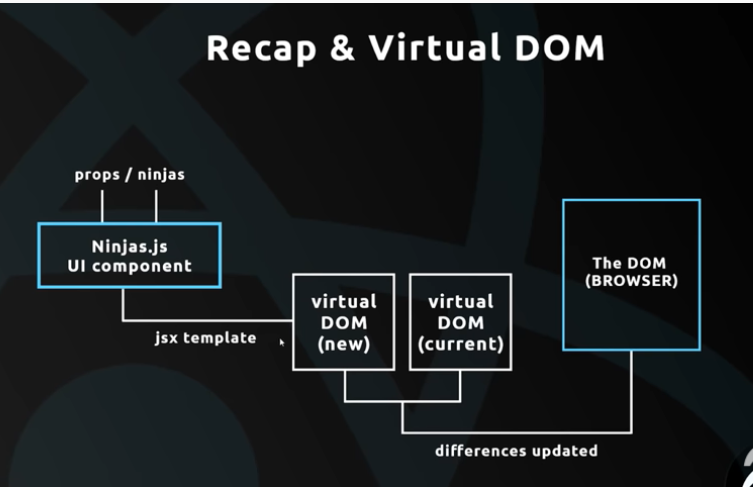
Dev Diaries

React Series

**Introduction**

What makes react special is its performance speed as it takes the code and stores it as a virtual DOM using JS and with each change it compares the current virtual DOM with the previous virtual DOM and only renders the additional content unlike others which renders the whole page again.



React can be using either by making a complete React app or by using CDN like a common JS package. The advantage of the latter is that it is ideal for beginners as it’s hazzle free and just like using any other kind of JS library.

I used Visual Studio Code for this but I needed extensions like ES7 React/Redux/GraphQL/React-Native code snippets for faster dev and Babel JavaScript for better syntax highlighting.

**Creating a React Component**

Made an index.html file and in the head tags added the src paths of the React and ReactDOM CDN. The React has the templates while the DOM renders it into JSX which is like HTML but most browsers can’t understand it which is why we need Babel. Went to the Babel site and got the src link under the prototype section. This is useful for testing prototypes without having to setup Babel for each project. Stuck it under the head tag.

To make a React component:

<script type="text/babel">

        class App extends React.Component {

            render(){

                return(

                    <div className="app-content">

                        <h1>Hey!</h1>

                        <p>{Math.random()\*10}</p>

                    </div>

                )

            }

        }

        ReactDOM.render(<App />,document.getElementById('app'));

        </script>

Every react component can have only one parent tag but can have nested tag within the parent tag which is why we use the div inside the return. Every class must have at least one function and here we use render. Unlike CSS JSX’s divs can’t have class tag as class is a keyword in JS so we use className.

The React.DOM.render() is very important as no matter what you do with react it can be rendered only if the DOM is made to render it. Here the App class is represented as <App /> and it is rendered in the app element in the page.

The use of React in this snippet of code is that unlike the others when we refresh this page only the number is refreshed every-time.

**State**

States are JS objects that can be used to describe the current state of the component which can be updated over time. A modal is another way of putting it. It can be used to store data that could change dynamically.

class App extends React.Component {

            state={

                name: 'HP',

                age: 19

            }

            render(){

                return(

                    <div className="app-content">

                        <h1>Hey!</h1>

                        <p>{Math.random()\*10}</p>

                        <p>My name is {this.state.name} & I'm {this.state.age} years old. </p>

                    </div>

                )

            }

        }

**React Devtools**

Added the react devtools to chrome and accessed it by clicking the components tab in Developer tools. We can use this for testing purposes as it allows us to play around with states etc.

**React DOM events**

Various events can be triggered by certain actions like

buttonHasBeenClicked(e){

                    console.log('Clicked!');

            }

            buttonHovered(e){

                console.log('Hovered');

            }

            whenCopied(e){

                console.log('I\'m Watching');

            }

            render(){

                return(

                    <div className="app-content">

                        <h1>Hey!</h1>

                        <p>{Math.random()\*10}</p>

                        <p>My name is {this.state.name} & I'm {this.state.age} years old. </p>

                        <button onClick={this.buttonHasBeenClicked}>Click Me</button>

                        <button onMouseOver={this.buttonHovered}>Hover Over</button>

                        <p onCopy={this.whenCopied}>When I think of you</p>

                    </div>

                )

            }

Inspite of all this if we try to change the state in the functions using this keyword it wont work as then the object changes therefore this no longer points to state.

stateChanger(e){

                console.log(this.state);

            }

This gives an error. So, we need to refer to them using arrow functions.

