient';

const Providers = ({ children }) => {
  const [queryClient : QueryClient ] = useState(() => getQueryClient());

  use100vh();

  return (
    <QueryClientProvider client={queryClient}>
      {children}
      <ReactQueryDevtools initialIsOpen={false} />
    </QueryClientProvider>
  );
};

export default Providers;
```


1. React Server Component

컴포넌트 경계 패턴??

- use client 지시자를 사용하면 import한 파일도 모두 클라이언트 컴포넌트여야 한다.
- 이 패턴을 사용하면 해당 문제를 회피 가능하다.

1. React Server Component

서버 컴포넌트의 장점

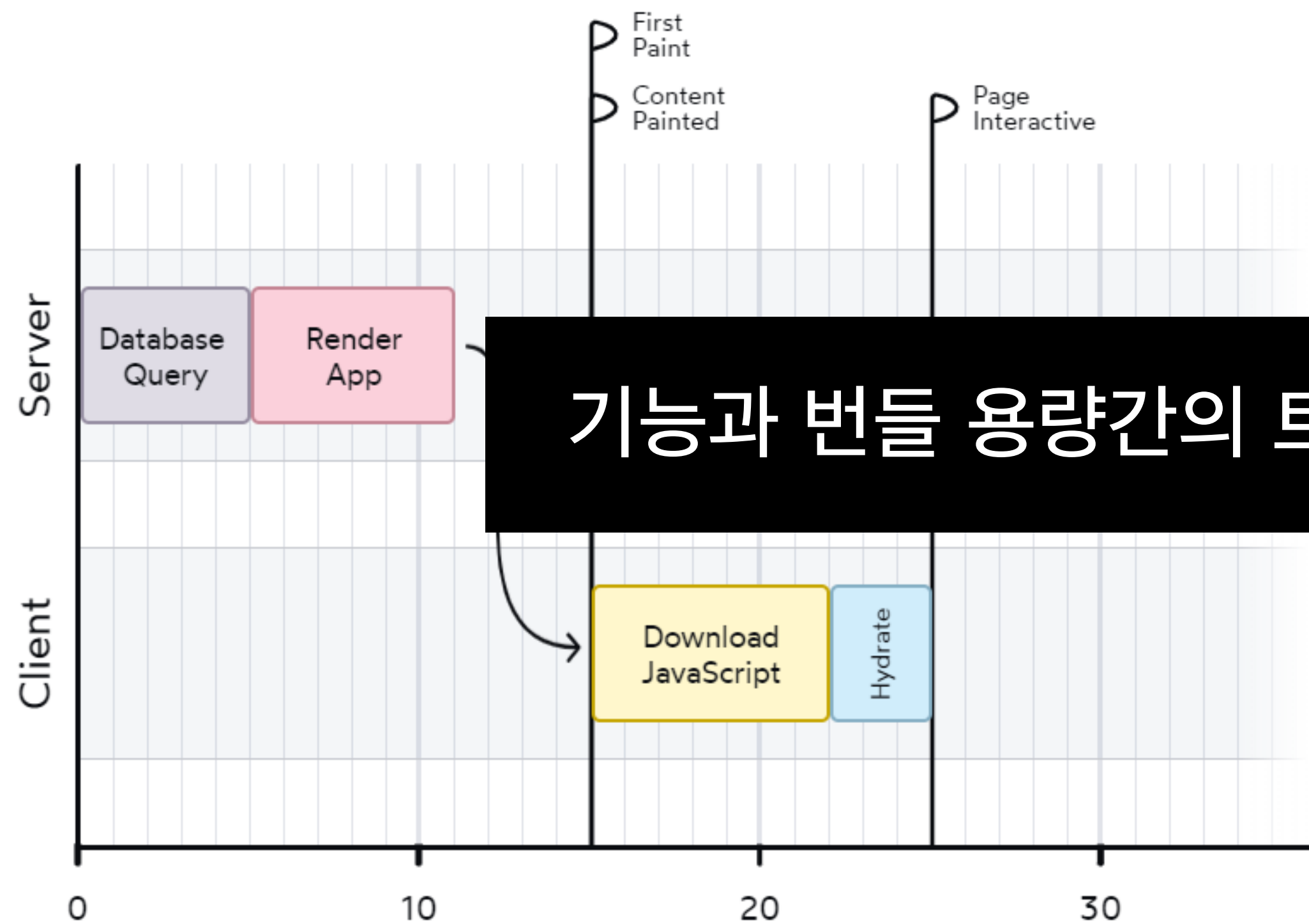
- 서버 컴포넌트는 서버에서만 코드를 실행한다.
- 자바스크립트 번들에 포함되지 않는다.
- 번들 파일과 하이드레이션해야 하는 컴포넌트의 수가 줄어든다.

1. React Server Component

서버 컴포넌트의 장점

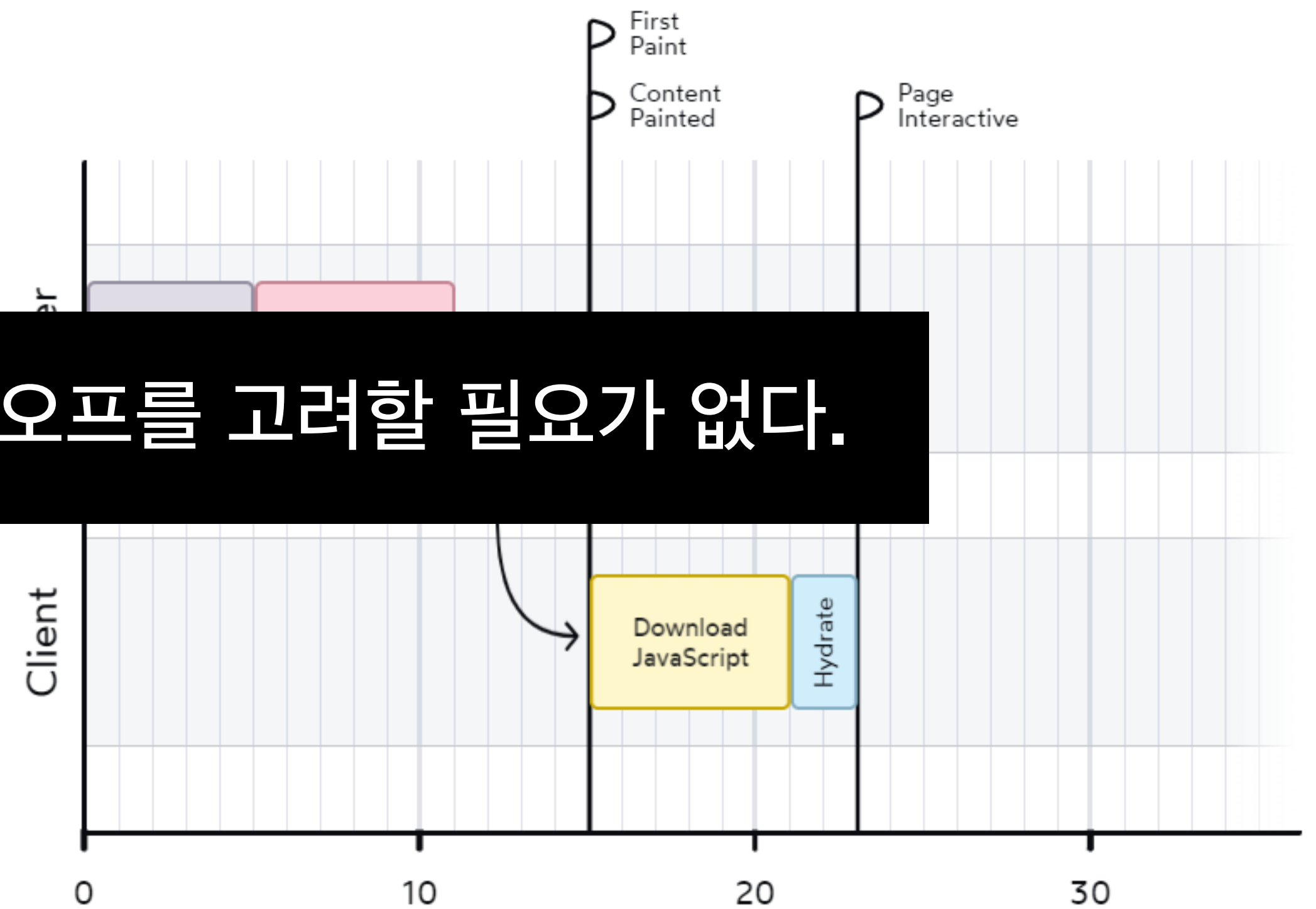
Legacy Next.js (pre-RSC)

Before After



Modern Next.js (with RSC)

Before After



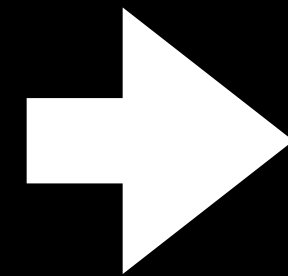
기능과 번들 용량간의 트레이드오프를 고려할 필요가 없다.

