React実習

07. Web APIの利用

株式会社ジードライブ

今回学ぶこと

- ReactでのAjax通信の実装方法
  - Axiosの利用
  - TanStack Queryの利用

Axiosの利用

Axiosとは 【復習】

- ブラウザとNode.jsのためのPromiseベースのHTTP通信用ライブラリ
  - https://axios-http.com/ja/
  - scriptタグを使い、AxiosのJSファイルを読み込んで利用する方法とモジュールとして利用する方法がある

- モジュールとして利用する場合の手順

① npmでパッケージをインストール

```
npm install axios
```

② import文でモジュールを読み込む

```
import axios from "axios";
```

Axiosの使用方法 【復習】

- then()メソッド、またはasync/await構文で使用する

async/await構文での利用例

```
async function getDataFromApi() {
  try {
    const response = await axios.get('APIのURL');
    const data = response.data;
    // DOMへの追加等を行う
    console.log(response.data);
  } catch(error) {
    // エラー発生時の処理
    console.log(error);
  }
}
```

レスポンス

- レスポンスは以下のようなプロパティをもつ

```
const response = await axios.get('APIのURL');
```

プロパティ	説明
data	商品情報、天気予報などAPIが提供するデータ
status	HTTPステータスコード(200, 404 など数値)
statusText	HTTPステータスコード("OK", "Not Found" など文字列)

headers	以下のようなヘッダ情報オブジェクト
	
	{
	"content-type": "application/json",
	"date": "Mon, 27 Dec 2024 00:00:00 GMT"
	}

エラー

- エラーは、axios.isAxiosError(error) でAxiosのエラーか判別できる

```
} catch(error) {
    if(axios.isAxiosError(error) { … }
}
```

- Axiosのエラーがもつプロパティ

プロパティ	説明
code	"ERR_NETWORK", "ERR_BAD_REQUEST" などAxios独自のエラーコード
request response	リクエスト、レスポンスに関する情報をもつオブジェクトで、 status, statusTextなどのプロパティをもつ

パラメータの送信 【復習】

方法① クエリストリングとして送信する

```
const url = "https://api.openweathermap.org/data/2.5/weather" +
            "?lat=35&lon=139&appid=123ABC";
const response = await axios.get(url);
```

方法② データオブジェクトとして送信する

⇒ **params**プロパティにセットする

```
const url = "https://api.openweathermap.org/data/2.5/weather";
const parameter = {
    lat: 35, lon: 139, appid: '123ABC'
};
const response = await axios.get(url, {params: parameter});
```

各種リクエストに対応するメソッド

- HTTPリクエストメソッドには、GET, POST以外にも、PUTやDELETEといった種類のメソッドも存在する
  - これらのリクエストメソッドは、データの更新や削除に利用される
- axiosはGETリクエストに対応するget()メソッド以外にも、POST, PUT, DELETEリクエストに対応するメソッドをもつ
  - post(), put(), delete()メソッドが用意されている

書式

```
axios.post(url, data, config)
axios.put(url, data, config)
axios.delete(url, config)
```

※ 第２引数、第３引数は省略可

axios.post()の使用例

例: 会員データを追加する

```
const url = "http://example.com/api/members";
const data = { name: "山田太郎", age: 25 };
const config = {
    headers: {"Content-Type": "application/json"}
};

const response = await axios.post(url, data, config);
```

fetchと違い、JSON.stringify()
によるデータのJSON文字列化が不要

json-server使用時は明示不要だが、
SpringBootでAPIを作成する際は必要

axios.create()

- axios.create()メソッドを使うことで、ベースとなるURLやヘッダ等の共通設定を施したaxiosインスタンスが利用できる

例：共通設定を施したaxiosの利用

```
const ax = axios.create({
   baseURL: "http://example.com/api",
   headers: { "Content-Type": "application/json" }
});

const res1 = await ax.get("/members");
const res2 = await ax.post("/item", { name: "りんご" });
const res3 = await ax.delete("/item/10");
```

