CMPS 356

Manage State

Dr. Abdelkarim Erradi
CSE@QU

Outline

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

Client State

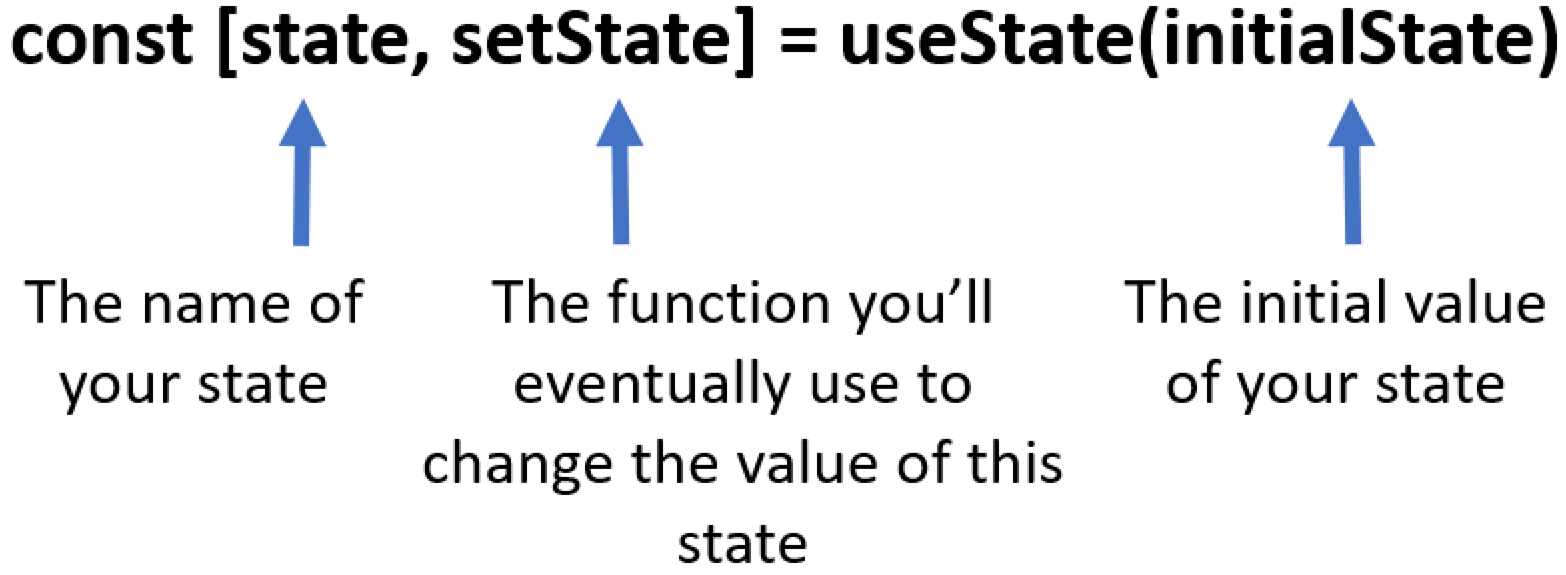
< Back

State

- 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
  - **useContext:** share data with child components without prop drilling