2. Next.js App directory

2. Next.js App directory

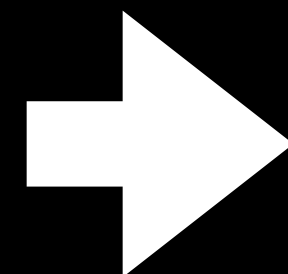
페이지 라우터와의 차이

레이아웃 페이지 구조



라우팅과 파일 컨벤션

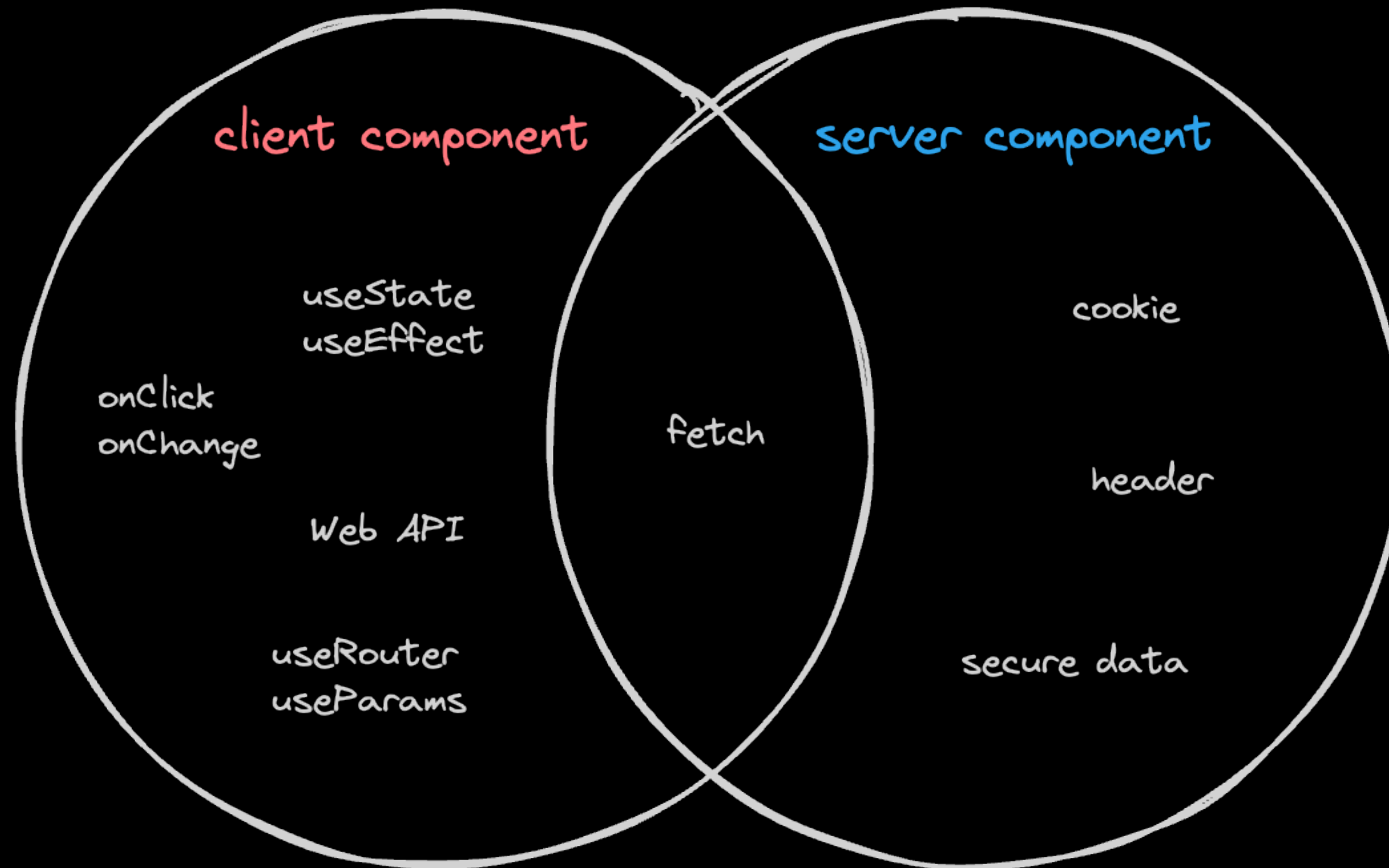
서버와의 통신 방법



서버 액션

2. Next.js App directory

Next.js에서 Client Component와 Server Component



2. Next.js App directory

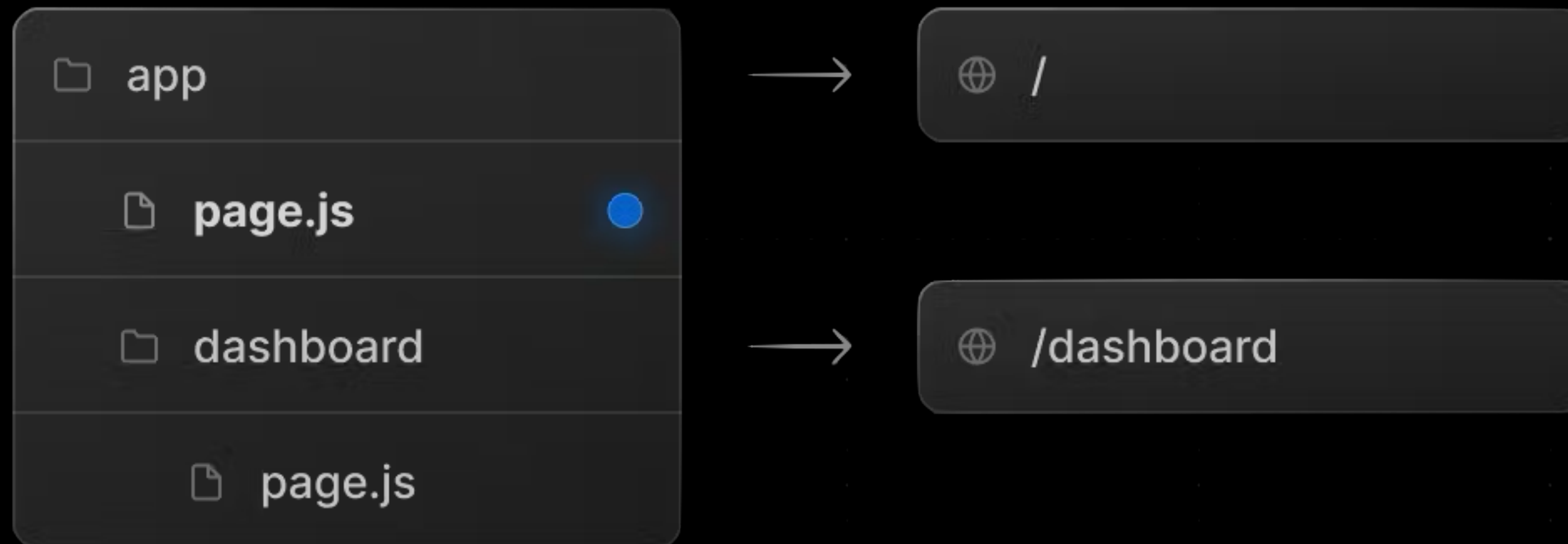
File Convention

- layout
- page
- loading
- error
- route
- template
- default
- global-error
- not-found

2. Next.js App directory

File Convention - page

