**리액트 심화**

**리액트 고급 기능**

**컴포넌트 최적화: Memoization 이해하기**

동일한 계산을 반복해야 할 때, 이전에 계산한 값을 저장해 두었다가 다시 사용함으로 계산 속도를 높임

**주요 특징**

반복되는 계산 결과를 캐싱

입력이 같다면 항상 같은 출력을 반환하는 순수 함수에 특히 유용

시간복잡도와 공간복잡도 사이의 트레이드오프를 최적화

**장점**

반복적인 고비용 계산을 피할 수 있음

애플리케이션의 전반적인 성능을 향상

**단점**

메모리 사용량이 증가

과도한 사용은 오히려 성능 저하

**컴포넌트 최적화: Memoization 이해하기**

**React에서 컴포넌트가 리렌더링이 되는 조건**

컴포넌트의 state가 변경

컴포넌트의 props가 변경

**React에서의 메모이제이션**

불필요한 리렌더링 방지 성능 최적화

주로 사용되는 메모이제이션 React.memo, useMemo, useCallback

**컴포넌트 최적화: React.memo**

* ﻿﻿React.memo는 고차 컴포넌트(Higher Order Component)로, 컴포넌트의 props가 변경되지 않았을 때 리렌더링을 방지합니다.
* ﻿﻿React.memo는 props의 얕은 비교(shallow comparison)를 수행합니다.

﻿﻿props가 복잡한 객체를 가지고 있을 경우, 커스텀 비교 함수를 제공할 수 있습니다.

**컴포넌트 최적화: useMemo**

* ﻿﻿이 Hook은 계산 비용이 높은 함수의 결과값을 메모이제이션하여 불필요한 재계산을 방지해 렌더링 성능을 최적화합니다.
* ﻿﻿의존성 배열의 값이 변경될 때만 재계산됩니다.

﻿﻿주로 복잡한 계산을 하거나 큰 데이터를 처리하는 함수에 사용합니다.

**컴포넌트 최적화: useCallback**

* ﻿﻿이 Hook은 함수 자체를 메모이제이션하여 자식 컴포넌트의 불필요한 리렌더링 방지해 렌더링 성능을 최적화합니다.
* ﻿﻿의존성 배열의 값이 변경될 때만 재계산됩니다.

﻿﻿주로 자식 컴포넌트에 전달되는 콜백 함수에 사용합니다.

**컴포넌트 최적화 주의사항**

* ﻿﻿**과도한 사용 피하기**
* ﻿﻿모든 값이나 함수를 메모이제이션할 필요가 없습니다.
* ﻿﻿실제 성능 향상이 필요한 경우에만 사용하는 것이 좋습니다.
* ﻿﻿**의존성 배열 관리**
* ﻿﻿필요한 모든 값을 의존성 배열에 포함해야 합니다.
* ﻿﻿불필요한 값이 의존성 배열에 포함되지 않도록 합니다.
* ﻿﻿**얕은 비교 이해**
* ﻿﻿React는 의존성 배열에 대해 얕은 비교 수행합니다. 즉, 객체의 참조가 바뀌면 값이 바뀐 것으로 판단합니다.
* ﻿﻿따라서 객체나 배열의 참조가 변경되면 메모이제이션이 무효화됩니다.
* ﻿﻿**성능 측정**

﻿﻿React DevTools의 Profiler 활용하여 실제 성능이 향상했는지 확인합니다.

**얕은 비교**

* ﻿﻿useMemo와 useCallback은 의존성 배열의 각 항목에 대해 얕은 비교를 수행합니다.
* ﻿﻿이는 객체나 배열의 경우 참조 동등성만을 확인한다는 의미입니다.

﻿﻿따라서 객체나 배열의 내용이 같더라도 새로운 참조가 생성되면 React는 이를 변경된 것으로 간주하고 메모이제이션된 값이나 함수를 재생성합니다.

* 이 경우, obj 의 내용이 변경되지 않더라도 컴포넌트가 리렌더링될 때마다 새로운 객체 참조가 생성되어 memoizedValue 가 재계산됩니다.
* ﻿﻿**이를 방지하기 위해서는 다음과 같은 방법을 사용할 수 있습니다.**
* ﻿﻿객체나 배열 대신 원시 값을 의존성으로 사용
* ﻿﻿객체나 배열을 컴포넌트 외부에서 생성하거나 useMemo를 사용하여 메모이제이션

﻿﻿필요한 경우 커스텀 비교 함수를 사용하는 방법을 고려

**코드 분할과 지연 로딩**

* ﻿﻿코드 분할은 애플리케이션의 한 페이지를 더 작은 조각(청크)으로 나누고, 필요한 시점에 로드하는 기술입니다.
* ﻿﻿이 기술은 초기 로딩 시간을 줄이고 전체적인 애플리케이션 성능을 향상시키는 데 중요한 역할을 합니다.
* ﻿﻿React에서는 Redct.lazy()와 Suspense를 통해 코드 분할과 지연 로딩(Lazy Loading)을 구현할 수 있습니다.

