React Query

useEffect + fetch/axios

We've seen the use of `useEffect` and data fetching APIs.

This approach requires us to recognize that we cannot use `async` methods as the `useEffect` callback.

We also need to be aware of the need to correctly specify the dependency `array`

Given that we, and any other developer, can create a custom hook. Several libraries have been developed to make data fetching easy. We'll be looking at `React Query`

React Query

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

Excellent! We won't have to rely on `useState` + `useEffect` + `fetch`. React Query will combine all of this for us!

React Query

Motivation

Installation

```
npm install react-query
```

Setup

Much like React Router we will need to wrap our <App> in code to configure React Query

```
const queryClient = new QueryClient()

ReactDOM.render(
  <React.StrictMode>
    <Router>
      <QueryClientProvider client={queryClient}>
        <App />
      </QueryClientProvider>
    </Router>
  </React.StrictMode>,
  document.getElementById('root')
)
```

Now we can modify our code to start using React Query

Start with fetching the list of todo items in `<TodoList>`

Remove state management with `useState`

Delete this line:

```
const [todoItems, setTodoItems] = useState<TodoItemType[]>([])
```

Add code to load the todo items

```
// Function to return the axios data.

async function getTodos() {
  //                                  This describes the format of `data`
  //                                  vvvvvvvvvvvvvv
  const response = await axios.get<TodoItemType[]>(
    'https://one-list-api.herokuapp.com/items?access_token=cohort22'
  )

  return response.data
}
```

Use the function with `useQuery`

```
//
//    The data returned from axios
//       |
//       |              Function to let us reload the data (renamed)
//       |                 |
//       |                 |                   Unique identifier for this query
//       |                 |                      |
//       |                 |                      |     Function that returns a Promise
//       |                 |                      |       |
//       v                 v                      v       v
const { data: todoItems, refetch } = useQuery('todos', getTodos)
```

This replaces:

- `useState` for `todoItems`
- `useEffect` to load items
- `loadAllItems`

Notice we get an "object is possibly undefined"

Add a default value for the todoItems

```
const { data: todoItems = [], refetch } = useQuery('todos', getTodos)
```

We can also detect when the query is actively loading

```
const { data: todoItems = [], refetch, isLoading } = useQuery('todos', getTodos)

// ...
// ...
// ...

if (isLoading) {
  return <div>Loading</div>
}
```

Replace use of `loadAllItems` with `refetch`

Lots of refactoring ... reduced complexity

- No `useEffect + useState` combo
- Loading state
- Refetch function
- More... See documentation (caching, etc.)

TodoItemPage

```
async function getOneTodo(id: string) {
  const response = await axios.get<TodoItemType>(
    `https://one-list-api.herokuapp.com/items/${id}?access_token=cohort22`
  )

  return response.data
}
```

```
//
//                                          Dynamic unique identifier based
//                                          on the id parameter from router
//                                               |                  |
//                                               vvvvvvvvvvvvvvvvvvv
const { data: todoItem, isLoading } = useQuery(['todo', params.id], () =>
  getOneTodo(params.id)
)
```

TodoList create a todo

Mutations

Define function to create todo item

```
async function createNewTodoItem(newTodoText: string) {
  return await axios.post(
    'https://one-list-api.herokuapp.com/items?access_token=cohort42',
    { item: { text: newTodoText } }
  )
}
```

Define a mutation

Place right below our existing useQuery

```
const todoItemMutation = useMutation((newTodoText: string) =>
  createNewTodoItem(newTodoText)
)
```

or

```
const todoItemMutation = useMutation(function (newTodoText: string) {
  return createNewTodoItem(newTodoText)
})
```

Use the mutation where we'd want the todo item created

The arguments to `mutate` become the arguments to our mutation function.

```
todoItemMutation.mutate(newTodoText)
```

How to handle calling code when the mutation is done?

