# <Activity>

`<Activity>`를 사용하면 자식 컴포넌트의 UI와 내부 상태를 숨기고 복원할 수 있습니다.

```
<Activity mode={visibility}>
  <Sidebar />
</Activity>
```



---

## 레퍼런스

### `<Activity>`

Activity를 사용하여 애플리케이션의 일부를 숨길 수 있습니다.

```
<Activity mode={isShowingSidebar ? "visible" : "hidden"}>
  <Sidebar />
</Activity>
```

Activity 경계가 숨겨지면, React는 `display: "none"` CSS 프로퍼티를 사용해 자식 컴포넌트를 시각적으로 숨깁니다. 또한 Effect를 클린업하고 활성 구독을 모두 해제합니다.

숨겨진 상태에서도 자식 컴포넌트는 새로운 props에 반응하여 리렌더링되지만, 나머지 콘텐츠보다 낮은 우선순위로 처리됩니다.

경계가 다시 보이게 되면, React는 이전 상태를 복원한 상태로 자식 컴포넌트를 표시하고 Effect를 다시 생성합니다.

이러한 방식으로 Activity는 "백그라운드 작업"을 렌더링하는 메커니즘으로 생각할 수 있습니다. 다시 표시될 가능성이 있는 콘텐츠를 완전히 삭제하는 대신, Activity를 사용하면 해당 콘텐츠의 UI와 내부 상태를 유지하고 복원할 수 있으며, 동시에 숨겨진 콘텐츠가 원치 않는 부작용을 일으키지 않도록 보장합니다.

아래에서 더 많은 예시를 확인하세요.

### Props

- `children`: 표시하거나 숨길 UI입니다.
- `mode`: `'visible'` 또는 `'hidden'` 중 하나의 문자열 값입니다. 생략하면 기본값은 `'visible'` 입니다.

### 주의 사항

- Activity가 ViewTransition 내부에서 렌더링되고, startTransition으로 인한 업데이트의 결과로 보이게 되면 ViewTransition의 `enter` 애니메이션이 활성화됩니다. 숨겨지면 `exit` 애니메이션이 활성화됩니다.
- 텍스트만 렌더링하는 Activity는 아무것도 렌더링하지 않습니다. 가시성 변경을 적용할 대응하는 DOM 엘리먼트가 없기 때문입니다. 예를 들어 `const ComponentThatJustReturnsText = () => "Hello, World!"` 인 경우, `<Activity mode="hidden"><ComponentThatJustReturnsText /></Activity>` 는 DOM에 아무런 출력도 생성하지 않습니다.

# 사용법

## 숨겨진 컴포넌트의 상태 복원하기

React에서 컴포넌트를 조건부로 표시하거나 숨기려면 일반적으로 해당 조건에 따라 마운트하거나 마운트 해제합니다.

```
{isShowingSidebar && (
  <Sidebar />
)}
```

하지만 컴포넌트를 마운트 해제하면 내부 상태가 사라지는데, 이것이 항상 원하는 동작은 아닙니다.

Activity 경계를 사용해 컴포넌트를 숨기면 React는 나중을 위해 상태를 “저장”합니다.

```
<Activity mode={isShowingSidebar ? "visible" : "hidden"}>
  <Sidebar />
</Activity>
```

이렇게 하면 컴포넌트를 숨긴 후 나중에 이전 상태 그대로 복원할 수 있습니다.

다음 예시에는 펼칠 수 있는 섹션이 있는 사이드바가 있습니다. “Overview”를 누르면 아래에 세 개의 하위 항목이 표시됩니다. 메인 앱 영역에는 사이드바를 숨기고 표시하는 버튼도 있습니다.

Overview 섹션을 펼친 다음 사이드바를 닫았다가 다시 열어보세요.

App.js Sidebar.js 새로고침 Clear 포크

```
import { useState } from 'react';
import Sidebar from './Sidebar.js';

export default function App() {
  const [isShowingSidebar, setIsShowingSidebar] = useState(true);

  return (
```

```
        <>
          {isShowingSidebar && (
            <Sidebar />
          )}
```

자세히 보기

Overview 섹션은 항상 접힌 상태로 시작합니다. `isShowingSidebar` 가 `false` 로 바뀌면서 사이드바를 마운트 해제하기 때문에 모든 내부 상태가 손실됩니다.