When we use an arrow function, it wraps the value of this into the new this.

stateChanger=(e)=>{

                console.log(this.state);

            }

We can also change the state values using this technique:

stateChanger=(e)=>{

                console.log(this.state);

                this.setState({

                    name:'Hrishi',

                    age: 20

                });

            }

**Forms**

We use the form tag just like in HTML. Here we use the form onSubmit event rather than the button onClick event so as to ensure form submission irrespective of whether the user clicks Submit or presses Enter. The e.preventDefault() prevents the page from being reloaded again otherwise once we submit the name goes back to the default name and the text field becomes empty.

handleChange=(e)=>{

                this.setState({

                    name: e.target.value

                })

            }

            handleSubmit=(e)=>{

                e.preventDefault();

                console.log('form submitted',this.state.name)

            }

            render(){

                return(

                    <div className="app-content">

                        <h1>Hey!</h1>

                        <p>{Math.random()\*10}</p>

                        <p>My name is {this.state.name} & I'm {this.state.age} years old. </p>

                        <button onClick={this.buttonHasBeenClicked}>Click Me</button>

                        <button onMouseOver={this.buttonHovered}>Hover Over</button>

                        <p onCopy={this.whenCopied}>When I think of you</p>

                        <button onClick={this.stateChanger}>Change state</button>

                        <form onSubmit={this.handleSubmit}>

                            <input type="text" onChange={this.handleChange}/>

                            <button>Submit</button>

                        </form>

                    </div>

                )

            }

        }

**Create a react app**

We now use npx to create a default react app:

npx create-react-app myapp

cd myapp

npm start

We do this so that all the react files get stored in our local directory so that instead of a test build we can make a deployment build. A react app also makes the code modular and very reusable.

**SPAs (Single Page Applications)**

React apps are typically SPA’s which gives dynamic content dependent on user interaction. Typically in multipage apps the index.html gives a request to server and shows the response and this is so with other pages but in react like applications the initial page is requested from the server and shows the response most other requests don’t go to the server but are handled using components thereby making the app much faster and responsive.

Everything we are concerned with would be in the src. If we go to the public folder we see the index.html page and all the react components are served through the <div> folder inside the index.html.

We just need to work with the App.js file and the export < App /> at the end means that class is exported out to be rendered using the DOM. The DOM rendering bit happens in the index.js file which imports the app, renders it and injects it into the index.html’s dive component.

To make a development build:

npm start

To make a production build:

npm run build

Once we create the app, we must delete the unnecessary code and files like the provision for the logo etc.

**Nesting Components**

A react component has a root component in most cases it’s App.js and other components like Navbar.js, Contact.js etc. can be included by nesting them in the root component. The App.js is rendered first and additional components are rendered according to user behaviour.

Example make a file in src called Ninjas.js

import React , {Component} from 'react'

class Ninjas extends Component{

    render(){

        return(

            <div className="ninja">

                <div>Name: HP</div>

                <div>Age: 19</div>

            </div>

        )

    }

}

export default Ninjas

Here we make the Ninjas class a component by importing React and Component. Then we use the render etc. to make the component as needed and finally we export the component to be rendered elsewhere. This makes this component both modular and reusable.

If we want to use it in App.js just import the component (we can import it only if it has been exported) and to use the component just write it in tags within the code just like we do with HTML.

import React from 'react';

import Ninjas from './Ninjas';

function App() {

  return (

    <div className="App">

      <h1>My First React app !</h1>

      <Ninjas />

    </div>

  );

}

export default App;

**Props**

It’s short for proxies and is used to give values to components before they are rendered. Props helps to avoid hardcoding and makes components much more usable. Sometimes there might be cases where the same structure is to be repeated only the values change so we use props to make just one component and give the values during render time.

Eg:

import React , {Component} from 'react'

class Ninjas extends Component{

    render(){

        return(

            <div className="ninja">

                <div>Name: {this.props.name}</div>

                <div>Age: {this.props.age}</div>

            </div>

        )

    }

}

export default Ninjas

this is how we access the prop value within the component.

import React from 'react';

import Ninjas from './Ninjas';

function App() {

  return (

    <div className="App">

      <h1>My First React app !</h1>

      <Ninjas name="Hrishi" age="19"/>

    </div>

  );

}

export default App;

this is how we pass values to props.

Another way of using props easily in a component is by making use of de-structuring so that we don’t have to use this.props every time we access a prop.

import React , {Component} from 'react'

class Ninjas extends Component{

    render(){

        const {name,age} = this.props;

        return(

            <div className="ninja">

                <div>Name: {name}</div>

                <div>Age: {age}</div>

            </div>

        )

    }

}

export default Ninjas

The use of props is illustrated here:

<Ninjas name="Hrishi" age="19"/>

      <Ninjas name="HP" age="20"/>

The same component can be used to give different results.

