View katmanını kullanır. (modal-view-controller). Virtual DOM ‘u kullanır. Yani sadece değişiklik yapılan kısımları ve onların bağlı olduğu state ‘leri **render()** ediyor. Component tabanlı bir dildir. Kendi router ve routing sistemi yoktur?!. (bundan dolayı framework değişdir aslında, kütüphanedir).

**Virtual DOM** ise ; gerçek dom’un bellekteki kopyasıdır. Document objeleri key-value olarak tutar. Güncelleme anında bellekte 2 adet virdual dom bulunur(snapshot) . sayfade sadece statelerin değiştiği yerler güncellenir. Yani kopya dom ile orijinal dom un farklarını bularak tespit ediliyor.

Function componentlerin Component’den extends edilmesi gerekmiyor. Ama bütün class componentlerin Component’den extends edilmesi gerekiyor. Eğer ki componentlerin içerisinde jsX formatında bir şey döneceksek **render()** olmak zorunda. App.js ‘in içerisinde bulunan serviceWorker ise pwa(progressive web apps) yapmamıza olanak sağlayan bir yapıdır.

**jsX formatı**; **render()** içerisinde return ile gelmektedir. Return içerisinde sadece bir parent element olmak zorundadır. **Render()** sadece jsX formatı return edilirken kullanılmalı, içerinde setState’in oluşturulmasına izin vermez-hata verir.

**Props’lar** bir componentten başka bir componente veri aktarımı yapmamızı sağlayan bir özelliktir. Proptypes ve defaultProps’lar ise; özellikle gelmesi gereken bir props varsa props’un key’i: PropTypes.string.isRequired & defaultProps ise props’un key’i:”value” şeklinde atanabilmektedir. Yani hangi props’un öenmli olduğunu ve türünü belirleyebiliyoruz ve gelmeyen props’lar içinde default değerler atayabiliyoruz.

Class componentlerinin mutlaka **render** fonksiyonu olması gerekmektedir. **Render** methodu ise component classından miras alınınca otomatik olarak gelen bir methodtur. Bu methodu isteğimize göre biçimlendirip override ediyoruz ve componentlerimizi sayfamıza yerleştiriyoruz.

**State**; bir componentin o anki durumunu gösteren ve değişik property lerden oluşan js objesidir. Her component statelere sahip olabilir ve o stateler o componentin durumunu (özelliklerini) simgeler diyebiliriz. İki şekilde verilmektedirler. 1-constructor ların içerisinde , 2-state in kendi adıyla objesinş oluşturabiliriz. State ler sadece setState fonksiyonu ile değiştirilebilirler.

İlk çağlarda; bu state ler her bir componente özel olduğu için başka bir componentin kullanması için props mantığıyla aktarılması, taşınması gerekiyordu. Bir family tree olduğu için main parent component den (app.js), içerisine dahil edilen componentlere aktarılması gerekiyor. ( 5. Child componentin o state i kullanması için 5 componentten sırasıyla aktarılması gerekiyordu). Prop drilling yöntemi deniyor.

Bunun önüne geçmek için context-api veya redux mantığı ile yapmak gerekmektedir. Bu stateleri; redux ta (storage componenti) içerisinde tutuyoruz, context-api de ise (provider componenti) içerisinde tutuyoruz. Context-api da bulunan provider componenti tüm family tree yapısını(react yapısını) sarmallıyor, kapsama alanına alıyor ve böylelikle oluşturulan state yapısını istediğimiz yere -props mantığı ile aktarmadan- taşıyabiliyoruz. Provider componenti bize data mızı sağlıyor, consumer componenti ise bu data yı istediğimiz yerde kullanmamızı sağlıyor. Küçük -orta ölçekli uygulamalarda context-api daha makul ama büyük ölçekli uygulamalarda redux yapısı daha mantıklıdır.

const UserContext = React.createContext();

Context-api; React.createContext() ile context yapısı oluşturuyoruz ve bir değişkene örneğin const UserContext;. **Render()** -return yapısı içerisinde UserContext.Provider diyerek

      <UserContext.Provider *value*={this.state}>

        {this.props.children}

      </UserContext.Provider>

Şeklinde bir yapı oluşturuyoruz. Burada ki props.children yapısı ise sarmal yapısı oluşturduğumuz zaman App.js i ifade ediyor. Value ile de var olan state yapımızı vermiş oluyoruz. (axios.get yapısı ile de gelebilir, kendimiz de oluşturabiliriz). Context.js içerisinde ki class ın adı UserProvider.

ReactDOM.render(

  <React.StrictMode>

    <UserProvider>

      <App />

    </UserProvider>

  </React.StrictMode>,

  document.getElementById("root")

);

Şeklinde index.js içerisinde ise sarmal yapımızı oluşturuyoruz. Böylelikle yukarıda bahsettiğim this.props.children yapısında App e denk geliyor.

const UserConsumer = UserContext.Consumer;

export default UserConsumer;

şeklinde UserContext ten Consumer a ulaştık? UserProvider yapısı içerisinde ki state lerimi kullanmamız için UserConsumer lazım. Bu nesneyi diğer componentlerin kendi içerisinde sarmal yapısı oluşturmasında kullanıyoruz. İşte bu şekilde yapıyoruz.

  render() {

    return (

      <UserConsumer>

        {(*value*) => {

          const { dispatch } = *value*;

          return ();

        }}

      </UserConsumer>

    );

  }

}

**Action**; Context-api de provider componentimizin içerisinde hangi işlemin gerçekleşeceğini ve hangi verilerin göndereleceğini belirten js objesidir. 2 tür property bulundurur. Type: hangi işlemin gerçekleşeceği – Payload: hangi verinin gönderileceğini belirtir.

    dispatch({ type: "DELETE\_USER", payload: id });

**Dispatch**; action ları context.js ye göndermek ile görevli js fonksiyonudur. Provider state nin içerisinde bulunur.

**Reducer**; gelen action ın tipine göre state i değiştirecek işlemlerden sorumlu bir js fonksiyonudur.

export class UserProvider extends Component {

  state = {

    users: [],

    dispatch: (*action*) => {

      this.setState((*state*) => reducer(*state*, *action*));

    },

  };

Şeklinde UserProvider ın içerisinde state in içerisinde bulunur ve gelen actionlara göre state lerde değişiklik yapılmasını sağlar. Reducer fonksiyonu ise örneğin şöyledir;

const reducer = (*state*, *action*) => {

  switch (*action*.type) {

    case "DELETE\_USER":

      return {

        ...*state*,

        users: *state*.users.filter((*user*) => *action*.payload !== *user*.id),

      };

    case "ADD\_USER":

      return {

        ...*state*,

        users: [...*state*.users, *action*.payload],

      };

    case "UPDATE\_USER":

      return {

        ...*state*,

        users: *state*.users.map((*data*) =>

*data*.id === *action*.payload.id ? *action*.payload : *data*

        ),

      };

    default:

      return *state*;

  }

};

**MOUNTING**; Bir component (componentler aslında class’dır diyebiliriz) sayfaya yerleştirilecekse bu işleme Mounting denilmektedir. İlk başta constructor’ı çağırılmaktadır. Burada propslarımızı geçirebiliyoruz, statelerimizi oluşturabiliyoruz. Daha sonra ki işlem ise, bu componentin render işlemi (jsX formatında render işlemi). Componentimiz sayfaya mount edildikten sonra son işlem olan **componentDidMount** (cdm) fonksiyonu çalışmaktadır.(lifecycle methodlarından birisidir. Component yerleşmiş oluyor.

