김상현 포트폴리오

엔지니어링 관점으로 좋은 제품을 만드는 노력들

disclaimer

- 가장 논의할 가치가 큰 플래시카드 프로젝트 위주로 설명
- 백엔드도 직접 작업했지만 프론트엔드 위주로 설명
- 프로젝트 다양하게 여러개 보단 하나 깊게가 더 가치있다고 보고 결정
- 이번 발표로 최대한 인사이트 제공
- 일부 예시 코드는 원본 그대로가 아닌 생략과 편집을 함
- 토큰은 인사이트 보단 기본이 보여주기
    - 발표시간 부족하면 생략

플래시 카드 프로젝트

목차

플래시카드란 무엇인가?

- 인덱스 카드라고 다른 이름도 있음
- 앞면에 문제 뒷면에 정답
- 정답은 내일 다시 틀리면 1시간 후 다시
- 정답을 맞출 때마다 다음에 풀이까지 간격 누적
    - 예: 1일, 2일 3일 1주일 ...

프로젝트 배경

- 영단어 어휘력이 약함
  - 영타가 느림
  - 영단어 검색을 자주함
- 플래시카드로 암기가 효율이 제일 좋음
- 이미 만들어진 서비스를 사용하면 광고가 너무 많음
- 무료는 성능 문제도 너무 많음
- 대부분 서비스는 실제로 원하는 단어 입력을 못함

해결책: 직접 만든다.

- 서비스의 모든 부분을 직접 만들기
- 프론트엔드 백엔드 모두 솔로 프로젝트로 진행
    - 백엔드 없어서 프로젝트 중단을 방지
- 1.0으로 끝내지말고 장기적으로 보수하고 기능추가해보기

> "dogfooding": the practice of using one's own products or services.
>
> - 출처: Eating your own dog food - 위키 피디아

기술 스택

프론트엔드

- React(Vite)
- Axios
- React Router DOM
- React-Query
- Jotai
- Emotion
- React-Spinner

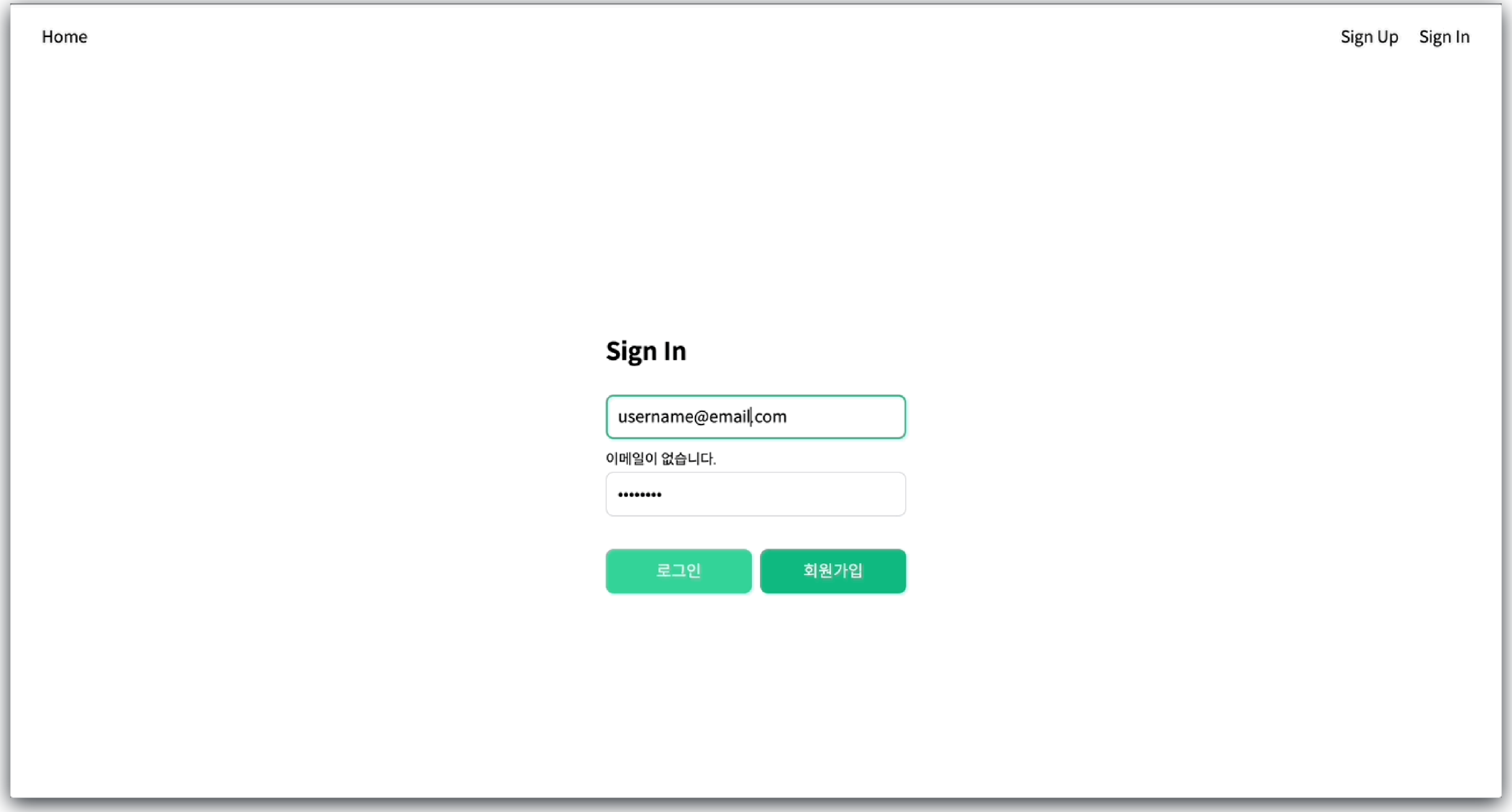
백엔드

- Deno
- Deno deploy
- Oak
- mongoDB

직접 만들면서 다양한 트러블 슈팅

Request Waterfall 숨기기

- UI상 로그인 로딩 1번만 하기


Home
Sign Up
Sign In
Sign In
username@email.com
이메일이 없습니다.
로그인
회원가입

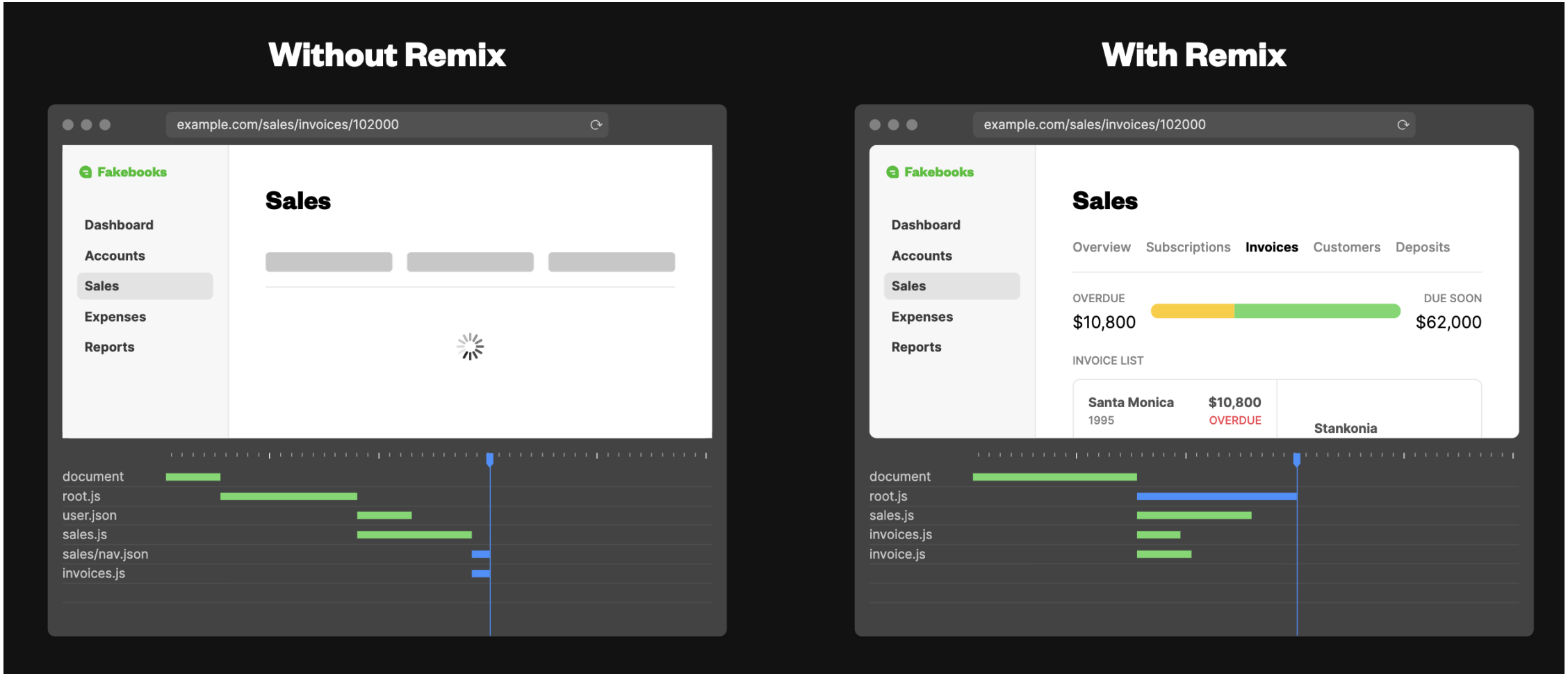
배경

- 사용자를 2번 기다리게 하는 경험
- A를 요구한 사용자에게 잠시 B를 주고 A로 교체하는 경험
    - action과 feedback이 불일치
- 참고. 로딩 중 사용자 경험이 좋은 순서:
    - 로딩 없음 > 스켈레톤 > 스피너 > 피드백 없음

Request Waterfall이란?

