renderToReadableStream

`renderToReadableStream`는 Readable Web Stream을 이용해 React 트리를 그립니다.

```
const stream = await renderToReadableStream(reactNode, options?)
```


중요합니다!

이 API는 Web Stream에 의존합니다. Node.js의 경우, `renderToPipeableStream`을 대신 사용하세요.

레퍼런스

`renderToReadableStream(reactNode, options?)`

`renderToReadableStream` 을 호출하여 React 트리를 HTML로 Readable Web Stream에 렌더링합니다.

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

async function handler(request) {
  const stream = await renderToReadableStream(<App />, {
    bootstrapScripts: ['/main.js']
  });
  return new Response(stream, {
    headers: { 'content-type': 'text/html' },
  });
}
```

클라이언트에서, `hydrateRoot` 를 호출해 서버에서 생성된 HTML을 상호작용 가능하도록 만듭니다.

아래에서 더 많은 예시를 확인하세요.

매개변수

- `reactNode` : 사용자가 HTML로 렌더링하고 하고자하는 React node입니다. `<App />` 같은 JSX 요소가 그 예시입니다. reactNode 인자는 문서 전체를 표현할 수 있는 것이어야하며, 따라서 `App` 컴포넌트는 `<html>` 에 렌더링됩니다.
- **optional** `options` : 스트리밍 옵션을 지정할 수 있는 객체입니다.
  - **optional** `bootstrapScriptContent` : 지정될 경우, 해당 문자열은 `<script>` 태그에 인라인 형식으로 추가됩니다.
  - **optional** `bootstrapScripts` : 문자열 배열 형식의 단수 혹은 복수의 URL로 페이지에 함께 작성될 `<script>` 태그에 사용됩니다. `hydrateRoot` 를 호출할 때, `<script>` 태그를 포함 시키기 위해 사용합니다. 클라이언트에서 React가 실행되길 원하지 않는다면, 제외시켜주세요.

- **optional** `bootstrapModules` : `bootstrapScripts` 와 비슷합니다, 하지만 `<script type="module">` 형식으로 추가됩니다.
- **optional** `identifierPrefix` : React가 ID로서 사용할 문자열 앞머리로 `useId` 로 생성된 문자열입니다. 같은 페이지에서 여러 root를 사용할 때, 각 root간의 충돌을 방지하기 위해 유용합니다. `hydrateRoot` 에 전달한 앞머리와 반드시 동일해야합니다.
- **optional** `namespaceURI` : 문자열로 스트림을 위한 기준 namespace URI입니다. 일반 HTML에 해당하는 기본값이 설정되어있습니다. SVG를 위해 `'http://www.w3.org/2000/svg'` 를 설정하거나 MathML을 위해 `'http://www.w3.org/1998/Math/MathML'` 을 설정할 수 있습니다.
- **optional** `nonce` : `script-src` Content-Security-Policy를 허용하기 위한 `nonce` (한번만 사용되는) 문자열입니다.
- **optional** `onError` : 회복할 수 있든 있든 [없든] 상관없이, 서버에서 오류가 발생할 때마다 호출되는 콜백입니다. 기본적으로, 이 콜백은 `console.error` 만 호출합니다. 크래시 리포트를 로그하기 위해 오버라이드하거나, 상태 코드를 조정하기 위해 오버라이드할 수 있습니다.
- **optional** `progressiveChunkSize` : 청크의 바이트 수를 설정합니다. 기본 휴리스틱에 대해 더 읽어보기.
- **optional** `signal` : 서버 렌더링을 취소하고, 그 나머지를 클라이언트에 렌더링하기 위한 거절 신호(abort signal)를 설정합니다.

반환값

`renderToReadableStream` 는 Promise를 반환합니다.

- shell 렌더링에 성공했다면, 반환된 Promise는 Readable Web Stream으로 해결됩니다.
- shell 렌더링에 실패하면, 반환된 Promise는 취소됩니다. shell 렌더링에 실패시, 이것을 이용해 실패 결과를 출력하세요.

반환된 스트림은 다음과 같은 추가적인 프로퍼티를 가지고 있습니다.

- `allReady` : 모든 추가 컨텐츠와 shell의 렌더링을 포함한 모든 렌더링이 완료된 Promise의 추가 프로퍼티입니다. 크롤러와 정적 생성을 위해 `await stream.allReady` 를 응답 반환 전에 사용할 수 있습니다. 설정 시엔, 로딩 진행 상태를 받을 수 없습니다. 스트림은 최종 HTML을 포함할 것입니다.

사용 예시

Readable Web Stream을 이용해 React 트리를 HTML처럼 렌더링하기

`renderToReadableStream` 을 호출해 Readable Web Stream을 통해 React 트리를 HTML로 렌더링합니다.

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

async function handler(request) {
  const stream = await renderToReadableStream( <App /> , {
    bootstrapScripts: ['/main.js']
  });
  return new Response(stream, {
    headers: { 'content-type': 'text/html' },
  });
}
```

