upGrad

#RahoAmbitious

Full Stack Software Development

upGrad

Course: Advanced Frontend Development Using React

Lecture On: React Hooks - Part II and Custom Hooks

In the last class, we discussed

- React Hooks for handling state management and side effects
- How to implement lifecycle methods of a component using React Hooks
- How React Hooks help segregate logically separate functionalities and help reduce the complexity of large components
- useState, useEffect and useContext hooks

Poll 1

Which of the following built-in React Hooks are used for lifecycle methods in components?

A. useState

B. useCallback

C. useEffect

D. useReducer

Poll 1 (Answer)

Which of the following built-in React Hooks are used for lifecycle methods in components?

A. useState

B. useCallback

C. useEffect

D. useReducer

Poll 2

Which of the following built-in Hooks is used to access the React context object?

A. useContext

B. useContextProvider

C. createContext

D. None of the above

Poll 2 (Answer)

Which of the following built-in Hooks is used to access the React context object?

A. useContext

B. useContextProvider

C. createContext

D. None of the above

Today's Agenda

1. useMemo and useCallback are built-in hooks in React that help with performance optimisation through memoization
2. The useReducer hook is used to manage more complex state transitions and state variables which are correlated as compared to useState
3. Creating our own hooks lets us extract component logic into reusable functions that can then be added into multiple components

More Hooks

More Hooks

upGrad

- useState and useEffect are basic hooks in React
- There are some additional hooks that are either variants of these basic hooks, or only needed for specific edge cases, e.g., optimisation

useMemo

upGrad

- Used for memoization, an optimisation technique that works by storing the results of expensive computations and returning the cached result when the same inputs are provided again
- Improves performance by not doing complex calculations on every render unless values change

useMemo

upGrad

Search and Filter Example

```
function ContactList(props) {
  const [text, setText] = React.useState('');
  const [searchTerm, setSearchTerm] = React.useState('');
  const getFilteredContacts = (filterKey) => {
    if (!filterKey) return props.items;
    return props.contacts.filter(contact => contact.name.toLowerCase().indexOf(filterKey.toLowerCase())
>= 0);
  }
  return (
  <div>
       <input type="text" value={searchTerm} onChange={e => setText(e.target.value)} />
       <button onClick={e => setSearchTerm(searchTerm)}>Search</button>
       <div class="results">
         {getFilteredContacts(searchTerm).map(contact =>
           (<div className="contact">
             <div className="name">{contact.name}</div>
             <div className="phone">{contact.phoneNumber}</div>
           </div>)
         )}
       </div>
     </div>
  );
}
```

useMemo

upGrad

Search and Filter Example

- Re-render happens in two cases:
  - User types something in the input box
  - User clicks the ‘Search’ button
- The list of contacts is filtered in each case, although it only needs to be filtered in the second case
- Can you prevent the unnecessary filtering?

useMemo

upGrad

Search and Filter with useMemo for Optimisation

```
const filteredContacts = React.useMemo(() =>
  props.contacts.filter(contact =>
    contact.name.toLowerCase().indexOf(searchTerm.toLowerCase())),
  [searchTerm]
);
```

This ensures that filteredContacts is not recomputed unless the value of search term changes

useMemo

upGrad

Input and Output

Input
useMemo expects a function and an array of dependencies

Output
useMemo returns the memoized value computed by the function passed to it, i.e., it will recompute the value only if one or more of the dependencies have changed, otherwise it will just return the cached result, leading to optimised performance

If no array is provided, you do not get any performance optimisation

useCallback

upGrad

- Used to memoize a callback function
- Useful for passing callbacks to child components to prevent unnecessary re-renders

Search and Filter Example With a Click Handler

Refactor the previous example a bit, abstracting out a separate child component for rendering individual contacts:

```
function Contact(props) {
  return (
    <div className="contact" onClick={props.onClick}>
      <div className="name">{props.name}</div>
      <div className="phone">{props.phoneNumber}</div>
    </div>
  );
}
```

upGrad

Search and Filter Example With a Click Handler

The updated component is shown below

```
function ContactList(props) {
  const [text, setText] = React.useState('');
  const [searchTerm, setSearchTerm] = React.useState('');
  const getFilteredContacts = (filterKey) { ... }
  const onContactClick = () => { ... }
  return (
    <div>
      <input type="text" value={searchTerm} onChange={e => setText(e.target.value)} />
      <button onClick={e => setSearchTerm(searchTerm)}>Search</button>
      <div class="results">
        {getFilteredContacts(searchTerm).map((contact, index) =>
          <Contact key={index} {...contact} onClick={onContactClick} />
        )}
      </div>
    </div>
  );
}
```

