

# React Query와 상태관리

“요새 React Query가 뜨고 있는데 그게 무엇이고 우리도 도입해야할까요?”

우아한형제들 주문프론트팀 배민근

# 이 발표는 기술블로그에서 시작되어..



조은옥/DR팀 2:18 PM

안녕하세요 민근님! Developer Relations팀의 조은옥입니다 😊

다름이 아니라, 이번 2월 우아한테크세미나의 연사로 모실수 있을까해서 DM드립니다.

기술블로그에 기고해주신 글([Store에서 비동기 통신 분리하기 \(feat. React Query\)](#))이 많은 PV수와 더불어 댓글로 질문도 활발해 반응이 좋은데요,  
이 글에 대한 내용을 주제로 우아한테크세미나에서 2시간동안 발표+Q&A를 해주시면 어떨지 요청드립니다 😊

## 많은 관심 감사합니다! (꾸벅)

우아한테크 기술 블로그

우아한Tech에 문의하기 개발자 채용



## Store에서 비동기 통신 분리하기 (feat. React Query)

Nov.12.2021 배민근

Web Frontend



스토어

비동기 통신

# 먼저, 오늘은 웹프론트엔드(이하 FE) 세미나입니다.

우아한형제들은 생각보다 많은 곳에 FE 프로덕트를 활용하고 있습니다.



React 개발자시거나  
관심이 있으신 분



FE 상태관리에 대해  
고민이 있으신 개발자



요즘 FE 개발에 대해  
궁금하신 분

+ 모던 FE 개발환경에 대한 기본 지식이 있어야 듣기 수월하실 거예요!

본격적인 이야기 전에 기본지식 잠깐!

# FE 상태관리에 대해

# 상태관리하면 떠오르는 것

라이브러리가 먼저 떠오르는 현실..



Redux



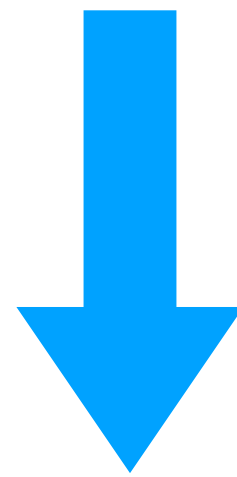
MobX



Recoil

# 상태란?

- ☑ **주어진 시간**에 대해 시스템을 나타내는 것으로 언제든지 변경될 수 있음  
즉 문자열, 배열, 객체 등의 형태로 응용 프로그램에 저장된 데이터



**개발자 입장에서 관리해야하는 데이터들!**

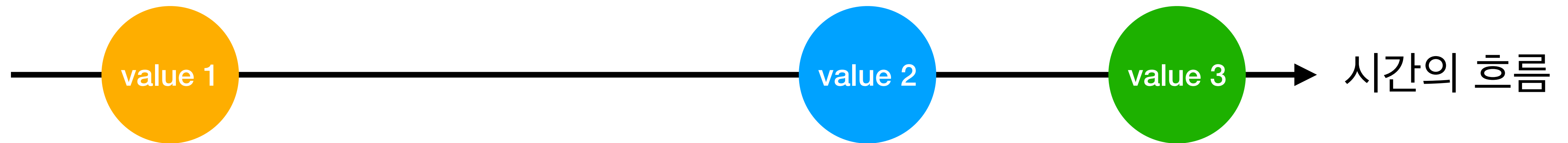
# 모던 웹프론트엔드 개발



UI/UX의 중요성과 함께 프로젝트 규모가 많이 커지고 FE에서 수행하는 역할이 늘어남  
→ 관리하는 상태가 많아짐!

# 상태관리는? (feat. FE)

- ❑ 상태를 관리하는 방법에 대한 것 → 프로젝트가 커짐에 따라 어려움도 커짐
- ❑ 상태들은 시간에 따라 변화함



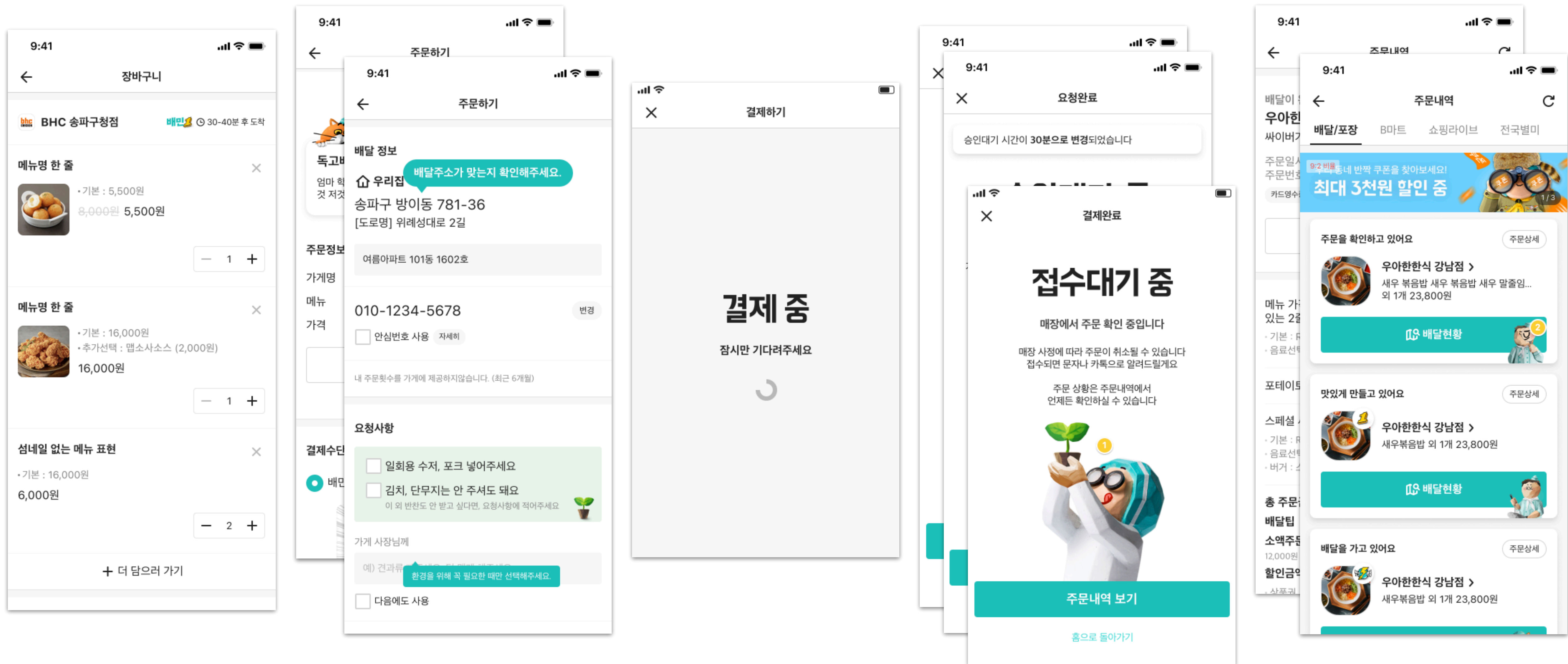
- ❑ React에선 단방향 바인딩이므로 Props Drilling 이슈도 존재
- ❑ Redux와 MobX 같은 라이브러리를 활용해 해결하기도 함



제가 왜 이 'React Query와 상태관리' 라는 주제를 가지고 왔을까요?

**주문 FE 프로젝트를 보며 가진 고민**

사실 이거보다 더 있어요..ㅎ 근데 이게 다 웹뷰인줄 모르셨죠?



# 때는 바야흐로 2021년의 여름..

레거시를 청산하고 앞으로를 위한 기틀을 만들자

Store에 상태관련 코드보다 API 통신관련 코드가 더 많은거 같은데요..

저희 한 팀 FE에서 관리하는 레포가 몇 개..?

3명<sub>(지금은 6명!)</sub>이서 이 많은 서비스, 레포를 어떻게 관리하죠?

→ 이왕할거 Repo합치고, 레거시도 치우고, 신기술 검토도 하면서 주문 FE 아키텍처 통합해보아요!

반복되는 컴포넌트, 로직, 우리 또 Button 만들어요..?

여긴 되는데 여긴 안돼요 왜죠? 아 의존성 모듈 버전..

