* **Demo on Controlled Forms**

Add Form.js with the following code

import React, { Component } from 'react';

class Form extends Component{

render(){

return (

<form>

<label>

Name:

<input type="search" name="name" value="Search" />

</label>

<input type="submit" value="Submit" />

</form>

);

}

}

export default Form;

Add an entry in the grocerylist. Note on rendering you will not be able to change the value of the form.

Add constructor

constructor(){

super();

this.state = {

searchTerm : "Search"

};

}

Change the input component to

<input type="search" name="name" value={this.state.searchTerm} />

Change the value using events

**handleChange(event) {**

**this.setState({searchTerm: event.target.value});**

}

Add the following lines to the input component

onChange={this.handleChange.bind(this)}

Run to see that you can change the values

* **Demo of a uncontrolled form**

import React, { Component } from 'react';

class UncontrolledForm extends Component{

constructor(){

super();

this.state = {

searchTerm : "Search"

};

}

handleSubmit(event) {

console.log("Submitted values are: ",

event.target.name.value, event.target.email.value);

event.preventDefault();

}

render(){

return (

<form onSubmit={this.handleSubmit}>

<div className="formGroup">

Name: <input name="name" type="text" />

</div>

<div className="formGroup">

E-mail: <input name="email" type="mail" />

</div>

<button type="submit">Submit</button>

</form>

)

}

}

export default UncontrolledForm;

* **(Contacts Search Application)Create a new project for Contacts Search and type the following code**

class ContactsApp extends Component {

render(){

return(

<div>

<SearchBar />

<ContactList contacts={this.props.contacts} />

</div>

)

}

}

ContactsApp.propTypes = {

contacts: PropTypes.arrayOf(PropTypes.object)

}

class SearchBar extends Component {

render(){

return <input type="search" placeholder="search" />

}

}

class ContactList extends Component {

render(){

return(

<ul>

{this.props.contacts.map(

(contact) => <ContactItem key={contact.email}

name={contact.name}

email={contact.email} />

)}

</ul>

)

}

}

ContactList.propTypes = {

contacts: PropTypes.arrayOf(PropTypes.object)

}

class ContactItem extends Component {

render() {

return <li>{this.props.name} - {this.props.email}</li>

}

}

ContactItem.propTypes = {

name: PropTypes.string.isRequired,

email: PropTypes.string.isRequired,

}

let contacts = [

{ name: "Cassio Zen", email: "cassiozen@gmail.com" },

{ name: "Dan Abramov", email: "gaearon@somewhere.com" },

{ name: "Pete Hunt", email: "floydophone@somewhere.com" },

{ name: "Paul O'Shannessy", email: "zpao@somewhere.com" },

{ name: "Ryan Florence", email: "rpflorence@somewhere.com" },

{ name: "Sebastian Markbage", email: "sebmarkbage@here.com" },

];

ReactDOM.render(<ContactsApp contacts={contacts} />, document.getElementById('root'));

Run and check the functionality. Note that the list is not filtered. Let us add list functionality

Add this to the ContactsAPP (Add states to the ContactsApp)

**constructor(){**

**super();**

**this.state={**

**filterText: ''**

**};**

**}**

<**div**>

<**SearchBar filterText={this.state.filterText}** />

<**ContactList** contacts={this.props.contacts}

**filterText={this.state.filterText}**/>

</**div**>

Update the search bar

**return** <**input** type="search" placeholder="search"

**value={this.props.filterText}** />

// Don't forget to add the new propType requirements

**SearchBar.propTypes = {**

**filterText: PropTypes.string.isRequired**

**}**

Update ContactList Component (add the bold ones)

render(){

**let filteredContacts = this.props.contacts.filter(**

**(contact) => contact.name.indexOf(this.props.filterText)===0);**

return(

<ul>

{ **filteredContacts**.map(

(contact) => <ContactItem key={contact.email}

name={contact.name}

email={contact.email} />

)}

</ul>

)

}

\_\_ go to slide 62

After you run , you will not see that the filtered, because we have not made any changes for the Search bar to communicate to the parent. Hard code the values in filteredText to see the filtered contacts.

To communicate use events (don’t forget to remove the hardcorded text in filteredText

Update SearchBar with bold text

class SearchBar extends Component {

**handleChange(event){**

**this.props.onUserInput(event.target.value)**

**}**

render(){

return <input type="search"

placeholder="search"

value={this.props.filterText}

**onChange={this.handleChange.bind(this)}** />

}}

SearchBar.propTypes = {

**onUserInput: PropTypes.func.isRequired,**

filterText: PropTypes.string.isRequired

}

Update the ContactsApp component

<SearchBar filterText = {this.state.filterText}

**onUserInput={this.handleUserInput.bind(this)}**/>

handleUserInput(searchTerm){

this.setState({filterText:searchTerm});

}

Run to see the desired result.

