CMPS 356

State Management

Dr. Abdelkarim Erradi

CSE@QU

Outline

1. Client/App state
    - useState
    - useReducer
    - useContext
    - Zustand
2. Server State Cache using React Query

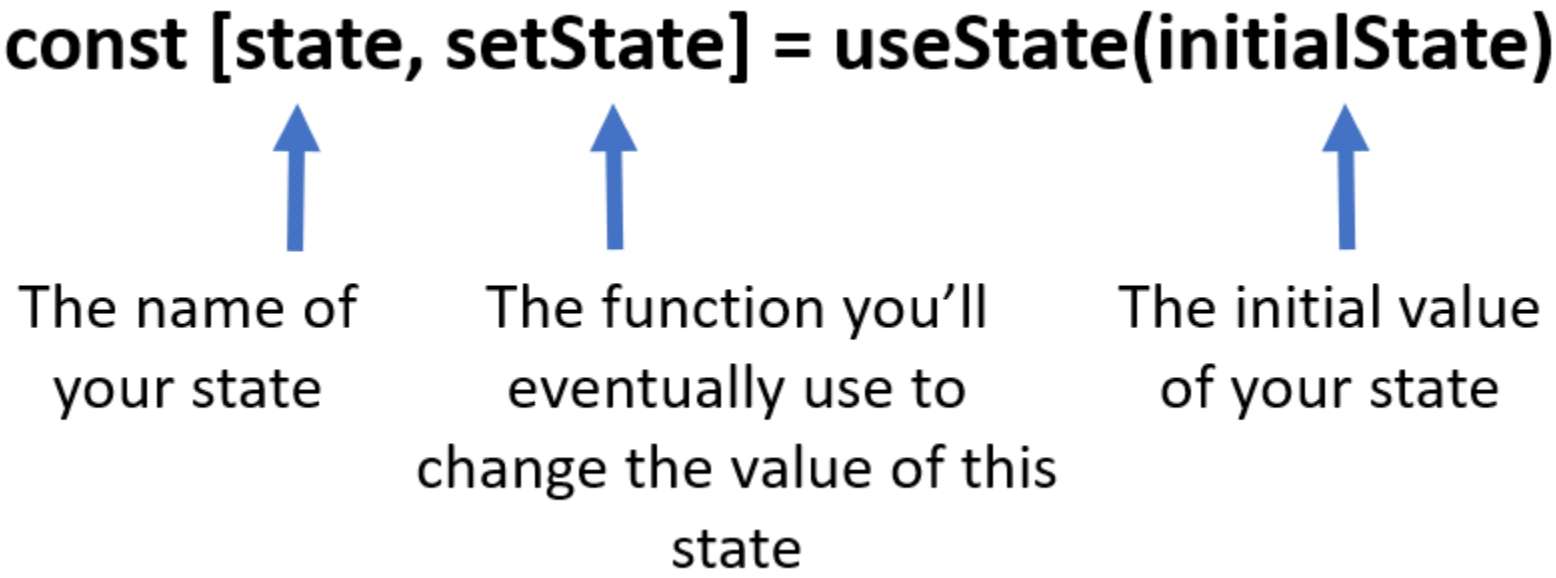
Client State

State Management

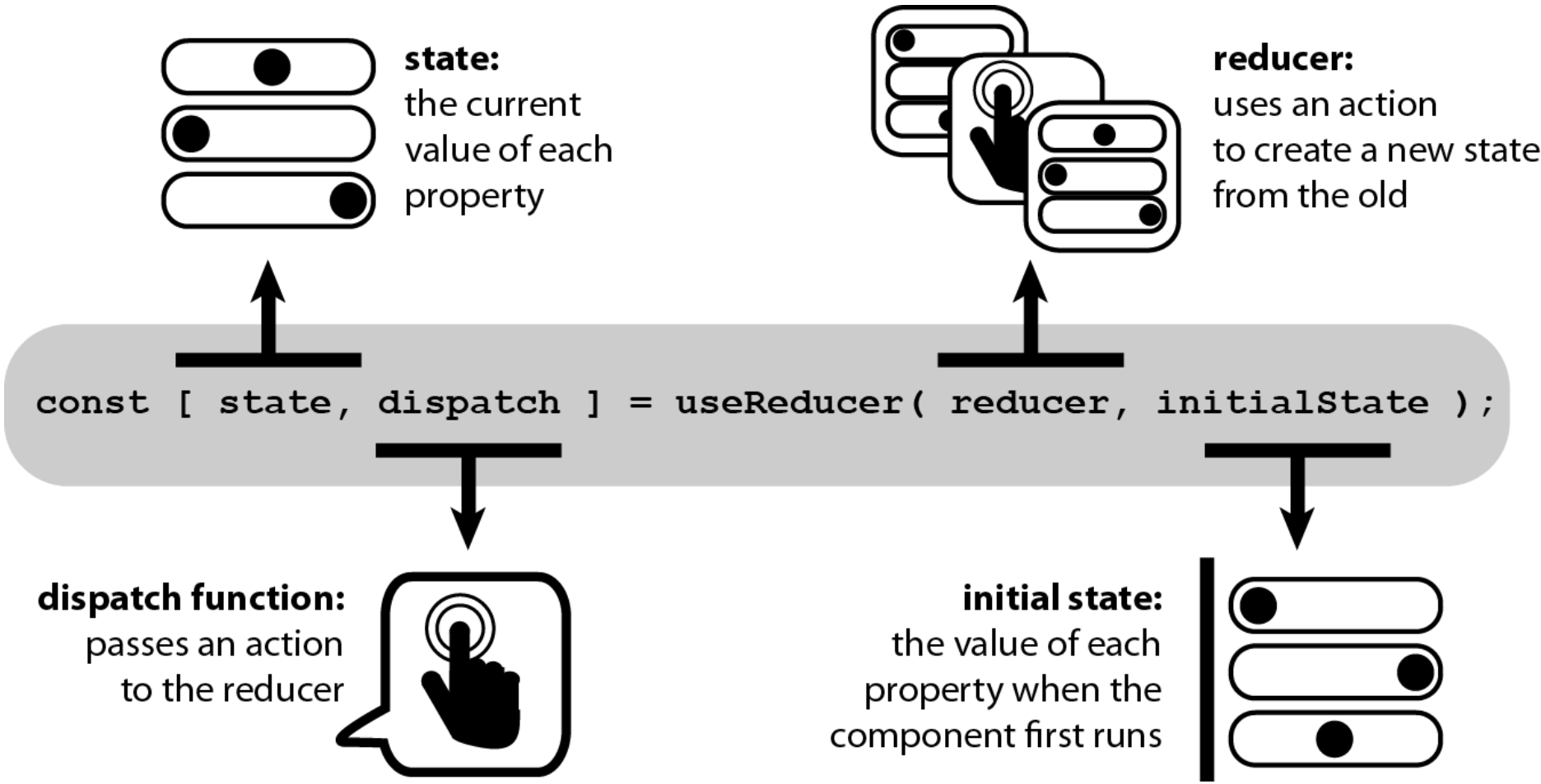
- State, in React, is any data that represents the user interface (UI)
- States can change over time, and React takes care of components re-rendering to reflect the new state
- State Management Hooks
  - **useState** : manage basic state variables
  - **useReducer:** manage multiple related state variables + Centralized, action-based state management
  - **useContext:** share data with child components without prop drilling
  - Use a third-party party state management library such as **Zustand** to manage shared state between components