# 그 중 상태관리에 관한 고민

Redux로 구성한 Store 코드이며 TypeScript로 작성되었습니다.

```
// Redux로 구현하는 Store 예시
export const FETCH_ORDER_REQUEST = 'FETCH_ORDER_REQUEST';
export const FETCH_ORDER_SUCCESS = 'FETCH_ORDER_SUCCESS';

export const fetchOrderRequest = createAction<number, string>(types.FETCH_ORDER_REQUEST, (page: string): number => {
  const parsed = Number.parseInt(page, 10);

  return Number.isNaN(parsed) ? 1 : parsed;
});

export const fetchOrderSuccess = createAction(types.FETCH_ORDER_SUCCESS, (order: OrderPayload) => order);

export const initialState: OrderState = {
  order: undefined,
  isFetching: false,
  ...
};

export default handleActions<OrderState, OrderPayload>({
  [FETCH_ORDER_REQUEST]: (state): OrderState => ({
    ...state,
    isFetching: true,
  }),
  [FETCH_ORDER_SUCCESS]: (state, action: Action<OrderPayload>): OrderState => ({
    ...state,
    order: {
      ...action.payload,
    },
    isFetching: false,
  }),
  ...
},
initialState
);
```

```
// redux-saga를 활용한 비동기 통신 예시
export function* fetchOrder$() {
  try {
    const params = yield select(paramsSelector);
    const { data } = yield call(api.fetchOrder, params);
    yield put(fetchOrderSuccess(data));
    ...
  } catch (error) {
    ...
  }
}

export function* fetchOrderSuccess$(action: Action<OrderPayload>){
  try {
    yield call(saveHistorys, action.payload);
  } catch (error) {
    ...
  }
}
...
export default function* order$() {
  yield all([
    takeLatest(FETCH_ORDER_REQUEST, fetchOrder$),
    takeLatest(FETCH_ORDER_SUCCESS, fetchOrderSuccess$),
    ...
  ]);
}
```

# 그 중 상태관리에 관한 고민

Redux로 구성한 Store 코드이며 TypeScript로 작성되었습니다.

```
// Redux로 구현하는 Store 예시
export const FETCH_ORDER_REQUEST = 'FETCH_ORDER_REQUEST';
export const FETCH_ORDER_SUCCESS = 'FETCH_ORDER_SUCCESS';

export const fetchOrderRequest = createAction<number, string>(types.FETCH_ORDER_REQUEST, (page: string): number => {
  const parsed = Number.parseInt(page, 10);

  return Number.isNaN(parsed) ? 1 : parsed;
});

export const fetchOrderSuccess = createAction<OrderPayload>(types.FETCH_ORDER_SUCCESS, (order: OrderPayload): OrderPayload => order);

export const initialState: OrderState = {
  order: undefined,
  isFetching: false,
  ...
};

export default handleActions<OrderState, OrderPayload>({
  [FETCH_ORDER_REQUEST]: (state): OrderState => ({
    ...state,
    isFetching: true,
  }),
  [FETCH_ORDER_SUCCESS]: (state, action: Action<OrderPayload>): OrderState => ({
    ...state,
    order: {
      ...action.payload,
    },
    isFetching: false,
  }),
  ...
},
initialState
);
```

Store는 전역 상태가 저장되고 관리되는 공간인데

## 상태관리보단 API 통신 코드..?

```
// redux-saga를 활용한 비동기 통신 예시
export function* fetchOrder$() {
  try {
    const params = yield select(paramsSelector);
    const { data } = yield call(api.fetchOrder, params);
    yield call(fetchOrderSuccess, data);
  } catch (error) {
    ...
  }
}

export function* fetchOrderSuccess$(action: Action<OrderPayload>){
  try {
    yield call(saveHistorys, action.payload);
  } catch (error) {
    ...
  }
}

...

export default function* order$() {
  yield all([
    takeLatest(FETCH_ORDER_REQUEST, fetchOrder$),
    takeLatest(FETCH_ORDER_SUCCESS, fetchOrderSuccess$),
    ...
  ]);
}
```



# 여기서 다 관리하는게 맞나..?

상태관리 영역이 서버값을 저장하는데까지 확장

- ❑ API 통신 관련 코드가 모두 Store에?
- ❑ 또, 반복되는 isFetching, isError 등 API 관련 상태
- ❑ 또또, 반복되는 비슷한 구조의 API 통신 코드

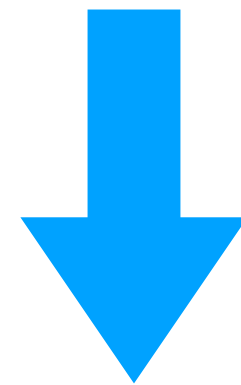
# 서버에서 받아야하는 상태들의 특성

Client에서 제어하거나 소유되지 않은 원격의 공간에서 관리되고 유지됨

Fetching이나 Updating에 비동기 API가 필요함

다른 사람들과 공유되는 것으로 사용자가 모르는 사이에 변경될 수 있음

신경 쓰지 않는다면 잠재적으로 "out of date"가 될 가능성을 지님



사실상 FE에서 이 값들이 저장되어있는 state들은 일종의 캐시

# 서버에서 받아야하는 상태들의 특성

Client에서 제어하거나 소유되지 않은 원격의 공간에서 관리되고 유지됨

Fetching이나 Updating에 비동기 API가 필요함  
Client에서 관리하는 일반적인 상태들의 특성과는 다르죠?

## 어쩌면 다른 관리방법이 있다면 좋을지도?

신경 쓰지 않는다면 잠재적으로 "out of date"가 될 가능성을 지님



사실상 FE에서 이 값들이 저장되어있는 state들은 일종의 캐시



# 상태를 두 가지로 나누어 봅시다



Client State

The diagram consists of two large circles. The left circle is orange and contains the text 'Client State'. The right circle is blue and contains the text 'Server State'. Both circles are positioned below the main title.

Server State

# Client State vs. Server State

Key Point는 데이터의 Ownership이 있는 곳

## Client State

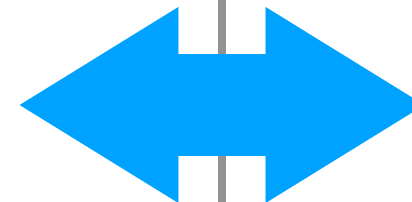
- ☑ Client에서 소유하며 온전히 제어가능함
- ☑ 초기값 설정이나 조작에 제약사항 없음
- ☑ 다른 사람들과 공유되지 않으며 Client 내에서 UI/UX 흐름이나 사용자 인터렉션에 따라 변할 수 있음
- ☑ 항상 Client 내에서 최신 상태로 관리됨

**Ownership이 Client에**

## Server State

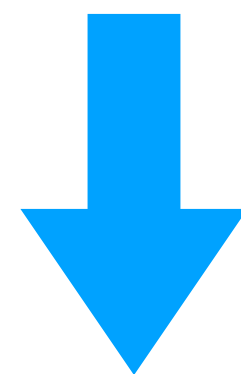
- ☑ Client에서 제어하거나 소유되지 않은 원격의 공간에서 관리되고 유지됨
- ☑ Fetching/Updating에 비동기 API가 필요함
- ☑ 다른 사람들과 공유되는 것으로 사용자가 모르는 사이에 변경될 수 있음
- ☑ 신경 쓰지 않는다면 잠재적으로 "out of date"가 될 가능성을 지님

**Ownership이 Server에**



# 다시, 상태관리 라이브러리

애네가 지금에 와서도 서버 상태를 관리하기 적합할까요?



주문에서 선택한 해답은



**React Query**

# React Query 살펴보기

# React Query 자기소개



Q Search docs

⌕ K

Docs

Examples

Learn

TanStack



## Performant and powerful data synchronization for React

Fetch, cache and update data in your React and React Native applications all without touching any "global state".

Get Started

GitHub

Still using v2? No problem! [Find the v2 docs here.](#)



### Declarative & Automatic

Writing your data fetching logic by hand is over. Tell React Query where to get your data and how fresh you need it to be and the rest is automatic. React Query handles caching, background updates and stale data out of the box with zero-configuration.

### Simple & Familiar

If you know how to work with promises or async/await, then you already know how to use React Query. There's no global state to manage, reducers, normalization systems or heavy configurations to understand. Simply pass a function that resolves your data (or throws an error) and the rest is history.

