성능 측정과 개선

1. 성능 측정 개요

❖프론트엔드 애플리케이션에서 성능이 중요한 이유

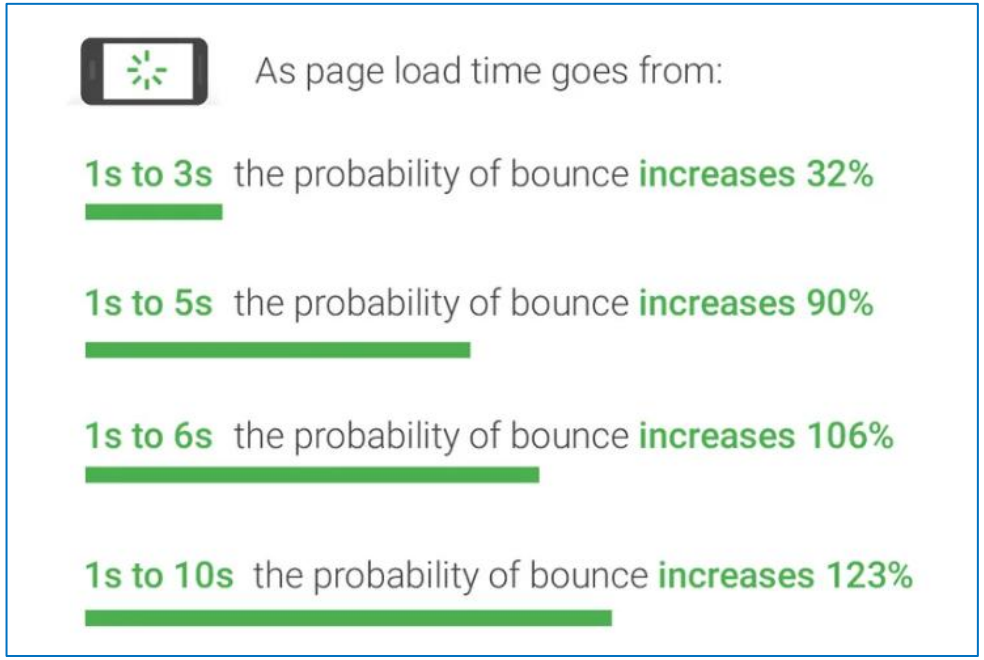
- 사용자가 떠나지 않도록 함
  - 웹 애플리케이션의 성능이 나쁜 경우 사용자는 현재의 웹 애플리케이션 화면에서 이탈함
- 웹사이트의 전환율을 높임
  - 웹 애플리케이션의 성능이 좋아질수록 웹사이트 방문 후 상품 구매, 유료 서비스 구독 등의 단계까지 진행하는 방문자의 비율이 증가함
  - 전환율 = 전환수 / 방문수
- UX 증대
  - 웹 애플리케이션의 성능은 UX에서 대단히 중요한 요소
  - 느린 성능은 사용자에게 나쁜 경험을 각인시키고, 사이트의 재방문율을 낮추거나 사이트를 이탈할 가능성을 높임
- 검색엔진 최적화
  - 구글에서는 우수한 성능을 제공하는 페이지에 대해 검색의 우선순위를 높여줌
  - 성능은 검색 노출 빈도 증가와 관련성을 가짐

❖ 웹애플리케이션에 대한 성능을 개선하는 것은 대단히 중요함

1. 성능 측정 개요

❖해외 조사 사례

- 스웨덴의 웹사이트 모니터링 기관 Pingdom에서의 조사결과
  - https://nownews.seoul.co.kr/news/newsView.php?id=20180123601013
  - 웹페이지 로딩시간별 이탈률
    - 2초 : 9%
    - 4초 : 23%
    - 5초 : 38%
  - 2초 이내로 억제해야 함.
- 2017년 구글에서의 조사 결과
  - https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/



1. 성능 측정 개요

❖성능 측정

- 웹 애플리케이션의 성능을 개선하려면 유의미한 성능 측정 지표가 필요하고
- 수치화된 형태로 성능 개선의 정보를 확인할 수 있어야 함

❖Web Vitals

- 웹 애플리케이션에서 개선된 UX를 제공하기 위해 필요한 필수적인 성능, 품질 지표에 대한 지침을 제공하는 Google의 측정 기준
- Core Web Vitals
  - Web Vitals의 하위 구성요소이면서 Web Vitals 중 가장 중요한 핵심 요소
  - LCP
  - FID
  - INP
  - CLS

2. 성능 측정 지표

❖LCP : Largest Contentful Paint

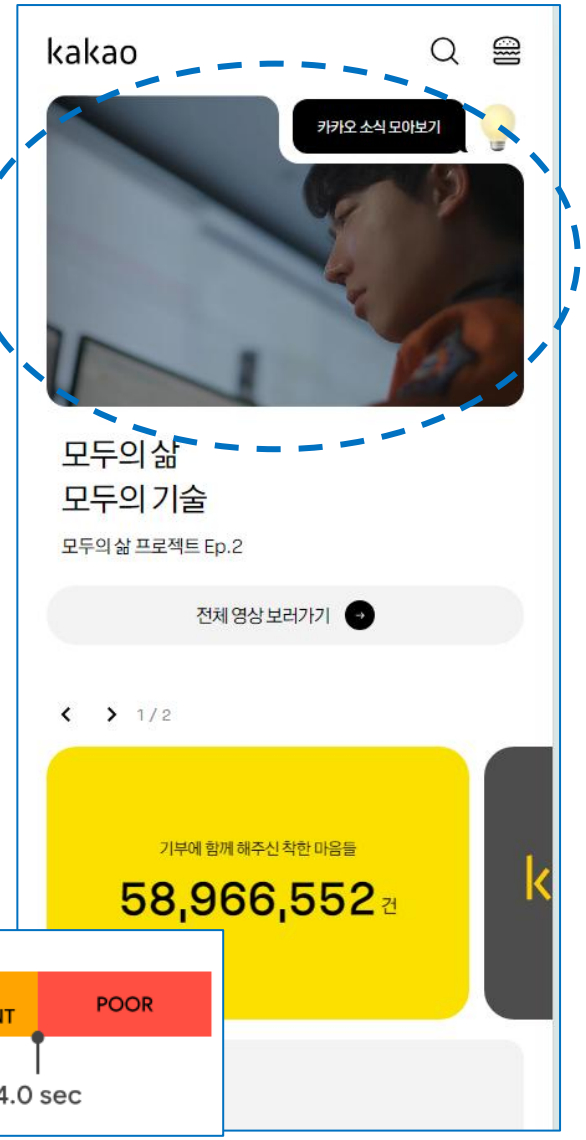
- 사용자가 웹 애플리케이션 화면을 로드할 때 페이지내에서 가장 큰 요소(텍스트, 이미지, 비디오)가 브라우저 화면에 렌더링되는 시간을 측정하는 지표
- 메인 컨텐츠가 빨리 로딩되는 시간을 측정하는 것

