React Router

It is an extension for React to create urls or *routes* to handle the view because React is library for single page application and need external supports to handle different pages.

```
import { BrowserRouter, Route } from "react-router-dom";
```

BrowserRouter is object which is going to interact with history and know whats happening in the url. **Route** is an object which knows what to do when **BrowserRouter** refers that the url is changed. Whenever we have an application and we want to use routes, it needs to wrap whole application inside the router.

```
class App extends Component {
state = {};
render() {
return (
<BrowserRouter>
<div>
{/* Wrapping in single parent */}
<Header />
{/* Header is always there regardless of path */}
<Route path="/" exact component={Home} />
<Route path="/profile" component={Profile} />
<Route path="/posts" component={Posts} />
</div>
</BrowserRouter>
);
}
}
```

React Link

In react we cannot just link with href. We have to import link for that.

```
import { BrowserRouter, Route, Link } from "react-router-dom";
```

To use this link:

```
<header>
<Link to="/">Home</Link>
<Link to="/profile">Profile</Link>
<Link to="/posts">Posts</Link>
</header>
```

With links we have some additional options. We can pass an object within link. The option can be **pathname, hash** and **search**. **Pathname** is actual route. **Hash** will add the keyword we specified at the end of every url. **Search** is gonna be the query string.

```
<Link
to={{
pathname: "/posts",
hash: "#saurya",
search: "?profile=true"
}}
>
```

Url in browser will be like this: *http://localhost:3000/posts?profile=true#saurya*

Lets add a link of *Posts* inside *Profile* component but when we click in that link it should redirect us to …../*profile/posts* instead of …../*posts* . Lets remember that when we code inside of **BrowserRouter** its passing set of options like props which gives all the information about the routes. Lets add props in one of our component and see in console what the props carry.

```
const Profile = props => {
console.log(props);
```

We will see information about *history, location, match* etc and lots of things inside them.

```
const Profile = props => {
console.log(props);
return (
<div>
<Link
to={{
pathname: `${props.match.url}/posts`
}}
>
Takes me to /profile/post
</Link>
</div>
);
};
```

Params and Urls

Lets create a module which listens a specific route and render the data related to that route. Create new file *post_item.js* which will have post detail of each post.

```
import React from "react";

const Posts = () => {
return <h1>Post Detail</h1>;
};

export default Posts;
```

Acquire *post_item* in our main component **App** and add a route to *post_item* as *post_1*

```
<Route path="/" exact component={Home} />
<Route path="/profile" component={Profile} />
<Route path="/posts" component={Posts} />
<Route path="/posts/1" component={PostItem} />
```

In *posts* component we have

```
const Posts = () => {
return (
<div>
<Link to="/posts/1">Post 1</Link>
<Link to="/posts/2">Post 2</Link>
<Link to="/posts/3">Post 3</Link>
</div>
);
};
```

Now we are still getting other post items because parent url *.../posts/* is same for all. So add **exact** to the parent url. In our main component:

```
<Route path="/posts" exact component={Posts} />
```

Now we will get rid of other post items.

Right now in `<Route path="/posts/1" component={PostItem} />`

posts/1 the id is hardcoded. In real application this id is gonna be dynamic and these are called **params**.

We will put colon and whatever we want to listen & in this case we are gonna listen to id.

```
<Route path="/posts/:id" component={PostItem} />
```

Now after */posts/* it will listen to whatever id we enter in url like *2, 45*, etc

Lets assume we also want it to listen to username as well. For this

```
<Route path="/posts/:id/:username" component={PostItem} />
```

Now it will listen to *.../any random id/any random username/* to render *PostItem*

Now to make it specific(non-random) value and grab it to a particular *PostItem* we want. First lets add props in our *PostItem* component and inside ***props.match.params*** we have all the data type we specified like id and username.

