




7장 비동기 호출

API 요청

7.1.1 fetch로 API 요청하기

! 비동기 호출 코드는 변경 요구에 취약하다. 새로운 API 요청 정책이 추가될 때마다 계속해서 비동기 호출 코드를 수정해야 하는 번거로움이 발생한다.

7.1.2 서비스 레이어로 분리하기

 비동기 호출 코드는 컴포넌트 영역에서 분리되어 다른 영역 **서비스 레이어** 에서 처리되어야 한다.

7.1.3 Axios 활용하기

```
import axios, { AxiosInstance, AxiosPromise } from "axios";

export type FetchCartResponse = unknown;
export type PostCartRequest = unknown;
export type PostCartResponse = unknown;

export const apiRequester: AxiosInstance = axios.create({
  baseURL: "https://api.baemin.com",
  timeout: 5000,
});

export const fetchCart = (): AxiosPromise<FetchCartResponse> =>
  apiRequester.get<FetchCartResponse>("cart"); // GET https://api

export const postCart = (
  postCartRequest: PostCartRequest
```

```
) : AxiosPromise<PostCartResponse> =>
  apiRequester.post<PostCartResponse>("cart", postCartRequest);
```

API Entry가 2개 이상일 경우 각 서버의 기본 URL을 호출하도록 `orderApiRequester`, `orderCartApiRequester` 같이 2개 이상의 API 요청을 처리하는 인스턴스를 따로 구성해야 한다.

```
import axios, { AxiosInstance } from "axios";

const defaultConfig = {}; // 모든 API 요청자가 공통으로 사용할 기본 설정

// 특별한 baseURL이 지정되지 않은 일반적인 API 호출에 사용
const apiRequester: AxiosInstance = axios.create(defaultConfig);

//
const orderApiRequester: AxiosInstance = axios.create({
  baseURL: "https://api.baemin.or/",
  ...defaultConfig,
});
const orderCartApiRequester: AxiosInstance = axios.create({
  baseURL: "https://cart.baemin.order/",
  ...defaultConfig,
});
```

7.1.4 Axios 인터셉터 사용하기

requester별로 다른 헤더를 설정해줘야 하는 로직이 필요할 수 있다.

인터셉터를 사용하여 requester에 따라 비동기 호출 내용을 추가해서 처리한다.

```
import axios, { AxiosInstance, AxiosRequestConfig, AxiosResponse } from "axios";

const getUserToken = () => "";
const getAgent = () => "";
const getOrderClientToken = () => "";
const orderApiBaseUrl = "";
```

```

const orderCartApiBaseUrl = "";
const defaultConfig = {};
const httpErrorHandler = () => {};

const apiRequester: AxiosInstance = axios.create({
  baseUrl: "https://api.baemin.com",
  timeout: 5000,
});

const setRequestDefaultHeader = (requestConfig: AxiosRequestConfig): AxiosRequestConfig => {
  const config = requestConfig;
  config.headers = {
    ...config.headers,
    "Content-Type": "application/json;charset=utf-8",
    user: getUserToken(),
    agent: getAgent(),
  };
  return config;
};

const setOrderRequestDefaultHeader = (requestConfig: AxiosRequestConfig): AxiosRequestConfig => {
  const config = requestConfig;
  config.headers = {
    ...config.headers,
    "Content-Type": "application/json;charset=utf-8",
    "order-client": getOrderClientToken(),
  };
  return config;
};

// `interceptors` 기능을 사용해 header를 설정하는 기능을 넣거나 에러를 처리
apiRequester.interceptors.request.use(setRequestDefaultHeader);

const orderApiRequester: AxiosInstance = axios.create({
  baseUrl: orderApiBaseUrl,
  ...defaultConfig,
});
// 기본 apiRequester와는 다른 header를 설정하는 `interceptors`
orderApiRequester.interceptors.request.use(setOrderRequestDefaultHeader);
// `interceptors`를 사용해 httpError 같은 API 에러를 처리할 수도 있다

```

```

orderApiRequester.interceptors.response.use(
  (response: AxiosResponse) => response,
  httpErrorHandler
);

const orderCartApiRequester: AxiosInstance = axios.create({
  baseUrl: orderCartApiBaseUrl,
  ...defaultConfig,
});
orderCartApiRequester.interceptors.request.use(setRequestDefault

```

✓ 빌더 패턴

요청 옵션에 따라 다른 인터셉터를 만들기 위해 **빌더 패턴** 을 추가하여 APIBuilder 같은 클래스 형태로 구성하기도 한다.

빌더 패턴 : 객체 생성을 더 편리하고 가동성 있게 만들기 위한 디자인 패턴 중 하나. 주로 복잡한 객체의 생성을 단순화하고, 객체 생성 과정을 분리하여 객체를 조립하는 방법을 제공한다.

기본 API 클래스로 실제 호출 부분을 구성하고, 해당 API를 호출하기 위한 Wrapper를 빌더 패턴으로 만든다.