共通の設定

ReactでのAjax通信 【復習】

- 初期データの取得時は、基本的にuseEffectフックを使用する

例：Reactコンポーネント内で、Web APIからユーザーデータを取得する

```
const [isLoading, setIsLoading] = useState(true);
const [users, setUsers] = useState([]);

useEffect(() => {
    const getUserData = async () => {
      setIsLoading(true);
      const res = await axios.get(https://exapmle.com/api/users);
      setUsers(res.data);
      setIsLoading(false);
    };

    getUserData();
}, []);
```

ReactでのAjax通信

- 必ずしもuseEffectが必要なわけではなく、イベントハンドラ内に記述することもできる

例：フォーム送信を通じて、Web APIにデータを送る

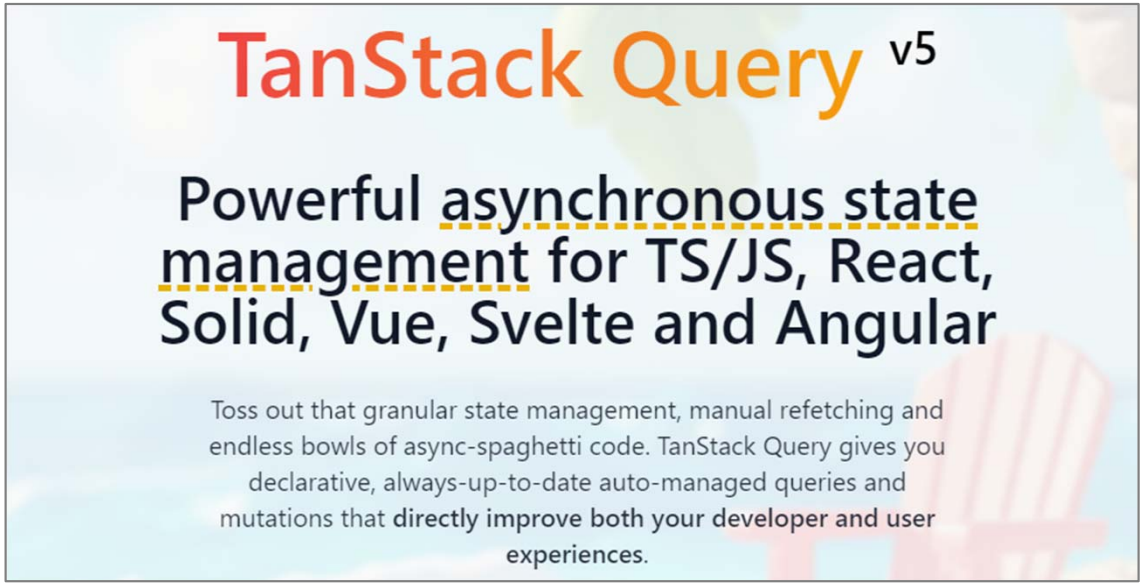
```
const handleSubmit = async (e) => {
    e.preventDefault();
    const data = {
      name: nameRef.current.value,
      age: ageRef.current.value
    };
    const url = "http://example.com/api/users";
    const config = { headers: {"Content-Type": "application/json"} };
    const res = await axios.post(url, data, config);
    setMessage(res.data);
};
```

練習

- 練習07-1
- 練習07-2
- 練習07-3

TanStack Query

- Web APIとの連携を補助するライブラリ
  - https://tanstack.com/query/latest
  - 元々は**React Query**という名称だったが、現在はTanStack Queryに名称が変更され、React以外でも利用できる