useState: creates a state variable

- Used for basic state management inside a component



useReducer: manage multiple related state variables

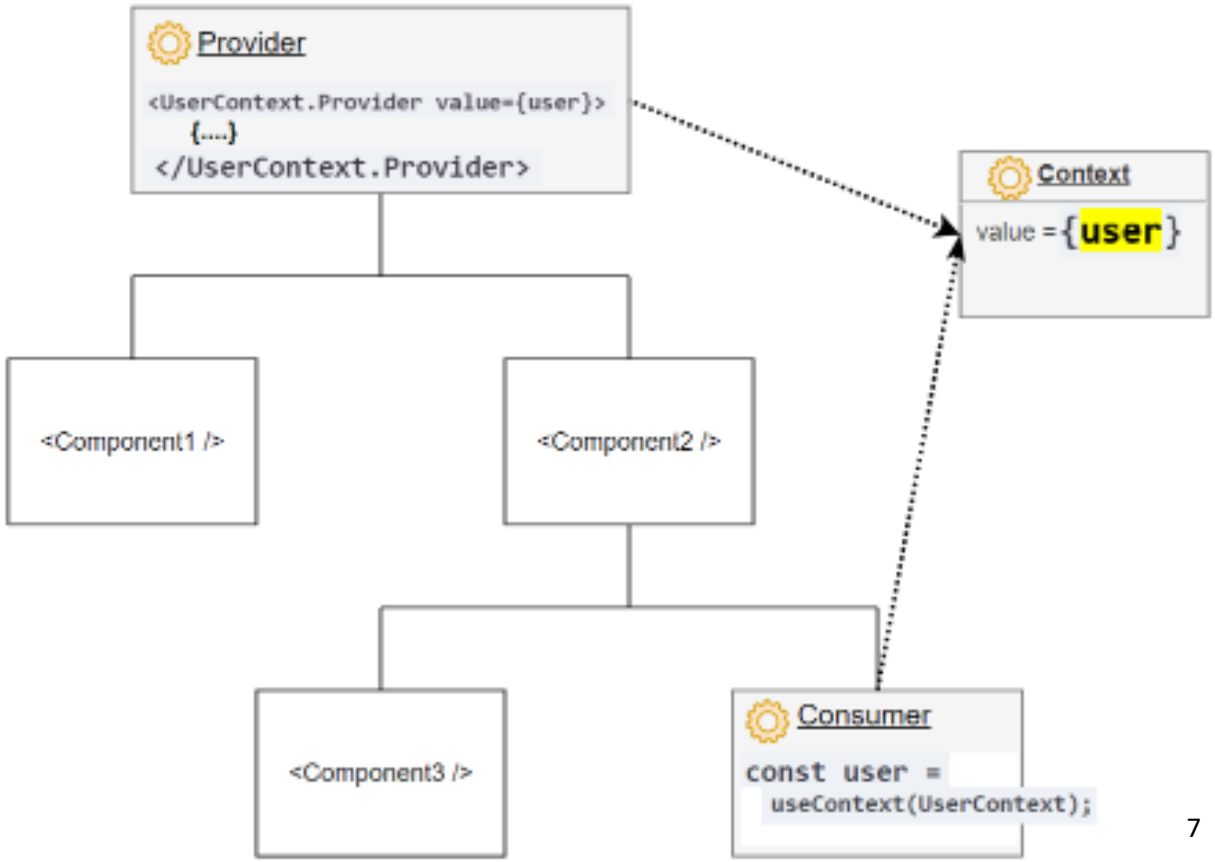


useContext

- Share state (e.g., current user, user settings) between deeply nested components more easily than prop drilling (i.e., without pass the state as props through each nested component)

- Using the context requires 3 steps: creating, providing, and consuming the context

- If the context variables change then all consumers are notified and re-rendered



useContext – provides shared variables and functions

1. **Create a context** instance (i.e., a container to hold shared variables and functions)

```
import React from 'react';
const UserContext = React.createContext();
export default UserContext;
```

2. **Provider places shared variables / functions in the context** to make them available to child components

```
import UserContext from './components/UserContext';
function App() { return (
     <UserContext.Provider value={ user }>
            <Welcome /> …
      </UserContext.Provider>
   ); }
```

3. **Consumer access the shared variables / functions in the context**

```
import React, {useContext} from "react";  import UserContext from './UserContext';
export default function Welcome() {
    const user = useContext(UserContext);
    return <div>You are login as: {user.username}</div>;
}
```


ZUS
TAND

Zustand

- Zustand is a small library that **simplifies** state-management and requires little of boilerplate to create shareable global store accessible everywhere in the app
  - Centralized, action-based state management
  - E.g., a signed-in user object can be used to figure out what content we should be display or to restrict access to some pages by using route guards and redirect a user if they are not signed-in
- Make sure you don't put everything in a global state. Otherwise, the app will quickly become more complex than it needs to be and harder to maintain
  - As a rule of thumb, try to **put state as close as possible to where it needs to be used** and make it global only when it's truly necessary

Zustand Programming Steps

1. Create a Zustand store using **`create`** function and pass to it a state creator function that returns the state object

- The state creator function receives **`set`** and **`get`** arguments

```
export const useStore = create((set, get) => ({
  fruits: ["apple", "banana", "orange"],
  addFruit: (fruit) => {
    set((state) => ({ fruits: [...state.fruits, fruit] }))
  }})
```

- You can put anything in the store: primitives, objects, functions. State must be updated immutably using the **set** function that merges state

2. The **`create`** method returns a hook that can be used to access the store and the component will re-render on changes

```
  // Get access to the whole store
  const store = useStore()
  // Get access to specific properties by passing a selector
  const fruits = useStore((state) => state.fruits)
```

get function

- **get** function can be used to access the current state of the store, especially helps when we want to access the state within an action **without using the set function**

```
const useStore = create((set, get) => ({
  count: 0,
  action: () => {
    const count = get().count
    // ...
  }
})
```