﻿﻿실제 성능 향상을 측정하고, 사용자 경험을 고려하여 적절히 적용하는 것이 중요합니다.

**React.lazyO 를 활용한 동적 임포트**

* ﻿﻿React.lazy()는 동적으로 컴포넌트를 임포트할 수 있게 해주는 함수입니다.
* ﻿﻿이 함수를 사용하면 컴포넌트가 실제로 렌더링되어야 할 때 로드되도록 할 수 있습니다.
* ﻿﻿동적 임포트를 활용하면 웹 페이지의 초기 로딩 시간을 크게 줄이고, 필요한 컴포넌트만 그때그때 로드하여 전체적인 성능을 개선할 수 있습니다.

﻿﻿React.lazy로 로드되는 컴포넌트는 반드시 default export를 사용해야 합니다.

**Suspense로 로딩 처리하기**

* ﻿﻿Suspense는 아직 렌더링이 준비되지 않은 컴포넌트가 있을 때 렌더링할 컴포넌트를 설정할 수 있게 해주는 컴포넌트입니다.

﻿﻿React.lazy()와 함께 사용되어 동적으로 로드되는 컴포넌트의 로딩 상태를 처리할 수 있습니다.

**Error Boundaries**

* ﻿﻿**개요**
* ﻿﻿컴포넌트 트리의 하위에서 발생하는 자바스크립트 에러를 캐치하고, 에러 발생 시 폴백 UI를 표시하며, 전체 애플리케이션이 완전히 중단되는 것을 방지합니다.
* ﻿﻿**구현 방법**
* ﻿﻿Error Boundary는 다음 생명주기 메소드 중 하나 이상을 정의한 클래스 컴포넌트입니다
* ﻿﻿static getDerivedStateFromError()

﻿﻿componentDidCatch()

**Error Boundaries**

* **﻿﻿특징**
* ﻿﻿Error Boundaries는 트리 내에서 깊숙이 중첩된 에러도 캐치합니다.
* ﻿﻿여러 Error Boundaries를 중첩하여 사용할 수 있어, 세분화된 에러 처리가 가능합니다.
* ﻿﻿클래스 컴포넌트로 구현하지만, 함수형 컴포넌트에서도 Error Boundary를 사용할 수 있습니다.
* ﻿﻿**제한사항**
* ﻿﻿Error Boundaries는 다음과 같은 에러는 캐치하지 않습니다.
* ﻿﻿이벤트 핸들러 내부의 에러
* ﻿﻿비동기 코드 (예: setTimeout이나 requestAnimationFrame 콜백)
* ﻿﻿서버 사이드 렌더링

﻿﻿Error Boundary 자체에서 발생하는 에러

**Error Boundaries**

* ﻿﻿Suspense와 함께 사용하여, 비동기 컴포넌트를 불러오지 못했을 때 에러를 캐치할 수 있습니다
* ﻿﻿컴포넌트가 로딩되는 동안 Loading... 메시지가 표시됩니다.
* ﻿﻿컴포넌트 로딩이 성공하면 정상적으로 렌더링됩니다.

﻿﻿로딩 중 또는 렌더링 중 에러가 발생하면 Error Boundary가 이를 캐치하고 에러 메시지를 표시합니다

**라우팅과 네비게이션**

**Single Page Application (SPA)**

SPA(Single Page Application)는 하나의 HTML 페이지로 구성된 웹 애플리케이션을 말하며, 필요한 컨텐츠를 동적으로 변경하여 사용자에게 제공하는 방식입니다.

* ﻿﻿**SPA의 특징**
* ﻿﻿초기 로딩 후 페이지 전체를 다시 로드하지 않음
* ﻿﻿자바스크립트를 사용하여 동적으로 컨텐츠를 변경
* ﻿﻿URL에 따라 적절한 컴포넌트를 렌더링
* ﻿빠른 사용자 경험 제공

﻿﻿서버 부하 감소

**React Router**

React 애플리케이션에서 라우팅은 사용자가 다양한 페이지를 탐색할 수 있게 해주는 중요한 기능입니다. React Router는 이를 위한 가장 인기 있는 라이브러리입니다.

**React Router의 라우터 종류**

1. BrowserRouter

* ﻿﻿HTML5 History API를 사용합니다.
* ﻿﻿깔끔한 URL을 제공합니다. (예: /users/123)
* ﻿﻿서버 설정이 필요할 수 있습니다. (모든 요청을 index.html로 리다이렉트)

﻿﻿대부분의 현대적인 웹 애플리케이션에 권장됩니다

2. HashRouter

* ﻿﻿URL의 해시 부분을 사용합니다. (예: /#/users/123)
* ﻿﻿서버 설정이 필요 없어 정적 파일 호스팅에 적합합니다.
* ﻿﻿브라우저는 URL에서 해시(#) 이후의 부분을 서버로 전송하지 않습니다.
* ﻿﻿서버는 항상 해시 이전의 기본 URL만 받게 됩니다.

﻿﻿검색 엔진이 해시 이후의 콘텐츠를 인식하지 못할 수 있어, SEO에 불리할 수 있습니다.