- 요청의 폭포처럼 화면에 여러개의 로딩 스피너가 보임
- 컴포넌트가 nested 구조를 갖게 되면서 발생
  - 상위 mount 요청이후 하위 컴포넌테에서 mount 후 추가 요청이 생김
- 요청간 의존성 때문에도 발생

Request Waterfall 예:



출처: Remix 공식 홈페이지

분석

- 로그인 요청하고 본인 리소스를 요청해야 함
  - 이런 이유로 요청의 의존성은 반드시 가질 수 밖에 없음
- 로그인으로 라디이렉팅 전에 본인 리소스를 요청하는 방법이 있음
  - onMount 라이프 사이클 전에 요청응답을 처리
  - 로그인처리 시간에 본인 리소스 요청을 UI에 포함

해결

- tkdodo의 React-Router-DOM과 React-Query 연계 적용
- i. 로그인 요청
- ii. 로그인 성공시 loader 실행
- iii. loader에서 본인 카드를 서버 요청
- iv. 응답 받은 후 리다이렉팅

로그인

```
const signIn = async () => {
  setEmailError('');
  setPasswordError('');
  mutate(
    {
      email: emailValue,
      password: passwordValue,
    },
    {
      onSuccess(data) {
        const { access_token } = data;
        setTokens(access_token);
        navigate('/main');
      },
    }
  );
};
```

- 로그인 이벤트를 처리하는 함수

```tsx
const Cards = lazy(() => import('../pages/Cards'));
const SignIn = lazy(() => import('../pages/SignIn'));

const routes = createBrowserRouter([
  {
    path: '/',
    element: <Layout />,
    children: [
      {
        path: '/main',
        element: <Cards />,
        loader: cardLoader(),
      },
      {
        path: '/signin',
        element: <SignIn />,
      },
    ],
  },
]);
```

- 라우팅 전 실행

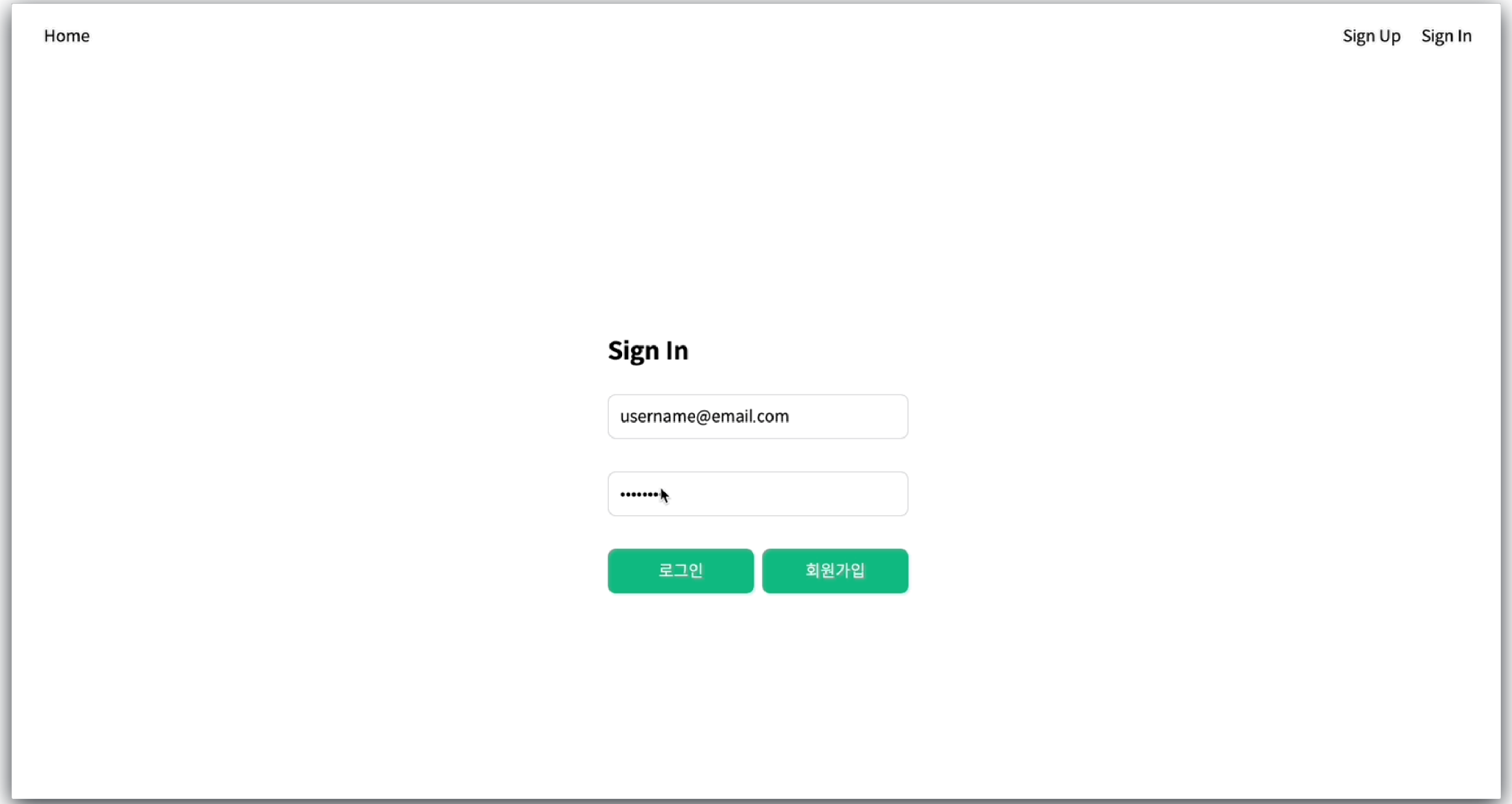
```tsx
import queryClient from '@/libs/queryClient';

export const cardLoader = () => async () => {
  const query = () => ({
    queryKey: ['cards'],
    queryFn: getCardsAPI,
    staleTime: 5000,
  });

  return (
    queryClient.getQueryData<Card[]>(query().queryKey) ??
    (await queryClient.fetchQuery(query))
  );
};
```

- query cache가 없으면(`undfined`)
  fetchQuery 실행

```tsx
function Main() {
  const loaderCards = useLoaderData() as Awaited<
    ReturnType<ReturnType<typeof cardLoader>>
  >;

  const query = cardsQuery();
  const {
    data: cards,
    isLoading,
    error,
  } = useQuery({ ...query, initialData: loaderCards });

  return <>{/* ... 생략 */}</>;
}
```

- `cardLoader`의 반환값을 활용해서 타입 지정
- 서버에서 받응 응답을 query cache로 caching
- 참고. 로그인 처리후 스피너는 전역 상태 flag로 처리

결과



에러바운더리 적용

1. 유저에게 에러 메시지를 보여주자
2. 재요청을 제어할 수 있게 해주자
3. loader에서 발생한 에러를 catch하기

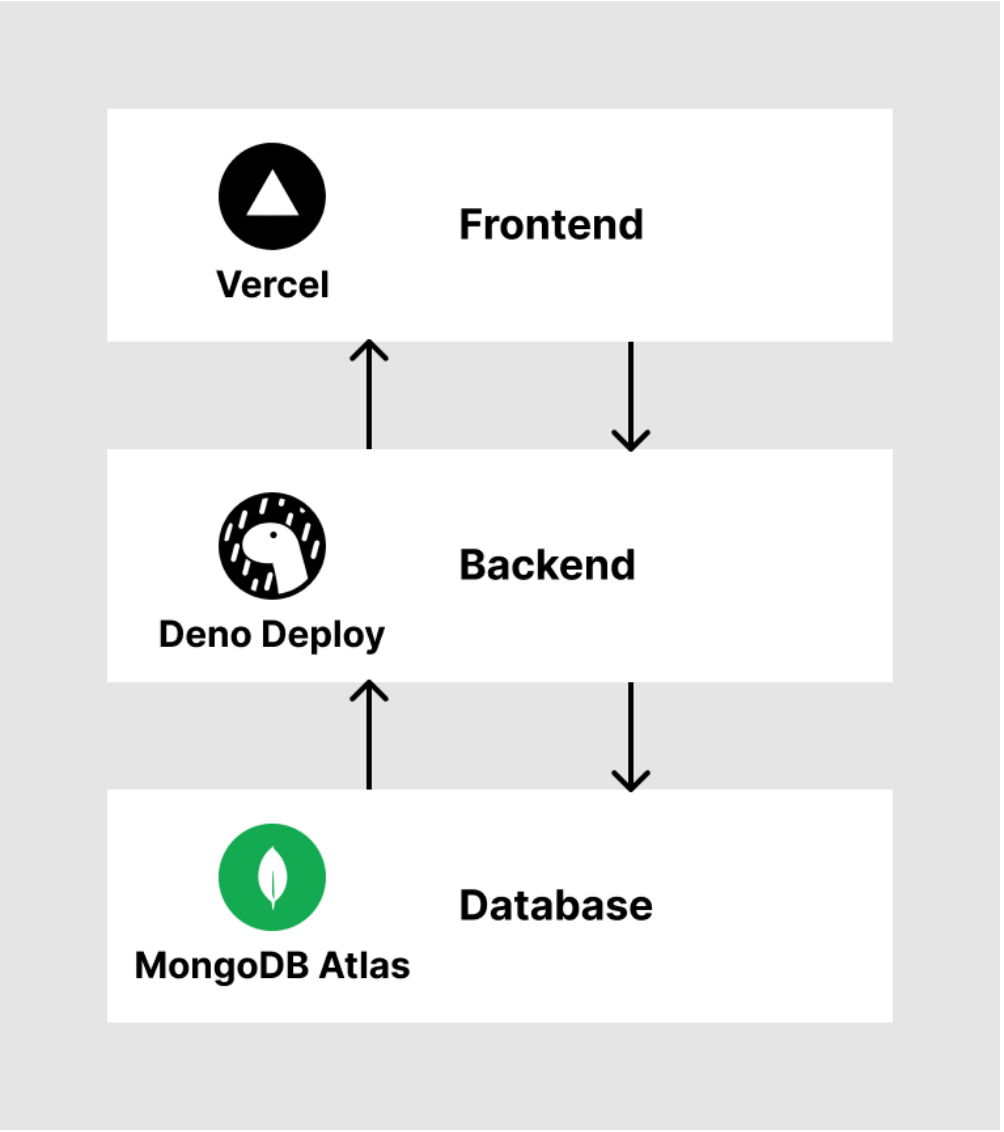
배경 1

- 기존 에러는 `<div>에러</div>` 형식으로만 표현
- 피드백을 더 구체적으로 제공하기 위해 추가함
- 유저가 싫어하는 것은 아무것도 할 수 없다는 무력감

```
function Main() {
  const { cards, isLoading, error } = useCards();

  if (typeof cards === 'string' || error) {
    return <div>{`${error}`}</div>;
  }

  return <div>생략</div>
```