Search and Filter Example With a Click Handler

Now, the problem with this code is that for each keystroke in the input search box, the component re-renders, and the onContactClick function is redefined each time during re-render, which is unnecessary

You can do this instead to preserve the same function reference across re-renders:

```
const onContactClick = React.useCallback(() => { ... }, []);
```

Note: useCallback must be used judiciously, i.e., only when the number of callbacks is very high; for simple use cases it is not worth the additional code complexity and overhead that useCallback brings

useCallback

upGrad

Input and Output

Input

- A callback function
- An array of dependencies

Output

A memoized version of the callback that only changes if any of the dependencies have changed

Note:

useCallback(fn, dependencies) is equivalent to *useMemo(() => fn, dependencies)*

Optimisations

upGrad

Phone Directory

You can make use of the optimisations provided by useMemo and useCallback in the phone directory app within the PhoneDirectory component:

```
// This function will not be redefined on every render unless subscribersList changes
const deleteSubscriberHandler = useCallback(async (subscriberId) => {
  // ...
},[subscribersList]);

// The subscriber count will not be recomputed on every render unless subscribersList
changes
const numberOfSubscriptions = useMemo(()=>{
  return subscribersList.length;
},[subscribersList])
```

Code Reference

Poll 3

Which of the following React hooks is used to prevent repeated expensive computations?

A. useState
B. useEffect
C. useMemo
D. useLayoutEffect

Poll 3 (Answer)

Which of the following React hooks is used to prevent repeated expensive computations?

A. useState

B. useEffect

C. useMemo

D. useLayoutEffect

Poll 4

What is the purpose of the useCallback hook?

A. Prevent unnecessary re-renders of the parent component

B. Prevent unnecessary expensive computations

C. Prevent callback functions to be unnecessarily redefined on each render

D. State management

Poll 4 (Answer)

What is the purpose of the useCallback hook?

A. Prevent unnecessary re-renders of the parent component

B. Prevent unnecessary expensive computations

C. Prevent callback functions to be unnecessarily redefined on each render

D. State management

Poll 5

State whether the following statement is true or false

The useMemo() hook returns a memoized function, whereas the useCallback() hook returns a memoized value

A. True

B. False

Poll 5 (Answer)

State whether the following statement is true or false

The useMemo() hook returns a memoized function, whereas the useCallback() hook returns a memoized value

A. True

B. False

Poll 6

Recall the implementation of the useCallback hook and select the correct option based on the code snippet given below

```
const deleteSubscriberHandler = useCallback(async (subscriberId)=>{
    const rawResponse = await
fetch("http://localhost:7081/api/contacts/"+subscriberId,{method:"D
ELETE"})
    const data = await rawResponse.json();
    loadData();
},[])
```

A. deleteSubscriberHandler is compiled whenever the subscriberId is changed

B. It will only be compiled once, as the dependency array is empty

Poll 6 (Answer)

Recall the implementation of the useCallback hook and select the correct option based on the code snippet given below

```
const deleteSubscriberHandler = useCallback(async (subscriberId)=>{
    const rawResponse = await
fetch("http://localhost:7081/api/contacts/"+subscriberId,{method:"D
ELETE"})
    const data = await rawResponse.json();
    loadData();
},[])
```