**Lists**

The above code can be a bit wearisome and not that useful. So we use lists and make the component loop through the list and render instances of it. For that we create a list using a state object. But for that we have to make App a class instead of a function:

import React, { Component } from 'react';

import Ninjas from './Ninjas';

class App extends Component {

  state={

    ninjas : [

      {name:"Hrishi", age:20, id:1},

      {name:"HP", age:19, id:2}

    ]

  }

  render(){

    return (

      <div className="App">

        <h1>My First React app !</h1>

        <Ninjas ninjas= {this.state.ninjas}/>

      </div>

    );

    }

}

export default App;

Then we pass the array as the prop

 <Ninjas ninjas= {this.state.ninjas}/>

Then inside the component we de-structure it and store the props value to a variable. We have to cycle through the array and output it to the DOM and this can be done using JS’s map function which maps each element in a new array. Then we make it loop through this array and return each component instance with the respective value which is stored in the new array. Then we render the new array.

import React , {Component} from 'react'

class Ninjas extends Component{

    render(){

        const {ninjas} = this.props;

        const ninjaList = ninjas.map(ninja=>{

        return(

            <div className="ninja">

                <div>Name: {ninja.name}</div>

                <div>Age: {ninja.age}</div>

            </div>

        )

        })

    return(

        <div className="ninja-list">

            {ninjaList}

        </div>

    )

    }

}

export default Ninjas

but React needs each component to have its unique id for optimum performance so we add a key tag to the div and to make it unique we set the value as the array element’s id value.

**Stateless components**

There are 2 types of components:

1. Container components-
   1. Contain state
   2. Contain lifecycle hooks
   3. Not concerned with UI
   4. Use classes to create
2. UI components-
   1. Don’t contain state
   2. Receive data from props
   3. Only concerned with UI
   4. Use functions to create

Eg: App.js, ContactForm.js are Container components whereas Navbar.js is a UI component

The Ninja.js is a container component right now as it makes use of class and has a state to make it a UI component just change the class to a function but then the prop won’t be called directly. To call props automatically in functions they must be specified in the parameters of the function. We no longer need the render function as that is for classes and we also don’t use this.props we only use props as we no longer have objects to use this.

import React , {Component} from 'react'

const Ninjas =(props) =>{

        const {ninjas} = props;

        const ninjaList = ninjas.map(ninja=>{

        return(

            <div className="ninja" id={ninja.id}>

                <div>Name: {ninja.name}</div>

                <div>Age: {ninja.age}</div>

            </div>

        )

        })

    return(

        <div className="ninja-list">

            {ninjaList}

        </div>

    )

}

export default Ninjas

instead of calling props first and then assigning them to a variable we can directly use de-structuring inside the function parameters.

import React , {Component} from 'react'

const Ninjas =({ninjas}) =>{

        const ninjaList = ninjas.map(ninja=>{

        return(

            <div className="ninja" id={ninja.id}>

                <div>Name: {ninja.name}</div>

                <div>Age: {ninja.age}</div>

            </div>

        )

        })

    return(

        <div className="ninja-list">

            {ninjaList}

        </div>

    )

}

export default Ninjas

if there are more than 1 props then just add commas and add them to the rest. Instead of using ninjaList we could put the whole code inside the outer return but that would make it a bit more difficult to understand.

**Conditional output**

This can be done using if-else or ternary operator.

Using if-else:

import React , {Component} from 'react'

const Ninjas =({ninjas}) =>{

        const ninjaList = ninjas.map(ninja=>{

        if(ninja.age>19){

            return(

                <div className="ninja" id={ninja.id}>

                    <div>Name: {ninja.name}</div>

                    <div>Age: {ninja.age}</div>

                </div>

            );

            }

        else{

            return null;

        }

    })

    return(

        <div className="ninja-list">

            {ninjaList}

        </div>

    )

}

export default Ninjas

using ternary operator:

import React , {Component} from 'react'

const Ninjas =({ninjas}) =>{

        const ninjaList = ninjas.map(ninja=>{

        return (ninja.age>19)?

                (

                <div className="ninja" id={ninja.id}>

                    <div>Name: {ninja.name}</div>

                    <div>Age: {ninja.age}</div>

                </div>

                ): null;

    })

    return(

        <div className="ninja-list">

            {ninjaList}

        </div>

    )

}

export default Ninjas

**More on forms**

Made a new file AddNinja.js which is a container component as forms require us to store state.

import React, { Component } from "react";

class AddNinja extends Component{

    state={

        name: null,

        age: null

    }

    handleChange=(e)=>{

        this.setState({

            [e.target.id]: e.target.value

        })

    }

    handleSubmit=(e)=>{

        e.preventDefault();

        console.log(this.state);

    }

    render(){

        return(

            <div>

                <form onSubmit={this.handleSubmit}>

                    <label htmlFor="name">Name: </label>

                    <input type="text" id="name" onChange={this.handleChange}/>

                    <label htmlFor="age">Age:</label>

                    <input typt="text" id="age" onChange={this.handleChange}/>

                    <button>Submit</button>

                </form>

            </div>

        )

    }

}

export default AddNinja

here a single function handleChange is used and the respective field value is changed using the target id which was given to each tag previously.

The e.preventDefault() prevents the default property of the page to refresh itself on Submitting which will only lead to loss of data and resetting to default value in this case.