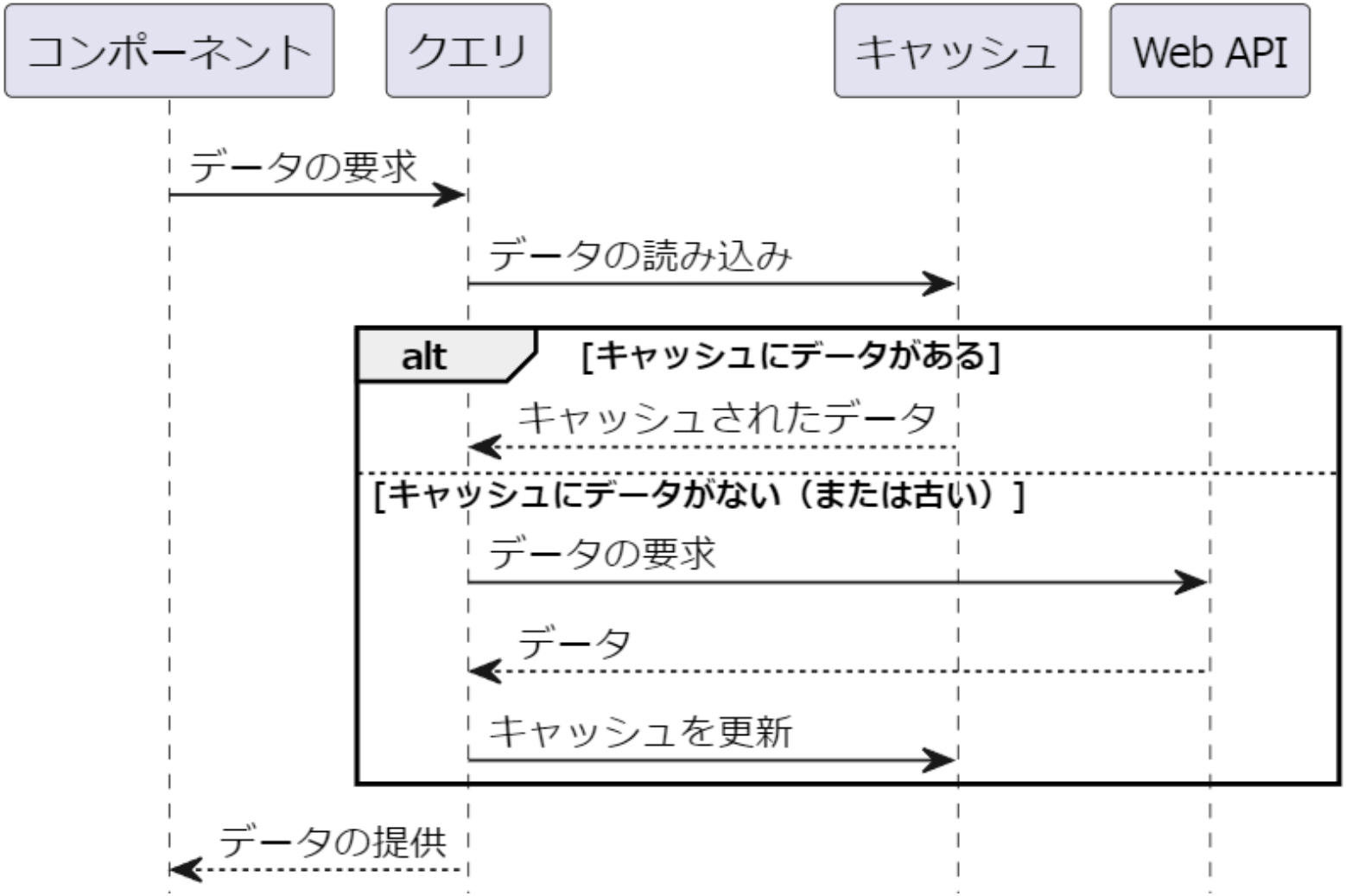
TanStack Query

- TanStack Queryを使うことで、Web API上のデータを取得・表示しているコンポーネントの状態を最新に保つことが容易になる
  - TanStack Queryを使用しない場合、コンポーネントの状態を最新に保つには、setInterval関数などを使い、定期的にWeb APIからデータを取得するための実装が必要になる
  - Web APIと連携する手段自体は、FetchやAxiosを使い、実装しておく必要がある
- TanStack Queryは、ローディング中やエラー発生といったWeb APIとの連携に係る状態の管理も補助する
  - useStateやuseEffectの記述が不要になる