root 컴포넌트 와 함께, bootstrap `<script>` 경로 리스트를 제공해야합니다. 제공된 root 컴포넌트는 **최상위 `<html>` 태그를 포함한 모든 문서를 포함해서** 반환되어야 합니다.

예를 들어, 다음과 같은 형태가 되어야 합니다.

```
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 bootstrap <script> 태그들 을 결과 HTML 스트림에 주입합니다.

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

클라이언트에선, 추가된 bootstrap 스크립트는 `hydrateRoot` 를 호출해 `document` 전체를 hydrate 해야합니다.

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

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

이 과정은 서버에서 렌더링된 HTML에 이벤트 리스너들을 붙이고, HTML을 상호작용 가능하게 만듭니다.

자세히 살펴보기

빌드 결과물에서 CSS와 JS의 경로 읽어오기

자세히 보기

더 많은 컨텐츠를 스트리밍하면서 로드하기

스트리밍은 사용자가 모든 데이터를 서버로부터 로드해오기 전에 컨텐츠를 볼 수 있도록 합니다. 예를 들어, 프로필 커버사진, 친구들과 사진들이 있는 사이드바 그리고 포스트 목록을 보여주는 프로필 페이지를 생각해봅시다:

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

`<Posts />` 의 데이터를 불러오는데 약간의 시간이 필요하다고 가정해봅시다. 이 경우, 사용자가 포스트 목록을 기다리지 않고도 프로필 페이지의 나머지 컨텐츠를 볼 수 있도록 하고 싶을 것입니다. 이를 위해, `<Suspense>` 를 사용해 `Posts` 를 감싸주세요.

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

이렇게 하면 React는 `Posts` 가 모든 데이터를 불러오기 전까지, HTML 스트리밍을 시작합니다. React는 먼저 로딩 대체 컨텐츠인 `<PostsGlimmer />` 를 HTML로 보내고, `Posts` 의 데이터 로딩이 완료되면, `<PostsGlimmer />` 를 `<Posts />` 로 교체할 HTML과 인라인 `<script>` 태그를 함께 보냅니다. 사용자 입장에선, 먼저 `<PostsGlimmer />` 를 보고, 후에 `<Posts />` 를 보게 됩니다.

더 정밀한 로딩 순서를 만들기 위해 `<Suspense>` 경계를 중첩할 수 있습니다.

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

이 예시를 보았을 때, React가 더 빠르게 스트리밍을 시작하게 할 수 있습니다. `<ProfileLayout>` 과 `<ProfileCover>` 는 어떤 `<Suspense>` 경계에도 감싸져있지 않기 때문에, React는 먼저 이 두 컴포넌트를 렌더링합니다. 하지만, `Sidebar` 나 `Friends` 혹은 `Photos` 가 데이터를 불러올 필요가 있는 경우엔, `BigSpinner` 를 대체 HTML로 보냅니다. 그 후, 데이터가 더 불러와지면, 더 많은 컨텐츠가 보여지게 되고 이 과정은 모든 컨텐츠가 보여질 때까지 반복됩니다.

스트리밍은 브라우저에서 React 자체가 로드되거나 앱이 상호 작용 가능해질 때까지 기다릴 필요가 없습니다. 서버로부터 로딩되는 HTML 콘텐츠는 `<script>` 태그 중 하나가 로드되기 전까지 점진적으로 표시될 것입니다.

스트리밍 HTML이 어떻게 동작하는지 더 읽어보기.

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 이용하는 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.

Specifying what goes into the shell

앱에서 `<Suspense>` 경계 밖에 있는 부분을 *shell*이라고 합니다.

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

이는 사용자가 보는 최초의 로딩 상태를 정해줍니다.

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

만약, `<Suspense>` 경계를 root에 걸어 앱 전체를 감쌌다면, shell은 spinner만을 보여줄 것입니다. 하지만, 이는 사용자 경험에 있어서 좋지 않습니다. 큰 spinner를 보는 것은 비록 더 기다리게 될 지 언정, 실제 레이아웃이 나타나는 것보다 더 느리고 더 짜증나는 경험을 줄 수 있습니다. 이런 이유로 개발자들은 `<Suspense>` 경계를 통해 shell을 전체 페이지 레이아웃의 뼈대처럼 *최소한으로 완성된* 상태라는 느낌을 줄 수 있도록 하고 싶을 것입니다.

`renderToReadableStream`를 비동기 호출하여 모든 shell이 렌더링될 때까지 `stream`으로 위 문제를 해결합니다. 보통, `stream`을 가진 응답을 생성하고 반환함으로서 스트리밍을 시작합니다.

