'use client'

React 서버 컴포넌트

`'use client'` 는 React 서버 컴포넌트와 함께 사용합니다.

`'use client'` 를 사용하여 클라이언트에서 실행되는 코드를 표시합니다.

레퍼런스

`'use client'`

파일의 최상단에 `'use client'` 를 추가하여 모듈과 해당 모듈의 전이적 의존성을 클라이언트 코드로 표시하세요.

```
'use client';

import { useState } from 'react';
import { formatDate } from './formatters';
import Button from './button';

export default function RichTextEditor({ timestamp, text }) {
  const date = formatDate(timestamp);
  // ...
  const editButton = <Button />;
  // ...
}
```

서버 컴포넌트에서 `'use client'` 라 표시된 파일을 가져오면 호환되는 번들러는 모듈 불러오기 Module Import를 서버 실행 코드와 클라이언트 실행 코드 사이의 경계로 처리합니다.

`RichTextEditor` 의 의존성으로 인하여, `formatDate` 와 `Button` 의 모듈에 `'use client'` 지시어가 포함되어 있지 않더라도 클라이언트에서 평가됩니다. 하나의 모듈이 서버 코드에서 가져올 때는 서버에서, 클라이언트 코드에서 가져올 때는 클라이언트에서 평가될 수 있음을 유의해야 합니다.

주의 사항

- `'use client'` 는 파일의 맨 처음에 있어야 하며, 다른 코드나 import 문보다 위에 있어야 합니다. (주석은 괜찮습니다.) 작은따옴표나 큰따옴표로 작성해야 하며 백틱은 사용할 수 없습니다.
- `'use client'` 모듈을 다른 클라이언트 렌더링 모듈에서 가져오면 지시어가 동작하지 않습니다.
- 컴포넌트 모듈에 `'use client'` 지시어가 포함된 경우 해당 컴포넌트의 사용은 클라이언트 컴포넌트임이 보장됩니다. 하지만 컴포넌트에 `'use client'` 지시어가 없더라도 클라이언트에서 평가될 수 있습니다.
  - 컴포넌트 사용은 `'use client'` 지시어가 포함된 모듈에 정의되어 있거나 `'use client'` 지시어를 포함한 모듈의 전이적 의존성일 경우 클라이언트 컴포넌트로 간주합니다. 그렇지 않으면 서버 컴포넌트로 간주합니다.
- 클라이언트 평가로 표시된 코드는 컴포넌트에만 국한되지 않습니다. 클라이언트 모듈 하위 트리의 모든 코드는 클라이언트에 전송되어 클라이언트에서 실행됩니다.

- 서버 평가 모듈이 `'use client'` 모듈에서 값을 가져올 때, 그 값은 React 컴포넌트이거나 클라이언트 컴포넌트에 전달될 수 있는 지원되는 직렬화 가능한 prop 값이어야 합니다.

`'use client'`가 클라이언트 코드를 표시하는 방법

React 앱에서 컴포넌트는 종종 별도의 파일 또는 모듈로 분리됩니다.

React 서버 컴포넌트를 사용하는 앱의 경우, 기본적으로 앱은 서버에서 렌더링됩니다. `'use client'`는 모듈 의존성 트리에 서버-클라이언트 경계를 도입하여 효과적으로 클라이언트 모듈의 하위 트리를 만듭니다.

이를 더 잘 설명하기 위해 다음과 같은 React 서버 컴포넌트 앱을 고려해 보세요.