이것이 바로 Activity를 사용하기 완벽한 사례입니다. 시각적으로 숨기면서도 사이드바의 내부 상태를 보존할 수 있습니다.

사이드바의 조건부 렌더링을 Activity 경계로 교체해보겠습니다.

```
// Before
{isShowingSidebar && (
  <Sidebar />
)}

// After
<Activity mode={isShowingSidebar ? 'visible' : 'hidden'}>
  <Sidebar />
```

</Activity>

새로운 동작을 확인해보세요.

App.js Sidebar.js 새로고침 Clear 포크

```
import { Activity, useState } from 'react';

import Sidebar from './Sidebar.js';

export default function App() {
  const [isShowingSidebar, setIsShowingSidebar] = useState(true);

  return (
    <>
      <Activity mode={isShowingSidebar ? 'visible' : 'hidden'}>
        <Sidebar />
      </Activity>
```

자세히 보기

이제 사이드바의 내부 상태가 구현을 변경하지 않고도 복원됩니다.

## 숨겨진 컴포넌트의 DOM 복원하기

Activity 경계는 `display: none` 을 사용해 자식 컴포넌트를 숨기기 때문에, 숨겨진 상태에서도 자식의 DOM이 보존됩니다. 이는 사용자가 다시 상호작용할 가능성이 있는 UI 부분의 임시 상태를 유지하는 데 유용합니다.

이 예시에서 Contact 탭에는 사용자가 메시지를 입력할 수 있는 `<textarea>` 가 있습니다. 텍스트를 입력한 후 Home 탭으로 변경했다가 다시 Contact 탭으로 돌아오면 입력한 메시지가 사라집니다.

App.js TabButton.js Home.js Contact.js 새로고침 Clear 포크

```
export default function Contact() {
  return (
    <div>
      <p>Send me a message!</p>

      <textarea />

      <p>You can find me online here:</p>
      <ul>
        <li>admin@mysite.com</li>
        <li>+123456789</li>
      </ul>
```

`App` 에서 `Contact` 를 완전히 마운트 해제하기 때문입니다. Contact 탭이 마운트 해제되면 `<textarea>` 엘리먼트의 내부 DOM 상태가 손실됩니다.

Activity 경계를 사용해 활성 탭을 표시하고 숨기도록 전환하면 각 탭의 DOM 상태를 보존할 수 있습니다. 텍스트를 입력하고 다시 탭을 전환해보면 입력한 메시지가 더 이상 초기화되지 않는 것을 확인할 수 있습니다.

App.js TabButton.js Home.js Contact.js 새로고침 Clear 포크

```
import { Activity, useState } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
import Contact from './Contact.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('contact');

  return (
    <>
      <TabButton
        isActive={activeTab === 'home'}
```

자세히 보기

다시 한번, Activity 경계를 통해 Contact 탭의 내부 상태를 구현 변경 없이 보존할 수 있었습니다.

---

## 표시될 가능성이 있는 콘텐츠 사전 렌더링하기

지금까지 Activity를 사용해 사용자가 상호작용한 콘텐츠를 임시 상태를 삭제하지 않고 숨기는 방법을 살펴봤습니다.

하지만 Activity 경계는 사용자가 아직 처음 보지 못한 콘텐츠를 *준비*하는 데도 사용할 수 있습니다.

```
<Activity mode="hidden">
  <SlowComponent />
</Activity>
```

Activity 경계가 초기 렌더링 중에 숨겨진 상태라면, 자식 컴포넌트는 페이지에 보이지 않지만 *여전히 렌더링* 됩니다. 다만 보이는 콘텐츠보다 낮은 우선순위로 렌더링되며, Effect는 마운트되지 않습니다.

이러한 *사전 렌더링*을 통해 자식 컴포넌트가 필요한 코드나 데이터를 미리 로드할 수 있으므로, 나중에 Activity 경계가 보이게 될 때 로딩 시간이 줄어들어 더 빠르게 표시할 수 있습니다.

예시를 살펴보겠습니다.

이 데모에서 Posts 탭은 일부 데이터를 로드합니다. 탭을 누르면 데이터를 가져오는 동안 Suspense 폴백이 표시됩니다.

App.js Home.js Posts.js 새로고침 Clear 포크

```
import { useState, Suspense } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
import Posts from './Posts.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('home');

  return (
```

```
      <>
        <TabButton
```

자세히 보기

`App` 이 탭이 활성화될 때까지 `Posts` 를 마운트하지 않기 때문입니다.