We have other type of routers like **HashRouter** and **MemoryRouter.**

Now like **Link** we can use **NavLink**. It has additional options like *activeStyle* with which we can change property whenever the link is active. To import:

```
import { BrowserRouter, Route, Link, NavLink } from "react-router-dom";
```

To use NavLink:

```
<NavLink
to="/profile"
activeStyle={{ color: "red" }}
activeClassName="selected"
>
```

Router Switch

Switch is a replacement for **exact**. To import:

```
import { BrowserRouter, Route, Link, NavLink, Switch } from "react-router-dom";
```

Wrap all the routes inside **Switch**. Now first we click *Home* and when we click profile then we get *Home*. This is because *Switch* acts like Switch – Case statement. So Home matches with ../ and it's not gonna look for other options. To make this work accurately make more specific options to top and less specific options to bottom.

```
<Switch>
<Route path="/posts/:id/:username" component={PostItem} />
<Route path="/posts" component={Posts} />
<Route path="/profile" component={Profile} />
<Route path="/" component={Home} />
</Switch>
```

Redirecting Users

```
import { Redirect } from "react-router-dom";
```

Using **Redirect**:

```
<Switch>
<Redirect from="/profile" to="/home" />
<Route path="/posts/:id/:username" component={PostItem} />
<Route path="/posts" component={Posts} />
```

Now when we click on *Profile* link then it will redirect us to *Home*. But when we are inside some route then we can only use **to**. Inside *Profile*

```
<div>
<Link
to={{
pathname: `${props.match.url}/posts`
}}
>
Takes me to /profile/post
</Link>
<Redirect to="/posts" />
</div>
```

Normally we use this for different context like when a user clicked profile but not authorized. We can use similar functionality and execute whenever our conditions met.

```
const Profile = props => {
const redir = () => props.history.push("/");
return (
<div>
<Link
to={{
pathname: `${props.match.url}/posts`
}}
>
Takes me to /profile/post
</Link>
{redir()}
</div>
);
};
```

404 Page

To implement 404 we put 404 route at end of switch but ../ will interfere because it already matched again. So again add *exact* for home route and implement 404 route as:

```
<Route path="/" exact component={Home} />
<Route render={() => <h2>404 Page not found!!!</h2>} />
</Switch>
```

If we want to render to specific component when a user is not finding:

```
<Route component={Posts} />
```

React Lifecycles

Lots of thing happening before we render something. React does everything in steps.

1. Get Default Props
2. Set Default State
3. Before Render: **componentWillMount()**,
4. **componentWillUpdate()**
5. Render Method
6. **componentDidUpdate()**
7. After Render: **componentDidMount()**

When we change or update the state *componentWillMount()* and *componentDidMount()* will not run. They will just run once. Rather *componentWillUpdate(), render(), componentDidUpdate()* will run consecutively.

Another method is **shouldComponentUpdate().** We can use this to check the changes of the props and the state. It will return either true of false. It will receive two arguments **nextProps** & **nextState**.

Lets make a clickable *div* to change the state and return either true or false in *componentShouldUpdate* to check the result.

```
<div
onClick={() =>
this.setState({
title: "This is changed message."
})
```

Before the render method:

```
shouldComponentUpdate(nextProps, nextState) {
console.log(nextState);
return true;
}
```

Now when we click the div console will return *“this is changed message”* and also shown in browser. However when we return *false* then console will return the new message but new message is not shown in browser because it is prevented to be updated(re-rendered).

Now lets add condition to update the state.

```
shouldComponentUpdate(nextProps, nextState) {
if (nextState.title === "This is changed message.") {
return false;
}

return true;
}
```

Another method is **componentWillReceiveProps().** It gets called when parent component is sending new props.

Another method is **componentWillUnmount()**. It gets called when we go out of that component. For example we are in *posts* component and we clicked *profile*, then *posts* will unmount. It will be helpful to run take actions when user is logged out etc.

Pure Components

Normally when we change the state, whatever the new state is, it will re-render. However in larger project this can cause overload. So in pure components, it will check the previous state and next state and if there is no change in new state it will not render just like the previous example of **shouldComponentUpdate().** In this case the condition is not hard-coded. Pure Component can be declared like this:

```
class LifeCycles extends PureComponent {
```

Returning Arrays in React

As for latest version of React we are allowed to directly return arrays in render.