**UPDATING;** bir parent componente bağlı child componentler varsa, parent component update ediliyorsa ve bu componente bağlı stateler ve propslar da update edilir. Yani bu propsların ve statelerin içerisinde bulunduğu componentler de update edilmiş olur. New props – setState() – ve daha az kullanılan forceUpdate() olmak üzere update etmenin birkaç yolu mevcuttur. Daha sonra ise yine render işlemi gerçekleşmektedir. (React updates, DOM and refs). En sonunda ise **componentDidUpdate**(cdu) fonksiyonu çalışmaktadır.

**UNMOUTING**; bir componentin sayfadan kaldırılma işlemidir. **componentWillUnmount** fonksiyonu çalışmaktadır.

* **getDerivedStateFromProps**(),**shouldComponentUpdate**(),**getSnapshotBeforeUpdate**() olmak üzere bu life-cycle da olan 3 tane fonksiyon daha vardır. Ama bunlar teknik ayrıntı kısmında yer almaktadır. Daha sonra açıklamasını yazacağım ...
* render fonksiyonu içerisinde jsX formatını kullanmak gerekir. State operasyonlarını yapsak bile hata verecektir. “Maximum update depth exceeded” yani; setState her değiştiğinde render’ı update eder, render da her update edildiğinde setState’i çağırır. İfinity loop olur. Sonuç olarak render’ın tek görevi jsX formatında o componentin yapısını dönmelidir.
* **componentDidMount**(); api istekleri yapmak istersek bu fonksiyonun içerisinde gerçekleştirebiliyoruz. Bir component yükleneceği zaman Constructor–Render–**ComponentDidMount** çalışmaktadır. Eğer ki **componentDidMount** içerisinde bi setState veya api isteği alıp yine bir update yaparsak, Render() methodu tekrar çalışmaktadır. Yani **Constructor** – **Render** – **ComponentDidMount** - **Render** olur. (**componentDidUpdate**() fonksiyonu burada yoktur). Bir nevi veri yüklemek için kullanılmaktadır. Sayfa ekrana ilk basıldığında en sonunda çalışmaktadır.
* **componentDidUpdate**() fonksiyonu ise (**New** **props** – **setState**() – ve daha az kullanılan **forceUpdate**()) gerçekleşirse çalışır. componentDidUpdate(prevProps, prevState) şeklinde iki tane parametre almaktadır. Props ve state’in update edilmeden önceki hallerine eşittir. Önceki datalarını kullanmak istersek bu parametrelerden yapıyoruz. Yani **Constructor** – **Render** – **ComponentDidMount** – **Render** – **componentDidUpdate** şeklinde ki sırayla çalışıyorlar. İçerik güncellendiğinde çalışmaktadır. Yeni bir props veya state değiştirildiğinde çalışmaktadır.
* **shouldComponentUpdate**(), bu fonksiyon ise **New** **State**-**New** **Props** olduğu zaman yani update edildiğinde veya yeni bir data eklendiğinde çalışmaktadır. Eğer ki çağırılıp yazılmazsa; return true dönmektedir, default olarak böyledir. Ama çağırılıp şöyle de kullanılabilir. Seçilen bir state’in veya props’un sonradan update edilmesine, değişmesine izin verilmeyecekse bu func. çağırılır ve false return edilir. Böylelikle yeniden render etmeyecektir. **Constructor** – **Render** – **ComponentDidMount** - **shouldComponentUpdate** – **Render** – **componentDidUpdate**
* **componentWillUnmount()** ise bir component silinmeden hemen önce çalışmaktadır. Örneğin bir silme işlemi gerçekleşecek ve bu silme işleminin id’si ne ait kayıtlı başka abonelikler var. İşte bunlarında kaldırılması için kullanılabilir.

Reducer ‘ın görevi sadece state değiştirmek olursa çok güzel olur

**React BrowserRouter** “react-router-dom” paketinden

import { BrowserRouter as Router, Route, Switch } from "react-router-dom"; şeklinde componentlerimizi dahil ediyoruz. React’ta sayfa yönlendirmelerini bu paket ile kolay bir şekilde yapabiliyoruz. => ( bu örnek için Browser as Router ‘ ı yoksayıp, sadece Router olarak import edilmiştir.)

**<Router>**

**<Route path=”/pagex” component = {...} />**

**<Router>**

Eğer bu şekilde yaparsak hata verir. “cannot read property location of undefined” hatasını verir. Sebebi, Router’ı kullanabilmemiz için Browser History’e ulaşmamız lazım. 2 çözüm yolu vardır. Yukarıda yoksaydığımız kısımı dahil edersek BrowserRouter ı Router olarak alıp Routeryerine yazarsak böylece; history özelliğini kullanıp Router işlemini yapabilir hale geliyoruz. BrowserRouter direkt kendisi olduğu tarayıcıda ki history değerini alabiliyor.

import createHistory from “history/createBrowseHistory” , şeklinde createHistory yi dahil edip <Router history={ createHistory()} > , şeklinde parametre olarak geçmeliyiz. Burada ki Router, yoksayılmış router. yani BrowserRouter as Router olan değil. Sadece salt Router u kullanırsak history değerini dışarıdan vermemiz lazım. Bu yüzden parametre olarak tanımlıyoruz.

return (

<Router>

<div>

<Switch>

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

<Route exact path="/add" component={AddUser} />

<Route exact path="/edit/:id" component={UpdateUser} />

<Route component={notFound} />

</Switch>

</div>

</Router>

);

Router => BrowserRouter as Router olarak tanımladık. History değerini vermemize gerek yoktur. Hepsi parent div in içerisindeler yoksa “many child” hatası verecektir.

Switch ise; sağlanan tüm pathlere aynı anda girip göstermesin diye kullanıyoruz. Sadece switch kullansak ve aşağısına exact koymadan salt path ve componentleri yazsak yine istediğimiz gibi çalışmaz. Çünkü burada da ilk satırın path ine girer aşağıda ki pathlere geçmez. Bu yüzden exact koyuyoruz ki sadece o path geldiğinde çalıştır. Diğer pathlerin gelmesine müdahale etme diyoruz. Eğer ki exact olmadan yazarsak hiyerşiye göre yazmamız lazım ve ilk child tan parent a doğru yazmamız lazım. En kapsayıcı path en aşağıda olması lazım. (exact={true} da yazabiliriz). Yani exact kullanıdığımızda route ların bir sırasının önemi olmuyor.