**Functions as props**

We can make changes to state by using functions as props. For that in App.js I made a new function called addNinja which takes in the latest entry and assigns it an id. Although the best way to do this is to create unique ids here I have taken a random number to keep things simple.

To add the entry into the state is a bit tricky. If needed we can just push it to the array like:

this.ninjas.push(ninja);

or using the setState method we can push it into the array:

this.setState({

ninjas.push(ninja)

})

But both these ways result in changing or deforming of the current state. We want to try to insert elements without deforming the state although I am not sure if that is really necessary. I have used the spread operator ( … ) to separate the array into individual elements and added them in [] along with the latest entry this is to make a new array comprising of the current elements and the newly added element and then assigning this new array to the state.

addNinja=(ninja)=>{

    ninja.id=Math.random();

    let newninjas = [...this.state.ninjas, ninja];

    this.setState({

      ninjas: newninjas

    })

  }

To call this function when we submit the entry I changed the handleSubmit method to :

    handleSubmit=(e)=>{

        e.preventDefault();

        this.props.addNinja(this.state);

        console.log(this.state);

    }

**Delete data from state**

To delete the data from the state first I made a function called deleteNinja and took id as its parameter. To delete an entry from an array without deforming or changing it we can just make a new array by filtering out the unnecessary values and assigning the value of the new array to the old array.

  deleteNinja=(id)=>{

    let newninjas = this.state.ninjas.filter(ninja=>{

      return ninja.id !== id

    })

    this.setState({

      ninjas : newninjas

    })

  }

If return value is true the element is added to the array and if it’s false the element is not included in the new array.

To ensure that this function can be called from the Ninjas component, I pass it too as a prop:

render(){

    return (

      <div className="App">

        <h1>My First React app !</h1>

        <Ninjas ninjas= {this.state.ninjas} deleteNinja={this.deleteNinja}/>

        <AddNinja addNinja={this.addNinja}/>

      </div>

    );

    }

}

In the ninjas function I made a new button which on being clicked will call the deleteNinja function. But instead of calling it like:

onClick={deleteNinja(ninja.id)}

I have to use another method. If it was onClick={deleteNinja} it would work because in that case the deleteNinja won’t be automatically triggered but in the above case it gets automatically triggered. To prevent it we do:

return (ninja.age>19)?

                (

                <div className="ninja" id={ninja.id}>

                    <div>Name: {ninja.name}</div>

                    <div>Age: {ninja.age}</div>

                    <button onClick={()=>{deleteNinja(ninja.id)}}>Delete</button>

                </div>

                ): null;

This ensures that the function doesn’t get triggered automatically. Here we are using an anonymous function along with an arrow function. Now deleteNinja fires only when the anonymous function fires and that fires only on a button click.

**Adding CSS to React Applications**

While using react, if you import a CSS file in any one component, then the file will also affect all other components. Eg:

Changing background to pink and importing the file to ninjas.js will affect App.js and AddNinja.js too.

To get around this we use tags for the components and write tag specific CSS code so it will affect only the parts that have that tag id. Another way is to make CSS components but for that we will have to make changes in the config file and that won’t be covered here.