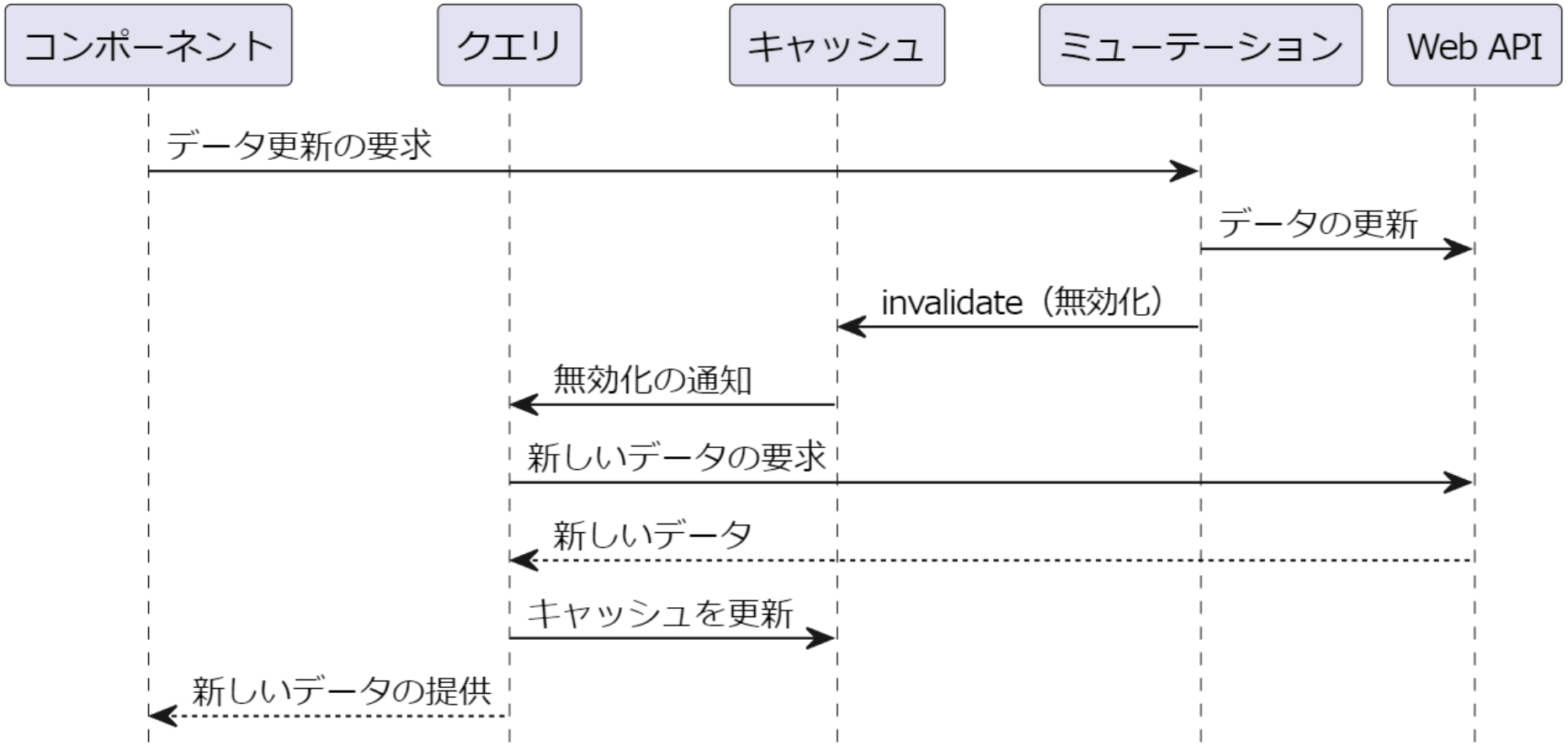
TanStack Queryのコアコンセプト

- クエリ
  - Web APIへのデータの取得要求 (APIからのデータ取得操作)
  - APIから取得されたデータはキャッシュとして保存される
- キャッシュ
  - クエリによって取得されたデータを保存しておき、同データに対するWeb APIへの再リクエストを減らすための機能
  - キャッシュとして保存しておく期間や、データの再取得までの期間を設定することができる
- ミューテーション
  - Web APIに対してのデータ変更要求 (データの追加、更新、削除操作)
  - この操作に合わせて、キャッシュを更新することも可能

クエリとキャッシュ



ミューテーション



TanStack Queryの利用手順

1. パッケージをインストールする

```
npm install @tanstack/react-query
```

2. Tanstack Queryを利用したいコンポーネントを QueryClientProviderコンポーネントの子孫にする
3. FetchやAxiosを使い、サーバー上のデータと連携する関数を作成する
4. 作成した関数をuseQueryやuseMutationフックを通じて利用する