▼ 기본 API 클래스

```

import axios, { AxiosPromise } from "axios";

// 임시 타이핑
export type HTTPMethod = "GET" | "POST" | "PUT" | "DELETE";
export type HTTPHeaders = any;
export type HTTPParams = unknown;

class API {
  // HTTP 요청에 필요한 기본 속성들 정의
  readonly method: HTTPMethod;
  readonly url: string;
  baseUrl?: string;

```

```

headers?: HTTPHeaders;
params?: HTTPParams;
data?: unknown;
timeout?: number;
withCredentials?: boolean;

constructor(method: HTTPMethod, url: string) {
  this.method = method;
  this.url = url;
}

// 실제 HTTP 요청 수행
call<T>(): AxiosPromise<T> {
  const http = axios.create();
  // 만약 `withCredential`이 설정된 API라면 아래 같이 인터셉터를 추가
  if (this.withCredentials) {
    http.interceptors.response.use(
      (response) => response,
      (error) => {
        if (error.response && error.response.status === 401)
          /* 에러 처리 진행 */
        }
        return Promise.reject(error);
      }
    );
  }
  return http.request({ ...this }); // 설정된 모든 옵션으로 요청
}

export default API;

```

▼ 빌더 패턴을 사용하여 API 객체를 생성하는 APIBuilder 클래스



앞에 붙은 밑줄(`_`) : 일종의 네이밍 컨벤션으로, 이 변수가 "private"하다는 것을 표시하는 방법



`static` 메서드 : 클래스의 인스턴스를 생성하지 않고도 호출할 수 있는 메서드

1. `static get` 이 호출되면
2. 내부적으로 `new APIBuilder("GET", url)` 을 실행
3. 이때 생성자가 실행됨

```
import API, { HTTPHeaders, HTTPMethod, HTTPParams } from "../api"

const apiHost = ""

class APIBuilder {
  private _instance: API;

  constructor(method: HTTPMethod, url: string, data?: unknown) {
    this._instance = new API(method, url);
    this._instance.baseURL = apiHost;
    this._instance.data = data;
    this._instance.headers = {
      "Content-Type": "application/json; charset=utf-8",
    };
    this._instance.timeout = 5000;
    this._instance.withCredentials = false;
  }

  // API 인스턴스를 만들고 설정하는 빌더
  static get = (url: string) => new APIBuilder("GET", url);
  static put = (url: string, data: unknown) => new APIBuilder("PUT", url, data);
  static post = (url: string, data: unknown) => new APIBuilder("POST", url, data);
  static delete = (url: string) => new APIBuilder("DELETE", url);

  // 설정 메서드들 : 메서드 체이닝 실행 - _instance의 설정을 변경
  baseURL(value: string): APIBuilder {
    this._instance.baseURL = value;
    return this;
  }

  headers(value: HTTPHeaders): APIBuilder {
```

```

        this._instance.headers = value;
        return this;
    }

    timeout(value: number): APIBuilder {
        this._instance.timeout = value;
        return this;
    }

    params(value: HTTPParams): APIBuilder {
        this._instance.params = value;
        return this;
    }

    data(value: unknown): APIBuilder {
        this._instance.data = value;
        return this;
    }

    withCredentials(value: boolean): APIBuilder {
        this._instance.withCredentials = value;
        return this;
    }

    build(): API {
        return this._instance;
    }
}

export default APIBuilder;

```

```

// APIBuilder를 사용하는 코드
import APIBuilder from "../7.1.4-3";

// ex
type Response<T> = { data: T };
type JobNameListResponse = string[];

const fetchJobNameList = async (name?: string, size?: number) =>

```

```
const api = APIBuilder.get("/apis/web/jobs")
  .withCredentials(true) // 이제 401 에러가 나는 경우, 자동으로 에러
  .params({ name, size }) // body가 없는 axios 객체도 빌더 패턴으로
  .build(); // 최종 API 인스턴스 반환
const { data } = await api.call<Response<JobNameListResponse>>
return data;
};
```



APIBuilder 클래스의 장단점

장점 : 옵션이 다양한 경우에 인터셉터를 설정값에 따라 적용하고, 필요 X는 인터셉터를 선택적으로 사용할 수 있다.

단점 : 보일러플레이트 코드가 많다.

보일러플레이트 코드 : 어떤 기능을 사용할 때 반복적으로 사용되는 기본적인 코드 ex) API를 호출하기 위한 기본적인 설정과 인터셉터 등을 설정하는 부분

7.1.5 API 응답 타입 지정하기

✓ 같은 서버에서 오는 응답의 형태는 대체로 통일되어 있어서 하나의 **Response** 타입으로 묶을 수 있다.

- Response 타입을 apiRequester 내에서 하면 UPDATE, CREATE 같이 응답이 없을 수 있는 API를 처리하기 까다로워진다. → Response 타입은 apiRequester가 모르게 관리되어야 한다.