Async operations

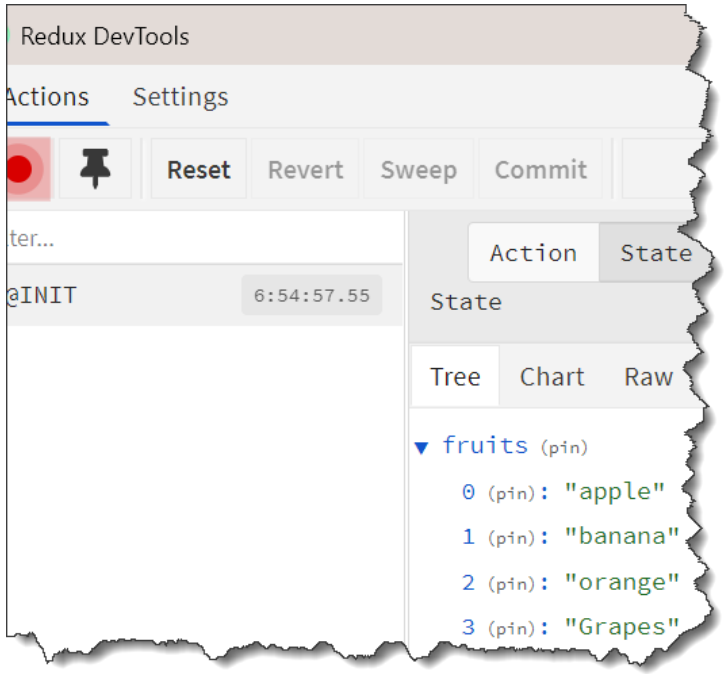
- Zustand supports async operations such as reading/writing data from Web API

```
let store = (set) => ({
  fruits: [],
  ...
  fetch: async () => {
    const response = await
      fetch('http://test.com/fruits')
    set({ fruits: await response.json() })
  }
})
```

devtools middleware

- Redux DevTools for debugging the app state changes (i.e., inspecting the current state as well as tracking the state updates)

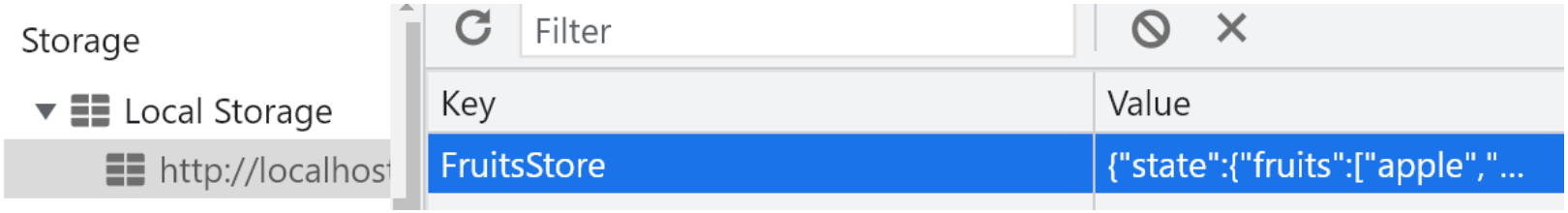
```
import create from 'zustand'
import {devtools} from "zustand/middleware"

let store = (set) => ({
  fruits: ["apple", "banana", "orange"],
  addFruit: (fruit) => {
    set((state) => ({
      fruits: [...state.fruits, fruit],
    }))
  }
})
store = devtools(store);
// create the store
export const useStore = create(store);
```



persist middleware

- **persist** middleware could be used to persist a Zustand store, so we don't lose its state if a user refreshes the website or comes back later

```
import create from 'zustand'
import {presist} from "zustand/middleware"

let store = (set) => ({
  fruits: ["apple", "banana", "orange"],
  addFruit: (fruit) => {
    set((state) => ({ fruits: [...state.fruits, fruit] }))
}})
store = persist(store, {name: "FruitsStore"})
// create the store
export const useStore = create(store);
```



Slicing the store into smaller stores

- Your store can become bigger and harder to maintain as you add more features. You can divide the store into smaller individual stores to achieve modularity

```
export const createFishSlice = (set) => ({
  fishes: 0,
  addFish: () => set((state) => ({ fishes: state.fishes + 1 })),
})
```

```
export const createBearSlice = (set) => ({
  bears: 0,
  addBear: () => set((state) => ({ bears: state.bears + 1 })),
  eatFish: () => set((state) => ({ fishes: state.fishes - 1 })),
})
```

```
import create from 'zustand'
import { createBearSlice } from './bearSlice'
import { createFishSlice } from './fishSlice'

export const useStore = create((set) => ({
  ...createBearSlice(set),
  ...createFishSlice(set),
}))
```