useState: creates a state variable

- Used for basic state management inside a component

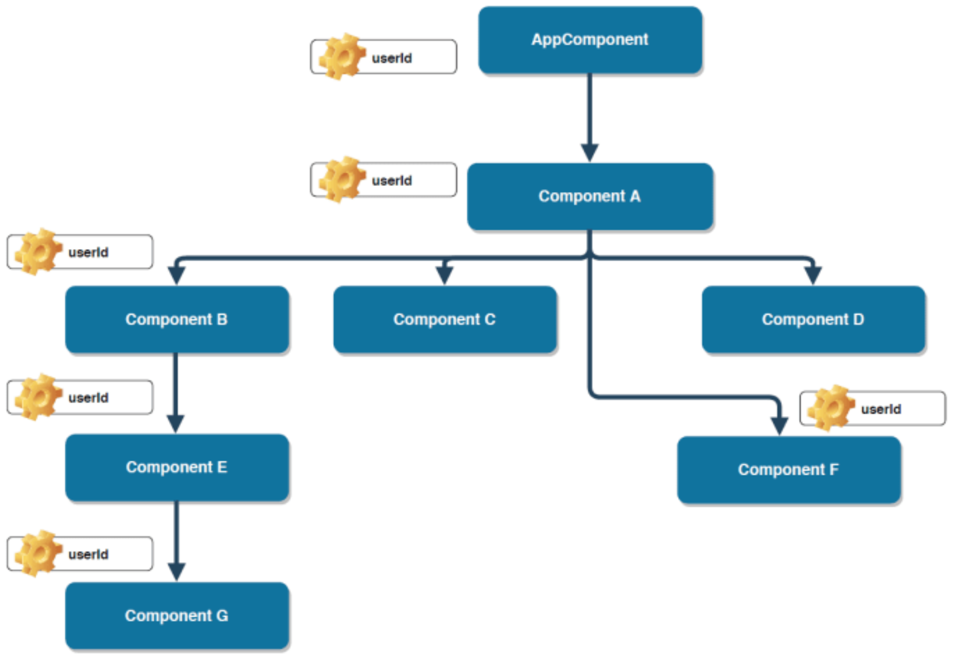


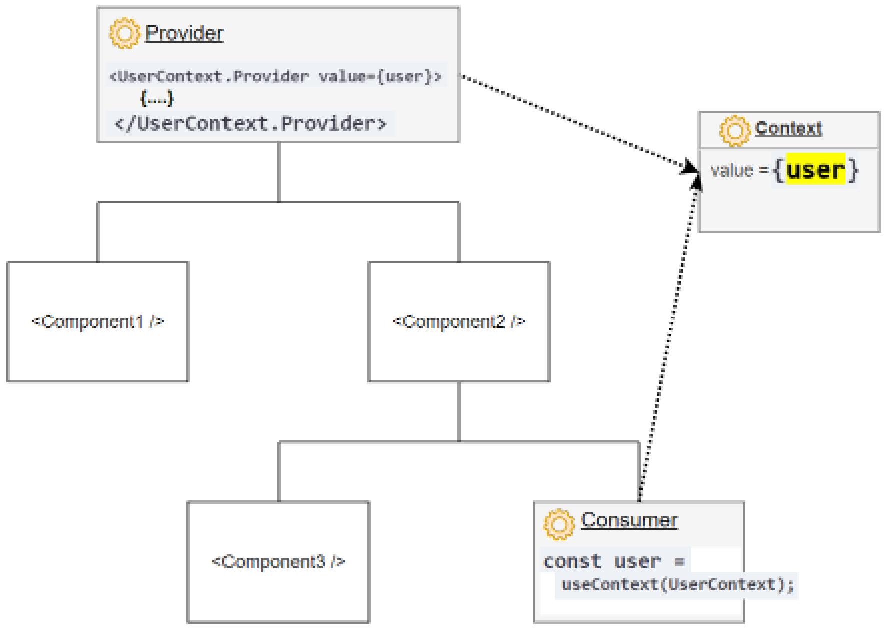
```
const [state, setState] = useState(initialState)
```

The name of your state
The function you'll eventually use to change the value of this state
The initial value of your state

useContext

- Share state between deeply nested components more easily "prop drilling" (i.e., pass the state as "props" through each nested component)
- Using the context requires 3 steps: creating, providing, and consuming the context

