مشکل دام این بود که هزینه بر برای ما هست یعنی وقتی المانی رو توش تغییر میدیم تمام فرزندان اون از اول رندر میشود و فشار الکی وارد شود .

ویرچوال دام : همان دام یا نسخه ای از دام با سرعت بیشتر کاری که این نسخه مجازی برای ما انجام میدهد اینه که یه کپی از دام ما دارد حالا ما تغییراتی اعمال میکنیم ری اکت میاد نسخه مجازی از تغییراتی که داشتیم را با دام اولیه مقایسه میکند و بهترین راه و کم هزینه ترین راه را برای تغییر در دام اصلی ما انتخاب و اجرا میکند در واقع ویرچوال دام سرعت ما را افزایش میدهد و هزینه ما را کم میکند . خوده ری اکت انجام میدهد و از استیت ها استفاده میکند . از دستوراتی که قبلا استفاده میکردیم و ایتم ها را سلکت میکردیم دیگر وجود ندارد و از استیت ها استفاده میکند .

ان پی ام : پکیج هایی رو میخوایم نصب بکنیم نود جی اس این را دارد

ببل که نصب میشود میاد کد های جدیدمون رو به کد های قدیمی تبدیل میکند برای مرور گرهایی که اپدیت نیستند .

وب پک هم میاد پروژه هامونو اپتیمایز میکنه و بهتر میکنه خط های خالی را حذف میکند و ...

جی اس ایکس : میشود جاوااسکریپت ایکس ام ال

مرورگر ها نمیتوانند کد ها جی اس ایکس را اجرا کنند به همین منظور همان ببل که بالا نام بردیم می آید و این کد ها را به کدی تبدیل میکند که برای مرورگر ها قابل درک باشد .

در داخل جاوا اسکریپ میتوانیم اچ تی ام ال بنویسیم. دیگه لازم نیست هعی بنویسیم داکیومنت کریت المنت و ....

فانکشنال کامپوننت ها تا قبل هوک ها فقط چیزای ساده میشد نوشت که نیاز به استیت نداشتند و فقط به صورت پراپس میگرفتیم.

ری اکت کامپوننت یک متدی به اسم رندر دارد که و داخل رندر اون تگ ها و یو ای و اون چیزی که میخوایم نشون بدیم رو میزاریم و ریترن حتما باید بکنیم .

وقتی اکسپورت کردیم کلاس اپ رو جایی که ایمپورت میکنیم و میخوایم استفاده کنیم باید مانند یک تگ باهاش برخورد کنیم یعنی تگ باز اپ پرانتز و تگ بسته .

کامپوننت ها ریوزبل هستند و هر کامپوننت یک قسمتی از سایت را نشان میدهد. ( متد رندر کلاس اپ اون تگ رو نشان میده . )

برای اون قسمت در کلاس اپ وقتی جی اس ایکس ما بیشتر از یک خط بود با پرانتز کد های چند خطی رو بنویس

قسمت 18 ببین .

وقتی در جی اس ایکس میخوای به یک تگی کلاس بدی از کلاس نیم استفاده کن تا جاوااسکریپت گیج نشه .

در جی اس ایکس نباید هیچ خواهر برادری از جنس یک تگ باشه و بیرونی ترین تگمون فقط یک دیو میتواند باشد . برای حل این مشکل میتوانیم اون تگ مثلا پی با تک مثلا دیو رو داخل یک دیو دیگر بگذاریم . البته میتوانیم جا اینکه تگ الکی ایجاد کنیم انرا داخل یک تگ با نام ری اکت دات فرگمنت بگذاریم .

آخرین راه حل هم این است که فقط تگ های باز و بسته را به صورت خالی بگذاریم .

وقتی داریم چیزی رو به صورت دیفالت اکسپورت میکنیم هم ما و هم جاوا اسکریپت میداند این کدام است و اسمش زیاد مهم نیست چی باشد .

تا قبل هوک ها فانکشن ها کاربرد چندانی نداشتند و فقط برای ساختن یه یو ای ساده یا تگ به کار میرفتن و لایف سایکل نداشتن و استیت هم نداشتن یا به اصطلاح استیت لس بودن و بیشتر استفاده با کلاس کامپوننت ها بود .

در جی اس ایکس خواستی کد جاوا اسکریپت اجرا کنی داخل اکولاد بازو بسته بزار هر چی بود مثلا داخل یه تگ پی بعد اون داخلش.

مفهوم پراپس :

پراپس قبل هوک ها هم بوده هم برا کلاس کامپوننت هم برا فانکشن کامپوننت . پراپس مخفف پراپرتی هست و یک سری دیتا اپشنال هست که میتونیم به یک کامپوننت بدیم و این کامپوننت میتونه اون هارو قبول بکنه با این کار ما از هارد کدینگ جلوگیری میکنیم . و کامپوننت ما را داینامیک میکند . هارد کدینگ یعنی اینکه ما برای اینکه یک کامپوننت ر و یک کاری کنیم که هر دفعه یک چیز رو نشان میده باید هر سری یک کامپوننت سریع درست کنیم و پراپس کاری میکند که با یک بار نوشتن یک کامپوننت آن را داینامیک کنیم . ما برای اینکه در پراپس برای دسترسی به اون مقادیری که بین دو تا تگ نوشتیم دسترسی داشته باشیم باید از چیلدرن استفاده بکنیم ( مثال ماشین که زدیم شماره 4 رو ببین )

فقط یه نکته وقتی یه تگی رو میدی بقیه هم میگیرن فقط براشون خالیه تو جایی که داری استفاده میکنی .

<Car *model*="bugatti" *color* = "blue">  
 i am khafan  
</Car>

استیت : استیت هم مانند پراپس میاد یک سری اطلاعات رو برای ما نگهدار میکند . فرقشون اینه که ما در پراپس یک سری مقادیر رو بهش پاس میدیم و غیر قابل تغییر هستند اما استیت ها داخل خوده کامپوننت ها هستند و قابل تغییر هستند .

برای تغییر استیت ها باید از ست استیت استفاده کنیم و به صورت مستقیم نمیتوانیم عوض کنیم چون به ما گیر میدهد. و وقتی ست میکنی کاری به بقیه مقادیر ندارد فقط مقادیر مشخص شده را تغییر میدهد.

<p> my first car is {*this*.state.name} and its highest speed is {*this*.state.speed} </p>  
<button *onClick*={*this*.changeHandler}>Change</button>

*constructor*() {  
 *super*();  
 *this*.state = {  
 name: "benz",  
 speed: 250  
 }  
}

changeHandler = () => {  
 *this*.setState({  
 name: "bmw"  
 })  
}

توجه شود که به ست استیت یک ابجکت میدهیم .

ست استیت یک اسینک کرونوس هستش پس اگه میخوایم کاری بعدش انجام شه باید از کال بک استفاده کنیم و به عنوان پارامتر دوم بعد بستن ابجکت داخل ست استیت یک ارو فانکشن میدیم که داخلش اون مقداری که میخوایم رو لاگ میگیریم یا عملیات میکنیم

نکته وقتی ما چند بار پشت هم ست استیت رو انجام میدیم ری اکت برای پرفورمنس بهتر فقط یک بار اپدیت میکند .

نکته هر موقع خواستید استیت را اپدیت بکنید جوری که در خوده اون اپدیت قراره از خوده اون استیت استفاده کنید بهتره تو این موارد از روش زیر استفاده کنیم ،

*this*.setState((prevState)=>({  
 number : prevState.number + 1,  
}))

دی استراکچرینگ :

برای اینکه کد هامون تمیز باشه از مبحث دی استراکچرینگ استفاده میکنیم و چون خوده استیت یک ابجکت است به این شکل استفاده میکنیم :

*const* {name,age} = *this*.state;

برای پراپس ها هم میشود همین کار را کرد .

تو فانکشنال :

*const* UserF = ({name,age}) => {  
 *return* (  
 <div>  
 <p>Hi i am {name} and i am {age} years old </p>  
 </div>  
 );  
};

در ری اکت همیشه سعی کنید استیت ها را تغییر بدهید و یو ای سایت خودش تغییر میکند .

*const* Event = () => {  
 *const* maked = () => {  
 *alert*("salam")  
 }  
 *return* (  
 <div>  
 <button *onClick*={maked}> 2click me</button>  
  
 </div>  
 );  
};  
*// for this setstate in func adi bayad dastor this.name = this.name.bind(this)*

*const* UserF = ({name,age}) => {  
 *return* (  
 <div>  
 <p>Hi i am {name} and i am {age} years old </p>  
 </div>  
 );  
};

از خوبی های اپراتور های یک خطی هستند برای چک کردن درستی و یا اشتباه بودن شرط بهتر هستند چون میتوان در جی اس ایکس هم در آن استفاده کرد ولی برای ایف معمولی نمیشود .

<div>  
 {*this*.state.isLoggedIn ? <h1>welcome</h1> : <button>Login </button>}  
</div>

{*this*.state.isLoggedIn && <h1>welcome !</h1>}

برای کلید دادن به عناصر آرایه ها :

{*this*.state.Cars.map(car => <p *key*={car}>{car}</p>)}

فقط حواست باشه نباید تکراری باشه اسامی در لیست و گرنه دو تا یکی میشود و به مشکل ایجاد میشود .

محتویاتی که از سمت بک می آید یک آرایه هستش .

با دستور ان پی ام انستال یو یو ای دی یک پکیج نصب میکنیم که اعداد یونیک تولید میکند .

*v4*()

*import* {*v4*} *from* "uuid"

برای استفاده از یو یو ایدی

به جای یو یو ایدی میتوانیم برای دیتا هایی که می آید از ایدی استفاده کنیم / اگر اسم تکراری نداشته با خوده نام آن ها هم میشود .

مفهوم لایک سایکل :

یک سری بازه های زمانی هستند که ما در آن بازه های زمانی میتوانیم کار انجام دهیم ، بعد از اینکه یک چیزی تغییر کرد یک اتفاقی بیفتد ...

سه لایف سایکل اصلی :

1. Mounting,
2. updating,
3. unmounting

اولی وقتی یک کامپوننت ساخته میشود و به دام اضافه میشود . به این بازه زمانی مشخص میگن همون موقع که سایت بالا میاد .

دومی وقتی میگن که ممکنه یک سری چیز ها تغییر بکنند و دوباره رندر بشود میگویند همین .

سومی وقتی میگن که یک کامپوننت از دام حذف میشود

نکته خارج از مبحث فعلی : اگه خواستی در یک کانستراکتور از یک پراپس استفاده کنیم باید آن را هم به ورودی متد و هم به سوپر بدهیم .

کانستراکتور در فاز اول بالا اولین چیزی است که اجرا میشود . فاز بعدی متد رندر است که میاد یو ای و المان ها و جی اس ایکس را رندر میکند و در صفحه نشان میدهد وقتی این اتفاقات افتاد یعنی کامپوننت ساخته شده است و به دام اضافه شده است و فاز یک بالا اتفاق افتاده است .