A. deleteSubscriberHandler is compiled whenever the subscriberId is changed

B. It will only be compiled once, as the dependency array is empty

Simple vs Complex State Management

Simple State Management

Managing state in primitives, e.g., a number or string

Directly updating the state variable

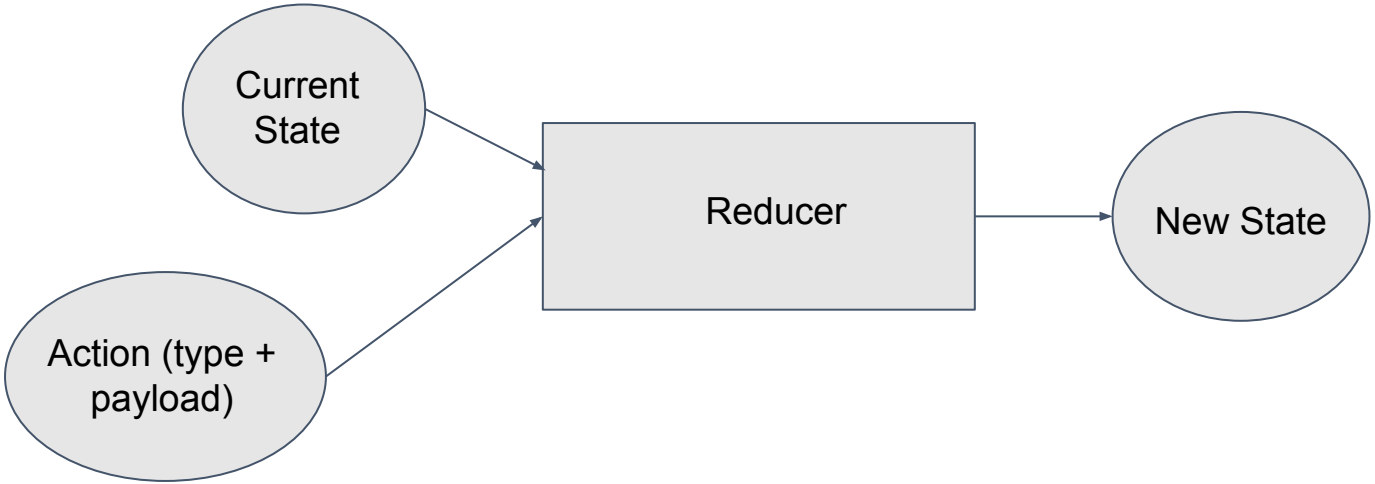
Complex State Management

When the state grows into a complex object and individual or deeply nested properties within the object needs to be updated, or when the next state depends on the previous state

State is not directly mutated, but rather an action is dispatched, and a reducer function updates the state based on this action

What is a Reducer?

A reducer is a pure function (without side effects) that determines changes to an application's state



useReducer Input and Output

```
const [state, dispatch] = useReducer(reducerFunction, initState, init)
```

Input

- A reducer function of the form (state, action) => newState
- The initial state object (optional, used to provide a default state when the application first loads)
- A function to generate the initial state lazily (optional, but if provided the initial state object will be passed as an argument to this function and the output used as the initial state, i.e., init(initState)

useReducer Input and Output

Output

useReducer returns a pair of the following:

- A reference to the state object
- A dispatch function used to trigger actions: this function takes in a single argument--the action

Action

Typically an action consists of two things: the type (identifier) and an optional payload, e.g.,

```
{
  type: "SET_DATA",
  payload: { ... }
}
```

useReducer

upGrad

Putting it All Together

```
function MyCounter() {
  function reducer(state, action) {
    switch (action.type) {
      case 'INCREMENT':
        return { count: state.count + 1 };
      case 'DECREMENT':
        return { count: state.count - 1 };
      default:
        return state;
    }
  }
  const [state, dispatch] = React.useReducer(reducer, { count: 0 });
  return (
    <div>
      The count is: {state.count}
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>+</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>-</button>
    </div>
  );
}
```

Comparison with useState

Use useState if you have any of the following aspects:

- JavaScript primitives as state
- simple view logic within your component
- different properties that do not change in any correlated way and can be managed by multiple useState hooks

Use useReducer if you have any of the following aspects:

- JavaScript objects or arrays as state
- Complex state transitions
- Complicated business logic
- Different properties tied together that should be managed in one state object and the need to update state deep down in your component tree

Phone Directory

In the Phone Directory app, you can maintain the count of subscribers in a separate state object and define a reducer (*TotalSubscribersReducer*) to update it

```
const [state,dispatch] = useReducer(TotalSubscribersReducer, {count: 0});
```

Dispatch an action to update the subscriber count within the loadData function:

```
dispatch({"type":"UPDATE_COUNT", payload: data.length});
```

Code Reference

Poll 7

Which of the following statements are true for reducers?
(**Note:** More than one option may be correct.)

A. A reducer maintains the internal app state within a closure
B. A reducer is a pure function
C. The input to a reducer is an array of actions
D. The input to a reducer is the previous state and an action

Poll 7 (Answer)

Which of the following statements are true for reducers?
(**Note:** More than one option may be correct.)

A. A reducer maintains the internal app state within a closure

B. A reducer is a pure function

C. The input to a reducer is an array of actions

D. The input to a reducer is the previous state and an action

Poll 8

Which of the following statement is true for the output from a reducer?

A. A reducer function does not return anything
B. A reducer function returns a new updated state object
C. A reducer function returns a reference to the previous state object which is updated as per the action
D. A reducer function always returns the initial state object

Poll 8 (Answer)

Which of the following statement is true for the output from a reducer?

A. A reducer function does not return anything

B. A reducer function returns a new updated state object

C. A reducer function returns a reference to the previous state object which is updated as per the action

D. A reducer function always returns the initial state object

Implementing Custom Hooks

Custom Hooks

upGrad

Another interesting aspect of React is that it allows you to build your own custom hooks based on your needs

You can always use functions to perform a task, but what if you want to dispatch an event that changes the state of the component that calls the function?