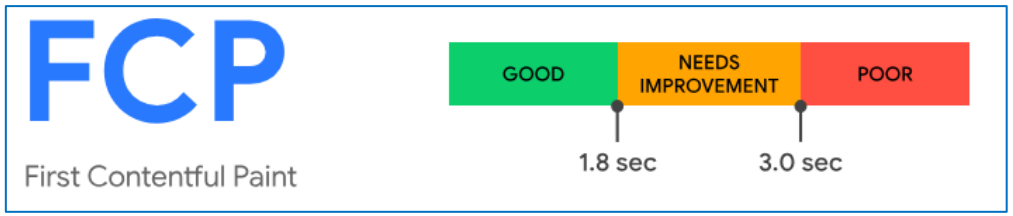
❖FCP : First Contentful Paint

- 첫번째 컨텐트가 렌더링되기까지의 시간 측정
- 첫번째 컨텐트가 로드되더라도 의미없는 컨텐트일 가능성 있음

❖LCP가 객관적인 지표

- 2.5초 이내라면 Good





2. 성능 측정 지표

❖FID : First Input Delay

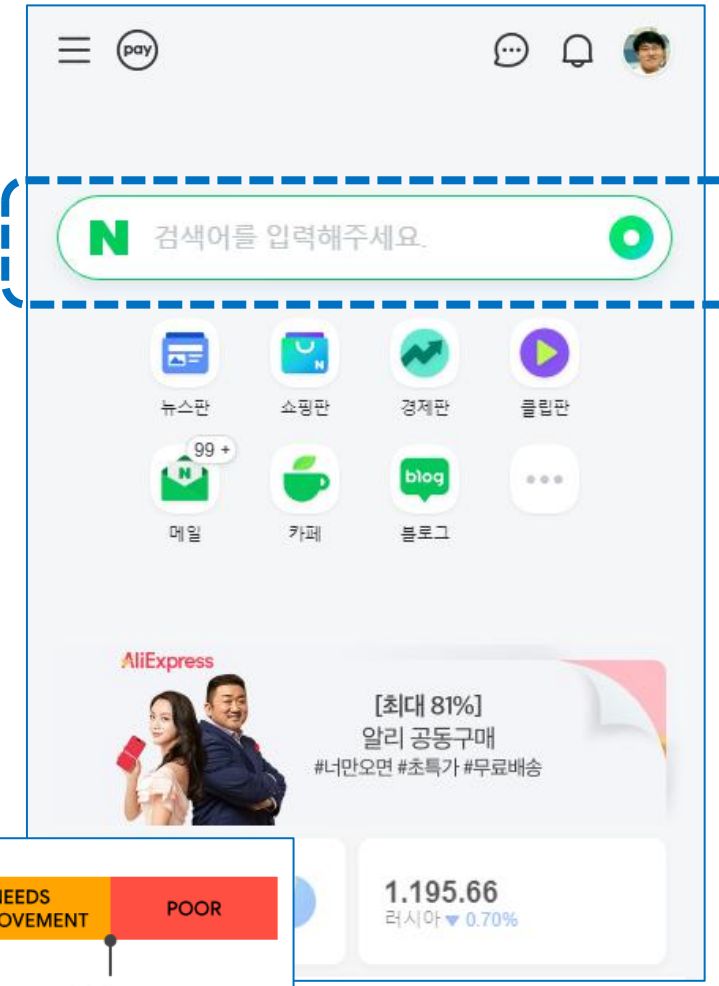
- 사용자가 웹화면에서 클릭, 타이핑 등의 이벤트를 일으킨 후 실제로 브라우저에서 이벤트 처리가 실행될 때까지의 시간

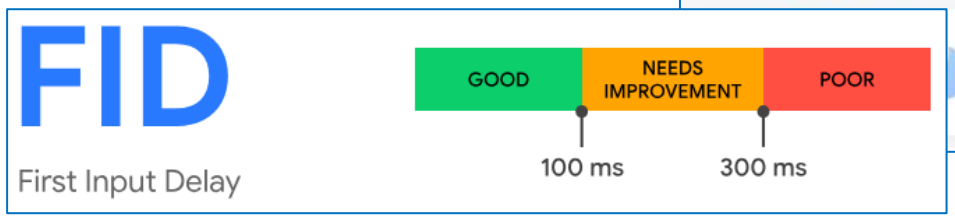
❖INP : Interaction to Next Paint

- Event Timing API 데이터를 사용하여 페이지의 응답성을 평가
- 페이지 실행 동안 클릭, 탭, 키보드 입력에 대한 지연 시간을 평하고 가장 긴 시간을 리포팅함

❖FID와 INP의 차이점

- FID는 첫번째 인터랙션만을 측정
- **INP는 페이지에서의 모든 인터랙션을 측정함

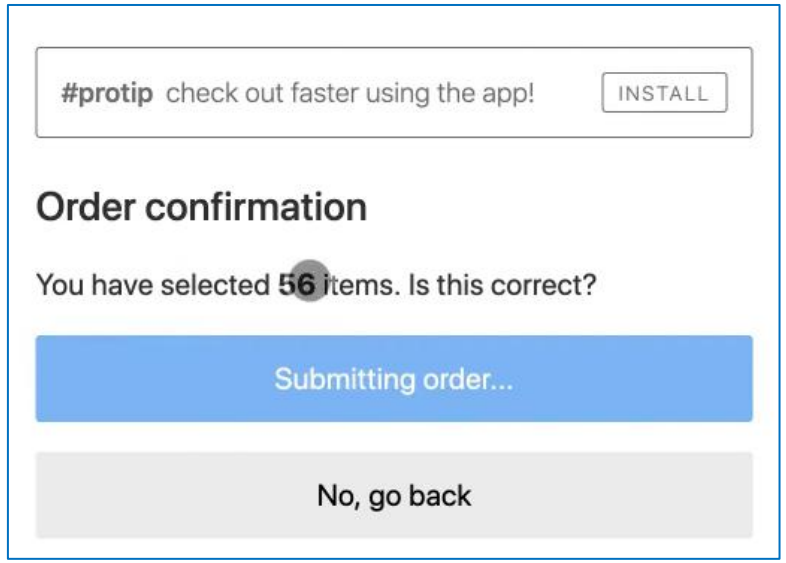




2. 성능 측정 지표

❖CLS : Cumulative Layout Shift

- 웹페이지에서 요소의 이동이나 크기 조정 등의 예상치 못한 레이아웃 변경이 일어나는 빈도와 양을 측정하는 지표
  - 시각적 안정성 및 상호작용 안정성 측정
- 시나리오
  - 화면이 로드되고 있는 도중 핵심 컨텐트는 로드된 상태지만 나머지 컨텐트는 로드되지 않은 상태에서
  - 사용자는 브라우저 화면에서 클릭과 같은 인터랙션을 시도함
  - 인터랙션을 수행하는 도중 레이아웃이 바뀌면?
  - 원하지 않는 요소를 클릭할 가능성이 존재함.
- CLS는 레이아웃의 변화하는 정도를 측정하는 것



❖구글 문서 참조

- https://web.dev/articles/cls?hl=ko

2. 성능 측정 지표

❖TBT : Total Blocking Time

- FCP(First Contentful Paint)이후 사용자 입력을 막을 만큼 기본 스레드가 충분히 오랫동안 차단된 총 시간을 측정
  - 기본 스레드에서 50ms 이상 실행되는 것이 있으면 기본 스레드가 차단된 것으로 간주함
  - 차단된 시간의 총합을 측정함
- TBT가 낮으면 페이지 사용성이 높아짐
- 좋은 TBT 스코어
  - 평균적인 모바일 하드웨어에서 300ms초 미만일 때 Good