En aşağıda exact olmadan, componenti NotFound olan son satır ise zaten adı üstünde farklı bir path, olmayan bir path veya yanlış bir path girildiğinde “aradığınız sayfa bulunamadı” yazısı çıkmaktadır.

* Src nin içerisinde (index.js) ReactDom.render ... , getElementById(“root”) , ayarlamalarını yapıyoruz. Burada <App/> i redux yapısı için <Provider/> yapısının içerisine alacağız ve burada stateler i store u içerisine koyacağız ki ulaşılabilsin.

**<Provider store={createStore(reducers)}>**

**<app/>**

**</Provider>**

İmport {Provider} from “react-redux”

İmport {createStore} from “redux”

Actions(index.sj) içerisinde actionlar olur. State ile ne yapılmak isteniyorsa, add-increase-decreas-.., burada ki functionların 2 tipi vardır. Type ve payload ları (isimler değişebilir ama ne olduğu anlaşılsın diye bu isimler kullanılır). Return olarak bu iki tipi dönerler. Type: “ADD\_BOOK” .. tarzında string yapıda olur. Payload ise, functionun kendisine parametre olarak döner, bu aldığı parametreleri ise payload: {..} içerisine obje olarak yazılır. İçerisinde hangi işlemlerin gerçekleşeceğini hangi verilerin gönderileceğini belirten js objesidir.

**Type: hangi işlemin gerçekleşeceğini**

**Paylaod: hangi veri türünün gönderileceğini**

Reducers(index.js) ise gelen action ın tipine göre state i değiştirecek işlemlerden sorumlu bir js fonksiyonudur.

**İmport {combineReducers} from “redux”**

Redux tan kullanıdığımız 2.fonk (1.createStore) ne işe yarıyor ? bizim reducer larımızı dışarıyla bağlantı kurmasını sağlıyor. Böylelikle action dan gelen isteklere cevap verebilecek.

Burada action.type lara göre switch, else if ypaısını kullaranak func. yazıyoruz. Mesela yapacağımız işlemleri ana kategorilere bölebiliyoruz. Hepsini ayrı ayrı da yazabiliriz. Daha kullanışlı olsun diye yoksa her kontrol için bir sürü fonksiyon yazarız.

**Const bookListReducer = (oldBookList = books, action) =>{**

**Switch(action.type)**

**Case “ADD\_BOOK”:**

**Return: ...**

**Const selectedBookReducer = (oldBookList = {}, action) =>{**

**Switch(action.type)**

**Case “ADD\_BOOK”:**

**Return: ...**

**Export default combineReducers({**

**bookList: bookListReducer,**

**selectedBook: selectedBookReducer,**

**})**

Şeklinde iki ana kategori, 1 ana başlık altında switch case yapısı ile action ları alabiliyoruz. En sonunda ise ; redux tan import ettiğimiz combineReducers ile 2 ana başlık. Func ları Reducers ların func ları olarak (combine leyip) export ederiz. Nerede kullanılacak peki ??

Bu reducers(index.js) functionları nerede kullanırız ?

Components klasörünün altında herhangi bir folder da kullanmak istedik. Default import ların dışında ek olarak

1 .hangi action u kullanmak istiyorsak onu import ediyoruz

2 .import {connect} from “react-redux”

Bu action ile state arasında, bu componenti bağlamak için kullanırız

**Class BookDetails extends Component{**

**.**

**.**

**Render(){**

**<div> {this.getContent()}, //şeklinde func halinde ayrı bir func oluşturup kullanabiliriz.**

**} }**

**Const mapStateToProps = (state) =>{**

**Return {selectedBook : state.selectedBook} ;**

**Export default connect(mapStateToProps, {unselectedBook})(BookList)**

unSelectedBook => herhangi bir action dur. Hangisini kullanmak istersek, action name

mapStateToProps ? bu func ın componentlerde kullanılması zorunlu değildir. Eğer ki yeni bir action u veya yeni bir state i veya varolan state leri ilk defa ekrana bastırmak istiyorsak, kullanmak istiyorsak bu func ı kullanırız. Nasıl ?

(state) parametresini alacaktır. Sonra return edecektir. Reducersta yaptığımız 2 ana başlıktan birinin adını yazıyoruz. Çünkü onları combineReducers layıp export ettik. Varolan state leri props olarak (props a) map le, yani bağla diyoruz.

Daha sonra aşağıda export default connect() () yapısını kullanıyoruz. Componenti export ediyoruz. Connect ise “react-redux” paketinden geliyor. Bizim componentimiz ile mapStateToProps(varsa, yoksa null yazabiliriz), {actions})(component\_class\_name). Actions ise, hangi actions u kullanacaksak onu yazıyoruz.

**Class ...**

**getContent({**

**const {selectedBook} = this.props;**

// şimdi bu ana reducers ın içerisinde ki action lara erişebileceğiz.

**{selectedBook.title}**

**{selectedBook.description}**

....

**<div onClick = { ()=> {this.props.unSelectedBook();}}>**

Şeklinde ise; bu div i clicklediğimiz zaman “unSelectedBook()” action una ulaşabiliyoruz. Yani actions tan unSelectedBook action una ulaşabiliyoruz.

Component in içerisine state almak için connect methodunu import etmemiz lazım. Nereden ? => “react-redux” paketinin içinden. Peki bu stateleri nasıl kullanacağız ? mapStateToProps func. ile varolan statelerimizi props olarak tekrar bize sunar ve böylece dahil ettiğimiz stateleri props olarak kullanabiliriz.

Connect methodunun bir diğer elemanı ise; actionslardan hangisine ihtiyacımız varsa onu dahil ederiz. Böylece o action ı kullanabiliriz

**Redux- thunk;** reducers ile actions arasına ara katman (yani middleware) ekliyor. Bunu yapmasının sebebi, actions func ları nşa da js objesi dönmektedir. Type ve payload u olan. Ama biz action yapımızı obje olarak değil de func olarak dönmek istersek ( async yapıda bir func) işte burada karmaşa oluyor. Reducers js objesi bekliyor ama actions ise func bir yapı dönüyor böyle olunca reducers a type ve payload olan obje iletilmiyor. Bu sorunu çözmek için “redux-thunk” yapısını kullanıyoruz. Thunk yapısı ile async bir işlemi yapabiliyoruz. Bu async func ın içerisinde dispatch ile js objesini reducers a gönderiyoruz. Böylelikle hem async bir işlem yapıp hem de reducers ın anlayacağı js objesi şeklinde dispatch göndererek sorunu çözüyoruz.