Usage in a React component

```
function App() {
  const bears = useStore((state) => state.bears)
  const fishes = useStore((state) => state.fishes)
  const addBear = useStore((state) => state.addBear)
```

Server State Cache using React Query

Server State Cache

- Server state has some unique characteristics and challenges
  - Is persisted remotely in a location you do not control or own
  - Requires asynchronous APIs for fetching and updating
  - Multi-users app implies that the data can be changed by others => can potentially become "out of date" on the client-side
- React-Query is a feature-rich library that can be used for asynchronous fetching, caching, synchronizing and updating server state

React-Query Features

- Tracks loading and error states of server queries
- Makes **cached server data** available for display while you're fetching updated data
  - Updating "out of date" data in the background
  - Reflecting updates to data as quickly as possible
- Deduping multiple requests for the same data into a single request
- Performance optimizations like pagination, lazy loading data, and infinite scroll
- Managing memory and garbage collection of server state
- Retry on error

useQuery Hook

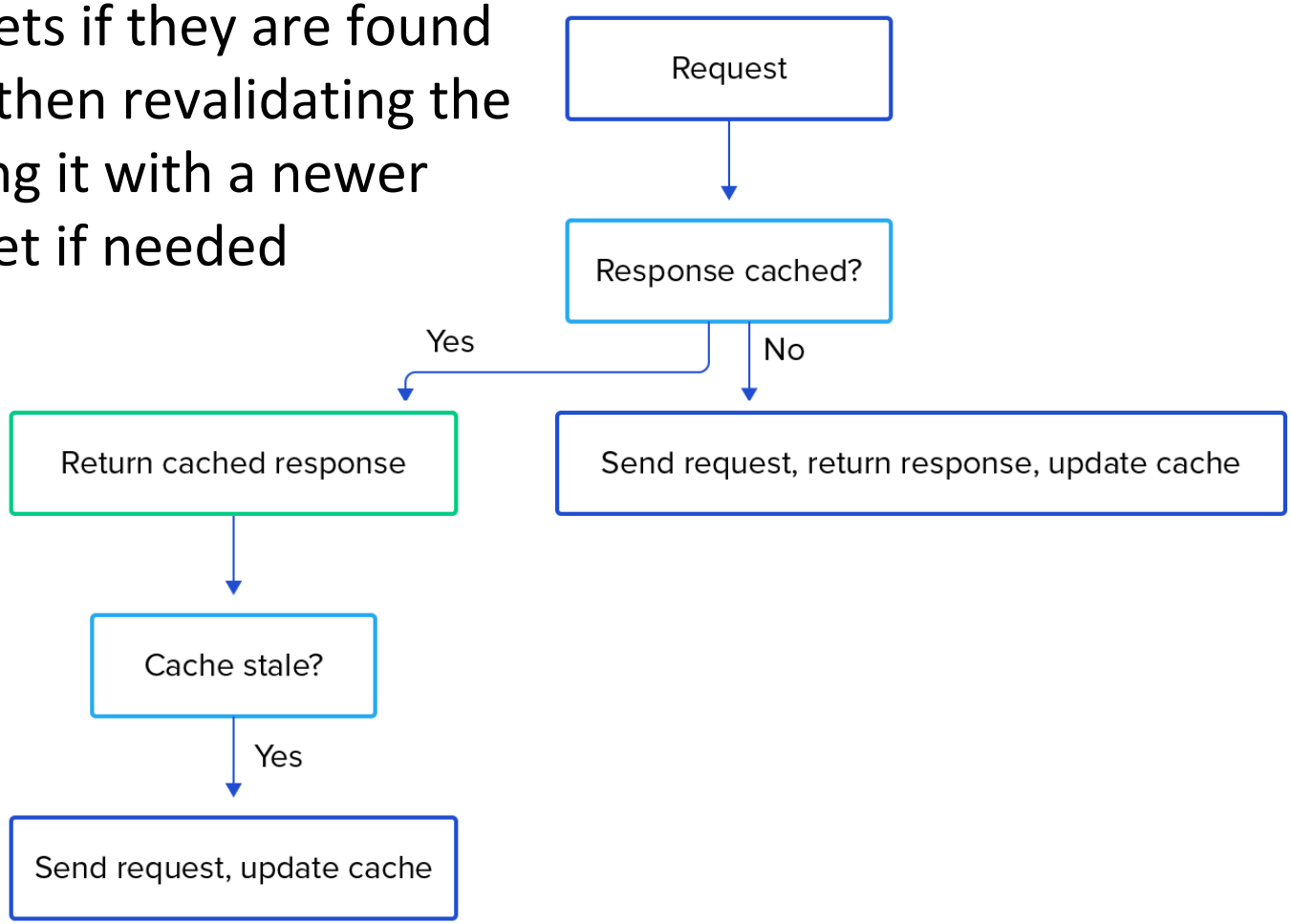
- The **useQuery** hook is used to manage data fetching. The parameters to pass to it are a **query key** and the **query function** (e.g., fetchToDos)
  - The query key is used internally for refetching, caching, and sharing the queries throughout the app
  - useQuery hook returns an object with a lot of properties including `data` , `isLoading` , `isSuccess` and `isError` properties for handling of loading and error states
  - e.g., ToDoList component (see next slide) displays an appropriate message based on the current API status of the fetchToDos. When the request is successful, we loop through and display the ToDos