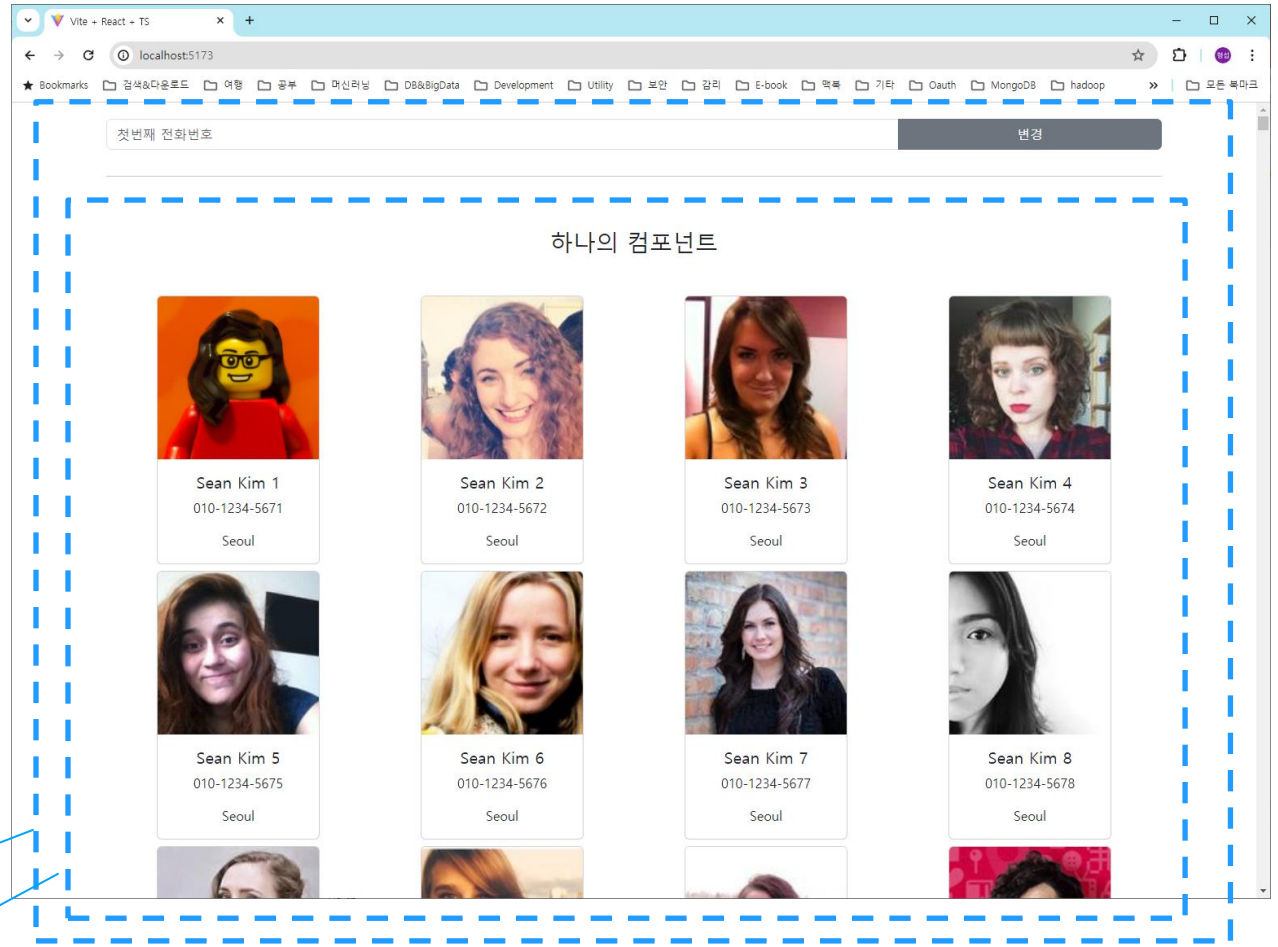
3. 다양한 성능 측정 도구

- ❖react devtools profiler
- ❖lighthouse
- ❖web-vitals library
- ❖chrome devtools
  - performance
  - performance insight
- ❖https://pagespeed.web.dev/

4. 성능 측정/개선을 위한 샘플 예제

❖강사로부터 제공받은 예제를 사용

- photo-card-list
- 샘플데이터 임의 생성
  - 1000건의 연락처 데이터 임의 생성
  - 101개의 사람 이미지를 랜덤하게 지정
  - 첫번째 연락처는 noimage.jpg 지정
  - 1~100.jpg는 128x128 크기의 이미지
  - noimage.jpg는 500x500 크기의 이미지
- 첫번째 연락처의 전화번호를 변경하는 기능
  - App – ContactList 두개의 컴포넌트로 구성



App 컴포넌트

ContactList 컴포넌트

5. React Developer Tools의 Profiler

❖리액트 개발자 도구의 프로파일러

- 리액트 앱의 각 컴포넌트별 성능을 측정하여 어느 컴포넌트에 대한 개선이 필요한지를 결정할 수 있는 정보를 제공하는 리액트 개발자 도구의 기능

❖프로파일러의 작동 방식

- 리액트 앱에서의 Commit에 의한 성능 정보를 수집하고 그룹화함
- 리액트의 2단계(2 phase) 실행
  - render 단계 : 리액트 앱에서 render()가 호출되고 가상 DOM에서 이전 트리와 render된 트리를 비교하는 단계
  - commit 단계 : 가상DOM에서의 비교 결과 차이가 나는 부분을 DOM에 업데이트하는 단계

❖레코딩 방법

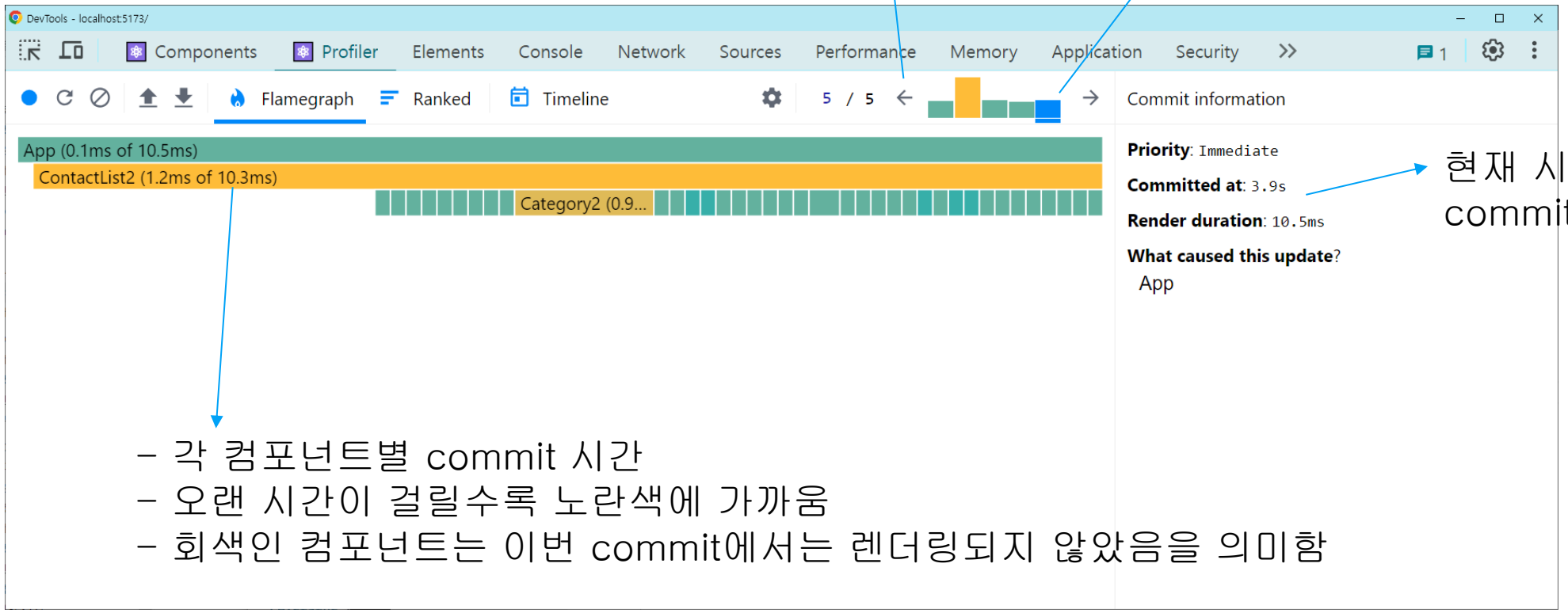
- 왼쪽 상단의 버튼(●) 을 클릭하여 레코딩 시작
- 사용자 인터랙션을 일으킨 후 마지막에 다시 ●을 클릭하여 레코딩 종료

5.1 React Developer Tools의 Profiler

❖분석 결과

좌우 화살표를 클릭하여 각 시점의 commit 정보를 볼 수 있음

선택된 시점(파란색)