`App` 을 수정하여 Activity 경계로 활성 탭을 표시하고 숨기도록 하면, 앱이 처음 로드될 때 `Posts` 가 사전 렌더링되어 보이기 전에 데이터를 가져올 수 있습니다.

이제 Posts 탭을 클릭해보세요.

App.js Home.js Posts.js 새로고침 Clear 포크

```
import { Activity, useState, Suspense } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
import Posts from './Posts.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('home');

  return (
    <>
```

```
        <TabButton
```

자세히 보기

숨겨진 Activity 경계 덕분에 `Posts` 가 더 빠른 렌더링을 준비할 수 있었습니다.

---

숨겨진 Activity 경계로 컴포넌트를 사전 렌더링하는 것은 사용자가 다음에 상호작용할 가능성이 있는 UI 부분의 로딩 시간을 줄이는 강력한 방법입니다.

### 중요합니다!

**사전 렌더링 중에는 Suspense가 가능한 데이터만 가져옵니다.** 여기에는 다음이 포함됩니다.

- Relay와 Next.js 같이 Suspense가 가능한 프레임워크를 사용한 데이터 가져오기.
- `lazy` 를 활용한 지연 로딩 컴포넌트.
- `use` 를 사용해서 캐시된 Promise 값 읽기.

Activity는 Effect 내부에서 가져온 데이터를 감지하지 **않습니다.**

위의 `Posts` 컴포넌트에서 데이터를 로드하는 정확한 방법은 프레임워크에 따라 다릅니다. Suspense가 가능한 프레임워크를 사용하는 경우, 프레임워크의 데이터 불러오기 관련 문서에서 자세한 내용을 확인할 수 있습니다.

독자적인 프레임워크를 사용하지 않는 Suspense가 가능한 데이터 가져오기 기능은 아직 지원되지 않습니다. Suspense 지원 데이터 소스를 구현하기 위한 요구 사항은 불안정하고 문서화되지 않았습니다. 데이터 소스를 Suspense와 통합하기 위한 공식 API는 향후 React 버전에서 출시될 예정입니다.

---

## 페이지 로드 중 상호작용 속도 높이기

React에는 선택적 하이드레이션이라는 내부 성능 최적화가 포함되어 있습니다. 이는 앱의 초기 HTML을 *청크 단위*로 하이드레이션하여, 페이지의 다른 컴포넌트가 코드나 데이터를 아직 로드하지 않았더라도 일부 컴포넌트를 상호작용 가능하게 만듭니다.

Suspense 경계는 자연스럽게 컴포넌트 트리를 서로 독립적인 단위로 나누기 때문에 선택적 하이드레이션에 참여합니다.

```
function Page() {
  return (
    <>
      <MessageComposer />

      <Suspense fallback="Loading chats...">
        <Chats />
      </Suspense>
    </>
  )
}
```

여기서 `MessageComposer`는 `Chats`가 마운트되어 데이터를 가져오기 시작하기 전에도 페이지의 초기 렌더링 중에 완전히 하이드레이션될 수 있습니다.

컴포넌트 트리를 개별 단위로 나누면 React가 앱의 서버 렌더링된 HTML을 청크 단위로 하이드레이션할 수 있어, 앱의 일부가 가능한 한 빠르게 상호작용 가능해집니다.

그렇다면 Suspense를 사용하지 않는 페이지는 어떻게 될까요?

다음 탭 예시를 보겠습니다.

```
function Page() {
  const [activeTab, setActiveTab] = useState('home');

  return (
    <>
      <TabButton onClick={() => setActiveTab('home')}>
        Home
      </TabButton>
      <TabButton onClick={() => setActiveTab('video')}>
        Video
      </TabButton>

      {activeTab === 'home' && (
        <Home />
      )}
      {activeTab === 'video' && (
        <Video />
      )}
    </>
  )
}
```

여기서 React는 전체 페이지를 한 번에 하이드레이션해야 합니다. `Home` 이나 `Video` 가 렌더링이 느리다면 하이드레이션 중에 탭 버튼이 반응하지 않는 것처럼 느껴질 수 있습니다.