کامپوننت دید اپدیت وقتی اتفاق میفتد که رندر اتفاق بیفتد و چایلد هاشم رندر بشوند بعد کامپوننت دید اپدیت اتفاق میفتد . اگه داخل چایلد هم کامپوننت دید اپدیت بود اول اون اجرا میشود و بعدش کامپوننت دید اپدیت متد اصلی اجرا میشود .

فاز سوم : این فاز فقط یک متد دارد وقتی اجرا میشود که قبل اینکه کامپوننت حذف شود اجرا میشود . بعد از آن حذف میشود .

فقط توجه شود که کامپوننتویل انمونت برای اونجایی استفاده میشود که قرار است حذف شود .

برای کامپوننت هایی که بعضی وقت ها ممکن است به ارور بخورند از ارور باندری استفاده میکنیم با استفاده از دستور کامپوننت دید کچ استفاده میکنیم و عملیات های خودمان را انجام میدهیم این دستور همیشگی نیست فقط برای کامپوننت هایی که ممکن است به ارور بخورند .

*import* React, {Component} *from* 'react';  
  
*class* ErrorBoundry *extends* Component {  
 *constructor*(props) {  
 *super*(props);  
 *this*.state = {  
 error : *false*,  
 errorMessage : ""  
 }  
 }  
 componentDidCatch(error, errorInfo) {  
 *this*.setState({  
 error : *true*,  
 errorMessage : error  
 })  
 }  
  
 render() {  
 *if* (*this*.state.error){  
 *return* <p>  
 {*this*.state.errorMessage}  
 </p>  
 }*else* {  
 *return this*.props.children  
 }  
 }  
}  
  
*export default* ErrorBoundry;

<ErrorBoundry>  
 <Cars/>  
</ErrorBoundry>

برای لیفت استیت اپ اوضاع به این گونه است که شما مثلا در یک کامپوننت یک متدی تعریف کردی یک استیت تعریف کردی اما میخوای این متد روی بقیه کامپوننت ها هم اعمال شود خوب فقط کافی است که شما یک درجه این متد را بالا ببری و در کلاس بالاتر تعریف کنی و بعد به صورت پراپس به کامپوننت ها بدهی و اعمال شوند .   
بریم سراغ های اوردر کامپوننت ها که ما میخوایم مثلا دو تا دکمه بزاریم و کامپوننت ها به صورت مجزا از هم تغییر کنند . خوب ببین مثلا ما میخوایم دو تا کامپوننت به صورت مجزا از هم عمل کنند خوب میتونیم تو هر کامپوننت این متد ها رو اعمال کنیم اما مسئله اصلی این است که ما الان در دو کامپوننتمون دو متد داریم که دقیقا دارند کار یکسان را میکنند یعنی اینکه دارند یک تکراری دو بار انجام میشود که این درست نیست خوب ما میایم و از تکنیک های اوردر کامپوننت استفاده میکنیم که این یک کامپوننت میگیرد و یک کامپوننت جدید به خروجی میدهد و ما میتوانیم استفاده کنیم بدون اینکه یک کد تکراری را دوبار بنویسیم .

نحوه استفادش یا میتونیم جایی که اون کامپوننت مورد نظر رو اکسپورت کردیم بزاریم و به عنوان ورودی بدیم اون کامپوننتی که صادر میکنیم رو یا میتونیم به صورت یه متغیر بدیم که در کد پایین نمونش رو میدم :

Cars = *HOCComponent*(Cars);

*import* React, {Component} *from* 'react';  
*const* withCounter = OldComponent => {  
 *class* newComponent *extends* Component {  
 *constructor*(props) {  
 *super*(props);  
 }  
  
 render() {  
 *return* <OldComponent/>  
  
 }  
 }  
 *return* newComponent  
}  
  
*export default* withCounter;

کد ها را خیلی کوتاه و تمیز تر میکند فقط هم حواست باشه لزوما متد های پیاده سازی شده شبیه همدیگر نیستند .

وقتی ما در قسمت اپ به یکی از کامپوننت های خود یک پراپس را میدهیم ان پراپس اول میایید در متد های اوردر کامپوننت برای همین باید این مشکل را در انجا رفع کنیم .

برای حل این مشکل ابتدا باید بیایم به متد های اوردر کامپوننت که در اند کامپوننت ها هستند در اخر جایی که ریترن میکنیم از اسپرید اپراتور استفاده کنیم و پراپس ها را بگیریم و بعد بدهیم به ان کامپوننت هایی که این پراپس را دارا هستند .

اگر هم خواستید مثلا هر بار ایتم یک دونه یک دونه اضافه نشود بیایم در همون کلاس کامپوننت ها اونجا که اولد کامپوننت را میگیریم به عنوان پارامتر دوم یک استپ هم بهش بدهیم و بعدش از ان در همانجا به جای عدد یک استپ را بگذاریم و بعد به کامپوننت هایی که صادر میکنیم به عنوان پارامتر دوم مقدار را بدهیم .

بخش رندر پراپس :

میخوایم به یک کامپوننت میخوایم یک متد بدهیم یک فانکشن بدهیم به آن کامپوننت .

<Sal *sayHi*={check => check ? "sal":"by"}/>

<p>{sayHi(*true*)}</p>

اون چک یک ورودی فانکشن هست سی های هم فانکشن است .

نحوه استفاده از فچ در ری اکت :

داخل استیت اول :

postData : [],

componentDidMount() {  
 *fetch*('https://jsonplaceholder.typicode.com/posts')  
 .then(response => response.json())  
 .then(json => *this*.setState({  
 postData : json  
 }))  
}

<h3>my posts:</h3>  
{*this*.state.postData.map(item => <p *key*={item.id}>{item.title}</p>)}

استایلد کامپوننت ها کامپوننت هایی هستند که دارای استایل هستند و با دستور ان پی ام اینستال استایلد دش کامپوننتز نصب میشوند . یعنی بهشون استایل میدهیم .

نحوه استفاده از استایلد کامپوننت ها بسیار ساده است :

*const* MyDiv = *styled*.div`  
 background-color: darkred;`

و

<MyDiv>  
 <p>Lorem ipsum dolor sit amet.</p>  
 <p>Lorem ipsum dolor sit amet.</p>  
 <p>Lorem ipsum dolor sit amet.</p>  
 <p>Lorem ipsum dolor sit amet.</p>  
</MyDiv>

و همچنین یادمان نرود که :

*import styled from* "styled-components";

از اسم مشخص است یک کامپوننت برای ما میسازد که میتوانیم به آن استایل بدهیم و از آن استفاده کنیم .

&:hover {  
 background-color: black;  
}   
`;

اون بالا میتواند هر تگ دیگه ای باشد جز دیو .

اولا برای خارج شدن از ترمینال وقتی ری اکت باز است از دستور کنترل سی استفاده میکنیم .

نکته دوم برای نصب کتابخانه اکسیوس :

Npm install axios

با دستور بالا این کتابخانه را نصب میکنیم .

*import* axios *from* "axios";

axios.get('https://jsonplaceholder.typicode.com/posts/')  
 .then(r => console.log(r))

هدر رو برای این میفرستیم که سرور متوجه شود تایپ این ورودی که می آید چیست . مثلا جی سون اینه

خوده اکسیوس به جی سون تبدیل میکند و هدر رو هم خودش میگذارد .

برای اینکه وقتی متد میزاری با پرانتز اجرا نشه قبلش یه ارو فانکشن بزار که وقتی کلید شد اجرا بشه .

استایلد کامپوننت :

گفتیم یک کامپوننت هستند پس میتوانیم بهش پراپس بدهیم .

<MyDiv *status* = {*this*.state.status}>

*const* Button = *styled*.button`  
 background-color: #bd2a2a;  
 border: none;  
 border-radius: 10px;  
  
 &:hover {  
 background-color: #921919;  
 }  
`

میتوانیم یک کامپوننت با نام بالا درست کنیم و بعد از آن در قسمت رندر از آن استفاده کنیم و همچنین میتوانیم به آن متد پاس بدهیم . همانطور که در خط پایین میبینیم بعد از اینکه به عنوان پراپس به مای دیو یک شرط دادیم میتوانیم در کد هایی که در بک تیک نوشته شده اند آن ها را تغییر بدهیم :

background-color: ${(props) => (props.status ? "silver":"orange")};

clickHandler = () => {  
 *this*.setState({  
 status : !*this*.state.status ,  
 })  
}

<Button *onClick*={*this*.clickHandler}>Change</Button>

نام باتن به آن کامپوننتی که ساختیم بستگی دارد .

درخواست پوت و پچ برای اپدیت کردن اطلاعات دیتا بیس ، فرقشون اینه که وقتی میخوایم فقط یک ایتم از اطلاعات ذخیره شده را عوض کنیم از پچ استفاده خواهیم کرد و وقتی بخواهیم همه اطلاعات را بفرستیم و جایگزین آن ها شود از پوت استفاده خواهیم کرد .

در پوت اطلاعات قبلی کاملا حذف خواهند شد .

نکته ای در باب اکسیوس ما در استیت یک آرایه خالی درست میکنیم و به وسیله ست استیت وقتی سایت مونت شد اطلاعات را از بک اند میگیریم و در داخل این آرایه میریزیم و بعد توسط مپ برو روی آن آرایه اعمال خواهیم کرد و بعد از آن در صفحه نمایش خواهیم داد . آنجایی که نمایش میدهیم میتواند هر تگ و کامپوننتی باشد .

ادامه پچ و پوت :

componentDidMount () {  
 axios.get ( 'https://jsonplaceholder.typicode.com/posts/' )  
 .then ( r => *this*.setState ( {  
 postData : r.data ,  
 } ) )  
 axios.patch ( 'https://jsonplaceholder.typicode.com/posts/1' , {  
 title : "new title"  
 })  
 .then(r => console.log(r.data));  
}

برای پوت هم مثل بالا :

componentDidMount () {  
 axios.get ( 'https://jsonplaceholder.typicode.com/posts/' )  
 .then ( r => *this*.setState ( {  
 postData : r.data ,  
 } ) )  
 axios.put ( 'https://jsonplaceholder.typicode.com/posts/1' , {  
 title : "new title"  
 })  
 .then(r => console.log(r.data));  
}

فقط توجه شود که ایدی که در کنسول چاپ میشود ایدی است که خود به خود سرور و بک اند قرار میدهد ولی بقیه موارد حذف شده اند .