TanStack Queryの導入

- Tanstack Queryを利用するコンポーネントを QueryClientProviderコンポーネントの子孫にする
  - 開発者向けのツールを利用する場合は、ReactQueryDevtools コンポーネントを配置する

```
const queryClient = new QueryClient();

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
      <QueryClientProvider client={queryClient}>
        <App />
        <ReactQueryDevtools />
      </QueryClientProvider>
  </StrictMode>
);
```

QueryClientを作成する

TanStack Query用の開発者ツール
・開発環境でのみ表示されるツール
・別途インストールが必要

```
npm install @tanstack/react-query-devtools
```

TanStack Queryの導入

- QueryClientを生成する際には、gcTimeやstaleTime等の基本設定ができる（クエリごとの設定も可能）
  - gcTime: コンポーネントに参照されていない(表示されていない)データのキャッシュを破棄(Garbage Collection)するまでの時間
  - staleTime: データをキャッシュしてから、それを「古い」と見なすまでの期間。古いキャッシュは、コンポーネントが再描画される際に更新される

```
const queryClient = new QueryClient({
    defaultOptions: {
      queries: {
        gcTime: 10 * 60 * 1000,  //10分(600,000ミリ秒)
        staleTime:  5 * 60 * 1000 //5分(300,000ミリ秒)
      }
    }
});
```

gcTimeの初期値は5分

staleTimeの初期値は0
⇒ 取得直後から「古い」と見なされる

一定時間ごとにデータを再取得する場合は、refetchIntervalを設定する

useQueryフック

- データを取得・管理するためのフック
  - 引数：クエリキーやデータ取得関数などを含むオブジェクト
  - 戻り値：取得されたデータ、エラー情報などを含むオブジェクト

記述例

```
// Web APIからデータを取得するための関数を定義
const getProducts = async () => {
    const res = await axios.get("http://example.com/products");
    return res.data;
}

const result = useQuery({
    queryKey: ["products"], queryFn: getProducts
});
```

APIからデータを取得する関数を指定

クエリ(キャッシュ)を一意に識別するためのキーで配列形式で指定する

参考：https://tanstack.com/query/latest/docs/framework/react/reference/useQuery

useQueryフックの引数

- 引数には以下のようなプロパティをもつオブジェクトを指定することができる

プロパティ	説明
queryKey（必須）	クエリ(とそれに紐づくキャッシュ)を一意に識別するためのキー。基本的に配列形式で指定する ⇒ ["product", "apple"] とした場合、productとappleという2つのキーを持つという意味ではなく、配列全体がキーとなる
queryFn（必須）	APIからデータを取得するための関数を指定する
gcTime	参照されていないキャッシュを破棄するまでの時間(ミリ秒)
staleTime	キャッシュを古いと見なすまでの時間(ミリ秒)
refetchInterval	データを再取得する間隔(ミリ秒)。初期値はfalse
initialData placeholderData	クエリ実行前に使用される初期データ。initialDataはキャッシュされるが、placeholderDataはキャッシュされない ⇒ 初期データが既知の場合はinitialData, 未知だが何か表示したい場合はplaceholderDataを使用する

useQueryフックの戻り値

- 戻り値オブジェクトには以下のようなプロパティが含まれている

プロパティ	説明
data	queryFnで指定した関数により取得されるデータを格納する データが取得されるまでは「undefined」
error	queryFnで指定した関数がスローするエラー。error.messageでエラーメッセージを取得することができる。エラーが発生していない場合は「undefined」
isError	データ取得中はfalseで、取得中にエラーがスローされた場合にtrueになる
isSuccess	データ取得中はfalseで、取得に成功した場合にtrueになる
isLoading isFetching	キャッシュが存在しない、またはキャッシュが古い状態で、データ取得中にtrueになり、取得が成功(または失敗)するとfalseになる 古くないキャッシュが存在している場面で、データを再取得することがあるが、その場合isLoadingはfalseになり、isFetchingはtrueになる

useQueryフックの戻り値

- useQueryの戻り値を使うことで、ローディングやエラーなどのステート管理が容易になる
  - 戻り値は、分割代入で取得することも可能

例: データ、ローディング状態、エラーの有無、エラー情報の取得と利用

```
const { data, isLoading, isError, error } = useQuery({
    queryKey: ["products"], queryFn: getProducts
});

if(isError) { return <p>{error.message}</p> }
if(isLoading) { return <p>...Now Loading</p> }

return (<ul>
   {data.map(product => (
      <li key={product.id}>{product.name}</li>
   ))}
</ul>);
```