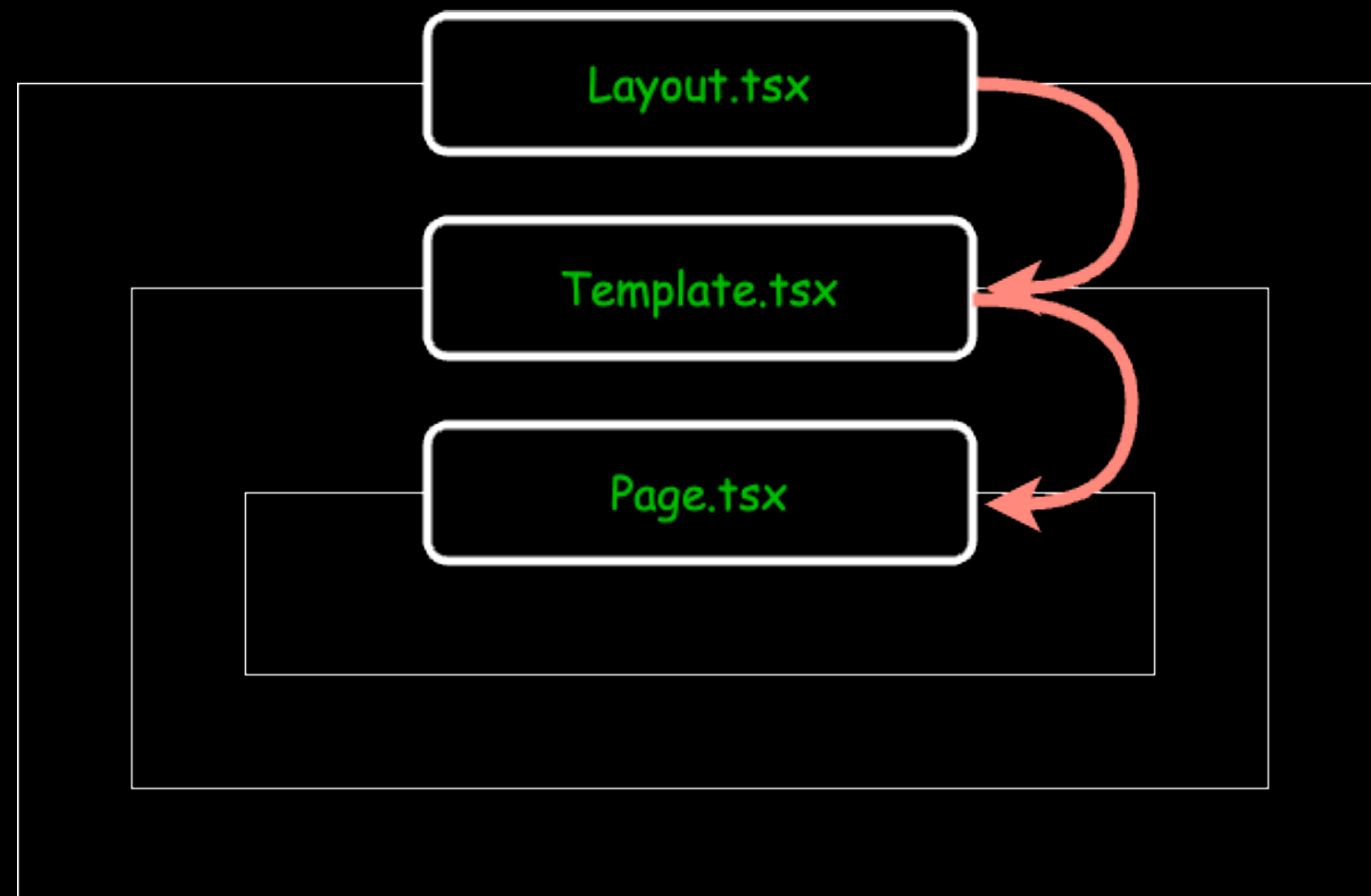
- 말 그대로 페이지이다.
- 라우팅 역할을 하며, 유니크하다.



2. Next.js App directory

File Convention - layout, template

- page가 한번 반환되기 전에 감싸진다.
- `_app`, `_document` 대신 root layout을 사용한다.
- `template`는 `layout`과 비슷하지만 라우팅 변경시 리렌더링 된다.



2. Next.js App directory

File Convention - error

- 컴포넌트 에러 바운드에 넘겨진다.
- global-error는 전역 에러를 처리한다.
- 404를 처리하는 not-found와 같은 특수한 파일도 있다.

2. Next.js App directory

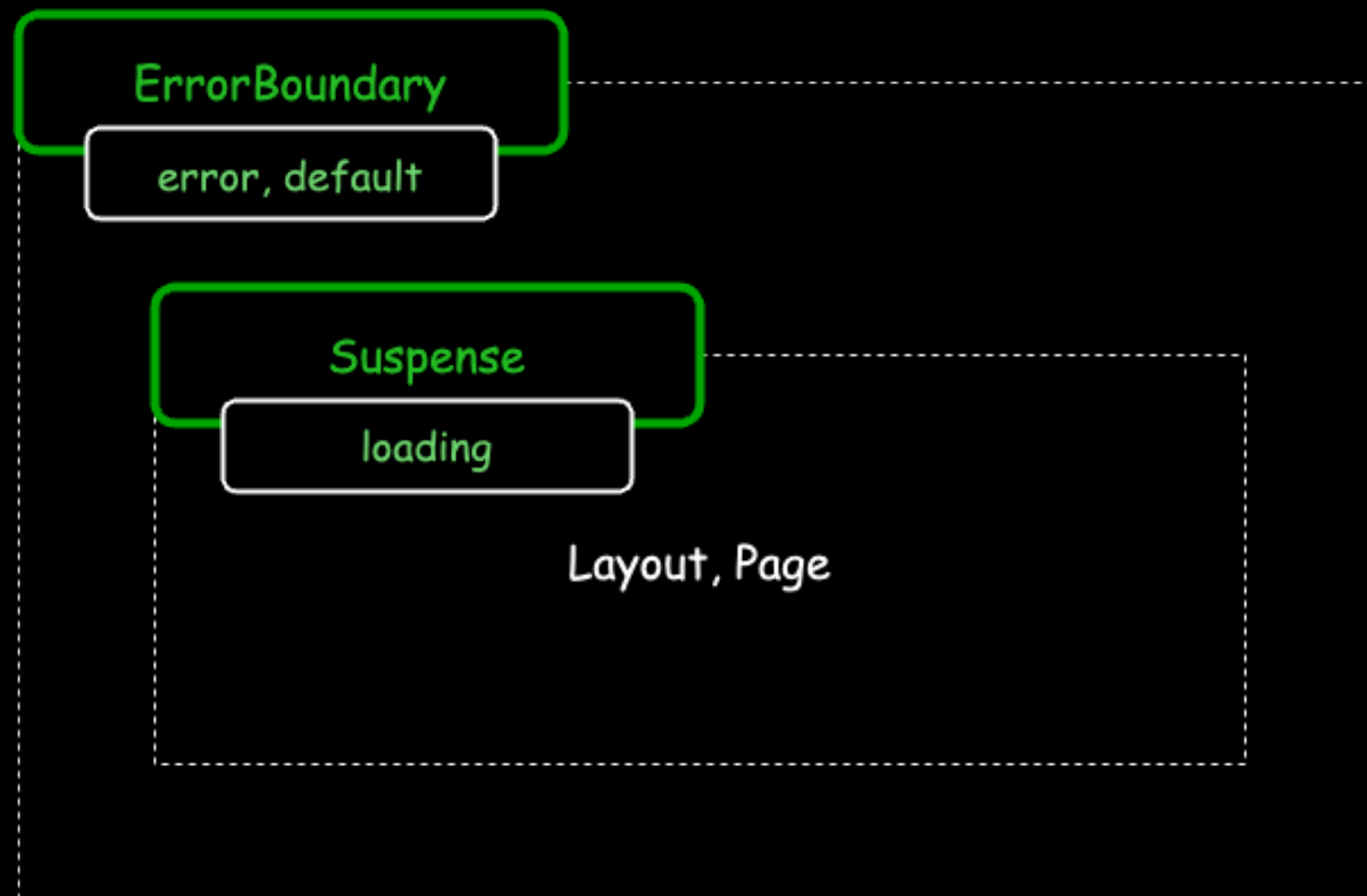
File Convention - route

- Page dir에서 page 폴더의 api폴더 역할을 하는 파일이다.