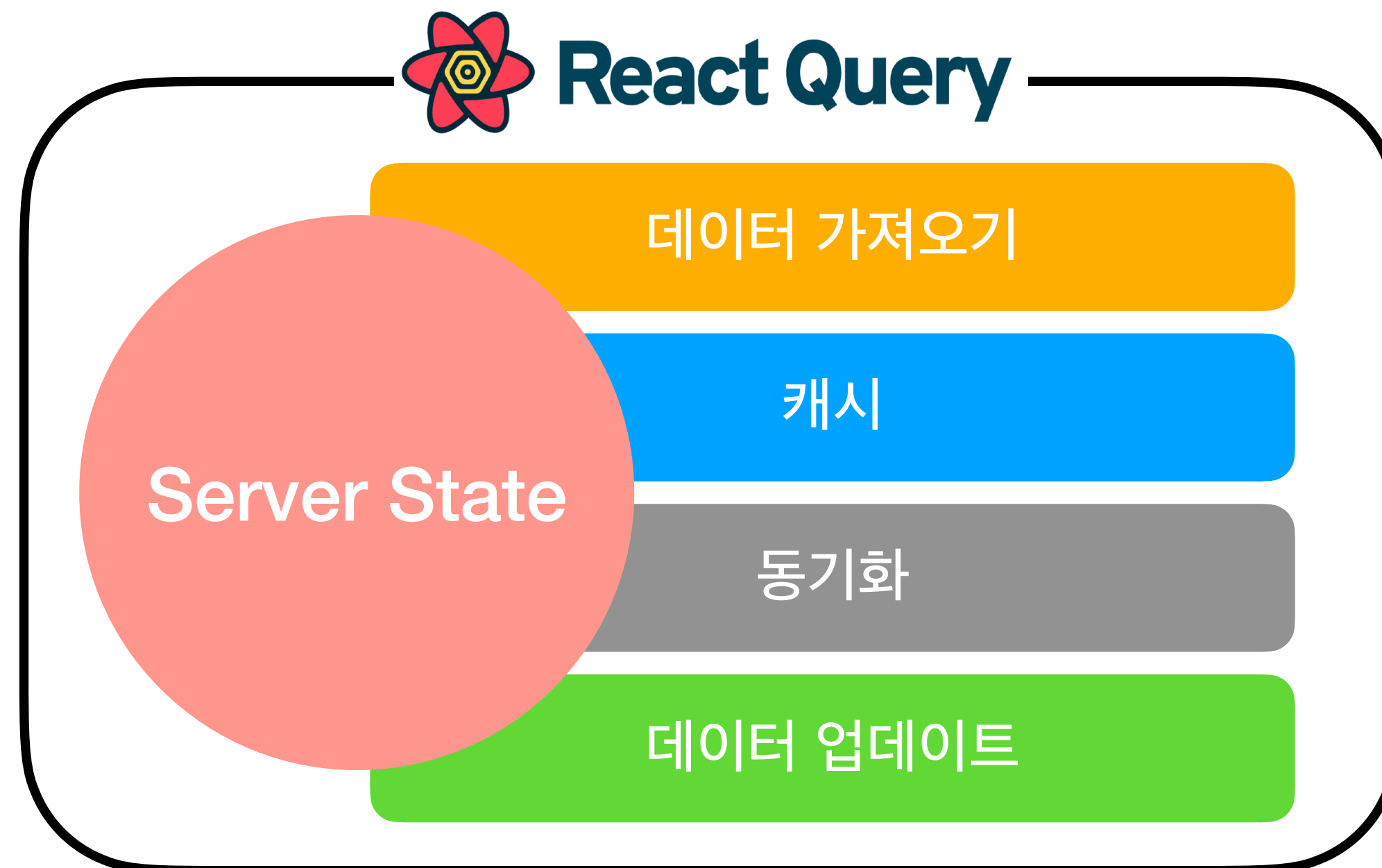
### Powerful & Configurable

React Query is configurable down to each observer instance of a query with knobs and options to fit every use-case. It comes wired up with dedicated devtools, infinite-loading APIs, and first class mutation tools that make updating your data a breeze. Don't worry though, everything is pre-configured for success!

# Overview

역시 최고의 문서는 공식 Docs죠!

- ❑ React Query is often described as the missing data-fetching library for React, but in more technical terms, it makes **fetching, caching, synchronizing and updating server state** in your React applications a breeze.



# Overview

역시 최고의 문서는 공식 Docs죠!

- ❑ React Query is hands down **one of the best libraries** for managing server state. It works amazingly well **out-of-the-box, with zero-config, and can be customized** to your liking as your application grows.



 **React Query**는 zero-config로 즉시 사용가능, But 원하면 언제든지 config도 커스텀 가능!



# 됐고 간단한 예제 하나



**React Query**

공식 문서의 예제입니다.

```
import { QueryClient, QueryClientProvider, useQuery } from 'react-query'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { isLoading, error, data } = useQuery('repoData', () =>
    fetch('https://api.github.com/repos/tannerlinsley/react-query').then(res =>
      res.json()
    )
  )

  if (isLoading) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.description}</p>
      <strong>👁️ {data.subscribers_count}</strong>{' '}
      <strong>🌟 {data.stargazers_count}</strong>{' '}
      <strong>🔗 {data.forks_count}</strong>
    </div>
  )
}
```



됐고 간단한 예제 하나

딱히 config도 없고 코드도 React Hooks 같고

첫 인상은 그래도 좀 간단..?

공식 문서의 예제입니다.

```
import { QueryClient, QueryClientProvider, useQuery } from 'react-query'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { data, error, isLoading, isFetching } = useQuery('repoData', () =>
    fetch('https://api.github.com/repos/tannerlinsley/react-query').then(res =>
      res.json()
    )
  )

  if (isLoading) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <h1>{data.name}</h1>
      <p>{data.description}</p>
      <strong>👤 {data.subscribers_count}</strong>{' '}
      <strong>👁️ {data.stargazers_count}</strong>{' '}
      <strong>🔗 {data.forks_count}</strong>
    </div>
  )
}
```

# 본격적으로 알아보기 전에!

React에서 쓰려면 QueryClientProvider 필수!

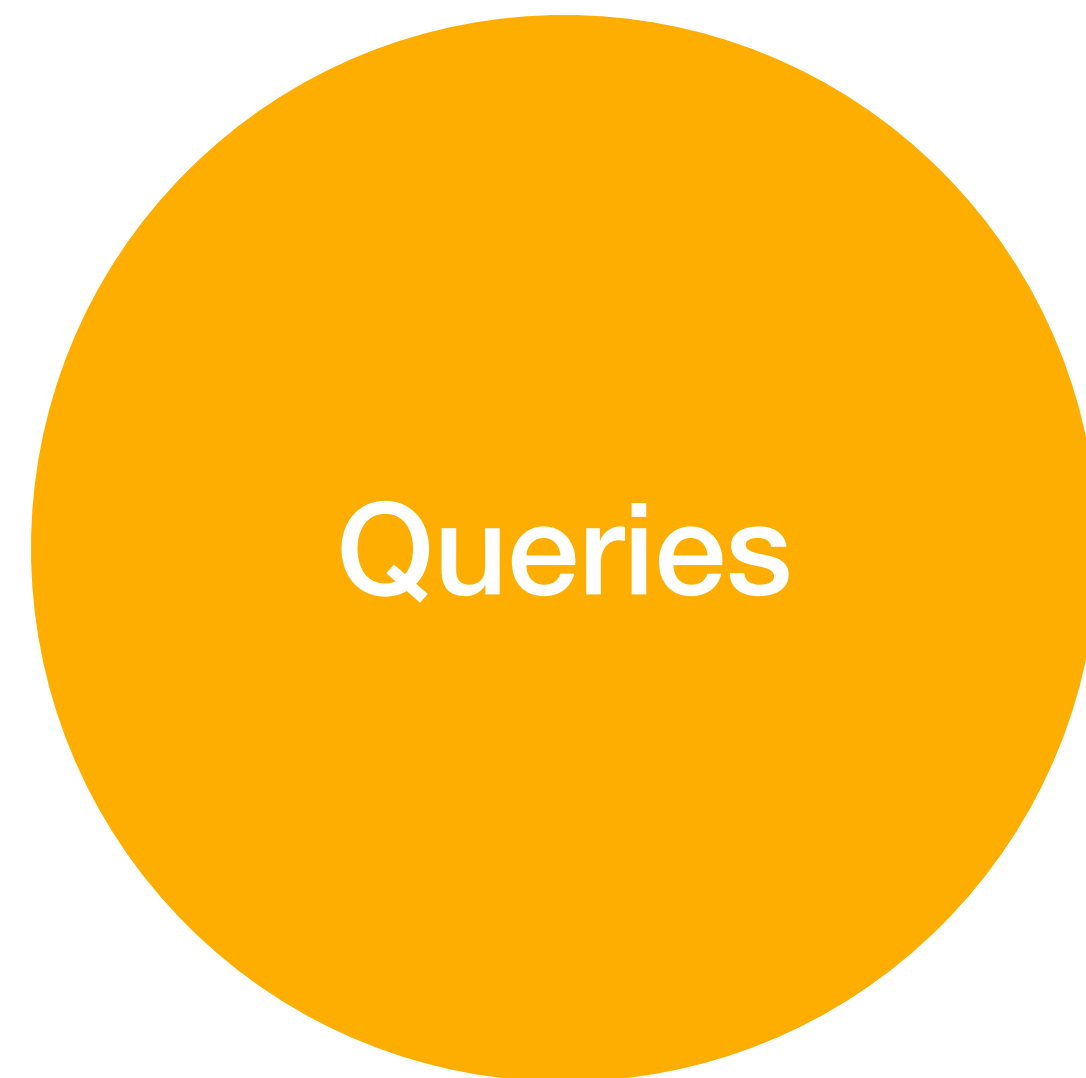
```
import { QueryClient, QueryClientProvider } from 'react-query'

const queryClient = new QueryClient()

function App() {
  return <QueryClientProvider client={queryClient}>...</QueryClientProvider>
}
```

# 세 가지 core 컨셉 살펴보기

공식 문서 Quick Start에서 짚은 3가지 개념



세미나에서 다루지 않는 내용은 공식 문서 잘 되어 있으니 공식 문서를 참고해주세요!

