renderToPipeableStream

`renderToPipeableStream`은 **React** 트리를 파이프 가능한 **Node.js 스트림**으로 렌더링합니다.

```
const { pipe, abort } = renderToPipeableStream(reactNode, options?)
```


중요합니다!

이 API는 Node.js 전용입니다. Deno 및 최신 엣지 런타임과 같은 Web 스트림이 있는 환경에서는 `renderToReadableStream`을 대신 사용하세요.

레퍼런스

renderToPipeableStream(reactNode, options?)

renderToPipeableStream 을 호출하여 React 트리를 HTML로 Node.js 스트림에 렌더링합니다.

```
import { renderToPipeableStream } from 'react-dom/server';

const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.setHeader('content-type', 'text/html');
    pipe(response);
  }
});
```

클라이언트에서 hydrateRoot 를 호출하여 서버에서 생성된 HTML을 상호작용할 수 있도록 만듭니다.

아래에서 더 많은 예시를 확인하세요.

매개변수

- reactNode : HTML로 렌더링하려는 React 노드. 예를 들어, <App /> 과 같은 JSX 엘리먼트입니다. 전체 문서를 나타낼 것으로 예상되므로 App 컴포넌트는 <html> 태그를 렌더링해야 합니다.
- **선택 사항** options : 스트리밍 옵션이 있는 객체입니다.
  - **선택 사항** bootstrapScriptContent : 지정하면 이 문자열이 인라인 <script> 태그에 배치됩니다.
  - **선택 사항** bootstrapScripts : 페이지에 표시할 <script> 태그에 대한 문자열 URL 배열입니다. 이를 사용하여 hydrateRoot 를 호출하는 <script> 를 포함하세요. 클라이언트에서 React를 전혀 실행하지 않으려면 생략하세요.
  - **선택 사항** bootstrapModules : bootstrapScripts 와 같지만 대신 <script type="module"> 를 출력합니다.

- **선택 사항** `identifierPrefix`: React가 `useId`에 의해 생성된 ID에 사용하는 문자열 접두사입니다. 같은 페이지에서 여러 루트를 사용할 때 충돌을 피하는 데 유용합니다. `hydrateRoot`에 전달된 것과 동일한 접두사여야 합니다.
- **선택 사항** `namespaceURI`: 스트림의 루트 네임스페이스 URI가 포함된 문자열입니다. 기본값은 일반 HTML입니다. SVG의 경우 `'http://www.w3.org/2000/svg'`를, MathML의 경우 `'http://www.w3.org/1998/Math/MathML'`를 전달합니다.
- **선택 사항** `nonce`: `script-src` Content-Security-Policy에 대한 스크립트를 허용하는 `nonce` 문자열입니다.
- **선택 사항** `onAllReady`: 셸과 모든 추가 콘텐츠를 포함하여 모든 렌더링이 완료되면 호출되는 콜백입니다. 크롤러 및 정적 생성에 `onShellReady` 대신 이 함수를 사용할 수 있습니다. 여기서 스트리밍을 시작하면 프로그레시브 로딩이 발생하지 않습니다. 스트림에는 최종 HTML이 포함됩니다.
- **선택 사항** `onError`: 복구 가능 또는 불가능에 관계없이 서버 오류가 발생할 때마다 호출되는 콜백입니다. 기본적으로 `console.error`만 호출합니다. 이 함수를 재정의하여 크래시 리포트를 기록하는 경우 `console.error`를 계속 호출해야 합니다. 셸이 출력되기 전에 상태 코드를 조정하는 데 사용할 수도 있습니다.
- **선택 사항** `onShellReady`: 초기 셸이 렌더링된 직후에 실행되는 콜백입니다. 여기서 상태 코드를 설정하고 `pipe`를 호출하여 스트리밍을 시작할 수 있습니다. React는 HTML 로딩 폴백을 콘텐츠로 대체하는 인라인 `<script>` 태그와 함께 셸 뒤에 추가 콘텐츠를 스트리밍합니다.
- **선택 사항** `onShellError`: 초기 셸을 렌더링하는 데 오류가 발생하면 호출되는 콜백입니다. 오류를 인자로 받습니다. 스트림에서 아직 바이트가 전송되지 않았고, `onShellReady`나 `onAllReady`도 호출되지 않으므로 폴백 HTML 셸을 출력 할 수 있습니다.
- **선택 사항** `progressiveChunkSize`: 청크의 바이트 수입니다. 기본 휴리스틱에 대해 자세히 알아보세요.