- page와 비슷하게 라우터 역할을 한다.

2. Next.js App directory

File Convention - loading, default

- fallback으로 쓰인다.
- default는 병렬 라우팅 시, loading은 안쪽 컴포넌트 작업 시 주로 보여진다.



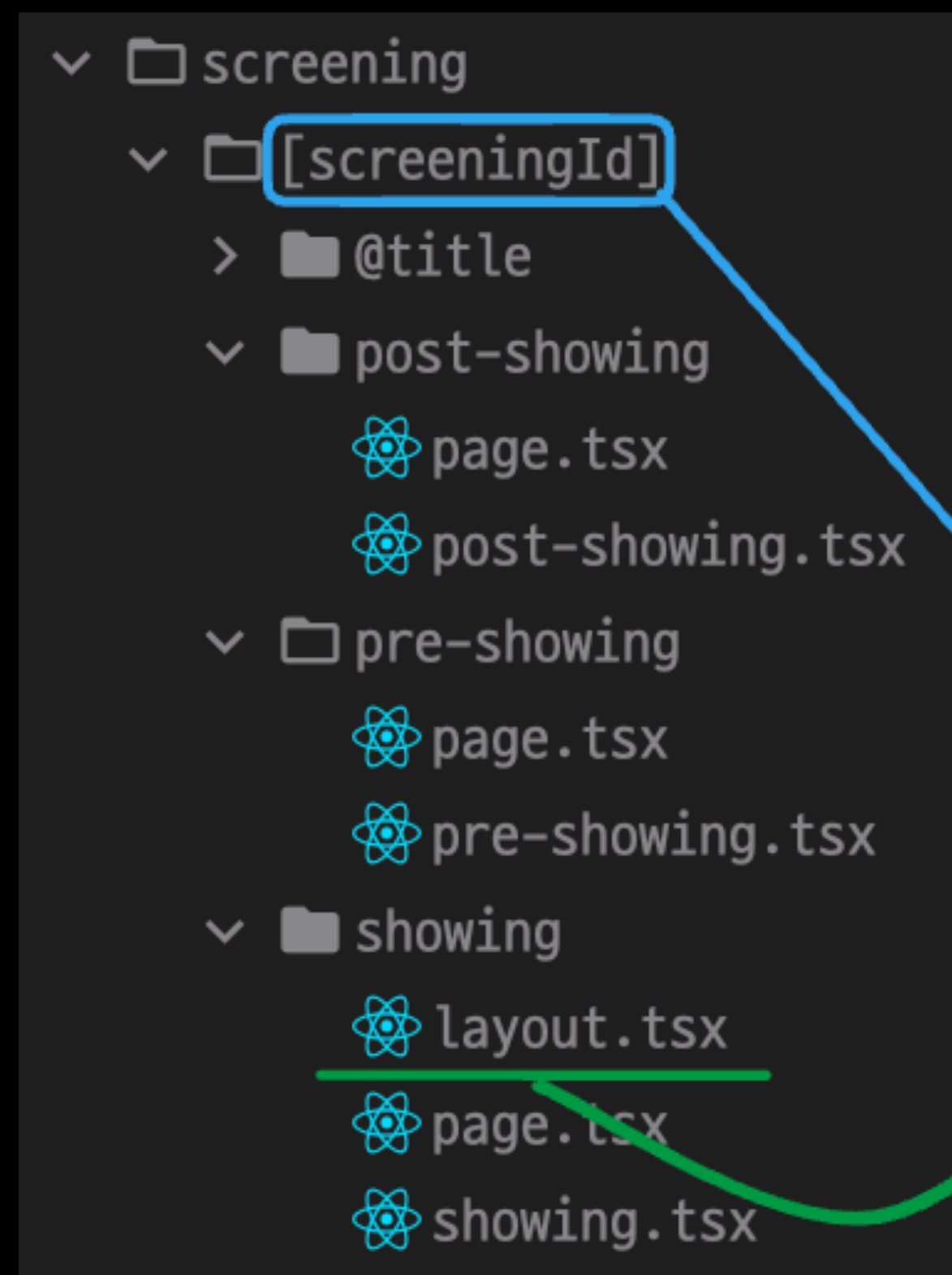
2. Next.js App directory

Folder Convention - dynamic routes

- [folder], [...folder], [...folder]]
- 동적 라우팅을 위한 세그먼트
- Page dir에선 useRouter hook이 여러 역할을 했다면 App dir에선 각각의 기능으로 쪼개진 hook이 존재한다.
- hook 없이 컴포넌트의 매개변수로도 가져올 수 있다.

2. Next.js App directory

Folder Convention - dynamic routes



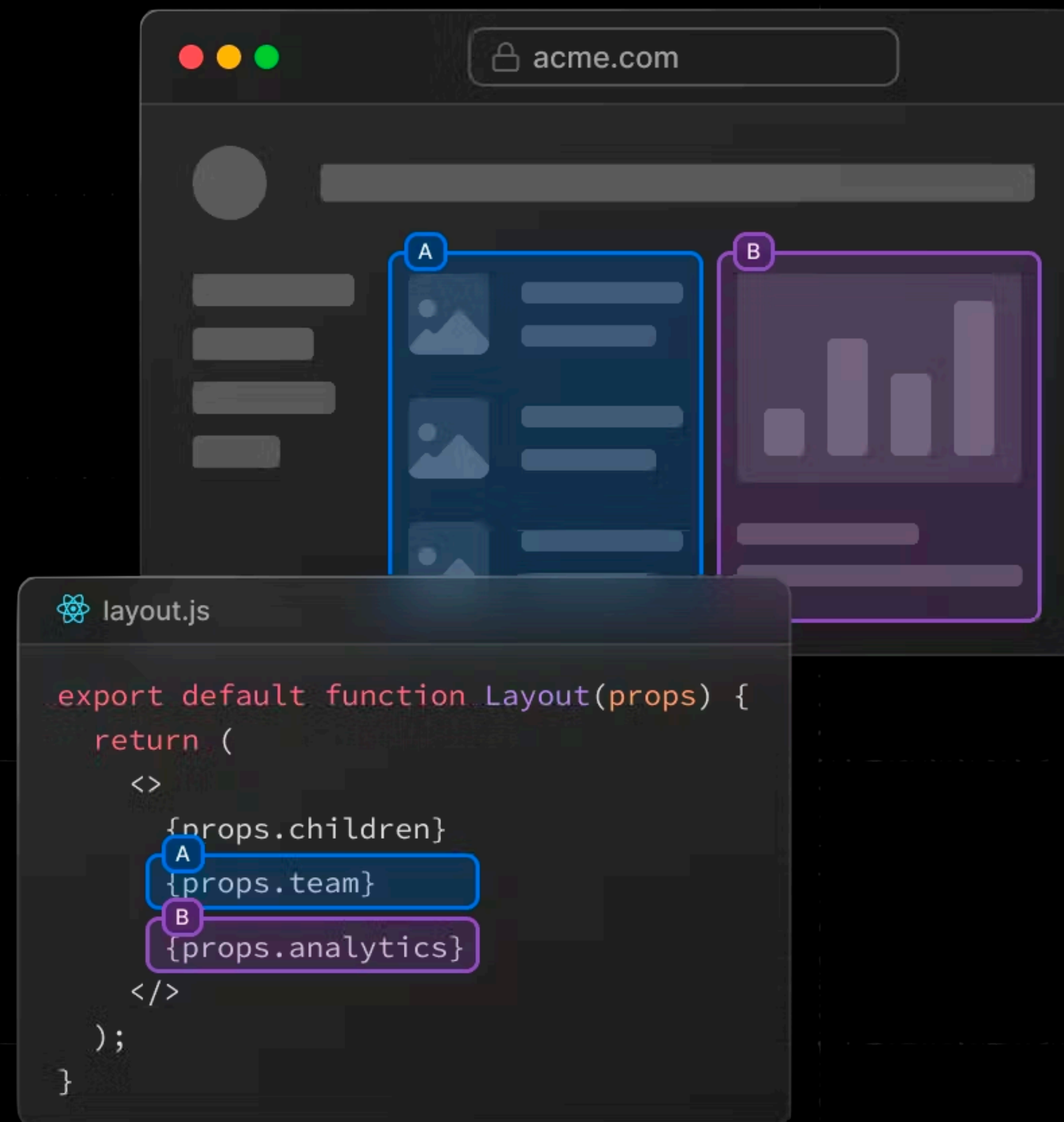
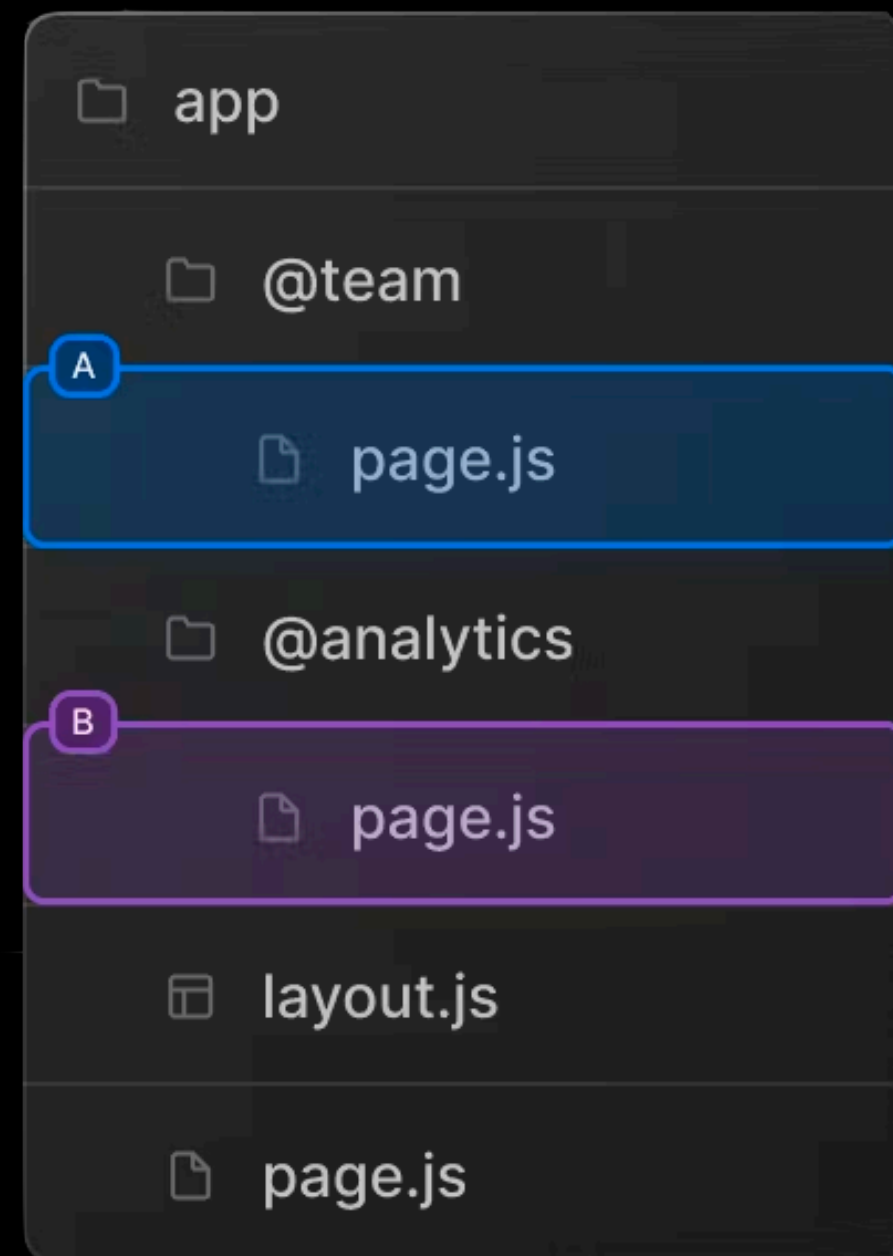
```
const router : AppRouterInstance = useRouter();