useQuery Example

```
function Todos() {
  const { isLoading, isError, data, error } = useQuery(['todos'], fetchTodoList)

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

  if (isError) {
    return <span>Error: {error.message}</span>
  }

  // We can assume by this point that `isSuccess === true`
  return (
    <ul>
      {data.map(todo => (
        <li key={todo.id}>{todo.title}</li>
      ))}
    </ul>
  )
}
```

stale-while-revalidate

- React Query embraces the stale-while-revalidate caching strategy
- stale-while-revalidate involves using cached (stale) assets if they are found in the cache, and then revalidating the cache and updating it with a newer version of the asset if needed



React Query Caching

- You can just use the same useQuery hook in multiple components, and it will only fetch data once and then subsequently return it from the cache
- React query default values for config parameters try to keep the cached server data as **fresh** as possible while at the same time showing data to the user as early as possible, making it feel near instant and thus providing a great User Experience (UX)
  - E.g., **refetchOnWindowFocus** (default true): if the user goes to a different browser tab, and then comes back to your app, a background refetch will be triggered automatically, and displayed data will be updated if something has changed on the server

staleTime and cacheTime

- **staleTime**: The duration until a query transitions from fresh to stale
  - As long as the query is fresh, data will always be read from the cache only - no network request will happen!
  - If the query is stale, you will still get data from the cache, but a background refetch is triggered
  - Default is 0 i.e., immediately, a longer staleTime can be configured both globally and per-query so as not to refetch data as often
- **cacheTime**: The duration until inactive queries will be removed from the cache
  - Defaults to 5 minutes
  - Queries transition to the inactive state when all components which use that query have unmounted

Stale Queries

- Stale queries are refetched automatically in the background when:
  - New instances of the query mount
  - The window is refocused (**`refetchOnWindowFocus`** default true)
  - The network is reconnected (**`refetchOnReconnect`** default true)
  - The query is optionally configured with a refetch interval (**`refetchInterval`**)
  - Calling queryClient.**invalidateQuery**('queryKey')