분석 1

- vercel, deno deploy, mongoDB Atlas 플랫폼 활용
- 3개 플랫폼 중 1개의 장애가 발생하면 서비스 이용이 불가능함



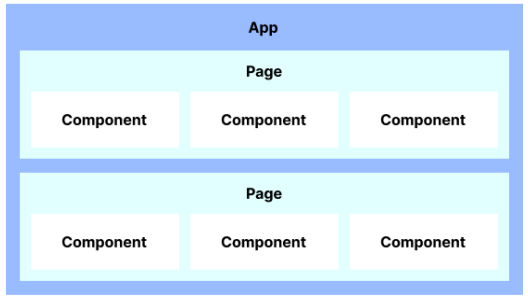
분석 2

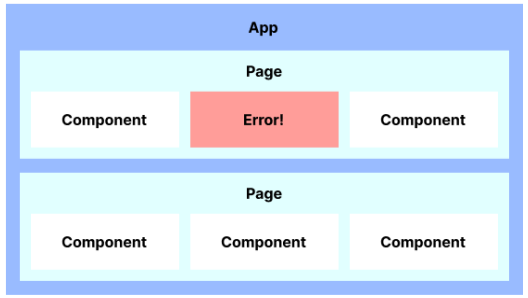
```
const Cards = lazy(() => import('../pages/Cards'));

const routes = createBrowserRouter([
  {
    path: ROUTE_PATHS.WELCOME,
    element: <Layout />,
    children: [
      {
        path: ROUTE_PATHS.CARDS,
        element: <Cards />,
        loader: cardLoader(),
      },
    ],
    errorElement: <ServerError />,
  },
]);
```

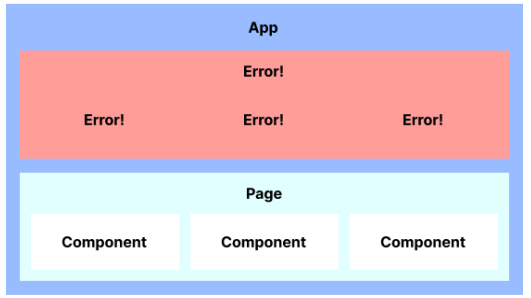
```
import queryClient from '@/libs/queryClient';

export const cardLoader = () => async () => {
  const query = () => ({
    queryKey: ['cards'],
    queryFn: getCardsAPI,
    staleTime: 5000,
  });

  return (
    queryClient.getQueryData<Card[]>(query().queryKey) ??
    (await queryClient.fetchQuery(query))
  );
};
```

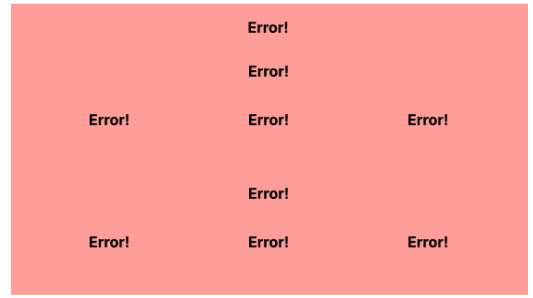
- loader에서 예외처리가 없음
- loader에서 에러가 발생할 경우 `<ServerError/>` 페이지를 랜더링함
- 통신에러의 부분이 전체로 확산(action과 feedback의 불일치)

No ErrorBoundary

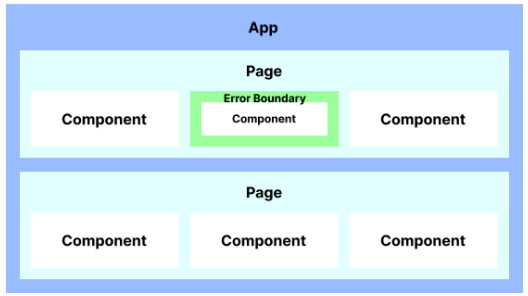


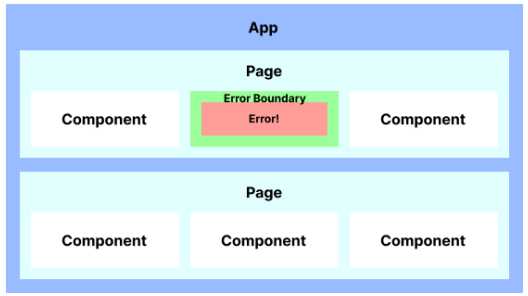


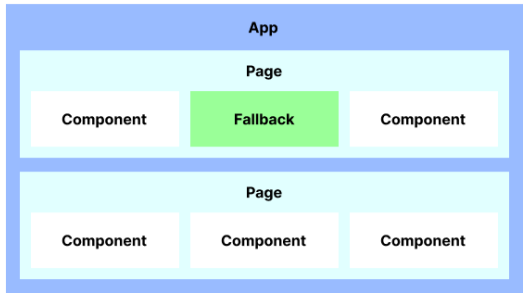




Error Boundary







Error Boundary란?

- React의 컴포넌트 차원에서 catch하는 방법
- 리액트 라이프 사이클 중 컴포넌트 내부에 에러가 발생하면
- 컴포넌트의 상태를 바꿔 fallback을 대신 랜더링하도록 함
- `getDerivedStateFromError` 에러 발생 시 실행(`hasError` 갱신)
- `componentDidCatch` 에러 logging

```tsx
import { Component, ErrorInfo, ReactNode } from 'react';

interface Props {
  children?: ReactNode;
  fallback: ReactNode;
}

interface State {
  hasError: boolean;
}

export class ErrorBoundary extends Component<Props, State> {
  public state: State = {
    hasError: false,
  };

  public static getDerivedStateFromError(_: Error): State {
    return { hasError: true };
  }

  public componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    console.error('Uncaught error:', error, errorInfo);
  }

  public render() {
    if (this.state.hasError) return this.props.fallback
    return this.props.children;
  }
}
```

해결 1

- 옆 예시는 ErrorBoundary 설치하고 적용
- 내부 동작은 동일함
- 이유는 React Query와 연계하기 위해
  - 참고. React-Query 공식 문서에서 설치버전을 권장함
- 실패하면 error를 컴포넌트 차원에서 catch

```
import { ErrorBoundary } from 'react-error-boundary';
import { useQueryErrorResetBoundary } from '@tanstack/react-query';

function Cards() {
  const { reset } = useQueryErrorResetBoundary();

  return (
    <CardPageContainer>
      <PageHeading>Cards</PageHeading>
      <ErrorBoundary onReset={reset} fallbackRender={ErrorCards}>
        <NowDeck />
      </ErrorBoundary>
    </CardPageContainer>
  );
}
```

```tsx
type ErrorCardsProps = {
  error: { success: boolean; msg: string };
  resetErrorBoundary: (...args: any[]) => void;
};

export function ErrorCards({ error, resetErrorBoundary }: ErrorCardsProps) {
  return (
    <CardContainer>
      <ErrorCardsContainer>
        <DisabledText>{error.msg}</DisabledText>
        <Button onClick={resetErrorBoundary}>Retry</Button>
      </ErrorCardsContainer>
    </CardContainer>
  );
}
```

- 엮은 유저가 보게될 fallback 컴포넌트
- 실제 error 메시지를 화면에 보여줌
- 백엔드작업 하면서 API 실패시 body는 `{ success: false, msg: "에러 내용" }` 으로 처리
- 요청 실패 후 재요청 버튼을 UI로 제공

결과

토큰 갱신 처리하기

- 트러블 슈팅보단 개발 여정과정

회원가입

```tsx
function SignUp() {
  const { mutate } = useMutation({ mutationFn: signUpAPI });

  const handleSignUp = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();

    mutate(
      { email, password },
      {
        onSuccess: (data) => {
          if (data.status === 201) navigate(ROUTE_PATHS.SIGN_IN);
        },
        onError: (err) => {
          // ...
        },
      }
    );
  };

  return <div>생략</div>;
}
```

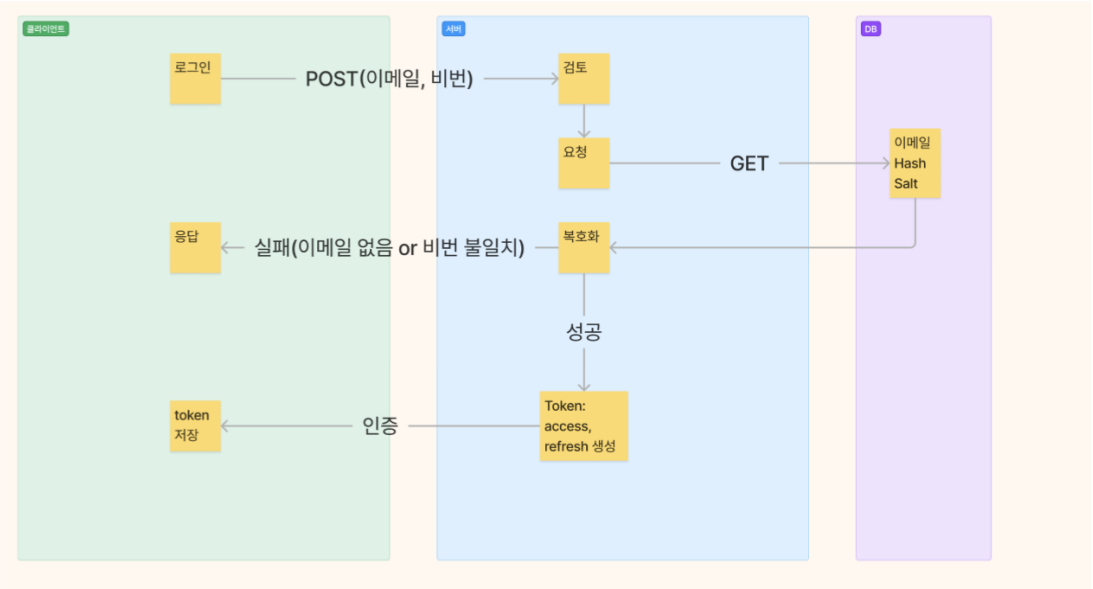
- 클라이언트는 비밀번호, 아이디를 서버에 보냄
- 참고. deno deploy 서버는 https가 기본

```tsx
async function signUp({ request, response }: Context) {
  try {
    // ...
    const input = await request.body().value;

    const document = await mongoAPI.getUser(input.email);
    if (document === undefined) throw Error('document is undefined');

    if (document !== null)
      throw Error(`이미 가입한 아이디입니다. ${document.email}`);

    const passwordSalt = await genSalt(8);
    const passwordHash = await hash(input.password, passwordSalt);

    await mongoAPI.postUser({
      email: input.email,
      passwordHash,
      passwordSalt,
    });

    response.status = 201;
    response.body = null;
  } catch (error) {
    // ...
  }
}
```