اکسیوس امکانی را به ما داده است که میتوانیم همزمان چند درخواست ارسال کنیم و بعد از اینکه پاسخ آمد یک جا اطلاعات را دریافت کنیم .

axios.all ( [  
 axios.get ( "https://jsonplaceholder.typicode.com/albums" ) ,  
 axios.get ( "https://jsonplaceholder.typicode.com/photos" ) ,  
 axios.get ( "https://jsonplaceholder.typicode.com/posts" ) ,  
 ] ).then (axios.spread((albums,photos,posts)=>{  
 console.log (posts.data);  
}) )

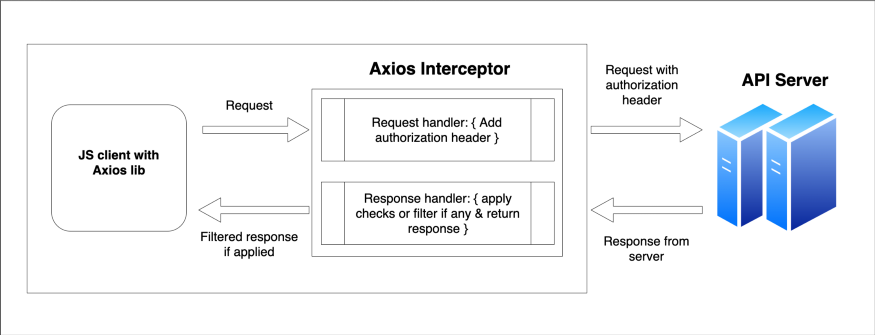
خوب توجه کنیم که پیرو صحبت بالام که میتوانیم چند درخواست بفرستیم باید از اکسیوس دات ال استفاده کنیم که داخلش یک آرایه باید بدهیم به کاما بین شاخص ها توجه شود . با دستور میگیریم و بعد با دستور دن میتوانیم دی استراکچر بکنیم آرایه ها را و با اسپرد پخش میکنیم هر لینک را به یک آرایه و بعد از آن به مقادیر آن ارایه دسترسی داریم .

اینترسپتور ها در اکسیوس :

یک فانکشنی که هر بار که ما یک درخواست را میفرستیم و به ما جواب برمیگردد آن فانکشن اجرا میشود .

در سمت سرور به اینگونه نیست که نسبت به هر درخواست جواب بدهند معمولا یک توکن به افراد میدهند و چک میکنند که اگر آن فرد دارای این توکن بود بتواند استفاده کند و درخواست بفرستد و منتظر جواب باشد و هر کس این توکن را نداشت یا مورد قبول نبود اجازه این کار را ندارند .

استفاده از این فانکشن وقتیه که ما میخوایم یک درخواست بفرستیم قبلش میتوانیم از این درخواست استفاده کنیم و کارهایمان را اعمال کنیم و بعد بفرستیم . برای منتظر جواب بودن را هم میتوانیم اعمال کنیم .



باید در جایی این متد را تعریف کنیم که به همه کامپوننت ها دسترسی داشته باشد یعنی در بالاترین مرحله آن را استفاده میکنیم . داخل ایندکس جی اس مینویسیم . که بالاترین مرحله پروژه هست و میتواند همه موارد را چک کند . ریکواست چیزی است که برای دفعه اول به سمت سرور میفرستیم و ریسپانس را دریافت میکنیم .

axios.interceptors.request.use(request => console.log(request))

همانطور که مشاهده میکنید ما در اینجا درخواستی که میخواهیم به سمت سرور بفرستیم را اول گرفته ایم و نگذاشته ایم سمت سرور برود پس باید اجازه دهیم که به سمت سرور برود . فقط توجه کنید که این متد یوز وقتی استفاده میشود که میخواهیم کارهای خودمان را در آن انجام دهیم . همچنین میتوانیم ری کواست را داخل پرانتز بگذاریم و ارو فانکشن ایجاد کنیم .

*return* request;

حالا میگذاریم این درخواست به سمت سرور برود حتما این کار را باید انجام دهیم .

console.log(`A ${request.method} request sent to ${request.url}`)

کنسول را نگاه کنید وقتی مثلا جوابی میخواهیم بگیریم نشان میدهد که اول درخواست چاپ میشود و بعد از آن پاسخی که سرور به ما میدهد .

axios.interceptors.request.use(request => {  
  
console.log(`A ${request.method} request sent to ${request.url}`)  
 *return* request;  
},error => {  
 console.log (error);  
 *return* Promise.reject(error)  
});

اگر به خطا خوردیم به این صورت باهاش برخورد میکنیم فقط توجه شود که پرامیس ریجکت همان چیزی است که خواندیم که دو حالت داشت یا موفقیت آمیز هست یا ارور میدهد .

حالا پاسخی که سرور میدهد را میخواهیم چک کنیم . نکته خیلی مهم این است که حتما حتما باید درخواست را ریترن بکنیم .

axios.interceptors.request.use(response => {  
  
console.log(`A ${response.method} response sent to ${response.url}`)  
 *return* response;  
},error => {  
 console.log (error);  
 *return* Promise.reject(error)  
});

مانند همان درخواست است . حالا میتوانیم در این پاسخ ها یک فیلتری هم میکنیم .

*return* response.data;

بقیه موارد را فیلتر کردیم و فقط دیتا را به ما میدهد . فیلتر های دیگری هم میتوانیم اعمال کنیم .

مدیریت خطا هم مانند پاسخ ها است .

در کلاس کامپوننت ها وقتی میخواهیم ست استیت بکنیم به وسیله متد های عادی داری مشکل و ارور میشویم که دلیلش فعلا نمیدانم ولی با ارو فانکشن یا راه حل بعدی که در ادامه میگم این مشکل حل خواهد شد .

سینتکس ارور بایندینگ :

This.clickHandler = this.clickHandler.bind(this)

فقط توجه شود که در کانستراکتور اجرا شود تا پرفورمنس سایت پایین نیاید .

بروزر کامپوننت میاید بالاترین حالت رو میگیرد یعنی اون هیستوری ابجکت رو میگیرد و از هیستوری ابجکت استفاده میکنیم دلیل استفادش همینه در واقع میایم در قسمت ایندکس دات جی اس اپ را میگذاریم در داخل بروزر روتر روت با توجه به یو ار ال اونجایی که هستیم ان کامپوننت را نمایش میدهد به طور الکی همه کامپوننت را ها درست نمیکند .

<div>  
 <Route *path*="/" *component*={HomePage}/>  
 <Route *path*="/blogs" *component*={blogs}/>  
 <Route *path*="/aboutus" *component*={AboutUs}/>  
 <Route *path*="/products" *component*={Products}/>  
   
</div>

بسته به اون مسیر کامپوننت مورد نظر را اجرا میکند .

<Route *exact path*="/" *component*={ HomePage }/>

چون ری اکت میاد میبینه هوم اسلش داره و میره اون رو هم برای صفحات بعدی میاره ولی وقتی از کلید اگزکت استفاده میکنیم دقیقا میگیم فقط این باشه .

<Switch>  
  
 <Route *path*="/blogs" *component*={ blogs }/>  
 <Route *path*="/aboutus" *component*={ AboutUs }/>  
 <Route *path*="/products" *component*={ Products }/>  
 <Route *path*="/" *component*={ HomePage }/>  
</Switch>

فقط اون پیجی که تو همه یو ار ال مشترک داره و تکرار میشود رو باید آخر بزاریم که ری اکت بقیه رو هم چک کنه .

مچ می آید یو ار ال سایت را با یو ار ال و بثی که در روت ها نوشتیم مچ میکنه و مشخص میکند .

در بالا به شکل کامپوننت هم میتوان یو ال ها را هم گذاشت .

نحوه پراپس دهی در روت ها :

<Route *path*="/blogs" *render*={(props)=><blogs *name*="ali" {...props} />}/>

نحوه مقدار دهی اینپوت ها و گرفتن ورودی :

inputHandler = (event) => {  
 *this*.setState({  
  
 text : event.target.value,  
 })  
 console.log (event.target.value)  
}

<input *type*="text" *value*={text} *onChange*={*this*.inputHandler}/>

قبل از اینکه قسمت اپشن ها رو توضیح بدهم اگر در استیت مقدار یک اپشن را بدهیم و به عنوان ولیو به تگ سلکت بدهیم وقتی سایت بارگذاری میشود آن اپشنتی که نوشته ایم به صورت دیفالت انتخاب شده است .

نکته برای اینکه یو ای رو تغییر بدهیم همیشه سعی کنیم استیت رو تغییر بدیم چو ن ری اکت به صورت اتوماتیک خودش یو ای رو اپدیت میکند .

<select *name*="check" *id*="salam" *value*={opt} *onChange*={*this*.selectHandler}>  
 <option *value*="opt1">opt1</option>  
 <option *value*="opt2">opt2</option>  
 <option *value*="opt3">opt3</option>  
</select>

opt : "opt2",

وقتی میخوایم یه سری اطلاعات رو از یه فرم بگیریم و به ای ای پی ای بفرستیم :

در فرم وقتی سابمیت میشود که یک باتن با تایپ سابمیت روش کلیک شود .

event.preventDefault();

از کلیک شدن به صورت دیفالت جلوگیری میکند .

<form *onSubmit*={*this*.submitHandler}>  
  
 <input *type*="text" *value*={text} *onChange*={*this*.inputHandler}/>  
 <select *name*="check" *id*="salam" *value*={opt} *onChange*={*this*.selectHandler}>  
 <option *value*="opt1">opt1</option>  
 <option *value*="opt2">opt2</option>  
 <option *value*="opt3">opt3</option>  
 </select>  
 <button *onClick*={*this*.clickHandler} *type*={"submit"}>Check</button>  
</form>

submitHandler = (event) => {  
 event.preventDefault();  
 console.log (*this*.state.text);  
 console.log (*this*.state.opt);  
}

ترتیب روت ها از خاص به عام است و گرنه روت ها مچ نمیشوند . یو ار ال ها که در تگ ها میگذاریم یا در روت ممکن است از سمت سرور بیاید و به ما وسیله مپ باید تولیدشون کنیم و قرار بدهیم .

وقتی میخواهیم به صفحات که میرویم شماره صفحات را ببینیم .

<h1>this is product #{*this*.props.match.params.id}</h1>

فانکشنال کامپوننت هم همین شکل است .

وقتی در روت ها یک مقدار تعریف کنیم در داخل یک کامپوننت دیگر فقط همان میماند و اگر نسخه لینک اولیه را بزنیم به صفحه اصلی میرود پس برای اینکه یو ار اله ایی که میدهیم ایتم های اختیاری داشته باشند در جایی که لینک میدهیم به یو ال ها باید علامت سوال را در قسمت روت در جایی که روت ها هستند بعد از ممیز ایتم علامت سوال بگذاریم و اون ایتم اختیاری میشود.

کوئری استرینگ با علامت سوال شروع میشود بعد از آخرین یو ار ال و بعدش هم اسم متد می آید .

<http://localhost:3000/blogs/milad?sortBy=date&newest=true>

<p>{*this*.props.location.search}</p>

نحوه استفاده از کوئری استرینگ ها فقط مختص به تگ پی نیست و در همه چی میشود از آن استفاده کرد.

فقط اولش علامت سوال بعدش بین ایتم ها علامت اند انگلیسی .

به جای اینکه خودمان کوئری استرینگ ها را جدا کنیم از یک پکیج استفاده میکنیم که خودش کوئری استرینگ ها را جدا میکند .

با دستور :

npm install query-string

parseQuery = () => {  
 console.log (*this*.props.location.search)  
 *const* result = queryString.*parse*(*this*.props.location.search)  
 console.log (result)  
}

*import* queryString *from* "query-string";

با اینکار در یو ار ال ها را جدا جدا میکند و داخل یک ابجکت میریزد فقط دقت کن این ابجکت به صورت استرینگ است و عدد و بولین را باید تبدیل کنی .