# Queries

보통 GET으로 받아올 대부분의 API에 사용할 아이

- ☑ A query is a declarative dependency on an asynchronous source of data that is tied to a **unique key**.
- ☑ A query can be used with any Promise based method (including GET and POST methods) to fetch data from a server.
- ☑ If your method modifies data on the server, we recommend using Mutations instead.

# Queries

보통 GET으로 받아올 대부분의 API에 사용할 아이

- ☑ A query is a declarative dependency on an asynchronous source of data that is tied to a **unique key**.

CRUD 중 Reading에만 사용할 거예요

- ☑ A query can be used with any Promise based method (including GET and POST methods) to fetch data from a server.

# Queries는 데이터 Fetching용!

- ☑ If your method modifies data on the server, we recommend using Mutations instead.

# Queries

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```

Query Key

Query Function

# Query Key

Key, Value 맵핑구조 생각하시면 됩니다.

- ☑ React Query는 Query Key에 따라 query caching을 관리합니다.

## String 형태

```
// A list of todos
useQuery('todos', ...) // queryKey === ['todos']

// Something else, whatever!
useQuery('somethingSpecial', ...) // queryKey === ['somethingSpecial']
```

## Array 형태

```
// An individual todo
useQuery(['todo', 5], ...)
// queryKey === ['todo', 5]

// An individual todo in a "preview" format
useQuery(['todo', 5, { preview: true }], ...)
// queryKey === ['todo', 5, { preview: true }]

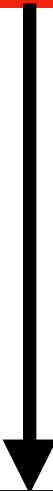
// A list of todos that are "done"
useQuery(['todos', { type: 'done' }], ...)
// queryKey === ['todos', { type: 'done' }]
```

# Query Function

Data Fetching 할 때 Promise 함수 만들죠?

- ☑ Promise를 반환하는 함수! → 데이터 resolve하거나 error를 throw

```
useQuery('fetchOrder', () => fetchOrder(orderNo), options)
```



```
/**
 * 주문 내역 조회
 * @param orderNo
 */
export const fetchOrder = (orderNo: string): Promise<ServerResponse<FetchOrderResponse>> =>
  orderHistoryApiRequester
    .get<ServerResponse<FetchOrderResponse>>(`[REDACTED]/${orderNo}`)
    .then((response: AxiosResponse<ServerResponse<FetchOrderResponse>>) => response.data);
```

fetch, axios, etc.



# 다시, Queries

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```

Query Key

Query Function

# 다시, Queries

```
import { useQuery } from 'react-query'
```

## 그럼 useQuery가 반환하는 건 뭔가요?

```
const info = useQuery('todos', fetchTodoList)
```

Query Key

Query Function

# useQuery

```
const {
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isIdle,
  isLoading,
  isLoadingError,
  isPlaceholderData,
  isPreviousData,
  isRefetchError,
  isRefetching,
  isStale,
  isSuccess,
  refetch,
  remove,
  status,
} = useQuery(queryKey, queryFn?, {
```

- ❑ data: 마지막으로 성공한 resolved된 데이터 (Response)
- ❑ error: 에러가 발생했을 때 반환되는 객체
- ❑ isFetching: Request가 in-flight 중일 때 true
- ❑ status, isLoading, isSuccess, isIdle 등등  
: 모두 현재 query의 상태
- ❑ refetch: 해당 query refetch하는 함수 제공
- ❑ remove: 해당 query cache에서 지우는 함수 제공
- ❑ etc.

# useQuery

```
const {
  data,
  dataUpdatedAt,
  error,
  errorUpdatedAt,
  failureCount,
  isError,
  isFetched,
  isFetchedAfterMount,
  isFetching,
  isIdle,
  isLoading,
  isLoadingError,
  isPlaceholderData,
  isPreviousData,
  isRefetchError,
  isRefetching,
  isStale,
  isSuccess,
  refetch,
  remove,
  status,
} = useQuery(queryKey, queryFn?, {
```

☑ data: 마지막으로 성공한 resolved된 데이터 (Response)

☑ error: 에러가 발생했을 때 반환되는 객체

라이브러리에서 제공안하면 사실 우리가 다 구현해야할..

## 유용한 여러 인터페이스 제공!

☑ isFetching: Request가 in-flight 중일 때 true

☑ status, isLoading, isSuccess, isLoading 등등  
: 모두 현재 query의 상태

☑ refetch: 해당 query refetch하는 함수 제공

☑ remove: 해당 query cache에서 지우는 함수 제공

# 또 다시, Queries

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```

Query Key

Query Function

또 다시, Queries

```
import { useQuery } from 'react-query'
```

**아까 config 커스텀 된다면서요?**

```
const info = useQuery('todos', fetchTodoList)
```

Query Key

Query Function

# useQuery Option

- ☑️ 아까 살짝 지나간 예제코드

```
useQuery('fetchOrder', () => fetchOrder(orderNo), options)
```

# useQuery Option

```
} = useQuery(queryKey, queryFn?, {
  cacheTime,
  enabled,
  initialData,
  initialDataUpdatedAt,
  isDataEqual,
  keepPreviousData,
  meta,
  notifyOnChangeProps,
  notifyOnChangePropsExclusions,
  onError,
  onSettled,
  onSuccess,
  placeholderData,
  queryKeyHashFn,
  refetchInterval,
  refetchIntervalInBackground,
  refetchOnMount,
  refetchOnReconnect,
  refetchOnWindowFocus,
  retry,
  retryOnMount,
  retryDelay,
  select,
  staleTime,
  structuralSharing,
  suspense,
  useErrorBoundary,
})
```

- ☑ `onSuccess, onError, onSettled`: query fetching 성공/실패/완료 시 실행할 Side Effect 정의
- ☑ `enabled`: 자동으로 query를 실행시킬지 말지 여부
- ☑ `retry`: query 동작 실패 시, 자동으로 retry 할지 결정하는 옵션
- ☑ `select`: 성공 시 가져온 data를 가공해서 전달
- ☑ `keepPreviousData`: 새롭게 fetching 시 이전 데이터 유지 여부
- ☑ `refetchInterval`: 주기적으로 refetch 할지 결정하는 옵션
- ☑ etc.