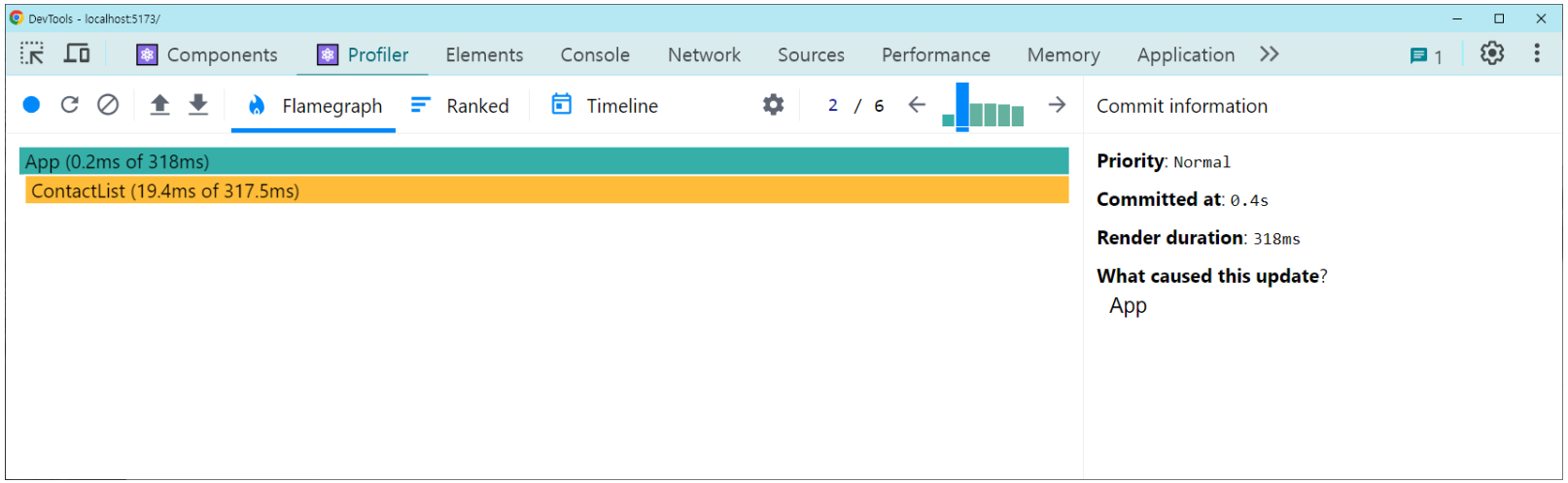
현재 시점의 commit 정보

- 각 컴포넌트별 commit 시간
- 오랜 시간이 걸릴수록 노란색에 가까움
- 회색인 컴포넌트는 이번 commit에서는 렌더링되지 않았음을 의미함

5.2 Profiler를 사용한 성능 측정/개선

❖photo-card-list 앱 실행 후 React Profiler를 이용해 성능을 측정해보자

- Profiler에서 레코딩을 시작한 후 첫번째 연락처의 전화번호를 두번 변경해봄(1로 변경, 2로 변경)
- 레코딩을 중지후 수집된 렌더링 시간 정보를 확인해 봄



- 컴포넌트 렌더링 시간 : H/W, 브라우저 성능에 따라 결과는 달라질 수 있음

	phase 1	phase 2	phase 3	phase 4	phase 5	phase 6
App	3.4	318	44.6	42.6	37.4	30.5
ContactList	0.8	317.5	44.3	38.1	37.1	29.5

5.2 Profiler를 사용한 성능 측정/개선

❖6단계의 내용

- phase 1
  - 컴포넌트가 마운트되고 빈 배열의 연락처 데이터로 초기 렌더링을 수행함.
- phase 2
  - 컴포넌트가 마운트된 후 샘플데이터 1000건이 contacts 상태 데이터에 채워지고 그 결과 다시 렌더링
- phase 3
  - 첫번째 연락처의 전화번호를 변경하기 위해 입력필드에 "1"을 입력함 --〉 tel 상태의 변경으로 컴포넌트 렌더링
- phase 4
  - '변경' 버튼을 클릭하여 연락처(contacts)의 첫번째 연락처의 전화번호로 변경
  - contacts 상태가 변경되었으므로 컴포넌트 전체 렌더링
- phase 5
  - 새로운 전화번호 "2" 입력 --〉 tel 상태의 변경으로 컴포넌트 렌더링
- phase 6
  - '변경' 버튼을 클릭하여 연락처(contacts)의 첫번째 연락처의 전화번호로 변경
  - contacts 상태가 변경되었으므로 컴포넌트 전체 렌더링

5.2 Profiler를 사용한 성능 측정/개선

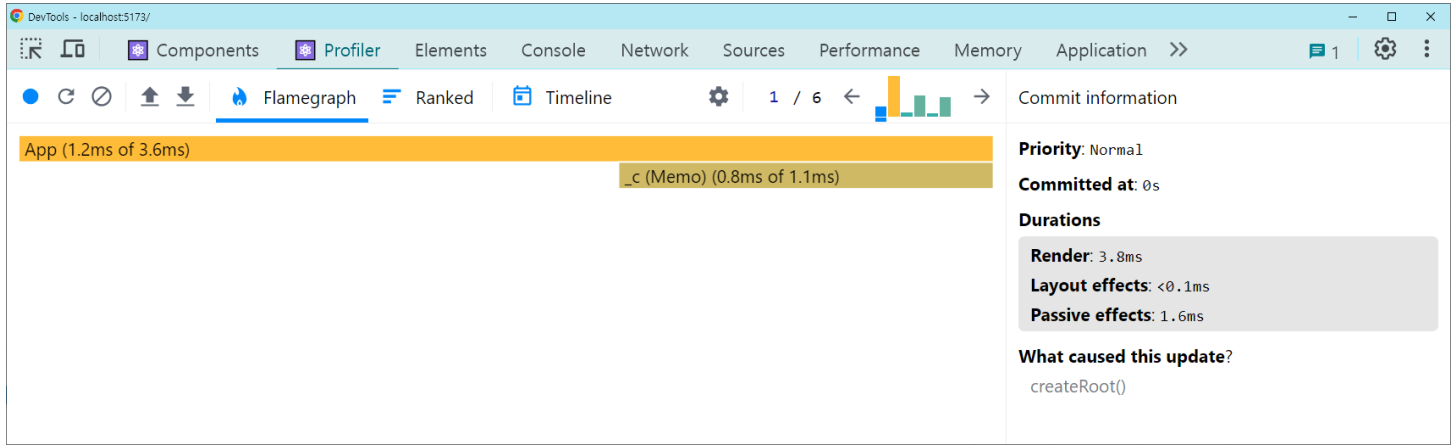
❖3단계와 4단계의 실행 시간이 차이가 없는 이유는?

- 상태가 바뀌었으므로 전체 컴포넌트 트리가 재실행됨
- 이것을 개선할 수 있는 방안은?
  - ContactList 컴포넌트에 React.memo() 고차함수를 적용한 후 성능을 측정해보자

```
const ContactList = React.memo(({ contacts }: PropsType) => {···
});
```

5.2 Profiler를 사용한 성능 측정/개선

❖React.memo() 적용 결과



	phase 1	phase 2	phase 3	phase 4	phase 5	phase 6
App	3.6	339.2	0.3	48.5	0.1	38.3
ContactList	1.1	338.8	X	45	X	37.5

- phase 3, phase 5에서 ContactList 컴포넌트로 전달된 속성이 변경된 것이 없으므로 렌더링되지 않음 --〉 성능 개선
- 하지만 phase4, phase5에서의 렌더링 시간을 줄일 수 없을까?
  - 컴포넌트 분할 후 memo()

5.2 Profiler를 사용한 성능 측정/개선

❖ContactItem 컴포넌트 추가

- src/ContactItem.tsx 추가

```
import { ContactType } from "./App"

type PropsType = { item : ContactType }

const ContactItem = ({ item }: PropsType) => {
  return (
    <div className="col-12 col-xs-12 col-sm-6 col-md-4 col-lg-3 col-xl-3 d-flex justify-content-center" >
        <div className="card mb-2" style={{ width:"200px" }}>
        <img src={item.photo} className="card-img-top" alt={item.name} width="200" height="200" />
        <div className="card-body">
            <h5 className="card-title">{item.name}</h5>
            <p className="card-text">{item.tel}</p>
            <p className="card-text">{item.address}</p>
        </div>
        </div>
    </div>
  )
}

export default ContactItem
```