**THE REACT II & Hooks API Reference**

* **useState**

The setState function is used to update the state. It accepts a new state value and enqueues a re-render of the component.

During subsequent re-renders, the first value returned by useState will always be the most recent state after applying updates.

const [state, setState] = useState(initialState);

Returns a stateful value, and a function to update it.

setState(newState);

function Counter({*initialCount*}) {

  const [count, setCount] = useState(*initialCount*);

  return (

    <>

      Count: {count}

      <button *onClick*={() => setCount(*initialCount*)}>Reset</button>

      <button *onClick*={() => setCount(*prevCount* => *prevCount* - 1)}>-</button>

      <button *onClick*={() => setCount(*prevCount* => *prevCount* + 1)}>+</button>

    </>

  );

}

const [state, setState] = useState({});

setState(*prevState* => {

*// Object.assign would also work*

  return {...*prevState*, ...updatedValues};

});

* **useEffect**

React'in useEffect'i çalıştırmayı tarayıcı boyanıncaya kadar erteler. Yani sayfa paint edildikten hemen sonra çalıştırır. Yani en son çalışır.(componenDidMount mantığında)

useEffect(

  () => {

    const subscription = props.source.subscribe();

    return () => {

      subscription.unsubscribe();

    };

  },

  [props.source],

);

Now the subscription will only be recreated when props.source changes.

componentDidUpdate mantığında çalışır.

useEffect(

  () => {

...

  },

  [],

);

Sadece (mounting ve unmounting) işleminde çalışmasını istersek, ikinci bir parametre olarak “[]” geçebiliriz. Böylelikle bileşen ilk mounting edildiğinde ve unmounting edildiğinde çalışır.

componentDidMount mantığında çalışır.

* **useContext**

Context Api mantığında bir yapı oluşturursak kullanıyoruz

const value = useContext(MyContext);

* **Correct:** useContext(MyContext)
* **Incorrect:** useContext(MyContext.Consumer)
* **Incorrect:** useContext(MyContext.Provider)

export const UserContext = React.createContext();

export const useAppContext = () => React.useContext(UserContext);

..

const {

    loginName,

    setLoginName,

} = useAppContext();

* **useReducer**

const [state, dispatch] = useReducer(reducer, initialArg, init);

An alternative to [useState](https://reactjs.org/docs/hooks-reference.html#usestate). Accepts a reducer of type (state, action) => newState, and returns the current state paired with a dispatch method. (If you’re familiar with Redux, you already know how this works.)

Here’s the counter example from the [useState](https://reactjs.org/docs/hooks-reference.html#usestate) section, rewritten to use a reducer:

const initialState = { count: 0 };

function reducer(*state*, *action*) {

  switch (*action*.type) {

    case "increment":

      return { count: *state*.count + 1 };

    case "decrement":

      return { count: *state*.count - 1 };

    default:

      throw new Error();

  }

}

function Counter() {

  const [state, dispatch] = useReducer(reducer, initialState);

  return (

    <>

      Count: {state.count}

      <button *onClick*={() => dispatch({ type: "decrement" })}>-</button>

      <button *onClick*={() => dispatch({ type: "increment" })}>+</button>

    </>

  );

}

const useReducer = (*reducer*, *initialState*) => {

  const [state, setState] = useState(*initialState*);

  const dispatch = (*action*) => {

    const newState = reducer(state, *action*);

    setState(newState);

  };

  return [state, dispatch];

};

const todosReducer = (*state*, *action*) => {

  switch (*action*.type) {

    case "ADD\_USER":

      return {

        ...*state*,

        users: [...*state*, *action*.paylaod],

      };

    default:

      return *state*;

  }

};

const Test = () => {

  const [todos, dispatch] = useReducer(todosReducer, initialState);

  const firstAdd = (*test*) => {

    dispatch({ type: "ADD\_USER", payload: *test* });

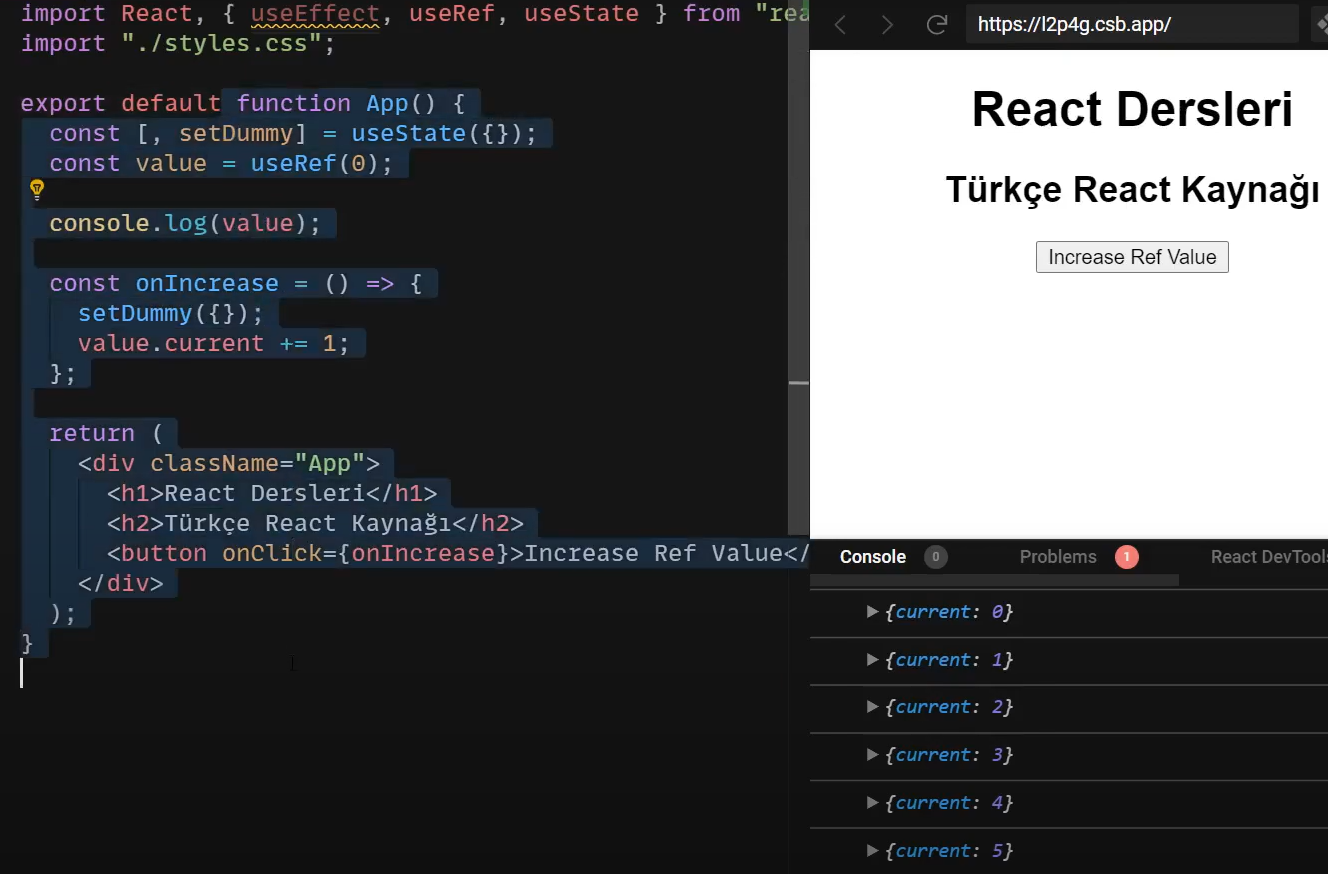
  };