```
async function handler(request) {
  const stream = await renderToReadableStream(<App />, {
    bootstrapScripts: ['/main.js']
  });
  return new Response(stream, {
    headers: { 'content-type': 'text/html' },
  });
}
```

`stream`이 반환되었을 때, 중첩된 내부의 `<Suspense>` 경계의 컴포넌트는 아직 데이터를 로딩중일 수도 있습니다.

서버에서의 충돌을 기록하기

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

```
async function handler(request) {
```

```
  const stream = await renderToReadableStream(<App />, {
    bootstrapScripts: ['/main.js'],
    onError(error) {
      console.error(error);
      logServerCrashReport(error);
    }
  });
  return new Response(stream, {
    headers: { 'content-type': 'text/html' },
  });
}
```

만약 `onError` 를 직접 제공했다면, 위와 같이 콘솔에 오류를 로깅하는 것도 잊지 마세요.

---

shell 내부의 오류로부터 회복하기

이번 예시에서, shell은 `ProfileLayout`, `ProfileCover` 그리고 `PostsGlimmer` 를 포함하고 있습니다.

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

만약, 위의 컴포넌트들을 렌더링하다가 오류가 발생했다면, React는 클라이언트로 보낼 의미있는 HTML을 가지고 있지 않을 것 입니다. 이런 때를 대비해 `renderToReadableStream` 을 `try...catch` 로 감싸 서버 렌더링에 의존하지 않는 대체 HTML을 보낼 수 있도록 하세요.