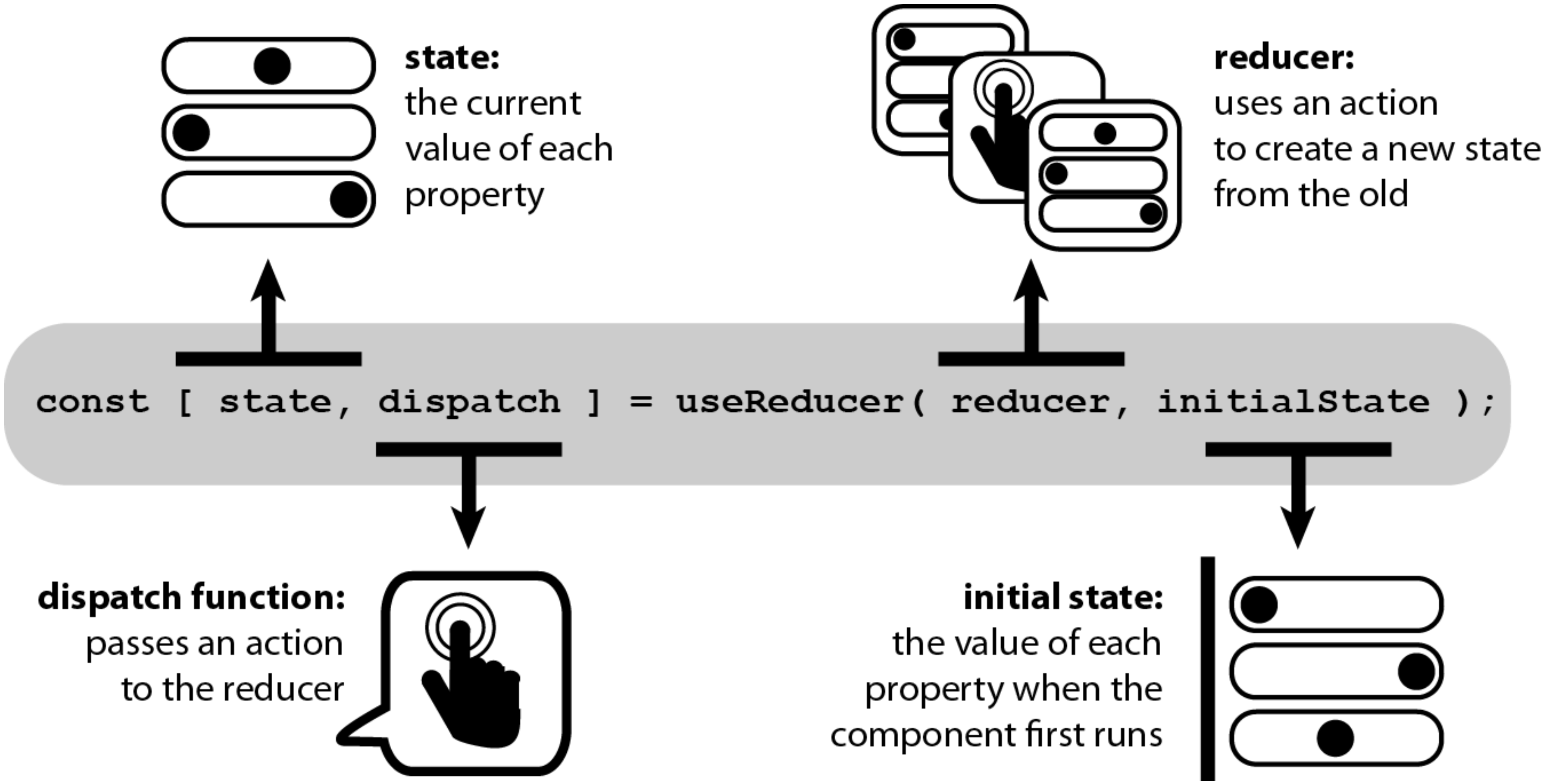




```
<UserContext.Provider value={user}>
   {....}
</UserContext.Provider>
```

```
const user =
  useContext(UserContext);
```

useReducer: manage multiple related state variables



```
const [ state, dispatch ] = useReducer( reducer, initialState );
```

state:
the current
value of each
property

reducer:
uses an action
to create a new state
from the old

dispatch function:
passes an action
to the reducer

initial state:
the value of each
property when the
component first runs

useContext – Define global variables and functions

1. **Create a context** (i.e., a global container to provide global variables and functions available to all components)

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

2. **Provider places global variables / functions in the context**

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

3. **Consumer access the global 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>;
}
```

Back

Zustand

- Zustand is a small and fast library that simplifies state-management and requires little of boilerplate to create shareable global store accessible everywhere in the app
  - 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

Server Cache State using React Query

< Back

Server Cache State

- Server cache state has some unique characteristics, such as re-fetching and managing cache revalidation
- React-Query is a feature-rich library that can be used for fetching, updating data, caching, background re-fetching, and more.

useQuery Hook

- The **useQuery** hook is used to manage data fetching. The parameters we are passing to it are a **query key** and the **query function** (e.g., fetchToDos)
  - The query key will be associated with the data that is returned by the query function.
  - useQuery hook returns an object with a lot of properties including data , isLoading , isSuccess and isError properties
- The ToDoList displays an appropriate message based on the current API status of the quotes requests. When the request is successful, we loop through and display the quotes.

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>
  )
}
```

What is React Query

Data-fetching library for React.