};

### useRef

.jsx içerisinde DOM yapısında ki bir aktivite için veya DOM yapısında ki elemente ulaşmanın kolay yoludur. Örneğin dinamik olarak style a erişip özellik ekleyip-değiştireceğiz. Addeventlistener(“click”) veya Addeventlistener(“load”) gibi şeyler kullanacağız ve bundan sonra çeşitli işlemler yapcacağız. Bu yüzden useRef kullanmak oldukça mantıklıkdır. Genellikle 2 şekilde kullanabiliriz.

1 - ;



Addeventlistener(“click”) yerine => button ın onClick özelliği ile onIncrease i çağırıyoruz. Daha sonra value.current ile de REF verdiğimiz elementi aktif edip, istediğimiz koşulu yerine getiriyoruz. İçerideki setState ne oluyor peki? Bunu olmasının sebebi ; onIncrease in her “click” te return-çalışması için içerisine bir obje referans veriyoruz. Usestate ile bir state-hook oluşturuyoruz. Amacımız sadece obje referans vermektir. Bundan dolayı state’e ihtiyacımız yok. Zaten sadece setDummy vardır. Ki o da initial state olarak {} obje tutmaktadır. React mantığına göre eğer ki render edilen tür vey return edilen element, referans veri tipiyse yani objeyse her defasında tekrar yenilenir. Bunun sebebi her return olduğunda, objenin bellekteki yeri tekrar atanır. Yukarıda ki örnekte bunu yapmasaydık, her clicke bastığımızda çalışmayacaktı .çünkü herhangi bir state imiz yok ve olmayan state imiz update edilmiyor. UseRef i atadığımız value yi de, button ın içerisine ref olarak vermedik. Bundan dolayı bu örnekte useRef i bu şekilde kullanmalıyız.

2 - ;

  const openNav = useRef();

  const closeNavX = useRef();

  const closeNav = useRef();

  useEffect(() => {

    openNav.current.onclick = () => {

      document.documentElement.style.setProperty("--nav-width", "100%");

    };

    closeNavX.current.onclick = () => {

      document.documentElement.style.setProperty("--nav-width", "0%");

    };

    closeNav.current.onclick = () => {

      document.documentElement.style.setProperty("--nav-width", "0%");

    };

  }, []);

...

    <div *id*="myNav" *className*="overlay">

      <a *href*="#!" *className*="closebtn" *ref*={*closeNavX*}>

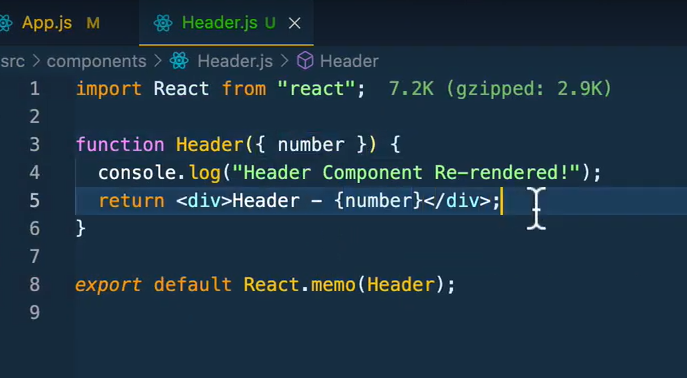
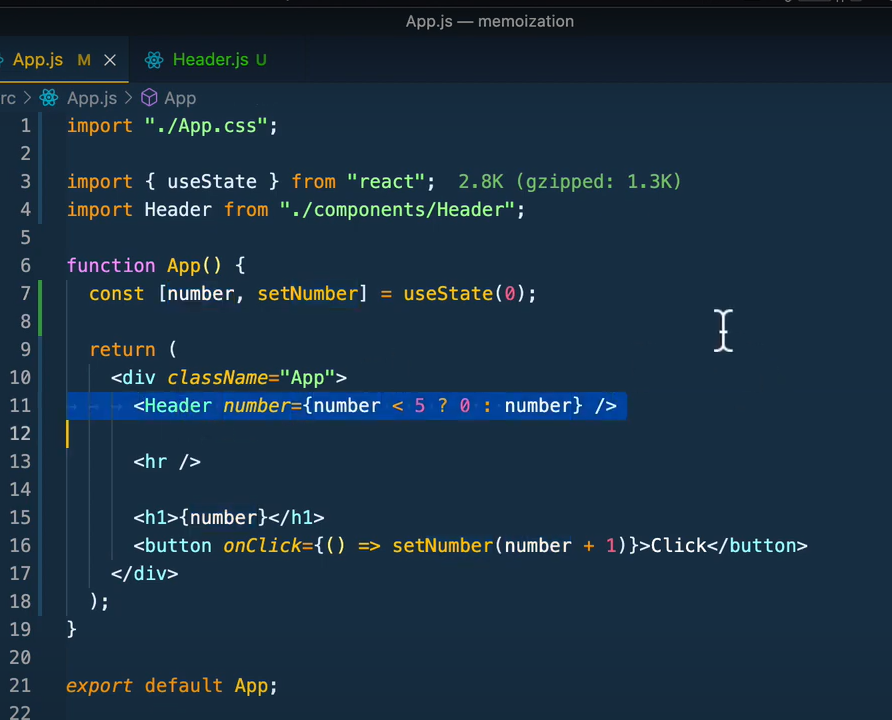
        &times;