▼ apiRequester

```
export const apiRequester: AxiosInstance = axios.create({
  baseURL: "https://api.baemin.com",
  timeout: 5000,
});
```



```
import { AxiosPromise } from "axios";
import {
  FetchCartResponse,
  PostCartRequest,
  PostCartResponse,
  apiRequester,
} from "../7.1.3-1";

export interface Response<T> {
  data: T;
  status: string;
  serverDateTime: string;
  errorCode?: string; // FAIL, ERROR
  errorMessage?: string; // FAIL, ERROR
}

const fetchCart = (): AxiosPromise<Response<FetchCartResponse>>
  apiRequester.get<Response<FetchCartResponse>>("cart");

const postCart = (
  postCartRequest: PostCartRequest
): AxiosPromise<Response<PostCartResponse>> =>
  apiRequester.post<Response<PostCartResponse>>("cart", postCart
```

✓ ex) forPass는 서버에서 언제든지 구조가 바뀔 수 있는 데이터로 대부분 로그용으로 사용된다. → `unknown`

```
interface response {
  data: {
    cartItems: CartItem[];
    forPass: unknown; // 서버에서 여러 용도로 보내는 추가 데이터
  };
}
```

forPass 안에 프론트 로직에서 사용해야 하는 값이 있다면, 그 값에 대해서만 타입을 선언한 다음에 사용하는 게 좋다.

```
type ForPass = {
  type: "A" | "B" | "C";
```

```
};
```

```
const isTargetValue = () => (data.forPass as ForPass).type === "
```

7.1.6 뷰 모델 사용하기

API 응답은 변환 가능성이 크기 때문에 뷰 모델을 사용하여 API 변경에 따른 범위를 한정해줘야 한다.



뷰 모델 : API 응답과 컴포넌트 사이에 위치하는 중간 계층

- 데이터를 변환하고 일관된 인터페이스를 제공

특정 객체 리스트를 조회하여 리스트 각각의 내용과 리스트 전체 길이 등을 보여줘야 하는 화면

▼ fetchList API

```
// 해당 리스트를 조회하는 fetchList API

interface ListResponse {
  items: ListItem[];
}

const fetchList = async (filter?: ListFetchFilter): Promise<ListResponse> => {
  const { data } = await api
    .params({ ...filter })
    .get("/apis/get-list-summaries")
    .call<Response<ListResponse>>();

  return { data };
};
```

▼ fetchList API를 사용하는 컴포넌트

```
// 해당 API를 사용하는 경우
// 실제 비동기 함수는 컴포넌트 내부에서 직접 호출하지 X
```

```

const ListPage: React.FC = () => {
  const [totalCount, setTotalItemCount] = useState(0);
  const [items, setItems] = useState<ListItem[]>([]);

  useEffect(() => {
    // 예시를 위한 API 호출과 then 구문
    fetchList(filter).then(({ items }) => {
      setTotalItemCount(items.length);
      setItems(items);
    });
  }, []);

  return (
    <div>
      <Chip label={totalCount} />
      <Table items={items} />
    </div>
  );
};

```

! API 응답의 `items` 인자를 좀 더 정확한 개념으로 나타내기 위해 `jobItems` 나 `cartItems` 같은 이름으로 수정하면, 이 API 응답을 직접 사용하는 모든 컴포넌트를 수정해야 한다.



뷰 모델

```

// 기존 ListResponse에 더 자세한 의미를 담기 위한 변화 : 두 개의 새로운
interface JobListItemResponse {
  name: string;
}

interface JobListResponse {
  jobItems: JobListItemResponse[];
}

// 뷰모델
class JobList {
  readonly totalCount: number;
  readonly items: JobListItemResponse[];

  constructor({ jobItems }: JobListResponse) {

```

```

        this.totalItemCount = jobItems.length;
        this.items = jobItems;
    }
}

const fetchJobList = async (filter?: ListFetchFilter): Promise<JobList> => {
    const { data } = await api
        .params({ ...filter })
        .get("/apis/get-list-summaries")
        .call<Response<JobListResponse>>();

    return new JobList(data);
};

```



▼ 뷰모델 실제 사용 예시

```
const JobListPage: React.FC = () => {
  const [jobList, setJobList] = useState<JobList | null>(null);

  useEffect(() => {
    fetchJobList(filter).then(setJobList);
  }, []);

  if (!jobList) return null;

  return (
    <div>
      <Chip label={jobList.totalItemCount} /> // 뷰 모델
      <Table items={jobList.items} />         // 뷰 모델
    </div>
  );
};
```

서버에서 응답 구조가 이렇게 변경되었다.

```
interface JobListResponse {
  positions: JobListItemResponse[]; // jobItems -> pos
  total: number;                    // 새로운 필드 추가
}
```