```
async function handler(request) {
  try {
```

```
    const stream = await renderToReadableStream(<App />, {
      bootstrapScripts: ['/main.js'],
      onError(error) {
        console.error(error);
        logServerCrashReport(error);
      }
    });
    return new Response(stream, {
      headers: { 'content-type': 'text/html' },
    });
  } catch (error) {
    return new Response('<h1>Something went wrong</h1>', {
      status: 500,
      headers: { 'content-type': 'text/html' },
    });
  }
}
```

shell을 렌더링하면서 오류가 발생한다면, `onError` 와 `catch` 블록이 동시에 실행됩니다. `onError` 는 오류를 보고하기 위해 사용하고, `catch` 블록은 대체 HTML 문서를 보내기 위해 사용하세요. 대체 HTML은 반드시 오류 페이지일 필요는 없습니다. 대신, 클라이언트에서만 렌더링되는 대체 shell을 포함할 수 있습니다.

---

shell 외부의 오류로부터 회복하기

이번 예시에서, `<Posts />` 컴포넌트는 `<Suspense>` 에 감싸져있기 때문에, shell의 일부가 *아닙니다*.

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

`Posts` 컴포넌트 혹은 그 내부 어딘가에서 오류가 발생했을 경우, React는 오류로 부터 회복하려고 할 것입니다.

1. 가장 가까운 `<Suspense>` 경계의 로딩 대체인 (`PostsGlimmer`)를 HTML로 보냅니다.
2. 서버에서 더이상의 `Posts` 와 그 내부를 렌더링하는 것을 "포기"합니다.
3. 클라이언트에서 자바스크립트 코드가 로딩되었을 때, React는 `Posts` 를 다시 렌더링하려고 시도할 것입니다.

만약 클라이언트에서도 `Posts` 렌더링 재시도가 실패한다면, React는 클라이언트에서 오류를 던지게 됩니다. 렌더링 중에 일어난 모든 오류과 함께, 가장 가까운 부모 오류 경계로 사용자에게 어떤 오류를 보여줘야할 지를 결정하게 됩니다. 실제로는, 사용자가 오류가 복구될 수 없다는 것이 확실시 될 때까지 로딩 표시기를 보고있어야 한 다는 것을 의미합니다.

클라이언트에서 `Posts` 렌더링 재시도가 성공하면, 서버에서 온 로딩 대체 HTML이 클라이언트에서 렌더링된 결과로 교체됩니다. 사용자는 서버에서 오류가 있었는지 모를 것입니다. 하지만, 서버의 `onError` 콜백과 클라이언트의 `onRecoverableError` 콜백은 그대로 실행됩니다. 이를 통해 오류 내용을 받아서 로깅할 수 있습니다.

상태 코드 설정하기

스트리밍은 트레이드오프를 동반합니다. 사용자가 컨텐츠를 더 빨리 볼 수 있도록 페이지를 스트리밍하겠지만, 한번 스트리밍을 시작하면, 응답 상태 코드를 설정할 수 없습니다.

앱을 shell(`<Suspense>` 경계 바깥의 모든 것)과 나머지 컨텐츠들로 나누는 것으로, 이 문제는 이미 해결된 것입니다. 만약 shell에 오류가 있다면, `catch` 블록이 실행되기 때문에, 상태 코드를 설정할 수 있습니다. 혹은, 클라이언트에서 오류가 복구된 다는 것을 알고 있다면, 그냥 "OK"를 보낼 수도 있습니다.

```
async function handler(request) {
  try {
    const stream = await renderToReadableStream(<App />, {
      bootstrapScripts: ['/main.js'],
      onError(error) {
        console.error(error);
        logServerCrashReport(error);
      }
    });
```

```
    return new Response(stream, {
      status: 200,
      headers: { 'content-type': 'text/html' },
    });
  } catch (error) {
    return new Response('<h1>Something went wrong</h1>', {
      status: 500,
      headers: { 'content-type': 'text/html' },
    });
  }
}
```

만약 shell 바깥 (`<Suspense>` 경계의 안쪽)에서 오류가 발생했다면, React는 렌더링을 멈추지 않을 것입니다. 즉, `onError` 콜백은 실행되지만, `catch` 블록은 실행되지 않은 채로 코드가 계속해서 실행된다는 의미입니다. 그 이유는, 위에서 설명했던 것 처럼, React가 클라이언트에서 해당 오류를 복구하려고 하기 때문입니다.

하지만, 그래도 상태 코드를 설정하고 싶다면, 오류가 발생했다는 사실을 이용하여 상태 코드를 설정할 수 있습니다.

```
async function handler(request) {
  try {
    let didError = false;
    const stream = await renderToReadableStream(<App />, {
      bootstrapScripts: ['/main.js'],
      onError(error) {
        didError = true;
        console.error(error);
        logServerCrashReport(error);
      }
    });
    return new Response(stream, {
      status: didError ? 500 : 200,
      headers: { 'content-type': 'text/html' },
    });
  } catch (error) {
    return new Response('<h1>Something went wrong</h1>', {
      status: 500,
      headers: { 'content-type': 'text/html' },
    });
  }
```

```
}
```

이는, 초기 shell 콘텐츠를 생성하는 동안 발생한 shell 외부에서 일어난 오류만 잡을 것이므로, 완전한 방법은 아닙니다. 만약, 어떤 컨텐츠가 정말 중요해서 해당 컨텐츠에 발생한 오류를 알고 싶다면, 그것을 shell 안으로 옮겨 오류를 알아낼 수 있습니다.

각기 다른 방식으로 다른 종류의 오류를 처리하기

`Error` 서브클래스를 직접 만들 수 있고, `instanceof` 연산자를 이용해 어떤 오류가 발생했는지 구별할 수 있습니다. 예를 들어, `NotFoundError` 라는 서브클래스를 정의했고 이를 컴포넌트에서 발생시켰다고 한다면, `onError` 에서 오류를 저장하고 응답을 반환하기 전에 오류 타입에 따라 다른 동작을 할 수 있습니다.