- 요청을 받으면 처리하는 함수
- 이메일 중복을 확인
- 없으면 이메일, salt, hash를 저장

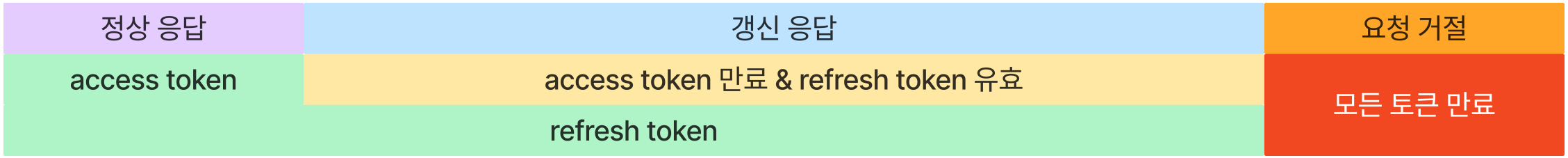
로그인

```
async function signIn({ request, response }: Context) {
  try {
    // ...
    const input = await request.body().value;

    const document = await mongoAPI.getUser(input.email);

    if (document === null) throw Error('이메일이 없습니다.');

    if (!(await compare(input.password, document.passwordHash)))
      throw Error('비밀번호가 일치하지 않습니다.');

    const { jwt: refresh_token } = await generateRefreshToken(document._id);
    const { jwt: access_token } = await generateAccessToken(document._id);

    response.status = 201;
    response.body = {
      success: true,
      access_token,
      refresh_token,
    };
  } catch (error) {
    // ...
  }
}
```



- 참고. setcookie를 못한 이유는 origin이 프론트엔드와 백엔드가 다름
- 임시방편으로 token 2개를 응답함

3가지 기간



- 2개의 토큰이 모두 기간 내일 경우 정상 처리
- access 토큰이 짧고 먼저 말료하면 refresh token 확인
- refresh token이 만료되면 현재 클라이언트는 로그아웃 처리

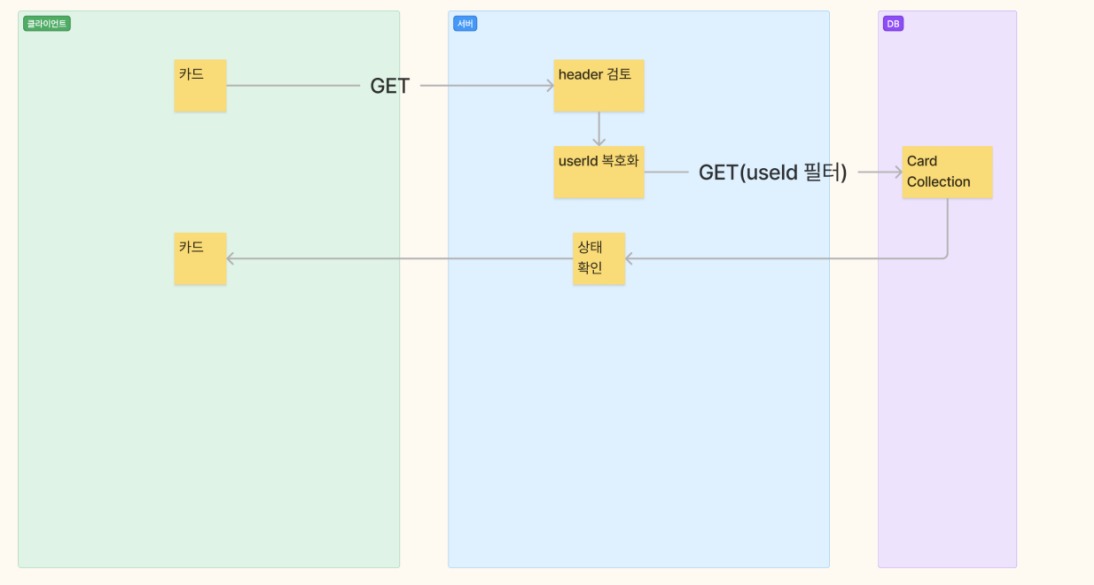
일반요청

```
const authClient: AxiosInstance = axios.create({
  baseURL: BASE_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});

axiosClient.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem(STORAGE_KEY.ACCESS_TOKEN);

    const configCopy = { ...config };
    if (token) configCopy.headers.Authorization = `Bearer ${token}`;
    else throw new Error('token이 없습니다.');

    return configCopy;
  },
  (error) => Promise.reject(error)
);
```

- 본인만 접근 가능한 리소스는 토큰 확인
- `interceptors` 로 요청 직전에 확인

```
const authMiddleware: Middleware = async (
  { request, response, state },
  next
) => {
  try {
    const accessToken = request.headers.get('Authorization');
    if (!accessToken || !accessToken.startsWith('Bearer '))
      throw new BadRequestError('Bad Request');

    const userId = await convertTokenToUserId(accessToken.split(' ')[1]);
    if (!userId) throw new AuthorizationError('expired');

    state.userId = userId;
    await next();
  } catch (error) {
    if (error instanceof BadRequestError) {
      response.status = 400;
      // ...
    }
    if (error instanceof AuthorizationError) {
      response.status = 401;
      // ...
    }
    response.status = 406;
    // ...
  }
};
```

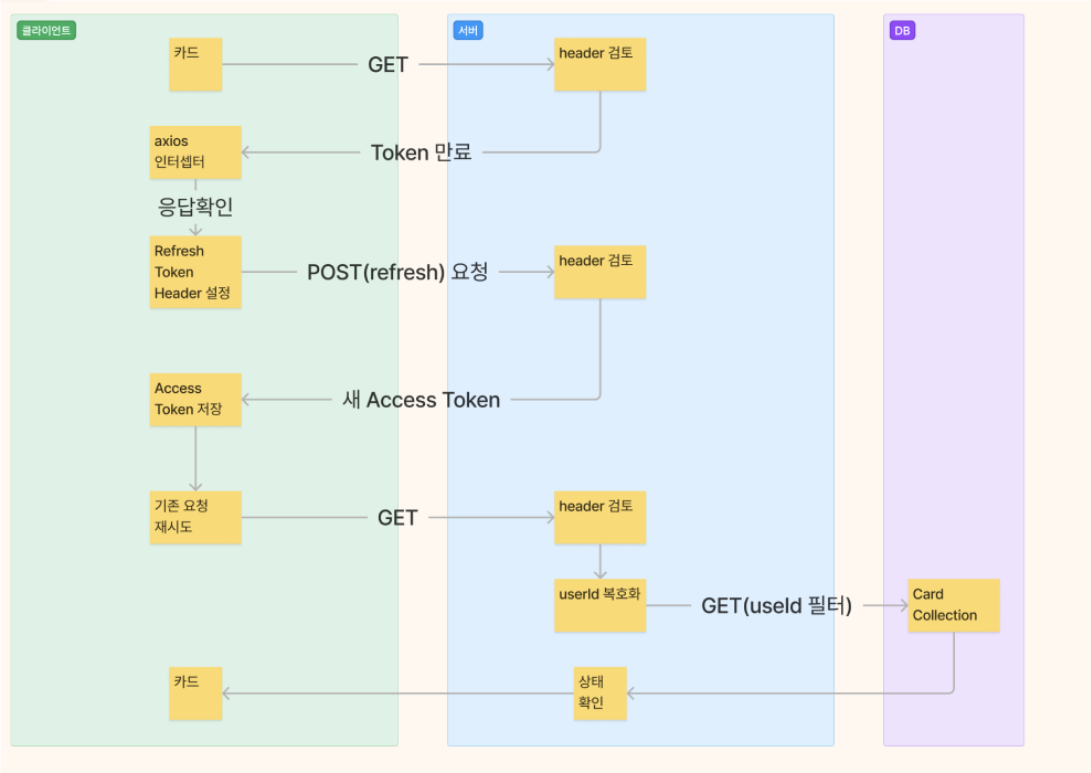
- JWT 토큰을 복호화하면 user 테이블의 id를 획득
- userId에 user테이블의 id를 할당
  - 다음에 실행할 함수에게 전달




```
async function getCards({ response, state }: Context) {
  try {
    const userId = state.userId ?? '';

    response.status = 200;
    response.body = await mongoAPI.getCards(userId);
  } catch (error) {
    response.status = 400;
    response.body = {
      success: false,
      msg: `${error}`,
    };
  }
}
```

- state에 저장된 userId 읽기
- userId로 DB에 필터 요청

갱신



- access는 만료 & refresh는 유효
- i. access 만료 상태로 요청
- ii. 서버는 만료응답
- iii. 클라이언트는 access 만료 확인
- iv. 클라이언트는 access 갱신 요청
- v. 서버는 refresh 확인
- vi. 새 access 토큰 응답
- vii. 클라이언트 동일 요청 재시도
- viii. 서버 정상 응답

토큰 갱신 서버측

```
const authMiddleware: Middleware = async (
  { request, response, state },
  next
) => {
  try {
    const accessToken = request.headers.get('Authorization');
    if (!accessToken || !accessToken.startsWith('Bearer '))
      throw new BadRequestError('Bad Request');

    const userId = await convertTokenToUserId(accessToken.split(' ')[1]);
    if (!userId) throw new AuthorizationError('expired');

    state.userId = userId;
    await next();
  } catch (error) {
    if (error instanceof BadRequestError) {
      response.status = 400;
      // ...
    }
    if (error instanceof AuthorizationError) {
      response.status = 401;
      // ...
    }
    response.status = 406;
    // ...
  }
};
```