const params : Params = useParams();
const queryClient : QueryClient = useQueryClient();
const [runInterval : boolean, setRunInterval : Dispatch] = useState(false);
const [isScreeningActive : boolean, setIsScreeningActive] = useState(false);
const endPoint : MutableRefObject<number | undefined> = useRef(0);

const screeningData : Screening | undefined = queryClient.getQueryData(['screeningId']);
const { screeningId : string | string[] } = params;
```

The code snippet shows the setup for a dynamic route. The `params` property is highlighted with a red circle. The `screeningId` property is highlighted with a blue box. A red arrow points from the `params` property to the `screeningId` property in the `const { screeningId : string | string[] } = params;` line.

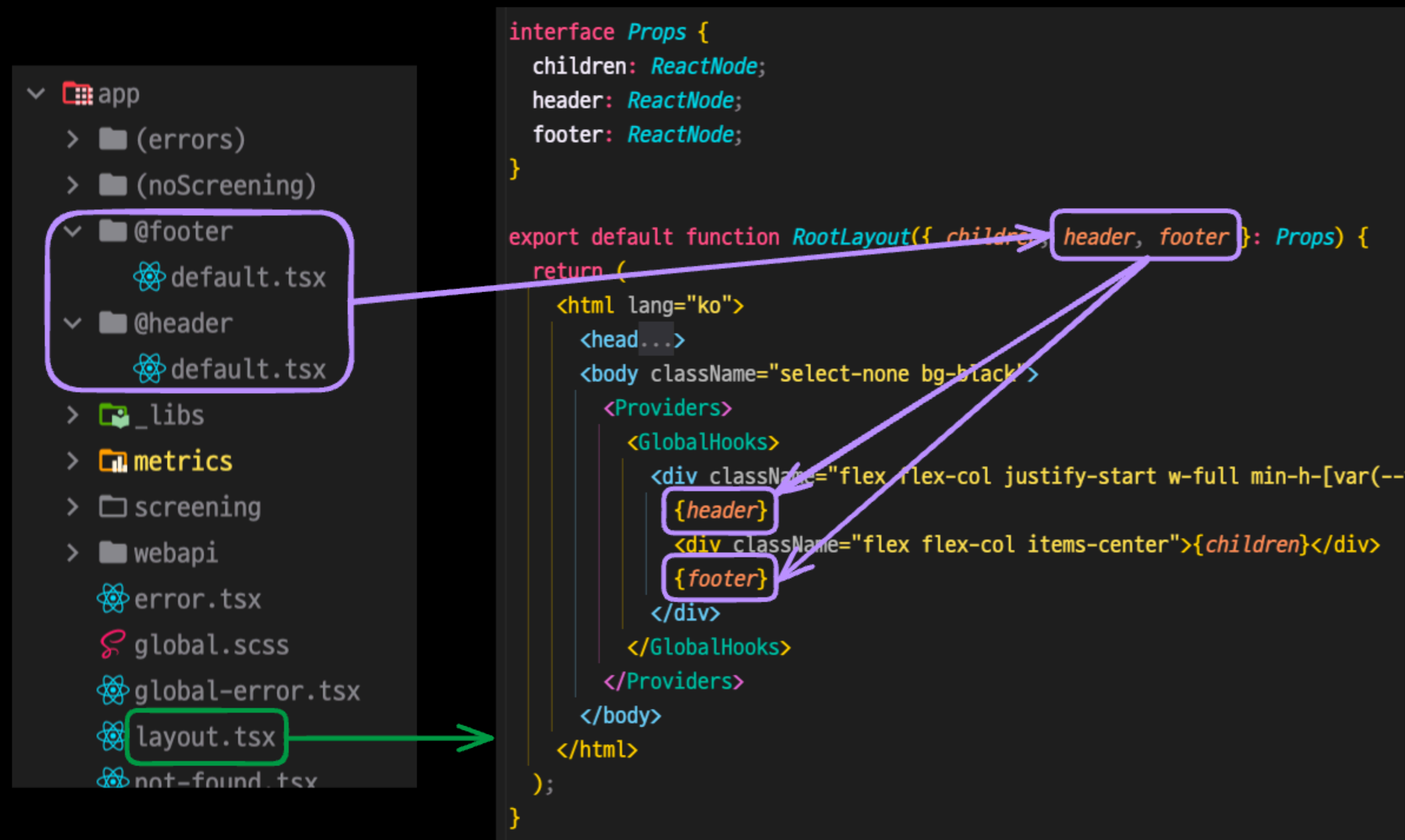
2. Next.js App directory

Folder Convention - parallel routes



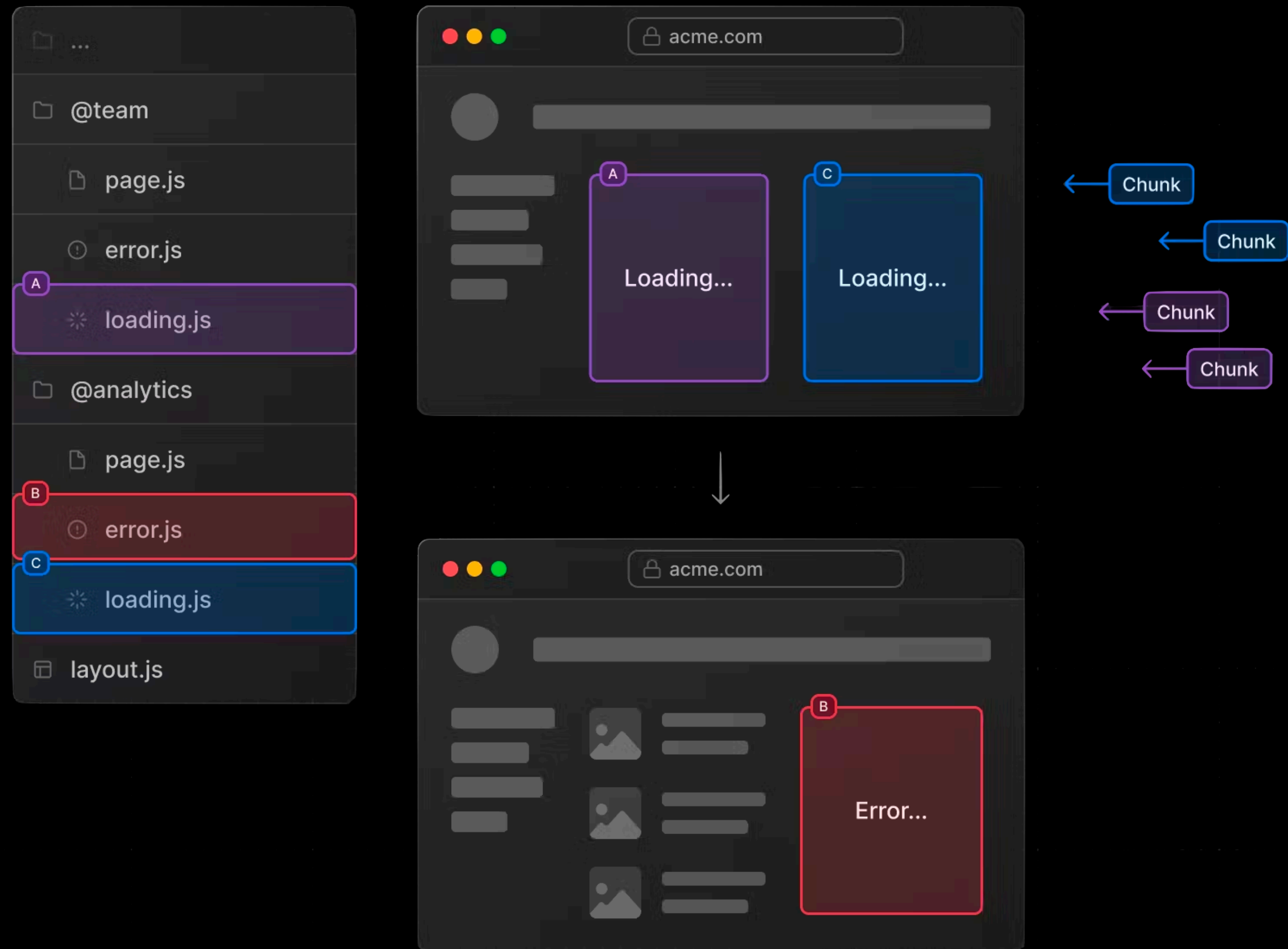
2. Next.js App directory

Folder Convention - parallel routes



2. Next.js App directory

Folder Convention - parallel routes



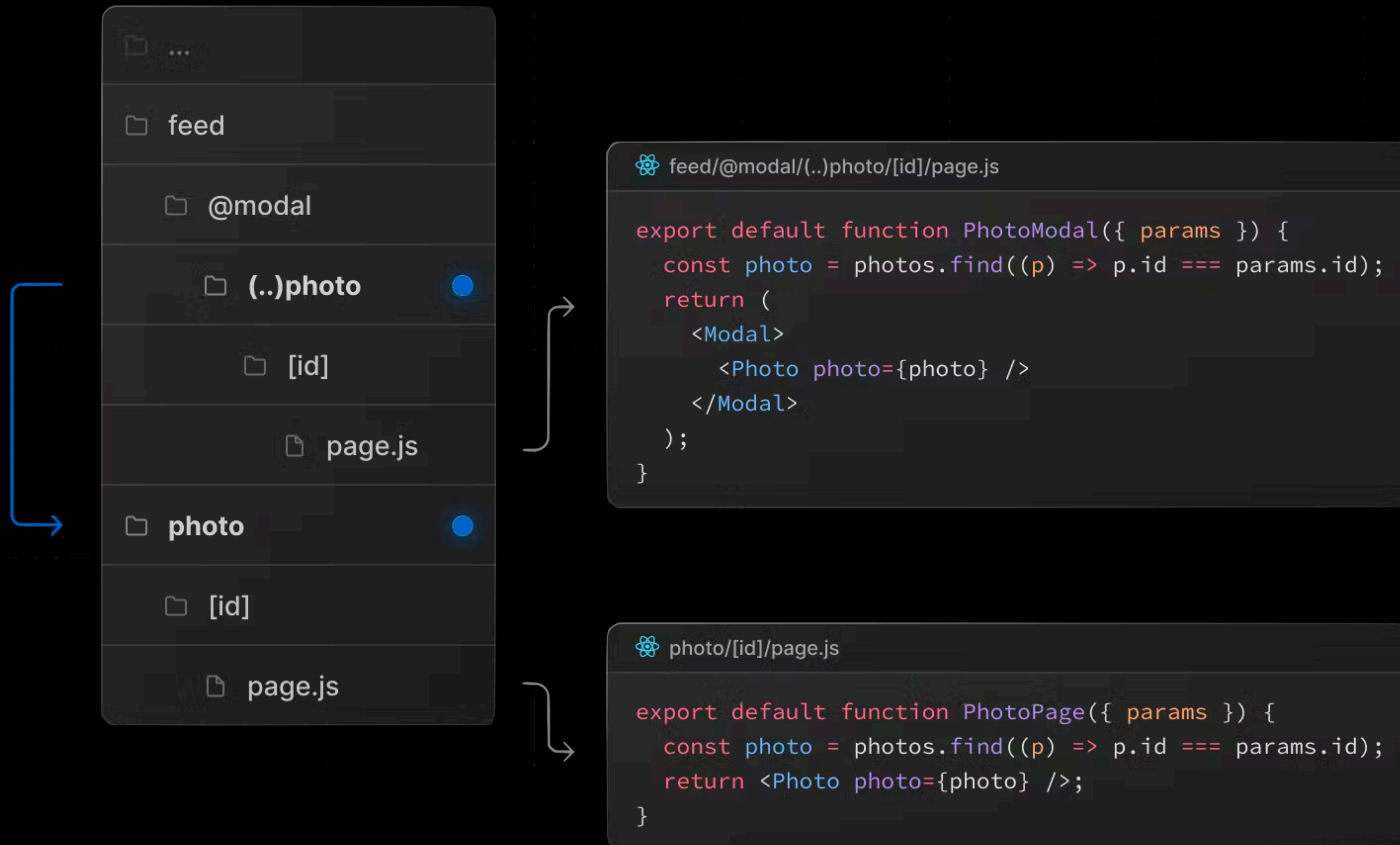
2. Next.js App directory

Folder Convention - parallel routes

- 컴포넌트를 조건문으로 분기 시키는 것이 아닌 선언적으로 분기 시킨다.