활성 탭 주위에 Suspense를 추가하면 이 문제를 해결할 수 있습니다.

```
function Page() {
  const [activeTab, setActiveTab] = useState('home');

  return (
    <>
      <TabButton onClick={() => setActiveTab('home')}>
        Home
      </TabButton>
      <TabButton onClick={() => setActiveTab('video')}>
```

```
        Video
      </TabButton>

      <Suspense fallback={<Placeholder />}>
        {activeTab === 'home' && (
          <Home />
        )}
        {activeTab === 'video' && (
          <Video />
        )}
      </Suspense>
    </>
  )
}
```

...하지만 이렇게 하면 초기 렌더링에서 `Placeholder` 폴백이 표시되기 때문에 UI가 변경됩니다.

대신 Activity를 사용할 수 있습니다. Activity 경계는 자식을 표시하고 숨기기 때문에 이미 자연스럽게 컴포넌트 트리를 독립적인 단위로 나눕니다. 그리고 Suspense처럼 이 기능을 통해 선택적 하이드레이션에 참여할 수 있습니다.

예시를 수정하여 활성 탭 주위에 Activity 경계를 사용해보겠습니다.

```
function Page() {
  const [activeTab, setActiveTab] = useState('home');

  return (
    <>
      <TabButton onClick={() => setActiveTab('home')}>
        Home
      </TabButton>
      <TabButton onClick={() => setActiveTab('video')}>
        Video
      </TabButton>

      <Activity mode={activeTab === "home" ? "visible" : "hidden"}>
        <Home />
      </Activity>
      <Activity mode={activeTab === "video" ? "visible" : "hidden"}>
        <Video />
      </Activity>
```

```
    </>
  )
}
```

이제 초기 서버 렌더링된 HTML은 원래 버전과 동일하게 보이지만, Activity 덕분에 React는 `Home` 이나 `Video` 를 마운트하기도 전에 탭 버튼을 먼저 하이드레이션할 수 있습니다.

---

따라서 콘텐츠를 숨기고 표시하는 것 외에도, Activity 경계는 페이지의 어느 부분이 독립적으로 상호작용 가능해질 수 있는지 React에 알려줌으로써 하이드레이션 중 앱의 성능을 향상시킵니다.

페이지가 콘텐츠의 일부를 숨기지 않더라도, 하이드레이션 성능을 향상시키기 위해 항상 보이는 Activity 경계를 추가할 수 있습니다.

```
function Page() {
  return (
    <>
      <Post />

      <Activity>
        <Comments />
      </Activity>
    </>
  );
}
```

---

# 문제 해결

## 숨겨진 컴포넌트에 원치 않는 부작용이 있습니다

Activity 경계는 자식에 `display: none` 을 설정하고 Effect를 클린업하여 콘텐츠를 숨깁니다. 따라서 부작용을 적절히 클린업하는 대부분의 잘 작성된 React 컴포넌트는 이미 Activity에 의해 숨겨지는 것에 대해 견고합니다.

하지만 숨겨진 컴포넌트가 마운트 해제된 컴포넌트와 다르게 동작하는 *몇 가지* 상황이 있습니다. 특히 숨겨진 컴포넌트의 DOM은 제거되지 않기 때문에, 해당 DOM의 부작용은 컴포넌트가 숨겨

진 후에도 지속됩니다.

예를 들어 `<video>` 태그를 생각해보세요. 일반적으로 클린업이 필요하지 않습니다. 비디오를 재생 중이더라도 태그를 마운트 해제하면 브라우저에서 비디오와 오디오 재생이 중지되기 때문입니다. 비디오를 재생한 후 이 데모에서 Home을 눌러보세요.

App.js Home.js Video.js 새로고침 Clear 포크

```
import { useState } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
import Video from './Video.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('video');

  return (
    <>
      <TabButton
        isActive={activeTab === 'home'}
```

자세히 보기

비디오가 예상대로 재생을 멈춥니다.

이제 사용자가 마지막으로 시청한 타임코드를 보존하여 비디오 탭으로 돌아왔을 때 처음부터 다시 시작하지 않도록 하고 싶다고 가정해봅시다.

이것은 Activity를 사용하기 완벽한 사례입니다!

App 을 수정하여 비활성 탭을 마운트 해제하는 대신 숨겨진 Activity 경계로 숨기고, 이번에는 데모가 어떻게 동작하는지 확인해보세요.