queryKeyの設定

- クエリキーはクエリを一意に識別するもので、配列で指定できる

例:製品の個別表示コンポーネント
URLが /product/**3** のようになり、末尾に製品のID番号が含まれることを想定

```
const getProductById = async (productId) => {
    const url = `http://example.com/products?id=${productId}`
    const res = await axios.get(url);
    return res.data;
}

const params = useParams();

const {data, isLoading} = useQuery({
    queryKey: ["product", params.id],
    queryFn: () => getProductById(params.id)
});
```

queryKeyの設定

- クエリキーは配列で指定できるため、階層構造をもつようなデータのキャッシュ管理を容易に行うことができる
  - この時、データの型に注意する必要がある

以下はそれぞれ異なるクエリを示すことになる

```
queryKey: ["product", "25"]
```

```
queryKey: ["product", 25]
```

特にinvalidateQueriesメソッド(後述)で
キャッシュを更新する際に気を付ける

練習

- 練習07-4

useQueryClientフック

- コンポーネント内にQueryClientインスタンスを呼び出すためのフック
- QueryClientインスタンスを使うことで、クエリを操作することができる
  - invalidateQueriesメソッドを使い、キャッシュされているデータを古いものと見なし、データの再取得を促すことができる

例：取得済みのキャッシュを更新

```
const queryClient = useQueryClient();
queryClient.invalidateQueries({ queryKey: ["products"] });
```

useQueryで設定したキー

参考：https://tanstack.com/query/latest/docs/reference/QueryClient

useMutationフック

- Web API上のデータを変更するために使用するフック
  - ミューテーションは、変更を加えるという意味で、データを追加、更新、削除する際に利用する

例：データを削除し、取得済みのキャッシュを更新する

```
const { id } = useParams();
const queryClient = useQueryClient();
const deleteMutation = useMutation({
    mutationFn: () => deleteProduct(id),
    onSuccess: () =>
        queryClient.invalidateQueries({queryKey: ["products"]})
});

const onClickDeleteButton = () => {
    deleteMutation.mutate();
}
```

ミューテーション関数の指定

ミューテーション成功時にキャッシュを更新

ミューテーションが実行される

参考：https://tanstack.com/query/latest/docs/framework/react/reference/useMutation

useMutationフックの引数

- 引数には以下のようなプロパティをもつオブジェクトを指定することができる

プロパティ	説明
mutationFn（必須）	ミューテーション関数（データ変更のための関数）
onMutate	ミューテーション開始直前に呼び出される処理
onSuccess	ミューテーション成功時の処理
onError	ミューテーション失敗時の処理
onSettled	成功/失敗に関わらず、ミューテーション完了時に呼び出される処理
retry	ミューテーションが失敗した際の再試行回数、または再試行の条件を指定
retryDelay	再試行するまでの遅延時間をミリ秒単位で指定する

useMutationフックの戻り値

- 戻り値オブジェクトには以下のようなプロパティが含まれている

プロパティ	説明
mutate mutateAsync	ミューテーションを実行するための関数 async/awaitを使い実行したい場合は、mutateAsync関数を使用する
data	ミューテーション成功時に返されるデータ ミューテーション未実行の場合は「undefined」
error	ミューテーション失敗時のエラー。エラー未発生時は「null」
isSuccess isError	ミューテーションの成功/失敗を判別するためのプロパティ
isPending	ミューテーションが進行中であることを示すプロパティ

mutate関数の利用例

- mutateの第1引数は、mutationFnに渡すことができる

```
const nameRef = useRef();
const priceRef = useRef();
const queryClient = useQueryClient();
const addMutation = useMutation({
    mutationFn: (product) => addProduct(product),
    onSuccess: () =>
        queryClient.invalidateQueries({queryKey: ["products"]})
});

const onClickAddButton = () => {
    const name = nameRef.current.value;
    const price = Number(priceRef.current.value);
    addMutation.mutate({name, price});
}
```

練習

- 練習07-5