در سوییچ میاد به ترتیب همه روت ها را نگاه میکند به مسیرهاشون اگر مچ شد میرود اگر مچ نشد میرود بعدی برای همین است که باید از خاص به عام بروی تا داخلی ها هم نمایش داده شوند .

دنبال مشترک میروند مثلا همون اسلش قسمت مشترک هست .

<Route *exact path*="/" *component*={ HomePage }/>

مثلا ادرس الکی داد هوم پیج را برایش نیاورد .

<Redirect *from*={"/articles"} *to*={"/blogs"}/>

وقتی مثلا کاربر یه جا دیگرو بگه ما بفرستیمش یه جا دیگه

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

توضیح : در واقع روت اون مسیری است که قرار است برویم ما با پث در قسمت لینک ها نمایش میدهیم و با کامپوننت آن کامپوننت را همچنین اون تگ های یو ال صرفا بخاطر این است که هم یک صفحه لود نشود و از لینک استفاده کردیم هم بیاید ما را به اون لینک بفرستد با دستور تو پس طبیعی است که لینک تو با لینکی که در مسیر روت ها هست برابر است چون میخواهیم به آنجا برویم . ......................................................................................................................

وقتی کاربر آدرس اشتباهی یا الکی میدهد اول باید در آخرین مرحله در قسمت سوییچ ری دایرکت کنیم و دستور تو را بزنیم به اون لینکی که میخواهیم بریم حالا که ری دایرکت کردیم باید دستور روت را بسازیم تا واقعا به اون لینک برویم و از دستور مسیر و کامپوننت استفاده میکنیم و آن کامپوننت را نمایش میدهیم .

<Redirect *to*={"/notfound"}/>

در همان صفحه که بقیه هستند .

<Route *path*={"/notfound"} *component*={NotFound}/>

آدرس اشتباهی و شباهت ها باید روت هاشون بالا تر از عام ترین باشه .

مثلا وقتی کاربر ثبت نامش در سایت تمام میشود باید با استفاده از هیستوری او را پوش بکنیم به صفحه اصلی و از این قبیل کاربرد ها پوش هم بخاطر این است که خوده کاربر که لینکی وارد نمیکند یا به جایی نمیرود که از روت این ها استفاده کنیم پس باید از پوش ها استفاده بکنیم .

reset = () => {  
 *this*.props.history.push("/blogs")  
}

مثلا در یک کامپوننتی تعریف میکنی که به وسیله یک دکمه به کامپوننت بعدی برود .

reset = () => {  
 *this*.props.history.replace("/blogs")  
}

این بخاطر این است که وقتی کاربر ثبت نامش تموم شد دیگر نتواند به قسمت لاگین برگردد . و هیستوری باقی نمیگذارد و این لینک جای آن قبلی را خواهد گرفت .

بحث نستد روتینگ :

وقتی از این استفاده میکنیم که مثلا میخواهیم وقتی داخل یکی از کامپوننت ها رفتیم در داخل همان کامپوننت هم کامپوننت ها و روت ها یا همون لینک های دیگری باشد که ما را به صفحات دیگری هدایت کند .

لازم نیست که فقط در اپ روت ها را تعریف کنیم در بقیه جا ها هم میشود .

*import* { *Link*,Route } *from* "react-router-dom";  
*import* Programmers *from* "./Programmers";  
*import* Developers *from* "./Developers";  
*class* ReachUs *extends* Component {  
 render () {  
 *return* (  
 <div>  
 <p>salam</p>  
 <ul>  
 <li><Link *to*={"/ReachUs/Programmer"}>Programmer</Link></li>  
 <li><Link *to*={"/ReachUs/Developer"}>Developer</Link></li>  
 </ul>  
 <div>  
 <Route *path*={"/ReachUs/Programmer"} *component*={Programmers}/>  
 <Route *path*={"/ReachUs/Developer"} *component*={Developers}/>  
 </div>  
 </div>  
 );  
 }  
}  
  
*export default* ReachUs;

پارامتر ها در روتینگ :

<Route *path*={ "/products/:id" } *component*={ Product }/>

<ul>  
 <li><Link *to*={"/products/1"}>product1</Link></li>  
 <li><Link *to*={"/products/2"}>product2</Link></li>  
 <li><Link *to*={"/products/3"}>product3</Link></li>  
 <li><Link *to*={"/products/4"}>product4</Link></li>  
</ul>

ادامه پارامتر ها :

<Route *path*="/blogs/:author?" *render*={ ( props ) => <Blogs *name*="ali" { ... props } /> }/>

مثلا کاربر فلان نام را میدهد .

<li><Link *to*="/blogs/milad">blogs</Link></li>

تو خوده بلاگز هم باید بگیریم طبیعتا و نمایش بدهیم .

<p>the author is {*this*.props.match.params.author}</p>

به خوده ری اکت تول در وب مراجعه شود برای درک این مسائل

فرق فایل های با پسوند جی اس و جی اس ایکس :

فایل های جی اس ایکس برای زمانی خوب است که فقط بخواهیم کد های یو ای را در آن بنویسیم یعنی یک قسمت از بخش گرافیکی سایت را آن وارد کنیم . فقط فقط جی اس ایکس خالی مثلا تگ اچ تی امل خالی

در نهایت جی اس ایکس به جی اس تبدیل میشود و مرورگر نمیتواند اجرا کند .

تغییرات نسخه جدید روت :

دیگر نیاز نیست ترتیب بندی داشته باشیم در روت نسبت خاص به عام یا از اگزکت استفاده کنیم . بعد به جای کامپوننت المنت و داخل جی اس ایکس کامپوننت بزار و همچنین روتز رو هم ایمپورت کن .

داینامیک روت هم یعنی اون مثلا دو نقطه مساوی ایدی که به صورت داینامیک میگرفتیم مثل نسخه قبلی است .

<Route *path*={ "/articles" } *element*={ <Navigate *to*="/blogs"/> }/>

ری دایرکت در نسخه جدید ری اکت روت حذف شده است و در نتیجه ما باید از نویگیت استفاده کنیم و به شکل بالا در واقع یک روت میسازیم و به عنوان المنت اون مسیر را نمایش میدهیم . و عوض هم میشود . برای لینک های الکی :

<Route *path*={ "/\*" } *element*={ <Navigate *to*={ "/notfound" }/> }/>

استار یعنی همه روت ها غیر از آنها که ما نوشتیم .

تفاوت روت ها تو در تو در نسخه 6 : اول اینکه ما هر جا روت رو تعریف میکنیم باید آن را در داخل روتز ها بپیچیم و این اجباری است . و همچنین برای آن روت هایی که میخواهید روت ها ی تو در تو بسازید حتما اسلش استار رو به آخر آدرس آن اضافه کنید تا متوجه شود و گرنه اجازه نمیدهد . و همچنین داخل روت های تو در تو داخل کامپوننتی که میخواهید روت تو در تو بنویسید اسلش نام اسلش رو حذف میکنید چون خوده ری اکت متوجه میشود و فقط مسیر دارای اسمی که به لینک ها دادیم فقط باید باشد این نکته هم در لینک ها هست هم در روتز ها

به کامپوننت ها میتوانیم متد هم به عنوان پراپس بدهیم و استفاده کنیم بدین شرح که در جایی که میخواهیم پراپس بدهیم تعریف میکنیم و سپس مثل قبل به آن میدهیم و بعد در خود کامپوننت داده شده از آن استفاده میکنیم همانند قبل حالا میتواند مثلا با گذاشتنش داخل متد ان چینج یک باتن به عنوان پراپس از آن استفاده کنیم مثال :

در اپ یک متد تعریف میکنیم به اسم سلام کن و بعد با نام سلام کن مینویسمیش و داخلش با استفاده از دیس بهش پاس میدیم و آن ور مثلا با استفاده از یک دکمه داخل ان چینجش آن پراپس را قرار خواهیم داد . در کنار متد میتوانیم استیت هم بدهیم مثلا استیت تعریف کن و بنداز داخل همون متد در قسمت اپ و بعدش همانند قبل .

نکته خارجی به عنوان المنت به روت ها میتوان جی اس ایکس هم داد که میدانستیم یعنی مثلا تگ های اچ تی ام ال و ...

ویژگی های جدید روت ها تو در تو اینکه دیگر لازم نیست داخل هر کامپوننت ما روت تعریف بکنیم و میتوانیم همه روت ها را در کامپوننت اصلی تعریف کنیم فقط توجه شود که اون کامپوننتی که میخواهیم در آن روت های دیگر بزاریم را تگش را به صورت باز و بسته چند خطی بگذاریم و بدین ترتیب روت ها را داخل اون تگ باز و بسته بگذاریم .

با تگ اوت لت داخل همون کامپوننتی که براش روت تو در تو تعریف کردیم میتوانیم کامپوننت ها را نمایش دهیم .

هوک ها :

یک فیچر و ابزاری که است که از ورژن 16.8 به ری اکت اضافه شده است و جدید است . از قابلیت هاش به ما اجازه میدهد از تمام امکانات ری اکت مانند استیت و لایف سایکل ها استفاده بکنیم بدون این که به کلاس ها نیازی داشته باشیم .

مفاهیم جدیدی دارد ؟ خیر از همان مفاهیم قبلی استفاده میکند با این تفاوت که نسبت به کلاس ها راحت تر است . فقط مقداری نحوه استفاده متفاوت است .

چرا هوک ؟ دیگر نیاز فانکشن های بزرگ که نیاز به استیت داشتند را به کلاس تبدیل بکنیم و دیگر نیازی به استفاده از کلمه دیس نیست . و دیگر نیازی به های اوردر کامپوننت و رندر پراپس نیست .

هوکی که به ما اجازه میدهد از استیت ها استفاده کنیم :

*useState*(0)

این یوز استیت همون استیت هست که با در کلاس کامپوننت ها تعریف میکردیم و در واقع یک استیت به ما میدهد و یک فانکشن که میتوانیم به وسیله آن این استیت را تغییر دهیم .

*const* [ number , setNumber ] = *useState* ( 0 );

ورودی اول همان مقدار و مقدار دوم همان فانکشن است . شبیه دی استراکچرینگ است .

<h1>{ number }</h1>  
<button *onClick*={() => setNumber(number + 1)}>Up one</button>

نحوه استفاده از فانکشن ها :

*const* clickHandler = () => {  
 setNumber(number + 1)  
}

مثل سری قبل به صورت متد میدیم . فقط کانست یادت نره .

setNumber(prevNumber => prevNumber + 1);

به عنوان پراپس نباید به اون متد ست نال بدهیم .

<p>{JSON.stringify(data)}</p>

نحوه نشان دادن استیت .

<input *type*="text" *value*={data.name} *onChange*={(event)=> setData({name:event.target.value})}/>  
<input *type*="text" *value*={data.age} *onChange*={(event)=> setData({age:event.target.value})}/>

چون بهش میگیم مقدار جدید رو بزار این تو پس فقط اون ابجکت رو قرار میده و بقیه رو حذف میکنیم در واقع فقط یکی رو نشون میده این نکته هم در نظر بگیر که وقتی استیت تغییر میکنه یو ای رندر میشود .