- convertTokenToUserId는 만료하면 토큰 검증에서 null값을 반환
- 만료를 확인하고 클라이언트에게 401 status code와 메시지로 응답

토큰 검사

```
const PUBLIC_KEY = JSON.parse(
  Deno.env.get('PUBLIC_KEY') || config()['PUBLIC_KEY']
);

const privateKey = await crypto.subtle.importKey(
  'jwk',
  PUBLIC_KEY,
  { name: 'HMAC', hash: { name: 'SHA-512' } },
  true,
  ['sign', 'verify']
);

async function convertTokenToUserId(jwt: string, key = privateKey) {
  try {
    const { sub: userId } = await verify(jwt, key);
    return userId;
  } catch (_error) {
    return null;
  }
}
```

- PUBLIC_KEY를 `.env` 에 저장
- 클라우드 서버의 재가동으로 토큰값 변경 방지
- 재가동해도 동일한 `privateKey` 생성

토큰 갱신 클라이언트 측

```
axiosClient.interceptors.response.use(
  (res) => res,
  async (err) => {
    const {
      config,
      response: { status },
    } = err;

    if (config.url === API_URLS.REFRESH || status !== 401 || config.sent)
      return Promise.reject(err);

    config.sent = true;
    const accessToken = await refreshAccessAPI();

    if (accessToken) config.headers.Authorization = `Bearer ${accessToken}`;
    return axiosClient(config);
  }
);
```

- `interceptors`로 응답 전에 로직 처리
- 요청 실패를 감청
- `Promise.reject(err)`이 반환값이면 호출자가 알아서 예외 처리
- 401 status code가 아닐 때, config.sent로 갱신 재요청 할 때, 갱신 요청이 실패할 때 중단
- 성공시에는 `axiosClient(config)`가 갱신된 토큰으로 재요청

```
type ResType = {
  success: boolean;
  access_token: string;
};

async function refreshAccessAPI() {
  try {
    const sessionToken = sessionStorage.getItem(STORAGE_KEY.SESSION_TOKEN);
    if (!sessionToken) throw Error('sessionToken');

    const {
      data: { access_token },
    } = await authClient.post<ResType>(API_URLS.REFRESH, null, {
      headers: {
        Authorization: `Bearer ${sessionToken}`,
      },
    });

    localStorage.setItem(STORAGE_KEY.ACCESS_TOKEN, `${access_token}`);

    return access_token;
  } catch (error) {
    localStorage.removeItem(STORAGE_KEY.ACCESS_TOKEN);
    sessionStorage.removeItem(STORAGE_KEY.SESSION_TOKEN);
    redirect(ROUTE_PATHS.SIGN_IN);
    if (error instanceof AxiosError) return error.response?.data;
  }
}
```

갱신 클라이언트

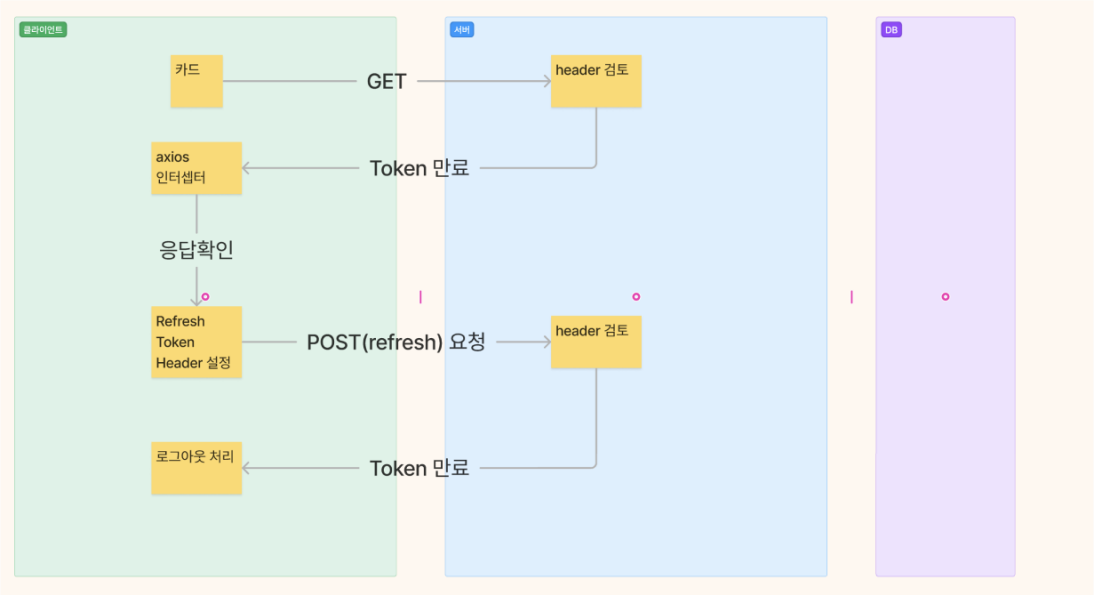
- sessionStorage를 확인하고 토큰을 교체
- 갱신할 토큰을 받아 `access_token`을 `localStorage`에 저장

```
async function refreshUserAccessToken({ request, response }: Context) {
  try {
    const refreshToken = request.headers.get('Authorization');
    if (!refreshToken || !refreshToken.startsWith('Bearer '))
      throw new Error('Bad Request');

    const { accessToken, success } = await refreshAccessToken(
      refreshToken.split(' ')[1]
    );
    if (!accessToken || !success) throw new Error('expired');

    response.status = 200;
    response.body = {
      success: true,
      access_token: accessToken,
    };
  } catch (error) {
    response.status = 400;
    response.body = {
      success: false,
      msg: `${error}`,
    };
  }
}
```

갱신 서버

- refresh 토큰을 검증하고 만료되면 Error
- 유효하면 갱신된 token을 응답

만료




```
async function refreshAccessAPI() {
  try {
    const sessionToken = sessionStorage.getItem(STORAGE_KEY.SESSION_TOKEN);
    if (!sessionToken) throw Error('sessionToken');

    const {
      data: { access_token },
    } = await authClient.post<ResType>(API_URLS.REFRESH, null, {
      headers: {
        Authorization: `Bearer ${sessionToken}`,
      },
    });

    localStorage.setItem(STORAGE_KEY.ACCESS_TOKEN, `${access_token}`);

    return access_token;
  } catch (error) {
    localStorage.removeItem(STORAGE_KEY.ACCESS_TOKEN);
    sessionStorage.removeItem(STORAGE_KEY.SESSION_TOKEN);
    redirect(ROUTE_PATHS.SIGN_IN);
    if (error instanceof AxiosError) return error.response?.data;
  }
}
```

- 토큰을 모두 비우고 리다이렉팅으로 로그아웃 처리

프로젝트 FAQ

- React Query에 Suspense를 적용 안한 이유?
- 현재 React Query 버전에서 experimental

error는 loader에서 하는데 catch는 어떻게 컴포넌트에서 하는가?

```
import queryClient from '@/libs/queryClient';

export const cardLoader = () => async () => {
  const query = () => ({
    queryKey: ['cards'],
    queryFn: getCardsAPI,
    staleTime: 5000,
  });

  try {
    return (
      queryClient.getQueryData<Card[]>(query.queryKey) ??
      (await queryClient.fetchQuery(query))
    );
  } catch (error) {
    queryClient.invalidateQueries({ queryKey: query.queryKey });
    return [];
  }
};
```

- 사실 catch하고 빈 배열을 캐싱함
- loader에서 예외처리가 됨
  - loader의 통신은 catch가 된 것
- React Query는 통신 실패를 어떻게 알 수 있는가?

React Query는 통신 실패를 어떻게 알 수 있는가? 1

- React Query는 통신의 성공/실패를 대입한 함수로 파악
- API를 호출하는 catch하고 다시 throw하면
- React Query의 `return` 의 `error` 속성으로 접근 가능

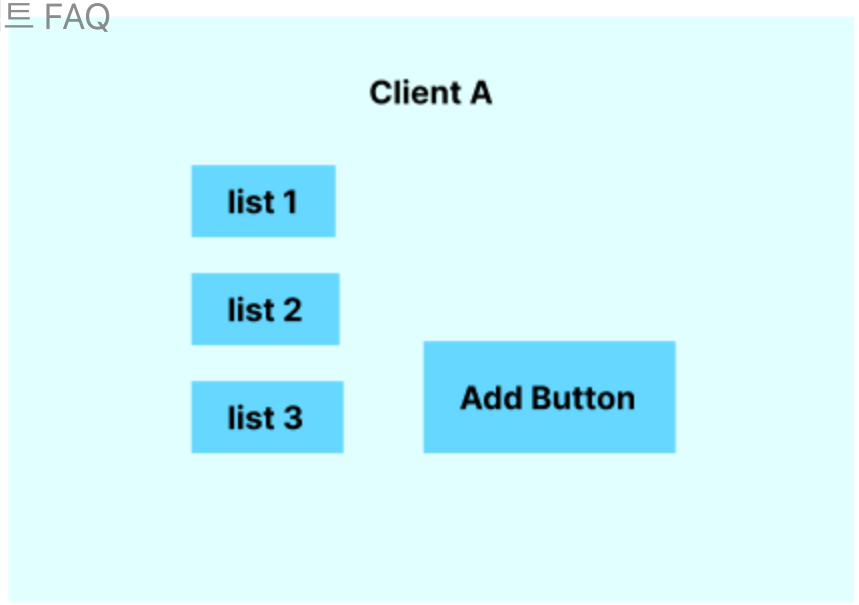
```
type CardResType = { documents: Card[] };

async function getCardsAPI() {
  try {
    const res = await axiosClient.get<CardResType>(API_URLS.CARDS);
    return res.data.documents;
  } catch (error) {
    if (axios.isAxiosError<ErrorResponse>(error)) throw error.response?.data;
  }
  return [];
}
```