뷰 모델이 없다면 모든 컴포넌트를 수정해야 하지만, 뷰 모델이 있다면 JobList 클래스만 수정하면 된다.


```
class JobList {
  readonly totalItemCount: number;
  readonly items: JobListItemResponse[];

  constructor({ positions, total }: JobListResponse) {
    this.totalItemCount = total;
    this.items = positions; // 변경된 필드명을 여기서 처리
```

```
}
}
```

❗ **뷰모델**의 **추상화 레이어** 추가는 결국 코드를 복잡하게 만들며, 레이어를 관리하고 개발하는 데도 비용이 든다.

ex) JobList 전체 리스트를 위한 뷰 모델 + JobListItem 개별 아이템을 위한 뷰 모델 추가

 API 응답이 바뀌었을 때 클라이언트 코드 수정 비용 줄이면서 도메인 일관성 지키기

- 꼭 필요한 곳에만 뷰 모델을 부분적으로 만들어서 사용하기
- 백엔드와 클라이언트 개발자가 충분히 소통한 다음에 개발하여 API 응답 변화를 최대한 줄이기
- 뷰 모델에 필드를 추가하는 대신에 getter 등의 함수를 추가하여 실제 어떤 값이 뷰 모델에 추가한 값인지 알기 쉽게 하기
 - `totalItemCount` 같이 서버가 내려준 API 응답에는 없는 새로운 필드를 프론트에서 만들어서 사용할 때, '서버가 내려준 응답 ≠ 클라이언트가 실제 사용하는 도메인' 문제점 해결

7.1.7 Superstruct를 사용해 런타임에서 응답 타입 검증하기

Superstruct 라이브러리 : 런타임에 API 응답의 타입 오류를 방지하기 위한 라이브러리

- Superstruct를 사용하여 인터페이스 정의와 JS 데이터의 유효성 검사를 쉽게 할 수 있다.
- Superstruct는 런타임에서의 데이터 유효성 검사를 통해 개발자와 사용자에게 자세한 에러를 보여 주기 위해 고안되었다.

➡ 컴파일 단계가 아닌 **런타임**에서도 적절한 데이터인지를 확인하는 검사가 필요할 때 유용하게 사용할 수 있다.

```
// <공식문서 제공 코드 예시>
import {
  assert,
  is,
  validate,
  object,
  number,
  string,
```

```

    array,
  } from "superstruct";

// Superstruct의 object() 모듈의 반환 결과
const Article = object({
  id: number(), // 숫자
  title: string(), // 문자열
  tags: array(string()), // 문자열 배열
  author: object({ // id라는 숫자를 속성으로 가진 객체 형태의 object
    id: number(),
  }),
});

const data = {
  id: 34,
  title: "Hello World",
  tags: ["news", "features"],
  author: {
    id: 1,
  },
};

assert(data, Article);
is(data, Article);
validate(data, Article);

```



데이터의 유효성 검사를 도와주는 모듈

- 데이터 정보를 담은 data 변수와 데이터 명세를 가진 스키마인 Article을 인자로 받아 데이터가 스키마와 부합하는지를 검사한다.

`assert` : 확인

- 유효하지 않을 경우 에러를 던진다.

`is` : ~이다

- 유효성 검사 결과에 따라 boolean 값을 반환한다.

`validate` : 검사하다

- [error, data] 형식의 튜플을 반환한다.

유효하지 않을 경우, 에러 값이 반환되고

유효한 경우, 첫번째 요소로 undefined, 두번째 요소로 data value가 반환된다.



▼ 실제 사용 예시

```
try {
  // 검증 성공 시 그대로 진행
  assert(data, Article);
  console.log("데이터가 유효합니다!");
} catch (error) {
  // 검증 실패 시 에러 처리
  console.error("데이터 구조가 잘못되었습니다:", error);
}

// boolean 검사
if (is(data, Article)) {
  console.log("데이터가 유효합니다!");
}

// 결과값으로 처리
const [error, validData] = validate(data, Article);
if (error) {
  console.error("검증 실패:", error);
} else {
  console.log("검증된 데이터:", validData);
}
```

✓ Superstruct와 TS의 시너지



Infer : 기존 타입 선언 방식과 동일하게 타입을 선언할 수 있다.

```
import { Infer, number, object, string } from "superstruct";

const User = object({
  id: number(),
  email: string(),
  name: string(),
});
```

```
type User = Infer<typeof User>; // Infer 유틸리티를 사용하여 TypeScript
```

- `assert` 를 통해 인자로 받는 `user`가 `User` 타입과 매칭되는지 확인하는 `isUser`

```
type User = { id: number; email: string; name: string };

import { assert } from "superstruct";

function isUser(user: User) {
  assert(user, User);
  console.log("적절한 유저입니다.");
}
```

```
// 1
const user_A = {
  id: 4,
  email: "test@woowahan.email",
  name: "woowa",
};

isUser(user_A); // ✅ "적절한 유저입니다."
```

```
// 2
const user_B = {
  id: 5,
  email: "wrong@woowahan.email",
  name: 4,
};

isUser(user_B); // ❌ 런타임에러 - error TS2345: Argument of type
```

7.1.8 실제 API 응답 시의 Superstruct 활용 사례

```
interface ListItem {
  id: string;
  content: string;
}
```

```


interface ListResponse {
  items: ListItem[];
}
const fetchList = async (filter?: ListFetchFilter): Promise<List
  const { data } = await api
    .params({ ...filter })
    .get("/apis/get-list-summaries")
    .call<Response<ListResponse>>();

  return { data };
};

```

fetchList 함수를 호출했을 때 id와 content가 담긴 ListItem 타입의 배열이 오기를 기대하지만, 실제 서버 응답 형식은 다를 수 있다.