App.js Home.js Video.js

새로고침 Clear 포크

```
import { Activity, useState } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
import Video from './Video.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('video');

  return (
    <>
      <TabButton
        isActive={activeTab === 'home'}
```

자세히 보기

이런! 비디오가 숨겨진 후에도 비디오와 오디오가 계속 재생됩니다. 탭의 `<video>` 엘리먼트가 여전히 DOM에 있기 때문입니다.

이를 해결하기 위해 비디오를 일시정지하는 클린업 함수가 있는 Effect를 추가할 수 있습니다.

```
export default function VideoTab() {
  const ref = useRef();

  useLayoutEffect(() => {
    const videoRef = ref.current;

    return () => {
      videoRef.pause()
    }
  }, []);

  return (
    <video
      ref={ref}
      controls
      playsInline
      src="..."
    />

  );
}
```

개념적으로 클린업 코드가 컴포넌트의 UI가 시각적으로 숨겨지는 것과 연결되어 있기 때문에 `useEffect` 대신 `useLayoutEffect` 를 호출합니다. 일반 effect를 사용하면 (예를 들어) 다시 suspend되는 Suspense 경계나 View Transition에 의해 코드가 지연될 수 있습니다.

새로운 동작을 확인해보세요. 비디오를 재생하고 Home 탭으로 전환한 다음 다시 Video 탭으로 돌아와보세요.

App.js Home.js Video.js

새로고침 Clear 포크

```
import { Activity, useState } from 'react';
import TabButton from './TabButton.js';
import Home from './Home.js';
```

```
import Video from './Video.js';

export default function App() {
  const [activeTab, setActiveTab] = useState('video');

  return (
    <>
      <TabButton
```

자세히 보기

완벽하게 작동합니다! 클린업 함수는 Activity 경계에 의해 숨겨질 때마다 비디오 재생이 중지되도록 보장하며, 더 좋은 점은 `<video>` 태그가 제거되지 않기 때문에 타임코드가 보존되고, 사용자가 시청을 계속하기 위해 다시 전환할 때 비디오를 초기화하거나 다시 다운로드할 필요가 없다는 것입니다.

이는 Activity를 사용하여 숨겨지지만 사용자가 곧 다시 상호작용할 가능성이 있는 UI 부분의 임시 DOM 상태를 보존하는 좋은 예시입니다.

---

예시에서 보듯이 `<video>` 와 같은 특정 태그의 경우 마운트 해제와 숨기기가 다른 동작을 보입니다. 컴포넌트가 부작용이 있는 DOM을 렌더링하고, Activity 경계가 이를 숨길 때 해당 부작용을 방지하고 싶다면 클린업을 위한 return 함수가 있는 Effect를 추가하세요.

가장 흔한 경우는 다음 태그일 것입니다.

- `<video>`
- `<audio>`
- `<iframe>`

하지만 일반적으로 대부분의 React 컴포넌트는 이미 Activity 경계에 의해 숨겨지는 것에 대해 견고해야 합니다. 개념적으로 "숨겨진" Activity는 마운트 해제된 것으로 생각해야 합니다.

적절한 클린업이 없는 다른 Effect를 미리 발견하려면 `<StrictMode>` 사용을 권장합니다. 이는 Activity 경계뿐만 아니라 React의 다른 많은 동작에도 중요합니다.

---

## 숨겨진 컴포넌트의 Effect가 실행되지 않습니다

`<Activity>`가 "hidden" 상태일 때 모든 자식의 Effect가 클린업됩니다. 개념적으로 자식은 마운트 해제되지만, React는 나중을 위해 상태를 저장합니다. 이는 Activity의 기능입니다. 숨겨진 UI 부분에 대한 구독이 활성화되지 않아 숨겨진 콘텐츠에 필요한 작업량이 줄어들기 때문입니다.

컴포넌트의 부작용을 클린업하기 위해 Effect가 마운트되는 것에 의존하고 있다면, 대신 반환된 클린업 함수에서 작업을 수행하도록 Effect를 리팩터링하세요.

문제가 있는 Effect를 미리 찾으려면 `<StrictMode>` 추가를 권장합니다. 이는 예상치 못한 부작용을 포착하기 위해 Activity 마운트 해제와 마운트를 미리 수행합니다.


이전
<Suspense>

다음
<ViewTransition>


---

## 커뮤니티



## 더 보기