Let us add data fetch in the lifecycle method componentWillMount. Install the module.

npm install --save whatwg-fetch

Add a new component ContactAppContainer

class ContactsAppContainer extends Component {

constructor(){

super();

this.state={

contacts: []

};

}

componentDidMount(){

fetch('./contacts.json')

.then((response) => response.json())

.then((responseData) => {

this.setState({contacts: responseData});

})

.catch((error) => {

console.log('Error fetching and parsing data', error);

});

}

render(){

return (

<ContactsApp contacts={this.state.contacts} />

);

}

}

Move the data to a new file contacts.json and store in the public folder.

* **Demo on Routing**

create-react-app routing

cd routing

npm install --save react-router@ 3.0.2

Copy the snippet css into app.css

Update the App.js to

import React, { Component } from 'react';

import './App.css';

import {Link} from 'react-router';

class App extends Component {

render() {

return (

<div>

<header>App</header>

<menu>

<ul>

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

<li><Link to="/repos">Repos</Link></li>

</ul>

</menu>

{this.props.children}

</div>

);

}

}

export default App;

Update the index.js to

import React from 'react';

import ReactDOM from 'react-dom';

import {Router,Route }from 'react-router';

import { browserHistory } from 'react-router'

import App from './App';

import Repos from './Repos';

import About from './About';

import './index.css';

ReactDOM.render(

<Router history={browserHistory}>

**<Route path="/" component={App}> (//Important Else the route display in a new page)**

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

<Route path="repos" component={Repos}/>

**</Route>**

</Router>,

document.getElementById('root')

);

Copy the About.js and Repos.js under Routing/simple snippets.

Run to the see the routes. Note the change in the browser url on selecting the links.

Show IndexRoute

Create a new Home.js component similar to About.js

Add the lines in the index.js

<Router history={browserHistory}>

<Route path="/" component={App}>

**<IndexRoute component={Home} />**

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

<Route path="repos" component={Repos}/>

</Route>

</Router>

You can see the home page in the “/” path.

Parameter passing

npm install --save whatwg-fetch

Update the Repos components so that we fetch the data and display in the list. Copy the snippet from “repo snippet”. Add the links and on-clicking , add the details.

<li key={repo.id}>

**<Link to={"/repos/details/"+repo.name}>{repo.name}</Link>**

</li>

Update the list to the bold ones and add the bold one after the ul tag

<div>

<h1>Github Repos</h1>

<ul>

{repos}

</ul>

**{this.props.children}**

</div>

Copy the Repo-details snippet.

Update the index.js

<Route path="repos" component={Repos}>

{/\* Add the route, nested where we want the UI to nest \*/}

<Route path="details/:repo\_name" component={RepoDetails} />

</Route>

Run to see the output

Update the bold snippets to Repo

render() {

let repos = this.state.repositories.map((repo) => (

<li key={repo.id}>

<Link to={"/repos/details/"+repo.name}>{repo.name}</Link>

</li>

));

**let child = this.props.children && React.cloneElement(this.props.children,**

**{ repositories: this.state.repositories }**

**); ..**

**</ul>**

**~~{this.props.children}~~ {child}**

**</div>**

RepoDetails Need not have the lifecycle methods since the data comes from its parent thro props. Comment out all the lifecycle methods and put the lines below

renderRepository() {

let repository = this.props.repositories.

find((repo)=>repo.name === this.props.params.repo\_name);

let stars = [];

for (var i = 0; i < repository.stargazers\_count; i++) {

stars.push('☆');

}

return (

<div>

<h2>{repository.name}</h2>

<p>{repository.description}</p>

<span>{stars}</span>

</div>

);

}

render() {

if(this.props.repositories.length > 0 ){

return this.renderRepository();

} else {

return <h4>Loading...</h4>;

}

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Crate a new React project FluxBank and npm install –save flux. Copy the index.css from snippets

**Step 1**

Create constants.js

export default {

CREATED\_ACCOUNT: 'created account',

WITHDREW\_FROM\_ACCOUNT: 'withdrew from account',

DEPOSITED\_INTO\_ACCOUNT: 'deposited into account'

};

**Step 2**

Create an AppDispatcher that extends the Dispatcher that dispatches the payload(actions)

import {Dispatcher} from 'flux';

class AppDispatcher extends Dispatcher{

dispatch(action = {}) {

console.log("Dispatched", action);

super.dispatch(action);

}

}

export default new AppDispatcher();

**Step 3**

Create BankActions.js class which has static methods defined for createAccount,depositAmount and withdrawAmount

import AppDispatcher from './AppDispatcher';

import bankConstants from './constants';

class BankActions{

static createAccount() {

AppDispatcher.dispatch({

type: bankConstants.CREATED\_ACCOUNT,

ammount: 0

});