Old Way:

```
const Posts = () => {
const ids = [
{ id: 1, name: "Post 1" },
{ id: 2, name: "Post 2" },
{ id: 3, name: "Post 3" },
{ id: 4, name: "Post 4" }
];

const eachId = ids.map(item => {
return (
<Link key={item.id} to={item.id}>
{item.name}
</Link>
);
});

return <div>{eachId}</div>;
};
```

New Way:

```
const Posts = () => {
const ids = [
{ id: 1, name: "Post 1" },
{ id: 2, name: "Post 2" },
{ id: 3, name: "Post 3" },
{ id: 4, name: "Post 4" }
];

return (
<div>
{ids.map(item => {
return (
<Link key={item.id} to={item.id}>
{item.name}
</Link>
);
})}
```

In *Simple Functional Component* there should be return statement to give output. As I remember we didn't need return in *Class Component.*

With **React 16** we could return *JSX* or *String* as an array like:

```
return (
<div>Hello</div>,
<div>I am</div>,
<div>React</div>
);
```

Or,

```
return "hello", "I am", "a react app";
```

High Order Components

One way of using it is to make a template type and pass the contents as a children. Lets make a **Card** component as a high order component. Normally we create all HOC's in different folder.

```
import React from "react";

const Card = props => {
const style = { background: "lightgrey" };

return <div style={style}>{props.children}</div>;
};

export default Card;
```

Now we can use **Card** as a parent of **Profile** as:

```
import Card from "../hoc/card";

const Profile = props => {
return (
<Card>
<Link
to={{
pathname: `${props.match.url}/posts`
}}
>
Takes me to /profile/post
</Link>
</Card>
);
```

Now we can use *Card* as a parent of any component which has a styling property of card like backgorund being lightgrey. This is the very basic way of using HOC. Lets make a different way of using HOC.

Lets pretend the profile section needs authentication. Create new file inside *hoc* folder named *auth.jsx*

```
import React from "react";

const Auth = props => {
const pass = "hesoyam";
if (pass !== "hesoyam") {
return <h3>You are not allowed</h3>;
} else {
return props.children;
}
};

export default Auth;
```

Use *Auth* in profile as:

```
import Auth from "../hoc/auth";
```

Inside *return:*

```
<Auth>
<Card>
<Link
to={{
pathname: `${props.match.url}/posts`
}}
>
Takes me to /profile/post
</Link>
</Card>
</Auth>
```

Another Way of doing:

Lets make **userHoc** which wraps **user** component. It looks weird but is very useful.

This will be continued……..

React Transition

We start with starter template with components called **TransitionComp, CSSTransition** and **Tgroup()**

TransitionComp and *CSSTransition* are simple functional components right now. In *Tgroup* we have buttons to add and remove element in a div with random number inside.

Inside *Transition.js*, the div will show and hide according to true or false in the state.

```
import React, { Component } from "react";
import "../css/App.css";

class TransitionComp extends Component {
state = {
show: false
};
render() {
return (
<div>
{this.state.show ? (
<div style={{ background: "red", height: "100px" }} />
) : null}
</div>
);
}
}

export default TransitionComp;
```

Now create a button and function to toggle the state between true and false.

```
<button className="btn btn-primary showDiv" onClick={this.showDiv}>
Show or Hide
</button>
```

Create a *showDiv* function to toggle the state.

```
showDiv = () => {
this.setState({
show: !this.state.show ? true : false
});
};
```

Now we want to add a nice transition & for this we have to install dependencies. In terminal:

npm install react-transition-group –save

Now import this as: `import Transition from "react-transition-group/Transition";`

Wrap targeted div inside *Transition* tag. It uses two main props *in* and *timeout*. *in* is like the state of *Transition.* Inside *Transition* it will return children only as a function so we should wrap children in curly braces like JS and this function has a state.

```
<Transition in={this.state.show} timeout={2000}>
{state => <h1>Hello</h1>}
</Transition>
```

Why is state there? We can check by putting *state* instead of Hello.

```
<Transition in={this.state.show} timeout={2000}>
{state => <h1>{state}</h1>}
</Transition>
```

Now when we click button it shows *entering* and after 2 second *entered.* When we click again, *exiting* and after 2 second *exited.* Now we can do animation according to these states.

```
<Transition in={this.state.show} timeout={2000}>
{state => (
<div
style={{
background: "red",
height: "100px",
transition: "all 2s ease",//Should be equal to timeout of tag for synchronization
opacity: state === "exited" || state === "exiting" ? 0 : 1
}}
>
{state}
</div>
)}
</Transition>
```

Here when div is hidden, it is occupying space. We have two options inside props of *Transition*

mountOnEnter & **unmountOnExit** which will initialize the div on enter and clear the div on exit.

```
<Transition
in={this.state.show}
timeout={2000}
mountOnEnter
unmountOnExit
>
```