! TS만으로는 실제 서버 응답의 형식과 명시한 타입이 일치하는지 확인할 수 없다.

 **Superstruct** → TS로 선언한 타입과 실제 런타임에서의 데이터 응답값을 매칭하여 유효성 검사를 한다.

```

import { assert } from "superstruct";

// listItems 배열 목록을 받아와 데이터가 ListItem 타입과 동일한지 확인하고
function isListItem(listItems: ListItem[]) {
  listItems.forEach((listItem) => assert(listItem, ListItem));
}

```



▼ `assert(value: unknown` 검증하고자 하는 값, `schema: Struct` value가 만족해야 하는 구조를 정의하는 Struct 객체)

```
import { object, string, assert } from "superstruct";

// 먼저 Superstruct 스키마 정의
const ListItemSchema = object({
  id: string(),
  content: string()
});

// TypeScript 타입도 함께 정의
interface ListItem {
  id: string;
  content: string;
}

interface ListResponse {
  items: ListItem[];
}

const fetchList = async (filter?: ListFetchFilter): Promise<ListResponse> {
  const { data } = await api
    .params({ ...filter })
    .get("/apis/get-list-summaries")
    .call<Response<ListResponse>>();

  // 스키마를 사용하여 검증
  isListItem(data.items);
  return { data };
};

function isListItem(listItems: ListItem[]) {
  // ListItem이 아닌 ListItemSchema를 사용
  listItems.forEach((listItem) => assert(listItem, ListItemSchema));
}
```



API 상태 관리하기

실제 API를 요청하는 코드는 컴포넌트 내에서 비동기 함수를 직접 호출하지 않는다. API의 성공·실패에 따른 상태가 관리 되어야 하므로 상태 관리 라이브러리 **액션** 이나 **훅**과 같이 재정의된 형태를 사용한다.

7.2.1 상태 관리 라이브러리에서 호출하기

상태관리 라이브러리의 비동기 함수들은 **서비스 코드** 를 사용해서 비동기 상태를 변화시킬 수 있는 함수를 제공한다.

컴포넌트는 이러한 함수를 사용하여 상태를 구독하며, 상태가 변경될 때 컴포넌트 재렌더링하는 방식으로 동작한다.

ex) Redux, MobX

! 모든 상태관리 라이브러리에서 비동기 처리 함수를 호출하기 위해 액션이 추가될 때마다 관련된 스토어나 상태가 계속 늘어난다. → 전역 상태 관리자가 모든 비동기 상태에 접근하고 변경할 수 있다는 것이 큰 문제이다.

ex) 2개 이상의 컴포넌트가 구독하고 있는 비동기 상태가 있다면 쓸데없는 비동기 통신이 발생하거나 의도치 않은 상태 변경이 발생할 수 있다.

7.2.2 훅으로 호출하기

캐시를 사용하여 비동기 함수를 호출한다.

- 상태관리 라이브러리에서 발생했던 의도치 않은 상태 변경을 방지하는 데 도움이 된다.

ex) **react-query**, useSwr

🌟 최근 사내에서도 Redux나 MobX와 같은 전역 상태 관리 라이브러리를 react-query로 변경하고자 하는 시도가 이루어지고 있다.

<Job 목록을 불러오는 훅과 Job 1개를 업데이트하는 예시>

만약 Job이 업데이트되면 해당 Job 목록의 정보가 유효하지 않게 되므로 다시 API를 호출해야 함을 알려줘야 한다.