반환값

`renderToPipeableStream`은 두 개의 메서드가 있는 객체를 반환합니다.

- `pipe`는 HTML을 제공된 쓰기 가능한 Node.js 스트림으로 출력합니다. 스트리밍을 활성화하려면 `onShellReady`에서, 크롤러와 정적 생성을 사용하려면 `onAllReady`에서 `pipe`를 호출하세요.
- `abort`를 사용하면 서버 렌더링을 중단하고 나머지는 클라이언트에서 렌더링할 수 있습니다.

사용법

React 트리를 HTML로 Node.js 스트림에 렌더링하기

`renderToPipeableStream`을 호출하여 React 트리를 HTML로 Node.js 스트림에 렌더링합니다.

```
import { renderToPipeableStream } from 'react-dom/server';

// 경로 핸들러 문법은 백엔드 프레임워크에 따라 다릅니다.
app.use('/', (request, response) => {
  const { pipe } = renderToPipeableStream(<App />, {
    bootstrapScripts: ['/main.js'],
    onShellReady() {
      response.setHeader('content-type', 'text/html');
      pipe(response);
    }
  });
});
```

`루트 컴포넌트`와 함께 `부트스트랩 <script> 경로 목록`을 제공해야 합니다. 루트 컴포넌트는 **루트 `<html>` 태그를 포함한 전체 문서를 반환해야 합니다.**

예를 들어 다음과 같이 표시할 수 있습니다.

```
export default function App() {
  return (
    <html>
      <head>
        <meta charSet="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1" />
        <link rel="stylesheet" href="/styles.css"></link>
        <title>My app</title>
      </head>
      <body>
        <Router />
      </body>
    </html>
  );
}
```

React는 doctype과 부트스트랩 <script> 태그를 결과 HTML 스트림에 삽입합니다.

```
<!DOCTYPE html>
<html>
  <!-- ... 컴포넌트의 HTML ... -->
</html>
<script src="/main.js" async=""></script>
```

클라이언트에서 부트스트랩 스크립트는 hydrateRoot를 호출하여 전체 document를 하이드레이트해야 합니다.

```
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document, <App />);
```

이렇게 하면 서버에서 생성된 HTML에 이벤트 리스너가 첨부되어 상호작용이 가능해집니다.

자세히 살펴보기

빌드 출력에서 CSS 및 JS 에셋 경로 읽기

자세히 보기

콘텐츠가 로드되는 동안 더 많은 콘텐츠 스트리밍하기

스트리밍을 사용하면 모든 데이터가 서버에 로드되기 전에도 사용자가 콘텐츠를 볼 수 있습니다. 예를 들어 표지와 친구 및 사진이 있는 사이드바, 글 목록이 표시되는 프로필 페이지를 생각해 보세요.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Sidebar>
        <Friends />
        <Photos />
      </Sidebar>
      <Posts />
    </ProfileLayout>
  );
}
```

`<Posts />` 에 대한 데이터를 로드하는 데 시간이 걸린다고 가정해 보겠습니다. 이상적으로는 게시물을 기다리지 않고 나머지 프로필 페이지 콘텐츠를 사용자에게 표시하고 싶을 것입니다. 이렇게 하려면, `<Posts>` 를 `<Suspense>` 경계로 감싸면 됩니다.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Sidebar>
        <Friends />
        <Photos />
      </Sidebar>
      <Suspense fallback={<PostsGlimmer />}>
        <Posts />
      </Suspense>
    </ProfileLayout>
  );
}
```