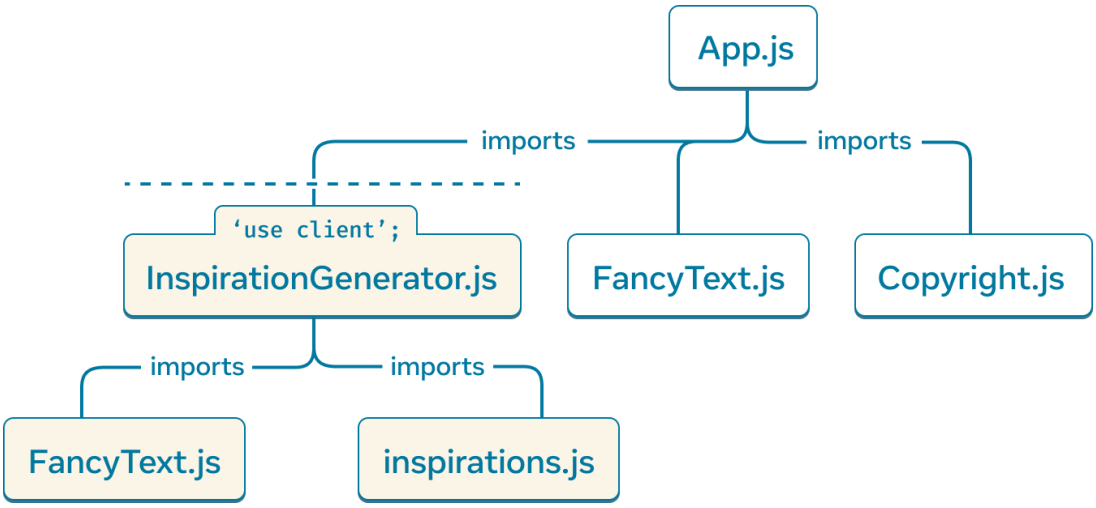
App.js

새로고침 Clear 포크

```
import FancyText from './FancyText';
import InspirationGenerator from './InspirationGenerator';
import Copyright from './Copyright';

export default function App() {
  return (
    <>
      <FancyText title text="Get Inspired App" />
      <InspirationGenerator>
        <Copyright year={2004} />
      </InspirationGenerator>
    </>
```

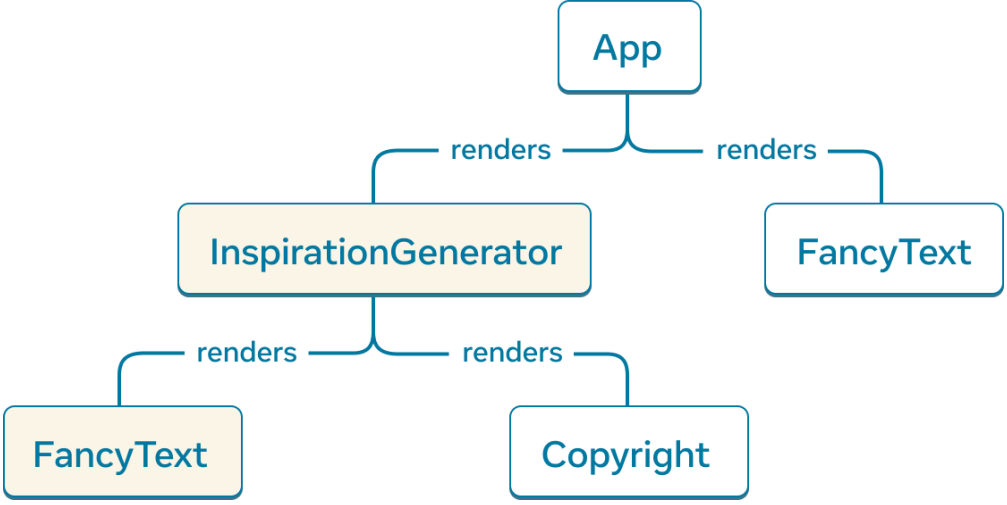
예시 앱의 모듈 의존성 트리에서 `InspirationGenerator.js` 의 `'use client'` 지시어는 해당 모듈과 모든 전이적 의존성을 클라이언트 모듈로 표시합니다. 이제 `InspirationGenerator.js` 에서 시작하는 하위 트리는 클라이언트 모듈로 표시됩니다.



`'use client'` 는 React 서버 컴포넌트 앱의 모듈 의존성 트리를 분할하여 `InspirationGenerator.js` 와 모든 의존성을 클라이언트-렌더링으로 표시합니다.

렌더링하는 동안 프레임워크는 루트 컴포넌트를 서버-렌더링하고 렌더 트리를 통해 계속 진행하여 클라이언트에서 가져온 코드를 평가하지 않습니다.

그런 다음 서버에서 렌더링한 렌더 트리 부분을 클라이언트로 보냅니다. 클라이언트 코드를 다운로드한 클라이언트는 트리의 나머지 부분 렌더링을 완료합니다.



React 서버 컴포넌트 앱을 위한 렌더 트리에서 `InspirationGenerator` 와 그 자식 컴포넌트 `FancyText` 는 클라이언트 표시 코드에서 내보낸 컴포넌트이며 클라이언트 컴포넌트로 간주합니다.

다음 정의를 소개합니다.

- **클라이언트 컴포넌트**는 클라이언트에서 렌더링되는 렌더 트리의 컴포넌트입니다.
- **서버 컴포넌트**는 서버에서 렌더링 되는 렌더 트리의 컴포넌트입니다.