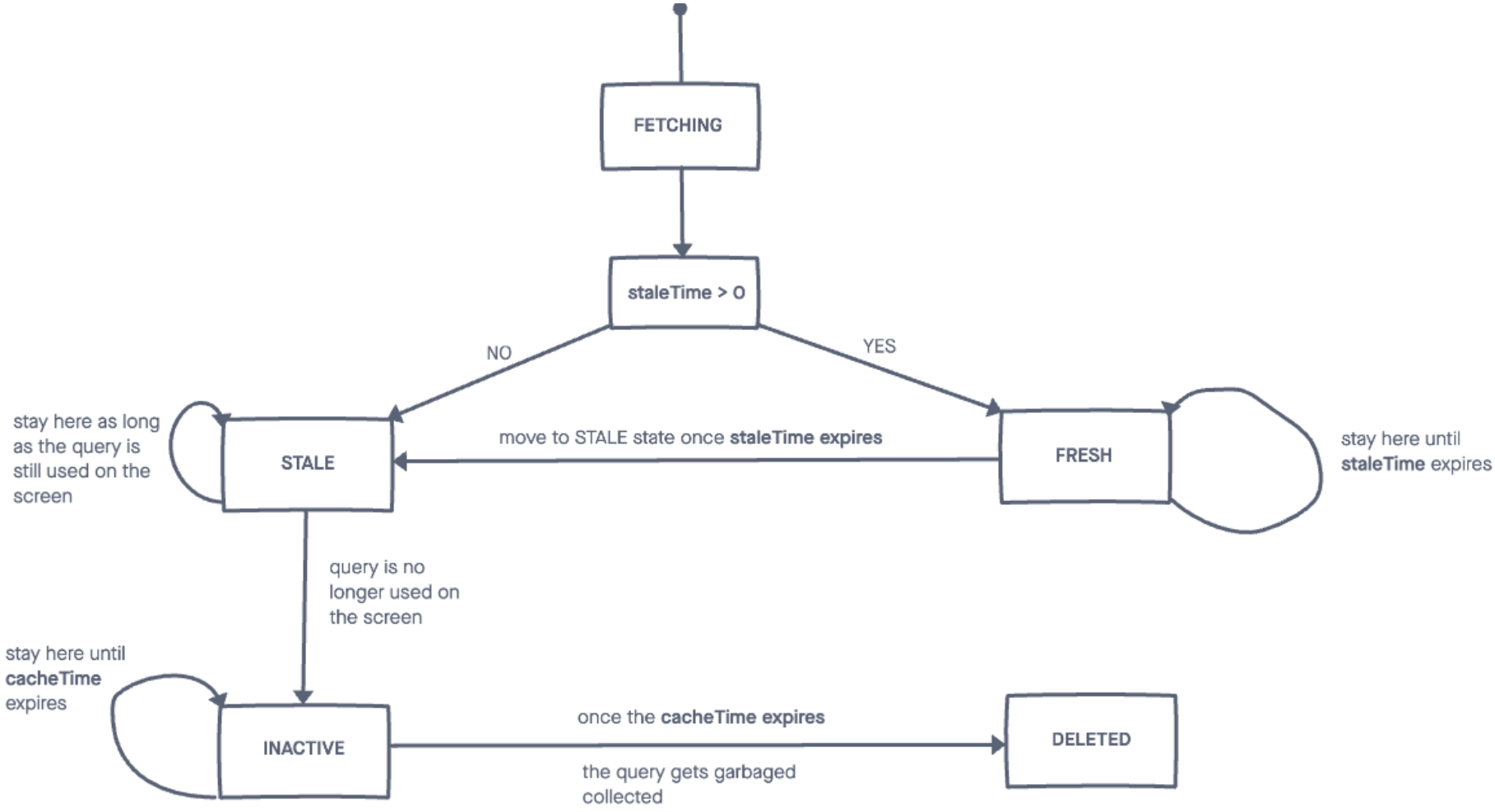
staleTime Example

- Let's assume we are using the default cacheTime of 5 minutes and the default staleTime of 0 or immediately
- A new instance of `useQuery(['todos'], fetchTodos)` mounts
  - A network request is made to fetch the data
  - Returned data will be cached under the ['todos'] key
  - The data is marked as stale (because staleTime defaults to 0)
- A second instance of **useQuery(['todos'], fetchTodos)** mounts elsewhere
  - Since the cache already has data for the ['todos'] key from the first query, that data is immediately returned from the cache
  - Trigger a new network request to fetch the data in the background
  - When the request completes successfully, the cache's data under the ['todos'] key is updated with the new data, and both instances are updated with the new data

cacheTime Example

- Both instances of the **`useQuery(['todos'], fetchTodos)`** query are unmounted and no longer in use
  - The cached data under the ['todos'] key is deleted and garbage collected after cacheTime (defaults to 5 minutes)
- Before the cache timeout has elapsed, another instance of `useQuery(['todos'], fetchTodos)` mounts
  - The query immediately returns the available cached data while the fetchTodos function is being run in the background
  - When it completes successfully, it will populate the cache with fresh data

staleTime and cacheTime Summary



retry and retryDelay options

- Queries that fail are auto-retried 3 times, with exponential backoff delay before capturing and displaying an error to the UI
- To change this, you can alter the default **`retry`** and **`retryDelay`** options for queries to something other than 3 and the default exponential backoff function
  - retryDelay is set to double (starting at 1000ms) with each attempt, but not exceed 30 seconds

```
import { useQuery } from '@tanstack/react-query'

// Make a specific query retry a certain number of times
const result = useQuery(['todos', 1], fetchTodoListPage, {
  retry: 5, // Will retry failed requests 5 times before displaying an error
  retryDelay: attemptIndex => Math.min(1000 * 2 * attemptIndex, 30000),
})
```