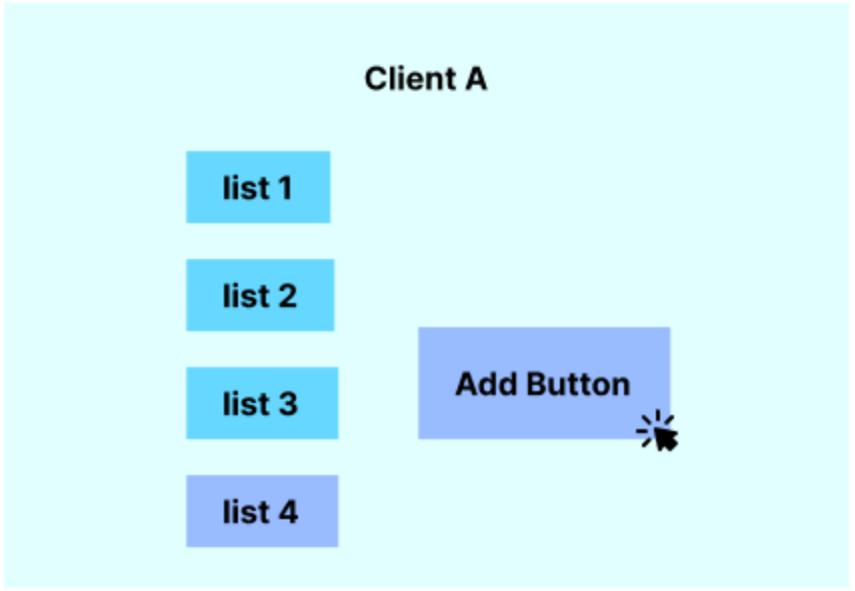
React Query는 통신 실패를 어떻게 알 수 있는가? 2

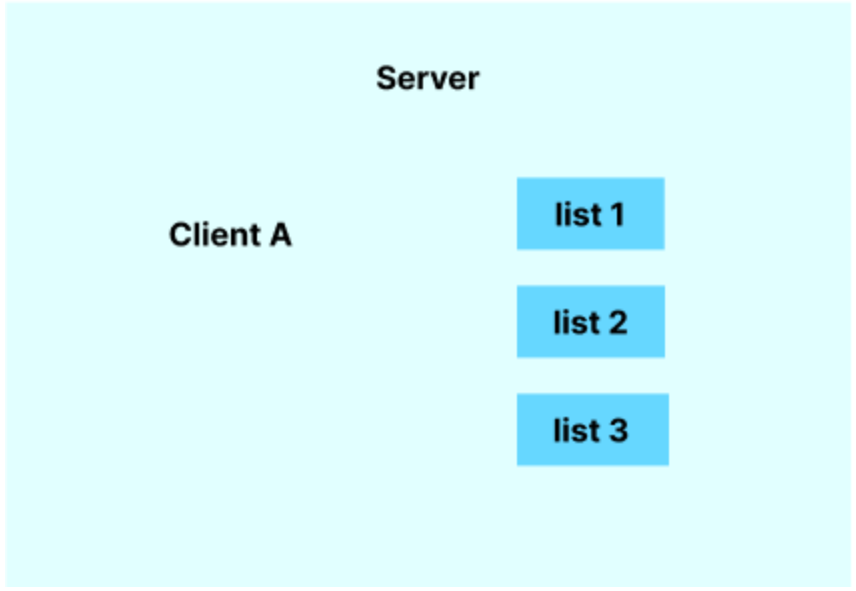
- React-Query는 통신을 대신해주는 라이브러리가 아닌 통신과 관련된 작업들을 추상화 해주는 라이브러리
- 추상화의 대상은 서버상태(A.K.A. data caching, data fetching state management)
- API에서 통신 실패를 throw만 하면 React-Query는 빈 배열이 캐싱되어도 통신이 실패했다는 것을 알 수 있음
  - 캐싱여부와 통신의 성공 실패여부는 다름(캐싱은 수동으로 해도 통신은 실패할 수 있음)
- 통신은 querykey와 API 함수로 묶었기 때문에 실패를 공유하는 querykey로 전달할 수 있음
  - 참고. querykey로 캐시 자원을 접근하기 때문에 하위 컴포넌트에서 같은 쿼리키로 같은 캐시를 접근할 수 있음
- React-Query가 예외처리를 throw로 처리하는 이유는 에러를 재가공할 수 있게 제공

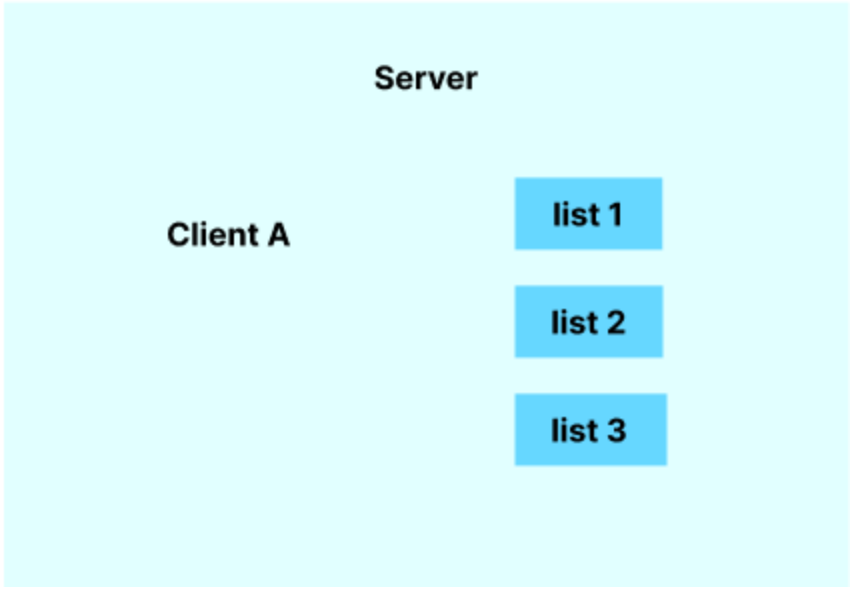
Optimistic Update란?

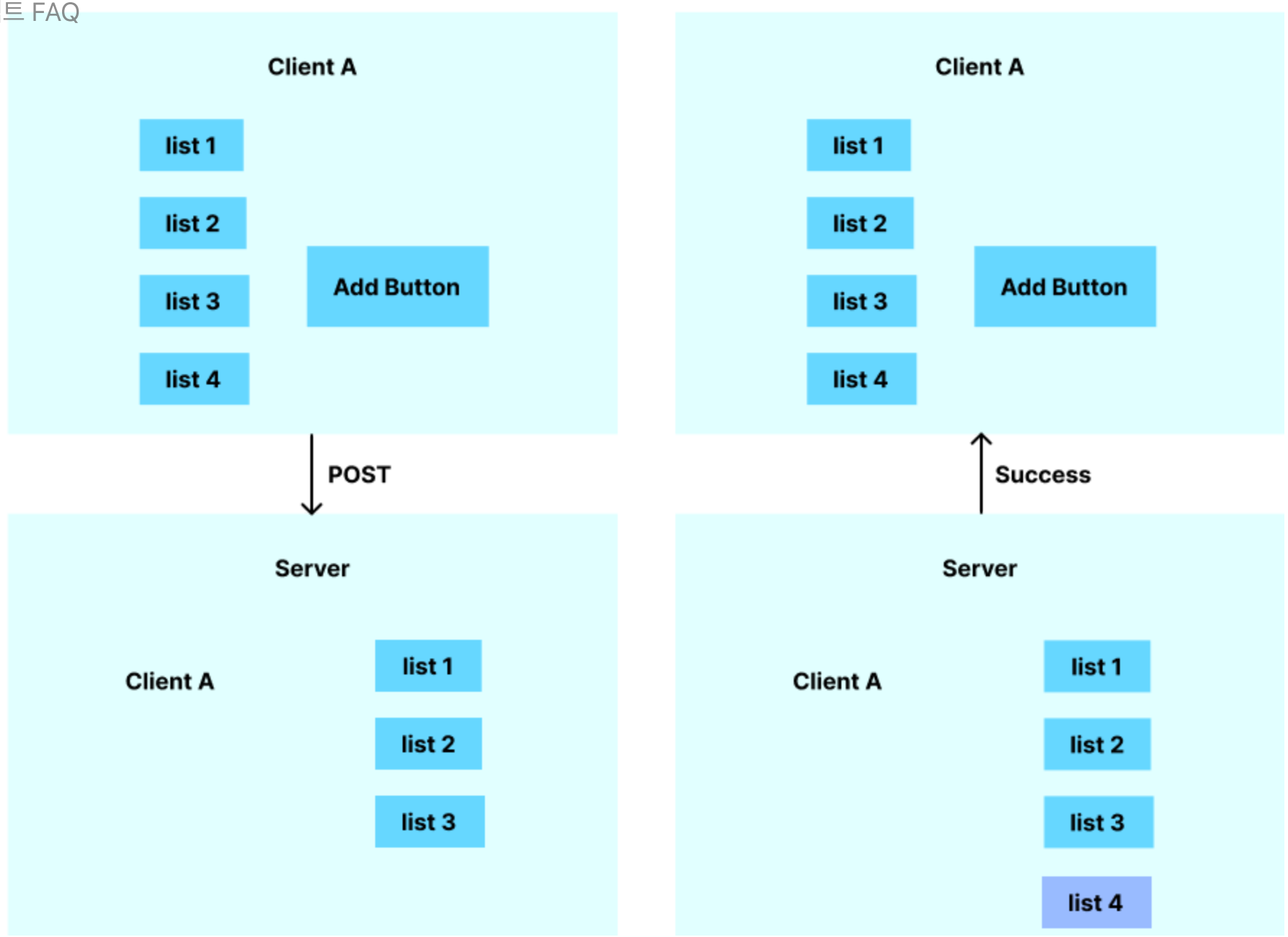
- 낙관적으로 성공할 것이라고 가정하고 클라이언트의 현재 리소스를 갱신하고 서버에 쓰기 요청을 보내는 것
- 다음은 Optimistic Update 예시