이것은 `Posts` 가 데이터를 로드하기 전에 React가 HTML 스트리밍을 시작하도록 지시합니다. React는 로딩 폴백(`PostsGlimmer`)을 위한 HTML을 먼저 전송한 다음, `Posts` 가 데이터 로딩을 완료하면 나머지 HTML을 인라인 `<script>` 태그와 함께 전송하여 로딩 폴백을 해당 HTML로 대체할 것입니다. 사용자 입장에서는 페이지가 먼저 `PostsGlimmer` 로 표시되고 나중에 `Posts` 로 대체됩니다.

`<Suspense>` 경계를 더 중첩하여 보다 세분화된 로딩 시퀀스를 만들 수 있습니다.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Suspense fallback={<BigSpinner />}>
        <Sidebar>
          <Friends />
          <Photos />
        </Sidebar>
        <Suspense fallback={<PostsGlimmer />}>
          <Posts />
        </Suspense>
      </Suspense>
    </ProfileLayout>
  );
}
```

이 예시에서 React는 페이지 스트리밍을 더 일찍 시작할 수 있습니다. `ProfileLayout`과 `ProfileCover`만 `<Suspense>` 경계로 둘러싸여 있지 않기 때문에 먼저 렌더링을 완료해야 합니다. 하지만 `Sidebar`, `Friends`, `Photos`가 일부 데이터를 로드해야 하는 경우, React는 대신 `BigSpinner` 폴백을 위한 HTML을 전송합니다. 그러면 더 많은 데이터를 사용할 수 있게 되면 모든 데이터가 표시될 때까지 더 많은 콘텐츠가 계속 표시됩니다.

스트리밍은 브라우저에서 React 자체가 로드되거나 앱이 상호작용 가능해질 때까지 기다릴 필요가 없습니다. 서버의 HTML 콘텐츠는 `<script>` 태그가 로드되기 전에 점진적으로 표시됩니다.

스트리밍 HTML의 작동 방식에 대해 자세히 알아보세요.

중요합니다!

Suspense를 지원하는 데이터 소스만 Suspense 컴포넌트를 활성화합니다. 이는 다음과 같습니다.

- Relay와 Next.js 같은 Suspense가 가능한 프레임워크를 사용한 데이터 가져오기.
- `lazy`를 활용한 지연 로딩 컴포넌트.

- `use`를 사용해서 Promise 값 읽기.

Suspense는 Effect 또는 이벤트 핸들러 내부에서 데이터를 가져올 경우, **이를 감지하지 못합니다.**

`Posts` 컴포넌트에서 데이터를 불러오는 정확한 방법은 앞서 설명한 프레임워크에 따라 다릅니다. Suspense를 지원하는 프레임워크를 사용하는 경우, 데이터를 가져오는 자세한 방법은 해당 프레임워크 문서에서 찾을 수 있습니다.

독자적인 프레임워크를 사용하지 않는 Suspense 지원 데이터 가져오기는 아직 지원하지 않습니다. Suspense를 지원하는 데이터 소스를 구현하기 위한 요구 사항은 불안정하고 문서화되지 않았습니다. 데이터 소스를 Suspense와 통합하기 위한 공식 API는 React의 향후 버전에서 출시할 예정입니다.

셸에 들어갈 내용 지정하기

앱의 `<Suspense>` 경계 밖에 있는 부분을 *셸*이라고 합니다.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Suspense fallback={<BigSpinner />}>
        <Sidebar>
          <Friends />
          <Photos />
        </Sidebar>
        <Suspense fallback={<PostsGlimmer />}>
          <Posts />
        </Suspense>
      </Suspense>
    </ProfileLayout>
  );
}
```