# 다시 공식예제

이제 좀 이해가 되시나요?

```
import { QueryClient, QueryClientProvider, useQuery } from 'react-query'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { isLoading, error, data } = useQuery('repoData', () =>
    fetch('https://api.github.com/repos/tannerlinsley/react-query').then(res =>
      res.json()
    )
  )

  if (isLoading) return 'Loading...'

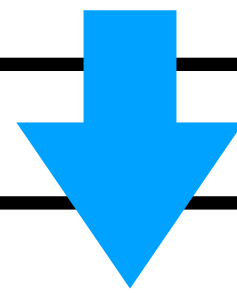
  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <div>
        <h1>{data.name}</h1>
        <p>{data.description}</p>
        <strong>👁️ {data.subscribers_count}</strong>{' '}
        <strong>🌟 {data.stargazers_count}</strong>{' '}
        <strong>🔗 {data.forks_count}</strong>
      </div>
    </div>
  )
}
```

# queries 파일 분리도 추천!

## Query 선언부

```
export const useFetchOrder = (  
  orderNo: string,  
  options?: UseQueryOptions<ServerResponse<FetchOrderResponse>, AxiosError<ErrorResponse>, ServerResponse<FetchOrderResponse>, 'fetchOrder'>  
) : UseQueryResult<ServerResponse<FetchOrderResponse>, AxiosError<ErrorResponse>> => useQuery('fetchOrder', () => fetchOrder(orderNo), options);  
  
export const useFetchDeliveryStatus = (  
  orderNo: string,  
  options?: UseQueryOptions<FetchDeliveryStatusResponse, AxiosError<ErrorResponse>, FetchDeliveryStatusResponse, 'fetchDeliveryStatus'>  
) : UseQueryResult<FetchDeliveryStatusResponse, AxiosError<ErrorResponse>> =>  
  useQuery('fetchDeliveryStatus', () => fetchDeliveryStatus(orderNo), options);
```



## Components

```
const fetchOrderResult = useFetchOrder(orderNo, {  
  onSuccess: fetchOrderResultData => {  
    // onSuccess 로직  
  },  
  onError: error => {  
    // onError 로직  
  },  
});
```

```
const fetchDeliveryStatusResult = useFetchDeliveryStatus(orderNo, {  
  refetchInterval: isPollingActive ? 5000 : false,  
  onSuccess: data => {  
    // onSuccess 로직  
  },  
  onError: error => {  
    // onError 로직  
  },  
});
```

# 그럼 query가 여러 개일 땐

알아서 잘 된다! (동적으로 하려면 다른 방법이 있습니다.)

```
function App () {  
  // The following queries will execute in parallel  
  const usersQuery = useQuery('users', fetchUsers)  
  const teamsQuery = useQuery('teams', fetchTeams)  
  const projectsQuery = useQuery('projects', fetchProjects)  
  ...  
}
```

# 그럼 질문!

기술블로그 댓글 중 발췌 (오래도록 못봐서 죄송합니다..ㅜ.ㅜ)



오 \* 영

좋은 글 감사합니다. 저도 최근 프로젝트에 리액트 쿼리를 써보고 있는데요. 몇 가지 질문 드리고 싶습니다.

1. `useQuery()`를 쓸 때 선언적인 성격 때문에 고민될 때가 있었습니다.

예를 들어, 마운트 시에는 데이터가 패치되지 않고 버튼을 클릭했을 때 데이터를 패치 받아서 데이터에 따라 `history.push` 해야하는 상황을 가정해보겠습니다. 이때

1) `enabled`을 `false`로 두고 이벤트 핸들러에서 `refetch()` 로 매뉴얼하게 패치하는 방법

2) `enabled` 옵션에 해당하는 상태를 `useState` 로 컴포넌트내에 두고, 이벤트 핸들러에서 해당 상태 값을 변경하여 `enabled` 를 조건부로 만족시켜 패치시키는 방법

을 고려해봤는데요. 저는 다소 복잡하다고 느꼈지만, github discussion 을 살펴보니 react query 제작자들은 선언적이라는 이유로 2) 를 권장하는 것으로 보입니다. (다만 찢러야할 API가 많을 때, 관리할 state가 너무 많아집니다)

관련된 상황이 있으신지 궁금합니다.



# 그럼 질문! 2

기술블로그 댓글 중 발췌 (오래도록 못봐서 죄송합니다..ㅜ.ㅜ)



**Jae** \* **Yoon**

글 잘 보았습니다.

궁금한 점이 하나 있습니다.

useQuery를 이용해 불러온 server state를 이용해 액션을 생성해야 하는 경우가 있을텐데,  
이때는 어느 단계 (action creator?) 에서 server state를 참조하나요?

# Mutations

데이터 updating 시 사용하는 아이

- ❑ Unlike queries, mutations are typically used to create/update/delete data or perform server side-effects. For this purpose, React Query exports a `useMutation` hook.

# Mutations

데이터 updating 시 사용하는 아이

- ❑ Unlike queries, mutations are typically used to create/update/delete data or perform server side-effects. For this purpose, React Query exports a `useMutation` hook.
- CRUD 중 Create/Update/Delete에 모두 사용할거예요

# Mutations는 데이터 생성/수정/삭제용!

# Mutations

```
const mutation = useMutation(newTodo => {  
  return axios.post('/todos', newTodo)  
})
```

useQuery 보다 더 심플하게 Promise 반환 함수만 있어도 됩니다!  
(단, Query Key 넣어주면 devtools에서 볼 수 있어요)



# useMutation

```
const {
  data,
  error,
  isError,
  isIdle,
  isLoading,
  isPaused,
  isSuccess,
  mutate,
  mutateAsync,
  reset,
  status,
} = useMutation(mutationFn, {
```

- ❑ mutate: mutation을 실행하는 함수
- ❑ mutateAsync: mutate와 비슷 But Promise 반환
- ❑ reset: mutation 내부 상태 clean
- ❑ 나머지 특별히 설명할 거 없이 useQuery랑 비슷합니다. (오히려 반환하는 객체 안의 내용이 더 적어요!)

# useMutation Option

```
} = useMutation(mutationFn, {  
  mutationKey,  
  onError,  
  onMutate,  
  onSettled,  
  onSuccess,  
  retry,  
  retryDelay,  
  useErrorBoundary,  
  meta,  
})
```

- ❑ onMutate: 본격적인 Mutation 동작 전에 먼저 동작하는 함수, Optimistic update 적용할 때 유용
- ❑ 나머지 특별히 설명할 거 없이 useQuery랑 비슷합니다. (오히려 Option이 더 적어요!)

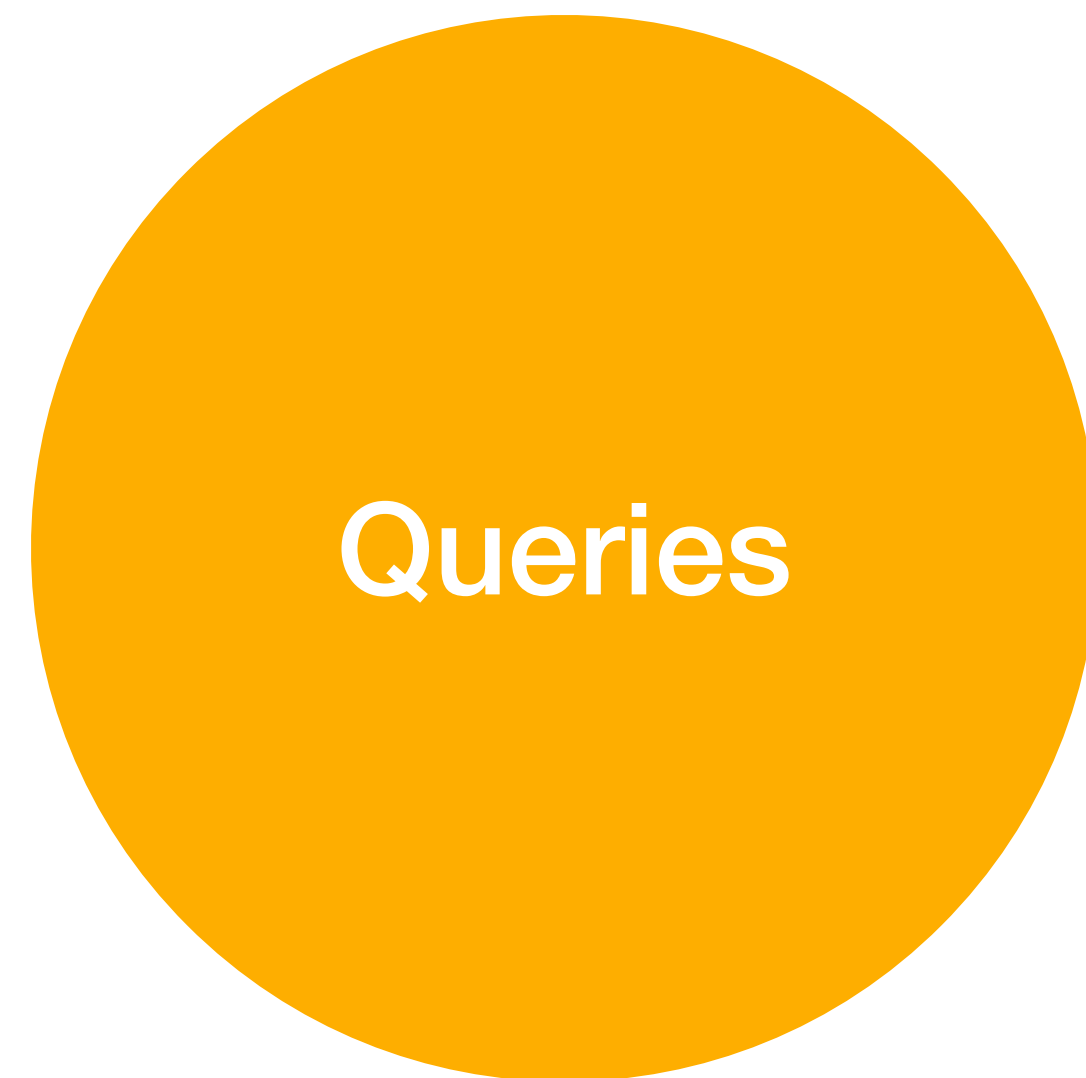
# Query Invalidation

- ❑ 간단히 queryClient를 통해 invalidate 메소드를 호출하면 끝!

```
// Invalidate every query in the cache  
queryClient.invalidateQueries()  
// Invalidate every query with a key that starts with `todos`  
queryClient.invalidateQueries('todos')
```

- ❑ 이러면 해당 Key를 가진 query는 stale 취급되고, 현재 rendering 되고 있는 query들은 백그라운드에서 refetch 됩니다.