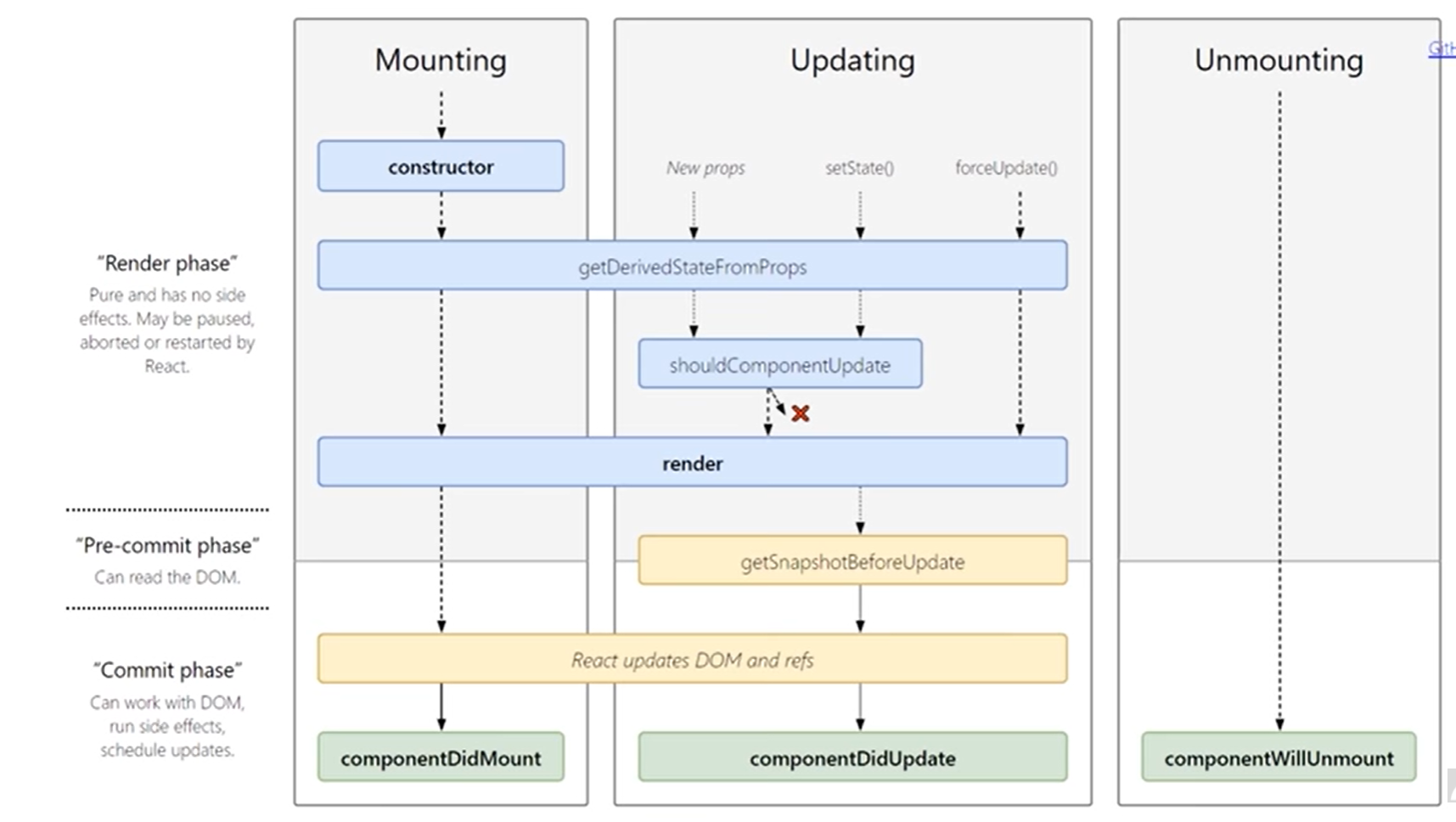
**Lifecycle methods**

During mounting the constructor (explicit definition not necessary) gives default values to the props. Then the derived state is obtained and then the render function is called. A snapshot of the DOM before update is taken and then the React updates the DOM and refs accordingly and finally, the componentDidMount function is called.

During updating the derived state is obtained from Props and if new props or setState is used the shouldComponentsUpdate is called if true then the render function is called else nothing is done but if the forceUpdate() function is called then render function is called no matter what. This then gets a snapshot of the DOM and then React updates the DOM and refs and the componentDidUpdate is called.

In unmount stage the componentWillUnmount is called.

In almost all programs we work with render. If we change the value of the props after render part of the program we might risk running into an infinite loop.



**Making a todo app**

First I made a new todo app using:

npx create-react-app todo app

then I deleted the unnecessary files like the logo svg files the logo css file etc. Then I removed the unnecessary imports from App.js and also changed it from function to class:

import React, {Component} from 'react';

class App extends Component{

  render(){

    return(

      <div className="App">

      </div>

    );

  }

}

export default App;

Then I went to materialize css website and copied the CDN link for the CSS only and pasted it in the index.html head tag.

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css"/>

After this I made a new state and made an array called todo with an id and content tag. To display it on the site a new file Todo.js was made:

App.js:

import React, {Component} from 'react';

import Todos from './todos';

class App extends Component{

  state={

    todos: [

      {

        id:1,

        content: "Play Wolfie"

      }

    ]

  }

  render(){

    return(

      <div className="App">

        <h1 className="center blue-text">Todo</h1>

        <Todos todos={this.state.todos} }/>

      </div>

    );

  }

}

export default App;

Todo.js:

import React from 'react';

const Todos = ({todos})=>{

    const todoList = todos.length ? (

        todos.map(todo=>{

            return(

                <div className="collection-item" key={todo.id}>

                    <span>{todo.content}</span>

                </div>

            )

        })

    ) : (

        <p className="center">You have no todos left </p>

    )

    return(

        <div className="todos collection">

            {todoList}

        </div>

    )

}

export default Todos

the todoList function checks if todos is empty. The todos.length returns true if not empty and false if empty. The ternary operator is used to either display the todo or to display the no todos left message.

Then made a function to delete the todos. This function is invoked whenever the todo is clicked.