برای حل این مشکل از اسپرید اپراتور استفاده میکنیم که اون میاد جاهایی که اپدیت شدن رو اپدیت میکنه و جاهایی که تغییری نکردن را همان نگه میدارد این مشکل در کلاس کامپوننت ها نبوده است .

<input *type*="text" *value*={data.name} *onChange*={(event)=> setData({...data,name:event.target.value})}/>  
<input *type*="text" *value*={data.age} *onChange*={(event)=> setData({...data,age:event.target.value})}/>

حل مشکل بالا :

همچنین این مشکل در آرایه ها هم هستش حل مشکلش :

<button *onClick*={() => setData([...data,"newItem"])}>Add</button>

نحوه تعریف آرایه :

*const* [data,setData] = *useState*(["item"])

وقتی برنامه ما اجرا بشود یوز افکت اجرا خواهد شد بعد از هر بار اجرا شدن این اجرا میشود همان کار کامپوننت دید مونت را انجام میدهد .

*useEffect*(()=>{  
 console.log (number);  
})

شرط برای یوز افکت که میخواهیم فقط وقتی اجرا شود که استیت مورد نظر ما تغییر میکند :

*useEffect*(()=>{  
 console.log (number);  
},[number])

یعنی اینکه فقط وقتی این استیت تغییر کرد اجرا شو .

فقط وقتی مونت شد اجرا کنیم و اگر استیت تغییر کرد دیگر تغییر نکند :

منظور فقط رندر شدن اولیه است .

جا نامبر فقط یک آرایه خالی بزار .

از دام حذف شدن همان ان مونت است و قبل اینکه حذف بشی میتونستیم یک سری کارها بکنیم به شرح زیر :

*return* () => {  
 console.log ("sik shodi");  
}

برای بالا :

فانکشن ریترن فقط وقتی اجرا میشود که یک کامپوننت میخواهد از دام حذف شود .

و بعد از آن دوباره مونت میشود .

برای استفاده از ای پی ای ها :

*useEffect*(()=>{  
 axios.get("https://jsonplaceholder.typicode.com/posts")  
 .then(res => setData((res.data)));  
  
},[])

همیشه سعی کن اینطوری استفاده کنی و در نظر داشته باش اون آرایه خالی رو همیشه بده تا هعی درخواست سمت سرور نفرسته .

*const* [data,setData] = *useState*([]);  
*useEffect*(()=>{  
 axios.get("https://jsonplaceholder.typicode.com/posts")  
 .then((response) => setData(response.data))  
 console.log (data)  
  
  
  
},[])  
  
*return* (  
 <div>  
 {  
 data.length ? data.map(item => <h3 *key*={item.id}>{item.title}</h3>)  
 : <h1>Loading..</h1>  
 }

فقط توجه کن این آرایه اولش خالیه چون یک زمانی طول میکشد تا بره و برگردد .

نکته از خوده یوز ردیوسر استفاده شده است تا یوز استیت نوشته شود .

*const* initialState = 0;  
*const* reducer = (state,action) => {  
   
}

اولین خط بخاطر این است که استیت تعریف میکند و مقدارش را صفر میدهد و در این تابع ردیوسر مقدار اولیه استیت فعلی است و اگر اضافه شود آن را هم تغییر میکند ولی مقدار اولیه ما همیشه صفر است ، اکشن هم اون اقدامی است که میخواهیم انجام بدهیم مثلا اگه اکشن اپ اتفاق افتاد ببرش بالا و ....

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

یک مقدار به ما میدهد که به صورت استیت ها با آن برخورد میکنیم

*const* initialState = 0;  
*const* reducer = ( state , action ) => {  
 *switch* ( action ) {  
 *case* "up":  
 *return* state + 1  
 *case* "Down":  
 *return* state - 1  
 *case* "Reset":  
 *return* initialState  
 *default*:  
 *return* state  
 }  
  
}  
  
*const* ShomareshGar = () => {  
 *const* [number,dispatch] = *useReducer* ( reducer , initialState );  
 *return* (  
 <div>  
 <h1>{number}</h1>  
 <button *onClick*={()=> dispatch("up")} >Up</button>  
 <button *onClick*={()=> dispatch("Reset")}>Reset</button>  
 <button *onClick*={()=> dispatch("Down")}>Down</button>  
 </div>  
 );  
};  
  
*export default* ShomareshGar;

حالا اگر یک ابجکت به عنوان استیت بدهیم :

<h1>{number.num1}</h1>

*const* initialState = {  
 num1 : 0,  
};  
*const* reducer = ( state , action ) => {  
 *switch* ( action ) {  
 *case* "up":  
 *return* { num1 : state.num1 + 1 }  
 *case* "Down":  
 *return* { num1 : state.num1 - 1 }  
 *case* "Reset":  
 *return* { num1 : 0}  
 *default*:  
 *return* state  
 }  
  
}

حالا میتوانیم خوده اکشن را هم به صورت ابجکت بدهیم ،

<button *onClick*={ () => dispatch ( { type : "up" } ) }>Up</button>  
<button *onClick*={ () => dispatch ( { type : "Reset" } ) }>Reset</button>  
<button *onClick*={ () => dispatch ( { type : "Down" } ) }>Down</button>

*switch* ( action.type ) {

در کد های بالا استیت و اکشن را به صورت ابجکت داده ایم .

*return* { num1 : state.num1 + action.amount }  
*case* "Down":  
 *return* { num1 : state.num1 - action.amount }

, amount : 5

و ..

*return* { ...state, num1 : state.num1 + action.amount }  
*case* "Down":  
 *return* { ...state,num1 : state.num1 - action.amount }

حل مشکل اینکه اگه دو تا بود اون یکی نپره چون مقدار اولی میره داخل اون ابجکت میشینه و اون یکی حذف میشود .

برای اینکه یک دیس پچ را دوباره ننویسیم که در مثال نیو کنتر دیدیم :

*const* [ number , dispatch ] = *useReducer* ( reducer , initialState );  
*const* [ number2 , dispatch2 ] = *useReducer* ( reducer , initialState );

یه دونه جی اس ایکس تولید کن .

فرق بین یوز استیت و یوز ردیوسر :

اولین نکته یوز استیت برای استفاده از پریمیتیو تایپ ها بهتر است پریمیتیو تایپ ها مانند استرینگ ها و عدد ها و ....

و یوز ردیوسر برای استفاده از رفرنس تایپ ها مانند ابجکت و آرایه ها مناسب است .

ولی این اجباری نیست فقط بهتر است ،

نکته دوم یوز استیت برای تعداد استیت کم بهتر است و یوز ردیوسر برای تعداد استیت زیاد و در نهایت هم یک استیت ما داریم .

نکته سوم اگه استیت هامون مرتبط باشند بهتر از یوز ردیوسر استفاده کنیم مثل درخواست فرستادن سمت سرور که هر سه تاش به هم مرتبط هستند .

نکته چهارم یوز استیت برای استیت های لوکال بهتر است و یوز ردیوسر برای استیت های گلوبال لوکال داخل یک کامپوننت است ولی گلوبال اینکه تو چند کامپوننت استفاده میشه مفهوم کانتکست هم مهم است .

هوک یوز ممو :

برای وقتی است که ما یک فانکشن یا یک کامپوننت یا یک برنامه داشته باشیم که خیلی طول بکشد تا اجرا شود و سنگین باشد بدین ترتیب از هوک یوز ممو استفاده خواهیم کرد . استیت تغییر بکند برنامه دوباره رندر میشود .

ما به یوز ممو یک فانکشن میدهیم و یک دپندسی مثل یوز افکت اون فانکشنی رو میدهیم که قرار است زمان ببرد تا اجرا شود آن را میگیرد و مقدارشو میگیرد و دپندسی یا مقدار دوم هم که میدهیم که به یوز ممو بگوییم اگر این تغییر کرد شما اجرا بشو اگر هم تغییر نکرد از اون چیزی که قبلا داشتی استفاده کن اینجوری اون فانکشن که زمان میبرد الکی اجرا نمیشود .

نحوه پیاده سازیش به این شرحه که همونجا که فانکشن رو ساختی در واقع کانست اسم فانکشن مساوی با یوز ممو یک پرانتز میگذاری و بعدش همان فانکشن که تعریف کردی را داخل آن پرانتز میگذاری دپندسی هم درست بعد از براکت بسته و قبل از پرانتز خوده متد یوز ممو قرار میدهی . دپندسی یک آرایه میتواند بگیرد و داخلش میتوانیم بگوییم این فانکشن به چه متغییری وابسته است مثلا در مثال خوده درس به ایز شون که در داخل همان قرار میدهیم . دفعه اول مقداری طول میکشید چون برای دفعه اول مقدار فانکشن را کچ میکند . ولی برای دفعه ها بعدی اگر ایز شون تغییر نکرده باشد از اجرای این فانکشن جلوگیری میکند .

تعریف هوک یوز کال بک :

کال بک یک مشکل را برای ما حل میکند که در ادامه میبینیم :

نکته خارج از این مطلب پراپس رو به عنوان ورودی به اون فانک کامپوننت میدهیم .

وقتی ما یک استیت را تغییر میدهیم چون آن کامپوننت داخل اپ تعریف شده می آید و بقیه کامپوننت ها هم دوباره رندر میشوند بخاطر اینکه اون استیت تغییر کرده و اپ و تمام فرزندانش دوباره رندر میشوند . برای حل این مشکل پیور کامپوننت در کلاس کامپوننت ها بود و برای حل این مشکل در فانک کامپوننت ها ممو رو داریم و توضیح دادیم . در واقع در کلاس کامپوننت ها پریمیتو تایپ ها و رفرنس تایپ ها را با هم مقایسه میکند هر کدام جدا جدا با هم نوع خودش و بررسی میکند . و همچنین چون کامپوننت های دیگه تاثیری در اون کامپوننتی که تغییر کرده ندارند و عوض نشده اند میخواهیم کاری کنیم که دیگر نیاز نباشد آن کامپوننت ها دوباره رندر شوند . فقط کافیه اونجا هایی که فانکشن ها را اکسپورت میکنی با ری اکت دات ممو دورش بپیچی یعنی درونش قرار بدی . خوده اپ رندر میشود .

منتها نکته ای که یوز ممو دارد وقتی که ما با دو تا دکمه به این استیت ها میدهیم که به وسیله اون تغییر کنند اگر یکی را بزنیم آن یکی هم دوباره رندر میشود.

نکته وقتی استیت تغییر میکند و رندر میشود فانکشن ما با یک مشخصات و رفرنس جدید ساخته میشود و با رفرنس قبلی فرق میکند . در واقع وقتی با رفرنس جدید ساخته میشود یوز ممو واقعا فکر میکند این فانکشن یا کامپوننت دوباره ساخته شده است و دوباره اجازه میدهد که رندر بکند .