# 세 가지 core 개념을 살펴보았습니다!



근데 뭔가 아까 React Query가 자기소개한 내용에 비해 부족하지 않나요?

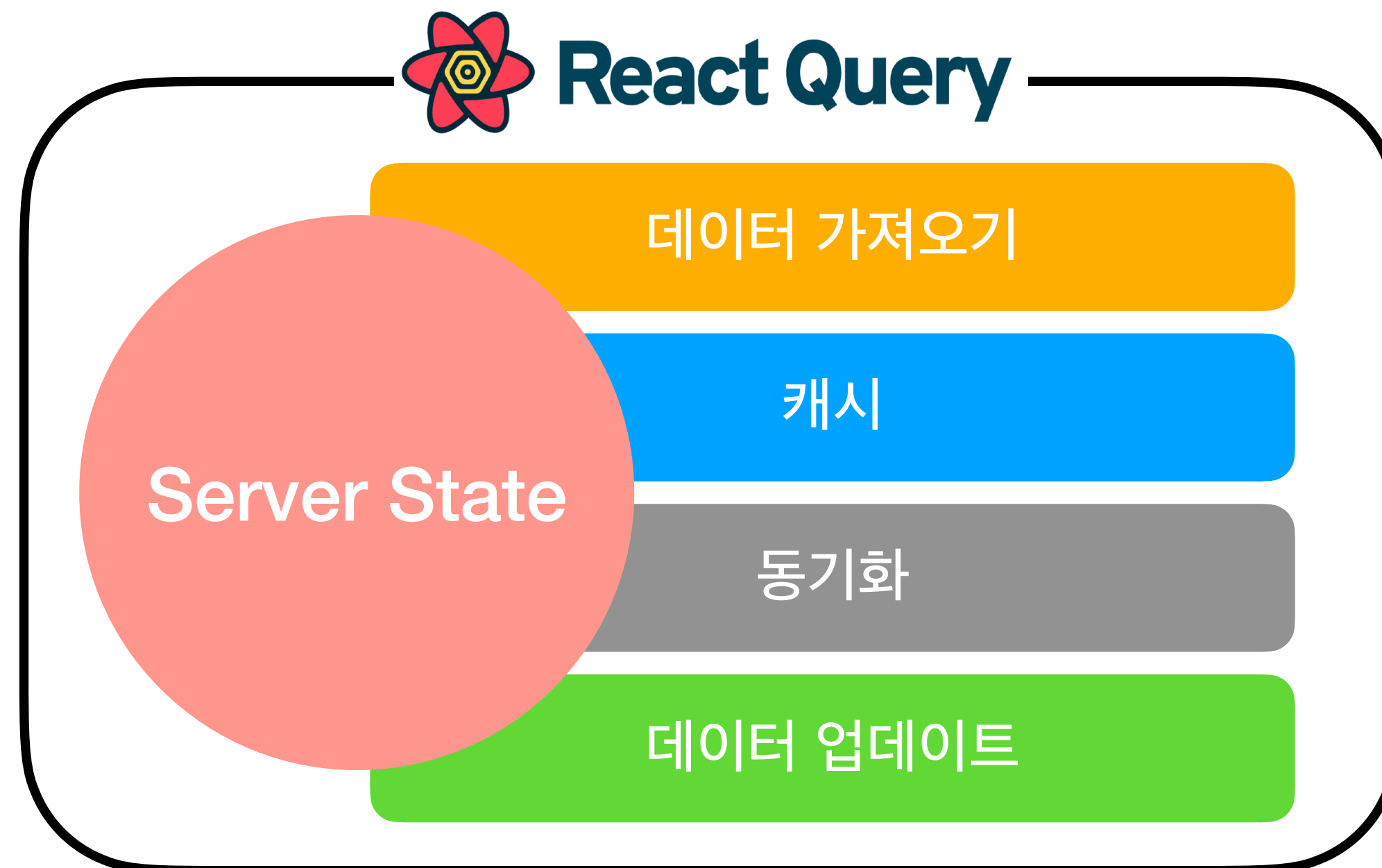
Data fetching하고 updating은 알겠고 그럼

# Caching하고 Synchronization은요?

# 살펴보기 전에 잠깐

여러분 모르는 사이에 등장한 옵션들

- ❑ 사실 아까 Option에 `cacheTime`, `staleTime`도 있었고
- ❑ `refetchOnWindowFocus`, `refetchOnMount` 같은 것도 있었고



# RFC 5861

## HTTP Cache-Control Extensions for Stale Content

- ☑ stale-while-revalidate
  - 백그라운드에서 stale response를 revalidate 하는 동안 캐시가 가진 stale response를 반환

Cache-Control: max-age=600, stale-while-revalidate=30

- ☑ 이렇게 동작하면 Latency가 숨겨지겠죠?
- ☑ (stale-if-error 도 이 명세에 있어요)

# 이 컨셉을 메모리 캐시에도 적용해보자!

이러고 나온게 react-query, swr, etc.

- ☑ cacheTime: 메모리에 얼마만큼 있을건지 (해당시간 이후 GC에 의해 처리, default 5분)
- ☑ staleTime: 얼마의 시간이 흐른 후에 데이터를 stale 취급할 것인지 (default 0)

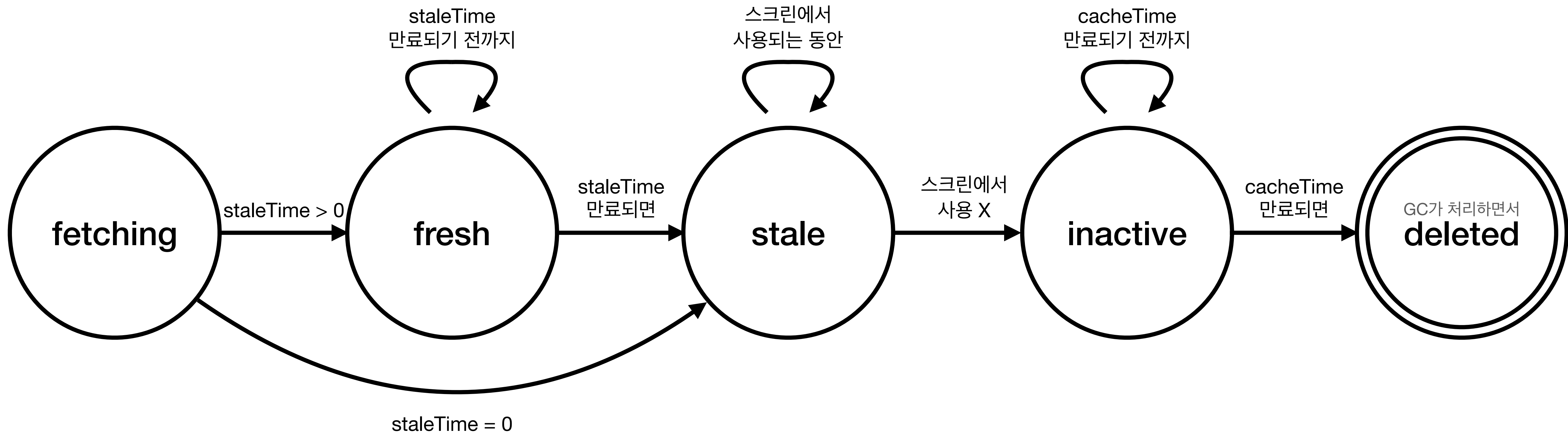
Q. 2. 개발하실 때 cacheTime 이나 staleTime 등을 변경해서 사용하는 경우가 있으신지 궁금합니다.