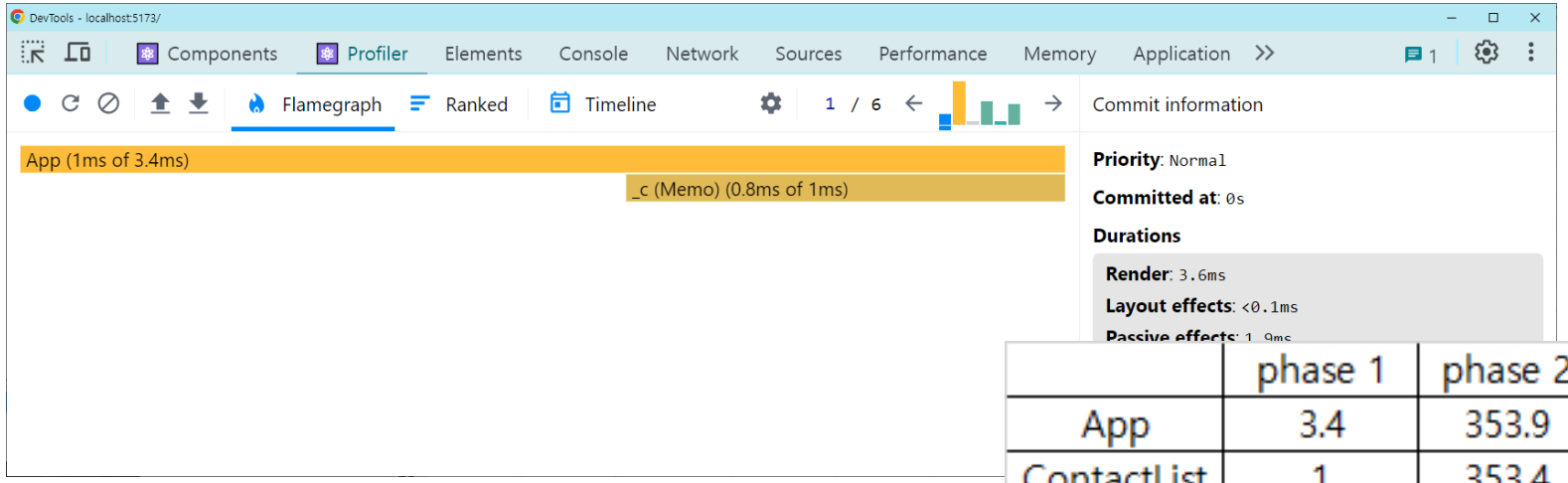
5.2 Profiler를 사용한 성능 측정/개선

❖ContactList 컴포넌트 변경 : src/ContactList.tsx

```
import React from "react";
import { ContactType } from "./App";
import ContactItem from "./ContactItem";

type PropsType = {  contacts: ContactType[]; };

const ContactList = React.memo(({ contacts }: PropsType) => {
  const ContactItemList = contacts.map((item) => (
    <ContactItem key={item.no} item={item}/>
  ));

  return (
    <div className="container">
      <div className="row">
        <div className="col-12">
          <h3 className="m-5">하나의 컴포넌트</h3>
        </div>
      </div>
      <div className="row justify-content-evenly text-center">{ContactItemList}</div>
    </div>
  );
});

export default ContactList;
```

5.2 Profiler를 사용한 성능 측정/개선

❖컴포넌트 분할만 수행 후 성능 측정



	phase 1	phase 2	phase 3	phase 4	phase 5	phase 6
App	3.4	353.9	0.1	52.4	0.1	36.3
ContactList	1	353.4	X	48.5	X	35.4

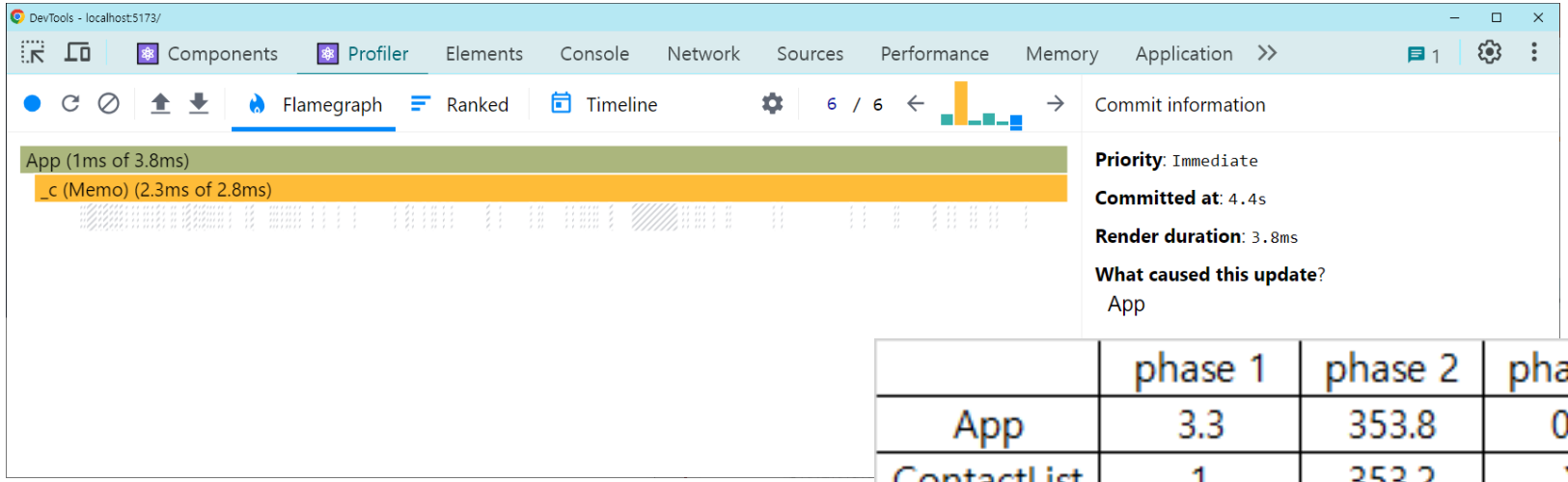
- 초기 렌더링
  - 컴포넌트 분할 이전보다 약간의 실행 시간 증가
- 상태 변경 후 : phase 4, phase 6
  - 컴포넌트 분할 이전과의 의미있는 차이 없음
- 결과
  - 컴포넌트를 분할만 해서는 성능의 개선 없음 --〉 오히려 약간의 성능 악화

5.2 Profiler를 사용한 성능 측정/개선

❖ContactItem 컴포넌트에 memo() 적용

```
const ContactItem = React.memo(({ item }: PropsType) => {…
});
```

❖성능 측정 결과



	phase 1	phase 2	phase 3	phase 4	phase 5	phase 6
App	3.3	353.8	0.3	7.1	0.2	3.8
ContactList	1	353.2	X	3	X	2.8

- phase 4, phase 6 렌더링 시간 감소
  - 첫번째 photocard의 ContactItem 컴포넌트만 렌더링되었기 때문에…

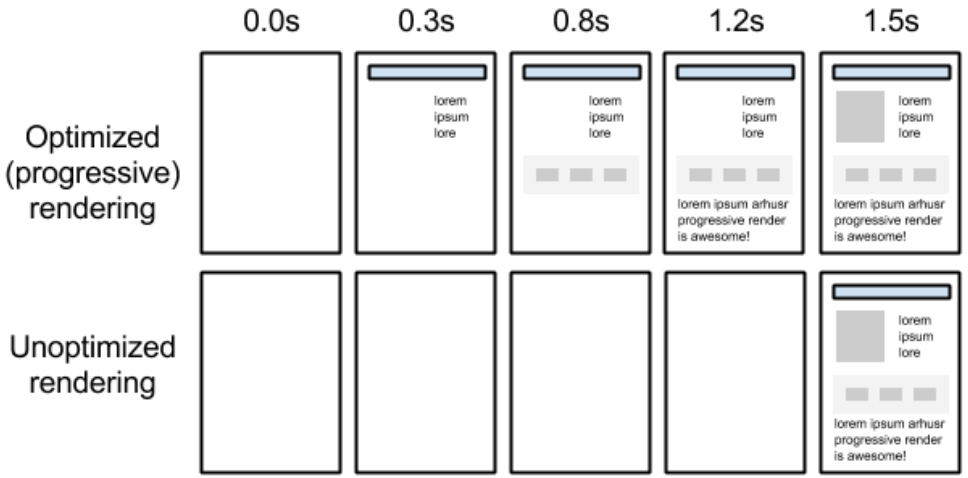
5.3 Infinite Scroll 적용으로 성능 개선

❖리액트 개발자 도구의 프로파일러만으로 성능 최적화가 가능한가?