Query Client

- In the App.js file we need to create and provide an instance of the **`QueryClient`**
- The queryClient is used by React-Query to manage all the queries and mutations

```
import TodoList from "./TodoList"
import { QueryClient, QueryClientProvider } from 'react-query'

export default function App() {
  const queryClient = new QueryClient()
  return (
    <QueryClientProvider client={queryClient}>
      <TodoList />
    </QueryClientProvider>
  )
}
```

Configuration for all Queries

- E.g., retryDelay is set to double (starting at 1000ms) with each attempt, but not exceed 30 seconds

```
// Configure for all queries
import { QueryCache, QueryClient, QueryClientProvider } from '@tanstack/react-query'

const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      retryDelay: attemptIndex => Math.min(1000 * 2 * attemptIndex, 30000),
    },
  },
})

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

Mutations

- Unlike queries, mutations are typically used to create/update/delete data or perform server side-effects. For this purpose, React Query provude a **`useMutation`** hook
- By default, React Query will not retry a mutation on error, but it is possible with the retry option:
  - retry failing requests 3 times before showing the final error thrown by the function

```
const mutation = useMutation(addTodo, {
  retry: 3,
})
```

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

  return (
    <div>
      {mutation.isLoading ? (
        'Adding todo...'
      ) : (
        <>
          {mutation.isError ? (
            <div>An error occurred: {mutation.error.message}</div>
          ) : null}

          {mutation.isSuccess ? <div>Todo added!</div> : null}

          <button
            onClick={() => {
              mutation.mutate({ id: new Date(), title: 'Do Laundry' })
            }}
          >
            Create Todo
          </button>
        </>
      )}
    </div>
  )
}
```

useMutation Example

Query Invalidation

- QueryClient has an `invalidateQueries` method that lets you mark queries as stale so they can be re-fetched in the background

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

- You can even invalidate queries with specific variables by passing a more specific query key to the `invalidateQueries` method:

```
queryClient.invalidateQueries(['todos', { type: 'done' }])

// The query below will be invalidated
const todoListQuery = useQuery(['todos', { type: 'done' }], fetchTodoList)

// However, the following query below will NOT be invalidated
const todoListQuery = useQuery(['todos'], fetchTodoList)
```

Invalidation from Mutations

- Usually when a mutation succeeds, it's VERY likely that there are related queries in the app that need to be invalidated and refetched to reflect the new changes from your mutation
- When a successful postTodo mutation happens, we want all todos queries to get invalidated and refetched to show the new todo item. To do this, you can use useMutation's onSuccess options and the client's invalidateQueries function:

```
import { useMutation, useQueryClient } from '@tanstack/react-query'

const queryClient = useQueryClient()

// When this mutation succeeds, invalidate any queries with the `todos` or `reminders`
const mutation = useMutation(addTodo, {
  onSuccess: () => {
    queryClient.invalidateQueries(['todos'])
    queryClient.invalidateQueries(['reminders'])
  },
})
```

Optimistic Updates

- Updating the cached todos when adding a new todo

```
const queryClient = useQueryClient()

useMutation(updateTodo, {
  // When mutate is called:
  onMutate: async newTodo => {
    // Cancel any outgoing refetches (so they don't overwrite our optimistic update)
    await queryClient.cancelQueries(['todos'])

    // Snapshot the previous value
    const previousTodos = queryClient.getQueryData(['todos'])

    // Optimistically update to the new value
    queryClient.setQueryData(['todos'], old => [...old, newTodo])

    // Return a context object with the snapshotted value
    return { previousTodos }
  },
  // If the mutation fails, use the context returned from onMutate to roll back
  onError: (err, newTodo, context) => {
    queryClient.setQueryData(['todos'], context.previousTodos)
  },
  // Always refetch after error or success:
  onSettled: () => {
    queryClient.invalidateQueries(['todos'])
  },
})
```

React Query DevTools

- DevTools can be used for debugging
  - Shows data currently in the cache and their state
  - Shows the states of queries and data re-fetch requests


```
import { QueryClient, QueryClientProvider } from "react-query";
import { ReactQueryDevtools } from "react-query/devtools";
import { Posts } from "./Posts";

const queryClient = new QueryClient();

function App() {
  return (
    // provide React Query client to App
    <QueryClientProvider client={queryClient}>
      <Posts />
      <ReactQueryDevtools />
    </QueryClientProvider>
  )
}
```

Resources

- React Query

https://tanstack.com/query/v4/docs/overview

https://tanstack.com/query/v4/docs/videos

- Zustand

https://github.com/pmndrs/zustand