React Hooks

Error Boundaries

React.Lazy

Wrapper Components and testing using Shallow/Render

react Hooks

Hooks are functions that let you “hook into” React state and lifecycle features from function components. Hooks don’t work inside classes — they let you use React without classes.

React provides a few built-in Hooks like useState. You can also create your own Hooks to reuse stateful behaviour between different components.

cited from

https://reactjs.org/docs/hooks-overview.htm

Our Require is to identify component that can be used for writing middle ware for react app

below is an example of a react app using hooks .The app accepts a name and a title and adds it as a todo list

The thing to understand is this is done in much lesser code and by using predefined hooks

import logo from './logo.svg';

import './App.css';

import {useState} from 'react';

function App() {

  const [todolist,setTodolist]=useState([{title:"",text:""}])

  const [text,setText]=useState('')

  const[title,setTitle]=useState('')

  return (

    <div className="App">

     <div>     <input type='text' onChange={(e)=>{

         setTitle(e.target.value)

     }} placeholder="Enter task title"/>

     </div>

     <input type='text' onChange={(e)=>{

       setText(e.target.value)

     }} placeholder="Enter task details"/>

    <div>

     <button onClick={()=>{

       setTodolist(todolist=>[...todolist,{"title":title,"text":text}])

     }}>

       Add task

     </button>

     </div>

     <center>

     {

     todolist.map((e)=>{

return <div style={{backgroundColor:'pink',width:200}}>

   <h3><b>{e.title}</b></h3>

 <h2> {e.text}</h2>

  </div>

     })

    }

    </center>

    </div>

  );

}

export default App;

Here todolist is captured using function setTodolist ,now there is no separate body defination for setTodolist and it is used straight in operation

We can also build hooks that execute as react life cycle ,like component did mount .In below example useEffect is used to create a hook that will execute as soon as UI is rendered

const[pageHeader,setPageHeader]=useState('')

  useEffect(()=>{

setPageHeader("To Do List")

  },[])

We can create our own custom hooks to call on events ,below is an example of a custom hook that can be called on a button click

function useSetDev()

  {

    const [devname,setDevname]=useState('')

    const setTaskDev=()=>setDevname('ASP')

    return [devname,setTaskDev]

  }

  const [devname ,setTaskDev]=useSetDev()

<button onClick={()=>{

 setTaskDev()

      }}>

        Add Assigner name

      </button>

{

devname

    }

Below is complete example of todo App that adds task list dynamically .

What we need to notice is that ui has been released from task of writing logic and this has been handed over to hooks

import logo from './logo.svg';

import './App.css';

import {useEffect, useState} from 'react';

function App() {

  const [todolist,setTodolist]=useState([{title:"",text:""}])

  const [text,setText]=useState('')

  const[title,setTitle]=useState('')

  const[pageHeader,setPageHeader]=useState('')

  useEffect(()=>{

setPageHeader("To Do List")

  },[])

  function useSetDev()

  {

    const [devname,setDevname]=useState('')

    const setTaskDev=()=>setDevname('ASP')

    return [devname,setTaskDev]

  }

  const [devname ,setTaskDev]=useSetDev()

  return (

    <div className="App">

      <center>

      <div style={{width:200,backgroundColor:'greenyellow'}}>{pageHeader}</div>

      </center>

     <div>     <input type='text' onChange={(e)=>{

         setTitle(e.target.value)

     }} placeholder="Enter task title"/>

     </div>

     <input type='text' onChange={(e)=>{

       setText(e.target.value)

     }} placeholder="Enter task details"/>

    <div>

     <button onClick={()=>{

       setTodolist(todolist=>[...todolist,{"title":title,"text":text}])

    }}>

       Add task

     </button>

     <button onClick={()=>{

 setTaskDev()

      }}>

        Add Assigner name

      </button>

     </div>

     <center>

     {

     todolist.map((e)=>{

return <div style={{backgroundColor:'pink',width:200}}>

   <h3><b>{e.title}</b></h3>

 <h2> {e.text}</h2>

  </div>

     })

    }

    </center>

    {

devname

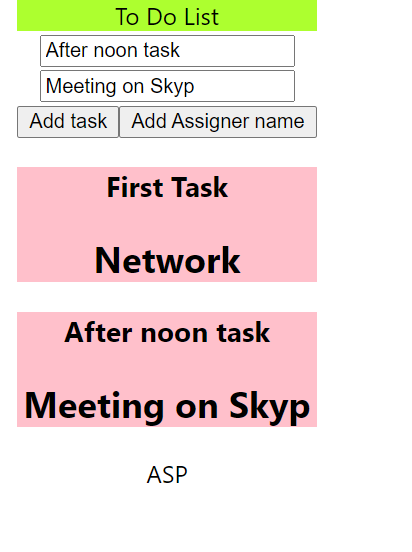
    }

    </div>

  );

}

export default App;



Error Boundaries

A JavaScript error in a part of the UI shouldn’t break the whole app. To solve this problem for React users, React 16 introduces a new concept of an “error boundary”.

Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

Error boundaries do **not** catch errors for:

* Event handlers ([learn more](https://reactjs.org/docs/error-boundaries.html#how-about-event-handlers))
* Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
* Server side rendering
* Errors thrown in the error boundary itself (rather than its children)

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods [static getDerivedStateFromError()](https://reactjs.org/docs/react-component.html#static-getderivedstatefromerror) or [componentDidCatch()](https://reactjs.org/docs/react-component.html" \l "componentdidcatch). Use static getDerivedStateFromError() to render a fallback UI after an error has been thrown. Use componentDidCatch() to log error information.

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

componentDidCatch(error, errorInfo) {

// You can also log the error to an error reporting service

logErrorToMyService(error, errorInfo);

}

render() {

if (this.state.hasError) {

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

<ErrorBoundary>

<MyWidget />

</ErrorBoundar

Error boundaries work like a JavaScript catch {} block, but for components. Only class components can be error boundaries. In practice, most of the time you’ll want to declare an error boundary component once and use it throughout your application.

Bubdling and Code-Splitting

Most React apps will have their files “bundled” using tools like [Webpack](https://webpack.js.org/), [Rollup](https://rollupjs.org/) or [Browserify](http://browserify.org/" \t "_blank). Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

Bundling is great, but as your app grows, your bundle will grow too. Especially if you are including large third-party libraries. You need to keep an eye on the code you are including in your bundle so that you don’t accidentally make it so large that your app takes a long time to load.

To avoid winding up with a large bundle, it’s good to get ahead of the problem and start “splitting” your bundle. Code-Splitting is a feature supported by bundlers like [Webpack](https://webpack.js.org/guides/code-splitting/), [Rollup](https://rollupjs.org/guide/en/#code-splitting) and Browserify (via [factor-bundle](https://github.com/browserify/factor-bundle)) which can create multiple bundles that can be dynamically loaded at runtime.

Code-splitting your app can help you “lazy-load” just the things that are currently needed by the user, which can dramatically improve the performance of your app. While you haven’t reduced the overall amount of code in your app, you’ve avoided loading code that the user may never need, and reduced the amount of code needed during the initial load.

The best way to introduce code-splitting into your app is through the dynamic import() syntax.

**Before:**

import { add } from './math';

console.log(add(16, 26));

**After:**

import("./math").then(math => {

console.log(math.add(16, 26));

});

When Webpack comes across this syntax, it automatically starts code-splitting your app. If you’re using Create React App, this is already configured for you and you can [start using it](https://create-react-app.dev/docs/code-splitting/) immediately. It’s also supported out of the box in [Next.js](https://nextjs.org/docs/advanced-features/dynamic-import).

If you’re setting up Webpack yourself, you’ll probably want to read Webpack’s [guide on code splitting](https://webpack.js.org/guides/code-splitting/). Your Webpack config should look vaguely [like this](https://gist.github.com/gaearon/ca6e803f5c604d37468b0091d9959269).

When using [Babel](https://babeljs.io/), you’ll need to make sure that Babel can parse the dynamic import syntax but is not transforming it. For that you will need [@babel/plugin-syntax-dynamic-import](https://classic.yarnpkg.com/en/package/@babel/plugin-syntax-dynamic-import).

## React.lazy

The React.lazy function lets you render a dynamic import as a regular component.

**Before:**

import OtherComponent from './OtherComponent';

**After:**

const OtherComponent = React.lazy(() => import('./OtherComponent'));

This will automatically load the bundle containing the OtherComponent when this component is first rendered.

React.lazy takes a function that must call a dynamic import(). This must return a Promise which resolves to a module with a default export containing a React component.

The lazy component should then be rendered inside a Suspense component, which allows us to show some fallback content (such as a loading indicator) while we’re waiting for the lazy component to load.

import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Loading...</div>}>

<OtherComponent />

</Suspense>

</div>

);

}

The fallback prop accepts any React elements that you want to render while waiting for the component to load. You can place the Suspense component anywhere above the lazy component. You can even wrap multiple lazy components with a single Suspense component.

import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Loading...</div>}>

<section>

<OtherComponent />

<AnotherComponent />

</section>

</Suspense>

</div>

);

}

Shallow renderer

When writing unit tests for React, shallow rendering can be helpful. Shallow rendering lets you render a component “one level deep” and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

function MyComponent() {

return (

<div>

<span className="heading">Title</span>

<Subcomponent foo="bar" />

</div>

);

}

import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:

const renderer = new ShallowRenderer();

renderer.render(<MyComponent />);

const result = renderer.getRenderOutput();

expect(result.type).toBe('div');

expect(result.props.children).toEqual([

<span className="heading">Title</span>,

<Subcomponent foo="bar" />

]);