- ☑ refetchOnMount / refetchOnWindowFocus / refetchOnReconnect  
→ true 이면 Mount / window focus / reconnect 시점에 data가 stale이라고 판단되면 모두 refetch (모두 default true)



# Query 상태흐름

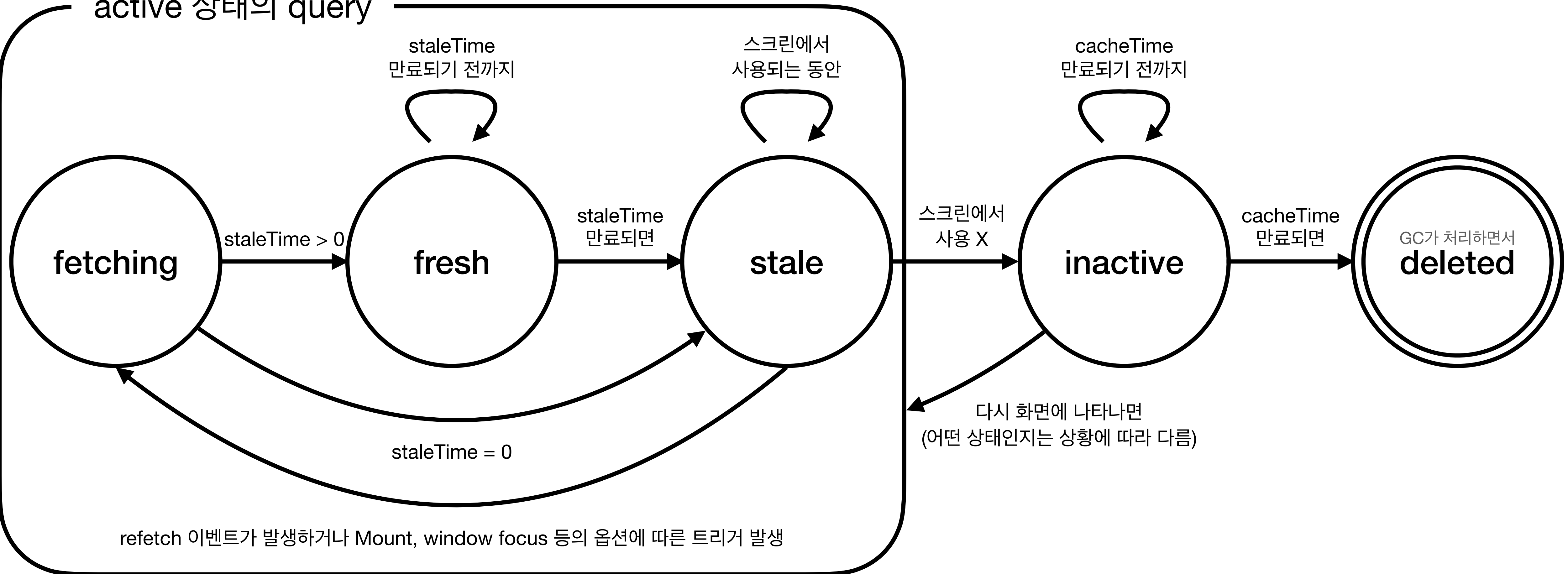
화면에 있다가 사라지는 query



# Query 상태흐름

화면에 있다가 없다가 좀 더 복잡한 query

active 상태의 query



# zero-config에서도 이런 역할을

알아서 하는 것들이 있어서 좋지만 주의도 해야합니다.

staleTime → default값 0

refetchOnMount/refetchOnWindowFocus/refetchOnReconnect → default값 true

cacheTime → default값  $60 * 5 * 1000$

retry → default값 3, retryDelay → default값 exponential backoff function

# zero-config에서도 이런 역할을

알아서 하는 것들이 있어서 좋지만 주의도 해야합니다.

Queries에서 cached data는 언제나 stale 취급

refetchOnMount/refreshOnMount 각 시점에서 data가 stale이라면 항상 refetch 발생 → default값 true

inactive Query들은 5분 뒤 GC에 의해 처리

retry → default값 3, retryDelayBackoff exponential backoff function  
Query 실패 시 3번까지 retry 발생

# 또 다시 공식예제

Caching 관점에서 살펴봅시다.

```
import { QueryClient, QueryClientProvider, useQuery } from 'react-query'

const queryClient = new QueryClient()

export default function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Example />
    </QueryClientProvider>
  )
}

function Example() {
  const { isLoading, error, data } = useQuery('repoData', () =>
    fetch('https://api.github.com/repos/tannerlinsley/react-query').then(res =>
      res.json()
    )
  )

  if (isLoading) return 'Loading...'

  if (error) return 'An error has occurred: ' + error.message

  return (
    <div>
      <div>
        <h1>{data.name}</h1>
        <p>{data.description}</p>
        <strong>👁️ {data.subscribers_count}</strong>{' '}
        <strong>🌟 {data.stargazers_count}</strong>{' '}
        <strong>🔗 {data.forks_count}</strong>
      </div>
    </div>
  )
}
```

여기까지 오니 그럼 React Query는

**어디에서 값들을 관리할까요?**

# 전역상태처럼 관리되는 데이터들

어떻게 Server State들을 전역상태처럼 관리할까요?

Components A

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```

Components B

```
import { useQuery } from 'react-query'

function App() {
  const info = useQuery('todos', fetchTodoList)
}
```



# 전역상태처럼 관리되는 데이터들

어떻게 Server State들을 전역상태처럼 관리할까요?

**해답은 Context API에 있습니다**

Components A

```
import { useQuery } from 'react-query'
```

```
function App() {
```

```
  const info = useQuery('todos', fetchTodoList)
```

Components B

```
import { useQuery } from 'react-query'
```

```
function App() {
```

```
  const info = useQuery('todos', fetchTodoList)
```



# QueryClient 내부적으로 Context 사용

```
function getClientContext(contextSharing: boolean) {
  if (contextSharing && typeof window !== 'undefined') {
    if (!window.ReactQueryClientContext) {
      window.ReactQueryClientContext = defaultContext
    }

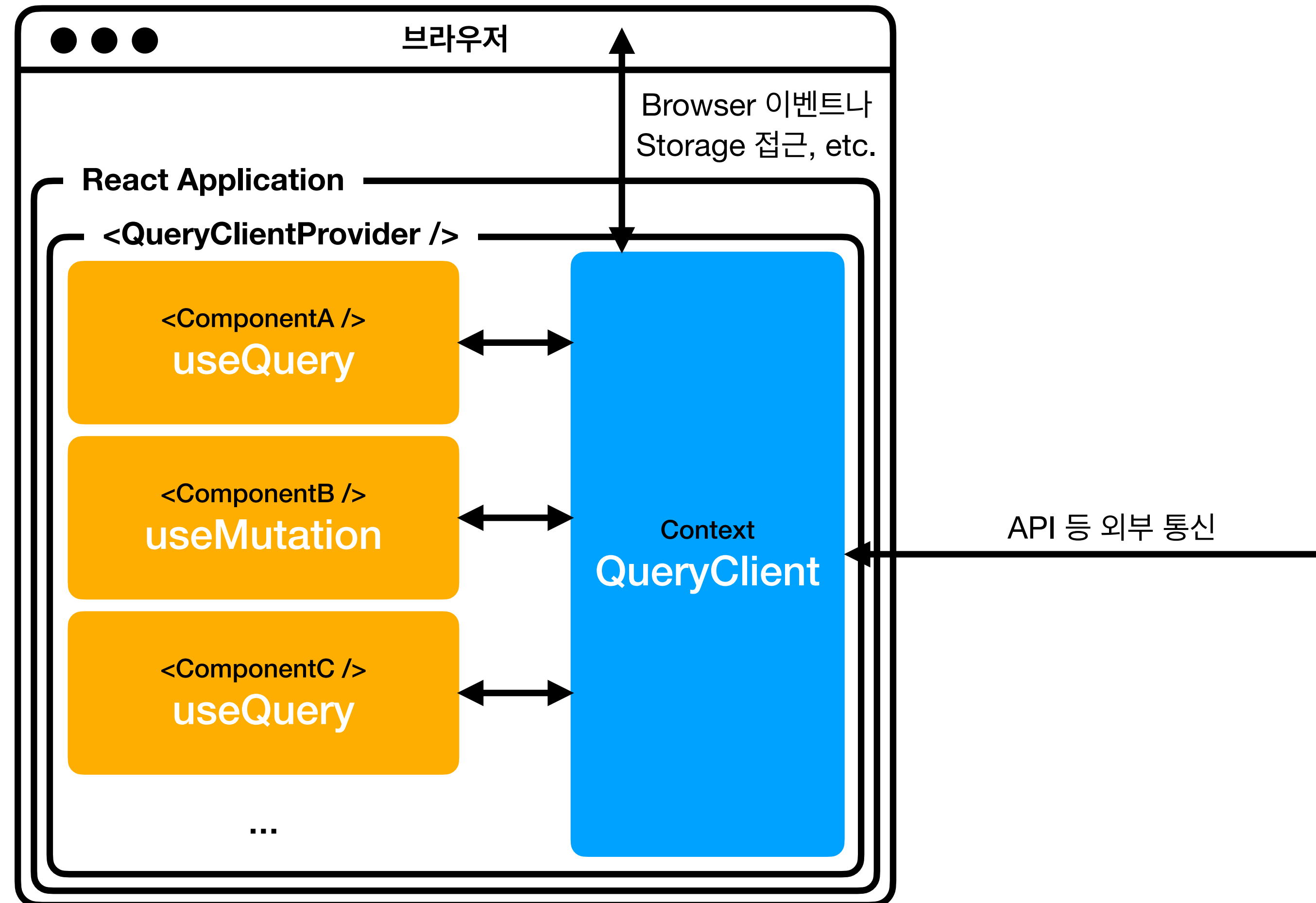
    return window.ReactQueryClientContext
  }

  return defaultContext
}

export const useQueryClient = () => {
  const queryClient = React.useContext(
    getClientContext(React.useContext(QueryClientSharingContext))
  )

  if (!queryClient) {
    throw new Error('No QueryClient set, use QueryClientProvider to set one')
  }

  return queryClient
}
```