사용자가 볼 수 있는 가장 빠른 로딩 상태를 결정합니다.

```
<ProfileLayout>
  <ProfileCover />
  <BigSpinner />
</ProfileLayout>
```

전체 앱을 루트의 `<Suspense>` 경계로 감싸면 셸에는 해당 스피너만 포함됩니다. 하지만 화면에 큰 스피너가 표시되면 조금 더 기다렸다가 실제 레이아웃을 보는 것보다 느리고 성가시게 느껴질 수 있으므로 사용자 경험이 좋지 않습니다. 그렇기 때문에 일반적으로 셸이 전체 페이지 레이아웃의 스켈레톤처럼 *최소한의 완전함*을 느낄 수 있도록 `<Suspense>` 경계를 배치하는 것이 좋습니다.

전체 셸이 렌더링되면 `onShellReady` 콜백이 실행됩니다. 보통 이때 스트리밍이 시작됩니다.

```
const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.setHeader('content-type', 'text/html');
    pipe(response);
  }
});
```

`onShellReady`가 실행될 때 중첩된 `<Suspense>` 경계에 있는 컴포넌트는 여전히 데이터를 로드하고 있을 수 있습니다.

서버에서의 충돌을 기록하기

기본적으로 서버의 모든 오류는 콘솔에 기록Logging됩니다. 이 동작을 재정의하여 충돌Crash 보고서를 기록할 수 있습니다.

```
const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.setHeader('content-type', 'text/html');
    pipe(response);
  },
```

```
  onError(error) {
    console.error(error);
    logServerCrashReport(error);
  }
});
```

사용자 정의 `onError` 구현을 제공하는 경우 위와 같이 콘솔에 오류를 기록하는 것도 잊지 마세요.

셸 내부의 오류로부터 복구하기

이 예시에서는 셸에 `ProfileLayout`, `ProfileCover`, `PostsGlimmer` 가 포함되어 있습니다.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Suspense fallback={<PostsGlimmer />}>
        <Posts />
      </Suspense>
    </ProfileLayout>
  );
}
```

이러한 컴포넌트를 렌더링하는 동안 오류가 발생하면 React는 클라이언트에 보낼 의미 있는 HTML을 갖지 못합니다. 마지막 수단으로 서버 렌더링에 의존하지 않는 폴백 HTML을 보내려면 `onShellError` 를 재정의하세요.

```
const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.setHeader('content-type', 'text/html');
    pipe(response);
  },
  onShellError(error) {
    response.statusCode = 500;
```

```
    response.setHeader('content-type', 'text/html');
    response.send('<h1>Something went wrong</h1>');
  },
  onError(error) {
    console.error(error);
    logServerCrashReport(error);
  }
});
```

셸을 생성하는 동안 오류가 발생하면 `onError` 와 `onShellError` 가 모두 실행됩니다. 오류 보고에는 `onError` 를 사용하고, 대체 HTML 문서를 보내려면 `onShellError` 를 사용합니다. 폴백 HTML이 오류 페이지일 필요는 없습니다. 대신 클라이언트에서만 앱을 렌더링하는 대체 셸을 포함할 수 있습니다.

셸 외부의 오류로부터 복구하기

이 예시에서는 `<Posts />` 컴포넌트가 `<Suspense>` 로 래핑되어 있으므로 셸의 일부가 *아닙니다*.