      </a>

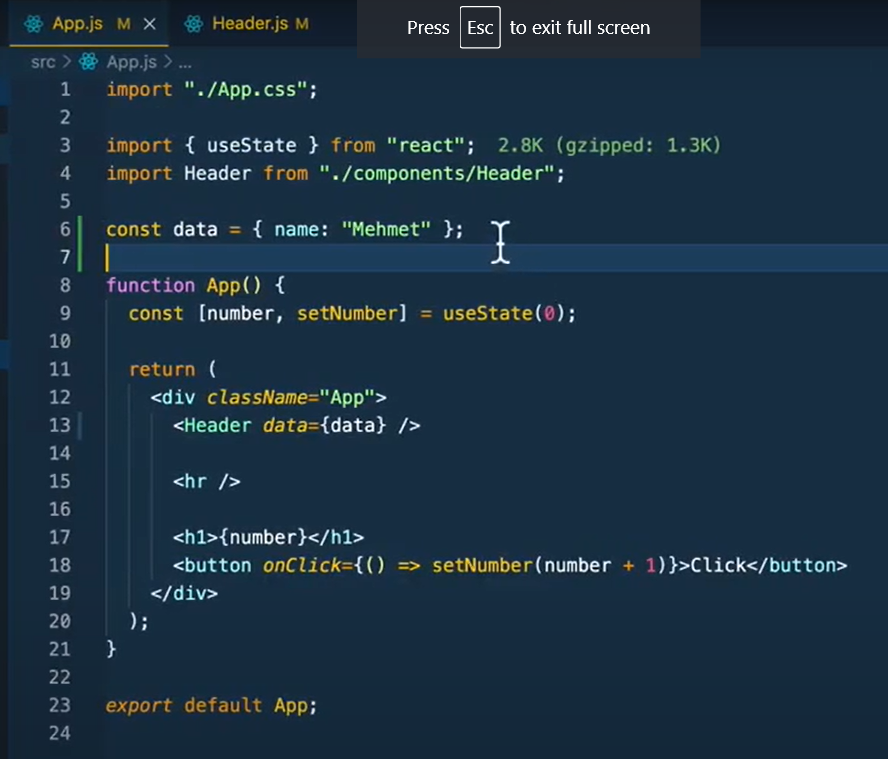
...

2. kullanım da bu şekildedir. useRef ile değişkenler oluşturduk. Ve bunları kullanmak istediğimiz elemnentlere referans olarak verdik. Daha sonra bu ref’leri useEffect’e bağladık. Böylelikle sayfa her mount edildiğinde ya da değişkenlerimiz değiştiğinde useEffect ile beraber bağlı olan reflerimiz de etkilenip-değişecektir.

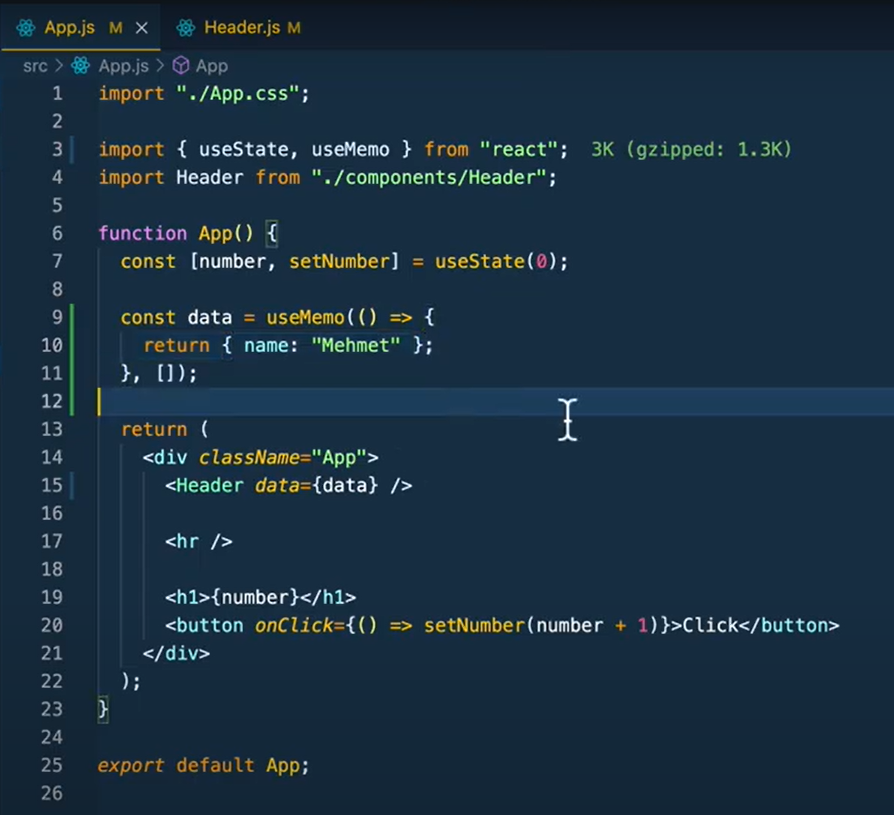
### useMemo - ReactMemo



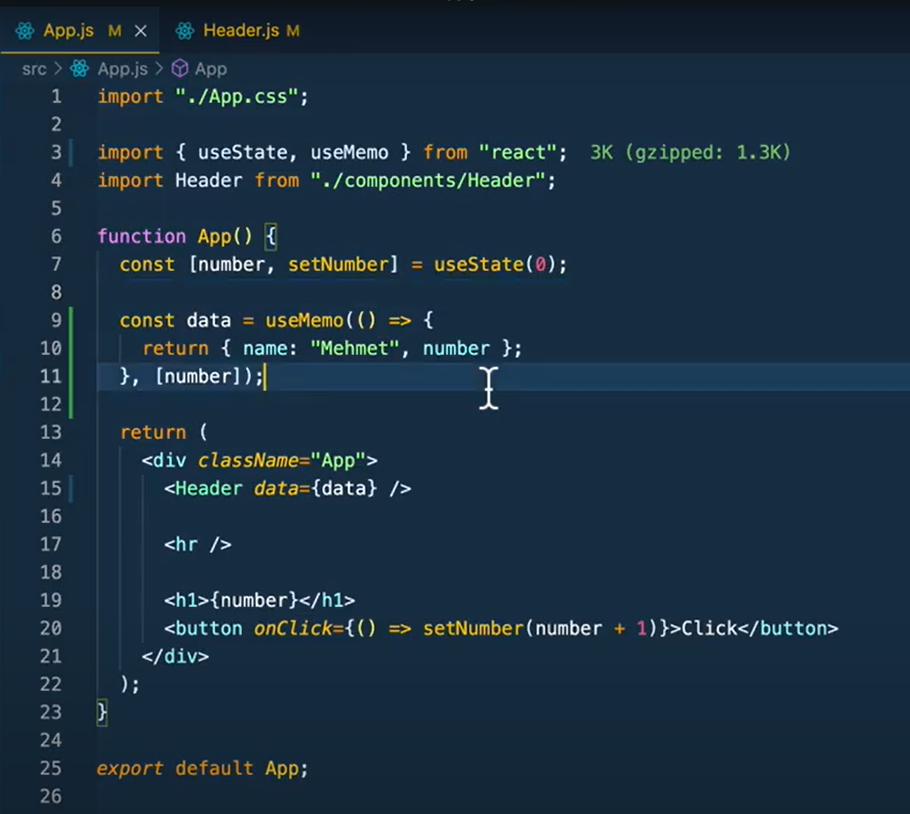
İlk resimde ki button her çalıştığında yukarıda ki header da update edilecektir. Ama bunu istemeyiz. Çünkü sayfa her returun edildiğinde o component de çalışacaktır. Gereksiz yere render edilecektir. Bunu önlemek için o component (HEADER) export default React.memo(Header) şeklinde export edilirse sorun çözülecektir.