Makes fetching, caching, synchronizing and updating server state in a React application a breeze.

Server state challenges:

- Is persisted remotely in a location you do not control or own
- Requires asynchronous APIs for fetching and updating
- Implies shared ownership and can be changed by other people without your knowledge
- Can potentially become "out of date" in your applications if you're not careful

Features

Caching

Deduping multiple requests

Knowing when data is "out of date"

Updating stale data in the background

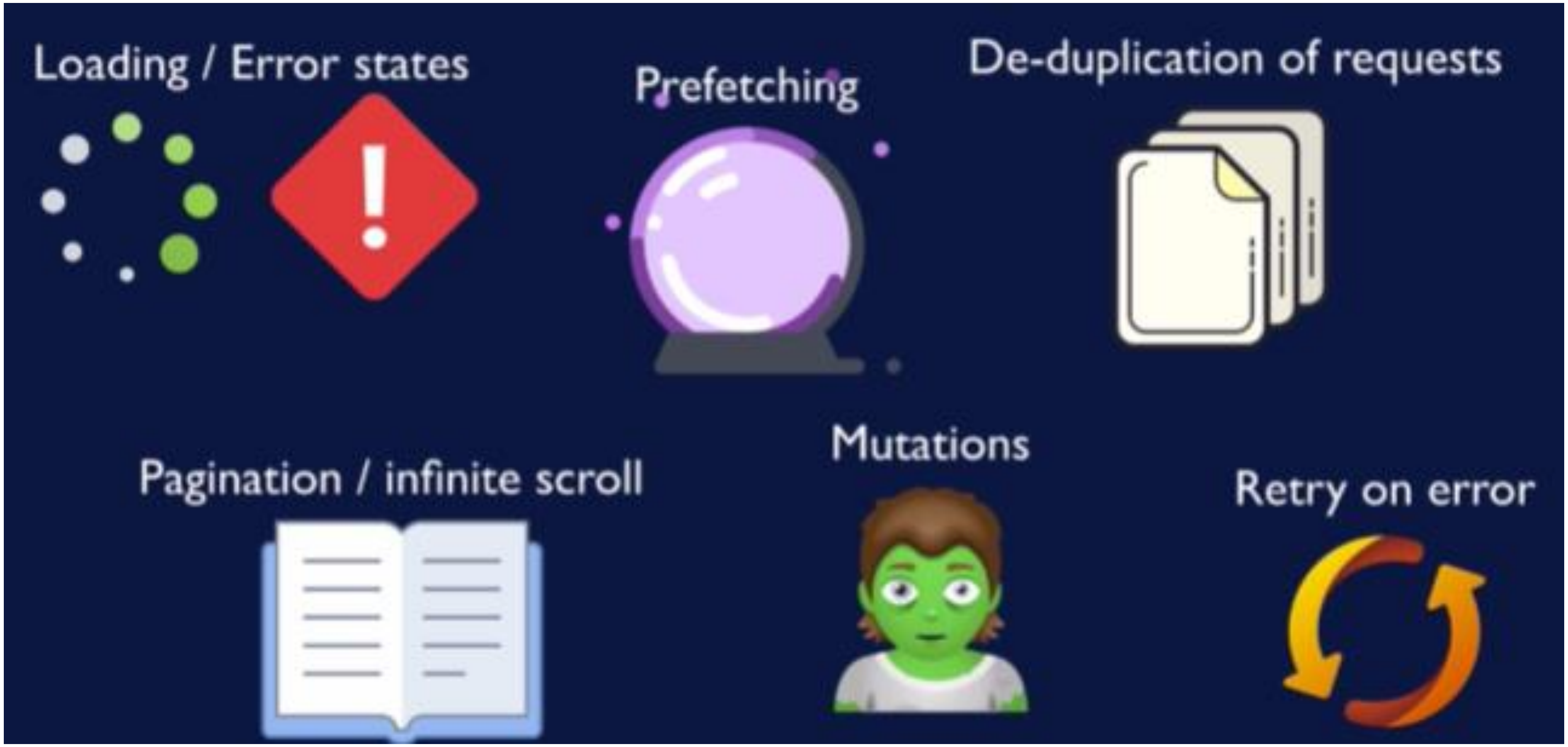
Reflecting UI updates faster

+ Performance optimizations (lazy, pages...)