- react-query의 onSuccess 옵션의 **invalidateQueries** : 특정 키의 API를 유효하지 않은 상태로 설정하여 해당 키에 대한 데이터를 다시 가져오도록 트리거한다.

```

// Job 목록을 불러오는 훅
const useFetchJobList = () => {
  return useQuery(["fetchJobList"], async () => {
    const response = await JobService.fetchJobList(); // View Model
    return new JobList(response);
  });
};

// Job 1개를 업데이트하는 훅
const useUpdateJob = (
  id: number,
  // Job 1개 update 이후 Query Option
  { onSuccess, ...options }: UseMutationOptions<void, Error, JobUpdateFormValue>
): UseMutationResult<void, Error, JobUpdateFormValue> => {
  const queryClient = useQueryClient();

  return useMutation(
    ["updateJob", id], // mutation 키. Job ID를 포함해 고유 키 설정
    async (jobUpdateForm: JobUpdateFormValue) => {
      await JobService.updateJob(id, jobUpdateForm); // 서버에 Job 업데이트
    },
    {
      onSuccess: (
        data: void, // updateJob의 return 값은 없다 (status 200으로 성공)
        values: JobUpdateFormValue,
        context: unknown
      ) => {
        // 성공 시 'fetchJobList'를 유효하지 않음으로 설정
        queryClient.invalidateQueries(["fetchJobList"]); // 1. fetchJobList
        onSuccess && onSuccess(data, values, context); // 2. 사용자 정의
      },
      ...options,
    }
  );
};

```

JobList 컴포넌트가 반드시 최신 상태를 표현하는 방법

→ 폴링, 웹소켓 ...

- **폴링** : 클라이언트가 주기적으로 서버에 요청을 보내 데이터를 업데이트하는 것. 클라이언트는 일정한 시간 간격으로 서버에 요청을 보내고, 서버는 해당 요청에 대해 최신 상태의 데이터를 응답으로 보내주는 방식을 말한다.

// <간단한 폴링 방식으로 최신 상태 업데이트하는 코드>

```
const JobList: React.FC = () => {
  // 비동기 데이터를 필요한 컴포넌트에서 자체 상태로 저장
  const {
    isLoading,
    isError,
    error,
    refetch,
    data: jobList,
  } = useFetchJobList();

  // 간단한 Polling 로직, 실시간으로 화면이 갱신돼야 하는 요구가 없어서
  // 30초 간격으로 갱신한다
  useInterval(() => refetch(), 30000);

  // Loading인 경우에도 화면에 표시해준다
  if (isLoading) return <LoadingSpinner />;

  // Error에 관한 내용은 11.3 API 에러 핸들링에서 더 자세하게 다룬다
  if (isError) return <ErrorAlert error={error} />;

  return (
    <>
      {jobList.map((job) => (
        <Job job={job} />
      ))}
    </>
  );
}
```

```
);  
};
```

API 에러 핸들링

비동기 API 호출을 하다 보면 상태 코드에 따라 401 인증되지 않은 사용자, 404 존재하지 않는 리소스, 500 서버 내부 에러 or CORS 에러 등 다양한 에러가 발생할 수 있다.

7.3.1 타입 가드 활용하기

Axios 라이브러리 - Axios 에러에 대한 `isAxiosError` 타입 가드 제공

```
interface ErrorResponse {  
  status: string;  
  serverDateTime: string;  
  errorCode: string;  
  errorMessage: string;  
}
```

```
// 사용자 정의 타입 가드
```

```
function isServerError(error: unknown): error is AxiosError<ErrorResponse> {  
  return axios.isAxiosError(error); // false일 경우 error는 unknown  
}
```

```
const onClickDeleteHistoryButton = async (id: string) => {  
  try {  
    await axios.post("https://....", { id });  
  
    alert("주문 내역이 삭제되었습니다.");  
  } catch (error: unknown) {  
    if (isServerError(e) && e.response && e.response.data.errorMessage) {  
      // error는 AxiosError<ErrorResponse> 타입  
      // 서버 에러일 때의 처리임을 명시적으로 알 수 있다  
      setErrorMessage(e.response.data.errorMessage);  
      return;  
    }  
  }  
}
```



```
// error는 unknown 타입
setErrorMessage("일시적인 에러가 발생했습니다. 잠시 후 다시 시도해주세요");
}
};
```



API 에러 핸들링 시 타입가드 사용 이유

- 타입의 안정성 보장 : 타입가드를 사용하면 컴파일 에러가 발생하지 않는다.
 - (이때 타입가드를 사용하여 컴파일 에러가 아니라 더 분명한 서버 에러 메시지를 확인할 수 있다.)

```
// 타입가드 x할 경우
catch (error: unknown) {
  console.log(error.response?.data.message);
  // ❌ 컴파일 에러: 'unknown' 타입에는 'response' 속성이 없음
}
```

- 에러의 종류 구분 : 일반적인 에러와 서버 에러를 분리해서 처리할 수 있다.

➡ 즉, 타입가드는 비동기 API 에러를 구체적이고 명시적으로 핸들링하는 방법 중 하나 것이다.

7.3.2 에러 서브클래싱하기

단순한 서버 에러뿐만 아니라 인증 정보 에러, 네트워크 에러, 타임아웃 에러 같은 다양한 에러가 발생한다. 이를 명시적으로 표시하기 위해 서브클래싱을 활용한다.

서브클래싱 : 기존(상위 or 부모) 클래스를 확장하여 새로운(하위 or 자식) 클래스를 만드는 과정.


새로운 클래스는 상위 클래스의 모든 속성과 메서드를 상속받아 사용할 수 있고, 추가적인 속성과 메서드를 정의할 수도 있다.