Now lets avoid inline styling.

```
<Transition in={this.state.show} timeout={2000}>
{state => (
<div
className={`square square-${state}`}
>{`square square-${state}`}</div>
)}
</Transition>
```

Lets add styles to *App.css*:

```
.square {
background: red;
height: 100px;
opacity: 1;
transition: all 2s ease;
}

.square-entering {
opacity: 1;
}

.square-entered {
background: blue;
transform: translateX(0%);
}

.square-exiting {
background: rgb(20, 145, 20);
}

.square-exited {
transform: translateX(-100%);
opacity: 0;
}
```

We can have more specific usage of *timeout* in *Transition*:

```
<Transition
in={this.state.show}
timeout={{
enter: 5000,
exit: 2000
}}
>
```

5 seconds between *entering* & *entered* and 2 seconds between *exiting* & *exited*.

We also have more specific options like:

```
<Transition
in={this.state.show}
timeout={{
enter: 5000,
exit: 2000
}}
enter={false}
exit={false}
>
```

This will skip *entering* and *exiting* state. Only *entered* and *exited* state.

We can use functions in between the states

```
<Transition
in={this.state.show}
timeout={1000}
onEnter={node => console.log("Entered")}
onExit={node => console.log("Exited")}
>
```

CSS Transition

Lets move to another component **CSSTransition.js** :

```
import { CSSTransition } from "react-transition-group";

class Fade extends Component {
state = {
show: true
};

showDiv = () => {
this.setState({
show: !this.state.show ? true : false
});
};

render() {
return (
<div>
<CSSTransition in={this.state.show} timeout={2000} classNames="square">
<div className={`square ${this.state.show}`}>Hello</div>
</CSSTransition>
<button className="btn btn-primary showDiv" onClick={this.showDiv}>
Show or Hide
</button>
</div>
);
}
}
```

Inside *CSSTransition tag*, we have option called **classNames=”square”** . Now this will append group of classes of *square* like * Hello * . Blue classes are classes we assigned in div and Green classes are appended automically by *CSSTransition.* This classes will change according to the state.

Transitions Group

In *tgroup.js*:

```
import React, { Component } from "react";
import "../css/App.css";

class Slide extends Component {
state = {
items: []
};

addElements() {
return this.state.items.map((item, i) => (
<div className="item" key={i}>
{item}
</div>
));
}

generateNumber() {
//let random = Math.floor(Math.random() * 100) + 1;
let newArray = [...this.state.items, Math.floor(Math.random() * 100) + 1];

this.setState({
items: newArray
});
}

removeNumber() {
let newArray = this.state.items.slice(0, -1);
this.setState({
items: newArray
});
}

render() {
return (
<div>
{this.addElements()}

<div className="btns">
<div className="btn-add" onClick={() => this.generateNumber()}>
Add Elements
</div>
<div className="btn-remove" onClick={() => this.removeNumber()}>
Remove Elements
```

It will add new element and append with previous elements when we click **Add Elements** and delete when we click **Remove Elements.**

Main difference of *TransitionsGroup* from *Transition* and *CSSTransition* is we use this one to render lists.

To use *TransitionsGroup* we have to import :

```
import { CSSTransition, TransitionGroup } from "react-transition-group";
```

Lets wrap *this.addElements()* by *TransitionGroup*

```
<TransitionGroup component="div" className="list">
{this.addElements()}
</TransitionGroup>
```

Now when we click *addElement()* new element is added with parent div with class name *list* and child element div with classname *item.*

To use CSSTransition in every item, map the array with *CSSTransition* tag.

```
addElements() {
return this.state.items.map((item, i) => (
<CSSTransition classNames="item" timeout={2000} key={i}>
<div className="item" key={i}>
{item}
</div>
</CSSTransition>
));
}
```

Now when we click *Add Elements* and inspect the element added, then the class will be *item-enter item-enter-active* and after 2 seconds class will be *item-enter-done.* To add some classes when entered, we can use *onEntered*

```
<CSSTransition
classNames="item"
timeout={2000}
key={i}
onEntered={node => node.classList.add("active", "hawa-class")}
>
```

Add CSS to following classes:

```
.item {
opacity: 0;
transition: all 1s ease-in;
transform: translateX(-100%);
}

.item.item-enter.item-enter-active,
.item.item-enter-done.active {
transform: translateX(0);
opacity: 1;
}
```