App.js:

import React, {Component} from 'react';

import Todos from './todos';

class App extends Component{

  state={

    todos: [

      {

        id:1,

        content: "Play Wolfie"

      }

    ]

  }

  deleteTodo=(id)=>{

    const newtodos=this.state.todos.filter(todo=>{

      return todo.id !== id

    }

    )

    this.setState({

      todos: newtodos

    })

  }

  render(){

    return(

      <div className="App">

        <h1 className="center blue-text">Todo</h1>

        <Todos todos={this.state.todos} deleteTodo={this.deleteTodo}/>

      </div>

    );

  }

}

export default App;

Todo.js:

import React from 'react';

const Todos = ({todos,deleteTodo})=>{

    const todoList = todos.length ? (

        todos.map(todo=>{

            return(

                <div className="collection-item" key={todo.id}>

                    <span onClick={()=>{deleteTodo(todo.id)}}>{todo.content}</span>

                </div>

            )

        })

    ) : (

        <p className="center">You have no todos left </p>

    )

    return(

        <div className="todos collection">

            {todoList}

        </div>

    )

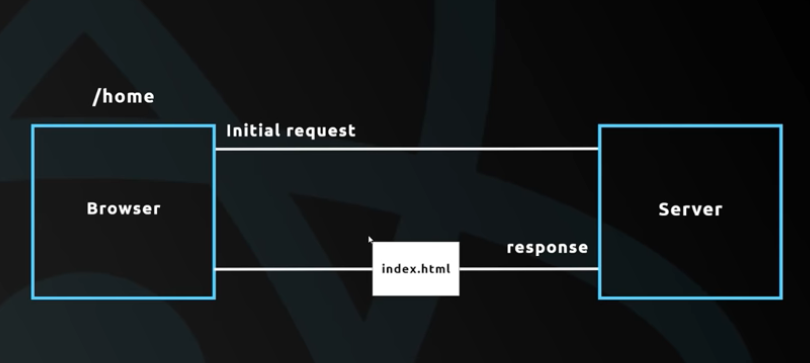
}

export default Todos

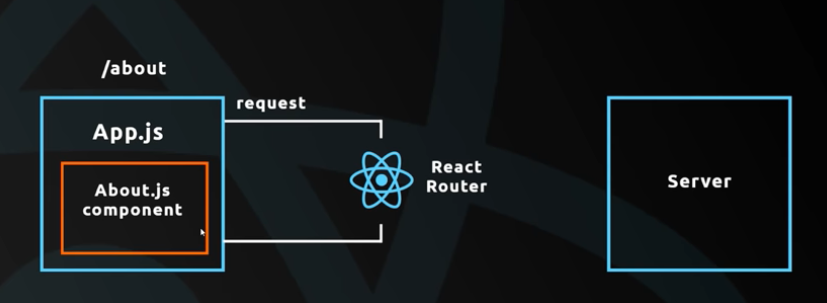
**React router**

To implement different pages in a single page react application we just update components.

Unlike std. paradigm where each time the request goes to the server:



React has a React router which intercepts and takes the request and injects the required components instead of contacting the server every time.



To make use of the react router to serve different components as if they were urls:

I made a new dir in src called components and made Home.js, About.js, Contact.js just a few components for a basic site.

Home.js:

import React from 'react'

const Home=()=>{

    return(

        <div className="container">

            <h4 className="center">Home</h4>

            <p>Lorem ipsom</p>

        </div>

    )

}

export default Home

About.js:

import React from 'react'

const About=()=>{

    return(

        <div className="container">

            <h4 className="center">About</h4>

            <p>Hey there this is about</p>

        </div>

    );

}

export default About

Contact.js:

import React from 'react'

const Contact=()=>{

    return(

        <div className="container">

            <h4 className="center">Contact</h4>

            <p>Contact us!</p>

        </div>

    )

}

export default Contact

made another file called Navbar.js:

import React from 'react'

const Navbar=()=>{

    return(

        <div className="nav-wrapper red darken-3">

            <div className="container">

                <a className="brand-logo">Poke Times</a>

                <ul className="right">

                    <li><a href="/">Home</a></li>

                    <li><a href="/about">About</a></li>

                    <li><a href="/contact">Contact</a></li>

                </ul>

            </div>

        </div>

    )

}

export default Navbar

it is as a hyperlink to the list that we give the urls. We don’t make separate html pages for these urls. All we do is to make the react router inject and eject code when the urls are accessed. This method makes it unnecessary to contact the server for each page.

To make use of the React router we must first install it in the project:

npm install react-router-dom

App.js:

import React, {Component} from 'react';

import Navbar from './components/Navbar'

import {BrowserRouter, Route} from 'react-router-dom'

import Home from './components/Home'

import About from './components/About'

import Contact from './components/Contact'

class App  extends Component{

  render(){

    return(

      <BrowserRouter>

      <div className="App">

        <Navbar />

        <Route exact path='/' component={Home}/>

        <Route path='/about' component={About}/>

        <Route path='/contact' component={Contact}/>

      </div>

      </BrowserRouter>

    );

  }

}

export default App;

next we have to import the BrowserRouter and Route. We use the <BrowserRoute> tag and only in this can we route between components. So whenever the / path is accessed the Home component is to be served which is why we have to import the Home component first. Whenever the /about is accessed the About component is served which is why we have to import the About component first and so on.

We use the exact path for ‘/’ because if we don’t then ‘/contact’ will be treated as a call for both Contact and Home because ‘/’ is a subset of ‘/contact’ and so on. If we use exact path the only if the url is exactly ‘/’ will Home be called.