- 단지 DOM에 render-commit하는 시간만 측정함
- 사용자가 체감하는 성능과는 괴리감이 있음

❖그렇기 때문에 core web-vitals 를 개선해야 함

- 예) photo-card-list 앱은 전체 렌더링 시간은 400ms 이하이지만 photo card들이 모두 한 요소(div) 내에 있으므로 400ms가 지난 후 전체 photo card들이 나타남
- 점진적으로 요소들이 나타나는 것이 사용자에게는 더 빠르게 느껴짐.

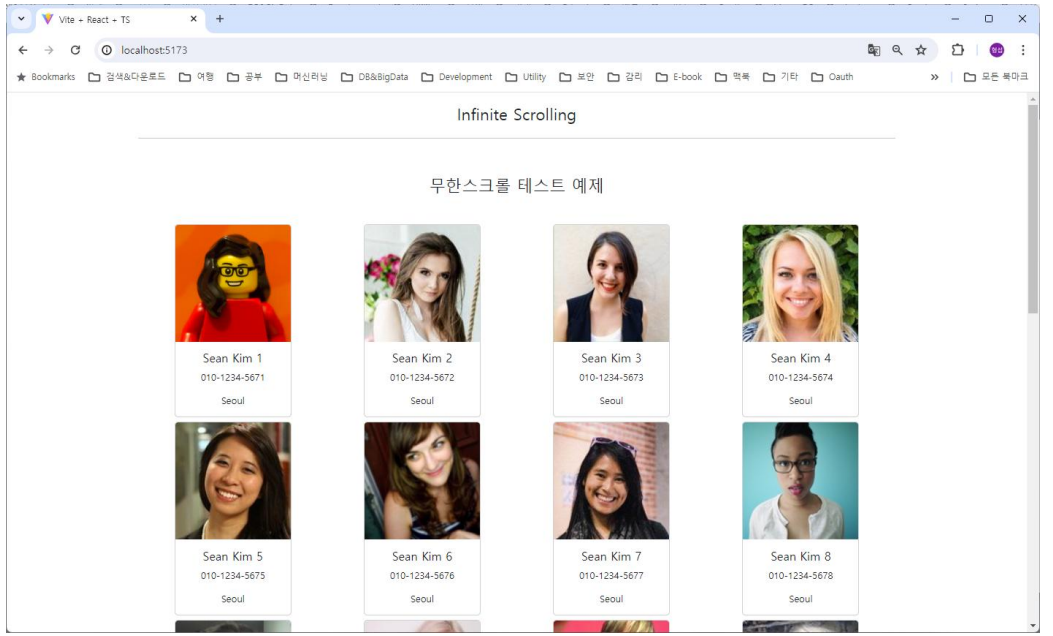


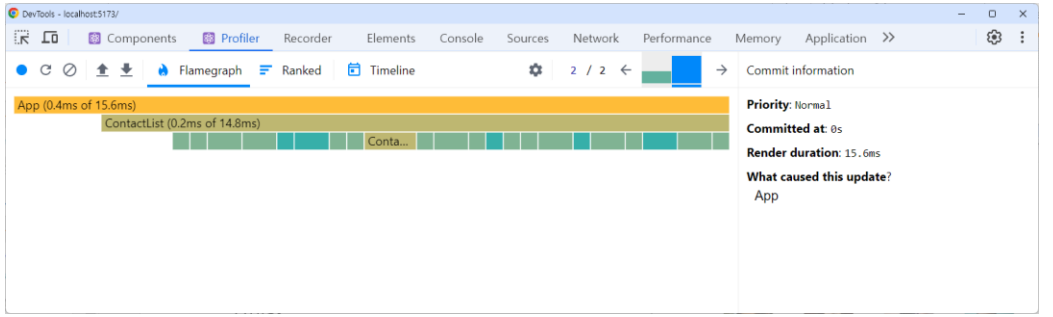
https://web.dev/articles/critical-rendering-path?hl=ko

5.3 Infinite Scroll 적용으로 성능 개선

❖개선 방법

- 전체 데이터가 아닌 데이터 조각들을 조금씩 보여주기
- 예1) 무한 스크롤
  - photo-card-list-531-infinite-scroll 예제 참조
    - 전체 샘플 데이터 : 1000건
    - 한번에 24건씩 끊어서 화면에 보여주고 스크롤바를 아래로 내리면 다음 24건의 데이터를 보여줌
    - 사용자 브라우저의 첫 렌더링시에 24건의 ContactItem 만 렌더링함
    - 첫렌더링 시간 비교
      - » 353ms VS 15.6ms
- 예2) 윈도우 스크롤, 가상 스크롤
  - photo-card-list-532-react-window
    - 가상 스크롤 방식
    - react-window 라이브러리 사용
    - 화면 영역에 보여줄 데이터만 렌더링함





6. Lighthouse를 이용한 성능 측정

❖Lighthouse
- Chrome, Edge 브라우저에 내장된 성능 측정 도구 또는 CLI 명령어

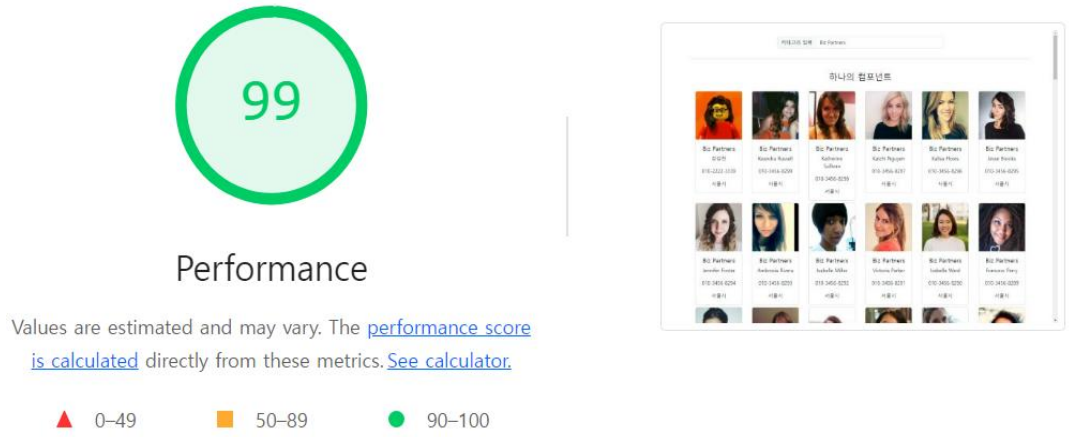
❖3가지 측정 모드
- Navigation Mode
- Timespan Mode
- Snapshot Mode

❖Lighthouse 스코어 가중치
- FCP(First Contentful Paint) : 15%
- LCP(Largest Contentful Paint) : 25%
- TTI(Time to InteractiveO : 15%
- CLS(Cumulative Layout Shift) : 5%
- TBT(Total Blocking Time) : 25%
- SI(Speed Index) : 15%