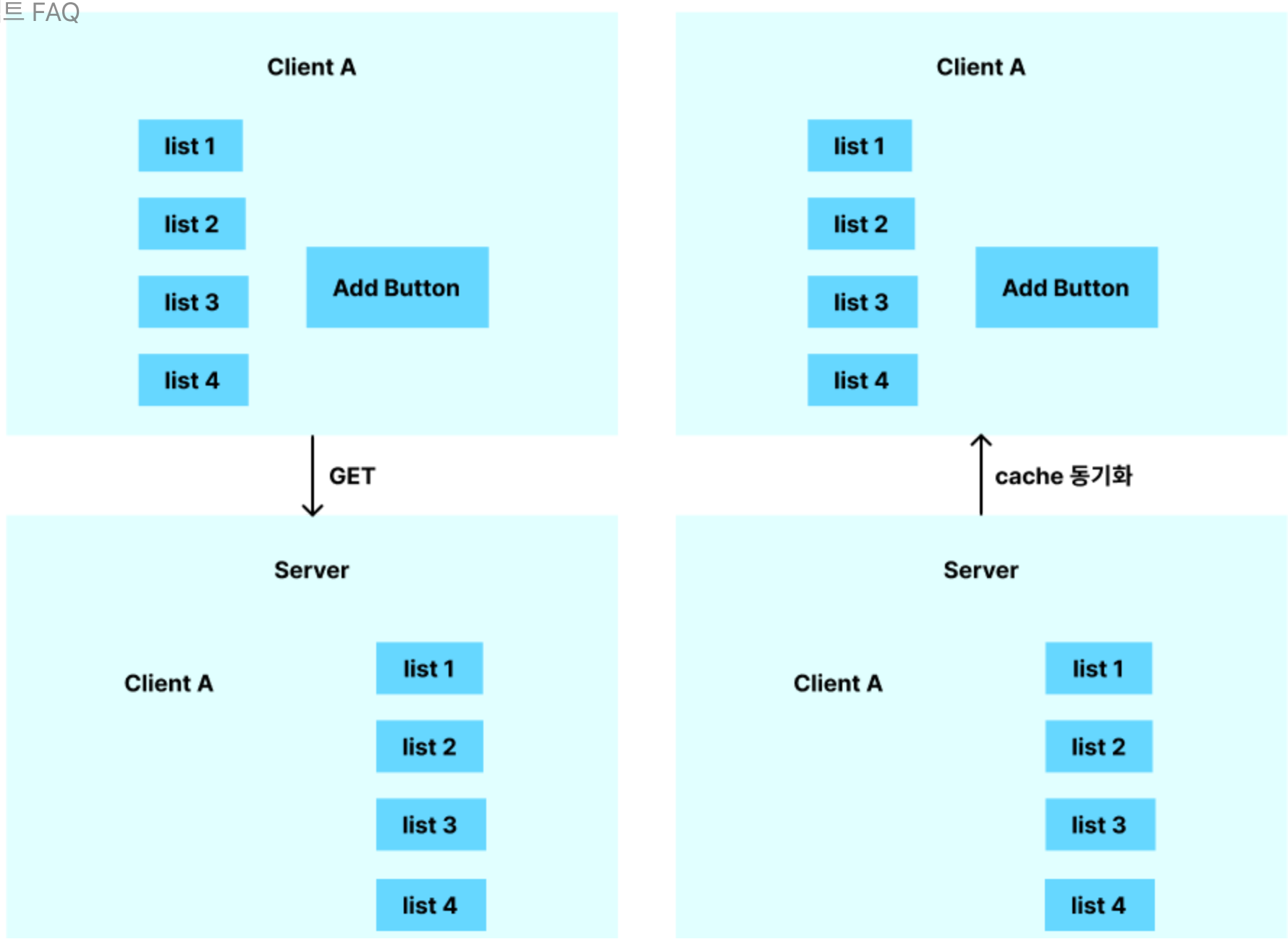

Client A
list 1
list 2
list 3
Add Button


Client A
list 1
list 2
list 3
list 4
Add Button


Server
Client A
list 1
list 2
list 3

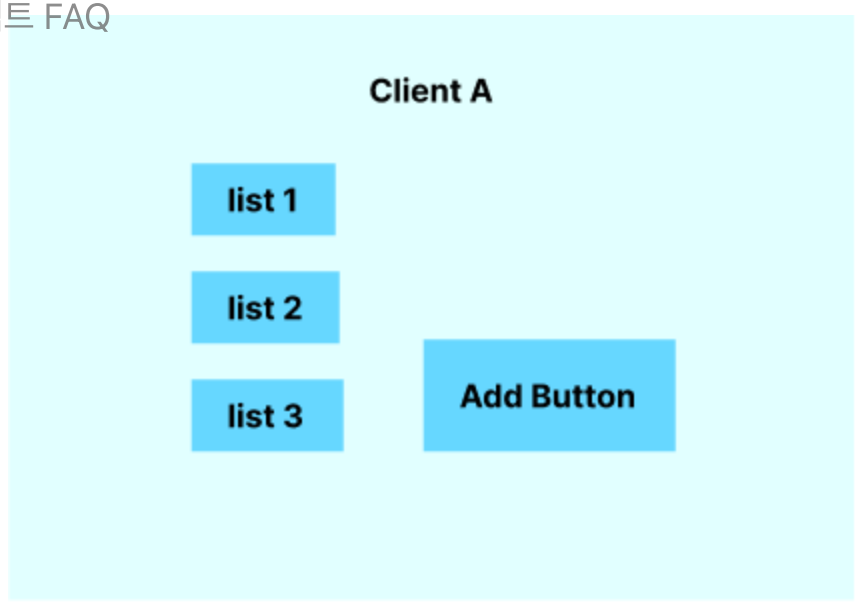

Server
Client A
list 1
list 2
list 3

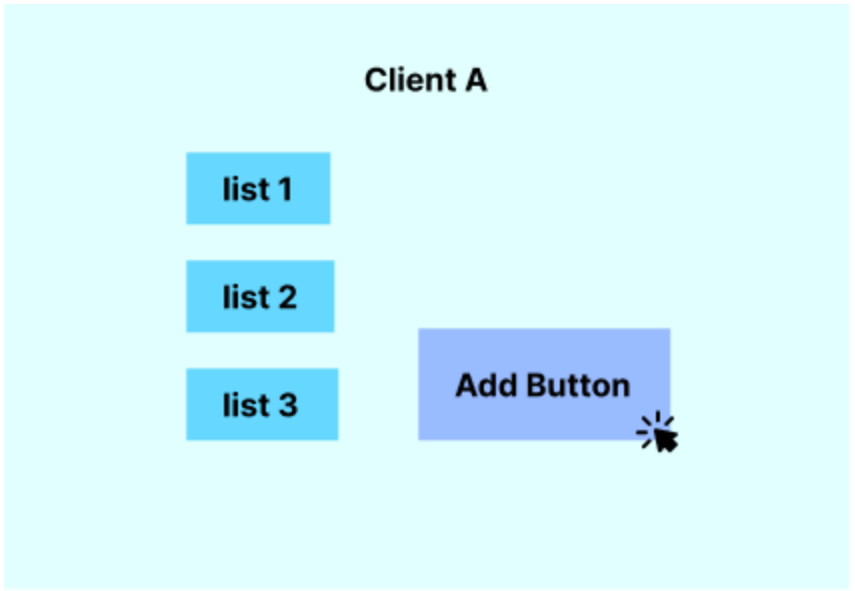

Client A
list 1
list 2
list 3
list 4
Add Button
POST
Server
Client A
list 1
list 2
list 3
Client A
list 1
list 2
list 3
list 4
Add Button
Success
Server
Client A
list 1
list 2
list 3
list 4

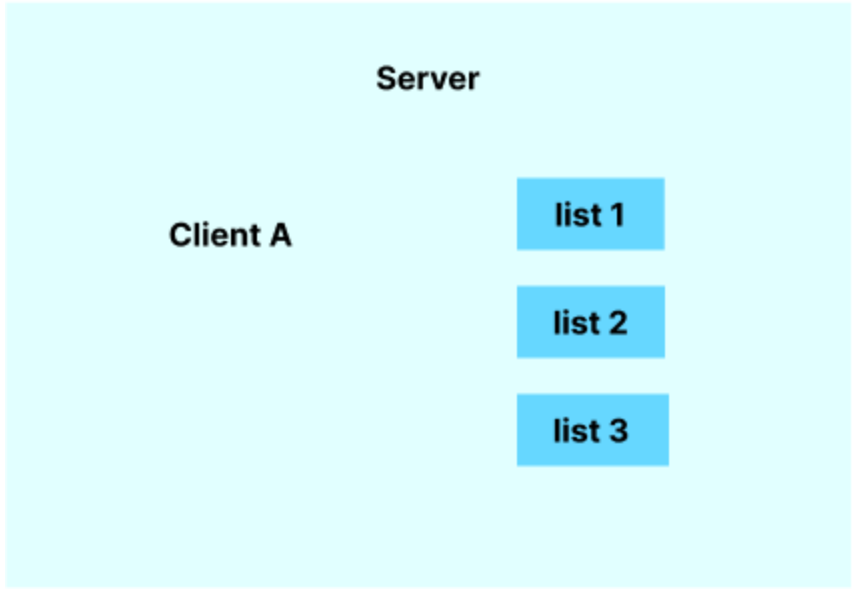

Client A
list 1
list 2
list 3
list 4
Add Button
GET
Server
Client A
list 1
list 2
list 3
list 4
Client A
list 1
list 2
list 3
list 4
Add Button
cache 동기화
Server
Client A
list 1
list 2
list 3
list 4