برای حل این مشکل ما میگیم که این فانکشن زمانی تغییر بکند که دپندسی مربوط به آن تغییر کرده باشد برای این کار یوز کال بک استفاده میکنیم . دپندسی تعریفش با بالاتر یکی است . و این باعث میشود این از ری رندر شدن برنامه جلوگیری شود . یوز کال بک هم مانند یوز ممو پیاده سازی میشود البته یوز ممو که برای برنامه های سنگین است . یوز کال بک می آید و فانکشن رو در نظر میگیرد و تغییرات آن را بررسی میکند و موقعی اجازه میدهد که این فانکشن تغییر کند که اون استیت یا دپندسی تغییر کند داخل آرایه شکل به عنوان پارامتر دوم مثل یوز ممو اون استیت که میخواهیم تغییر کند و به آن وابسته است را میدهیم . وقتی اون فانکشن تغییر نکند در واقع مقدار استیت آن هم تغییر نمیکند و از ری رندر شدن جلو گیری میکند .

یوز رف :

برای دستیابی به محتویات داخلی تگ اینپوت و ..

ادامه کانتکست که در بالا گفتم :

با یوز کانتکست مستقیم از اپ میدیم به اون کامپوننتی که میخواهیم یا برعکس

بیرون از کامپوننت اپ تعریف میکنیم و با استفاده از کانست تعریف میکنیم و میگیم کانست و نامش مساوی با ری اکت دات کریت کانتکست

داخل جایی که ریترن میکند :

<namecontext.provider value={name}></namecontext.provider>

به صورت تگ باز و بسته و قرار اون نام را منتقل کنیم پرووایدر قراره بگیم که کجا قراره بفرستیم و داخلش اون کامپوننت مورد نظر را میگذاریم ، چرا ؟ چون اونجا بالاترین درجه هست و به ما به اون درخت کامپوننت ها در آن دسترسی داریم و میتوانیم در کامپوننت های زیر آن از آن استفاده کنیم . اون تگ ولیو بخاطر این است که به پرووایدر بگیم این رو برای ما بفرست به کامپوننت ها پایینی یادت باشه کانست بالا رو تعریف کردی اکسپورت کنی که در جاهای دیگه بتونی استفاده کنی

نحوه پیاده سازی در کامپوننت مورد نظر :

Import {namecontext} from ….

و بعد داخل همان کامپوننت

Const name = useContext(NameContext)

داخل جایی که رپ میکنی تو داخل اپ باید اون کامپوننت و اون پراپس رو بدی

میتونی چند تا از این پرووایدر ها را در داخل همدیگه قرار بدهی .

*useEffect*(()=>{  
 document.title = "Landing Page";  
},[])

آرایه خالی بخاطر اینکه وقتی مونت شد اجرا شود .

مبحث ریداکس:

ریداکس یک استیت کانتینر است. در واقع همانطور که ما در یوز کانتکست می آمدیم و یک کانتکست میساختیم یا استیت و در بقیه کامپوننت ها به آن دسترسی داشتیم ریداکس هم مانند یوز کانتکست است. ریداکس یک استور دارد یعنی هر کامپوننت یک استور دارد که از نوع ابجکت است و میتوانیم از آن استفاده کنیم برای تغییرات اون استیت یا ابجکت . خیلی شبیه به کانتکست است و آن استور را به برنامه وصل میکنیم و میتوانیم از مقادیر داخل آن استفاده کنیم. ریداکس برای تمام اپ های جاوا اسکریپت کاربرد دارد.

ریداکس سه مفهوم اصلی دارد. اکشن ، ردیوسر و استور.

استور : جایی است که استیت های ما داخل آن است. و میتوانیم در هر کامپوننتی خواستیم از آن استفاده بکنیم.

اکشن : اکشن اون عمل و اقدامی است که ما میخواهیم روی استیت های استور انجام بدهیم هستش.

ردیوسر: چیزی است که هم اکشن ما را میگیرد و هم استیت فعلی را و اون اکشن را را روی استیت ها اعمال میکند. به این صورت استیت ها تغییر میکنند.

ما نمیتوانیم به صورت مستقیم بیاییم و در داخل اپ یا کامپوننت آن استیت را تغییر بدهیم بلکه باید آن اکشن را دیس پچ کنیم.

استور به اپ وصل میشود. در واقع ما میایم از اپ اکشن را دیس پچ میکنیم ردیوسر استیت و اکشن را میگیرد و آن را تغییر میدهد و بعد استور به اپ وصل میشود. این چرخه در ریداکس است.

در فولدر استراکچر ریداکس که ساختیم میخواهیم اکشن و استیت و بقیه را بنویسیم.

*const* increase = () => {  
 *return* { type : "INCREASE" }  
}

یک فانکشن است که ان اکشن را برای ما ریترن میکند وبه صورت ابجکت بهش دادیم.

در واقع وقتی این متد را صدا میزنیم به ردیوسر میفهمانیم که این اکشن را برای ما انجام دهد.

در داخل ردیوسر تعریف میکنیم که اگر تایپ این بود چیکار کنه.

اکسپورت کردن یادت نره.

*const* initialState = {  
 counter : 0 ,  
}  
  
  
*const* reducer = ( state = initialState , action ) => {  
 *switch* ( action.type ) {  
 *case* "INCREASE":  
 *return* {  
 counter : state.counter + 1  
 }  
 }  
}

ردیوسر استیت و اکشن را میگیرد اکشن را اعمال میکند و یک استیت جدید ریترن میکند

نکته این کیس یک ابجکت جدید را برای ما ریترن میکند در داخل استیت . اگه یه دونه باشه استیت ما مشکل خاصی پیش نمیاد ولی اگه داخل استیت چند تا مقادیر باشه بقیه مقادیر حذف میشن و فقط اونکه ریترن شده است باقی میماند.

*return* {  
 ... state ,  
 counter : state.counter + 1  
}

برای حل مشکل بالا که گفته شد .

دلیل اینکه استور را در خارج از این فایل کونتر تعریف کردیم بخاطر این است که ما میتوانیم چند ردیوسر داشته باشیم یا اکشن ولی فقط یک استور داریم .

برای تعریف استور از ریداکس استفاده میکنیم.

میخواهیم اون استوری که ساختیم را به برنامه خود وصل کنیم تا بدین وسیله بتوانیم از آن استفاده کنیم برای این کار باید پرووایدر را در بالاترین سطح برنامه که همه کامپوننت ها به آن میرسند یعنی اپ ایمپورت کنیم و :

*import* { Provider } *from* "react-redux";

*import* provider *from* "react-redux/lib/components/Provider

اولی درسته

*import* Store *from* "./redux/store";

بخاطر اینکه میخوایم به کامپوننت ها پایینی اپ این استور را بدهیم.

<Provider *store*={ Store }>  
 <div *className*="App">  
 <Counter/>  
 </div>  
  
</Provider>

به اون کامپوننتی که میخوایم یک عملی توش انجام بدیم اینطوری میپیچیم دور برنامه و ازش استفاده میکنیم.

همانطور که گفتیم ریداکس یک استور درست میکنه و استیت منیجر هست و ما میتوانیم توی اون کار های خودمون را روی استیت ها انجام دهیم با خوده ریداکس استور را درست کردیم و برای اینکه ریداکس را به ری اکت استفاده کنیم از ری اکت ریداکس استفاده باید کنیم که در بالا نحوه پیاده سازیش توضیح داده شد . در واقع پروایدر اون استور را میدهد به کامپوننت های زیر اپ

استاده از استور در کامپوننت های مورد نظر : استفاده از اون اکشن و ...

*const* mapStateToProps = state => {  
 *return* {  
 counter : state.counter  
 }  
}

به وسیله این یک کامپوننت تعریف کردیم که می آید استیت را میگیرد و مقدار کونتر را مقدار صفری که اول برای این استیت تعریف کردیم قرار میدهد. و این ابجکت را ریترن میکند

مقدار کانتر و اون استیت را به عنوان پراپس به آن متد کانتر میدهد و این میره اون مقدار داخل کانتر را میارد برای ما اون کانتر داخل استور ریداکس است.

*const* mapDispatchToProps = dispatch => {  
 *return* {  
 increase : () => dispatch ( *increase* () )  
 }  
}

در اینجا میایم دیس پچ رو میگیریم و به وسیله آن میگیم که چه عملی را باید انجام دهیم و وقتی دیس پچ میکنیم ردیوسر آن را انجام میدهد. عملکردش بالاست. در واقع همونجا هم مثل قبلا میتونستی تایپ رو مساوی اینکریز بزاری ولی پیاده سازی به این شکل است.

وقتی این ها را تعریف کردیم حالا برای اینکه در داخل همان کامپوننت از این استیت و دیس پچ استفاده کنیم باید ابتدا کانکت رو ایمپورت کنیم که آن یک های اوردر کامپوننت است که به وسیله آن میتوانیم به آن استیت و دیس پچ دسترسی داشته باشیم نحوه پیاده سازی :

*import* { connect } *from* "react-redux";

*export default* connect(mapStateToProps,mapDispatchToProps)(Counter);

در واقع این کانکت میاد استور مورد نظر ما را به این کامپوننت متصل میکند.

<h1>Counter - {props.counter}</h1>  
<button *onClick*={props.increase}>Increase</button>

نحوه استفاده:

( state = initialState ,

به صورت دیفالت به استیت دادیم.

نحوه استفاده از هوک ها در ریداکس:

<h1>Counter - { state }</h1>  
<button *onClick*={ () => dispatch ( *increase* () ) }>Increase</button>

و

*const* state = *useSelector* ( state => state.counter );  
*const* dispatch = *useDispatch* ();

با استفاده از میدل ویر ها میتوانیم به ریداکس یک سری ویژگی اضافه کنیم. از این ها در کامپوننت استور ما استفاده میکنیم .

نحوه استفاده از آن در استور :

*import* { createStore,*applyMiddleware* } *from* "redux";  
*import* rootReducer *from* "./rootReducer";  
*import* logger *from* "redux-logger";  
*const* store = createStore ( rootReducer,*applyMiddleware*(logger) );

در اینجا که به عنوان پی لود داریم چیزی برای اکشن میفرستیم که بتوانیم در ردیوسر استفاده کنیم همانند قبلا داخل ورودی این مقدار را میدهیم و یک مقدار دیفالت به آن میدهیم که اگر کاربر چیزی را وارد نکرد مقدار دیفالت آن برابر همین باشد.

value=1

*const* increase = (value=1) => {

تبدیل استرینگ به اینتجر   
( *decrease* ( +value ) ) }>Decrease</button>

نحوه پی لود دهی در ریداکس:

<input *type*="text" *value*={ value } *onChange*={ ( event ) => setValue ( event.target.value ) }/>  
<button *onClick*={ () => dispatch ( *increase* ( +value ) ) }>Increase</button>  
<button *onClick*={ () => dispatch ( *decrease* ( +value ) ) }>Decrease</button>

*const* [ value , setValue ] = *useState* ( 1 )

*const* increase = ( value = 1 ) => {  
 *return* { type : "INCREASE\_COUNTER" , payload : value }  
}  
  
*const* decrease = (value) => {  
 *return* { type : "DECREASE\_COUNTER" , payload : value }  
}

counter : state.counter + action.payload

ریداکس تانک: یک میدل ویری است که به ما اجازه میدهد تا به وسیله آن اکشن هایی بنویسیم که به صورت اسینک کرونوس باشند برای فرستادن به سمت سرور که یک مدتی صبر کنند.

npm install axios redux-thunk

پیور کامپوننت: وقتی استیت یا پراپس تغییر بکند کامپوننت ری رندر میشود ولی وقتی پیور کامپوننت بسازیم یک مقایسه ای میکنیم بین پراپس ها و استیت های فعلی با قبلیا میکنه اگه تغییری کرده باشند یا تغییری نکرده باشند و اگر تغییری نکرده باشند از ری رندر شدن سایت جلوگیری میکند و افیشنسی سایت را بالا میبرد. ممکنه پراپس یا استیتی تغییر کنه ولی مقدارش با مقدار قبلی یکی باشه.