```
function ProfilePage() {
  return (
    <ProfileLayout>
      <ProfileCover />
      <Suspense fallback={<PostsGlimmer />}>
        <Posts />
      </Suspense>
    </ProfileLayout>
  );
}
```

`Posts` 컴포넌트 또는 그 내부 어딘가에서 오류가 발생하면 React는 이를 복구하려고 시도합니다.

1. 가장 가까운 `<Suspense>` 경계(`PostsGlimmer`)에 대한 로딩 폴백을 HTML로 방출합니다.
2. 더 이상 서버에서 `Posts` 콘텐츠를 렌더링하는 것을 “포기”합니다.

3. 자바스크립트 코드가 클라이언트에서 로드되면 React는 클라이언트에서 `Posts` 렌더링을 *재시도*합니다.

클라이언트에서 `Posts` 렌더링을 *다시 시도해도* 실패하면 React는 클라이언트에서 오류를 던집니다. 렌더링 중에 발생하는 모든 오류와 마찬가지로, 가장 가까운 부모 오류 경계에 따라 사용자에게 오류를 표시하는 방법이 결정됩니다. 실제로는 오류를 복구할 수 없다는 것이 확실해질 때까지 사용자에게 로딩 표시기가 표시된다는 의미입니다.

클라이언트에서 `Posts` 렌더링을 다시 시도하여 성공하면 서버의 로딩 폴백이 클라이언트 렌더링 출력으로 대체됩니다. 사용자는 서버 오류가 발생했다는 사실을 알 수 없습니다. 그러나 서버 `onError` 콜백 및 클라이언트 `onRecoverableError` 콜백이 실행되어 오류에 대한 알림을 받을 수 있습니다.

상태 코드 설정하기

스트리밍에는 장단점이 있습니다. 사용자가 콘텐츠를 더 빨리 볼 수 있도록 가능한 한 빨리 페이지 스트리밍을 시작하고 싶을 수 있습니다. 그러나 스트리밍을 시작하면 더 이상 응답 상태 코드를 설정할 수 없습니다.

앱을 셸(특히 `<Suspense>` 경계 바깥)과 나머지 콘텐츠로 나누면 이 문제의 일부를 이미 해결한 것입니다. 셸에 오류가 발생하면 오류 상태 코드를 설정할 수 있는 `onShellError` 콜백을 받게 됩니다. 그렇지 않으면 앱이 클라이언트에서 복구될 수 있으므로 “OK”를 보낼 수 있습니다.

```
const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.statusCode = 200;
    response.setHeader('content-type', 'text/html');
    pipe(response);
  },
  onShellError(error) {
    response.statusCode = 500;
    response.setHeader('content-type', 'text/html');
    response.send('<h1>Something went wrong</h1>');
  },
  onError(error) {
    console.error(error);
    logServerCrashReport(error);
```

```
  }
});
```

셸 *외부*(즉, `<Suspense>` 경계 안쪽)에 있는 컴포넌트가 오류를 던져도 React는 렌더링을 멈추지 않습니다. 즉, `onError` 콜백이 실행되지만 `onShellError` 대신 `onShellReady` 가 반환됩니다. 이는 위에서 설명한 것처럼 React가 클라이언트에서 해당 오류를 복구하려고 시도하기 때문입니다.

그러나 원하는 경우 오류가 발생했다는 사실을 사용하여 상태 코드를 설정할 수 있습니다.

```
let didError = false;

const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.statusCode = didError ? 500 : 200;
    response.setHeader('content-type', 'text/html');
    pipe(response);
  },
  onShellError(error) {
    response.statusCode = 500;
    response.setHeader('content-type', 'text/html');
    response.send('<h1>Something went wrong</h1>');
  },
  onError(error) {
    didError = true;
    console.error(error);
    logServerCrashReport(error);
  }
});
```

이는 초기 셸 콘텐츠를 생성하는 동안 발생한 셸 외부의 오류만 포착하므로 완전한 것은 아닙니다. 일부 콘텐츠에서 오류가 발생했는지 여부를 파악하는 것이 중요한 경우 해당 콘텐츠를 셸로 이동하면 됩니다.

다양한 오류를 서로 다른 방식으로 처리하기

자신만의 `Error` 서브 클래스를 생성하고 `instanceof` 연산자를 사용해 어떤 오류가 발생하는지 확인할 수 있습니다. 예를 들어, 사용자 정의 `NotFoundError` 를 정의하고 컴포넌트에서 이를 던질 수 있습니다. 그러면 오류 유형에 따라 `onError` , `onShellReady` , `onShellError` 콜백이 다른 작업을 수행할 수 있습니다.