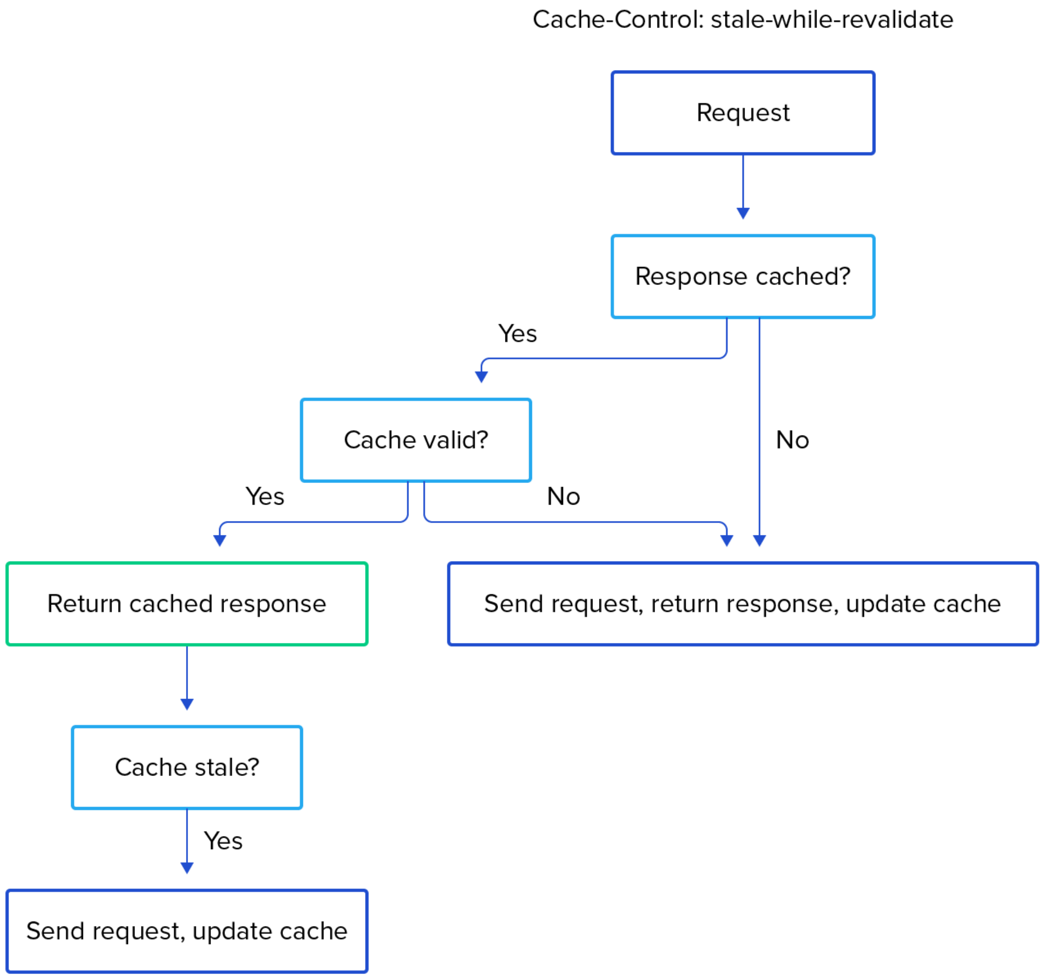
Server state memory and garbage collector

Memoizing query results

Features


Loading / Error states
Prefetching
De-duplication of requests
Pagination / infinite scroll
Mutations
Retry on error

stale-while-revalidate


Cache-Control: stale-while-revalidate
Request
Response cached?
Yes
No
Cache valid?
Yes
No
Return cached response
Send request, return response, update cache
Cache stale?
Yes
Send request, update cache

- 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