6.1 Lighthouse Mode

❖Navigation Mode

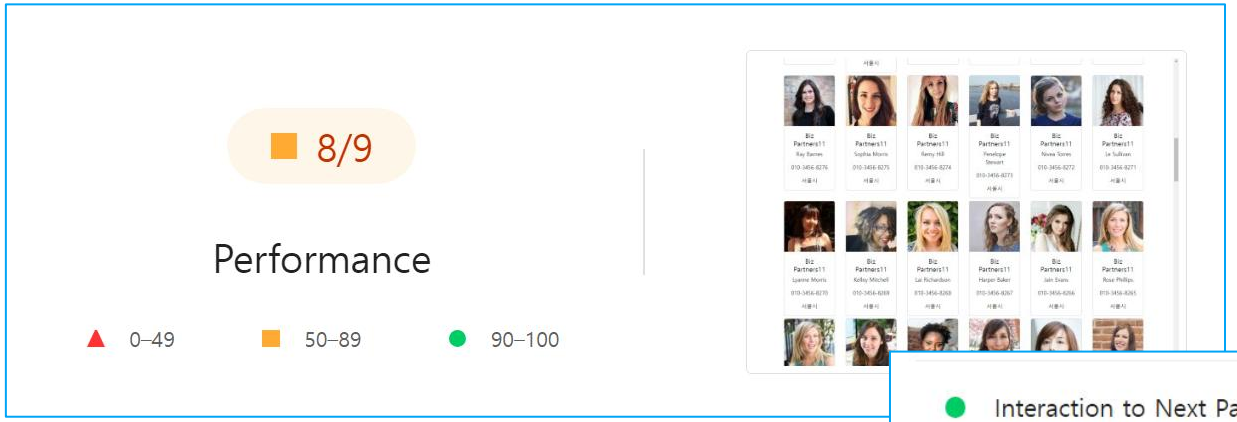
- 단일 페이지의 로드를 분석
- 측정 항목
  - Performanca
  - Accessibility
  - Best Practices
  - SEO
  - PWA
- 성능 관련 측정 지표
  - FCP
  - LCP
  - TBT
  - CLS
  - Speed Index
- FID 또는 CIP 측정은 수행하지 않음

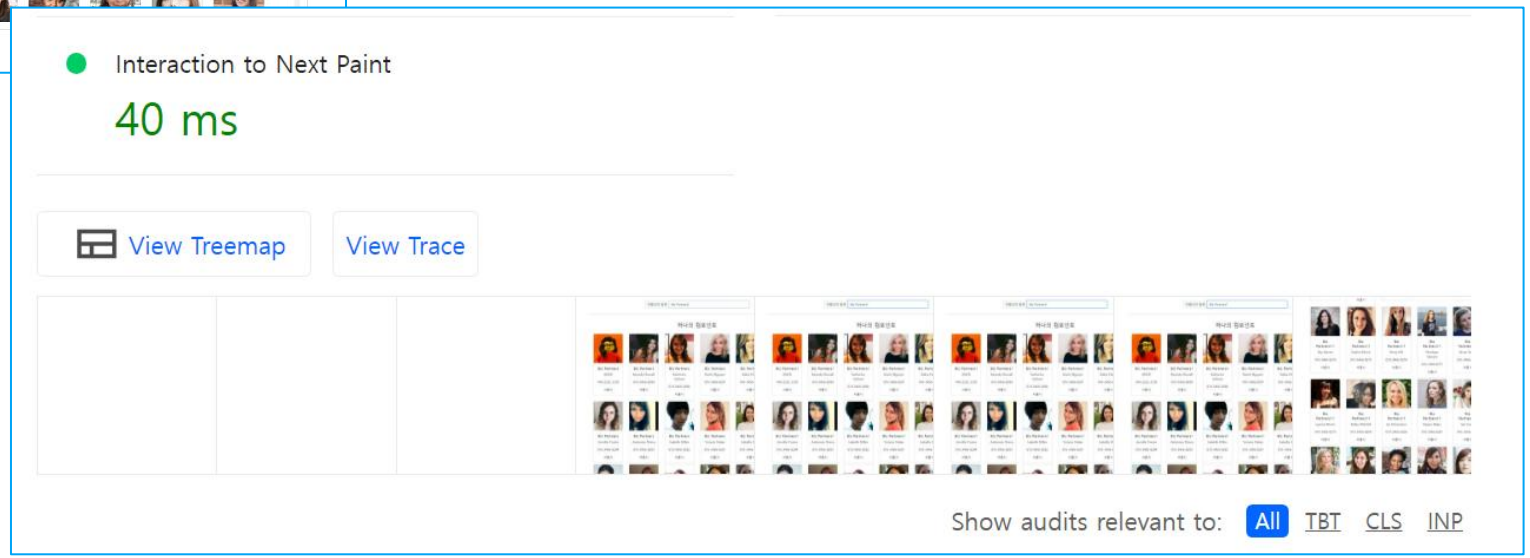


6.1 Lighthouse Mode

❖Timespan Mode

- 사용자의 인터랙션을 포함하는 시간 범위 동안의 성능, 모범사례 준수 여부를 확인함

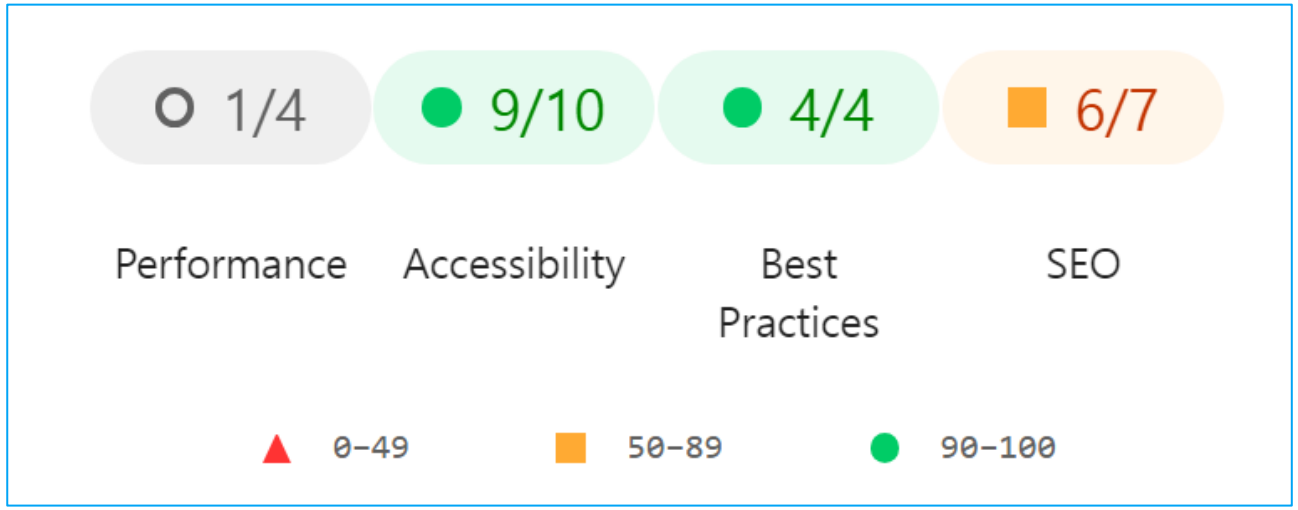




6.1 Lighthouse Mode

❖Snapshot Mode

- 특정 상태의 페이지를 분석하여 4가지 측면의 평가를 수행함
- 인터랙션이 일어나는 과정속에서의 UI 요소의 모범사례 적합 여부를 평가함
  - 4가지 측면에서의 개별 분석 결과를 볼 수 있음
  - Performance
  - Accessibility
  - Best Practices
  - SEO



6.2 Lighthouse 장점

❖Lighthouse의 장점

- 진단(Diagnostics) 기능
  - 진단 결과 스코어가 현저하게 낮은 것은 문제점과 해결책을 제시해줌

DIAGNOSTICS

- ▲ Preload Largest Contentful Paint image — Potential savings of 880 ms
- ▲ Enable text compression — Potential savings of 1,319 KiB
- ▲ Minify JavaScript — Potential savings of 568 KiB
- ▲ Serve images in next-gen formats — Potential savings of 551 KiB
- ▲ Efficiently encode images — Potential savings of 529 KiB
- ▲ Largest Contentful Paint element — 1,940 ms

▲ Largest Contentful Paint element — 1,940 ms

This is the largest contentful element painted within the viewport. Learn more about the Largest Contentful Paint element LCP

Element

img.card-img-top

Phase	% of LCP	Timing
TTFB	7%	130 ms
Load Delay	77%	1,490 ms
Load Time	1%	20 ms
Render Delay	16%	300 ms