Another way around this is to use /home instead of / but then when the url is / and (that is necessary for most service providers ) the page doesn’t show the Home component.

Although at this stage the components are served correctly we see that with each route the whole page is reloaded. This means that each route request goes to the actual server and not the react server. Although this works, this doesn’t achieve the full potential of react.

**Links and NavLinks**

We need to force the react server to take up route requests and to do this we use Links or NavLinks. If we inspect the HTML code which is rendered into the webpage we see that in place of Links and NavLinks <a> tags are used. We used <a> tags in the previous example but what makes this any different ?

When we use Links or NavLinks we are forcing it not to use the default property of <a> tags i.e. requesting to server instead the react server is used. Similar to the e.preventDefault() fn that prevents the default property of submit which would otherwise lead to page reload.

I changed all the <a> tags I used in Navbar.js to <Link> tags

Navbar.js:

import React from 'react'

import {Link, NavLink} from 'react-router-dom'

const Navbar=()=>{

    return(

        <div className="nav-wrapper red darken-3">

            <div className="container">

                <a className="brand-logo">Poke Times</a>

                <ul className="right">

                    <li><Link to="/">Home</Link></li>

                    <li><Link to="/about">About</Link></li>

                    <li><Link to="/contact">Contact</Link></li>

                </ul>

            </div>

        </div>

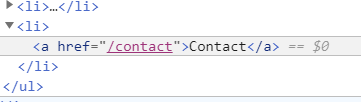
    )

}

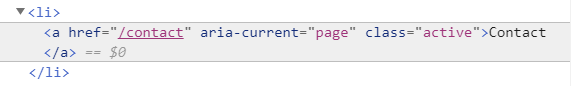
export default Navbar

Difference between Link and NavLink:

When I used Link this is what the particular part looked like when we inspect the page:



When I used NavLink this is what the part looked like when we inspect the page



So whenever we use NavLink, the aria-current and class=”active” is also set this tells us which tag is currently active i.e. which link are we currently in and can also be used for giving different styles to the current tab etc.

If you don’t want to do all that then it makes no difference whether you use Link or NavLink.

Programmatic redirects

If we take in props for say Contact.js and log it to the console:

import React from 'react'

import About from './About'

const Contact=(props)=>{

   console.log(props)

    return(

        <div className="container">

            <h4 className="center">Contact</h4>

            <p>Contact us!</p>

        </div>

    )

}

export default Contact

we get:



Notice the push function and the history where the path is given. We could do a programmatic redirect by pushing a new link to the history.

import React from 'react'

import About from './About'

const Contact=(props)=>{

    setTimeout(()=>{

        props.history.push("/about")

    },2000)

    return(

        <div className="container">

            <h4 className="center">Contact</h4>

            <p>Contact us!</p>

        </div>

    )

}

export default Contact

here the setTimeout() function pushes the /about url to the history which causes the site to redirect after 2 sec (I used the setTimeout() only to emphasize the transition which would otherwise be almost imperceptible)

But if I try the same thing with Navbar.js I get an error:

Navbar.js:

import React from 'react'

import {Link, NavLink} from 'react-router-dom'

const Navbar=(props)=>{

    setTimeout(()=>{

        props.history.push("/about")

    },2000)

    return(

        <div className="nav-wrapper red darken-3">

            <div className="container">

                <a className="brand-logo">Poke Times</a>

                <ul className="right">

                    <li><NavLink to="/">Home</NavLink></li>

                    <li><NavLink to="/about">About</NavLink></li>

                    <li><NavLink to="/contact">Contact</NavLink></li>

                </ul>

            </div>

        </div>

    )

}

export default Navbar



To clarify what went wrong let’s try logging the props to the console:



The props is empty i.e. it has no history property so the push is undefined. This is so because if we look at App.js:

import React, {Component} from 'react';

import Navbar from './components/Navbar'

import {BrowserRouter, Route} from 'react-router-dom'

import Home from './components/Home'

import About from './components/About'

import Contact from './components/Contact'

class App  extends Component{

  render(){

    return(

      <BrowserRouter>

      <div className="App">

        <Navbar />

        <Route exact path='/' component={Home}/>

        <Route path='/about' component={About}/>

        <Route path='/contact' component={Contact}/>

      </div>

      </BrowserRouter>

    );

  }

}

export default App;

the react router automatically adds properties to the routes inside a BrowserRouter. Contact is a Route so the properties get automatically added into the props but Navbar isn’t a route so the properties don’t get automatically added to the props for Navbar.

However, we can make the router add properties to the props of a non-route component using a higher order component.

A higher order component routes another components and gives it properties.