- 레이아웃 구조에 따라 선언적으로 가져온다.
- 바운더리를 병렬로 유지할 수 있다.

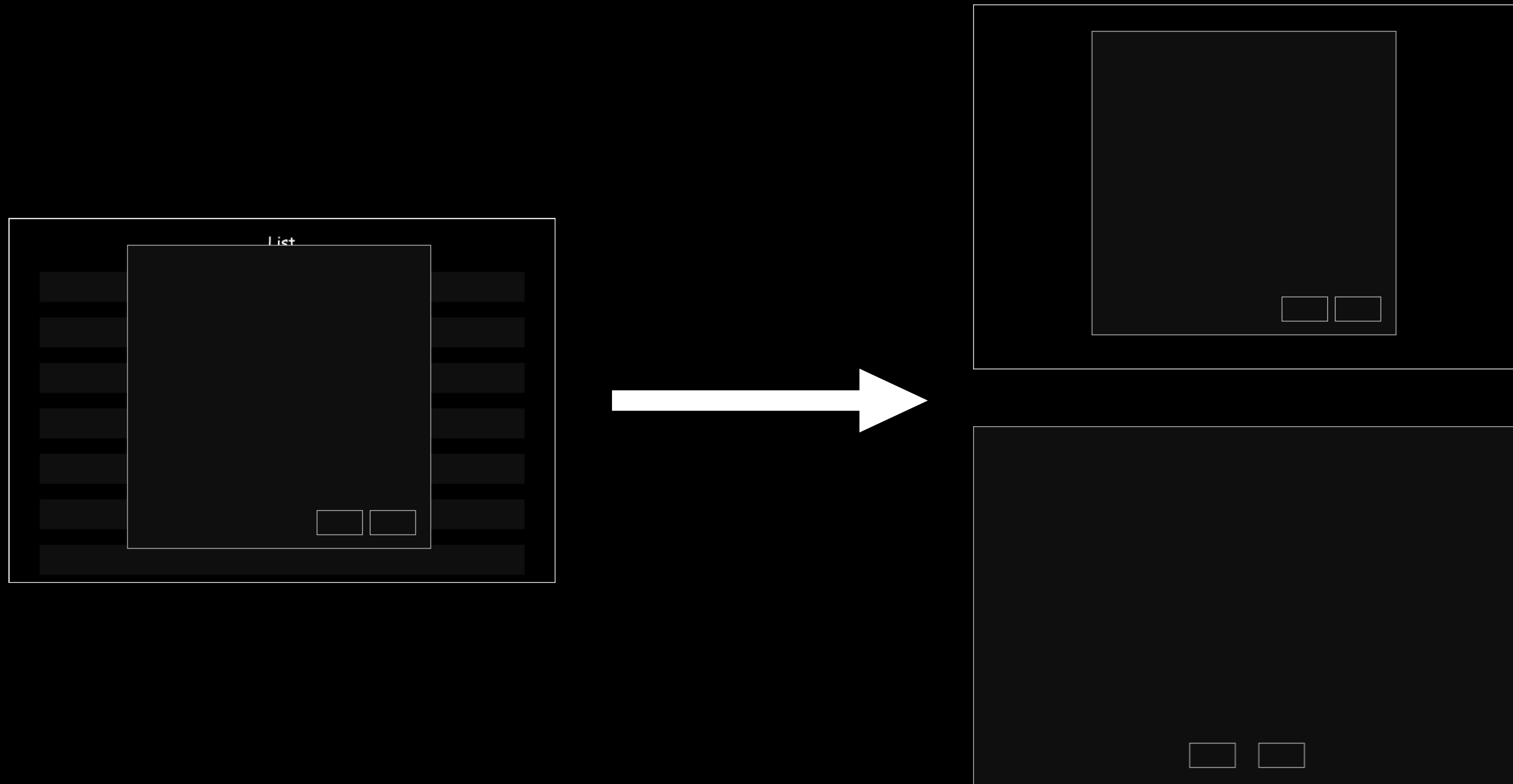
2. Next.js App directory

Folder Convention - intercepting routes



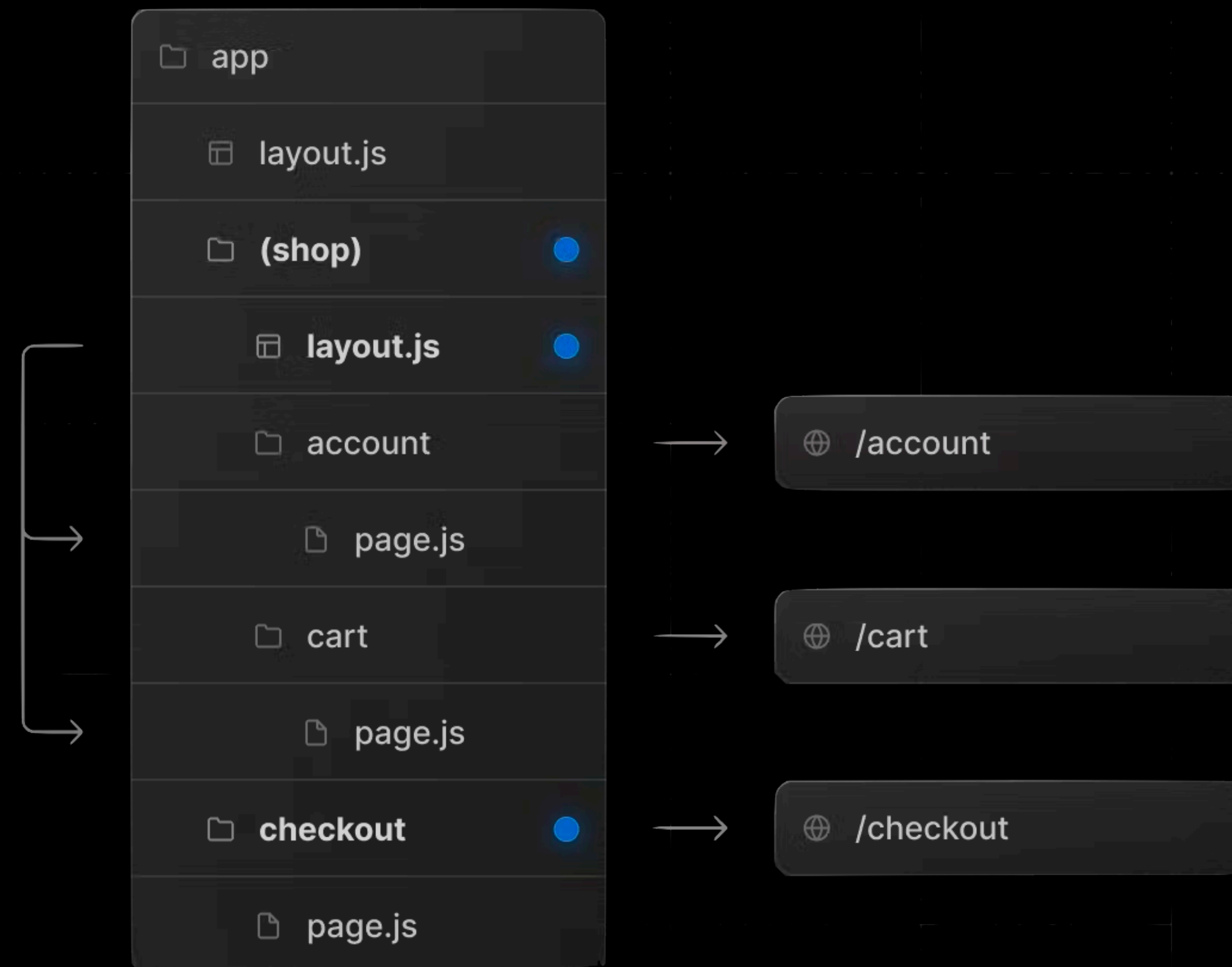
2. Next.js App directory

Folder Convention - intercepting routes



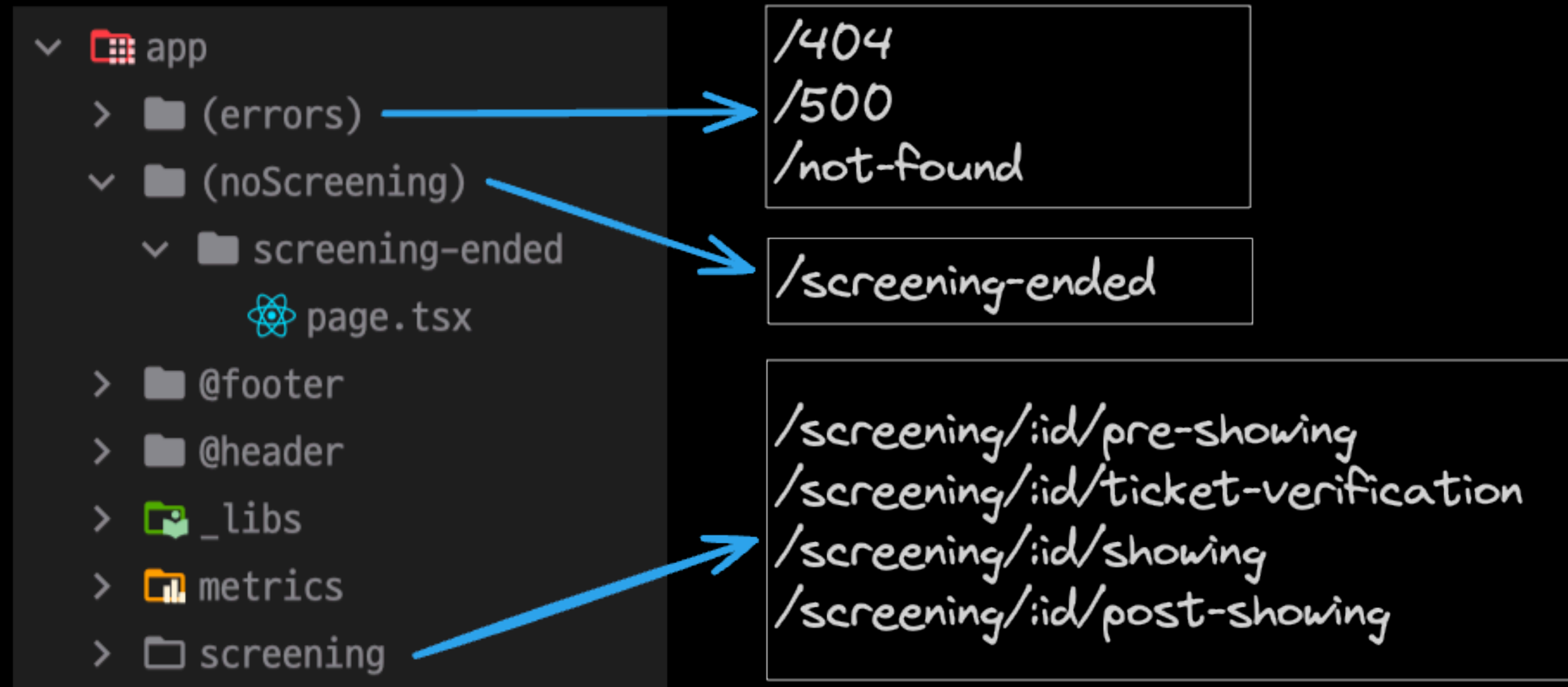
2. Next.js App directory

Folder Convention - group folder



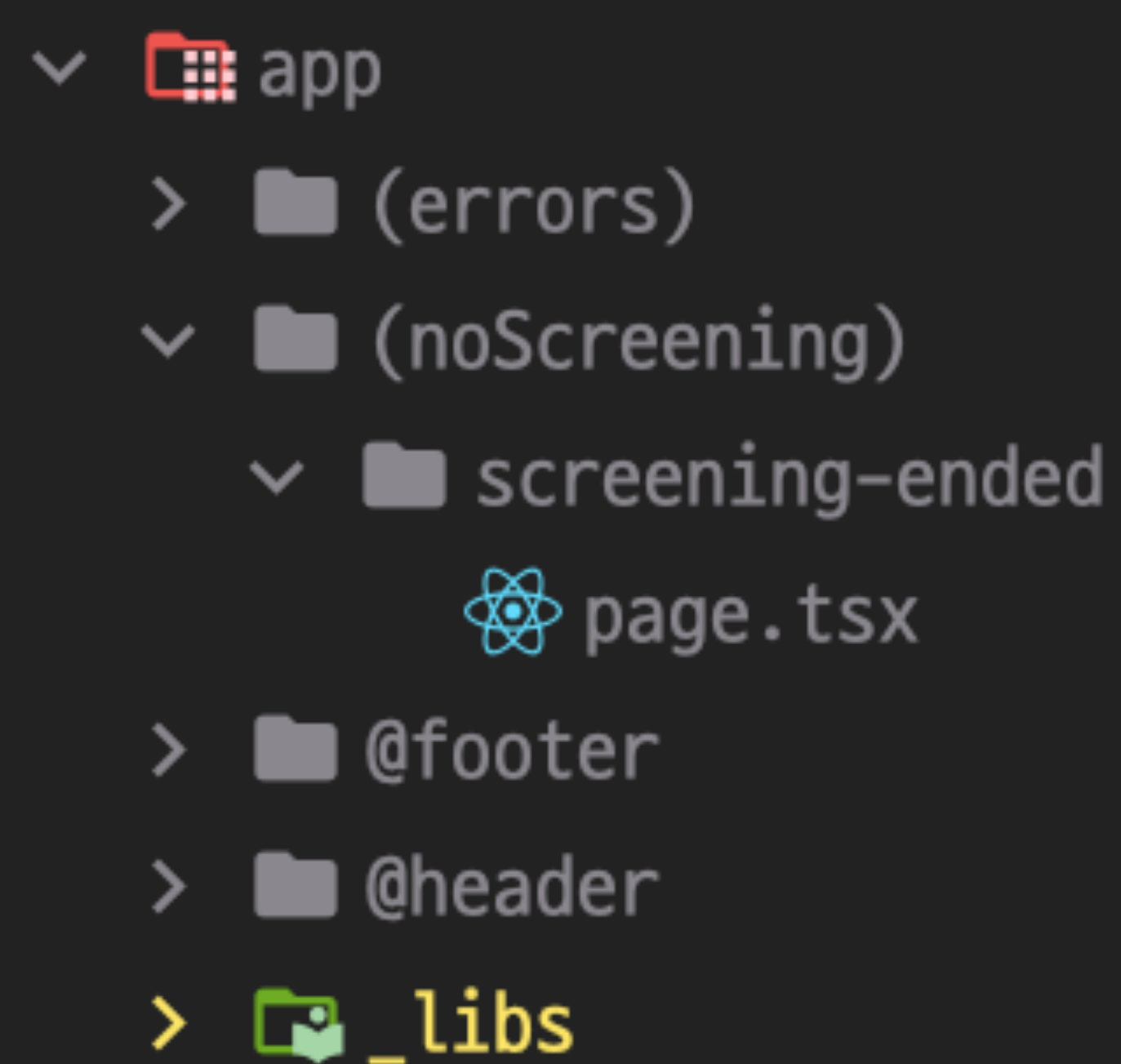
2. Next.js App directory

Folder Convention - group folder



2. Next.js App directory

Folder Convention - private folder



```

  app
  ├── (errors)
  ├── (noScreening)
  │   └── screening-ended
  │       └── page.tsx
  ├── @footer
  ├── @header
  └── _libs

```

The image shows a file explorer view of a Next.js application directory. The root folder is 'app', which is expanded. It contains several subfolders: '(errors)', '(noScreening)', '@footer', '@header', and '_libs'. The '(noScreening)' folder is further expanded, showing a subfolder 'screening-ended' which contains a file 'page.tsx'. The file explorer uses various icons: a folder icon for the root 'app', a folder icon with a grid for '(errors)', a folder icon for '(noScreening)', a folder icon for 'screening-ended', a file icon with a blue atom symbol for 'page.tsx', and a folder icon for '@footer', '@header', and '_libs'.