}

static depositIntoAccount(ammount) {

AppDispatcher.dispatch({

type: bankConstants.DEPOSITED\_INTO\_ACCOUNT,

ammount: ammount

});

}

static withdrawFromAccount(ammount) {

AppDispatcher.dispatch({

type: bankConstants.WITHDREW\_FROM\_ACCOUNT,

ammount: ammount

});

}

}

export default BankActions;

**Step 4**

Create a store that extends the ReduceStore

import AppDispatcher from './AppDispatcher';

import bankConstants from './constants';

import {ReduceStore} from 'flux/utils';

class BankBalanceStore extends ReduceStore {

getInitialState() {

return 0;

}

reduce(state, action){

switch (action.type) {

case bankConstants.CREATED\_ACCOUNT:

return 0;

case bankConstants.DEPOSITED\_INTO\_ACCOUNT:

return state + action.ammount;

case bankConstants.WITHDREW\_FROM\_ACCOUNT:

return state - action.ammount;

default:

return state;

}}}

export default new BankBalanceStore(AppDispatcher);

**Step 5**

Create a view with render method, getStores and calculateState. Create App.js and copy the snippets from FluxBank.

static getStores(){

return [BankBalanceStore];

}

static calculateState(prevState){

return{ balance: BankBalanceStore.getState()};

}

**Step 6**

Create AppContainer.js with

import {Container} from 'flux/utils';

import App from './App';

export default Container.create(App);

**Run to see the application. Go to slide 27**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Add a store**

* Create BankRewardStore.js (Displays rewards based on the balance )

Copy the code from the snippets and add the following reduce method

reduce(state, action){

this.getDispatcher().waitFor([

BankBalanceStore.getDispatchToken()

]);

* Update the store and the state in App.js

static getStores(){

return [BankBalanceStore, **BankRewardsStore**];

}

static calculateState(prevState){

return{

balance: BankBalanceStore.getState(),

**rewardsTier: BankRewardsStore.getState()**

};

}

* Uncomment the <h2> tag

Run to see the app

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Counter Redux Example

Create a React – redux-counter. Install react-redux

1. Create Constants.js with

export const INCREASE = "INCREASE";

1. Create actions/index.js. Actions contain simple object with “type” as mandatory. Can have other payload data

export const increaseAction = { type: INCREASE }

1. Create reducers/index.js

Reducers are simple functions that take state and action and returns a new state based on the action

export function counter(state = { count: 0 }, action) {

const count = state.count

switch (action.type) {

case INCREASE:

return { count: count + 1 }

default:

return state

}

}

1. Create Counter.js which is the view component

class Counter extends Component {

render() {

const { value, onIncreaseClick } = this.props

return (

<div>

<span>{value}</span>

<button onClick={onIncreaseClick}>Increase</button>

</div>

)

}

}

Counter.propTypes = {

value: PropTypes.number.isRequired,

onIncreaseClick: PropTypes.func.isRequired

}

export default Counter;

1. Update index.js to
2. Create a store from the reducer

import { createStore } from 'redux'

const store = createStore(counter)

1. // Map Redux state to component props

function mapStateToProps(state) {

return {

value: state.count

}

}

1. // Map Redux actions to component props

function mapDispatchToProps(dispatch) {

return {

onIncreaseClick: () => dispatch(increaseAction)

}

}

1. create a component by connecting the mapStateToProps and maprDispatchToProps using connect function with the view component ( Counter )

import { Provider, connect } from 'react-redux'

// Connected Component

const App = connect(

mapStateToProps,

mapDispatchToProps

)(Counter)

1. Wrap this component to Provider (Redux ) and add the store to its props so that the child components are Redux Context aware.

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

Install Redux Chrome Developer Tools.

npm install –save redux-thunk

For checking the Time debugging. Copy the configureStore snippet and create a configureStore.js

Change the store to

const store = configureStore();

Exercise for participants. Include Decrease for the counter.

Solution

**Update Constants**

const constants={

INCREASE: "INCREASE",

DECREASE: "DECREASE"

}

export default constants;

**Update Counter.js**

Add a button, props

<button onClick={onDecreaseClick}>Decrease</button>

const { value, onIncreaseClick, **onDecreaseClick** } = this.props

**Update actions/index.js**

import constants from '../constants';

export const CounterActions = {

increaseAction:{type: constants.INCREASE},

decreaseAction: {type: constants.DECREASE}

}

**Update index.js**

// Map Redux actions to component props

function mapDispatchToProps(dispatch) {

return {

onIncreaseClick: () => dispatch(CounterActions.increaseAction),

**onDecreaseClick: () => dispatch(CounterActions.decreaseAction)**

}

}