Form Events

- submit
- change
- blur
- focus

Document/Window Events

- load
- unload
- scroll
- resize

Note: A term "fires" is generally used with events. For example: The click event fires in the moment you press a key.

Syntax for event methods

Most of the DOM events have an equivalent jQuery method. To assign a click events to all paragraph on a page, do this:

1. $("p").click ();

The next step defines what should happen when the event fires. You must pass a function to the event.

UNIT – III

REACT JS

React Introduction

ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components. It is an open-source, component-based front end library responsible only for the view layer of the application. It was created by **Jordan Walke,** who was a software engineer at **Facebook.** It was initially developed and maintained by Facebook and was later used in its products like **WhatsApp**&**Instagram.** Facebook developed ReactJS in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013.**

Today, most of the websites are built using MVC (model view controller) architecture. In MVC architecture, React is the 'V' which stands for view, whereas the architecture is provided by the Redux or Flux.

A ReactJS application is made up of multiple components, each component responsible for outputting a small, reusable piece of HTML code. The components are the heart of all React applications. These Components can be nested with other components to allow complex applications to be built of simple building blocks. ReactJS uses virtual DOM based mechanism to fill data in HTML DOM. The virtual DOM works fast as it only changes individual DOM elements instead of reloading complete DOM every time.

To create React app, we write React components that correspond to various elements. We organize these components inside higher level components which define the application structure. For example, we take a form that consists of many elements like input fields, labels, or buttons. We can write each element of the form as React components, and then we combine it into a higher-level component, i.e., the form component itself. The form components would specify the structure of the form along with elements inside of it.

Why learn ReactJS?

Today, many JavaScript frameworks are available in the market(like angular, node), but still, React came into the market and gained popularity amongst them. The previous frameworks follow the traditional data flow structure, which uses the DOM (Document Object Model). DOM is an object which is created by the browser each time a web page is loaded. It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page. This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application.

Therefore, a new technology ReactJS framework invented which remove this drawback. ReactJS allows you to divide your entire application into various components. ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM. It means rather than manipulating the document in a browser after changes to our data, it resolves changes on a DOM built and run entirely in memory. After the virtual DOM has been updated, React determines what changes made to the actual browser's DOM. The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM. Due to this, when we write a React component, we did not write directly to the DOM; instead, we are writing virtual components that react will turn into the DOM.

React Router and Single Page Applications

Preparing the React App

Installing the create-react-app Package

If you've ever had the chance to try React, you've probably heard about the **create-react-app** package, which makes it super easy to start with a React development environment.

In this tutorial, we will use this package to initiate our React app.

So, first of all, make sure you have Node.js installed on your computer. It will also install

npm for you.

In your terminal, run npm install -g create-react-app. This will globally install **create-react-app** on your computer.

Once it is done, you can verify whether it is there by typing create-react-app -V.

Creating the React Project

Now it's time to build our React project. Just run create-react-app multi-page-app. You can, of course, replace multi-page-app with anything you want.

Now, **create-react-app** will create a folder named **multi-page-app**. Just type cd multi-page-app to change directory, and now run npm start to initialize a local server.

That's all. You have a React app running on your local server.

Now it's time to clean the default files and prepare our application.

In your src folder, delete everything but App.js and index.js. Then open index.js and replace the content with the code below.

```
import React from'react';


importReactDOMfrom'react-dom';


import App from'./App';


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

I basically deleted the registerServiceWorker related lines and also the import './index.css'; line.

Also, replace your App.js file with the code below.

```
import React, { Component } from'react';


class App extends Component {


render() {
```

```
return (

<div className="App">


</div>


    );

  }

}


exportdefault App;
```

Now we will install the required modules.

In your terminal, type the following commands to install the **react-router** and **react-transition-group** modules respectively.

```
npm install react-router-dom --save

npm install react-transition-group@1.x --save
```

After installing the packages, you can check the package.json file inside your main project directory to verify that the modules are included under **dependencies.**

Router Components

There are basically two different router options: **HashRouter** and **BrowserRouter**.

As the name implies, **HashRouter** uses hashes to keep track of your links, and it is suitable for static servers. On the other hand, if you have a dynamic server, it is a better option to use **BrowserRouter**, considering the fact that your URLs will be prettier.

Once you decide which one you should use, just go ahead and add the component to your index.js file.