- `onSuccess`
- `onError`
- `onSettled`

```
const todoItemMutation = useMutation(
  (newTodoText: string) => createNewTodoItem(newTodoText),
  {
    onSuccess: function () {
      refetch()

      setNewTodoText('')
    },
  }
)
```

```
function handleCreateNewTodoItem() {
  todoItemMutation.mutate(newTodoText)
}
```

Mark item complete

Define a method

```
async function toggleItemComplete(id: number | undefined, complete: boolean) {
  const response = axios.put(
    `https://one-list-api.herokuapp.com/items/${id}?access_token=cohort22`,
    { item: { complete: !complete } }
  )

  return response
}
```

```
const toggleMutation = useMutation(() => toggleItemComplete(id, complete), {
  onSuccess: function () {
    reloadItems()
  },
})
```

```
async function toggleCompleteStatus() {
  toggleMutation.mutate()
}
```

TodoItemPage delete

Define function:

```
async function deleteOneTodo(id: string) {
  const response = await axios.delete(
    `https://one-list-api.herokuapp.com/items/${id}?access_token=cohort22`
  )

  return response
}
```

Define mutation

```
const deleteMutation = useMutation((id: string) => deleteOneTodo(id), {
  onSuccess: function () {
    // Send the user back to the homepage
    history.push('/')
  },
})
```

Use mutation

```
async function deleteTodoItem() {
  deleteMutation.mutate(params.id)
}
```

Benefit: organize all the API code in one place: `api.ts`

- Create a module: `api.ts`
- Move all the get/load functions into that file
- Now we have one single place where all API logic is located

Other benefits of React Query

- Pagination
- Infinite Queries
- Window Focus Refetching
- Caching
- Query Cancellation
- Update From Mutation
- Invalidating Queries

Advanced Topics

Define custom hooks!

We can refactor our example of deleting an item into a custom hook.

useDeleteItemMutation hook

- Define a method that starts with use (requirement of hooks)
- Move implementation into this method and have it return the `useMutation`

```
function useDeleteItemMutation(id: string) {
  const history = useHistory()

  return useMutation(() => deleteOneTodo(id), {
    onSuccess: function () {
      // Send the user back to the homepage
      history.push('/')
    },
  })
}
```

Use our new hook

```
const deleteMutation = useDeleteItemMutation(params.id)

async function deleteTodoItem() {
  deleteMutation.mutate()
}
```

Define a mutation for loading a single todo item

```
function useLoadOneItem(id: string) {
  const { data: todoItem, isLoading } = useQuery(['todo', id], () =>
    getOneTodo(id)
  )

  return { todoItem, isLoading }
}
```

Use the new custom hook

```
const { todoItem, isLoading } = useLoadOneItem(params.id)
```

Refactor all this code into a common file: `api.ts`

```
import React from 'react'
import { useParams } from 'react-router'
import { Link } from 'react-router-dom'
import { useDeleteItemMutation, useLoadOneItem } from './api'

export function TodoItemPage() {
  const params = useParams<{ id: string }>()
  const { todoItem, isLoading } = useLoadOneItem(params.id)
  const deleteMutation = useDeleteItemMutation(params.id)

  async function deleteTodoItem() {
    deleteMutation.mutate()
  }

  if (isLoading) {
    return <div>Loading...</div>
  }

  return (
    <div>
      <p>
        <Link to="/">Home</Link>
      </p>
      <p className={todoItem.complete ? 'completed' : ''}>{todoItem.text}</p>
      <p>Created: {todoItem.created_at}</p>
      <p>Updated: {todoItem.updated_at}</p>
      <button onClick={deleteTodoItem}>Delete</button>
    </div>
  )
}
```

Separation of Concerns

- `TodoItemPage.tsx` only concerns itself with showing a todo item
- `api.ts` contains all the code for loading a todo item
- However, `api.ts` has UI code in it.

```
export function useDeleteItemMutation(id: string) {
  const history = useHistory()

  return useMutation(() => deleteOneTodo(id), {
    onSuccess: function () {
      // Send the user back to the homepage
      history.push('/')
    },
  })
}
```

Leave UI code in the UI

```
export function useDeleteItemMutation(id: string, onSuccess: () => void) {
  return useMutation(() => deleteOneTodo(id), { onSuccess })
}
```

Update the UI

```
const deleteMutation = useDeleteItemMutation(params.id, function () {
  history.push('/')
})
```

Architecture Choice

- Combined
  - API
  - CSS (see styled components)
  - State
  - Behavior
- Separate concerns
  - CSS all in one file
  - (see CSS Modules
  - `api.ts`