사용자에게 주문 내역을 보여주기 위해 서버에 주문 내역을 요청할 경우

```
const getOrderHistory = async (page: number): Promise<History> =
  try {
    const { data } = await axios.get(`https://some.site?page=${page}`);
    const history = await JSON.parse(data);

    return history;
  } catch (error) {
    alert(error.message);
    // "로그인 정보가 만료되었습니다."
    // "유효하지 않은 요청 데이터입니다."
  }
};
```

! 사용자는 어떤 에러인지 알 수 있지만, 개발자 입장에서는 구분 불가능

 **서브클래싱** → 코드상에서 어떤 에러인지 바로 확인 가능

▼ 사용자 정의 에러 클래스들

```
// HTTP 요청에서 발생한 일반적인 에러
class OrderHttpError extends Error {
  private readonly privateResponse: AxiosResponse<ErrorResponse>

  constructor(message?: string, response?: AxiosResponse<ErrorResponse>) {
    super(message);
    this.name = "OrderHttpError";
    this.privateResponse = response;
  }

  // get 접근자를 통해 외부에서 읽을 수 있도록 한다.
  get response(): AxiosResponse<ErrorResponse> | undefined {
    return this.privateResponse;
  }
}

class NetworkError extends Error {
  constructor(message = "") {
    super(message);
    this.name = "NetworkError";
  }
}
```

```

class UnauthorizedError extends Error {
  constructor(message: string, response?: AxiosResponse<Error>) {
    super(message, response);
    this.name = "UnauthorizedError";
  }
}

```

▼ httpErrorHandler - 에러를 표준화하는 함수

```

const httpErrorHandler = (
  error: AxiosError<ErrorResponse> | Error
): Promise<Error> => {
  let promiseError: Promise<Error>;

  if (axios.isAxiosError(error)) { // Axios 에러인지 확인
    if (Object.is(error.code, "ECONNABORTED")) { // 요청 시간 초과
      promiseError = Promise.reject(new TimeoutError());
    } else if (Object.is(error.message, "Network Error")) { // 네트워크 에러
      promiseError = Promise.reject(new NetworkError());
    } else {
      const { response } = error as AxiosError<ErrorResponse>;

      switch (response?.status) {
        case HttpStatusCode.UNAUTHORIZED: // 인증되지 않은 요청
          promiseError = Promise.reject(
            new UnauthorizedError(response?.data.message, response)
          );
          break;

        default:
          promiseError = Promise.reject(
            new OrderHttpError(response?.data.message, response)
          );
      }
    }
  } else {
    promiseError = Promise.reject(error);
  }
}

```

```

    }

    return promiseError;
};

```

▼ onActionError - 에러 타입에 맞춰 처리하는 함수

```

const alert = (message: string, { onClose }: { onClose?: () =

const onActionError = (
  error: unknown,
  params?: Omit<AlertPopup, "type" | "message">
) => {
  if (error instanceof UnauthorizedError) {
    onUnauthorizedError(
      error.message,
      errorCallback?.onUnauthorizedErrorCallback
    );
  } else if (error instanceof NetworkError) {
    alert("네트워크 연결이 원활하지 않습니다. 잠시 후 다시 시도해주세요.",
      onClose: errorCallback?.onNetworkErrorCallback,
    );
  } else if (error instanceof OrderHttpError) {
    alert(error.message, params);
  } else if (error instanceof Error) {
    alert(error.message, params);
  } else {
    alert(defaultHttpErrorMessage, params);
  }
}

const getOrderHistory = async (page: number): Promise<History> {
  try {
    const { data } = await fetchOrderHistory({ page });
    const history = await JSON.parse(data);

    return history;
  }
}

```

```

    } catch (error) { // httpErrorHandler에서 반환된 에러
      onActionError(error);
    }
  };
};

```



API 호출 흐름

1. `getOrderHistory` 호출

사용자가 주문 내역을 조회하려고 할 때 `getOrderHistory` 함수가 호출된다.

2. `fetchOrderHistory` 호출

`getOrderHistory` 내부에서 `fetchOrderHistory` 가 호출되며, 이 함수에서 실제 API 요청이 이루어진다.

```

const fetchOrderHistory = async (page: number) => {
  try {
    const response = await axios.get(`/api/orders/history`);
    return response.data; // API 응답 데이터 반환
  } catch (error) {
    return httpErrorHandler(error); // 에러 발생 시 httpErrorHandler 호출
  }
};

```

3. `httpErrorHandler` 실행

`fetchOrderHistory` 내에서 API 호출 중 에러가 발생하면, 이를 `httpErrorHandler` 로 전달하여 에러를 표준화된 형태로 처리한다.

- `httpErrorHandler` 는 Axios 에러를 구체적으로 분류하고, 그에 맞는 **사용자 정의 에러** (`TimeoutError`, `NetworkError`, `UnauthorizedError` 등)를 반환한다.