# React Query etc.

- ☑ useInfiniteQuery
- ☑ Prefetching
- ☑ TypeScript 지원
- ☑ GraphQL도 대응
- ☑ SSR & Next.js에서도 사용가능
- ☑ devtools
- ☑ etc.

**React Query 이후 주문 FE 프로젝트**

# 어떻게 바뀌었나요



(Client) Store는

Client State들만 남아  
목적에 맞고 심플하게



Server State들은

React Query와 함께  
간단하고 파워풀



Component는

조금 길어졌지만..ㅎ  
Container 컴포넌트 같기도

# 이거 좋아요

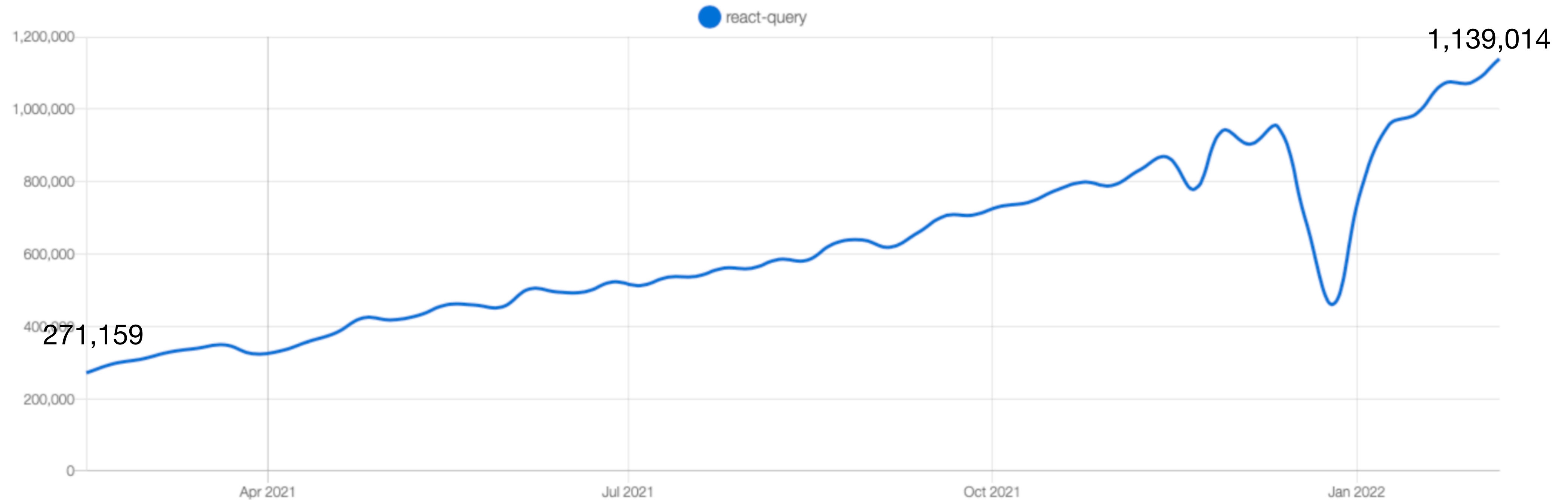
- ☑ 서버상태 관리 용이하며 (Redux, MobX 사용할 때보다) 직관적인 API 호출 코드
- ☑ API 처리에 관한 각종 인터페이스 및 옵션제공
- ☑ Client Store가 FE에서 정말로 필요한 전역상태만 남아 Store 답게 사용됨 (Boilerplate 코드 매우 감소)
- ☑ devtool 제공으로 원활한 디버깅
- ☑ Cache 전략 필요할 때 아주 좋음

# 이건 좀 더 고민이 필요할 거 같아요

- ❑ Component가 상대적으로 비대해지는 문제 (Component 설계/분리에 대한 고민 필요)
- ❑ 좀 더 난이도가 높아진 프로젝트 설계 (Component 유착 최소화 및 사용처 파악 필요)
- ❑ React Query의 장점을 더 잘 활용할 방법 (단순히 API 통신 이상의 가능성)

**저도 React Query를 써야 할까요?**

# 최근 React Query는요



1년 전(2021.02)보다 현재(2022.02)가 대략 4배 많은 npm 다운로드 수



# 최근 React Query는요



**중요한건 트렌드보단 Why!**

1년 전(2021.02)보다 현재(2022.02)가 대략 4배 많은 npm 다운로드 수

# 이런 분들에게 추천

- ❑ 수많은 전역상태가 API 통신과 얽여있어 비대해진 Store를 고민하시는 분
- ❑ API 통신 관련 코드를 보다 간단히 구현하고 싶으신 분
- ❑ FE에서 데이터 Caching 전략에 대해 고민하시는 분
- ❑ (공부가 목적이시라면) 모든 FE 개발자분들께!

# 이런 분들에게 추천

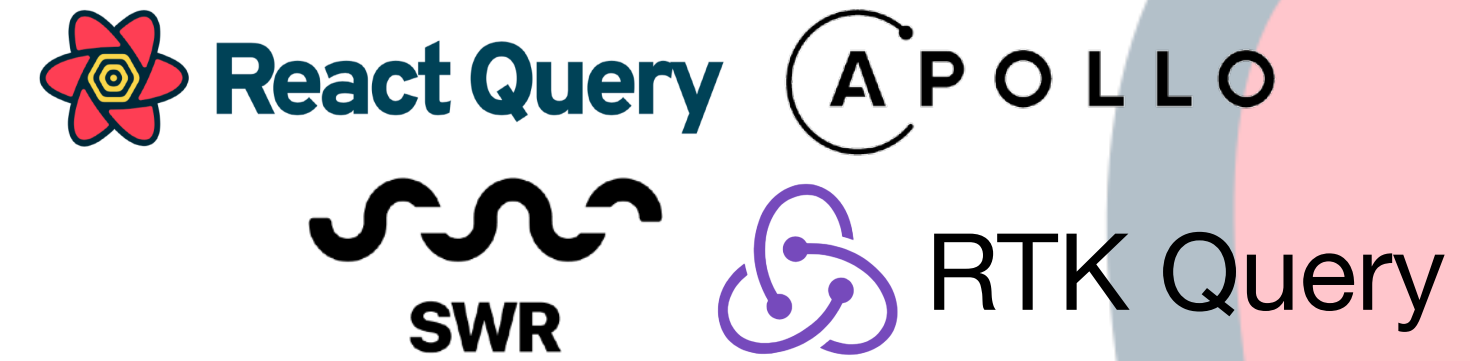
- ❑ 수많은 전역상태가 API 통신과 얽여있어 비대해진 Store를 고민하시는 분
- ❑ 반복되는 API 관련 Store 코드로 고민하시는 분

## ❑ 언제나 더 나은 방법이 나올 수 있으니

프로덕트를 만드는 우리 모두는 How, What에만 머물지 말고 Why를 고민해보아요

- ❑ (공부가 목적이시라면) 모든 FE 개발자분들께!

FE 상태와 상태관리



Client State / Server State

# React Query와 상태관리

useQuery, useMutation  
Query Invalidation

React Query 써야할까요?

Caching & Synchronization



**ۛۛ**