```
async function handler(request) {
  let didError = false;
  let caughtError = null;

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

  try {
    const stream = await renderToReadableStream(<App />, {
      bootstrapScripts: ['/main.js'],
      onError(error) {
        didError = true;
        caughtError = error;
        console.error(error);
        logServerCrashReport(error);
      }
```

```
    });
    return new Response(stream, {
      status: getStatusCode(),
      headers: { 'content-type': 'text/html' },
    });
  } catch (error) {
    return new Response('<h1>Something went wrong</h1>', {
      status: getStatusCode(),
      headers: { 'content-type': 'text/html' },
    });
  }
}
```

명심해야 할 것은, shell을 전송하고 스트리밍을 시작한 후엔 상태 코드를 변경할 수 없다는 것입니다.

정적 생성과 크롤러를 위해 모든 컨텐츠가 로딩되는 것을 기다리기

스트리밍은 사용자가 컨텐츠 상호작용이 가능해지는 것을 기다리지 않고도 컨텐츠를 볼 수 있어 더 나은 사용자 경험을 제공합니다.

하지만, 크롤러가 이 페이지를 방문했을 때, 혹은 페이지를 빌드했을 때 정적으로 생성한 경우엔 컨텐츠가 점진적으로 드러나는 것이 아니라 모든 컨텐츠가 처음부터 모두 불러와진 다음 최종 HTML 출력물을 생성하는 것을 원할 것입니다.

`stream.allReady` Promise를 기다림으로써 모든 컨텐츠가 로드될 때까지 기다릴 수 있습니다.

```
async function handler(request) {
  try {
    let didError = false;
    const stream = await renderToReadableStream(<App />, {
      bootstrapScripts: ['/main.js'],
      onError(error) {
        didError = true;
        console.error(error);
        logServerCrashReport(error);
      }
    });
    let isCrawler = // ... depends on your bot detection strategy ...
```

```
    if (isCrawler) {
      await stream.allReady;
    }
    return new Response(stream, {
      status: didError ? 500 : 200,
      headers: { 'content-type': 'text/html' },
    });
  } catch (error) {
    return new Response('<h1>Something went wrong</h1>', {
      status: 500,
      headers: { 'content-type': 'text/html' },
    });
  }
}
```

일반적인 방문자라면 컨텐츠를 점진적으로 받게 될 것입니다. 크롤러라면, 모든 컨텐츠가 로드될 때까지 기다린 후에 최종 HTML을 받게 될 것입니다. 하지만, 이는 크롤러가 모든 데이터를 받을 때까지 기다려야 한다는 것으로, 그 중에 어떤 데이터가 로드되는데 느리거나 오류가 발생할 수 있는 상황까지 기다려야 한다는 것을 의미합니다. 따라서 앱의 특성에 따라 크롤러에게 shell을 보내는 것이 더 좋을 수도 있습니다.

서버 렌더링 멈추기

일정 시간이 지난 후, 서버에게 강제로 렌더링을 “포기”하라고 할 수 있습니다.

```
async function handler(request) {
  try {
    const controller = new AbortController();
    setTimeout(() => {
      controller.abort();
    }, 10000);

    const stream = await renderToReadableStream(<App />, {
      signal: controller.signal,
      bootstrapScripts: ['/main.js'],
      onError(error) {
        didError = true;
        console.error(error);
        logServerCrashReport(error);
```

```
      }
    });
    // ...
  }
}
```

React는 나머지 로딩 대체 내용을 HTML로 내보낼 것이고, 클라이언트에서 그 나머지 렌더링을 계속할 것입니다.

이전
renderToPipeableStream

다음
renderToStaticMarkup

Meta Open Source

Copyright © Meta Platforms, Inc

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