예시 앱이 실행되는 동안 `App`, `FancyText` 및 `Copyright` 는 모두 서버에서 렌더링되며 서버 컴포넌트로 간주합니다. `InspirationGenerator.js` 와 그 전이적 의존성이 클라이언트 코드로 표시되므로 컴포넌트 `InspirationGenerator` 와 그 자식 컴포넌트 `FancyText` 는 클라이언트 컴포넌트입니다.

자세히 살펴보기

어떻게 `FancyText` 는 서버 컴포넌트이면서 클라이언트 컴포넌트인가요?

자세히 보기

자세히 살펴보기

왜 `Copyright` 가 서버 컴포넌트인가요?

자세히 보기

`'use client'` 의 사용 시기

`'use client'` 를 사용하면 컴포넌트가 클라이언트 컴포넌트인지 확인할 수 있습니다. 서버 컴포넌트가 기본값이므로, 클라이언트에서 렌더링할 것을 표시해야 하는 시기를 결정하기 위해 서버 컴포넌트의 장단점을 간단히 살펴보겠습니다.

간단히 설명하기 위해 서버 컴포넌트에 대해 이야기하지만, 서버에서 실행되는 앱의 모든 코드에는 동일한 원칙이 적용됩니다.

서버 컴포넌트의 장점

- 서버 컴포넌트는 클라이언트에 전송되고 실행되는 코드의 양을 줄일 수 있습니다. 클라이언트 모듈만 클라이언트에서 번들링되고 평가됩니다.
- 서버 컴포넌트는 서버에서 실행할 때 이점이 있습니다. 로컬 파일 시스템에 접근할 수 있으며 데이터 가져오기 및 네트워크 요청에 대한 짧은 지연 시간을 경험할 수 있습니다.

서버 컴포넌트의 한계

- 클라이언트에서 이벤트 핸들러를 등록하고 트리거해야 하므로 서버 컴포넌트는 상호작용을 지원할 수 없습니다.
  - 예를 들어 `onClick` 과 같은 이벤트 핸들러는 클라이언트 컴포넌트에서만 정의할 수 있습니다.
- 서버 컴포넌트는 대부분의 Hook을 사용할 수 없습니다.
  - 서버 컴포넌트가 렌더링될 때, 그 출력은 기본적으로 클라이언트가 렌더링할 컴포넌트 목록입니다. 서버 컴포넌트는 렌더링 후 메모리에 유지되지 않으며 자체적인 State를 가질 수 없습니다.

서버 컴포넌트에서 반환하는 직렬화 가능한 유형

React 앱에서와 같이 부모 컴포넌트는 자식 컴포넌트에 데이터를 전달합니다. 서로 다른 환경에서 렌더링되므로 서버 컴포넌트에서 클라이언트 컴포넌트로 데이터를 전달하는 것은 추가적인 고려 사항이 필요합니다.

서버 컴포넌트에서 클라이언트 컴포넌트로 전달하는 Prop 값은 직렬화할 수 있어야 합니다.

직렬화할 수 있는 Props는 다음과 같습니다.

- 원시 자료형
  - string
  - number
  - bigint
  - boolean
  - undefined
  - null
  - symbol (`Symbol.for` 를 통해 전역 심볼 레지스트리에 등록된 심볼만 해당)
- 직렬화할 수 있는 값을 포함하는 이터러블
  - String
  - Array
  - Map
  - Set
  - TypedArray 및 ArrayBuffer
- Date
- 일반 객체 (직렬화할 수 있는 프로퍼티를 사용하여 객체 이니셜라이저로 생성된 객체)
- 서버 함수로서의 함수
- 클라이언트 또는 서버 컴포넌트 요소(JSX)
- Promise

단, 다음은 지원되지 않습니다.

- 클라이언트로 표시된 모듈에서 내보내지 않았거나 `'use server'` 로 표시된 함수
- 클래스
- 위에서 언급한 내장형 클래스의 인스턴스가 아닌 객체 혹은 null 프로토타입을 가진 객체
- 전역에 등록되지 않은 Symbol (예: `Symbol('my new symbol')`)

사용법

상호작용 및 State를 가진 컴포넌트 구축

App.js 다운로드 새로고침 Clear 포크

```
'use client';

import { useState } from 'react';

export default function Counter({initialValue = 0}) {
  const [countValue, setCountValue] = useState(initialValue);
  const increment = () => setCountValue(countValue + 1);
  const decrement = () => setCountValue(countValue - 1);
  return (
    <>
      <h2>Count Value: {countValue}</h2>
      <button onClick={increment}>+1</button>
```

자세히 보기

`Counter` 에는 값을 증가시키거나 감소시키기 위해 `useState` Hook과 이벤트 핸들러가 모두 필요하므로 이 컴포넌트는 클라이언트 컴포넌트여야 하며 파일 최상단에 `'use client'` 지시어가

필요합니다.

반면에 상호작용 없이 UI를 렌더링하는 컴포넌트는 클라이언트 컴포넌트일 필요가 없습니다.

```
import { readFile } from 'node:fs/promises';
import Counter from './Counter';

export default async function CounterContainer() {
  const initialValue = await readFile('/path/to/counter_value');
  return <Counter initialValue={initialValue} />
}
```

예를 들어, `Counter` 의 상위 컴포넌트인 `CounterContainer` 는 상호작용이 없고 State를 사용하지 않기 때문에 `'use client'` 를 사용할 필요가 없습니다. 또한 `CounterContainer` 는 서버의 로컬 파일 시스템을 읽어야 하므로, 이것이 가능한 서버 컴포넌트여야만 합니다.

서버나 클라이언트 전용 기능을 사용하지 않고 렌더링 위치에 구애받지 않는 컴포넌트도 있습니다. 앞서 예로 든 `FancyText` 가 그러한 컴포넌트 중 하나입니다.

```
export default function FancyText({title, text}) {
  return title
    ? <h1 className='fancy title'>{text}</h1>
    : <h3 className='fancy cursive'>{text}</h3>
}
```

이 경우 `'use client'` 지시어를 추가하지 않으면 `FancyText` 의 *산출물*(소스 코드가 아닌)이 서버 컴포넌트에서 참조될 때 브라우저로 전송됩니다. 앞서 Inspirations 앱 예시에서 보여준 것처럼 `FancyText` 는 가져오고 사용되는 위치에 따라 서버 또는 클라이언트 컴포넌트로 사용됩니다.

하지만 `FancyText` 의 HTML 출력이 (의존성을 포함한) 소스 코드에 비해 크다면, 항상 클라이언트 컴포넌트로 강제하는 것이 더 효율적일 수 있습니다. 한 예로 긴 SVG 경로 문자열을 반환하는 컴포넌트를 클라이언트 컴포넌트로 강제하는 것이 더 효율적일 수 있는 것처럼 말입니다.

클라이언트 API 사용

React 앱에서는 웹 스토리지, 오디오 및 비디오 조작, 하드웨어 장치 등과 같은 브라우저의 API를 포함한 클라이언트 전용 API를 사용할 수 있습니다.

이 예시에서 컴포넌트는 DOM API를 사용해 `canvas` 요소를 조작합니다. 이러한 API는 브라우저에서만 사용할 수 있으므로 클라이언트 컴포넌트로 표시되어야 합니다.

```
'use client';

import {useRef, useEffect} from 'react';

export default function Circle() {
  const ref = useRef(null);
  useLayoutEffect(() => {
    const canvas = ref.current;
    const context = canvas.getContext('2d');
    context.reset();
    context.beginPath();
    context.arc(100, 75, 50, 0, 2 * Math.PI);
    context.stroke();
  });
  return <canvas ref={ref} />;
}
```

서드파티 라이브러리 사용

React 앱에서는 서드파티 라이브러리를 활용하여 일반적인 UI 패턴이나 로직을 처리하는 경우가 많습니다.

이러한 라이브러리들은 컴포넌트 Hook이나 클라이언트 API에 의존할 수 있습니다. 다음 React API를 사용하는 서드파티 컴포넌트는 클라이언트에서 실행되어야 합니다.

- `createContext`
- `use` 및 `useId`를 제외한 `react`와 `react-dom`의 Hook
- `forwardRef`
- `memo`
- `startTransition`
- 클라이언트 API를 사용하는 경우(예: DOM 삽입 혹은 네이티브 플랫폼 뷰 등)

이 라이브러리들이 React 서버 컴포넌트와 호환되도록 업데이트되었다면 이미 `'use client'`를 포함하고 있어 서버 컴포넌트에서 직접 사용할 수 있습니다. 라이브러리가 업데이트되지 않았거나 컴포넌트가 클라이언트에서만 사용할 수 있는 이벤트 핸들러와 같은 Props가 필요한 경우, 사용할 서드파티 클라이언트 컴포넌트와 서버 컴포넌트 사이에 자체 클라이언트 컴포넌트 파일을 추가해야 할 수 있습니다.

이전
지시어

다음
'use server'