```
import { HashRouter } from 'react-router-dom'
```

The next thing is to wrap our <App> component with the router component.

So your final index.js file should look like this:

```
import React from'react';

importReactDOMfrom'react-dom';

import{ HashRouter } from'react-router-dom'

import App from'./App';

ReactDOM.render(<HashRouter><App/></HashRouter>, document.getElementById('root'));
```

If you're using a dynamic server and prefer to use **BrowserRouter**, the only difference would be importing the **BrowserRouter** and using it to wrap the <App> component.

By wrapping our <App> component, we are serving the **history** object to our application, and thus other react-router components can communicate with each other.

Inside <App/> Component

Inside our <App> component, we will have two components named <Menu> and <Content>. As the names imply, they will hold the navigation menu and displayed content respectively.

Create a folder named "**components**" in your src directory, and then create the Menu.js and Content.js files.

Menu.js

Let's fill in our Menu.js component.

It will be a stateless functional component since we don't need states and life-cycle hooks.

```
import React from'react'

const Menu = () =>{

return(
```

```
<ul>

<li>Home</li>

<li>Works</li>

<li>About</li>

</ul>

  )

}

exportdefault Menu
```

Here we have a <ul> tag with <li> tags, which will be our links.

Now add the following line to your **Menu** component.

```
import { Link } from 'react-router-dom'
```

And then wrap the content of the <li> tags with the <Link> component.

The <Link> component is essentially a **react-router** component acting like an <a> tag, but it does not reload your page with a new target link.

Also, if you style your a tag in CSS, you will notice that the <Link> component gets the same styling.

Note that there is a more advanced version of the <Link> component, which is <NavLink>. This offers you extra features so that you can style the active links.

Now we need to define where each link will navigate. For this purpose, the <Link> component has a to prop.

```
import React from'react'
```

```
import{ Link } from'react-router-dom'

const Menu = () =>{

return(

<ul>

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

<li><Link to="/works">Works</Link></li>

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

</ul>

  )

}

exportdefault Menu
```

Content.js

Inside our <Content> component, we will define the **Routes** to match the **Links.**

We need the Switch and Route components from **react-router-dom.** So, first of all, import them.

```
import { Switch, Route } from 'react-router-dom'
```

Second of all, import the components that we want to route to. These are the Home, Works and About components for our example. Assuming you have already created those components inside the **components** folder, we also need to import them.

```
import Home from './Home'

import Works from './Works'

import About from './About'
```

Those components can be anything. I just defined them as stateless functional components with minimum content. An example template is below. You can use this for all three components, but just don't forget to change the names accordingly.

```
import React from'react'


const Home = () =>{


return(


<div>


    Home


</div>


  )


}


exportdefault Home
```

Switch

We use the <Switch> component to group our <Route> components. **Switch** looks for all the **Routes** and then returns the first matching one.

Route

Routes are components calling your target component if it matches the path prop.

The final version of our Content.js file looks like this:

```
import React from'react'

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

import Home from'./Home'

import Works from'./Works'

import About from'./About'

const Content = () =>{

return(

<Switch>

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

<Route path="/works" component={Works}/>

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

</Switch>

  )

}

exportdefault Content
```

Notice that the extra exact prop is required for the **Home** component, which is the main directory. Using exactforces the **Route** to match the exact pathname. If it's not used, other pathnames starting with / would also be matched by the **Home** component, and for each link, it would only display the **Home** component.

Now when you click the menu links, your app should be switching the content.

Animating the Route Transitions

So far, we have a working router system. Now we will animate the route transitions. In order to achieve this, we will use the **react-transition-group** module.

We will be animating the *mounting* state of each component. When you route different components with the **Route** component inside **Switch**, you are essentially *mounting* and *unmounting* different components accordingly.

We will use **react-transition-group** in each component we want to animate. So you can have a different mounting animation for each component. I will only use one animation for all of them.

As an example, let's use the <Home> component.

First, we need to import **CSSTransitionGroup**.