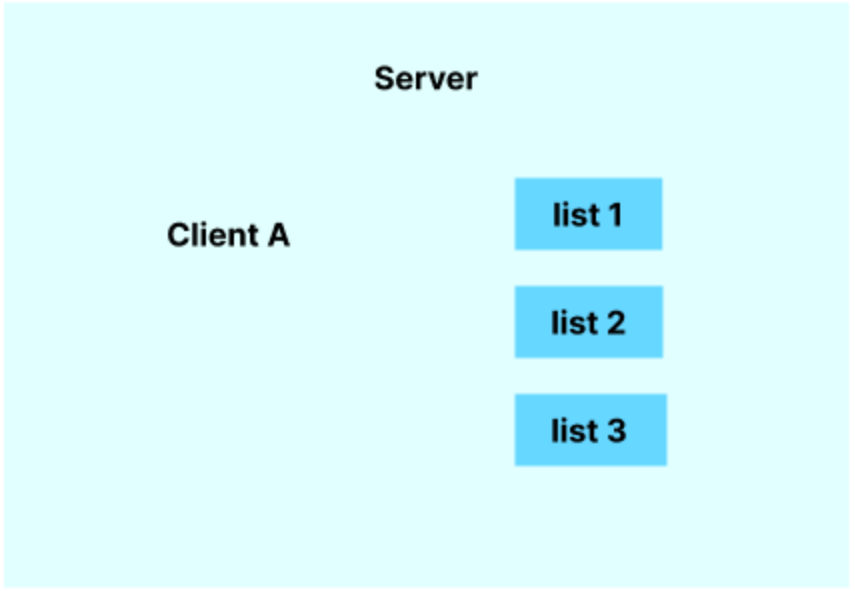
참고. Optimistic Update Best Practice

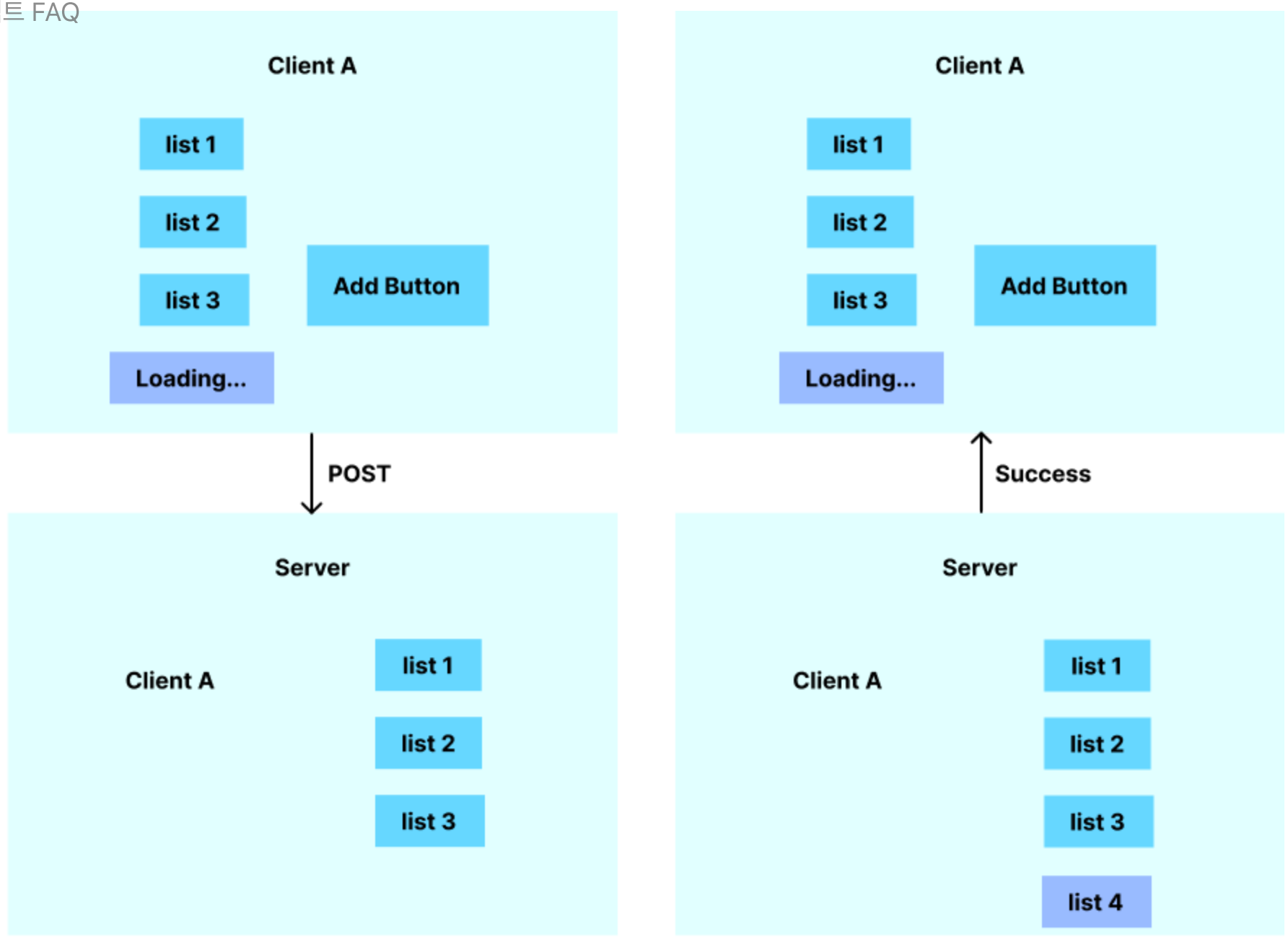
- tkdodo가 적용을 권장할 때랑 자제할 경우 인터뷰에서 알려줌
- 서버 저장해야 하지만 UI 상 블러킹이 없어야 하고 error가 발생해도 치명적이지 않은 경우
    - ex: 설정 toggle, 좋아요
- 하지만 대부분의 경우 `mutation` 의 `onSettled` 에 `query invalidation` 으로 충분
- 다음은 Optimistic Update가 아닌 일반 통신

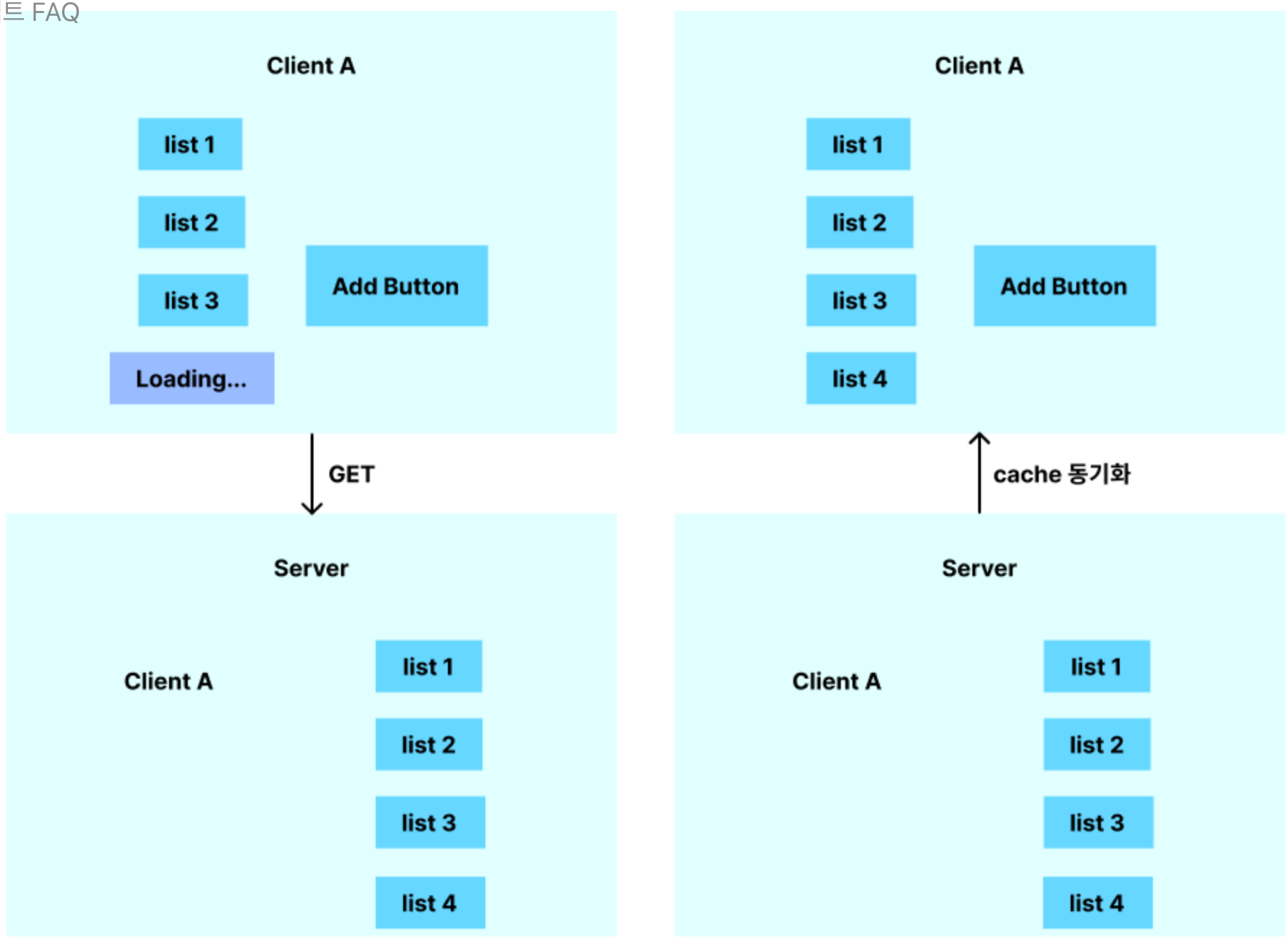

Client A
list 1
list 2
list 3
Add Button


Client A
list 1
list 2
list 3
Add Button


Server
Client A
list 1
list 2
list 3


Server
Client A
list 1
list 2
list 3


Client A
list 1
list 2
list 3
Add Button
Loading...
POST
Server
Client A
list 1
list 2
list 3
Client A
list 1
list 2
list 3
Add Button
Loading...
Success
Server
Client A
list 1
list 2
list 3
list 4


Client A
list 1
list 2
list 3
Add Button
Loading...
GET
Server
Client A
list 1
list 2
list 3
list 4
Client A
list 1
list 2
list 3
Add Button
list 4
cache 동기화
Server
Client A
list 1
list 2
list 3
list 4

Webp 적용할 때 크로스 브라우징 이슈

- picture 태그에 img태그 fallback을 설정

```
function Landing() {
  return (
    <picture>
      <source
        type="image/webp"
        media="all and (min-width: 1320px)"
        src={flashCardWEBP}
        srcSet={flashCardWEBP}
      />
      <source
        type="image/jpeg"
        media="all and (min-width: 1320px)"
        src={flashCardJPG}
        srcSet={flashCardJPG}
      />
      <img srcSet={flashCardJPG} alt="flash card picture" loading="lazy" />
    </picture>
  );
}
```

감사합니다.