4. `onActionError` 호출

`httpErrorHandler` 에서 반환된 에러가 `onActionError` 로 전달되며, 이 함수는 각 에러 타입에 맞는 처리를 하게 된다. → alert창(UI 알림)

➡ 서브클래싱을 통한 명시적인 에러 처리 : `error instanceof OrderHttpError` 와 같이 작성된 타입 가드문을 통해 코드상에서 에러 핸들링에 대한 부분을 한눈에 볼 수 있다.

7.3.3 인터셉터를 활용한 에러 처리

인터셉터를 통해 HTTP 에러에 일관된 로직을 적용할 수 있다.

```
const httpErrorHandler = (
  error: AxiosError<ErrorResponse> | Error
): Promise<Error> => {
  (error) => {
    // 401 에러인 경우 로그인 페이지로 이동
    if (error.response && error.response.status === 401) {
      window.location.href = `${backOfficeAuthHost}/login?target
    }
    return Promise.reject(error);
  };
};

orderApiRequester.interceptors.response.use(
  (response: AxiosResponse) => response, // 첫번째 콜백: 응답 성공
  httpErrorHandler // 두번째 콜백 : 응답 실패 시 처리
);
```

7.3.4 에러 바운더리를 활용한 에러 처리

에러 바운더리 : 리액트 컴포넌트 트리에서 에러가 발생할 때 공통으로 에러를 처리하는 리액트 컴포넌트 (에러가 발생했을 때 대체 UI를 보여주는 컴포넌트)

- 리액트 컴포넌트 트리 하위에 있는 컴포넌트에서 발생한 에러를 캐치하고, 해당 에러를 가장 가까운 부모 에러 바운더리에서 처리할 수 있다.
- 에러가 발생한 컴포넌트 대신에 에러 처리를 하거나 예상치 못한 에러를 공통 처리할 때 사용할 수 있다.

```
import React, { ErrorInfo } from "react";
import ErrorPage from "pages/ErrorPage";

interface ErrorBoundaryProps {}

// 에러 상태를 추적하기 위한 상태 타입
interface ErrorBoundaryState {
  hasError: boolean; // 플래그
}
```

```

}

class ErrorBoundary extends React.Component<
  ErrorBoundaryProps,
  ErrorBoundaryState
> {
  constructor(props: ErrorBoundaryProps) {
    super(props);
    this.state = { hasError: false };
  }

  // React가 자식 컴포넌트에서 에러를 감지하면, 자동으로 이 메서드를 호출
  // 렌더링단계에서 호출 - 자식 컴포넌트에서 에러가 발생했을 때 React에 의
  static getDerivedStateFromError(): ErrorBoundaryState {
    return { hasError: true };
  }

  // React가 에러를 감지하면, 이 메서드도 자동으로 호출
  // 커밋단계에서 호출 - 에러 로깅 or 에러 리포팅 서비스에 에러 보고 등 누
  componentDidCatch(error: Error, errorInfo: ErrorInfo): void {
    this.setState({ hasError: true });

    console.error(error, errorInfo);
  }

  // render메서드: 에러 발생 여부에 따라 다른 UI를 보여준다.
  // 에러 발생 시, ErrorPage
  // 에러 x, 자식 컴포넌트를 정상 렌더링
  render(): React.ReactNode {
    const { children } = this.props;
    const { hasError } = this.state;

    return hasError ? <ErrorPage /> : children;
  }
}

// OrderHistoryPage 컴포넌트나 그 하위 컴포넌트에서 에러가 발생하면, Error
const App = () => {
  return (
    <ErrorBoundary>

```

```

    <OrderHistoryPage />
  </ErrorBoundary>
);
};

```

부수 효과 : 함수나 연산이 자신의 범위를 벗어나 외부 세계에 영향을 미치는 모든 동작을 의미



`getDerivedStateFromError`, `componentDidCatch` : 생명주기 메서드 중 하나

7.3.5 상태관리 라이브러리에서의 에러 처리

ex) Redux, MobX

7.3.6 react-query를 활용한 에러 처리

```

const JobComponent: React.FC = () => {
  const { isError, error, isLoading, data } = useFetchJobList();

  if (isError) {
    return (
      <div>`${error.message}가 발생했습니다. 나중에 다시 시도해주세요.`
    );
  }

  if (isLoading) {
    return <div>로딩 중입니다.</div>;
  }

  return (
    <>
      {data.map((job) => (
        <JobItem key={job.id} job={job} />
      ))}
    </>
  );
}

```



```
</>  
);  
};
```

7.3.7 그 밖의 에러 처리

API가 조금 다른 커스텀 에러를 반환할 경우

- `successHandler`에 `if`문을 추가해 간단하게 커스텀 에러를 처리할 수 있다.
! 매번 `if`문을 추가해야 한다.
- API requester `axios.create`를 별도 선언하고, 인터셉터에 추가해준다.