```
import { CSSTransitionGroup } from 'react-transition-group'
```

Then you need to wrap your content with it.

Since we are dealing with the mounting state of the component, we enable transitionAppear and set a timeout for it. We also disable transitionEnter and transitionLeave, since these are only valid once the component is mounted. If you are planning to animate any children of the component, you have to use them.

Lastly, add the specific transitionName so that we can refer to it inside the CSS file.

```
import React from'react'


import{ CSSTransitionGroup } from'react-transition-group'


import'../styles/homeStyle.css'



const Home = () =>{
```

```
return(
<CSSTransitionGroup
transitionName="homeTransition"
transitionAppear={true}
transitionAppearTimeout={500}
transitionEnter={false}
transitionLeave={false}>
<div>
      Home
</div>
</CSSTransitionGroup>
  )
}
exportdefault Home
```

We also imported a CSS file, where we define the CSS transitions.

```
.homeTransition-appear{
```

```
opacity: 0;

}

.homeTransition-appear.homeTransition-appear-active{

opacity: 1;

transition: all .5s ease-in-out;

}
```

If you refresh the page, you should see the fade-in effect of the **Home** component.

If you apply the same procedure to all the other routed components, you will see their individual animations when you change the content with your **Menu.**

Conclusion

In this tutorial, we covered the **react-router-dom** and **react-transition-group** modules. However, there's more to both modules than we covered in this tutorial. Here is a **working demo** of what was covered.

So, to learn more features, always go through the documentation of the modules you are using.

Over the last couple of years, React has grown in popularity. In fact, we have a number of items in the marketplace that are available for purchase, review, implementation, and so on. If you're looking for additional resources around React, don't hesitate to check them out.

React Forms

HTML form elements work a bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
<label>
   Name:
<input type="text" name="name" />
```

```
</label>
<input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "controlled components".

Controlled Components

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the "single source of truth". Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a "controlled component".

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:

```
classNameFormextendsReact.Component{
constructor(props){
super(props);
this.state={value:''};
this.handleChange=this.handleChange.bind(this);
this.handleSubmit=this.handleSubmit.bind(this);
}

handleChange(event){this.setState({value:event.target.value});}
handleSubmit(event){
alert('A name was submitted: '+this.state.value);
event.preventDefault();
}

render(){
return(
<form onSubmit={this.handleSubmit}><label>
      Name:
<input type="text" value={this.state.value} onChange={this.handleChange} /></label>
<input type="submit" value="Submit" />
</form>
);
}
}
```

Try it on CodePen

Since the value attribute is set on our form element, the displayed value will always be this.state.value, making the React state the source of truth. Since handleChange runs on every keystroke to update the React state, the displayed value will update as the user types.

With a controlled component, the input's value is always driven by the React state. While this means you have to type a bit more code, you can now pass the value to other UI elements too, or reset it from other event handlers.

The textarea Tag

In HTML, a <textarea> element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

In React, a <textarea> uses a value attribute instead. This way, a form using a <textarea> can be written very similarly to a form that uses a single-line input:

```
classEssayFormextendsReact.Component{
constructor(props){
super(props);
this.state={value:'Please write an essay about your favorite DOM element.'};
this.handleChange=this.handleChange.bind(this);
this.handleSubmit=this.handleSubmit.bind(this);
}

handleChange(event){this.setState({value:event.target.value});}
handleSubmit(event){
alert('An essay was submitted: '+this.state.value);
event.preventDefault();
}

render(){
return(
<form onSubmit={this.handleSubmit}>
<label>
        Essay:
<textarea value={this.state.value} onChange={this.handleChange} /></label>
<input type="submit" value="Submit" />
</form>
);
}
}
```

Notice that this.state.value is initialized in the constructor, so that the text area starts off with some text in it.

The select Tag

In HTML, <select> creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```
<select>
<option value="grapefruit">Grapefruit</option>
<option value="lime">Lime</option>
<option selected value="coconut">Coconut</option>
<option value="mango">Mango</option>
```

```
</select>
```