Ya da bu şekilde de önleyebiliriz. Header componentine her defasında data yı gönderiyoruz fakat bu data function App in dışında yer aldığı için, button her click edildiğinde tekrar tekrar render edilmeyecektir.



Ama bu data props unu function App in içerisine yazmak zorunda kalırsak, işte o zaman useMemo() hook unu kullanacağız. UseMemo yu kullanarak gereksiz render ın önüne geçmiş olduk.



Peki Header componentinin de kendi içerisinde gönderdiğimiz dataya bağlı bir şekilde render edilmesini istersek, aynı useEffect in parametre alması gibi [] paramtre alarak, kendi içerisinde render edebilecek.

Eğer ki bu şekilde bir işlem de bunu yapmazsak gereksiz yere render edecek. Performans kaybı. 

Mesela burada calculateObject işlemi kendi içerisinde yoğun bir işlem yapmaktadır. Ve bunu data değişkeni ile çağırıp, Header componentine gönderiyoruz. UseMemo kullanarak çağırmazsak, aşağıdaki button ın her click işlemi bu calculateObject fonksiyounun tekrar tekrar çağıracaktır. Chain Dead ....

### useCallback (shouldComponentUpdate)

const memoizedCallback = useCallback(

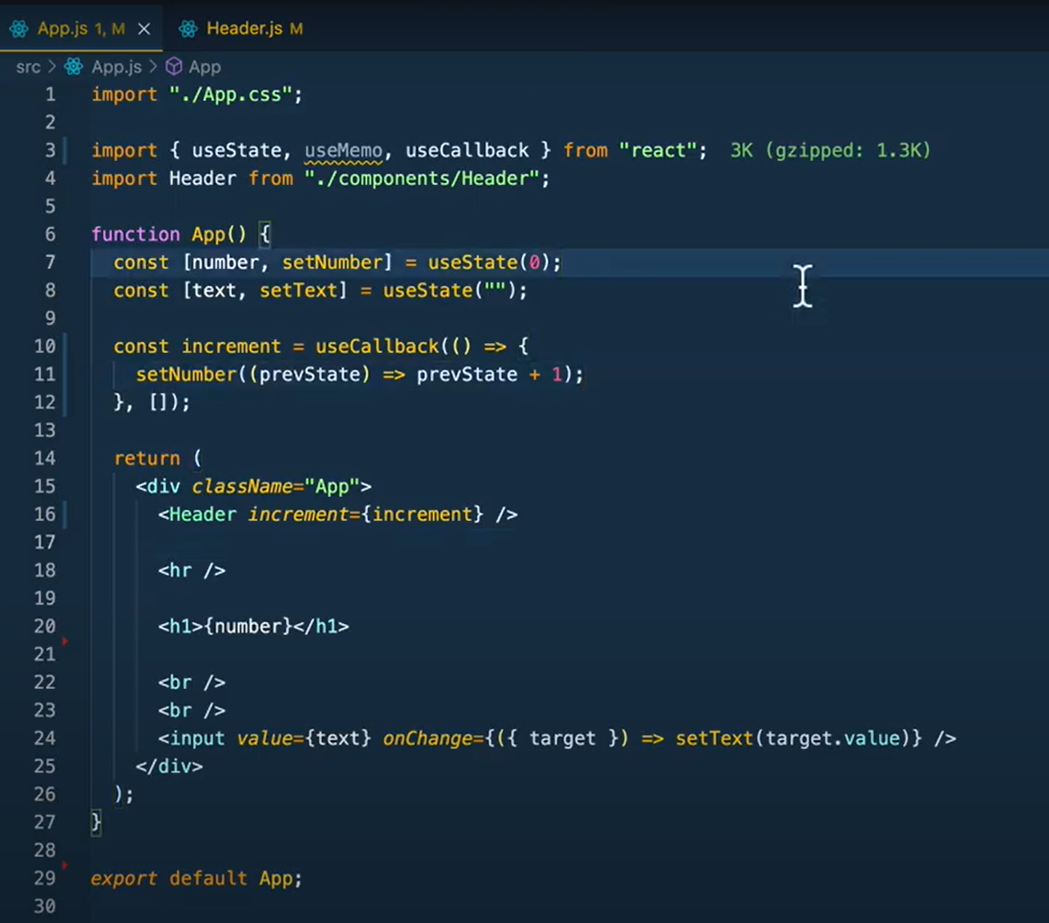
    () => {

      doSomething(a, b);

    },

    [a, b],

  );



Bir fonksiyonuz varsa ve bunu da bir componente props olarak geçiyorsak, bu fonksiyonun da tekrar tekrar render edilmesini istemiyorsak, useCallback i kullanırız. Prevstate yaparak sadece state değiştiğinde render edilmesini sağlarız. Number ve number+1 de yazabilirdik. Ama bu da gereksiz yere render edilmesini sağlarıdı. Çünkü state her değiştiğinde farklı bir prop geliyormuş gibi davranırdı. Böylelikle gereksiz yere tekrar render edilirdi. Component in render edilmesinden bahsediyorum. Ben bunu istemiyorum. Yapmak istediğim şey sadece number değiştiğinde number state ine bağlı olan fonksiyonu değiştirmek. Sadece o sayının artmasını sağlamak. Bağlı olduğu component in de gereksiz yere baştan aşağı render edilmesini istemiyorum. Bu yüzden prevstate ve prevstate+1 ifadelerini kullanıyorum.