```
let didError = false;
let caughtError = null;

function getStatusCode() {
  if (didError) {
    if (caughtError instanceof NotFoundError) {
      return 404;
    } else {
      return 500;
    }
  } else {
    return 200;
  }
}

const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    response.statusCode = getStatusCode();
    response.setHeader('content-type', 'text/html');
    pipe(response);
  },
  onShellError(error) {
   response.statusCode = getStatusCode();
   response.setHeader('content-type', 'text/html');
   response.send('<h1>Something went wrong</h1>');
  },
  onError(error) {
    didError = true;
    caughtError = error;
    console.error(error);
    logServerCrashReport(error);
  }
});
```

셸을 내보내고 스트리밍을 시작하면 상태 코드를 변경할 수 없다는 점에 유의하세요.

크롤러 및 정적 생성을 위해 모든 콘텐츠가 로드될 때까지 기다리기

스트리밍은 콘텐츠가 제공될 때 바로 볼 수 있기 때문에 더 나은 사용자 경험을 제공합니다.

그러나 크롤러가 페이지를 방문하거나 빌드 시점에 페이지를 생성하는 경우 모든 콘텐츠를 점진적으로 표시하는 대신 모든 콘텐츠를 먼저 로드한 다음 최종 HTML 출력을 생성하는 것이 좋을 수 있습니다.

`onAllReady` 콜백을 사용하여 모든 콘텐츠가 로드될 때까지 기다릴 수 있습니다.

```
let didError = false;
let isCrawler = // ... 봇 탐지 전략에 따라 달라집니다 ...

const { pipe } = renderToPipeableStream(<App />, {
  bootstrapScripts: ['/main.js'],
  onShellReady() {
    if (!isCrawler) {
      response.statusCode = didError ? 500 : 200;
      response.setHeader('content-type', 'text/html');
      pipe(response);
    }
  },
  onShellError(error) {
    response.statusCode = 500;
    response.setHeader('content-type', 'text/html');
    response.send('<h1>Something went wrong</h1>');
  },
  onAllReady() {
    if (isCrawler) {
      response.statusCode = didError ? 500 : 200;
      response.setHeader('content-type', 'text/html');
      pipe(response);
    }
  },
  onError(error) {
    didError = true;
    console.error(error);
```

```
    logServerCrashReport(error);
  }
});
```

일반 방문자는 점진적으로 로드되는 콘텐츠 스트림을 받게 됩니다. 크롤러는 모든 데이터가 로드된 후 최종 HTML 출력을 받게 됩니다. 그러나 이는 크롤러가 *모든* 데이터를 기다려야 한다는 것을 의미하며, 그중 일부는 로드 속도가 느리거나 오류가 발생할 수 있습니다. 앱에 따라 크롤러에도 셸을 보내도록 선택할 수 있습니다.

서버 렌더링 중단하기

시간 초과 후 서버 렌더링을 강제로 '포기'할 수 있습니다.

```
const { pipe, abort } = renderToPipeableStream(<App />, {
  // ...
});

setTimeout(() => {
  abort();
}, 10000);
```

React는 나머지 로딩 폴백을 HTML로 플러시하고 나머지는 클라이언트에서 렌더링을 시도합니다.

이전
서버 API

다음
renderToReadableStream

Meta Open Source

uwu?

React 학습하기

- 빠르게 시작하기
- 설치하기
- UI 표현하기
- 상호작용성 더하기
- State 관리하기
- 탈출구

API 참고서

- React APIs
- React DOM APIs

커뮤니티

- 행동 강령
- 팀 소개
- 문서 기여자
- 감사의 말

더 보기

- 블로그
- React Native
- 개인 정보 보호
- 약관