استیت: یک سری اطلاعات و دیتا ها هستش که در داخل هر کامپوننت ما قرار میدیم و ممکنه که تو اون لایف سایکل یا چرخه حیاط که وجود داره اون اطلاعات تغییر بکنند.

پراپس : یک سری اطلاعات و دیتا هستند که توسط کامپوننت های پرنت یا پدرو مادر به کامپوننت های فرزند داده میشوند.

تفاوت استیت و پراپس:

یک. استیت ها داخل هر کامپوننت هستند ولی پراپس ها پاس داده میشوند به آن کامپوننت

دو. ما استیت ها را در داخل خوده کامپوننت میتوانیم تغییر دهیم ولی پراپس ها را نمیتوانیم تغییر دهیم مگر اینکه یک متدی یا استیت به عنوان پراپس اومده باشه که ما اونو تو پرنتش تغییر بدیم ولی عملا نمیتونیم پراپس رو تغییر بدیم. مثل استیت نمیشه برخورد کرد.

ایونت هندرلر ها در کلاس کامپوننت ها چطور بایند میشند؟ یا از ارو فانکشن ها استفاده کنیم یا همون داخل کانستراکتور بایند کنیم.

کاندیشنال رندرینگ: از ترنری اپراتور ها استفاده میکنیم یا امپرسن به این شکله که مثلا میگیم اگه یه استیتی ترو بود فلان چیزو نشون بده و اگه فالس بود یه چیز دیگرو نشون بده.

نقش کی در مپ کردن یک آرایه : به وسیله این کلید ها که یونیک هستند یا خاص هستند برای هر ایتم ری اکت میتونه این ایتم ها رو بشناسه و اگه در این ایتم ها چیزی حذف بشه یا اضافه بشه یا تغییری بکنه با استفاده از این کلید ها میتونه اون هارو تشخیص بده.

لایف سایکل ها :

مونتینگ : برای دفعه اوله که سایت رندر میشه و بالا میاد اتفاق میفته

اپدیتینگ: برای وقتیه که یک استیت یا پراپسی تغییر میکرد و باعث میشد که اون استیت دوباره ری رندر بشه که اون حالت اپدیتینگه

ان مونتینگ : برا وقتیه که اون کامپوننت ما از دام خارج میشد یا اون رو دیگه نشون نمیدادیم یا حذف میکردیم.

های اوردر کامپوننت ها :

کامپوننت هایی هستند که یک سری کامپوننت ها رو میگیرن و یک سری قابلیت به اونها میدن مثل های اوردر فانکشن ها هستند.

کانتکست : یک سری دیتا هست که میخوایم به چند تا کامپوننت ها بدیم ولی خوب جالب نیستش که به عنوان پراپس هعی اون هارو به کامپوننت ها پاس بدیم تا به کامپوننت مورد نظر برسه از کانتکست استفاده میکنیم و داخل هر کامپوننتی خواستیم استفاده میکنیم خیلی شبیه به ریداکس هستش .

وقتی ازت پرسیدن از چه استیت منیجری استفاده میکنی من هم کانتکست رو بلدم استفاده کنم هم ریداکس رو

چرا در ری اکت به جای کلاس از کلاس نیم استفاده میشود ؟ چون کلاس کلمه ای هستش که توسط خوده جاوا اسکریپت رزرو شده توی اچ تی ام ال این مشکل نبود ولی وقتی اینجا استفاده میکنیم موقع کامپایل گیج میشه که این کدوم کلاسه . پس بهتره از کلاس نیم استفاده کنیم.

فرگمنت چیست ؟ ما وقتی که میخوایم یک چیزی رو ریترن بکنیم داخل کامپوننت همه اون در نهایت باید درون یه تگ باشه . و موقعی استفاده میکنیم که نمیخوایم یک تگ اضافی داخل دام مورد نظر ما باشه در واقع از این استفاده میکنیم که در داخلش هر چه قدر خواستیم تگ بنویسیم بدون اینکه در نهایت نیاز باشه یک تگ جدید تولید کنیم . اون تگ خالی هم میشه  
چگونه میتوان روی پراپس ها اعتبار سنجی انجام داد؟

با وسیله پراپس تایپ که برا وقتیه که چند نفر همزمان میخوان کار کنن و مثلا میگیم که این پراپس من فقط میتونه یک رشته رو قبول کنه. وقتی که مثلا یه کامپوننتی رو توسعه بدیم با استفاده از پراپس تایپه میفهمیم که این پراپس چی میپذیره.

اس پی ای :

سینگل پیج اپلیکیشن سایت های تک صفحه ای هستند که بدون اینکه نیاز باشه صفحه لود بشه این اون اطلاعات به کاربر نمایش داده میشه

روش های مدیریت کردن استیت ها هم که بالا گفتیم.

ویرچوال دام:

ری اکت میاد یک نسخه مجازی از همون دامی که در سایت ما هست رو میسازه و وقتی که یک تغییری میدیم اون تغییر روی اون ویرچوال دام اعمال میشه و بعدش خوده ری اکت میاد بهترین راه و بهنیه ترین راه برای پیاده سازی رو روی دام اصلی پیدا میکنه و فقط میره اون چیز هایی که تغییر کردن رو تغییر میده و کل دام دوباره رندر نمیشه چون دوباره رندر شدن دام به اصطلاح هزینه بر هستش یا افیشنسی سایت رو پایین میاره سرعت سایت میره بالاتر

نکست جی اس :

نکست جی اس یک فریم ورک داخل ری اکت هستش که با استفاده از ویژگی هایی که داره خیلی از مشکلات عادی رو حل میکنه و داخل خودش داره و نیازی نداره که تو بخوای داخل خوده ری اکت اپ چیز های دیگه نصب کنی و برای برنامه های ری اکت تو مقیاس های بزرگ به کار میره.

نکته راجب کامپوننت نیوز که اول ساختیم حتی اگه یک فایل جدید بسازیم به اسم نیوز و این کامپوننت رو ببریم داخلش و اسم ایندکس رو روش بزاریم باز هم همون یو ار ال قبلی هستش بخاطر اینکه خوب یه بخشی از سایته که نیوزه و همچنین دیگه نیازی به ری اکت کامپوننت ایمپورت کردنش نیست و به ایندکس اچ تی امل هم که تگ روت داخلش بود نیازی نیست.

در واقع اون فولدر بندی که بالا گفتیم حالت پث عمل میکنه. برای اینکه یک کامپوننت دیگه داخلش بسازی میتونی یک کامپوننت همونجا که نیوز رو ساختی بسازی و استفاده کنی یا میتونی یک فایل به همون نام بزنی و داخلش یه ایندکس جی اس بزاری مثل قبل.

اگه یک کامپوننت رو نامشو داخل براکت بزاری به نکست جی اس میگه که لینک این داینامیک هست و هر چیزی وارد کنی باعث میشه اون صفحه مورد نظر رو ببینی.

در واقع ایندکس جی اس در فایل پیجز در نکست آدرسش اسلش خالی هست و هیچی بعدش نیست و همون پیج اصلی ما تلقی میشود و نام خاص است.

*function MyApp*({ Component, pageProps }) {  
 *return* <Component {...pageProps} />  
}

این پیج روت ماست مثل ری اکت که داشتیم و در نهایت اینو نکست میبینه و خوب اون کامپوننت کامپوننتی هستند که وقتی تو هر صفحه میریم ریترن میشه و اون پراپس ها کامپوننت هایی هستند که ممکنه پراپس داشته باشند و اون شکلی دادیم.

وقتی چیز کلی میخوای بزاری از این استفاده کن.

تو نام یو ار اله ایی که توی پیجز میسازی برای نکست جی اس لازم نیست اون فانکشنش حتما هم نام باشه با خوده اون کامپوننته.

استاتیک وقتی پری رندر میشه سمت سرور که وقتی که پروژتو ساختی پروژه ساختن همون ان پی ام بیلد میشه

Npm run build

یعنی وقتی که سایت رو اولین ساختی همون یک بار پری رندر میشه و دیگه تغییری نمیکنه. خوب مشکلش اینه که بخوای دوباره دیتا رو اپدیت کنی و بگیری دوباره باید پروژه رو بسازی.

به این روش بالا میگن استاتیک جنریشن .

*export function getStaticProps* () {  
   
}

این نام رزرو شدست و نکست دنبال این فانکشن تو کامپوننت میگیرده . قبل از فانکشنی که داخل کامپوننت داری فراخونی میشه. اسمش اینه بخاطر اینکه قراره پراپس رو برای این کامپوننت آماده کنه.

*export async function getStaticProps* () {  
  
}

یکی از دلایل خوبیش اینه که میتونی به عنوان اسینکرونوس ازش استفاده کنی چون نکست مثل ری اکت عمل میکنه. در واقع کد اینجا وقتی که پروژه بیلد شده رندر شده و در قسمت سرور یا در قسمت کلاینت رندری انجام نشده است. داخلش خوب میدونیم دیتا بگیریم یا بفرستیم. همیشه یه ابجکت ریترن میکنیم. و یک پراپس میفرستیم و حتما باید نامش پراپس باشه.

اون پیج هایی که قرار نیست هیچ دیتایی نشان دهند یا از سرور بگیرند اینکه استاتیک جنریشن بمانند هیچ مشکلی ندارد ولی برای بقیه پیج ها که قرار است دیتا نمایش دهند باید از اس اس جی استفاده کنیم.

همانطور که قبلا گفتیم مشکلات استاتیک جنریشن اینه که فقط وقتی پروژه رو میسازیم دیتا رو میگیره و پری رندر میشه و اگه تو بازه بعدش سمت دیتا بیس چیز جدید رو اد کنیم نمایش داده نمیشه و همیشه فقط اون چیز هایی نمایش داده میشه که موقع بیلد کردن پروژه فچ کرده و

*export async function getStaticProps* () {  
 *return* {  
 props : {  
 meetups : MEETUPS  
 } ,  
 revalidate : 10  
 }  
}

با استفاده از دستور ری ولی دیت به این معنی است که هر ده ثانیه یکبار این درخواست میفرست سمت سرور و ری جنریت میشه در واقع یک بار وقتی بیلد میشه جنریت میشه و دفعه بعدی وقتی ده ثانیه گذشته دوباره درخواست میفرسته سمت سرور اگه درخواست جدیدی باشه . و جنریت جدید جا به جا میشه با جنریت قبلی . و با این دستور تو مطمئن میشی دیتات قدیمی تر از ده ثانیه پیش نیست. بخاطر همین دستور دیگه نیاز نیست هعی پروژتو بیلد کنی باری دستور جدید.