**REACT REDUX**

**React-Redux usage with Hooks**

Class component yapısını kullanmadan, hooks yapısını kullanarak redux kullanabiliriz. Redux mantığında olan temel şeyler; connect ile class’ımızı state ve propslar ile kullanmamız için bir func.’tır.

class Try extends Component {

  componentDidMount() {

    this.props.movieActions();

  }

  render() {

    const trump = this.props.movies;

    return (

      <div>

        {

          <ul>

            {trump.user.map((*tr*, *i*) => (

              <li *key*={*i*}>{*tr*.name}</li>

            ))}

          </ul>

        }

      </div>

    );

  }

}

const mapStateToProps = (*state*) => {

  return {

    movies: *state*.movies,

  };

};

const mapDispatchToProps = {

  movieActions,

};

export default connect(mapStateToProps, mapDispatchToProps)(Try);

mapStateToProps ve mapDispatchToProps adında iki func. vardır. mapStateToProps; bizim statelerimizi Props’a aktararak, props aracılığıyla kullanmamıza imkan sağlıyor. Aynı şekilde mapDispatchToProps; ise actionlarımızı props aracılığıyla kullanmamıza imkan sağlıyor. Yukarıda ki satırlarda componentDidMount ile action’ımızı kullanıyoruz. Render’ın içerisinde ki this.props.movies ile de state’imizi kullanıyoruz.

export const App = () => {

  const trump = useSelector((*state*) => *state*.user);

  const dispatch = useDispatch();

  console.log(trump);

  useEffect(() => {

    dispatch(userAction());

  }, [dispatch]);

  return (

    <div>

      {

        <ul>

          {trump.user.map((*tr*, *i*) => (

            <li *key*={*i*}>{*tr*.name}</li>

          ))}

        </ul>

      }

    </div>

  );

};

Burada ise redux mantığını Hooks kullanarak yapıyoruz. Yukarıda ki örnekte bulunan mapStateToProps ve mapDispatchToProps ‘un yerlerine useDispatch ve useSelector adında hooks func. kullanıyoruz. mapStateToProps => useSelector, yukarıda da görüldüğü gibi içine state parametresi alıp, state’den user datasını alıyoruz. mapDispatchToProps => useDispatch, useDispatch func.’ını kullanarak dispatch değişkenine assign ediyoruz. Daha sonra bu değişken ile action’ımızı çağırıp kullanıyoruz. Böylelikle connect ve map.. func.’larına gerek kalmadan sadece hooks kullanarak state ve props işlerimizi yaptık.

rootReducer’ın içerisinde useReducer’dan gelen reducer’ı yani state’imizi user key’i olarak atıyoruz. Böylece state’i kullanırken, state parametresi alıp => state.user diyoruz.

Reducer’ın yapısı aynı

const initialState = {

  user: [],

};

export const userReducer = (*state* = initialState, *action*) => {

  switch (*action*.type) {

    case "LOAD":

      return {

        ...*state*,

        user: *action*.payload,

      };

    default:

      return *state*;

Action’ın yapısı

import axios from "axios";

export const userAction = () => {

  return async (*dispatch*) => {

    const response = await axios.get(

      "https://jsonplaceholder.typicode.com/users"

    );

    const data = await response.data;

    try {

      return dispatch({

        type: "LOAD",

        payload: data,

      });

    } catch (error) {

      dispatch({

        type: "LOAD-DATA-ERROR",

        payload: error,

      });

    }

  };

};

rootReducer’ın yapısı

import { combineReducers } from "redux";

import { userReducer } from "./reducers";

export const rootReducer = combineReducers({

  user: userReducer,

});

Provider ve state’in yapısı

import { createStore } from "redux";

import { rootReducer } from "./redux/rootReducer";

import thunk from "redux-thunk";

import { applyMiddleware } from "redux";

const store = createStore(rootReducer, {}, applyMiddleware(thunk));

ReactDOM.render(

  <React.StrictMode>

    <Provider *store*={store}>

      <App />

    </Provider>

  </React.StrictMode>,

  document.getElementById("root")

);

Action ’da fetch kullanarak datamızı çekiyoruz. Ve bu datayı state’imize koyması için reducer’a gönderiyoruz. ~~Reducer’da initialState olarak user’ı tutuyoruz. Daha sonra bu state’i redux yapısında da kullanabilmek için combineReducers’ın içerisine koymamız lazım. rootReducers’ın içerisinde bu işlemi yapıyoruz.~~ Burada önemli olan, reducers’tan hangi reducers alıp nasıl kullanacağımızı belirlemek. userReducer’ı alıp user olarak kullanmak istediğimizi belirliyoruz. user:userReducer. useSelector’da, state.user dememizin sebebi budur. Reducer’ın içerisinde initialState olarak user:[] olarak user dememizin sebebi, gelen datayı user değişkenine assign edip öyle kullanmak.

Ek olarak action’ı çağırıp kullanırken;

  const trump = useSelector((*state*) => *state*.user);

  const dispatch = useDispatch();

  const dist = bindActionCreators(userAction, dispatch);

  console.log(trump);

  useEffect(() => {

*// dispatch(userAction());*

    dist();

  }, []);

Şeklinde yapabiliriz. Burada ki amaç, action ve dispathc func.’larını bindActionCreators ile bind’ladık. Böylelikle action’ı her çağırdığımızda dispatch ve actionName’i yazmak zorunda kalmayız.

.jsx in içerisinde mesela input alanında onChange= ( console.log(“sdsd”)} dersek, sayfa render edildiği anda yazacaktır-çalışacaktır. Ki biz bunu istemiyoruz. Çünkü func. olarak onChange valuesi vermedik. Ne yapılmalı? ()=> {console.log(“fdswf”)} şeklinde bir func. vermemiz lazım. Çünkü input alanında ki onChange style attr.’si her değiştiğinde react tarafından anlaşılıp ona göre sonuç vermektedir. Bu durumda biz bu onChange alanına bir func. verdiğimiz için, sadece ama sadece bu input-onChange alanını tetiklendiği zaman çalışacaktır. Böylelikle bu sorunu çözmüş oluruz.

* Reducer’da herhangi bir dispatch e göre state update etmemiz ve react tarafından anlaşılması şu şekilde olmaktadır. Örneğin state e gelen datayı update edeceğiz. Yeni gelen datayı state’e push layabiliriz. Fakat bunu yaparsak, kiş objenin elemanlarına erişip birisini değiştirmek gibi olmaktadır, bu hareket react farkedemez. Yani gerekli renderları yapamayacaktır. Bu yüzden sayfa yenilenene kadar state’de görünür bir değişiklik olmayacaktır. Bu yüzdendir ki reducer’ların içerisinde, return olarak tekrar bir state objesi döneriz. Böyle olunca her yeni objede, react farklı bir ram adresi göstermektedir. Böylelikle react, ram’de bir değişklik olduğunu farkedip, değişikliğin olduğu ve buna bağlı olan adresleri de değiştirir. Bu react in virtual-dom’unun çalışma mantığıdır aslında. Biz de bu mantığı baz alarak reducer’lardan statelerimizi update emiş oluyoruz.

function mapStateToProps(*state*) {

  return {

    currentCategory: *state*.changeCategoryReducer,

    products: *state*.productListReducer,

  };

}

function mapDispatchToProps(*dispatch*) {

  return {

    actions: {

      getProducts: bindActionCreators(productActions.getProducts, *dispatch*),

      addToCart: bindActionCreators(cartActions.addToCart, *dispatch*),

    },

  };

}