Note that the Coconut option is initially selected, because of the selected attribute. React, instead of using this selected attribute, uses a value attribute on the root select tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
classFlavorFormextendsReact.Component{
constructor(props){
super(props);
this.state={value:'coconut'};
this.handleChange=this.handleChange.bind(this);
this.handleSubmit=this.handleSubmit.bind(this);
}

handleChange(event){this.setState({value:event.target.value});}
handleSubmit(event){
alert('Your favoriteflavor is: '+this.state.value);
event.preventDefault();
}

render(){
return(
<form onSubmit={this.handleSubmit}>
<label>
      Pick your favoriteflavor:
<select value={this.state.value} onChange={this.handleChange}><option
value="grapefruit">Grapefruit</option>
<option value="lime">Lime</option>
<option value="coconut">Coconut</option>
<option value="mango">Mango</option>
</select>
</label>
<input type="submit" value="Submit" />
</form>
);
}
}
```

Try it on CodePen

Overall, this makes it so that <input type="text">, <textarea>, and <select> all work very similarly - they all accept a value attribute that you can use to implement a controlled component.

Note

You can pass an array into the value attribute, allowing you to select multiple options in a select tag:

```
<select multiple={true} value={['B', 'C']}>
```

The file input Tag

In HTML, an <input type="file"> lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the File API.

```
<input type="file" />
```

Because its value is read-only, it is an **uncontrolled** component in React. It is discussed together with other uncontrolled components later in the documentation.

Handling Multiple Inputs

When you need to handle multiple controlled input elements, you can add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name.

For example:

```
classReservationextendsReact.Component{
constructor(props){
super(props);
this.state={
isGoing:true,
numberOfGuests:2
};

this.handleInputChange=this.handleInputChange.bind(this);
}

handleInputChange(event){
const target =event.target;
const value =target.type==='checkbox'?target.checked:target.value;
const name = target.name;
this.setState({
[name]: value});
}

render(){
return(
<form>
<label>
        Is going:
<input
name="isGoing"          type="checkbox"
          checked={this.state.isGoing}
onChange={this.handleInputChange} />
</label>
<br />
<label>
        Number of guests:
<input
name="numberOfGuests"          type="number"
          value={this.state.numberOfGuests}
onChange={this.handleInputChange} />
</label>
</form>
);
}
}
```

Try it on CodePen

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

```
this.setState({
[name]: value});
```

It is equivalent to this ES5 code:

```
varpartialState={};
partialState[name]=value;this.setState(partialState);
```

Also, since setState() automatically merges a partial state into the current state, we only needed to call it with the changed parts.

Controlled Input Null Value

Specifying the value prop on a controlled component prevents the user from changing the input unless you desire so. If you've specified a value but the input is still editable, you may have accidentally set value to undefined or null.

The following code demonstrates this. (The input is locked at first but becomes editable after a short delay.)

```
ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function(){
ReactDOM.createRoot(mountNode).render(<input value={null} />);
},1000);
```

Alternatives to Controlled Components

It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out uncontrolled components, an alternative technique for implementing input forms.

Introduction to Redux

Redux Toolkit

Redux Toolkit is our official recommended approach for writing Redux logic. It wraps around the Redux core, and contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

RTK includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, **Redux Toolkit** can help you make your Redux code better.

Redux Toolkit is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npminstall @reduxjs/toolkit

# Yarn
yarnadd @reduxjs/toolkit
```

Create a React Redux App

The recommended way to start new apps with React and Redux is by using the official Redux+JS template or Redux+TS template for Create React App, which takes advantage of **Redux Toolkit** and React Redux's integration with React components.

```
# Redux + Plain JS template
npx create-react-app my-app --template redux

# Redux + TypeScript template
npx create-react-app my-app --template redux-typescript
```

Redux Core

The Redux core library is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npminstall redux

# Yarn
yarnadd redux
```

It is also available as a precompiled UMD package that defines a `window.Redux` global variable. The UMD package can be used as a <script> tag directly.

For more details, see the Installation page.

Basic Example

The whole global state of your app is stored in an object tree inside a single *store*. The only way to change the state tree is to create an *action*, an object describing what happened, and *dispatch* it to the store. To specify how state gets updated in response to an action, you write pure *reducer* functions that calculate a new state based on the old state and the action.

```
import { createStore } from 'redux'

/**
 * This is a reducer - a function that takes a current state value and an
 * action object describing "what happened", and returns a new state value.
 * A reducer's function signature is: (state, action) => newState
```

```
 *
 * The Redux state should contain only plain JS objects, arrays, and primitives.
 * The root state value is usually an object. It's important that you should
 * not mutate the state object, but return a new object if the state changes.
 *
 * You can use any conditional logic you want in a reducer. In this example,
 * we use a switch statement, but it's not required.
 */
functioncounterReducer(state = { value: 0 }, action) {
switch (action.type) {
case'counter/incremented':
return { value: state.value + 1 }
case'counter/decremented':
return { value: state.value - 1 }
default:
return state
  }
}

// Create a Redux store holding the state of your app.
// Its API is { subscribe, dispatch, getState }.
let store = createStore(counterReducer)

// You can use subscribe() to update the UI in response to state changes.
// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.
// There may be additional use cases where it's helpful to subscribe as well.

store.subscribe(() => console.log(store.getState()))

// The only way to mutate the internal state is to dispatch an action.
// The actions can be serialized, logged or stored and later replayed.
store.dispatch({ type: 'counter/incremented' })
// {value: 1}
store.dispatch({ type: 'counter/incremented' })
// {value: 2}
store.dispatch({ type: 'counter/decremented' })
// {value: 1}
```

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called *actions*. Then you write a special function called a *reducer* to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like a lot for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

Redux Toolkit Example

Redux Toolkit simplifies the process of writing Redux logic and setting up the store. With Redux Toolkit, that same logic looks like:

```
import { createSlice, configureStore } from '@reduxjs/toolkit'

const counterSlice = createSlice({
name: 'counter',
initialState: {
value: 0
  },
reducers: {
incremented: state => {
// Redux Toolkit allows us to write "mutating" logic in reducers. It
// doesn't actually mutate the state because it uses the Immer library,
// which detects changes to a "draft state" and produces a brand new
// immutable state based off those changes
state.value += 1
    },
decremented: state => {
state.value -= 1
    }
  }
})

exportconst { incremented, decremented } = counterSlice.actions

const store = configureStore({
reducer: counterSlice.reducer
})

// Can still subscribe to the store
store.subscribe(() => console.log(store.getState()))

// Still pass action objects to `dispatch`, but they're created for us
store.dispatch(incremented())
// {value: 1}
store.dispatch(incremented())
// {value: 2}
store.dispatch(decremented())
// {value: 1}
```

Redux Toolkit allows us to write shorter logic that's easier to read, while still following the same Redux behavior and data flow.

Learn Redux

We have a variety of resources available to help you learn Redux.

Redux Essentials Tutorial

The **Redux Essentials tutorial** is a "top-down" tutorial that teaches "how to use Redux the right way", using our latest recommended APIs and best practices. We recommend starting there.

Redux Fundamentals Tutorial

The **Redux Fundamentals tutorial** is a "bottom-up" tutorial that teaches "how Redux works" from first principles and without any abstractions, and why standard Redux usage

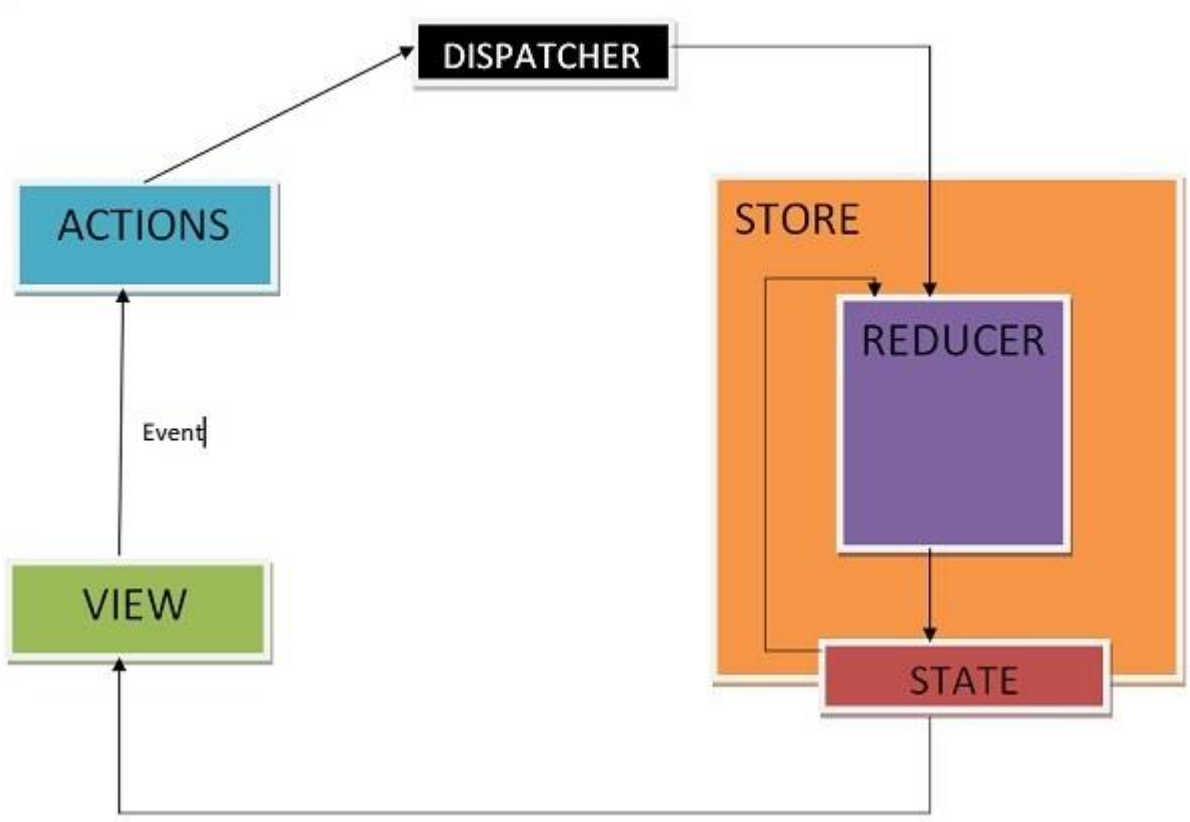
patterns exist.

Redux - Data Flow

Redux - Data Flow

Redux follows the unidirectional data flow. It means that your application data will follow in one-way binding data flow. As the application grows & becomes complex, it is hard to reproduce issues and add new features if you have no control over the state of your application.

Redux reduces the complexity of the code, by enforcing the restriction on how and when state update can happen. This way, managing updated states is easy. We already know about the restrictions as the three principles of Redux. Following diagram will help you understand Redux data flow better −



- An action is dispatched when a user interacts with the application.
- The root reducer function is called with the current state and the dispatched action. The root reducer may divide the task among smaller reducer functions, which ultimately returns a new state.
- The store notifies the view by executing their callback functions.
- The view can retrieve updated state and re-render again.

Client-Server Communication

Let's expand the application so that the notes are stored in the backend. We'll use json-server, familiar from part 2.

The initial state of the database is stored in the file *db.json*, which is placed in the root of the project:

```
{
  "notes": [
    {
      "content": "the app state is in redux store",
      "important": true,
      "id": 1
    },
    {
      "content": "state changes are made with actions",
      "important": false,
      "id": 2
    }
  ]
}
```

We'll install json-server for the project...

```
npm install json-server --save-dev
```

and add the following line to the *scripts* part of the file *package.json*

```
"scripts": {
  "server": "json-server -p3001 --watch db.json",
  // ...
}
```

Now let's launch json-server with the command *npm run server*.

Next, we'll create a method into the file *services/notes.js*, which uses *axios* to fetch data from the backend

```
import axios from 'axios'

constbaseUrl = 'http://localhost:3001/notes'

constgetAll = async () => {
const response = await axios.get(baseUrl)
  return response.data
}

export default { getAll }
```

We'll add axios to the project

```
npm install axios
```

We'll change the initialization of the state in *noteReducer*, so that by default there are no notes:

```
constnoteSlice = createSlice({
  name: 'notes',
initialState: [],  // ...
})
```

Let's also add a new action *appendNote* for adding a note object:

```
constnoteSlice = createSlice({
  name: 'notes',
initialState: [],
  reducers: {
createNote(state, action) {
const content = action.payload

state.push({
        content,
        important: false,
        id: generateId(),
      })
    },
toggleImportanceOf(state, action) {
const id = action.payload

constnoteToChange = state.find(n => n.id === id)

constchangedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }

     return state.map(note =>
note.id !== id ? note : changedNote
    )
   },
appendNote(state, action) {      state.push(action.payload)    }  },
})

export const{ createNote, toggleImportanceOf, appendNote } = noteSlice.actions
export default noteSlice.reducer
```

A quick way to initialize the notes state based on the data received from the server is to fetch the notes in the *index.js* file and dispatch an action using the *appendNote* action creator for each individual note object:

```
// ...
import noteService from './services/notes'importnoteReducer, { appendNote } from './reducers/noteReducer'
const store = configureStore({
  reducer: {
    notes: noteReducer,
    filter: filterReducer,
  }
})

noteService.getAll().then(notes =>notes.forEach(note => {    store.dispatch(appendNote(note))  }))
// ...
```

Dispatching multiple actions seems a bit impractical. Let's add an action creator *setNotes* which can be used to directly replace the notes array. We'll get the action creator from the

createSlice function by implementing the *setNotes* action:

```
// ...

constnoteSlice = createSlice({
  name: 'notes',
initialState: [],
  reducers: {
createNote(state, action) {
const content = action.payload

state.push({
        content,
        important: false,
        id: generateId(),
      })
    },
toggleImportanceOf(state, action) {
const id = action.payload

constnoteToChange = state.find(n => n.id === id)

constchangedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }

     return state.map(note =>
note.id !== id ? note : changedNote
    )
   },
appendNote(state, action) {
state.push(action.payload)
   },
setNotes(state, action) {      return action.payload    }  },
})

export const{ createNote, toggleImportanceOf, appendNote, setNotes } = noteSlice.actions
export default noteSlice.reducer
```

Now, the code in the *index.js* file looks a lot better:

```
// ...
import noteService from './services/notes'
import noteReducer, { setNotes } from './reducers/noteReducer'
const store = configureStore({
  reducer: {
    notes: noteReducer,
    filter: filterReducer,
  }
})

noteService.getAll().then(notes =>
store.dispatch(setNotes(notes)))
```

NB: why didn't we use await in place of promises and event handlers (registered to *then*-methods)?

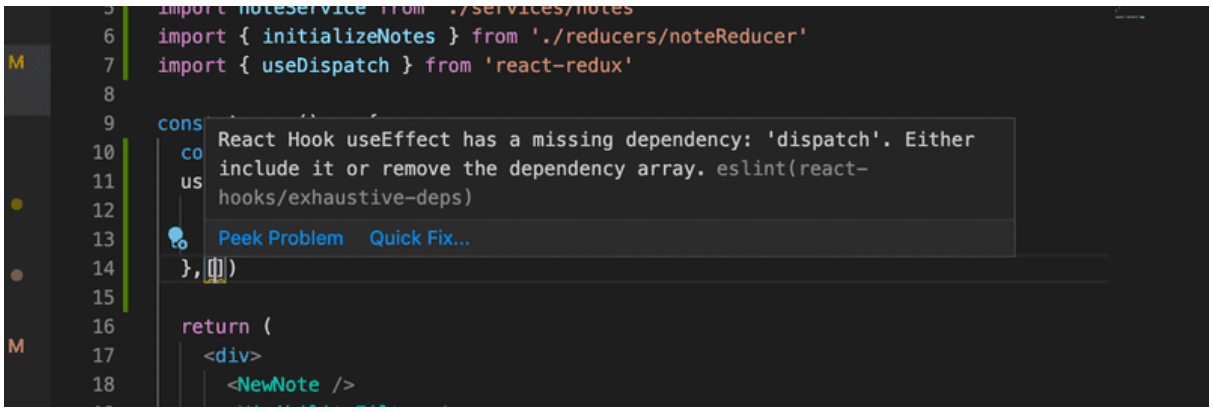
Await only works inside *async* functions, and the code in *index.js* is not inside a function, so due to the simple nature of the operation, we'll abstain from using *async* this time.

We do, however, decide to move the initialization of the notes into the *App* component, and, as usual, when fetching data from a server, we'll use the *effect hook*.