Modified Navbar.js:

import React from 'react'

import {Link, NavLink, withRouter} from 'react-router-dom'

const Navbar=(props)=>{

    console.log(props)

    setTimeout(()=>{

        props.history.push("/about")

    },2000)

    return(

        <div className="nav-wrapper red darken-3">

            <div className="container">

                <a className="brand-logo">Poke Times</a>

                <ul className="right">

                    <li><NavLink to="/">Home</NavLink></li>

                    <li><NavLink to="/about">About</NavLink></li>

                    <li><NavLink to="/contact">Contact</NavLink></li>

                </ul>

            </div>

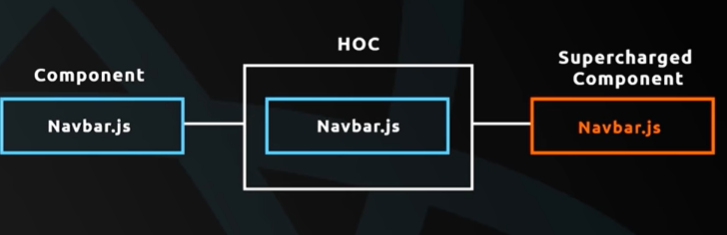
        </div>

    )

}

export default withRouter(Navbar)

the higher order component we use here is withRouter. We first import it and at the end we use the function and pass the component as parameter. This wraps the component (Navbar here) and applies the properties of the Route to the component props and then returns the final modified component.



**Higher order components**

An HOC is any component that wraps another component and gives it additional features. We can create custom HOCs.

I made a new folder called hoc and in it made a new file Rainbow.js which wraps another component and changes the text of the component to a random rainbow color.

Rainbow.js:

import React from 'react'

const Rainbow=(WrappedComponent)=>{

    const colors=['red','violet','blue','green','yellow','purple','orange']

    const randomColor=colors[Math.floor(Math.random()\*7)];

    const className=randomColor+'-text';

    return (props)=>{

        return(

            <div className={className}>

                <WrappedComponent {...props}/>

            </div>

        )

    }

}

export default Rainbow

the Rainbow function takes a component as a parameter. Here I named the component variable as WrappedComponent. Made an array for colors and got a random one by flooring the product of 7 & a random value (which is from 0-1 so can’t be used just like that)

made a string called className which is actually the name for a CSS element when used in div.

eg: red-text is the materialize CSS for getting red text

then I made a return function which takes props as parameter just so I can pass the current props to the component (optional) and then I returned the component enclosed in a <div> that changes the text color.

In order to pass the props into WrappedComponent we do it like:

<WrappedComponent {…props}>

Now to make the About component use this HOC we modify it to:

import React from 'react'

import Rainbow from '../hoc/Rainbow'

const About=()=>{

    return(

        <div className="container">

            <h4 className="center">About</h4>

            <p>Hey there this is about</p>

        </div>

    );

}

export default Rainbow(About)

so now whenever the page is **refreshed** the About component’s color changes. It has to be refreshed or else if we just navigate the request just go to the react server as we use Links or NavLinks and there is nothing to be requested from the server.

**Axios**

To show data on our site we could use REST APIs or Firebase etc. As I don’t have my own data I will be using a third party REST API called JSON placeholders which gives us access to some REST API endpoints thereby giving us dummy data in JSON format.

To extract the data we use an HTTP request library called axios which is to be installed in the project by:

npm install axios

A good place to go and acquire data was by using Lifecycle hook after the component has mounted in the DOM. To do so we must first make sure that the component we use is class-based so as to be able to use lifecycle hooks.

Then we use the componentDidMout() function to run the rest of the code after the component has mounted. To use axios we must first import it.

componentDidMount(){

        axios.get('https://jsonplaceholder.typicode.com/posts')

    }

The axios.get() is an asynchronous function. It just promises an output but there is no telling when we are going to get that output we need. To ensure the rest of the code takes place only after the function returns a value we use a .then() method which fires only when the above function is executed completely.

  componentDidMount(){

        axios.get('https://jsonplaceholder.typicode.com/posts')

        .then()

    }

Inside the .then() method we use a callback function that takes the response of the axios.get() fn as a parameter.

    componentDidMount(){

        axios.get('https://jsonplaceholder.typicode.com/posts')

        .then(res=>{

        })

    }

We use this to set the state to the first 10 data entries and then display them on the page.

Modified Home.js:

import React, {Component} from 'react'

import axios from 'axios'

class Home extends Component{

    state={

        posts:[]

    }

    componentDidMount(){

        axios.get('https://jsonplaceholder.typicode.com/posts')

        .then(res=>{

            this.setState({

                posts:res.data.slice(0,10)

            })

        })

    }

    render(){

        const {posts} = this.state;

        const postList=posts.length? (

            posts.map(post=>{

                return(

                    <div className="post card" key={post.id}>

                        <div className="card-content">

                            <span className="card-title">{post.title}</span>

                            <p>{post.body}</p>

                        </div>

                    </div>

                )

            })

            ):(

            <div className="center">No posts yet</div>

        )

        return(

            <div className="container">

                <h4 className="center">Home</h4>

                {postList}

            </div>

        )

    }

}

export default Home

and the output for this:

