서버 컴포넌트

서버 컴포넌트는 번들링 전에 클라이언트 앱이나 SSR(Server Side Rendering) 서버와는 분리된 환경에서 미리 렌더링되는 새로운 유형의 컴포넌트입니다.

이 별도의 환경이 바로 React 서버 컴포넌트에서의 "서버"입니다. 서버 컴포넌트는 빌드 시간에 CI 서버에서 한 번 실행되거나, 각 요청마다 웹 서버를 통해 실행될 수 있습니다.

중요합니다!

서버 컴포넌트를 지원하려면 어떻게 해야 하나요?

React 19의 서버 컴포넌트는 안정적이며 마이너Minor 버전 간에는 변경되지 않습니다. 그러나 React 서버 컴포넌트 번들러나 프레임워크를 구현하는 데 사용되는 기본 API는 시맨틱 버전SemVer을 따르지 않으며 React 19.x의 마이너Minor 버전 간에 변경될 수 있습니다.

React 서버 컴포넌트를 번들러나 프레임워크로 지원하려면, 특정 React 버전에 고정하거나 Canary 릴리즈를 사용하는 것을 권장합니다. 향후 React 서버 컴포넌트를 구현하는 데 사용되는 API를 안정화하기 위해 번들러 및 프레임워크와 계속 협력할 것입니다.

서버 없이 서버 컴포넌트 사용하기

서버 컴포넌트는 빌드 시간에 파일 시스템을 읽거나 정적 콘텐츠를 가져올 수 있으므로 웹 서버가 필요하지 않습니다. 예를 들어, 콘텐츠 관리 시스템CMS에서 정적 데이터를 읽고 싶을 때 유용합니다.

서버 컴포넌트 없이 클라이언트에서 Effect를 사용해 정적 데이터를 가져오는 일반적인 패턴은 다음과 같습니다.

```
// bundle.js
import marked from 'marked'; // 35.9K (11.2K gzipped)
import sanitizeHtml from 'sanitize-html'; // 206K (63.3K gzipped)

function Page({page}) {
  const [content, setContent] = useState('');
  // NOTE: loads *after* first page render.
  useEffect(() => {
    fetch(`/api/content/${page}`).then((data) => {
      setContent(data.content);
    });
  }, [page]);

  return <div>{sanitizeHtml(marked(content))}</div>;
}
```

```
// api.js
app.get(`/api/content/:page`, async (req, res) => {
  const page = req.params.page;
  const content = await file.readFile(`${page}.md`);
  res.send({content});
});
```

이 패턴은 사용자가 정적 콘텐츠를 렌더링하기 위해 페이지가 로드된 후 추가 75K (gzipped) 라이브러리를 다운로드하고 파싱해야 하며, 데이터를 가져오기 위한 두 번째 요청을 기다려야 합니다.

서버 컴포넌트를 사용하면 이러한 컴포넌트를 빌드 시간에 한 번만 렌더링할 수 있습니다.

```
import marked from 'marked'; // Not included in bundle
```

```
import sanitizeHtml from 'sanitize-html'; // Not included in bundle

async function Page({page}) {
  // NOTE: loads *during* render, when the app is built.
  const content = await file.readFile(`${page}.md`);

  return <div>{sanitizeHtml(marked(content))}</div>;
}
```

렌더링된 출력은 서버 측 렌더링SSR을 통해 HTML로 변환되어 CDN에 업로드될 수 있습니다. 앱이 로드될 때, 클라이언트는 기존의 `Page` 컴포넌트나 마크다운 렌더링을 위한 고비용의 라이브러리를 보지 않게 됩니다. 클라이언트는 렌더링된 출력만 보게 됩니다.

```
<div><!-- html for markdown --></div>
```

이렇게 하면 첫 페이지 로드 시 콘텐츠가 표시되고, 번들에 정적 콘텐츠를 렌더링하는 데 필요한 고비용의 라이브러리가 포함되지 않습니다.

중요합니다!

아래의 서버 컴포넌트는 비동기 함수임을 알 수 있습니다.

```
async function Page({page}) {
  //...
}
```

비동기 컴포넌트는 렌더링 중에 `await` 을 사용할 수 있게 해주는 서버 컴포넌트의 새로운 기능입니다.

자세한 내용은 아래의 서버 컴포넌트와 함께 비동기 컴포넌트 사용하기를 참조하세요.

서버와 함께 서버 컴포넌트 사용하기

서버 컴포넌트는 웹 서버에서 페이지 요청 시 실행될 수 있어, API를 구축할 필요 없이 데이터 레이어에 접근할 수 있습니다. 이들은 애플리케이션이 번들링되기 전에 렌더링되며, 데이터와 JSX를 클라이언트 컴포넌트에 Props로 전달할 수 있습니다.

서버 컴포넌트 없이 클라이언트에서 Effect를 사용해 동적 데이터를 가져오는 일반적인 패턴은 다음과 같습니다.

```
// bundle.js
function Note({id}) {
  const [note, setNote] = useState('');
  // NOTE: loads *after* first render.
  useEffect(() => {
    fetch(`/api/notes/${id}`).then(data => {
      setNote(data.note);
    });
  }, [id]);

  return (
    <div>
      <Author id={note.authorId} />
      <p>{note}</p>
    </div>
  );
}

function Author({id}) {
  const [author, setAuthor] = useState('');
  // NOTE: loads *after* Note renders.
  // Causing an expensive client-server waterfall.
  useEffect(() => {
    fetch(`/api/authors/${id}`).then(data => {
      setAuthor(data.author);
    });
  }, [id]);

  return <span>By: {author.name}</span>;
}
```

```
// api
import db from './database';
```

```
app.get(`/api/notes/:id`, async (req, res) => {
  const note = await db.notes.get(id);
  res.send({note});
});

app.get(`/api/authors/:id`, async (req, res) => {
  const author = await db.authors.get(id);
  res.send({author});
});
```

서버 컴포넌트를 사용하면 데이터를 읽고 컴포넌트 내에서 렌더링할 수 있습니다.

```
import db from './database';

async function Note({id}) {
  // NOTE: loads *during* render.
  const note = await db.notes.get(id);
  return (
    <div>
      <Author id={note.authorId} />
      <p>{note}</p>
    </div>
  );
}

async function Author({id}) {
  // NOTE: loads *after* Note,
  // but is fast if data is co-located.
  const author = await db.authors.get(id);
  return <span>By: {author.name}</span>;
}
```

번들러는 데이터를 결합하여 서버 컴포넌트를 렌더링하고 동적 클라이언트 컴포넌트와 함께 번들을 만듭니다. 선택적으로 이 번들은 서버 측 렌더링(SSR)을 통해 페이지의 초기 HTML을 생성할 수 있습니다. 페이지가 로드될 때 브라우저는 기존의 `Note` 및 `Author` 컴포넌트를 보지 않고, 렌더링된 출력만 클라이언트에 전송합니다.

```
<div>
  <span>By: The React Team</span>
  <p>React 19 is...</p>
</div>
```

서버 컴포넌트는 서버에서 다시 페칭함으로써 데이터에 액세스하고 다시 렌더링하여 동적으로 만들 수 있습니다. 이 새로운 애플리케이션 아키텍처는 서버 중심의 다중 페이지 앱(Server-Centric Multi-Page Apps)의 간단한 “request/response” 모델과 클라이언트 중심의 단일 페이지 앱(Client-Centric Single-Page Apps)의 원활한 상호작용을 결합하여 두 가지 장점을 모두 제공합니다.

서버 컴포넌트에 상호작용 추가하기

서버 컴포넌트는 브라우저로 전송되지 않으므로 `useState`와 같은 상호작용 API를 사용할 수 없습니다. 서버 컴포넌트에 상호작용을 추가하려면 `"use client"` 지시어를 사용하여 클라이언트 컴포넌트와 함께 구성할 수 있습니다.

중요합니다!

서버 컴포넌트에 대한 지시어는 없습니다.

서버 컴포넌트는 `"use server"`로 표시된다는 오해가 있지만, 서버 컴포넌트에 대한 지시어는 없습니다. `"use server"` 지시어는 서버 함수에 사용됩니다.

자세한 내용은 지시어를 참조하세요.

다음 예시에서는 `Notes` 서버 컴포넌트가 State를 사용하여 `expanded` 상태를 토글하는 `Expandable` 클라이언트 컴포넌트를 가져옵니다.

```
// Server Component
import Expandable from './Expandable';

async function Notes() {
```

```
  const notes = await db.notes.getAll();
  return (
    <div>
      {notes.map(note => (
        <Expandable key={note.id}>
          <p note={note} />
        </Expandable>
      ))}
    </div>
  )
}
```

```
// Client Component
"use client"

export default function Expandable({children}) {
  const [expanded, setExpanded] = useState(false);
  return (
    <div>
      <button
        onClick={() => setExpanded(!expanded)}
      >
        Toggle
      </button>
      {expanded && children}
    </div>
  )
}
```

이 예시는 먼저 `Notes` 를 서버 컴포넌트로 렌더링한 다음 번들러에 `Expandable` 클라이언트 컴포넌트의 번들을 생성하도록 지시합니다. 브라우저에서는 클라이언트 컴포넌트가 서버 컴포넌트의 출력을 Props로 받게 됩니다.

```
<head>
  <!-- the bundle for Client Components -->
  <script src="bundle.js" />
</head>
<body>
  <div>
```

```
    <Expandable key={1}>
      <p>this is the first note</p>
    </Expandable>
    <Expandable key={2}>
      <p>this is the second note</p>
    </Expandable>
    <!--...-->
  </div>
</body>
```

서버 컴포넌트와 함께 비동기 컴포넌트 사용하기

서버 컴포넌트는 `async` 와 `await` 을 사용하는 새로운 방법을 소개합니다. 비동기 컴포넌트에서 `await` 을 사용할 때, React는 렌더링을 지연Suspend하고 Promise가 해결될 때까지 기다린 후 렌더링을 다시 시작합니다. 이는 서버와 클라이언트의 경계를 넘어 동작하며, Suspense에 대한 스트리밍을 지원합니다.

심지어 서버에서 Promise를 생성하고 클라이언트에서 이를 기다릴 수 있습니다.

```
// Server Component
import db from './database';

async function Page({id}) {
  // Will suspend the Server Component.
  const note = await db.notes.get(id);

  // NOTE: not awaited, will start here and await on the client.
  const commentsPromise = db.comments.get(note.id);
  return (
    <div>
      {note}
      <Suspense fallback={<p>Loading Comments...</p>}>
        <Comments commentsPromise={commentsPromise} />
      </Suspense>
    </div>
  );
}
```

```
// Client Component
"use client";
import {use} from 'react';

function Comments({commentsPromise}) {
  // NOTE: this will resume the promise from the server.
  // It will suspend until the data is available.
  const comments = use(commentsPromise);
  return comments.map(comment => <p>{comment}</p>);
}
```

`note` 콘텐츠는 페이지 렌더링에 중요한 데이터이므로 서버에서 `await` 합니다. 댓글은 중요도가 낮아 페이지 아래에 표시되므로 서버에서 Promise를 시작하고 클라이언트에서 `use` API를 사용하여 기다립니다. 이는 클라이언트에서 지연되지만 `note` 콘텐츠가 렌더링되는 것을 차단하지 않습니다.

비동기 컴포넌트는 클라이언트에서 지원되지 않으므로 Promise를 `use` 로 기다립니다.

다음
서버 함수 ›

커뮤니티

행동 강령

팀 소개

문서 기여자

감사의 말

더 보기

블로그

React Native

개인 정보 보호

약관