3. MemoryRouter

* ﻿﻿애플리케이션 메모리에 히스토리가 저장되어 URL이 변경되지 않습니다.
* ﻿﻿내부적으로 자바스크립트 배열을 사용하여 히스토리를 관리합니다.

﻿﻿테스팅이나 비 브라우저 환경(React Native 등)에 유용합니다.

스크린샷, 텍스트, 웹사이트, 웹 페이지이(가) 표시된 사진

자동 생성된 설명

**중첩 라우팅**

* ﻿﻿중첩 라우팅은 부모 라우트 내에 자식 라우트를 정의하는 것을 말합니다.

﻿﻿이 때, Outlet 컴포넌트는 부모 라우트 컴포넌트에서 자식 라우트 컴포넌트가 렌더링될 위치를 지정합니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

CSR 과 SSR

**CSR vs SSR**

* ﻿﻿**SSR (Server-Side Rendering)**
* ﻿﻿서버에서 HTML을 생성하여 클라이언트에 전송합니다.
* ﻿﻿초기 로딩 속도가 빠릅니다.
* ﻿SEO(검색 엔진 최적화)에 유리합니다.
* ﻿﻿서버 부하가 증가할 가능성이 있습니다.
* ﻿﻿**CSR (Client-Side Rendering)**
* ﻿﻿클라이언트에서 JavaScript로 HTML을 생성합니다.
* ﻿﻿초기 로딩 속도가 느립니다.
* ﻿﻿사용자 상호작용에 유리합니다.

﻿서버 부하가 감소합니다.

**SSR 성능 최적화**

* **﻿﻿코드 스플리팅**
* ﻿﻿필요 시점에 필요한 코드만 로드하여 초기 로딩 시간을 단축합니다.
* ﻿﻿일반적으로 페이지 별로 코드 스플리팅을 구현합니다.
* ﻿﻿**캐싱 전략**
* ﻿﻿서버 응답을 캐싱하여 동일한 요청에 대해 빠르게 응답할 수 있도록 합니다.
* ﻿﻿Cache-Control 헤더를 설정하여 브라우저와 CDN에서 캐싱을 최적화합니다. stale-while-revalidate 등의 캐싱 전략을 통해 캐싱된 데이터를 사용하면서 백그라운드에서 새로운 데이터를 가져옵니다.
* ﻿﻿**리소스 최적화**
* ﻿﻿리소스 크기를 최적화하고, 리소스가 필요할 때 로드되도록 합니다.
* ﻿﻿WebP와 같은 현대적인 이미지 포맷을 사용하여 파일 크기를 줄입니다.
* ﻿﻿**최소화 및 번들링**
* ﻿﻿Terser 플러그인을 사용하여 JavaScript 코드를 최소화합니다.

﻿﻿CSS와 JavaScript 파일을 별도로 분리하여 필요한 리소스만 로드합니다.

**서버 통신 고급 라이브러리**

**데이터 페칭 라이브러리 (React Query, SWR)**

* **﻿﻿React Query**
* ﻿﻿서버 데이터를 쉽게 관리하고 캐싱하는 라이브러리입니다.
* ﻿﻿요청의 상태 관리, 캐시 관리, 재시도 처리 등을 자동으로 처리합니다.
* ﻿﻿GraphOL, REST API 등 모든 종류의 백엔드와 통합하여 사용할 수 있습니다.
* ﻿﻿**SWR (Stale-While-Revalidate)**
* ﻿﻿Next js 팀에서 개발한 데이터 페칭 라이브러리입니다.
* ﻿﻿캐시된 데이터를 보여주고, 백그라운드에서 데이터를 다시 검증하며 업데이트합니다.

﻿﻿데이터 요청을 최적화하여 성능을 향상시킵니다 텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**실시간 데이터 처리 (WebSocket)**

* ﻿﻿실시간 양방향 데이터 통신을 가능하게 하는 컴퓨터 통신 프로토콜입니다.
* ﻿﻿HTTP와 달리 지속적인 연결을 통해 데이터를 주고받을 수 있습니다.
* ﻿﻿WebSocket은 클라이언트와 서버 간의 지속적인 연결을 유지합니다.
* ﻿﻿이로 인해 서버 자원을 지속적으로 사용하므로, 연결 관리와 리소스 사용에 주의해야 합니다.
* ﻿﻿WebSocket 동작 과정
* ﻿﻿클라이언트가 서버에 WebSocket 연결을 요청합니다.
* ﻿﻿서버는 요청을 수락하고 연결을 설정합니다.
* ﻿﻿연결이 설정되면, 양방향으로 데이터를 주고받을 수 있습니다.
* ﻿﻿클라이언트나 서버가 필요할 때 연결을 종료할 수 있습니다.
* ﻿﻿WebSocket 사용 예시
* ﻿﻿실시간 채팅: 사용자 간 메시지를 실시간으로 전송하고 받을 수 있습니다.

﻿﻿실시간 경고 및 알림: 서버 측 데이터의 변화를 실시간으로 클라이언트에 알릴 수 있습니다.