خوب گاهی ممکن است که تو بخوای برای هر ری کواستی که سمت سرور میاد ری جنریت کنی این صفحتو و برای هر یک ثانیه یا تو موقع بیلد نباشه. که یک جانشنین هست . :

*export async function getServerSideProps* () {  
   
 *return* {  
 props : {  
 meetups : MEETUPS  
 }  
 }  
}

فانکشنی که به ازای هر درخواستی که میاد جنریت میشه سمت سرور.

*export async function getStaticProps* ( context ) {  
 *const* meetupId = context.params.meetupId  
 *return* {  
 props : {  
 meetupData : {  
 image : { imageIcon } ,  
 id : meetupId ,  
 title : `First meetup` ,  
 address : `some Street 5, some city` ,  
 description : `this is a first meetup`  
 }  
 }  
 }  
}

تو فانکشن هایی که نیاز داری اطلاعات بگیری هم باید همینو بنویسی اون کانتکست هم بعدا باید روش فکر کنم همچنین برای پیج های داینامیک یک متد دیگه هم نیاز هست که جلوتر میگم.

*export async function getStaticPaths* () {  
 *return* {  
 path : [  
 {  
 params : {  
 meetupId : 'm1' ,  
 }  
 } ,  
 {  
 params : {  
 meetupId : 'm2' ,  
 }  
 }  
 ];  
 }  
}

خوب ما اول به یک ارور میخوریم برای پیج های داینامیک برای اینکه وقتی قراره نکست جی اس تمام صفحاتی که داینامیک هستند رو پری جنریت کنه باید بدونه دقیقا کدوما رو بکنه که خوب ما از این متد استفاده میکنیم و به تعداد صفحاتی که داری بهش ایدی رو پاس میدیم البته اینجا هارد کورد کردیم و میتوانیم به صورت دیتا از سرور بگیریم .

fallback :*false* ,

قبل پس وقتی فالس بزاریم در واقع داریم به نکست جی اس میگیم تمام صفحات مورد نیاز رو ما تعریف کردیم و اگه جز اینا نبود چیزی نشون نمیده 404 نشون میده ولی اگه ترو بزاریم نکست جی اس سعی میکنه یه چیز داینامیک بسازه.

*import* { MongoClient } *from* 'mongodb'  
  
  
*function handler* ( req , res ) {  
 *if* ( req.method === 'POST' ) {  
 *const* data = req.body;  
  
 *const* { title , image , address , description } = data;  
 MongoClient.*connect* ( 'mongodb+srv://mamali-main:5owHsJEhDKmBDwlL@cluster0.phffq.mongodb.net/meetups?retryWrites=true&w=majority' )  
 }  
}  
  
*export default handler*;

تو پوشه ای پی این رو میسازیم و این هیچ وقت به کاربر نشان داده نمیشود.

*const* db = client.db;

به اون دیتا بیس وصل میشه یا اگه نباشه میسازه .

Concurrent: به ما کمک میکند که state هایی که طول میکشند تا انجام شوند را مدیریت کنیم به این ترتیب بگوییم آن state مورد نظر که زمان بیشتری میبرد را آخر از همه اجرا شود و دیگری که زمان کمتر میبرد اول اجرا شود.

usetrainsition: با استفاده از این هوک میتوانیم state ها را اولویت بندی کنیم به این صورت که اون هایی که میخواهیم اجرا شوند و زمان زیادی نمیبرند را اجرا میکنیم و کاری نداریم ولی برای آن ها که زمان زیادی میبرند اول هوکی که در بالا نامش را بردیم را ایمپورت کن و سپس دستور زیر را بنویس:

const [isPending, startTransition] = usetrainsition ();

و بعد از آن دور آن set state یی که زمان بیشتری برای اجرا میبرد به شکل زیر wrap میکنیم:

startTransition (() => {setNumbers (….)});

با دستور isPending هم یک دستور true و false یی هست که به وسیله آن میتوانیم بفهمیم که آیا برنامه ما در حالت انتظار است یا خیر که مثلا اگه بود میتونیم یه چیز نشون بدیم و ...

تا الان و با استفاده از هوک بالا میتوانستیم تغییراتی که یک state داشت را زیر نظر بگیریم اما وقتی این state مورد نظر ما به عنوان props به یک component دیگر پاس داده میشود دیگر نمیتوانیم آن را کنترل کنیم ، در واقع بحث کلا بین دسترسی داشتند به set state و نداشتن آن است .

راه حل : با استفاده از هوک useDeferredValue میتوانیم مشکل بالا برای وقتی که به آن state که زمان زیادی میبرد برای اجرا کردن دسترسی نداریم مشکل کند شدن را حل کنیم بدین ترتیب که ابتدا آن را ایمپورت میکنیم و سپس :

Const deferredValue = useDeferredValue(numbers)

و بعد از آن از این مقداری که به عنوان const در بالا تعریف کردیم مانند state رفتار میکنیم و مشکل کند شدن حل میشود .

StrictMode: برای وقتی است که برنامه در حالت توسعه است و سخت گیری بیشتری میکند برای react ورژن 17 مد نظر بود چیزی که گفتیم و وقتی برنامه بر روی سرور واقعی قرار میگیرد دیگر نیازی به این نیست.

در ورژن 18 هم هستش فقط باید در قسمت root.render بزاری .

نکته : وقتی دو state پشت هم بیایند react هر دو را با هم انجام میدهد ولی component ما به ازای تغییرات فقط یکبار render میشود به این مطلب state batching گفته میشود این ویژگی در ورژن 17 هم هست ولی اگر به حالت asynchronous باشد در ورژن 17 کار نمیکرد و به ازای مثلا 2 تغییر 2 بار صفحه ما render میشود در ورژن 18 در حالت معمولی هم کار نمیکند بخاطر این است که برنامه ما در حالت StrictMode است و همین باعث میشود که احتیاط بیشتری در برنامه رخ دهد و اگر این حالت را برداریم این مشکل حل میشود. همچنین در ورژن 18 state batching در حالت async هم کار میکند

Suspense: این کامپوننت برای موقعی است که میخواهیم از یک سرور یک سری دیتا fetch بکنیم و وقتی داریم این دیتا رو از اونجا میگریم و میاریم در این بازه یک عکسی یا متنی یا اسپینر نمایش بدهیم حتما برای این ویژگی باید از هوک use axios را استفاده کنیم و فقط کافی است جایی که آن کامپوننت مورد نظر وجود دارد Suspense را دور آن wrap کنیم و بعد از این عمل در قسمت اول تگی که باز و بسته شده به عنوان پراپس fallback را میدهیم و سپس براکت را باز و بسته میکنیم و داخل آن میتوانیم هر کامپوننت یا جی اس ایکس و ... نمایش دهیم مانند سابق ، چون خوده ساسپنس میفهمد که دیتا داره میاد و بر میگرده .

هوک useParams:استفاده از این هوک در ورژن 5 و 6 route ها به یک صورت است ،با استفاده از این دستور دیگر لازم نیست دستورات پیچیده قبلی را بنویسیم و خیلی راحت میتوانیم برای route ها تو در تو و یا route هایی که به صورت داینامیک هستند میتوانیم تمام props ها و اطلاعات لازم را با استفاده از این دستور بگیریم ، فقط کافی است ابتدا دستور را import کنیم :

Import {useParams} from ‘react-router-dom’;

و بعد در قسمت تابع مورد نظر دستور زیر را بنویسیم تا امکاناتی که این هوک در اختیار ما قرار میدهد را ببینیم:

Const params = useParams();

هوک useNavigate: این هوک به همان مطالب push و replace کردن ها با استفاده از لینک ها مرتبط است که قبلا پروژه های مربوطه را انجام داده ایم ، کلا بعضی وقت ها لازم است که خودمون کاربر رو بفرستیم به یک مسیر دیگه بدون اینکه لازم باشه کاربر بر روی لینکی کلیک کند ، در ورژن 5 با استفاده از هوک useHistory انجام میشد که الان کار نمیکند ، در ورژن 6 ما هوک useNavigate را داریم فقط کافی است دستور زیر را بنویسیم:

Import {useNavigate} from ‘react-router-dom’;

و در داخل تابع مورد نظر :

Const navigate = useNavigate();

navigate(‘/’);

دستور بالا به صورت پیش فرض برای ما push انجام میدهد ولی اگر نیاز به replace بود فقط کافی است تغییر زیر را انجام دهیم:

navigate(‘/’,{replace:true});

فقط توجه شود که replace امکان برگشت ندارد. و نکته جالب تر این است که وقتی از دستور زیر استفاده میکنیم به صفحه قبلی که کاربر در آن بوده است میرویم :

navigate(-1);

میتوانیم -2 هم بدهیم که به دو صفحه قبلی برمیگردد ، اعداد مثبت هم به صفحه جلو که کاربر قبلا در آن بوده است و برگشته است استفاده میکنیم.

Graphql: یک زبان query language هست که برای API ما هست، ما تا الان داشتیم از REST API ها استفاده میکردیم و به آنها شناخت داشتیم اما میخواهیم یک روش دیگر را به کار ببریم که به آن GraphQL API گفته میشود، در واقع GraphQL هم یک interface یا واسط دیگر برای ما هست، یعنی بین پشت سایت و جلوی سایت که ارتباط برای ما ایجاد میکند در REST ما فقط قوانینی را داشتیم که برای آن نوشته بودند ولی در اینجا دیگر آن قوانین مطرح نیست، و GraphQL قوانین خودش را دارد و همان کار ها را برای ما انجام میدهد، یک مشکلی که REST ها دارند این است که وقتی به یکی از end point ها ما درخواست میفرستیم همه اطلاعاتی که در داخل آن API است برای ما ارسال میشود در صورتی که ما شاید با برخی از آن اطلاعات کاری نداشته باشیم ولی این اطلاعات اضافه مدام برای ما ارسال میشود که در اینجا به اصطلاح over fetching رخ داده است، و یا در یک صفحه blog میخواهیم هم جزئیات نمایش داده شود و هم کامنت های کاربران نمایش داده شود ولی ما با درخواست فرستادن به آن API، کامنت های کاربران را دریافت نمیکنیم که به اصطلاح under fetching رخ داده است، این ها مشکلاتی هستند که ما در REST داریم، اما در GraphQL ما داده هایی که مورد نیاز ما هست را توصیف و مشخص میکنیم و درخواست میفرستیم و جوابی را میگیریم که واقعا مورد نیاز ما است.

ما در GraphQL دو عمل اصلی داریم اول query هست که عمل get را برای ما انجام میدهد و عمل دوم mutation است که post و put و patch و delete را برای ما انجام میدهد.

تا 179 دیده شد.