6.2 Lighthouse CLI

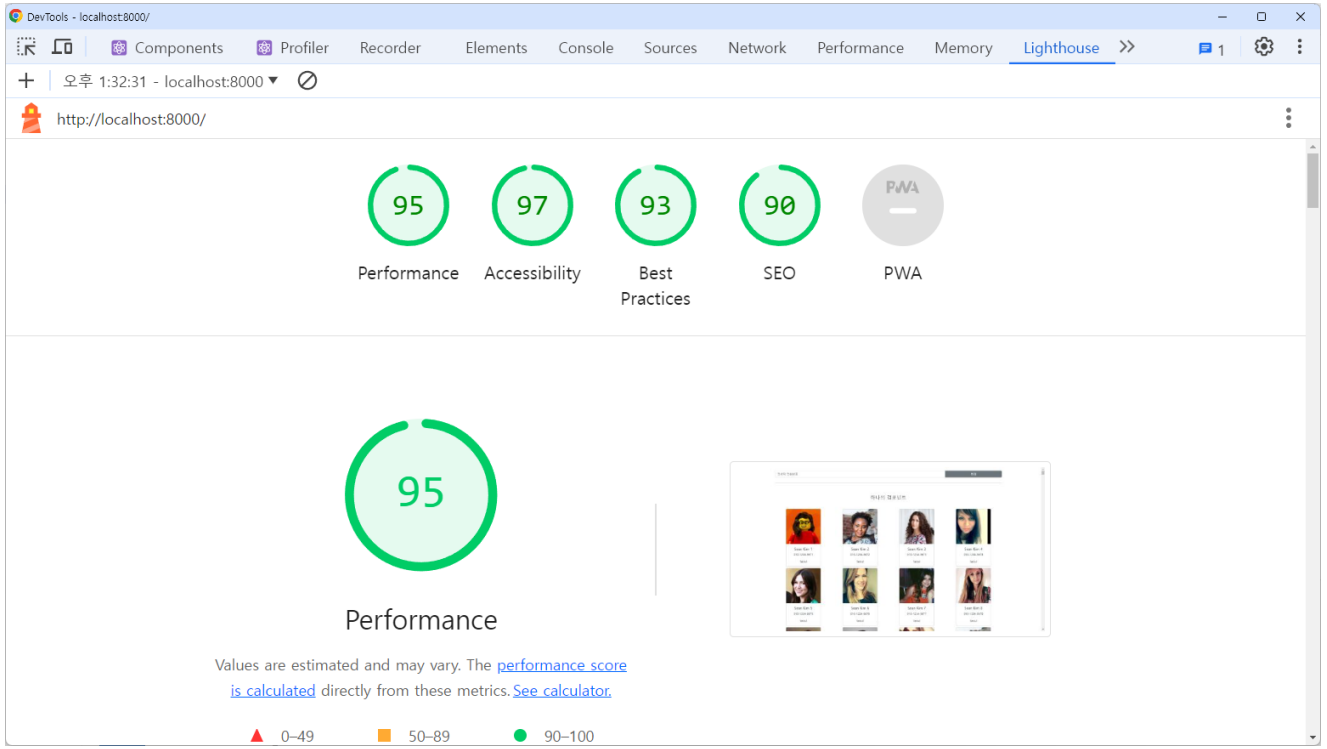
❖Lighthouse CLI 도구 사용하기

```
** 사용 방법 보기
npx lighthouse --help

** 사용 예시(윈도우 터미널)
npx lighthouse http://localhost:5173  --view `
   --screenEmulation.width=1280 `
   --screenEmulation.height=800 `
   --no-emulated-user-agent
```

6.3 Lighthouse를 이용한 성능 개선

❖photo-card-list 앱을 사용하여 성능 측정, 개선

- 5.2절까지 진행한 예제를 이용함
  - npm run build
  - npx serve dist -p 8000
- Edge 또는 Chrome 브라우저에서 lighthouse 실행후 결과 확인



6.3 Lighthouse를 이용한 성능 개선

❖측정 결과 분석

- 전반적으로 성능은 바람직해보이지만 몇가지 개선항목이 나타남
- Diagnostics(진단) 섹션의 항목을 살펴보자

❖개선할 사항

- 과도한 DOM 크기를 피하자 – TBT 개선
- 이미지 크기를 적절한 크기로 줄이고 차세대 이미지 형식 사용 – FCP, LCP 개선
- 렌더링 차단 리소스 제거 – FCP, LCP 개선
- 이미지에 명시적인 크기 지정 – CLS 개선
- 정적 리소스 캐싱

6.3 Lighthouse를 이용한 성능 개선

❖과도한 DOM 크기를 피하자 – TBT 개선

- 〈div /〉 요소 하나에 7000여개의 요소가 포함되어 있음
- Infinite Scrolling 기법등을 적용하여 한번에 로드되는 요소 수를 줄이거나 7000여개의 요소를 적절히 분할하여 여러 〈div /〉 요소에 분산하여 로딩함.

❖이미지 크기를 적절한 크기로 줄이고 차세대 이미지 형식 사용 – FCP, LCP 개선

- noimage.jpg 의 크기는 500x500이지만 화면에는 200x200 크기로 보여주고 있음
  - 이미지의 크기를 축소시켜 사용하면 성능 개선
  - png, jpeg 과 같은 형식보다 webp, AVIF 형식이 압축율이 좋으므로 데이터 전송 크기를 줄일 수 있음
- 도구/방법
  - sharp-cli 도구 등의 이미지 크기 변경도구 사용.
  - image CDN 사용
  - 반응형 이미지 제공

6.3 Lighthouse를 이용한 성능 개선

- sharp-cli 도구를 이용한 크기 조정
  - 설치 : npm install -D sharp-cli
  - 사용 예시
    - **npx sharp resize 128 128 --compression webp -i ./public/photow/noimage.jpg -o ./public/photow/noimage.webp**
    - npx sharp resize 128 128 --compression webp -i ./public/photow/*.jpg -o ./public/photow2
  - 생성된 파일 크기 비교 : 546KB VS 2.11KB

❖렌더 블로킹 리소스를 제거- FCP, LCP 개선

- 주요 블로킹 리소스
  - 자바스크립트 코드 , CSS
- 제거 방법
  - 자바스크립트 코드
    - 사용하지 않는 코드 삭제
    - 중요하지 않은 코드 또는 화면 초기화시에 사용되지 않는 코드는 async 속성부여
  - CSS
    - CSS 파일 압축 : webpack, rollup 등이 제공, CRA 또는 Vite 기반으로 개발했다면 빌드 기능에 내장되어 있음
    - 중요하지 않은 스타일은 rel="preload" 속성을 추가하여 로딩을 지연시킴

6.3 Lighthouse를 이용한 성능 개선

- CSS preload 설정
  - npm run build 로 프로젝트 빌드
  - dist/index.html 파일에서 다음 코드의 볼드체 부분을 변경

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React + TS</title>
    <script type="module" crossorigin="" src="/assets/index-xxxxx.js" async></script>
    <link rel="preload" as="style" onload="this.onload=null;this.rel='stylesheet'"
      crossorigin="" href="/assets/index-xxxxxx.css" />

  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

  - npx serve dist -p 8000

6.3 Lighthouse를 이용한 성능 개선

❖이미지에 명시적인 크기 지정 – CLS 개선

- 이미지에 명시적인 크기를 지정하지 않으면?
  - 브라우저 렌더링 과정중 이미지를 위한 공간을 미리 확보하지 않음
  - 이미지가 로드되면서 Layout의 Shift가 일어나므로 CLS가 악화됨
- 명시적 크기 지정 방법
  - width, height 속성으로 크기 정보를 고정값으로 지정함

```
<img src={item.photo} className="card-img-top" alt={item.name} width="200" height="200" />
```

❖LCP 컨텐트 미리 로드 –〉 리소스 발견 시점을 더 빨리

- HTML문서에서 미리 로드해야할 것을 link 태그로 지정
- LCP 컨텐트가 무엇인지 명확할 때 사용

```
<link rel="preload" fetchpriority="high" as="image" href="/photow/noimage.jpg" type="image/jpeg" />
```

6.3 Lighthouse를 이용한 성능 개선

❖정적 리소스 캐싱

- 리액트 애플리케이션이 배포되는 서버에 다음과 같은 Response Header를 설정함
  - 10일간의 캐시 설정 : Cache-Control : max-age=864000
  - 86400초 = 1day
- 캐시 기간이 길어지면 정적 자원의 변경이 사용자 브라우저 화면에 반영되지 않을 수 있음