You can accomplish this only by using hooks. A custom hook does not need to have a specific signature. You can decide what it takes as arguments and what it should return

The Need For Custom Hooks

Creating your own hooks allows you to extract component logic into reusable functions

Consider an app with multiple components that need to fetch and render some data from an API. Until the data is fetched, a loader is displayed on the screen. Instead of maintaining this logic individually (and repeatedly) within the components, you can abstract the functionality out as a custom hook and use it with any component in our app

Custom Hooks

upGrad

Data Polling Example

```
// Let's define the reducer for this first
const initialState = {
  data: null,
  isLoading: true,
  error: null
};

function reducer(state, action) {
  switch (action.type) {
    case 'SET_DATA': {
      return Object.assign(state, { data: action.data, isLoading: false, error:
null });
    case 'ERROR': {
      return Object.assign(state, { data: null, isLoading: false, error:
action.error });
    default:
      return state;
  }
}
```

Custom Hooks

upGrad

Data Polling Example

```
function MyComponent(props) {
  const [state, dispatch] = React.useReducer(reducer, initialState);
  const { isLoading, data, error } = state;

  useEffect(() => {
    fetch(props.url).then(data => {
      dispatch({ type: 'SET_DATA', data });
    })
    .catch(e => {
      dispatch({ type: 'ERROR', error });
    });
  });

  if (isLoading) return <div>Fetching data. Please wait...</div>;
  if (error) return <div>An error occurred while fetching data</div>;
  return <div>{/* Render the data */}</div>;
}
```

Now, let's abstract this out as a custom hook.

Custom Hooks

upGrad

Data Polling Example With a Custom Hook

```
// our custom hook...
function useDataFromApi(url) {
  const [state, dispatch] = React.useReducer(reducer, initialState);
  const { isLoading, data, error } = state;

  useEffect(() => {
    fetch(props.url).then(data => {
      dispatch({ type: 'SET_DATA', data });
    })
    .catch(e => {
      dispatch({ type: 'ERROR', error });
    });
  });
  return [isLoading, data];
}
function MyComponent(props) {
  const [isLoading, data] = useDataFromApi(props.url);
  if (isLoading) return <div>Fetching data. Please wait...</div>;
  return <div>{/* Render the data */}</div>;
}
```

Custom Hooks

upGrad

- Custom hooks follow all the rules applicable for other built-in hooks, e.g., a custom hook should also be called at the top-level scope within a component and never within conditions or loops
- It is highly recommended to name a custom hook with the **use** prefix
- Two components using the same custom hook do not share the state; they just reuse the stateful logic within the hook
- Each invocation of a custom hook gets its own isolated state
- It is possible to share information among custom hooks through the parameters that they accept

Poll 9

Which of the following is the purpose of a custom hook?

A. Lifecycle management of a React component
B. Extracting out common component logic for reuse
C. Optimising re-renders
D. Fetching data from APIs

Poll 9 (Answer)

Which of the following is the purpose of a custom hook?

A. Lifecycle management of a React component

B. Extracting out common component logic for reuse

C. Optimising re-renders

D. Fetching data from APIs

Hands-on Exercise

upGrad

Create a React app with the following pages and a navigation bar containing links to each of the pages:

- Home
- About
- Contact Us

When each page loads (mounts), it should log the name of the page and the timestamp on the console. Extract this functionality as a custom hook and use it in each of the pages

The stub code is provided here
The solution code is provided here

All the code used in today's session can be found in the link provided below:

https://github.com/upgrad-edu/react-hooks/tree/Session11

Doubt Clearance (5 minutes)

Important Questions

Important Questions

upGrad

1. What is the React memo function?
2. How do you memoize a component?
3. How to optimize React app performance?
4. What is the difference between useCallback and useMemo in practice?
5. What is the difference between React context and React Redux?

Key Takeaways

- useMemo and useCallback are built-in hooks in React that help with performance optimisation through memoization
- The useReducer hook is used to manage more complex state transitions and state variables which are correlated as compared to useState
- Creating our own hooks lets us extract component logic into reusable functions that can then be added to multiple components

The following tasks are to be completed after today's session:

MCQs
Coding Questions
Course Project (Part B) - Checkpoint 2

In the next class, we will discuss

- Implementing forms in React
  - Difference between React forms and traditional HTML forms
  - Controlled vs uncontrolled components
- Form validation using Material UI
  - Higher order validator components
  - Validation rules

upGrad

#RahoAmbitious

Thank You!