```
import { useEffect } from 'react'importNewNote from './components/NewNote'
import Notes from './components/Notes'
import VisibilityFilter from './components/VisibilityFilter'
import noteService from './services/notes'import { setNotes } from './reducers/noteReducer'import { useDispatch
} from 'react-redux'
const App = () => {
const dispatch = useDispatch()  useEffect(() => {    noteService      .getAll().then(notes =>
dispatch(setNotes(notes)))  }, [])
  return (
<div>
<NewNote />
<VisibilityFilter />
<Notes />
</div>
  )
}

export default App
```

Using the useEffect hook causes an eslint warning:



We can get rid of it by doing the following:

```
const App = () => {
const dispatch = useDispatch()
useEffect(() => {
noteService
.getAll().then(notes => dispatch(setNotes(notes)))
  }, [dispatch])
  // ...
}
```

Now the variable *dispatch* we define in the *App* component, which practically is the dispatch function of the redux store, has been added to the array useEffect receives as a parameter. **If** the value of the dispatch variable would change during runtime, the effect would be executed again. This however cannot happen in our application, so the warning is unnecessary.

Another way to get rid of the warning would be to disable ESlint on that line:

```
const App = () => {
const dispatch = useDispatch()
useEffect(() => {
noteService
.getAll().then(notes => dispatch(setNotes(notes)))
  }, []) // eslint-disable-line react-hooks/exhaustive-deps
  // ...
}
```

Generally disabling ESlint when it throws a warning is not a good idea. Even though the ESlint rule in question has caused some arguments, we will use the first solution.

More about the need to define the hooks dependencies in the react documentation.

We can do the same thing when it comes to creating a new note. Let's expand the code communicating with the server as follows:

```
constbaseUrl = 'http://localhost:3001/notes'

constgetAll = async () => {
const response = await axios.get(baseUrl)
  return response.data
}

constcreateNew = async (content) =>{  const object = { content, important: false }  const response = await
axios.post(baseUrl, object)  return response.data}
export default {
getAll,
createNew,
}
```

The method *addNote* of the component *NewNote* changes slightly:

```
import { useDispatch } from 'react-redux'
import { createNote } from '../reducers/noteReducer'
import noteService from '../services/notes'
constNewNote = (props) => {
const dispatch = useDispatch()

constaddNote = async (event) =>{  event.preventDefault()
const content = event.target.note.value
event.target.note.value = ''
constnewNote = await noteService.createNew(content)    dispatch(createNote(newNote))  }

  return (
<form onSubmit={addNote}>
<input name="note" />
<button type="submit">add</button>
</form>
  )
}

export default NewNote
```

Because the backend generates ids for the notes, we'll change the action creator *createNote*

accordingly:

```
createNote(state, action) {
state.push(action.payload)
}
```

Changing the importance of notes could be implemented using the same principle, by making an asynchronous method call to the server and then dispatching an appropriate action.

UNIT – IV

Java Web Development

Web development is known as website development or web application development. The web development creates, maintains, and updates web development applications using a browser. This web development requires web designing, backend programming, and database management. The development process requires software technology.

Web development creates web applications using servers. We can use a web server or machine server like a CPU. The Web server or virtual server requires web application using technology. Web development requires server-side programming language or technology. Mostly Java, PHP, and other server-side languages require for web development.

Java web development creates a server-side website and web application. The majority of Java web apps do not execute on the server directly. A web container on the server hosts Java web applications.

For Java web applications, the container acts as a runtime environment. What the Java Virtual Machine is for locally running Java applications, the container is for Java web applications. JVM is used to run the container itself.

Java distinguishes between two types of containers: web and Java EE. Additional functionality, such as server load distribution, can be supported by a container. A web container supports Java servlets and JSP (JavaServer Pages). In Java technology, Tomcat is a common web container.

A web container is usually a minimal need for web frameworks. GWT, Struts, JavaServer Faces, and the Spring framework are common Java web frameworks. Servlets are at the heart of most modern Java web frameworks.